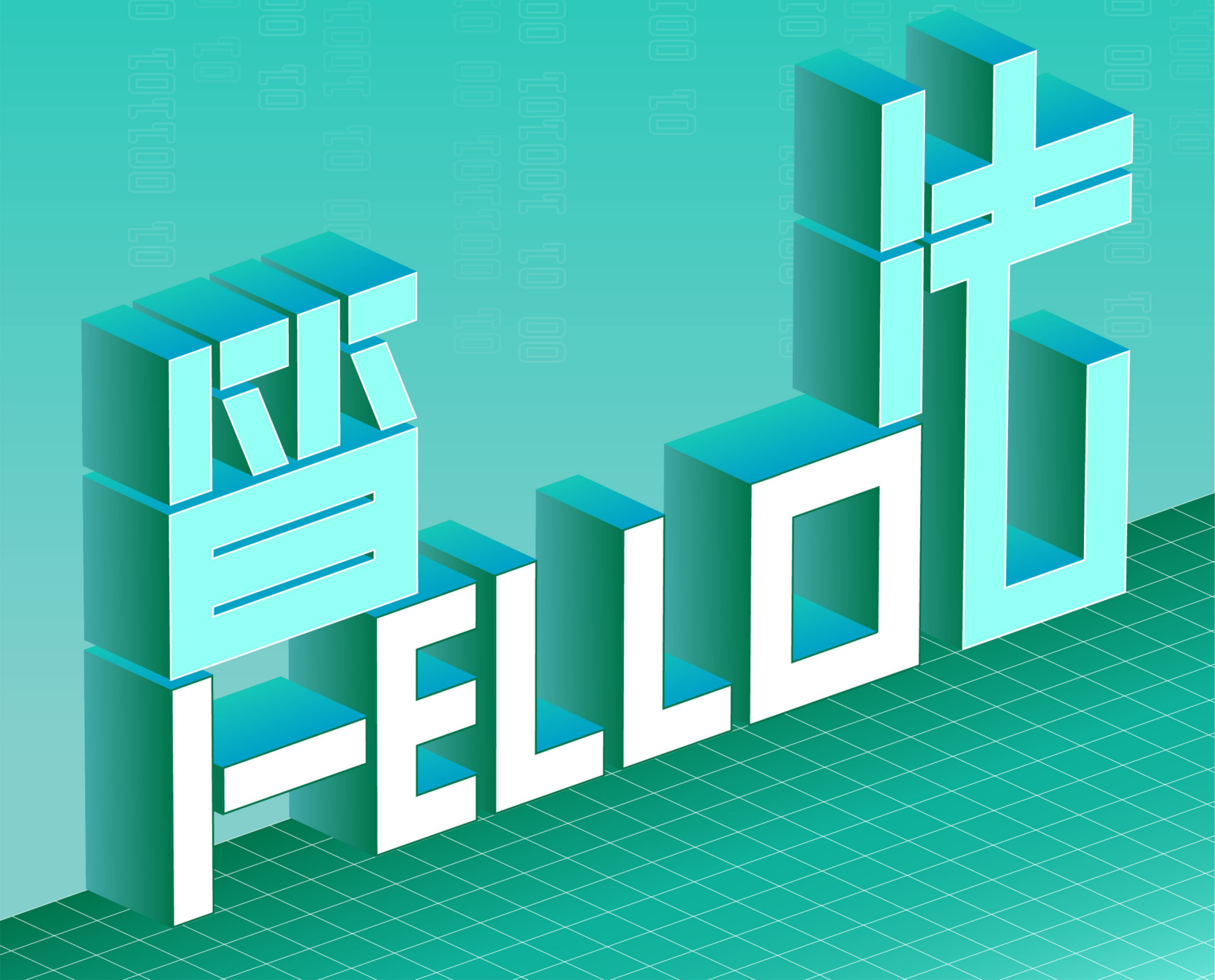
Hello Algo

Data structures and algorithms crash course with animated illustrations and off-the-shelf code

Yudong Jin (@krahets)



Hello Algorithms

JavaScript

Author: Yudong Jin (@krahets)
Code Review: Fa Xie (@justin-tse)

Release 1.3.0
2026-01-01

Before Starting

A few years ago, I shared the "Sword for Offer" problem solutions on LeetCode, receiving encouragement and support from many readers. During interactions with readers, the most frequently asked question I encountered was "how to get started with algorithms." Gradually, I developed a keen interest in this question.

Diving straight into problem-solving seems to be the most popular approach—it's simple, direct, and effective. However, problem-solving is like playing Minesweeper: those with strong self-learning abilities can successfully defuse the mines one by one, while those with insufficient foundations may end up bruised and battered, retreating step by step in frustration. Reading through textbooks is also a common practice, but for job seekers, graduation theses, resume submissions, and preparations for written tests and interviews have already consumed most of their energy, making working through thick books an arduous challenge.

If you're facing similar struggles, then it's fortunate that this book has "found" you. This book is my answer to this question—even if it may not be the optimal solution, it is at least a positive attempt. While this book alone won't directly land you a job offer, it will guide you to explore the "knowledge map" of data structures and algorithms, help you understand the shapes, sizes, and distributions of different "mines," and enable you to master various "mine-clearing methods." With these skills, I believe you can tackle problems and read technical literature more confidently, gradually building a complete knowledge system.

I deeply agree with Professor Feynman's words: "Knowledge isn't free. You have to pay attention." In this sense, this book is not entirely "free." In order to live up to the precious "attention" you invest in this book, I will do my utmost and devote my greatest "attention" to completing this work.

I'm acutely aware of my limited knowledge and shallow expertise. Although the content of this book has been refined over a period of time, there are certainly still many errors, and I sincerely welcome critiques and corrections from teachers and fellow students.

> The code in this book is hosted in the github.com/krahets/hello-algo repository.
> For a better reading experience, please visit www.hello-algo.com.

Endorsements

> "An easy-to-understand book on data structures and algorithms, which guides readers to learn by minds-on and hands-on. Strongly recommended for algorithm beginners!"
> ——**Junhui Deng, Professor, Department of CS, Tsinghua University**

> "If I had 'Hello Algo' when I was learning data structures and algorithms, it would have been 10 times easier!"
> ——**Mu Li, Senior Principal Scientist, Amazon**

The advent of computers has brought tremendous changes to the world. With their high-speed computing capabilities and excellent programmability, they have become the ideal medium for executing algorithms and processing data. Whether it's the realistic graphics in video games, the intelligent decision-making in autonomous driving, AlphaGo's brilliant Go matches, or ChatGPT's natural interactions, these applications are all exquisite interpretations of algorithms on computers.

In fact, before the advent of computers, algorithms and data structures already existed in every corner of the world. Early algorithms were relatively simple, such as ancient counting methods and tool-making procedures. As civilization progressed, algorithms gradually became more refined and complex. From the ingenious craftsmanship of master artisans, to industrial products that liberate productive forces, to the scientific laws governing the operation of the universe, behind almost every ordinary or astonishing thing lies ingenious algorithmic thinking.

Similarly, data structures are everywhere: from large-scale social networks to small subway systems, many systems can be modeled as “graphs”; from a nation to a family, the primary organizational forms of society exhibit characteristics of “trees”; winter clothing is like a “stack,” where the first item put on is the last to be taken off; a badminton tube is like a “queue,” with items inserted at one end and retrieved from the other; a dictionary is like a “hash table,” enabling quick lookup of target entries.

This book aims to help readers understand the core concepts of algorithms and data structures through clear and accessible animated illustrations and runnable code examples, and to implement them through programming. Building on this foundation, the book endeavors to reveal the vivid manifestations of algorithms in the complex world and showcase the beauty of algorithms. I hope this book can be of help to you!

Table of Contents

Chapter 0. Preface

Abstract

Algorithms are like a beautiful symphony, each line of code flows like a melody.

May this book gently resonate in your mind, leaving a unique and profound melody.

0.1 About This Book

This project aims to create an open-source, free, beginner-friendly introductory tutorial on data structures and algorithms.

- The entire book uses animated illustrations, with clear and easy-to-understand content and a smooth learning curve, guiding beginners to explore the knowledge map of data structures and algorithms.
- The source code can be run with one click, helping readers improve their programming skills through practice and understand how algorithms work and the underlying implementation of data structures.
- We encourage readers to learn from each other, and everyone is welcome to ask questions and share insights in the comments section, making progress together through discussion and exchange.

0.1.1 Target Audience

If you are an algorithm beginner who has never been exposed to algorithms, or if you already have some problem-solving experience and have a vague understanding of data structures and algorithms, oscillating between knowing and not knowing, then this book is tailor-made for you!

If you have already accumulated a certain amount of problem-solving experience and are familiar with most question types, this book can help you review and organize your algorithm knowledge system, and the repository's source code can be used as a "problem-solving toolkit" or "algorithm dictionary."

If you are an algorithm "expert," we look forward to receiving your valuable suggestions, or participating in creation together.

Prerequisites
You need to have at least a programming foundation in any language, and be able to read and write simple code.

0.1.2 Content Structure

The main content of this book is shown in Figure 0-1.

- **Complexity analysis**: Evaluation dimensions and methods for data structures and algorithms. Methods for calculating time complexity and space complexity, common types, examples, etc.
- **Data structures**: Classification methods for basic data types and data structures. The definition, advantages and disadvantages, common operations, common types, typical applications, implementation methods, etc. of data structures such as arrays, linked lists, stacks, queues, hash tables, trees, heaps, and graphs.
- **Algorithms**: The definition, advantages and disadvantages, efficiency, application scenarios, problem-solving steps, and example problems of algorithms such as searching, sorting, divide and conquer, backtracking, dynamic programming, and greedy algorithms.

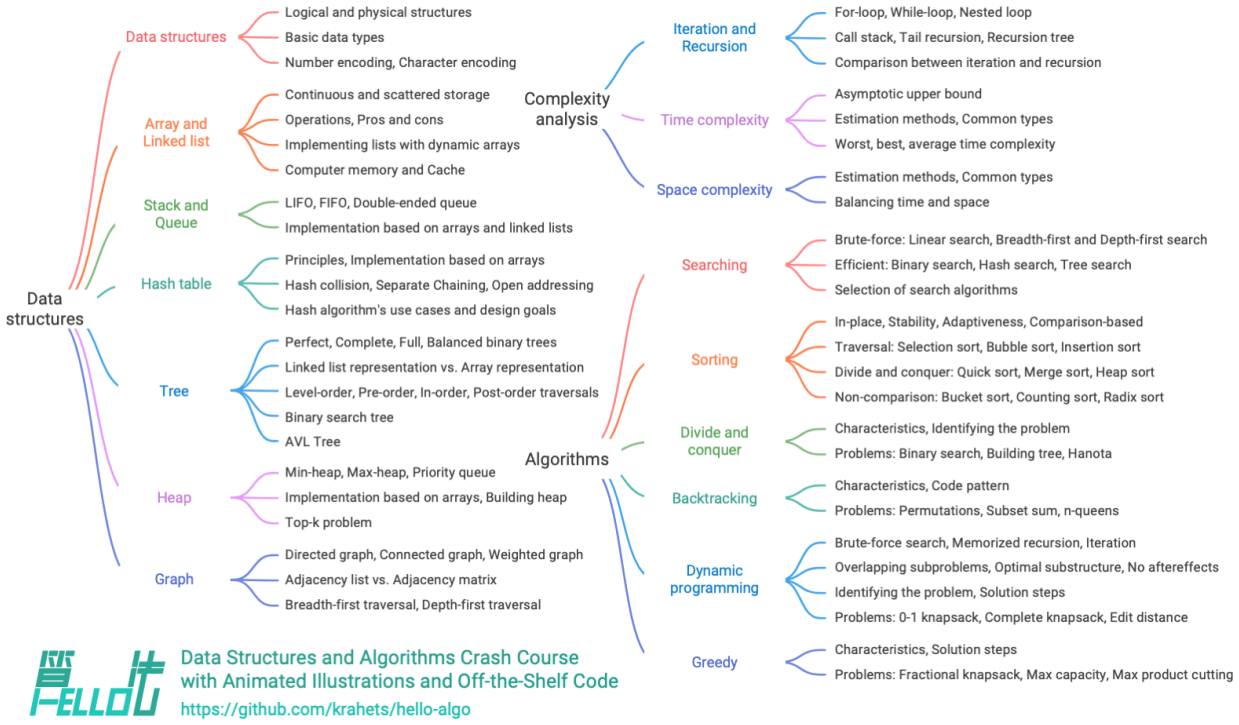


Figure 0-1 Main content of this book

0.1.3 Acknowledgements

This book has been continuously improved through the joint efforts of many contributors in the open-source community. Thanks to every contributor who invested time and effort, they are (in the order automatically generated by GitHub): krahets, coderonion, Gonglja, nuomi1, Reanon, justin-tse, hpstory, danielsss, curtishd, night-cruise, S-N-O-R-L-A-X, rongyi, msk397, gvenusleo, khoaxuantu, rivertwilight, K3v123, gyt95, zhuoqinyue, yuelinxin, Zuoxun, mingXta, Phoenix0415, FangYuan33, GN-Yu, longsizhuo, IsChristina, xBLACKICEx, guowei-gong, Cathay-Chen, pengchzn, QiLOL, magentaqin, hello-ikun, JoseHung, qualifier1024, thomasq0, sunshinesDL, L-Super, Guanngxu, Transmigration-zhou, WSL0809, Slone123c, lhxsm, yuan0221, what-is-me, Shyam-Chen, theNefelibatas, longranger2, codeberg-user, xiongsp, JeffersonHuang, prinpal, seven1240, Wonderdch, malone6, xiaomiusa87, gaofer, bluebean-cloud, a16su, SamJin98, hongyun-robot, nanlei, XiaChuerwu, yd-j, iron-irax, mgisr, steventimes, junminhong, heshuyue, danny900714, MolDuM, Nigh, Dr-XYZ, XC-Zero, reeswell, PXG-XPG, NI-SW, Horbin-Magician, Enlightenus, YangXuanyi, beatrix-chan, DullSword, xjr7670, jiaxianhua, qq909244296, iStig, boloboloda, hts0000, gledfish, wenjianmin, keshida, kilikilikid, lclc6, lwbaptx, linyejoe2, liuxjerry, llql1211, fbigm, echo1937, szu17dmy, dshlstarr, Yucao-cy, coderlef, czruby, bongbongbakudan, beintentional, ZongYangL, ZhongYuuu, ZhongGuanbin, hezhizhen, linzeyan, ZJKung, luluxia, xb534, ztkuaikuai, yw-1021, ElaBosak233, baagod, zhouLion, yishangzhang, yi427, yanedie, yabo083, weibk, wangwang105, th1nk3r-ing, tao363, 4yDX3906, syd168, sslmj2020, smilelsb, siqyka, selear, sdshaoda, Xi-Row, popozhu, nuquist19, noobcodemaker, XiaoK29, chadyi,

lyl625760, lucaswangdev, 0130w, shanghai-Jerry, EJackYang, Javesun99, eltociear, lipusheng, KNChiu, BlindTerran, ShiMaRing, lovelock, FreddieLi, FloranceYeh, fanchenggang, gltianwen, goerll, nedchu, curly210102, CuB3y0nd, KraHsu, CarrotDLaw, youshaoXG, bubble9um, Asashishi, Asa0oo0o0o, fanenr, eagleanurag, akshiterate, 52coder, foursevenlove, KorsChen, GaochaoZhu, hopkings2008, yang-le, realwujing, Evilrabbit520, Umer-Jahangir, Turing-1024-Lee, Suremotoo, paoxiaomooo, Chieko-Seren, Allen-Scai, ymmmas, Risuntsy, Richard-Zhang1019, RafaelCaso, qingpeng9802, primexiao, Urbaner3, zhongfq, nidhoggfgg, MwumLi, CreatorMetaSky, martinx, ZnYang2018, hugtyftg, logan-qiu, psychelzh, Keynman, KeiichiKasai, and KawaiiAsh.

The code review work for this book was completed by coderonion, curtishd, Gonglja, gvenusleo, hpstory, justin-tse, khoaxuantu, krahets, night-cruise, nuomi1, Reanon and rongyi (in alphabetical order). Thanks to them for the time and effort they put in, it is they who ensure the standardization and unity of code in various languages.

The Traditional Chinese version of this book was reviewed by Shyam-Chen and Dr-XYZ, and the English version was reviewed by yuelinxin, K3v123, QiLOL, Phoenix0415, SamJin98, yanedie, RafaelCaso, pengchzn, thomasq0 and magentaqin. It is because of their continuous contributions that this book can serve a wider readership, and we thank them.

The ePub ebook generation tool for this book was developed by zhongfq. We thank him for his contribution, which provides readers with a more flexible way to read.

During the creation of this book, I received help from many people.

- Thanks to my mentor at the company, Dr. Li Xi, who encouraged me to "take action quickly" during a conversation, strengthening my determination to write this book;
- Thanks to my girlfriend Bubble as the first reader of this book, who provided many valuable suggestions from the perspective of an algorithm beginner, making this book more suitable for novices to read;
- Thanks to Tengbao, Qibao, and Feibao for coming up with a creative name for this book, evoking everyone's fond memories of writing their first line of code "Hello World!";
- Thanks to Xiaoquan for providing professional help in intellectual property rights, which played an important role in the improvement of this open-source book;
- Thanks to Sutong for designing the beautiful cover and logo for this book, and for patiently making revisions multiple times driven by my obsessive-compulsive disorder;
- Thanks to @squidfunk for the typesetting suggestions, as well as for developing the open-source documentation theme Material-for-MkDocs.

During the writing process, I read many textbooks and articles on data structures and algorithms. These works provided excellent examples for this book and ensured the accuracy and quality of the book's content. I would like to thank all the teachers and predecessors for their outstanding contributions!

This book advocates a learning method that combines hands and brain, and in this regard I was deeply inspired by Dive into Deep Learning. I highly recommend this excellent work to all readers.

Heartfelt thanks to my parents, it is your support and encouragement that has given me the opportunity to do this interesting thing.

0.2 How to Use This Book

> **Tip**
> For the best reading experience, it is recommended that you read through this section.

0.2.1 Writing Style Conventions

- Titles marked with `*` are optional sections with relatively difficult content. If you have limited time, you can skip them first.
- Technical terms will be in bold (in paper and PDF versions) or underlined (in web versions), such as array. It is recommended to memorize them for reading literature.
- Key content and summary statements will be **bolded**, and such text deserves special attention.
- Words and phrases with specific meanings will be marked with "quotation marks" to avoid ambiguity.
- When it comes to nouns that are inconsistent between programming languages, this book uses Python as the standard, for example, using `None` to represent "null".
- This book partially abandons the comment conventions of programming languages in favor of more compact content layout. Comments are mainly divided into three types: title comments, content comments, and multi-line comments.

```
/* Title comment, used to label functions, classes, test cases, etc. */

// Content comment, used to explain code in detail

/**
 * Multi-line
 * comment
 */
```

0.2.2 Learning Efficiently with Animated Illustrations

Compared to text, videos and images have higher information density and structural organization, making them easier to understand. In this book, **key and difficult knowledge will mainly be presented in the form of animated illustrations**, with text serving as explanation and supplement.

If you find that a section of content provides animated illustrations as shown in Figure 0-2 while reading this book, **please focus on the illustrations first, with text as a supplement**, and combine the two to understand the content.

图 9-7 邻接矩阵的初始化、增删边、增删顶点

以下是基于邻接矩阵表示图的实现代码：

Python C++ Java C# Go Swift JS TS Dart Rust C Zig

graph_adjacency_matrix.py

Figure 0-2 Example of animated illustrations

0.2.3 Deepening Understanding Through Code Practice

The accompanying code for this book is hosted in the GitHub repository. As shown in Figure 0-3, **the source code comes with test cases and can be run with one click**.

If time permits, **it is recommended that you type out the code yourself**. If you have limited study time, please at least read through and run all the code.

Compared to reading code, the process of writing code often brings more rewards. **Learning by doing is the real learning**.

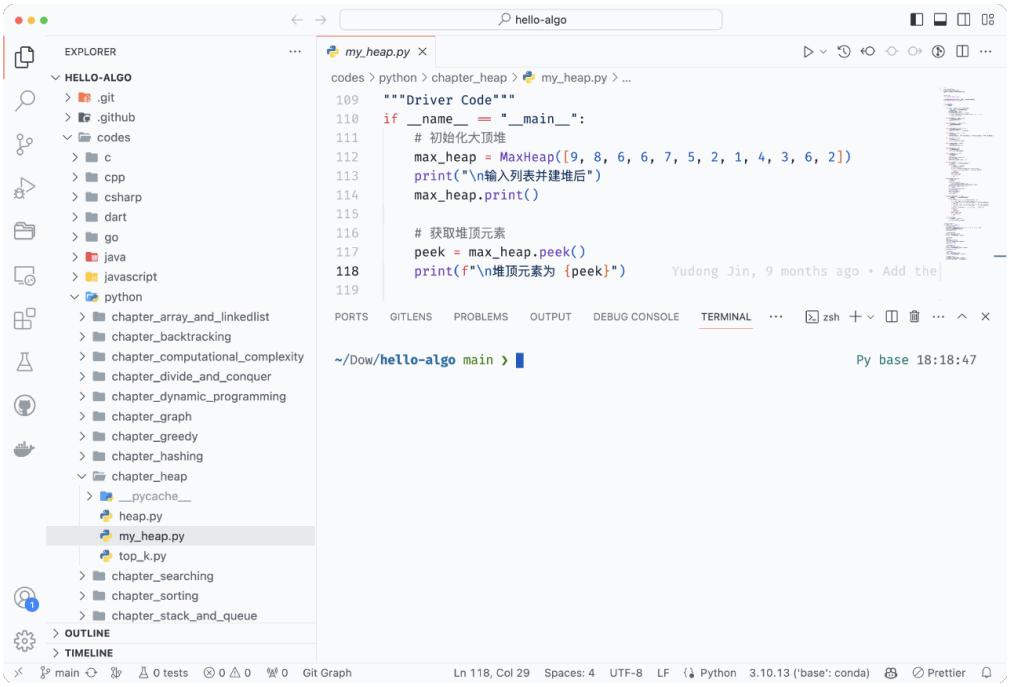


Figure 0-3 Example of running code

The preliminary work for running code is mainly divided into three steps.

Step 1: Install the local programming environment. Please follow the tutorial shown in the appendix for installation. If already installed, you can skip this step.

Step 2: Clone or download the code repository. Visit the GitHub repository. If you have already installed Git, you can clone this repository with the following command:

```
git clone https://github.com/krahets/hello-algo.git
```

Of course, you can also click the "Download ZIP" button at the location shown in Figure 0-4 to directly download the code compressed package, and then extract it locally.

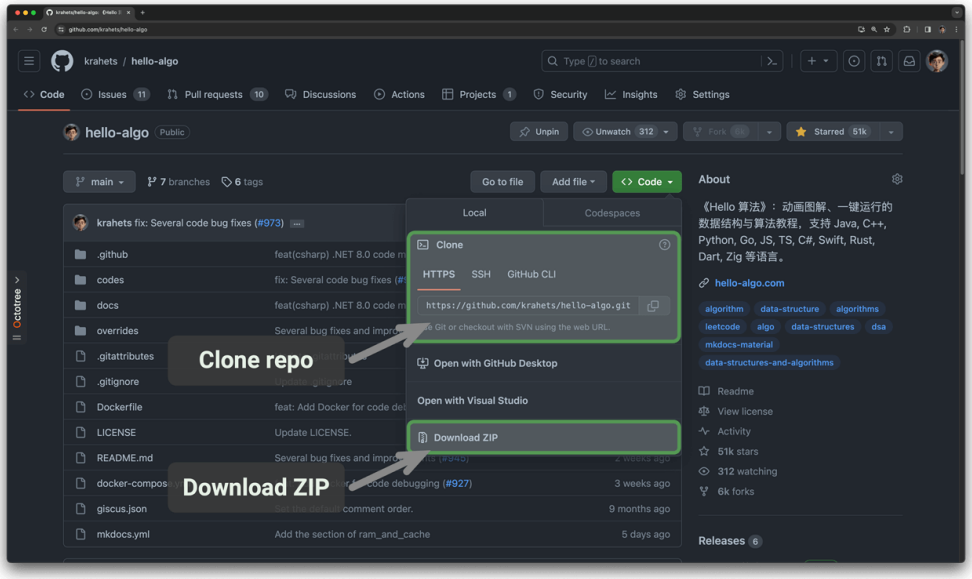


Figure 0-4 Clone repository and download code

Step 3: Run the source code. As shown in Figure 0-5, for code blocks with file names at the top, we can find the corresponding source code files in the `codes` folder of the repository. The source code files can be run with one click, which will help you save unnecessary debugging time and allow you to focus on learning content.

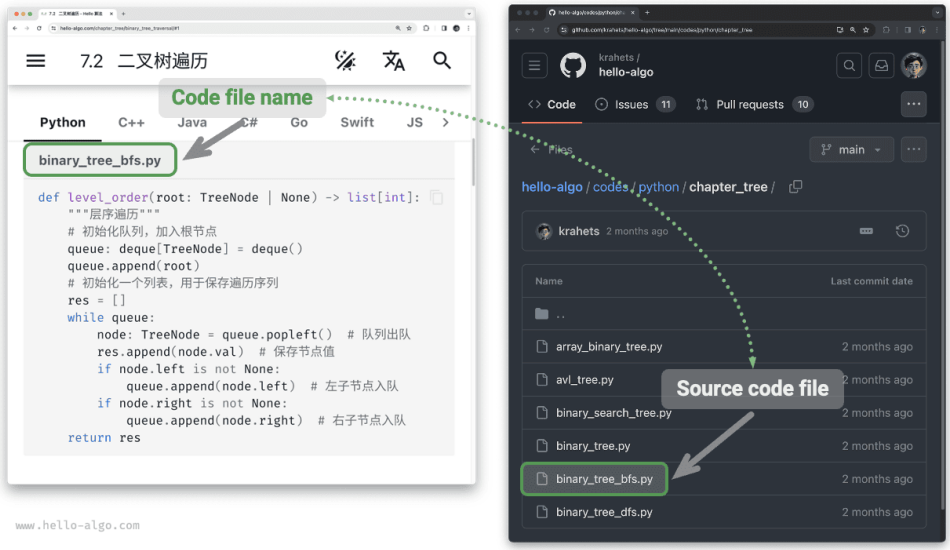


Figure 0-5 Code blocks and corresponding source code files

In addition to running code locally, **the web version also supports visual running of Python code** (implemented based on pythontutor). As shown in Figure 0-6, you can click "Visual Run" below the code block to expand the view and observe the execution process of the algorithm code; you can also click "Full Screen View" for a better viewing experience.

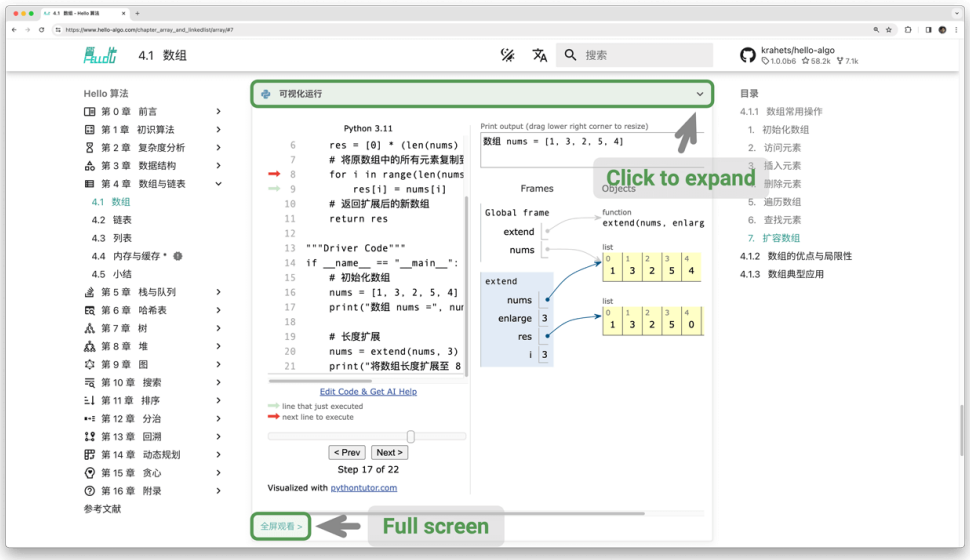


Figure 0-6 Visual running of Python code

0.2.4 Growing Together Through Questions and Discussions

When reading this book, please do not easily skip knowledge points that you have not learned well. **Feel free to ask your questions in the comments section**, and my friends and I will do our best to answer you, and generally reply within two days.

As shown in Figure 0-7, the web version has a comments section at the bottom of each chapter. I hope you will pay more attention to the content of the comments section. On the one hand, you can learn about the problems that everyone encounters, thus checking for omissions and stimulating deeper thinking. On the other hand, I hope you can generously answer other friends' questions, share your insights, and help others progress.

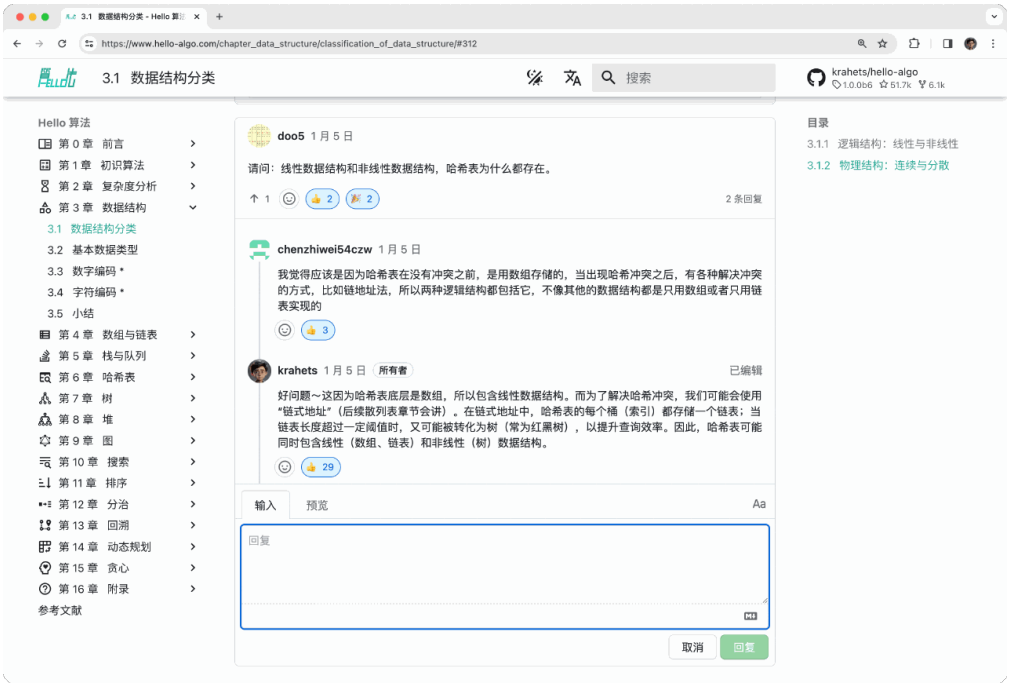


Figure 0-7 Example of comments section

0.2.5 Algorithm Learning Roadmap

From an overall perspective, we can divide the process of learning data structures and algorithms into three stages.

1. **Stage 1: Algorithm introduction**. We need to familiarize ourselves with the characteristics and usage of various data structures, and learn the principles, processes, uses, and efficiency of different algorithms.
2. **Stage 2: Practice algorithm problems**. It is recommended to start with popular problems, and accumulate at least 100 problems first, to familiarize yourself with mainstream algorithm problems. When first practicing problems, "knowledge forgetting" may be a challenge, but rest assured, this is very normal. We can review problems according to the "Ebbinghaus forgetting curve", and usually after 3-5 rounds of repetition, we can firmly remember them. For recommended problem lists and practice plans, please see this GitHub repository.
3. **Stage 3: Building a knowledge system**. In terms of learning, we can read algorithm column articles, problem-solving frameworks, and algorithm textbooks to continuously enrich our knowledge system. In terms of practicing problems, we can try advanced problem-solving strategies, such as categorization by topic, one problem multiple solutions, one solution multiple problems, etc. Related problem-solving insights can be found in various communities.

As shown in Figure 0-8, the content of this book mainly covers "Stage 1", aiming to help you more efficiently carry out Stage 2 and Stage 3 learning.

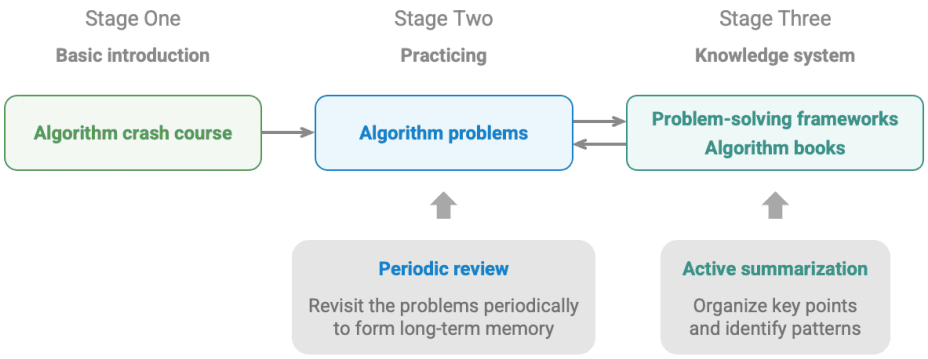


Figure 0-8 Algorithm learning roadmap

0.3 Summary

1. Key Review

- The main audience of this book is algorithm beginners. If you already have a certain foundation, this book can help you systematically review algorithm knowledge, and the source code in the book can also be used as a "problem-solving toolkit."
- The content of the book mainly includes three parts: complexity analysis, data structures, and algorithms, covering most topics in this field.
- For algorithm novices, reading an introductory book during the initial learning stage is crucial, as it can help you avoid many detours.
- The animated illustrations in the book are usually used to introduce key and difficult knowledge. When reading this book, you should pay more attention to these contents.
- Practice is the best way to learn programming. It is strongly recommended to run the source code and type the code yourself.
- The web version of this book has a comments section for each chapter, where you are welcome to share your questions and insights at any time.

Chapter 1. Encounter with Algorithms

Abstract

A young girl dances gracefully, intertwined with data, her skirt flowing with the melody of algorithms.

She invites you to dance with her. Follow her steps closely and enter the world of algorithms, full of logic and beauty.

1.1 Algorithms Are Everywhere

When we hear the term "algorithm," we naturally think of mathematics. However, many algorithms do not involve complex mathematics but rely more on basic logic, which can be seen everywhere in our daily lives.

Before we start discussing about algorithms officially, there's an interesting fact worth sharing: **you've learned many algorithms unconsciously and are used to applying them in your daily life**. Here, I will give a few specific examples to prove this point.

Example 1: Looking Up a Dictionary. In an English dictionary, words are listed alphabetically. Assuming we're searching for a word that starts with the letter r, this is typically done in the following way:

1. Open the dictionary to about halfway and check the first vocabulary of the page, let's say the letter starts with m.
2. Since r comes after m in the alphabet, the first half can be ignored and the search space is narrowed down to the second half.
3. Repeat steps `1.` and `2.` until you find the page where the word starts with r.

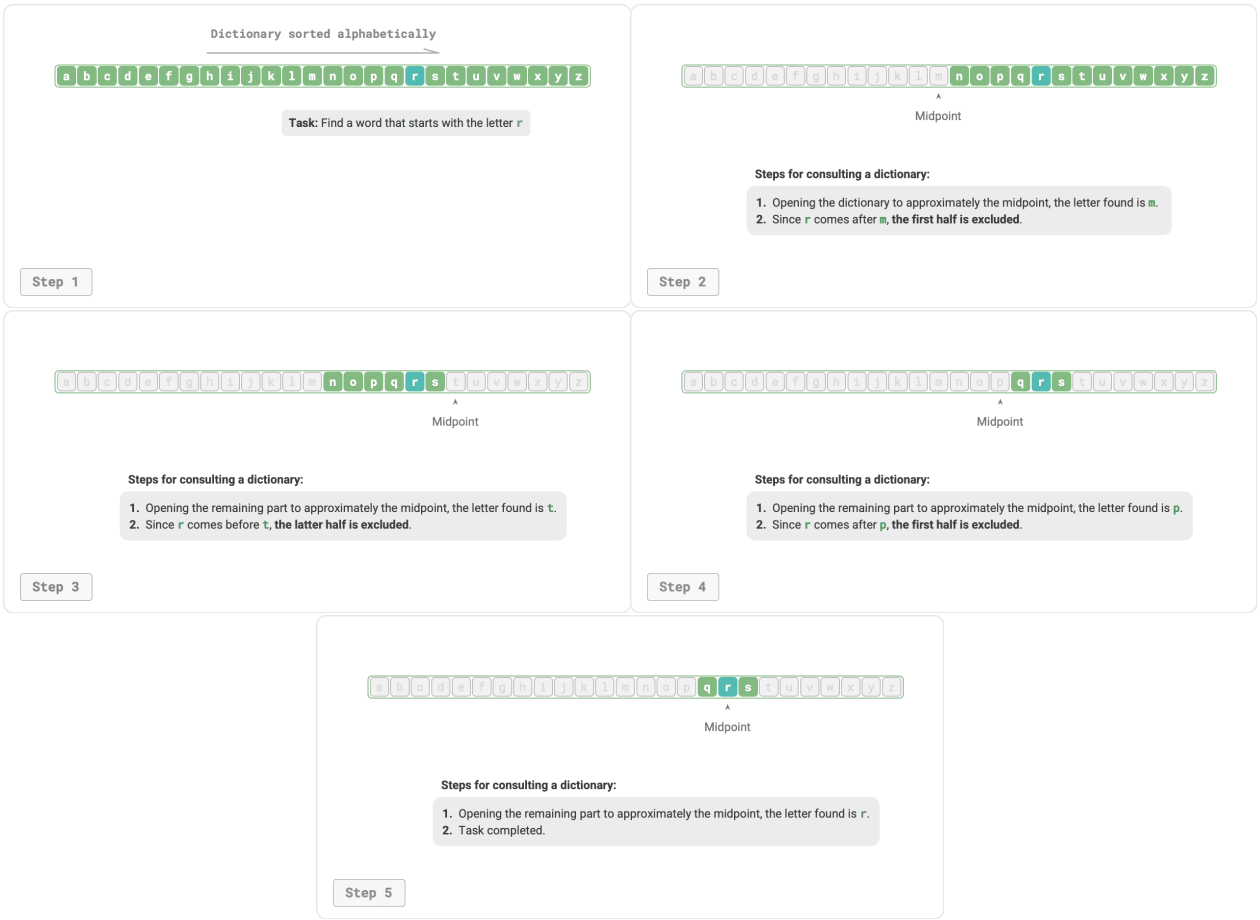


Figure 1-1 Process of looking up a dictionary

Looking up a dictionary, an essential skill for elementary school students is actually the famous "Binary Search" algorithm. From a data structure perspective, we can consider the dictionary as a sorted "array"; from an algorithmic perspective, the series of actions taken to look up a word in the dictionary can be viewed as the algorithm "Binary Search."

Example 2: Organizing Card Deck. When playing cards, we need to arrange the cards in our hands in ascending order, as shown in the following process.

1. Divide the playing cards into "ordered" and "unordered" sections, assuming initially the leftmost card is already in order.
2. Take out a card from the unordered section and insert it into the correct position in the ordered section; after this, the leftmost two cards are in order.
3. Repeat step `2` until all cards are in order.

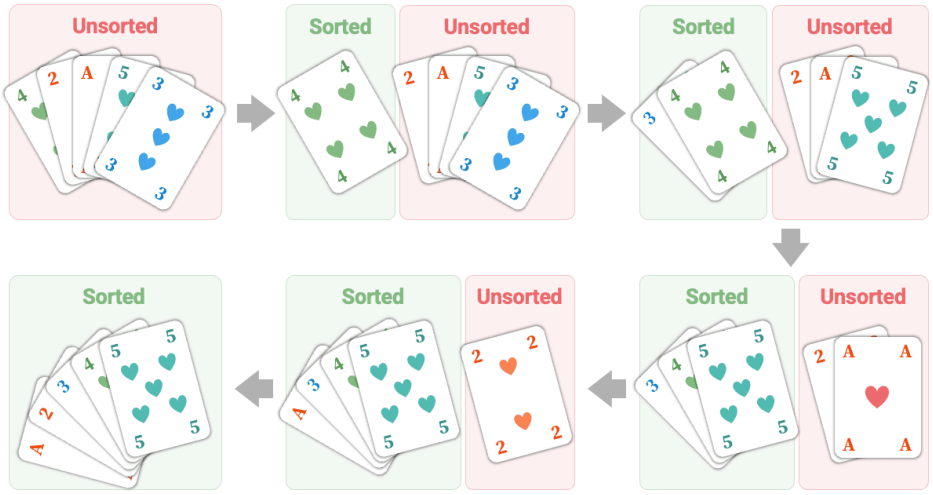


Figure 1-2 Process of sorting a deck of cards

The above method of organizing playing cards is practically the "Insertion Sort" algorithm, which is very efficient for small datasets. Many programming languages' sorting functions include the insertion sort.

Example 3: Making Change. Assume making a purchase of 69 at a supermarket. If you give the cashier 100, they will need to provide you with 31 in change. This process can be clearly understood as illustrated in Figure 1-3.

1. The options are currencies valued below 31, including 1, 5, 10, and 20.
2. Take out the largest 20 from the options, leaving $31 - 20 = 11$.
3. Take out the largest 10 from the remaining options, leaving $11 - 10 = 1$.
4. Take out the largest 1 from the remaining options, leaving $1 - 1 = 0$.
5. Complete change-making, the solution is $20 + 10 + 1 = 31$.

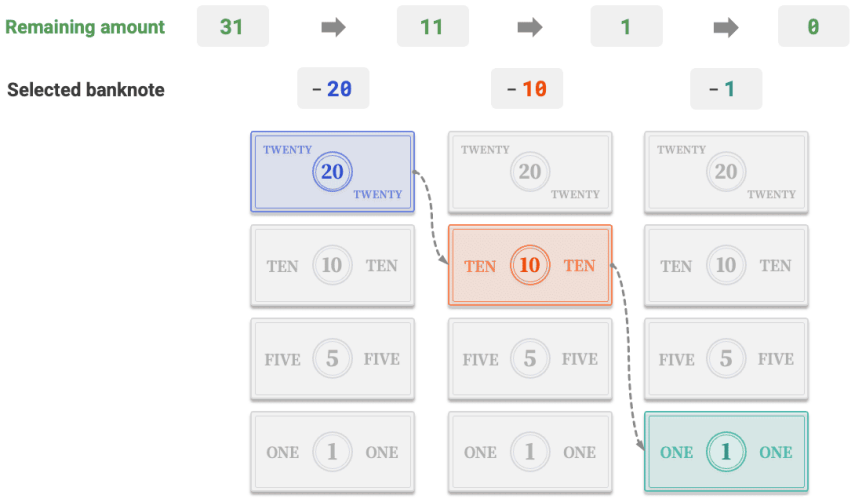


Figure 1-3 Process of making change

In the steps described, we choose the best option at each stage by utilizing the largest denomination available, which leads to an effective change-making strategy. From a data structures and algorithms perspective, this approach is known as a "Greedy" algorithm.

From cooking a meal to interstellar travel, almost all problem-solving involves algorithms. The advent of computers allows us to store data structures in memory and write code to call the CPU and GPU to execute algorithms. In this way, we can transfer real-life problems to computers and solve various complex issues in a more efficient way.

Tip

If you are still confused about concepts like data structures, algorithms, arrays, and binary searches, I encourage you to keep reading. This book will gently guide you into the realm of understanding data structures and algorithms.

1.2 What Is an Algorithm

1.2.1 Algorithm Definition

An algorithm is a set of instructions or operational steps that solves a specific problem within a finite amount of time. It has the following characteristics.

- The problem is well-defined, with clear input and output definitions.
- It is feasible and can be completed within a finite number of steps, time, and memory space.
- Each step has a definite meaning, and under the same input and operating conditions, the output is always the same.

1.2.2 Data Structure Definition

A data structure is a way of organizing and storing data, covering the data content, relationships between data, and methods for data operations. It has the following design objectives.

- Occupy as little space as possible to save computer memory.
- Data operations should be as fast as possible, covering data access, addition, deletion, update, etc.
- Provide a concise data representation and logical information so that algorithms can run efficiently.

Data structure design is a process full of trade-offs. If we want to achieve improvements in one aspect, we often need to make compromises in another aspect. Here are two examples.

- Compared to arrays, linked lists are more convenient for data addition and deletion operations but sacrifice data access speed.
- Compared to linked lists, graphs provide richer logical information but require larger memory space.

1.2.3 The Relationship Between Data Structures and Algorithms

As shown in Figure 1-4, data structures and algorithms are highly related and tightly coupled, specifically manifested in the following three aspects.

- Data structures are the foundation of algorithms. Data structures provide algorithms with structured storage of data and methods for operating on data.
- Algorithms breathe life into data structures. Data structures themselves only store data information; combined with algorithms, they can solve specific problems.
- Algorithms can usually be implemented based on different data structures, but execution efficiency may vary greatly. Choosing the appropriate data structure is key.

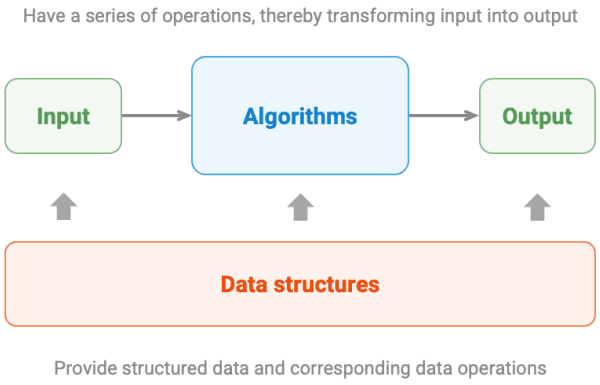


Figure 1-4 The relationship between data structures and algorithms

Data structures and algorithms are like assembling building blocks as shown in Figure 1-5. A set of building blocks, in addition to containing many parts, also comes with detailed assembly instructions. By following the instructions step by step, we can assemble an exquisite building block model.

Figure 1-5 Assembling blocks

The detailed correspondence between the two is shown in Table 1-1.

Table 1-1 Comparing data structures and algorithms to assembling building blocks

Data structures and algorithms	Assembling building blocks
Input data	Unassembled building blocks
Data structure	Organization form of building blocks, including shape, size, connection method, etc.
Algorithm	A series of operational steps to assemble the blocks into the target form
Output data	Building block model

It is worth noting that data structures and algorithms are independent of programming languages. For this reason, this book is able to provide implementations based on multiple programming languages.

Conventional abbreviation
In actual discussions, we usually abbreviate "data structures and algorithms" as "algorithms". For example, the well-known LeetCode algorithm problems actually examine knowledge of both data structures and algorithms.

1.3 Summary

1. Key Review

- Algorithms are ubiquitous in daily life and are not distant, esoteric knowledge. In fact, we have already learned many algorithms unconsciously and use them to solve problems big and small in life.
- The principle of looking up a dictionary is consistent with the binary search algorithm. Binary search embodies the important algorithmic idea of divide and conquer.
- The process of organizing playing cards is very similar to the insertion sort algorithm. Insertion sort is suitable for sorting small datasets.
- The steps of making change are essentially a greedy algorithm, where the best choice is made at each step based on the current situation.
- An algorithm is a set of instructions or operational steps that solves a specific problem within a finite amount of time, while a data structure is the way computers organize and store data.
- Data structures and algorithms are closely connected. Data structures are the foundation of algorithms, and algorithms breathe life into data structures.
- We can compare data structures and algorithms to assembling building blocks. The blocks represent data, the shape and connection method of the blocks represent the data structure, and the steps to assemble the blocks correspond to the algorithm.

2. Q & A

Q: As a programmer, I have never used algorithms to solve problems in my daily work. Common algorithms are already encapsulated by programming languages and can be used directly. Does this mean that the problems in our work have not yet reached the level where algorithms are needed?

If we compare specific work skills to "techniques" in martial arts, then fundamental subjects should be more like "internal skills".

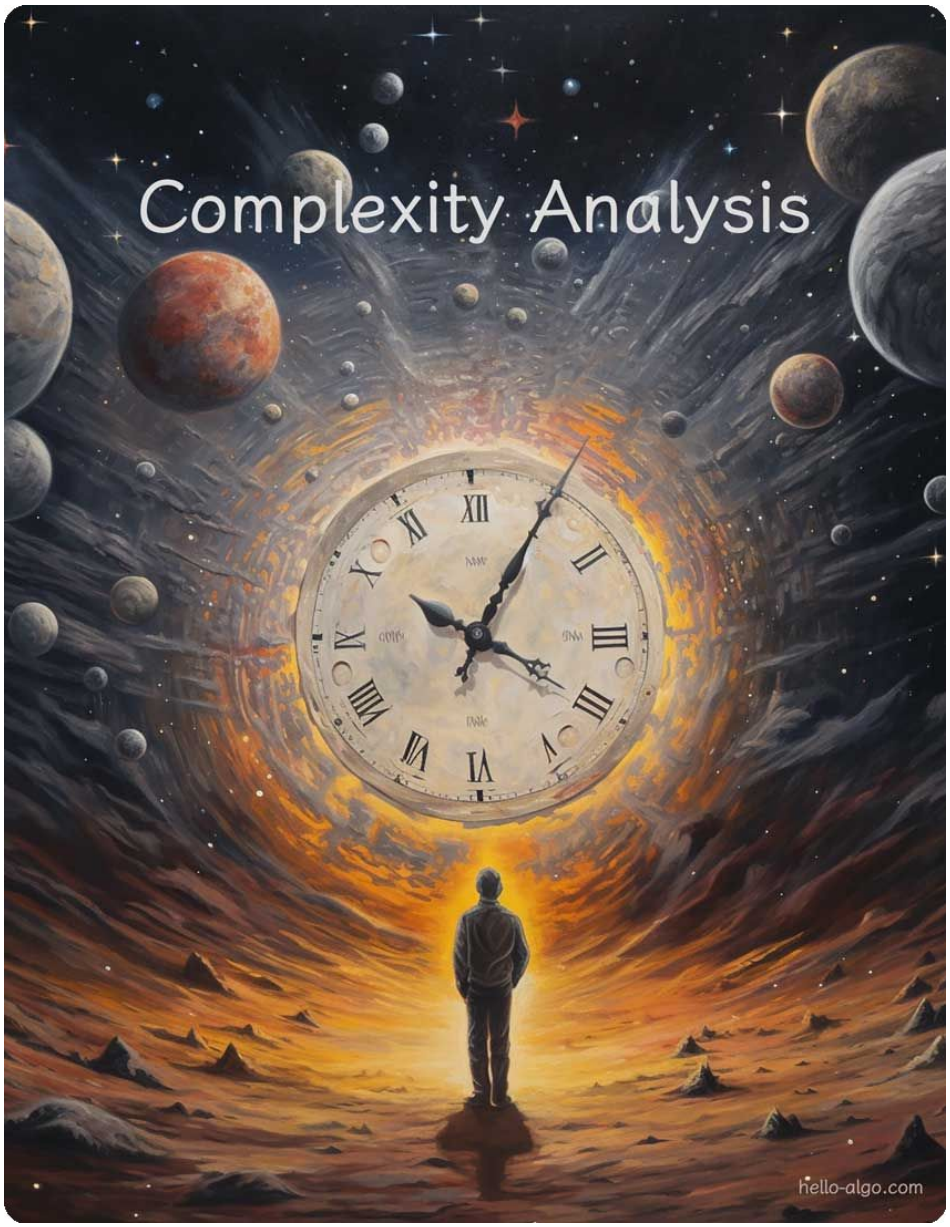
I believe the significance of learning algorithms (and other fundamental subjects) is not to implement them from scratch at work, but rather to be able to make professional reactions and judgments when solving problems based on the knowledge learned, thereby improving the overall quality of work. Here is a simple example. Every programming language has a built-in sorting function:

- If we have not studied data structures and algorithms, we might simply feed any given data to this sorting function. It runs smoothly with good performance, and there doesn't seem to be any problem.
- But if we have studied algorithms, we would know that the time complexity of the built-in sorting function is $O(n \log n)$. However, if the given data consists of integers with a fixed number of digits (such as student IDs), we can use the more efficient "radix sort", reducing the time complexity to $O(nk)$, where k is the number of digits. When the data volume is very large, the saved running time can create significant value (reduced costs, improved experience, etc.).

In the field of engineering, a large number of problems are difficult to reach optimal solutions, and many problems are only solved "approximately". The difficulty of a problem depends on one hand on the

nature of the problem itself, and on the other hand on the knowledge reserve of the person observing the problem. The more complete a person's knowledge and the more experience they have, the deeper their analysis of the problem will be, and the more elegantly the problem can be solved.

Chapter 2. Complexity Analysis



Abstract

Complexity analysis is like a space-time guide in the vast universe of algorithms.

It leads us to explore deeply within the two dimensions of time and space, seeking more elegant solutions.

2.1 Algorithm Efficiency Evaluation

In algorithm design, we pursue the following two levels of objectives sequentially.

1. **Finding a solution to the problem**: The algorithm must reliably obtain the correct solution within the specified input range.
2. **Seeking the optimal solution**: Multiple solutions may exist for the same problem, and we hope to find an algorithm that is as efficient as possible.

In other words, under the premise of being able to solve the problem, algorithm efficiency has become the primary evaluation criterion for measuring the quality of algorithms. It includes the following two dimensions.

- **Time efficiency**: The length of time the algorithm runs.
- **Space efficiency**: The size of memory space the algorithm occupies.

In short, **our goal is to design data structures and algorithms that are "both fast and memory-efficient"**. Effectively evaluating algorithm efficiency is crucial, because only in this way can we compare various algorithms and guide the algorithm design and optimization process.

Efficiency evaluation methods are mainly divided into two types: actual testing and theoretical estimation.

2.1.1 Actual Testing

Suppose we now have algorithm `A` and algorithm `B`, both of which can solve the same problem, and we need to compare the efficiency of these two algorithms. The most direct method is to find a computer, run these two algorithms, and monitor and record their running time and memory usage. This evaluation approach can reflect the real situation, but it also has considerable limitations.

On one hand, **it is difficult to eliminate interference factors from the testing environment**. Hardware configuration affects the performance of algorithms. For example, if an algorithm has a high degree of parallelism, it is more suitable for running on multi-core CPUs; if an algorithm has intensive memory operations, it will perform better on high-performance memory. In other words, the test results of an algorithm on different machines may be inconsistent. This means we need to test on various machines and calculate average efficiency, which is impractical.

On the other hand, **conducting complete testing is very resource-intensive**. As the input data volume changes, the algorithm will exhibit different efficiencies. For example, when the input data volume is small, the running time of algorithm `A` is shorter than algorithm `B`; but when the input data volume is large, the test results may be exactly the opposite. Therefore, to obtain convincing conclusions, we need to test input data of various scales, which requires a large amount of computational resources.

2.1.2 Theoretical Estimation

Since actual testing has considerable limitations, we can consider evaluating algorithm efficiency through calculations alone. This estimation method is called asymptotic complexity analysis, or

complexity analysis for short.

Complexity analysis can reflect the relationship between the time and space resources required for algorithm execution and the input data scale. **It describes the growth trend of the time and space required for algorithm execution as the input data scale increases**. This definition is somewhat convoluted, so we can break it down into three key points to understand.

- "Time and space resources" correspond to time complexity and space complexity, respectively.
- "As the input data scale increases" means that complexity reflects the relationship between algorithm running efficiency and input data scale.
- "Growth trend of time and space" indicates that complexity analysis focuses not on the specific values of running time or occupied space, but on how "fast" time or space grows.

Complexity analysis overcomes the drawbacks of the actual testing method, reflected in the following aspects.

- It does not need to actually run the code, making it more environmentally friendly and energy-efficient.
- It is independent of the testing environment, and the analysis results are applicable to all running platforms.
- It can reflect algorithm efficiency at different data volumes, especially algorithm performance at large data volumes.

> **Tip**
> If you are still confused about the concept of complexity, don't worry—we will introduce it in detail in subsequent chapters.

Complexity analysis provides us with a "ruler" for evaluating algorithm efficiency, allowing us to measure the time and space resources required to execute a certain algorithm and compare the efficiency between different algorithms.

Complexity is a mathematical concept that may be relatively abstract for beginners, with a relatively high learning difficulty. From this perspective, complexity analysis may not be very suitable as the first content to be introduced. However, when we discuss the characteristics of a certain data structure or algorithm, it is difficult to avoid analyzing its running speed and space usage.

In summary, it is recommended that before diving deep into data structures and algorithms, **you first establish a preliminary understanding of complexity analysis so that you can complete complexity analysis of simple algorithms**.

2.2 Iteration and Recursion

In algorithms, repeatedly executing a task is very common and closely related to complexity analysis. Therefore, before introducing time complexity and space complexity, let's first understand how to implement repeated task execution in programs, namely the two basic program control structures: iteration and recursion.

2.2.1 Iteration

Iteration is a control structure for repeatedly executing a task. In iteration, a program repeatedly executes a segment of code under certain conditions until those conditions are no longer satisfied.

1. For Loop

The `for` loop is one of the most common forms of iteration, **suitable for use when the number of iterations is known in advance**.

The following function implements the summation $1 + 2 + \cdots + n$ based on a `for` loop, with the sum result recorded using the variable `res`. Note that in Python, `range(a, b)` corresponds to a "left-closed, right-open" interval, with the traversal range being $a, a+1, \ldots, b-1$:

```
// ═══ File: iteration.js ═══

/* for loop */
function forLoop(n) {
    let res = 0;
    // Sum 1, 2, ..., n-1, n
    for (let i = 1; i <= n; i++) {
        res += i;
    }
    return res;
}
```

Figure 2-1 shows the flowchart of this summation function.

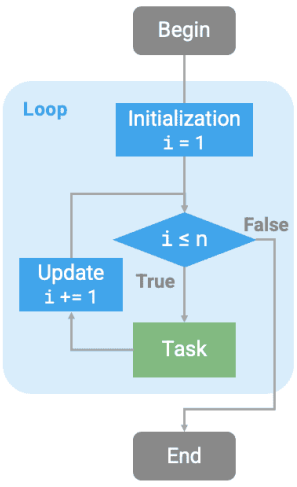


Figure 2-1 Flowchart of the summation function

The number of operations in this summation function is proportional to the input data size n, or has a "linear relationship". In fact, **time complexity describes precisely this "linear relationship"**. Related content will be introduced in detail in the next section.

2. While Loop

Similar to the `for` loop, the `while` loop is also a method for implementing iteration. In a `while` loop, the program first checks the condition in each round; if the condition is true, it continues execution, otherwise it ends the loop.

Below we use a `while` loop to implement the summation $1 + 2 + \cdots + n$:

```
// ═══ File: iteration.js ═══

/* while loop */
function whileLoop(n) {
    let res = 0;
    let i = 1; // Initialize condition variable
    // Sum 1, 2, ..., n-1, n
    while (i <= n) {
        res += i;
        i++; // Update condition variable
    }
    return res;
}
```

The `while` loop has greater flexibility than the `for` loop. In a `while` loop, we can freely design the initialization and update steps of the condition variable.

For example, in the following code, the condition variable i is updated twice per round, which is not convenient to implement using a `for` loop:

```
// ═══ File: iteration.js ═══

/* while loop (two updates) */
function whileLoopII(n) {
    let res = 0;
    let i = 1; // Initialize condition variable
    // Sum 1, 4, 10, ...
    while (i <= n) {
        res += i;
        // Update condition variable
        i++;
        i *= 2;
    }
    return res;
}
```

Overall, **`for` loops have more compact code, while `while` loops are more flexible**; both can implement iterative structures. The choice of which to use should be determined based on the requirements of the specific problem.

3. Nested Loops

We can nest one loop structure inside another. Below is an example using `for` loops:

```
// ═══ File: iteration.js ═══

/* Nested for loop */
function nestedForLoop(n) {
    let res = '';
    // Loop i = 1, 2, ..., n-1, n
    for (let i = 1; i <= n; i++) {
        // Loop j = 1, 2, ..., n-1, n
        for (let j = 1; j <= n; j++) {
            res += `(${i}, ${j}), `;
        }
    }
    return res;
}
```

Figure 2-2 shows the flowchart of this nested loop.

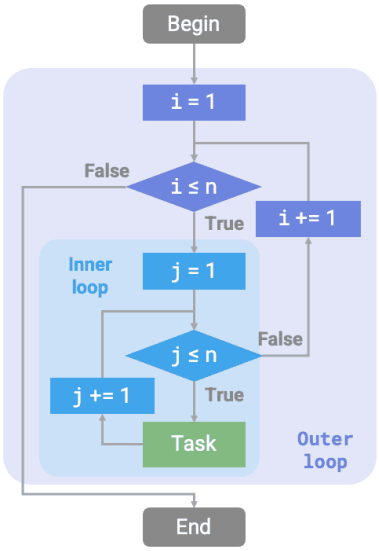


Figure 2-2 Flowchart of nested loops

In this case, the number of operations of the function is proportional to n^2, or the algorithm's running time has a "quadratic relationship" with the input data size n.

We can continue adding nested loops, where each nesting is a "dimension increase", raising the time complexity to "cubic relationship", "quartic relationship", and so on.

2.2.2 Recursion

Recursion is an algorithmic strategy that solves problems by having a function call itself. It mainly consists of two phases.

1. **Descend**: The program continuously calls itself deeper, usually passing in smaller or more simplified parameters, until reaching a "termination condition".

2. **Ascend**: After triggering the "termination condition", the program returns layer by layer from the deepest recursive function, aggregating the result of each layer.

From an implementation perspective, recursive code mainly consists of three elements.

1. **Termination condition**: Used to determine when to switch from "descending" to "ascending".
2. **Recursive call**: Corresponds to "descending", where the function calls itself, usually with smaller or more simplified parameters.
3. **Return result**: Corresponds to "ascending", returning the result of the current recursion level to the previous layer.

Observe the following code. We only need to call the function `recur(n)` to complete the calculation of $1 + 2 + \cdots + n$:

```
// ═══ File: recursion.js ═══

/* Recursion */
function recur(n) {
    // Termination condition
    if (n ═══ 1) return 1;
    // Recurse: recursive call
    const res = recur(n - 1);
    // Return: return result
    return n + res;
}
```

Figure 2-3 shows the recursive process of this function.

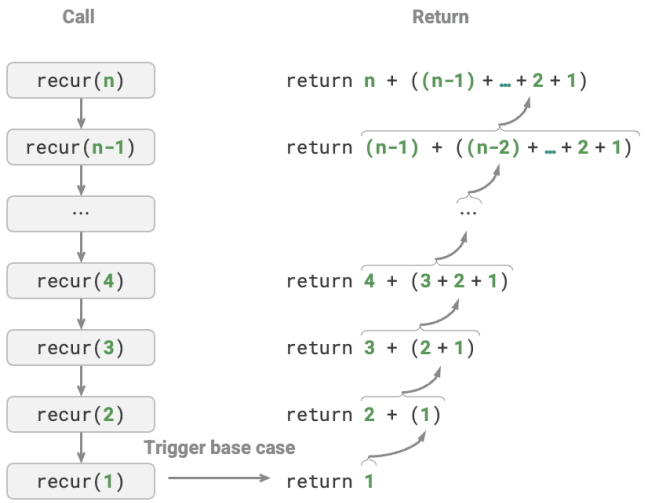


Figure 2-3 Recursive process of the summation function

Although from a computational perspective, iteration and recursion can achieve the same results, **they represent two completely different paradigms for thinking about and solving problems**.

- **Iteration**: Solves problems "bottom-up". Starting from the most basic steps, these steps are then repeatedly executed or accumulated until the task is complete.
- **Recursion**: Solves problems "top-down". The original problem is decomposed into smaller subproblems that have the same form as the original problem. These subproblems continue to be decomposed into even smaller subproblems until reaching the base case (where the solution is known).

Taking the above summation function as an example, let the problem be $f(n) = 1 + 2 + \cdots + n$.

- **Iteration**: Simulates the summation process in a loop, traversing from 1 to n, performing the summation operation in each round to obtain $f(n)$.
- **Recursion**: Decomposes the problem into the subproblem $f(n) = n + f(n-1)$, continuously decomposing (recursively) until terminating at the base case $f(1) = 1$.

1. Call Stack

Each time a recursive function calls itself, the system allocates memory for the newly opened function to store local variables, call addresses, and other information. This leads to two consequences.

- The function's context data is stored in a memory area called "stack frame space", which is not released until the function returns. Therefore, **recursion usually consumes more memory space than iteration**.
- Recursive function calls incur additional overhead. **Therefore, recursion is usually less time-efficient than loops**.

As shown in Figure 2-4, before the termination condition is triggered, there are n unreturned recursive functions existing simultaneously, with a **recursion depth of** n.

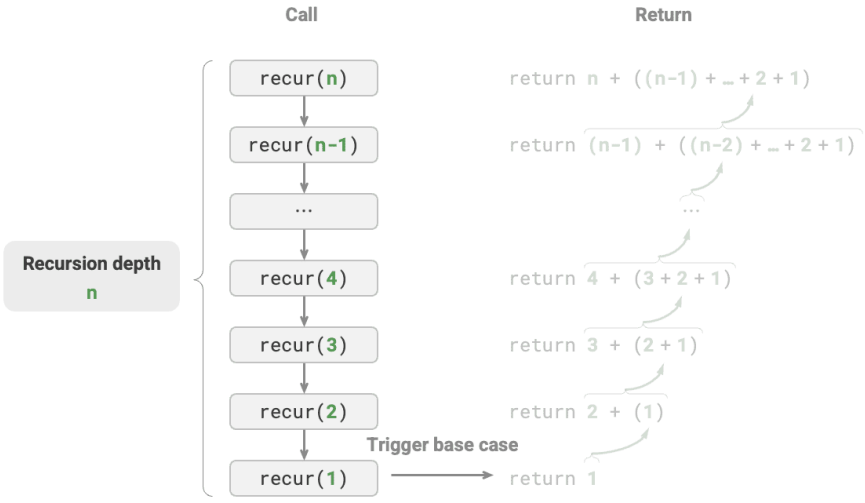


Figure 2-4 Recursion call depth

In practice, the recursion depth allowed by programming languages is usually limited, and excessively deep recursion may lead to stack overflow errors.

2. Tail Recursion

Interestingly, **if a function makes the recursive call as the very last step before returning**, the function can be optimized by the compiler or interpreter to have space efficiency comparable to iteration. This case is called tail recursion.

- **Regular recursion**: When a function returns to the previous level, it needs to continue executing code, so the system needs to save the context of the previous layer's call.
- **Tail recursion**: The recursive call is the last operation before the function returns, meaning that after returning to the previous level, there is no need to continue executing other operations, so the system does not need to save the context of the previous layer's function.

Taking the calculation of $1 + 2 + \cdots + n$ as an example, we can set the result variable `res` as a function parameter to implement tail recursion:

```
// ═══ File: recursion.js ═══

/* Tail recursion */
function tailRecur(n, res) {
    // Termination condition
    if (n === 0) return res;
    // Tail recursive call
    return tailRecur(n - 1, res + n);
}
```

The execution process of tail recursion is shown in Figure 2-5. Comparing regular recursion and tail recursion, the execution point of the summation operation is different.

- **Regular recursion**: The summation operation is performed during the "ascending" process, requiring an additional summation operation after each layer returns.
- **Tail recursion**: The summation operation is performed during the "descending" process; the "ascending" process only needs to return layer by layer.

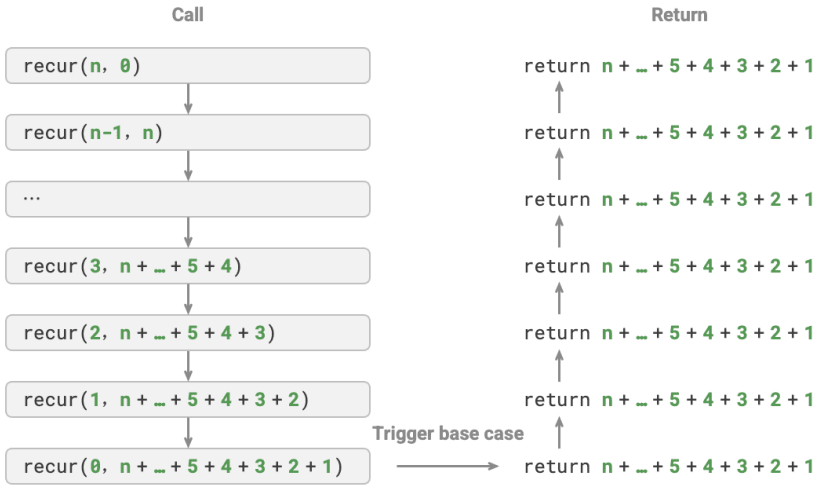


Figure 2-5 Tail recursion process

> **Tip**
> Please note that many compilers or interpreters do not support tail recursion optimization. For example, Python does not support tail recursion optimization by default, so even if a function is in tail recursive form, it may still encounter stack overflow issues.

3. Recursion Tree

When dealing with algorithmic problems related to "divide and conquer", recursion often provides a more intuitive approach and more readable code than iteration. Taking the "Fibonacci sequence" as an example.

> **Question**
> Given a Fibonacci sequence $0, 1, 1, 2, 3, 5, 8, 13, \ldots$, find the n-th number in the sequence.

Let the n-th number of the Fibonacci sequence be $f(n)$. Two conclusions can be easily obtained.

- The first two numbers of the sequence are $f(1) = 0$ and $f(2) = 1$.
- Each number in the sequence is the sum of the previous two numbers, i.e., $f(n) = f(n-1) + f(n-2)$.

Following the recurrence relation to make recursive calls, with the first two numbers as termination conditions, we can write the recursive code. Calling `fib(n)` will give us the n-th number of the Fibonacci sequence:

```
// ═══ File: recursion.js ═══

/* Fibonacci sequence: recursion */
function fib(n) {
    // Termination condition f(1) = 0, f(2) = 1
    if (n === 1 || n === 2) return n - 1;
    // Recursive call f(n) = f(n-1) + f(n-2)
    const res = fib(n - 1) + fib(n - 2);
    // Return result f(n)
    return res;
}
```

Observing the above code, we recursively call two functions within the function, **meaning that one call produces two call branches**. As shown in Figure 2-6, such continuous recursive calling will eventually produce a recursion tree with n levels.

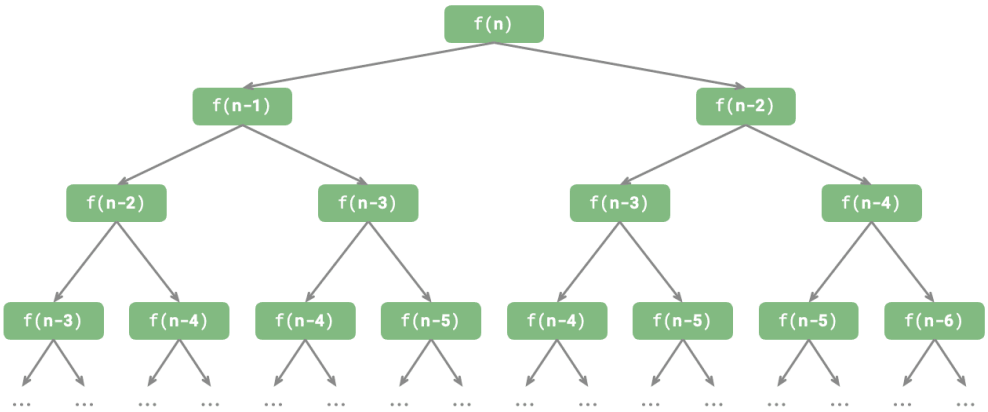


Figure 2-6 Recursion tree of the Fibonacci sequence

Fundamentally, recursion embodies the paradigm of "decomposing a problem into smaller subproblems", and this divide-and-conquer strategy is crucial.

- From an algorithmic perspective, many important algorithmic strategies such as searching, sorting, backtracking, divide and conquer, and dynamic programming directly or indirectly apply this way of thinking.
- From a data structure perspective, recursion is naturally suited for handling problems related to linked lists, trees, and graphs, because they are well-suited for analysis using divide-and-conquer thinking.

2.2.3 Comparison of the Two

Summarizing the above content, as shown in Table 2-1, iteration and recursion differ in implementation, performance, and applicability.

Table 2-1 Comparison of iteration and recursion characteristics

	Iteration	Recursion
Implementation	Loop structure	Function calls itself
Time efficiency	Generally more efficient, no function call overhead	Each function call incurs overhead
Memory usage	Usually uses a fixed amount of memory space	Accumulated function calls may use a large amount of stack frame space
Suitable problems	Suitable for simple loop tasks, with intuitive and readable code	Suitable for subproblem decomposition, such as trees, graphs, divide and conquer, backtracking, etc., with concise and clear code structure

> **Tip**
> If you find the following content difficult to understand, you can review it after reading the "Stack" chapter.

What is the intrinsic relationship between iteration and recursion? Taking the above recursive function as an example, the summation operation is performed during the "ascending" phase of recursion. This means that the function called first actually completes its summation operation last, **and this working mechanism is similar to the "last-in, first-out" principle of stacks**.

In fact, recursive terminology such as "call stack" and "stack frame space" already hints at the close relationship between recursion and stacks.

1. **Descend**: When a function is called, the system allocates a new stack frame on the "call stack" for that function to store the function's local variables, parameters, return address, and other data.
2. **Ascend**: When the function completes execution and returns, the corresponding stack frame is removed from the "call stack", restoring the execution environment of the previous function.

Therefore, **we can use an explicit stack to simulate the behavior of the call stack**, thus transforming recursion into iterative form:

```
// ═══ File: recursion.js ═══

/* Simulate recursion using iteration */
function forLoopRecur(n) {
    // Use an explicit stack to simulate the system call stack
    const stack = [];
    let res = 0;
    // Recurse: recursive call
    for (let i = n; i > 0; i--) {
        // Simulate "recurse" with "push"
        stack.push(i);
    }
    // Return: return result
    while (stack.length) {
        // Simulate "return" with "pop"
        res += stack.pop();
    }
    // res = 1+2+3+...+n
    return res;
}
```

Observing the above code, when recursion is transformed into iteration, the code becomes more complex. Although iteration and recursion can be converted into each other in many cases, it may not be worthwhile to do so for the following two reasons.

- The transformed code may be more difficult to understand and less readable.
- For some complex problems, simulating the behavior of the system call stack can be very difficult.

In summary, **choosing between iteration and recursion depends on the nature of the specific problem**. In programming practice, it is crucial to weigh the pros and cons of both and choose the appropriate method based on the context.

2.3 Time Complexity

Runtime can intuitively and accurately reflect the efficiency of an algorithm. If we want to accurately estimate the runtime of a piece of code, how should we proceed?

1. **Determine the running platform**, including hardware configuration, programming language, system environment, etc., as these factors all affect code execution efficiency.
2. **Evaluate the runtime required for various computational operations**, for example, an addition operation `+` requires 1 ns, a multiplication operation `*` requires 10 ns, a print operation `print()` requires 5 ns, etc.
3. **Count all computational operations in the code**, and sum the execution times of all operations to obtain the runtime.

For example, in the following code, the input data size is n:

```
// On a certain running platform
function algorithm(n) {
    var a = 2; // 1 ns
    a = a + 1; // 1 ns
    a = a * 2; // 10 ns
    // Loop n times
    for(let i = 0; i < n; i++) { // 1 ns
        console.log(0); // 5 ns
    }
}
```

According to the above method, the algorithm's runtime can be obtained as $(6n + 12)$ ns:

$$1 + 1 + 10 + (1 + 5) \times n = 6n + 12$$

In reality, however, **counting an algorithm's runtime is neither reasonable nor realistic**. First, we do not want to tie the estimated time to the running platform, because algorithms need to run on various different platforms. Second, it is difficult to know the runtime of each type of operation, which brings great difficulty to the estimation process.

2.3.1 Counting Time Growth Trends

Time complexity analysis does not count the algorithm's runtime, **but rather counts the growth trend of the algorithm's runtime as the data volume increases**.

The concept of "time growth trend" is rather abstract; let us understand it through an example. Suppose the input data size is n, and given three algorithms `A`, `B`, and `C`:

```
// Time complexity of algorithm A: constant order
function algorithm_A(n) {
    console.log(0);
}
// Time complexity of algorithm B: linear order
function algorithm_B(n) {
```

```
    for (let i = 0; i < n; i++) {
        console.log(0);
    }
}
// Time complexity of algorithm C: constant order
function algorithm_C(n) {
    for (let i = 0; i < 1000000; i++) {
        console.log(0);
    }
}
```

Figure 2-7 shows the time complexity of the above three algorithm functions.

- Algorithm `A` has only 1 print operation, and the algorithm's runtime does not grow as n increases. We call the time complexity of this algorithm "constant order".
- In algorithm `B`, the print operation needs to loop n times, and the algorithm's runtime grows linearly as n increases. The time complexity of this algorithm is called "linear order".
- In algorithm `C`, the print operation needs to loop 1000000 times. Although the runtime is very long, it is independent of the input data size n. Therefore, the time complexity of `C` is the same as `A`, still "constant order".

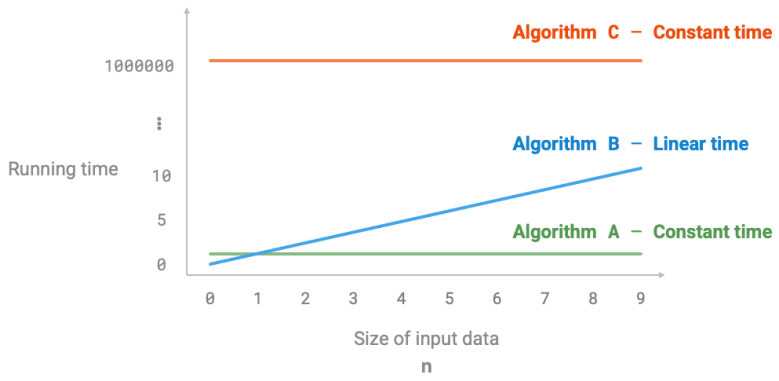


Figure 2-7 Time growth trends of algorithms A, B, and C

Compared to directly counting the algorithm's runtime, what are the characteristics of time complexity analysis?

- **Time complexity can effectively evaluate algorithm efficiency**. For example, the runtime of algorithm `B` grows linearly; when $n > 1$ it is slower than algorithm `A`, and when $n > 1000000$ it is slower than algorithm `C`. In fact, as long as the input data size n is sufficiently large, an algorithm with "constant order" complexity will always be superior to one with "linear order" complexity, which is precisely the meaning of time growth trend.
- **The derivation method for time complexity is simpler**. Obviously, the running platform and the types of computational operations are both unrelated to the growth trend of the algorithm's runtime. Therefore, in time complexity analysis, we can simply treat the execution time of all computational operations as the same "unit time", thus simplifying "counting computational operation

runtime" to "counting the number of computational operations", which greatly reduces the difficulty of estimation.

- **Time complexity also has certain limitations**. For example, although algorithms `A` and `C` have the same time complexity, their actual runtimes differ significantly. Similarly, although algorithm `B` has a higher time complexity than `C`, when the input data size n is small, algorithm `B` is clearly superior to algorithm `C`. In such cases, it is often difficult to judge the efficiency of algorithms based solely on time complexity. Of course, despite the above issues, complexity analysis remains the most effective and commonly used method for evaluating algorithm efficiency.

2.3.2 Asymptotic Upper Bound of Functions

Given a function with input size n:

```
function algorithm(n) {
    var a = 1; // +1
    a += 1; // +1
    a *= 2; // +1
    // Loop n times
    for(let i = 0; i < n; i++){ // +1 (i++ is executed each round)
        console.log(0); // +1
    }
}
```

Let the number of operations of the algorithm be a function of the input data size n, denoted as $T(n)$. Then the number of operations of the above function is:

$$T(n) = 3 + 2n$$

$T(n)$ is a linear function, indicating that its runtime growth trend is linear, and therefore its time complexity is linear order.

We denote the time complexity of linear order as $O(n)$. This mathematical symbol is called big-O notation, representing the asymptotic upper bound of the function $T(n)$.

Time complexity analysis essentially calculates the asymptotic upper bound of "the number of operations $T(n)$", which has a clear mathematical definition.

> **Asymptotic upper bound of functions**
> If there exist positive real numbers c and n_0 such that for all $n > n_0$, we have $T(n) \leq c \cdot f(n)$, then $f(n)$ can be considered as an asymptotic upper bound of $T(n)$, denoted as $T(n) = O(f(n))$.

As shown in Figure 2-8, calculating the asymptotic upper bound is to find a function $f(n)$ such that when n tends to infinity, $T(n)$ and $f(n)$ are at the same growth level, differing only by a constant coefficient c.

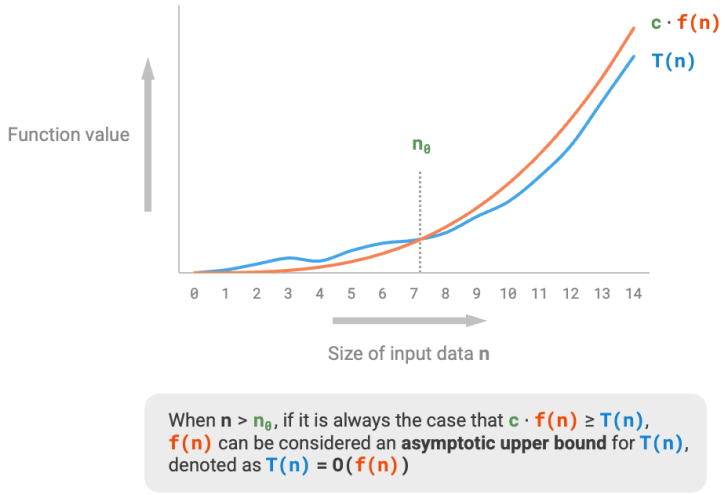


Figure 2-8 Asymptotic upper bound of a function

2.3.3 Derivation Method

The asymptotic upper bound has a bit of mathematical flavor. If you feel you haven't fully understood it, don't worry. We can first master the derivation method, and gradually grasp its mathematical meaning through continuous practice.

According to the definition, after determining $f(n)$, we can obtain the time complexity $O(f(n))$. So how do we determine the asymptotic upper bound $f(n)$? Overall, it is divided into two steps: first count the number of operations, then determine the asymptotic upper bound.

1. Step 1: Count the Number of Operations

For code, count from top to bottom line by line. However, since the constant coefficient c in $c \cdot f(n)$ above can be of any size, **coefficients and constant terms in the number of operations $T(n)$ can all be ignored**. According to this principle, the following counting simplification techniques can be summarized.

1. **Ignore constants in $T(n)$**. Because they are all independent of n, they do not affect time complexity.
2. **Omit all coefficients**. For example, looping $2n$ times, $5n + 1$ times, etc., can all be simplified as n times, because the coefficient before n does not affect time complexity.
3. **Use multiplication for nested loops**. The total number of operations equals the product of the number of operations in the outer and inner loops, with each layer of loop still able to apply techniques `1.` and `2.` separately.

Given a function, we can use the above techniques to count the number of operations:

```
function algorithm(n) {
    let a = 1;  // +0 (Technique 1)
    a = a + n;  // +0 (Technique 1)
    // +n (Technique 2)
    for (let i = 0; i < 5 * n + 1; i++) {
        console.log(0);
    }
    // +n*n (Technique 3)
    for (let i = 0; i < 2 * n; i++) {
        for (let j = 0; j < n + 1; j++) {
            console.log(0);
        }
    }
}
```

The following formula shows the counting results before and after using the above techniques; both derive a time complexity of $O(n^2)$.

$$
\begin{aligned}
T(n) &= 2n(n+1) + (5n+1) + 2 \quad \text{Complete count (-.-|||)} \\
&= 2n^2 + 7n + 3 \\
T(n) &= n^2 + n \quad \text{Simplified count (o.O)}
\end{aligned}
$$

2. Step 2: Determine the Asymptotic Upper Bound

Time complexity is determined by the highest-order term in $T(n)$. This is because as n tends to infinity, the highest-order term will play a dominant role, and the influence of other terms can be ignored.

Table 2-2 shows some examples, where some exaggerated values are used to emphasize the conclusion that "coefficients cannot shake the order". When n tends to infinity, these constants become insignificant.

Table 2-2 Time complexities corresponding to different numbers of operations

Number of Operations $T(n)$	Time Complexity $O(f(n))$
100000	$O(1)$
$3n + 2$	$O(n)$
$2n^2 + 3n + 2$	$O(n^2)$
$n^3 + 10000n^2$	$O(n^3)$
$2^n + 10000n^{10000}$	$O(2^n)$

2.3.4 Common Types

Let the input data size be n. Common time complexity types are shown in Figure 2-9 (arranged in order from low to high).

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) <$$

Constant order $<$ Logarithmic order $<$ Linear order $<$ Linearithmic order $<$ Quadratic order $<$ Exponential order

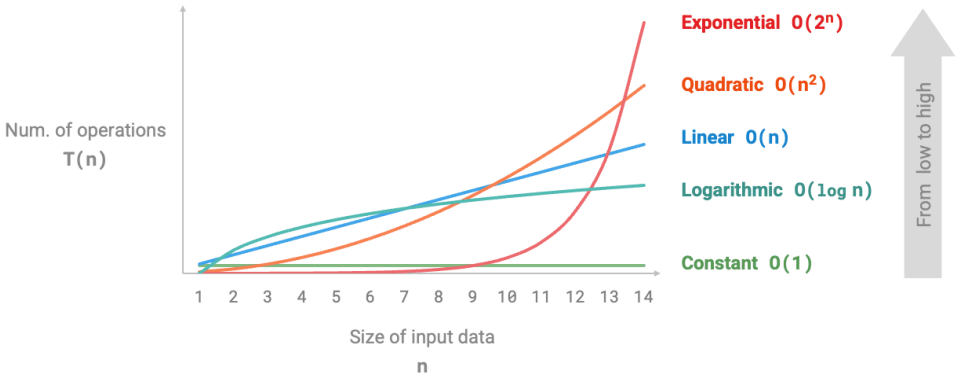


Figure 2-9 Common time complexity types

1. Constant Order $O(1)$

The number of operations in constant order is independent of the input data size n, meaning it does not change as n changes.

In the following function, although the number of operations `size` may be large, since it is independent of the input data size n, the time complexity remains $O(1)$:

```
// ═══ File: time_complexity.js ═══

/* Constant order */
function constant(n) {
    let count = 0;
    const size = 100000;
    for (let i = 0; i < size; i++) count++;
    return count;
}
```

2. Linear Order $O(n)$

The number of operations in linear order grows linearly relative to the input data size n. Linear order typically appears in single-layer loops:

```
// ═══ File: time_complexity.js ═══

/* Linear order */
function linear(n) {
    let count = 0;
```

```
    for (let i = 0; i < n; i++) count++;
    return count;
}
```

Operations such as traversing arrays and traversing linked lists have a time complexity of $O(n)$, where n is the length of the array or linked list:

```
// == File: time_complexity.js ==

/* Linear order (traversing array) */
function arrayTraversal(nums) {
    let count = 0;
    // Number of iterations is proportional to the array length
    for (let i = 0; i < nums.length; i++) {
        count++;
    }
    return count;
}
```

It is worth noting that **the input data size n should be determined according to the type of input data**. For example, in the first example, the variable n is the input data size; in the second example, the array length n is the data size.

3. Quadratic Order $O(n^2)$

The number of operations in quadratic order grows quadratically relative to the input data size n. Quadratic order typically appears in nested loops, where both the outer and inner loops have a time complexity of $O(n)$, resulting in an overall time complexity of $O(n^2)$:

```
// == File: time_complexity.js ==

/* Exponential order */
function quadratic(n) {
    let count = 0;
    // Number of iterations is quadratically related to the data size n
    for (let i = 0; i < n; i++) {
        for (let j = 0; j < n; j++) {
            count++;
        }
    }
    return count;
}
```

Figure 2-10 compares constant order, linear order, and quadratic order time complexities.

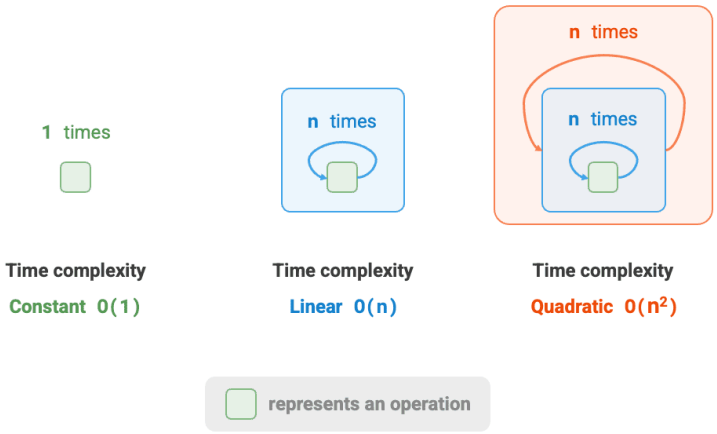


Figure 2-10 Time complexities of constant, linear, and quadratic orders

Taking bubble sort as an example, the outer loop executes $n-1$ times, and the inner loop executes $n-1$, $n-2$, ..., 2, 1 times, averaging $n/2$ times, resulting in a time complexity of $O((n-1)n/2) = O(n^2)$:

```
// ═══ File: time_complexity.js ═══

/* Quadratic order (bubble sort) */
function bubbleSort(nums) {
    let count = 0; // Counter
    // Outer loop: unsorted range is [0, i]
    for (let i = nums.length - 1; i > 0; i--) {
        // Inner loop: swap the largest element in the unsorted range [0, i] to the rightmost end
        ↪ of that range
        for (let j = 0; j < i; j++) {
            if (nums[j] > nums[j + 1]) {
                // Swap nums[j] and nums[j + 1]
                let tmp = nums[j];
                nums[j] = nums[j + 1];
                nums[j + 1] = tmp;
                count += 3; // Element swap includes 3 unit operations
            }
        }
    }
    return count;
}
```

4. Exponential Order $O(2^n)$

Biological "cell division" is a typical example of exponential order growth: the initial state is 1 cell, after one round of division it becomes 2, after two rounds it becomes 4, and so on; after n rounds of division there are 2^n cells.

Figure 2-11 and the following code simulate the cell division process, with a time complexity of $O(2^n)$. Note that the input n represents the number of division rounds, and the return value `count` represents the total number of divisions.

```
// ═══ File: time_complexity.js ═══

/* Exponential order (loop implementation) */
function exponential(n) {
    let count = 0,
        base = 1;
    // Cells divide into two every round, forming sequence 1, 2, 4, 8, ..., 2^(n-1)
    for (let i = 0; i < n; i++) {
        for (let j = 0; j < base; j++) {
            count++;
        }
        base *= 2;
    }
    // count = 1 + 2 + 4 + 8 + .. + 2^(n-1) = 2^n - 1
    return count;
}
```

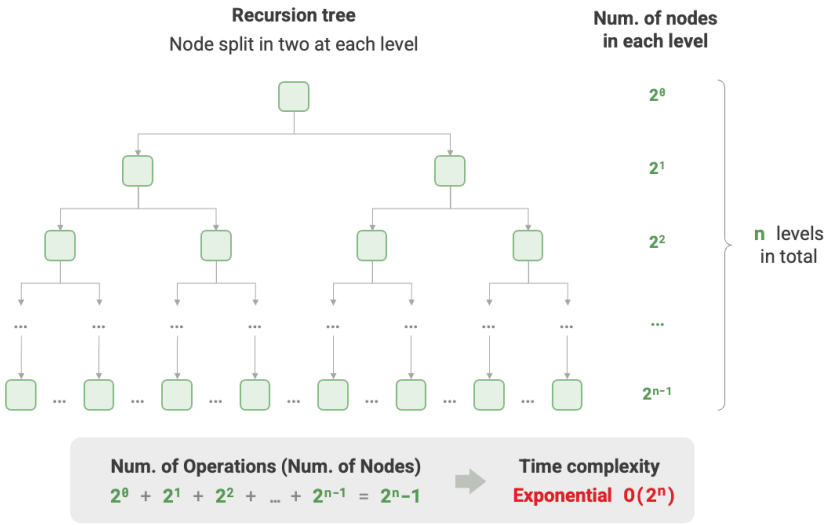


Figure 2-11 Time complexity of exponential order

In actual algorithms, exponential order often appears in recursive functions. For example, in the following code, it recursively splits in two, stopping after n splits:

```
// ═══ File: time_complexity.js ═══

/* Exponential order (recursive implementation) */
function expRecur(n) {
    if (n === 1) return 1;
    return expRecur(n - 1) + expRecur(n - 1) + 1;
}
```

Exponential order growth is very rapid and is common in exhaustive methods (brute force search, backtracking, etc.). For problems with large data scales, exponential order is unacceptable and typically requires dynamic programming or greedy algorithms to solve.

5. Logarithmic Order $O(\log n)$

In contrast to exponential order, logarithmic order reflects the situation of "reducing to half each round". Let the input data size be n. Since it is reduced to half each round, the number of loops is $\log_2 n$, which is the inverse function of 2^n.

Figure 2-12 and the following code simulate the process of "reducing to half each round", with a time complexity of $O(\log_2 n)$, abbreviated as $O(\log n)$:

```
// == File: time_complexity.js ==

/* Logarithmic order (loop implementation) */
function logarithmic(n) {
    let count = 0;
    while (n > 1) {
        n = n / 2;
        count++;
    }
    return count;
}
```

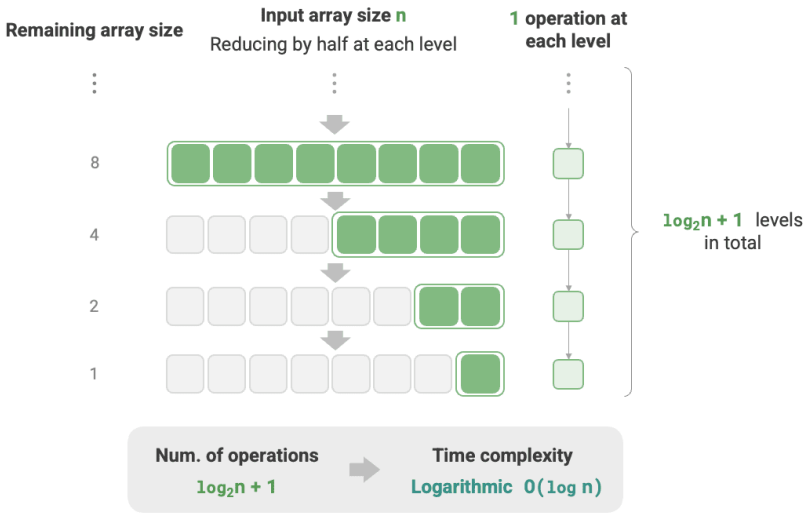


Figure 2-12 Time complexity of logarithmic order

Like exponential order, logarithmic order also commonly appears in recursive functions. The following code forms a recursion tree of height $\log_2 n$:

```
// == File: time_complexity.js ==

/* Logarithmic order (recursive implementation) */
function logRecur(n) {
    if (n <= 1) return 0;
    return logRecur(n / 2) + 1;
}
```

Logarithmic order commonly appears in algorithms based on the divide-and-conquer strategy, embodying the algorithmic thinking of "dividing into many" and "simplifying complexity". It grows slowly and is the ideal time complexity second only to constant order.

> **What is the base of $O(\log n)$?**
> To be precise, "dividing into m" corresponds to a time complexity of $O(\log_m n)$. And through the logarithmic base change formula, we can obtain time complexities with different bases that are equal:
>
> $$O(\log_m n) = O(\log_k n / \log_k m) = O(\log_k n)$$
>
> That is to say, the base m can be converted without affecting the complexity. Therefore, we usually omit the base m and denote logarithmic order simply as $O(\log n)$.

6. Linearithmic Order $O(n \log n)$

Linearithmic order commonly appears in nested loops, where the time complexities of the two layers of loops are $O(\log n)$ and $O(n)$ respectively. The relevant code is as follows:

```
// ═══ File: time_complexity.js ═══

/* Linearithmic order */
function linearLogRecur(n) {
    if (n <= 1) return 1;
    let count = linearLogRecur(n / 2) + linearLogRecur(n / 2);
    for (let i = 0; i < n; i++) {
        count++;
    }
    return count;
}
```

Figure 2-13 shows how linearithmic order is generated. Each level of the binary tree has a total of n operations, and the tree has $\log_2 n + 1$ levels, resulting in a time complexity of $O(n \log n)$.

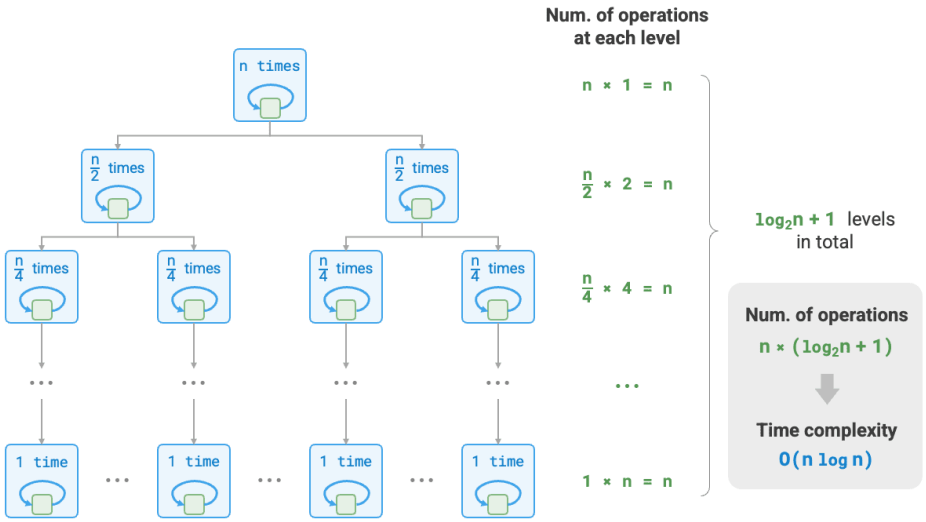


Figure 2-13 Time complexity of linearithmic order

Mainstream sorting algorithms typically have a time complexity of $O(n \log n)$, such as quicksort, merge sort, and heap sort.

7. Factorial Order $O(n!)$

Factorial order corresponds to the mathematical "permutation" problem. Given n distinct elements, find all possible permutation schemes; the number of schemes is:

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

Factorials are typically implemented using recursion. As shown in Figure 2-14 and the following code, the first level splits into n branches, the second level splits into $n-1$ branches, and so on, until the n-th level when splitting stops:

```
// ═══ File: time_complexity.js ═══

/* Factorial order (recursive implementation) */
function factorialRecur(n) {
    if (n ═══ 0) return 1;
    let count = 0;
    // Split from 1 into n
    for (let i = 0; i < n; i++) {
        count += factorialRecur(n - 1);
    }
    return count;
}
```

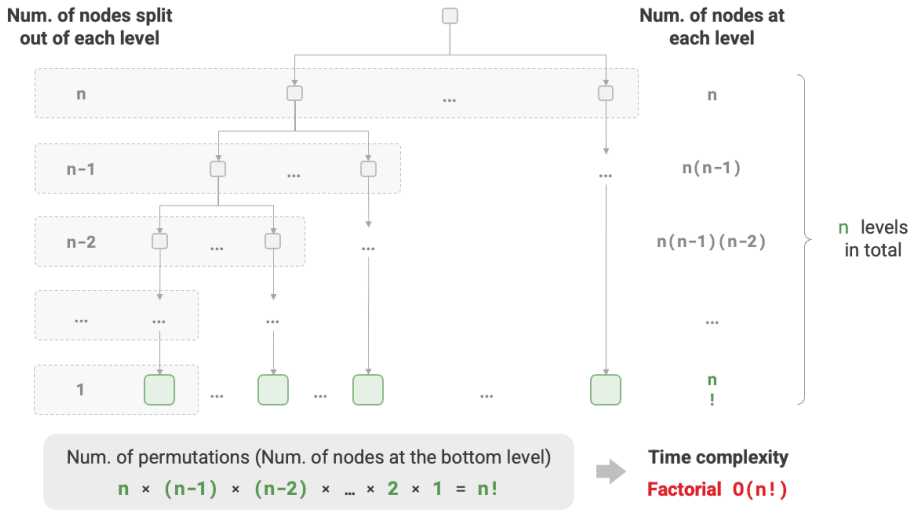


Figure 2-14 Time complexity of factorial order

Note that because when $n \geq 4$ we always have $n! > 2^n$, factorial order grows faster than exponential order, and is also unacceptable for large n.

2.3.5 Worst, Best, and Average Time Complexities

The time efficiency of an algorithm is often not fixed, but is related to the distribution of the input data. Suppose we input an array `nums` of length n, where `nums` consists of numbers from 1 to n, with each number appearing only once, but the element order is randomly shuffled. The task is to return the index of element 1. We can draw the following conclusions.

- When `nums = [?, ?, ..., 1]`, i.e., when the last element is 1, it requires a complete traversal of the array, **reaching worst-case time complexity** $O(n)$.
- When `nums = [1, ?, ?, ...]`, i.e., when the first element is 1, no matter how long the array is, there is no need to continue traversing, **reaching best-case time complexity** $\Omega(1)$.

The "worst-case time complexity" corresponds to the function's asymptotic upper bound, denoted using big-O notation. Correspondingly, the "best-case time complexity" corresponds to the function's asymptotic lower bound, denoted using Ω notation:

```
// ═══ File: worst_best_time_complexity.js ═══

/* Generate an array with elements { 1, 2, ..., n }, order shuffled */
function randomNumbers(n) {
    const nums = Array(n);
    // Generate array nums = { 1, 2, 3, ..., n }
    for (let i = 0; i < n; i++) {
        nums[i] = i + 1;
    }
    // Randomly shuffle array elements
    for (let i = 0; i < n; i++) {
```

```
        const r = Math.floor(Math.random() * (i + 1));
        const temp = nums[i];
        nums[i] = nums[r];
        nums[r] = temp;
    }
    return nums;
}

/* Find the index of number 1 in array nums */
function findOne(nums) {
    for (let i = 0; i < nums.length; i++) {
        // When element 1 is at the head of the array, best time complexity O(1) is achieved
        // When element 1 is at the tail of the array, worst time complexity O(n) is achieved
        if (nums[i] === 1) {
            return i;
        }
    }
    return -1;
}
```

It is worth noting that we rarely use best-case time complexity in practice, because it can usually only be achieved with a very small probability and may be somewhat misleading. **The worst-case time complexity is more practical because it gives a safety value for efficiency**, allowing us to use the algorithm with confidence.

From the above example, we can see that both worst-case and best-case time complexities only occur under "special data distributions", which may have a very small probability of occurrence and may not truly reflect the algorithm's running efficiency. In contrast, **average time complexity can reflect the algorithm's running efficiency under random input data**, denoted using the Θ notation.

For some algorithms, we can simply derive the average case under random data distribution. For example, in the above example, since the input array is shuffled, the probability of element 1 appearing at any index is equal, so the algorithm's average number of loops is half the array length $n/2$, giving an average time complexity of $\Theta(n/2) = \Theta(n)$.

But for more complex algorithms, calculating average time complexity is often quite difficult, because it is hard to analyze the overall mathematical expectation under data distribution. In this case, we usually use worst-case time complexity as the criterion for judging algorithm efficiency.

> **Why is the Θ symbol rarely seen?**
> This may be because the O symbol is too catchy, so we often use it to represent average time complexity. But strictly speaking, this practice is not standard. In this book and other materials, if you encounter expressions like "average time complexity $O(n)$", please understand it directly as $\Theta(n)$.

2.4 Space Complexity

Space complexity measures the growth trend of memory space occupied by an algorithm as the data size increases. This concept is very similar to time complexity, except that "running time" is replaced

with “occupied memory space”.

2.4.1 Algorithm-Related Space

The memory space used by an algorithm during execution mainly includes the following types.

- **Input space**: Used to store the input data of the algorithm.
- **Temporary space**: Used to store variables, objects, function contexts, and other data during the algorithm's execution.
- **Output space**: Used to store the output data of the algorithm.

In general, the scope of space complexity statistics is “temporary space” plus “output space”.

Temporary space can be further divided into three parts.

- **Temporary data**: Used to save various constants, variables, objects, etc., during the algorithm's execution.
- **Stack frame space**: Used to save the context data of called functions. The system creates a stack frame at the top of the stack each time a function is called, and the stack frame space is released after the function returns.
- **Instruction space**: Used to save compiled program instructions, which are usually ignored in actual statistics.

When analyzing the space complexity of a program, **we usually count three parts: temporary data, stack frame space, and output data**, as shown in the following figure.

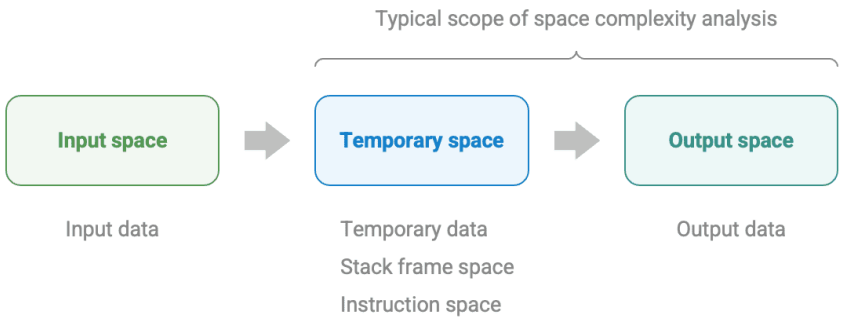


Figure 2-15 Algorithm-related space

The related code is as follows:

```
/* Class */
class Node {
    val;
    next;
    constructor(val) {
        this.val = val === undefined ? 0 : val; // Node value
        this.next = null;                       // Reference to the next node
```

```
    }
}

/* Function */
function constFunc() {
    // Perform some operations
    return 0;
}

function algorithm(n) {       // Input data
    const a = 0;              // Temporary data (constant)
    let b = 0;                // Temporary data (variable)
    const node = new Node(0); // Temporary data (object)
    const c = constFunc();    // Stack frame space (function call)
    return a + b + c;         // Output data
}
```

2.4.2 Calculation Method

The calculation method for space complexity is roughly the same as for time complexity, except that the statistical object is changed from "number of operations" to "size of space used".

Unlike time complexity, **we usually only focus on the worst-case space complexity**. This is because memory space is a hard requirement, and we must ensure that sufficient memory space is reserved for all input data.

Observe the following code. The "worst case" in worst-case space complexity has two meanings.

1. **Based on the worst input data**: When $n < 10$, the space complexity is $O(1)$; but when $n > 10$, the initialized array `nums` occupies $O(n)$ space, so the worst-case space complexity is $O(n)$.
2. **Based on the peak memory during algorithm execution**: For example, before executing the last line, the program occupies $O(1)$ space; when initializing the array `nums`, the program occupies $O(n)$ space, so the worst-case space complexity is $O(n)$.

```
function algorithm(n) {
    const a = 0;                    // O(1)
    const b = new Array(10000);     // O(1)
    if (n > 10) {
        const nums = new Array(n); // O(n)
    }
}
```

In recursive functions, it is necessary to count the stack frame space. Observe the following code:

```
function constFunc() {
    // Perform some operations
    return 0;
}
/* Loop has space complexity of O(1) */
function loop(n) {
    for (let i = 0; i < n; i++) {
        constFunc();
    }
```

```
}
/* Recursion has space complexity of O(n) */
function recur(n) {
    if (n === 1) return;
    return recur(n - 1);
}
```

The time complexity of both functions `loop()` and `recur()` is $O(n)$, but their space complexities are different.

- The function `loop()` calls `function()` n times in a loop. In each iteration, `function()` returns and releases its stack frame space, so the space complexity remains $O(1)$.
- The recursive function `recur()` has n unreturned `recur()` instances existing simultaneously during execution, thus occupying $O(n)$ stack frame space.

2.4.3 Common Types

Let the input data size be n. The following figure shows common types of space complexity (arranged from low to high).

$$O(1) < O(\log n) < O(n) < O(n^2) < O(2^n)$$

$$\text{Constant} < \text{Logarithmic} < \text{Linear} < \text{Quadratic} < \text{Exponential}$$

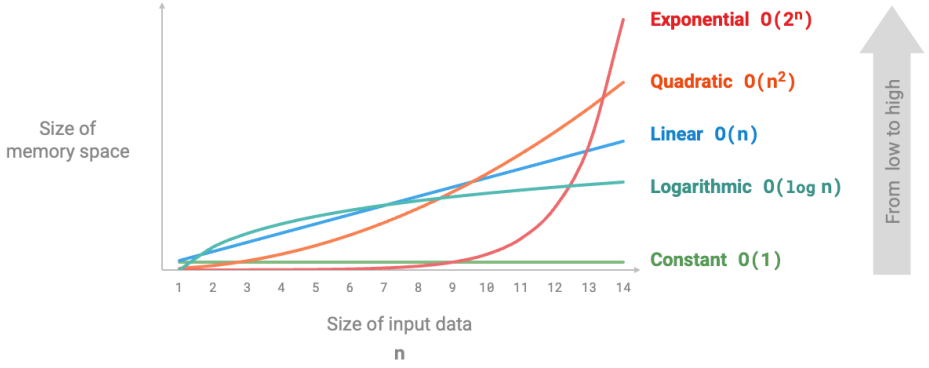


Figure 2-16 Common types of space complexity

1. Constant Order $O(1)$

Constant order is common in constants, variables, and objects whose quantity is independent of the input data size n.

It should be noted that memory occupied by initializing variables or calling functions in a loop is released when entering the next iteration, so it does not accumulate space, and the space complexity remains $O(1)$:

```
// ═══ File: space_complexity.js ═══

/* Function */
function constFunc() {
    // Perform some operations
    return 0;
}

/* Constant order */
function constant(n) {
    // Constants, variables, objects occupy O(1) space
    const a = 0;
    const b = 0;
    const nums = new Array(10000);
    const node = new ListNode(0);
    // Variables in the loop occupy O(1) space
    for (let i = 0; i < n; i++) {
        const c = 0;
    }
    // Functions in the loop occupy O(1) space
    for (let i = 0; i < n; i++) {
        constFunc();
    }
}
```

2. Linear Order $O(n)$

Linear order is common in arrays, linked lists, stacks, queues, etc., where the number of elements is proportional to n:

```
// ═══ File: space_complexity.js ═══

/* Linear order */
function linear(n) {
    // Array of length n uses O(n) space
    const nums = new Array(n);
    // A list of length n occupies O(n) space
    const nodes = [];
    for (let i = 0; i < n; i++) {
        nodes.push(new ListNode(i));
    }
    // A hash table of length n occupies O(n) space
    const map = new Map();
    for (let i = 0; i < n; i++) {
        map.set(i, i.toString());
    }
}
```

As shown in the following figure, the recursion depth of this function is n, meaning that there are n unreturned `linear_recur()` functions existing simultaneously, using $O(n)$ stack frame space:

```
// ═══ File: space_complexity.js ═══

/* Linear order (recursive implementation) */
function linearRecur(n) {
```

```
    console.log(`Recursion n = ${n}`);
    if (n === 1) return;
    linearRecur(n - 1);
}
```

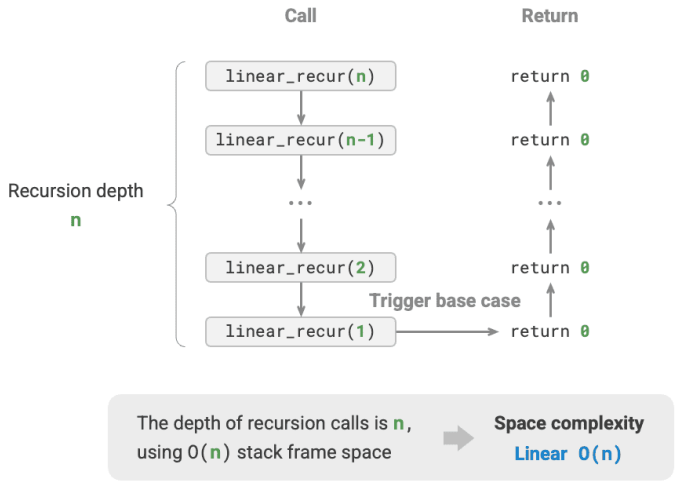


Figure 2-17 Linear order space complexity generated by recursive function

3. Quadratic Order $O(n^2)$

Quadratic order is common in matrices and graphs, where the number of elements is quadratically related to n:

```
// === File: space_complexity.js ===

/* Exponential order */
function quadratic(n) {
    // Matrix uses O(n^2) space
    const numMatrix = Array(n)
        .fill(null)
        .map(() => Array(n).fill(null));
    // 2D list uses O(n^2) space
    const numList = [];
    for (let i = 0; i < n; i++) {
        const tmp = [];
        for (let j = 0; j < n; j++) {
            tmp.push(0);
        }
        numList.push(tmp);
    }
}
```

As shown in the following figure, the recursion depth of this function is n, and an array is initialized in each recursive function with lengths of $n, n-1, \ldots, 2, 1$, with an average length of $n/2$, thus occupying $O(n^2)$ space overall:

```
// ═══ File: space_complexity.js ═══

/* Quadratic order (recursive implementation) */
function quadraticRecur(n) {
    if (n <= 0) return 0;
    const nums = new Array(n);
    console.log(`In recursion n = ${n}, nums length = ${nums.length}`);
    return quadraticRecur(n - 1);
}
```

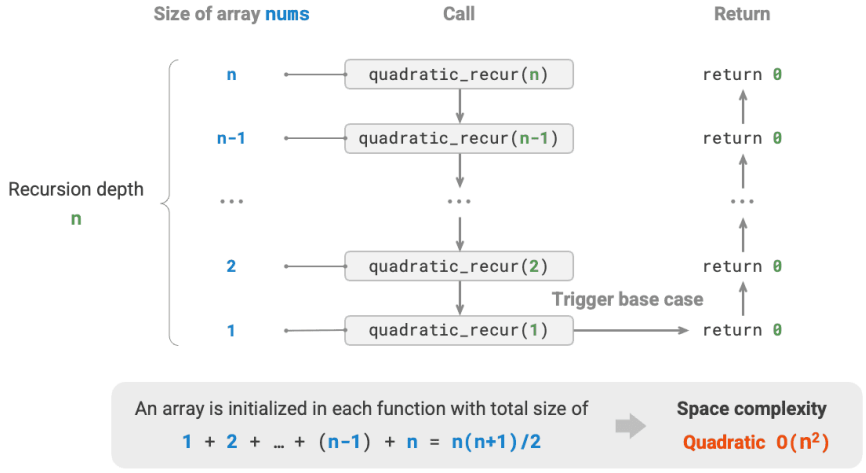


Figure 2-18 Quadratic order space complexity generated by recursive function

4. Exponential Order $O(2^n)$

Exponential order is common in binary trees. Observe the following figure: a "full binary tree" with n levels has $2^n - 1$ nodes, occupying $O(2^n)$ space:

```
// ═══ File: space_complexity.js ═══

/* Driver Code */
function buildTree(n) {
    if (n === 0) return null;
    const root = new TreeNode(0);
    root.left = buildTree(n - 1);
    root.right = buildTree(n - 1);
    return root;
}
```

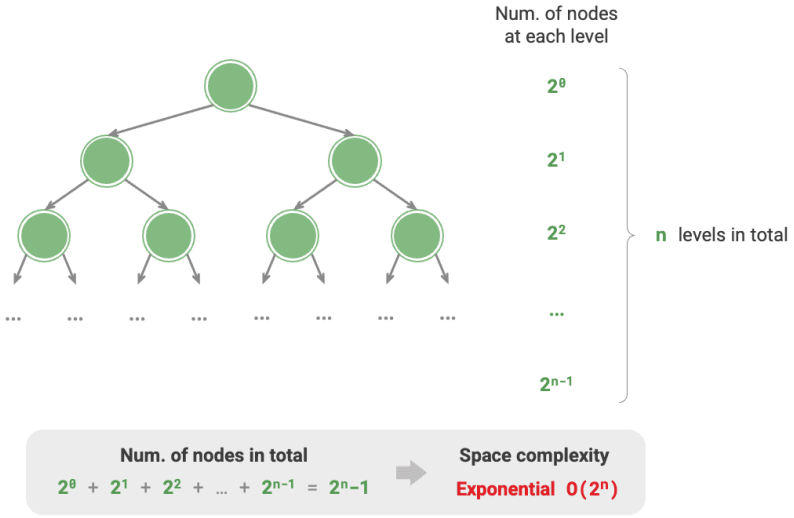


Figure 2-19 Exponential order space complexity generated by full binary tree

5. Logarithmic Order $O(\log n)$

Logarithmic order is common in divide-and-conquer algorithms. For example, merge sort: given an input array of length n, each recursion divides the array in half from the midpoint, forming a recursion tree of height $\log n$, using $O(\log n)$ stack frame space.

Another example is converting a number to a string. Given a positive integer n, it has $\lfloor \log_{10} n \rfloor + 1$ digits, i.e., the corresponding string length is $\lfloor \log_{10} n \rfloor + 1$, so the space complexity is $O(\log_{10} n + 1) = O(\log n)$.

2.4.4 Trading Time for Space

Ideally, we hope that both the time complexity and space complexity of an algorithm can reach optimal. However, in practice, optimizing both time complexity and space complexity simultaneously is usually very difficult.

Reducing time complexity usually comes at the cost of increasing space complexity, and vice versa. The approach of sacrificing memory space to improve algorithm execution speed is called "trading space for time"; conversely, it is called "trading time for space".

The choice of which approach depends on which aspect we value more. In most cases, time is more precious than space, so "trading space for time" is usually the more common strategy. Of course, when the data volume is very large, controlling space complexity is also very important.

2.5 Summary

1. Key Review

Algorithm Efficiency Assessment

- Time efficiency and space efficiency are the two primary evaluation metrics for measuring algorithm performance.
- We can evaluate algorithm efficiency through actual testing, but it is difficult to eliminate the influence of the testing environment, and it consumes substantial computational resources.
- Complexity analysis can eliminate the drawbacks of actual testing, with results applicable to all running platforms, and it can reveal algorithm efficiency under different data scales.

Time Complexity

- Time complexity is used to measure the trend of algorithm runtime as data volume increases. It can effectively evaluate algorithm efficiency, but may fail in certain situations, such as when the input data volume is small or when time complexities are identical, making it impossible to precisely compare algorithm efficiency.
- Worst-case time complexity is represented using Big O notation, corresponding to the asymptotic upper bound of a function, reflecting the growth level of the number of operations $T(n)$ as n approaches positive infinity.
- Deriving time complexity involves two steps: first, counting the number of operations, then determining the asymptotic upper bound.
- Common time complexities arranged from low to high include $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(2^n)$, and $O(n!)$.
- The time complexity of some algorithms is not fixed, but rather depends on the distribution of input data. Time complexity is divided into worst-case, best-case, and average-case time complexity. Best-case time complexity is rarely used because input data generally needs to satisfy strict conditions to achieve the best case.
- Average time complexity reflects the algorithm's runtime efficiency under random data input, and is closest to the algorithm's performance in practical applications. Calculating average time complexity requires statistical analysis of input data distribution and the combined mathematical expectation.

Space Complexity

- Space complexity serves a similar purpose to time complexity, used to measure the trend of algorithm memory usage as data volume increases.
- The memory space related to algorithm execution can be divided into input space, temporary space, and output space. Typically, input space is not included in space complexity calculations. Temporary space can be divided into temporary data, stack frame space, and instruction space, where stack frame space usually affects space complexity only in recursive functions.
- We typically only focus on worst-case space complexity, which is the space complexity of an algorithm under worst-case input data and worst-case runtime.
- Common space complexities arranged from low to high include $O(1)$, $O(\log n)$, $O(n)$, $O(n^2)$, and $O(2^n)$.

2. Q & A

Q: Is the space complexity of tail recursion $O(1)$?

Theoretically, the space complexity of tail recursive functions can be optimized to $O(1)$. However, most programming languages (such as Java, Python, C++, Go, C#, etc.) do not support automatic tail recursion optimization, so the space complexity is generally considered to be $O(n)$.

Q: What is the difference between the terms function and method?

A function can be executed independently, with all parameters passed explicitly. A method is associated with an object, is implicitly passed to the object that invokes it, and can operate on data contained in class instances.

The following examples use several common programming languages for illustration.

- C is a procedural programming language without object-oriented concepts, so it only has functions. However, we can simulate object-oriented programming by creating structures (struct), and functions associated with structures are equivalent to methods in other programming languages.
- Java and C# are object-oriented programming languages where code blocks (methods) are typically part of a class. Static methods behave like functions because they are bound to the class and cannot access specific instance variables.
- C++ and Python support both procedural programming (functions) and object-oriented programming (methods).

Q: Does the diagram for "common space complexity types" reflect the absolute size of occupied space?

No, the diagram shows space complexity, which reflects growth trends rather than the absolute size of occupied space.

Assuming $n = 8$, you might find that the values of each curve do not correspond to the functions. This is because each curve contains a constant term used to compress the value range into a visually comfortable range.

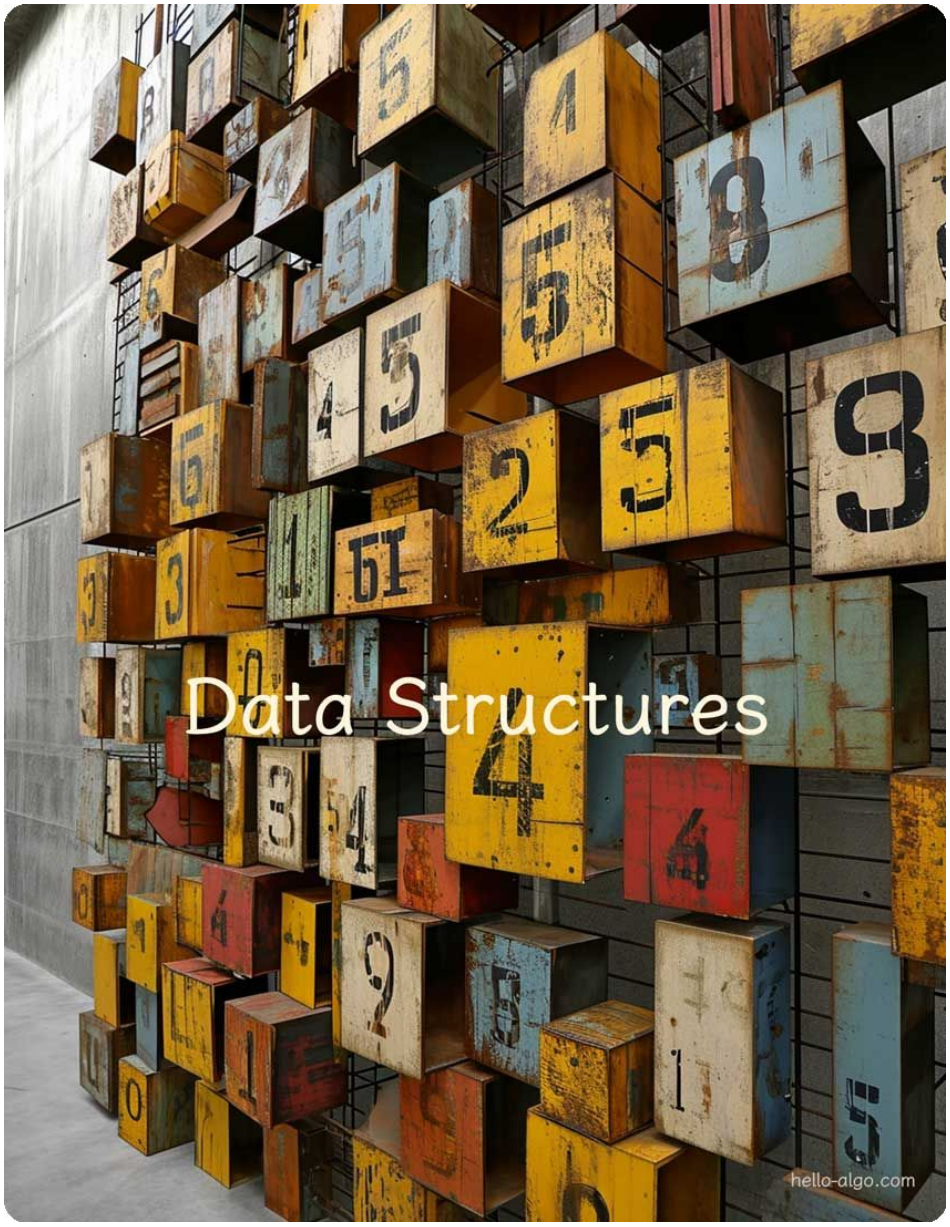
In practice, because we generally do not know what the "constant term" complexity of each method is, we usually cannot select the optimal solution for $n = 8$ based on complexity alone. But for $n = 8^5$, the choice is straightforward, as the growth trend already dominates.

Q: Are there situations where algorithms are designed to sacrifice time (or space) based on actual use cases?

In practical applications, most situations choose to sacrifice space for time. For example, with database indexes, we typically choose to build B+ trees or hash indexes, occupying substantial memory space in exchange for efficient queries of $O(\log n)$ or even $O(1)$.

In scenarios where space resources are precious, time may be sacrificed for space. For example, in embedded development, device memory is precious, and engineers may forgo using hash tables and choose to use array sequential search to save memory usage, at the cost of slower searches.

Chapter 3. Data Structures



Abstract

Data structure is like a sturdy and diverse framework.

It provides a blueprint for the orderly organization of data, upon which algorithms come to life.

3.1 Classification of Data Structures

Common data structures include arrays, linked lists, stacks, queues, hash tables, trees, heaps, and graphs. They can be classified from two dimensions: "logical structure" and "physical structure".

3.1.1 Logical Structure: Linear and Non-Linear

Logical structure reveals the logical relationships between data elements. In arrays and linked lists, data is arranged in a certain order, embodying the linear relationship between data; while in trees, data is arranged hierarchically from top to bottom, showing the derived relationship between "ancestors" and "descendants"; graphs are composed of nodes and edges, reflecting complex network relationships.

As shown in Figure 3-1, logical structures can be divided into two major categories: "linear" and "non-linear". Linear structures are more intuitive, indicating that data is linearly arranged in logical relationships; non-linear structures are the opposite, arranged non-linearly.

- **Linear data structures**: Arrays, linked lists, stacks, queues, hash tables, where elements have a one-to-one sequential relationship.
- **Non-linear data structures**: Trees, heaps, graphs, hash tables.

Non-linear data structures can be further divided into tree structures and network structures.

- **Tree structures**: Trees, heaps, hash tables, where elements have a one-to-many relationship.
- **Network structures**: Graphs, where elements have a many-to-many relationship.

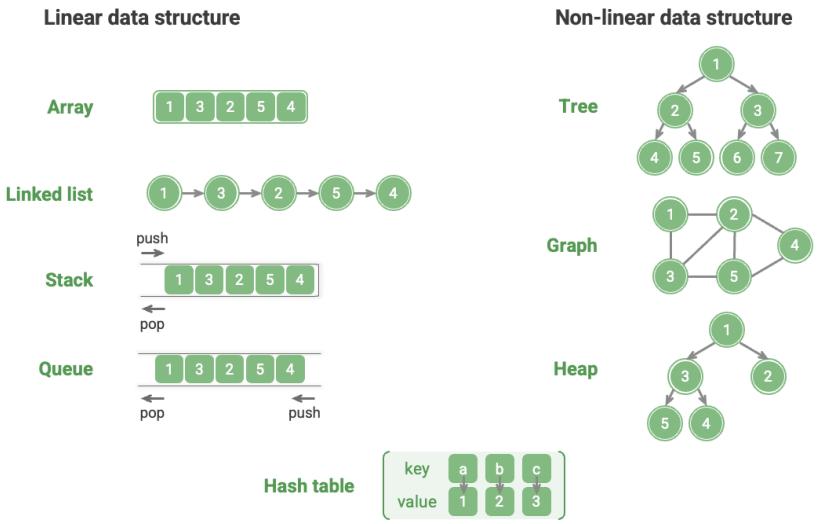


Figure 3-1 Linear and non-linear data structures

3.1.2 Physical Structure: Contiguous and Dispersed

When an algorithm program runs, the data being processed is mainly stored in memory. Figure 3-2 shows a computer memory stick, where each black square contains a memory space. We can imagine memory as a huge Excel spreadsheet, where each cell can store a certain amount of data.

The system accesses data at the target location through memory addresses. As shown in Figure 3-2, the computer assigns a number to each cell in the spreadsheet according to specific rules, ensuring that each memory space has a unique memory address. With these addresses, the program can access data in memory.

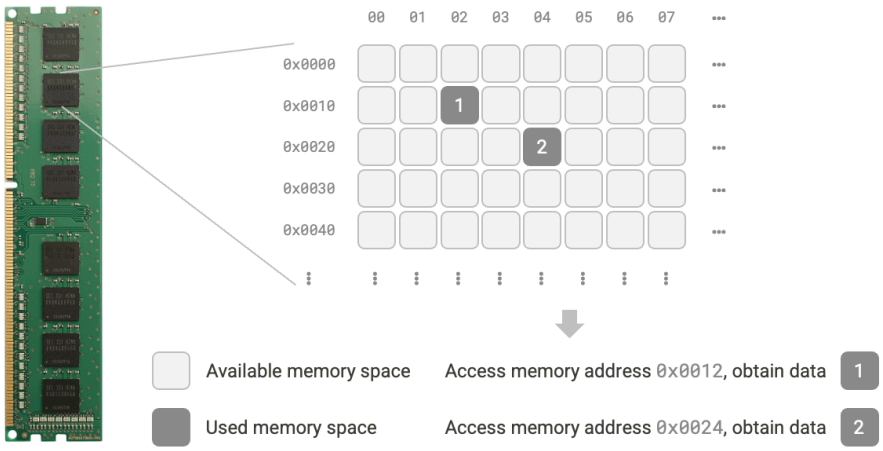


Figure 3-2 Memory stick, memory space, memory address

> **Tip**
> It is worth noting that comparing memory to an Excel spreadsheet is a simplified analogy. The actual working mechanism of memory is quite complex, involving concepts such as address space, memory management, cache mechanisms, virtual memory, and physical memory.

Memory is a shared resource for all programs. When a block of memory is occupied by a program, it usually cannot be used by other programs at the same time. **Therefore, in the design of data structures and algorithms, memory resources are an important consideration**. For example, the peak memory occupied by an algorithm should not exceed the remaining free memory of the system; if there is a lack of contiguous large memory blocks, then the data structure chosen must be able to be stored in dispersed memory spaces.

As shown in Figure 3-3, **physical structure reflects the way data is stored in computer memory**, and can be divided into contiguous space storage (arrays) and dispersed space storage (linked lists). The two physical structures exhibit complementary characteristics in terms of time efficiency and space efficiency.

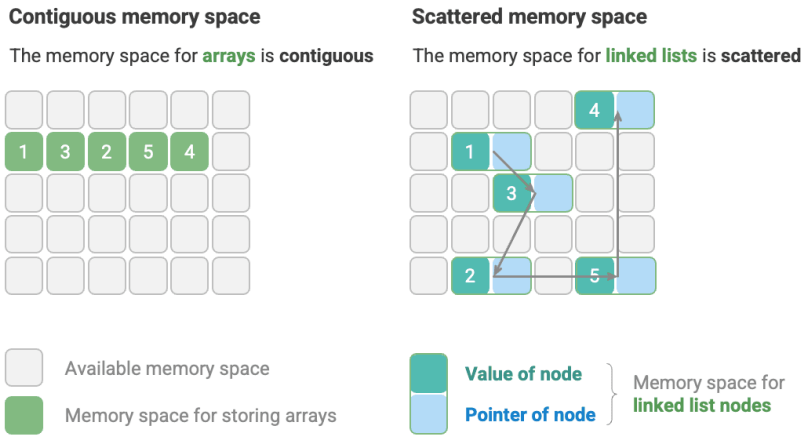


Figure 3-3 Contiguous space storage and dispersed space storage

It is worth noting that **all data structures are implemented based on arrays, linked lists, or a combination of both**. For example, stacks and queues can be implemented using either arrays or linked lists; while the implementation of hash tables may include both arrays and linked lists.

- **Can be implemented based on arrays**: Stacks, queues, hash tables, trees, heaps, graphs, matrices, tensors (arrays with dimensions ≥ 3), etc.
- **Can be implemented based on linked lists**: Stacks, queues, hash tables, trees, heaps, graphs, etc.

After initialization, linked lists can still adjust their length during program execution, so they are also called "dynamic data structures". After initialization, the length of arrays cannot be changed, so they are also called "static data structures". It is worth noting that arrays can achieve length changes by reallocating memory, thus possessing a certain degree of "dynamism".

Tip
If you find it difficult to understand physical structure, it is recommended to read the next chapter first, and then review this section.

3.2 Basic Data Types

When we talk about data in computers, we think of various forms such as text, images, videos, audio, 3D models, and more. Although these data are organized in different ways, they are all composed of various basic data types.

Basic data types are types that the CPU can directly operate on, and they are directly used in algorithms, mainly including the following.

- Integer types `byte`, `short`, `int`, `long`.
- Floating-point types `float`, `double`, used to represent decimal numbers.

- Character type `char`, used to represent letters, punctuation marks, and even emojis in various languages.
- Boolean type `bool`, used to represent "yes" and "no" judgments.

Basic data types are stored in binary form in computers. One binary bit is 1 bit. In most modern operating systems, 1 byte consists of 8 bits.

The range of values for basic data types depends on the size of the space they occupy. Below is an example using Java.

- Integer type `byte` occupies 1 byte = 8 bits, and can represent 2^8 numbers.
- Integer type `int` occupies 4 bytes = 32 bits, and can represent 2^{32} numbers.

The following table lists the space occupied, value ranges, and default values of various basic data types in Java. You don't need to memorize this table; a general understanding is sufficient, and you can refer to it when needed.

Table 3-1 Space occupied and value ranges of basic data types

Type	Symbol	Space Occupied	Minimum Value	Maximum Value	Default Value
Integer	`byte`	1 byte	-2^7 (-128)	$2^7 - 1$ (127)	0
	`short`	2 bytes	-2^{15}	$2^{15} - 1$	0
	`int`	4 bytes	-2^{31}	$2^{31} - 1$	0
	`long`	8 bytes	-2^{63}	$2^{63} - 1$	0
Float	`float`	4 bytes	1.175×10^{-38}	3.403×10^{38}	0.0f
	`double`	8 bytes	2.225×10^{-308}	1.798×10^{308}	0.0
Character	`char`	2 bytes	0	$2^{16} - 1$	0
Boolean	`bool`	1 byte	false	true	false

Please note that the above table is specific to Java's basic data types. Each programming language has its own data type definitions, and their space occupied, value ranges, and default values may vary.

- In Python, the integer type `int` can be of any size, limited only by available memory; the floating-point type `float` is double-precision 64-bit; there is no `char` type, a single character is actually a string `str` of length 1.
- C and C++ do not explicitly specify the size of basic data types, which varies by implementation and platform. The above table follows the LP64 data model, which is used in Unix 64-bit operating systems including Linux and macOS.
- The size of character `char` is 1 byte in C and C++, and in most programming languages it depends on the specific character encoding method, as detailed in the "Character Encoding" section.
- Even though representing a boolean value requires only 1 bit (0 or 1), it is usually stored as 1 byte in memory. This is because modern computer CPUs typically use 1 byte as the minimum addressable memory unit.

So, what is the relationship between basic data types and data structures? We know that data structures are ways of organizing and storing data in computers. The subject of this statement is "structure", not "data".

If we want to represent "a row of numbers", we naturally think of using an array. This is because the linear structure of an array can represent the adjacency and order relationships of numbers, but the content stored—whether integer `int`, floating-point `float`, or character `char`—is unrelated to the "data structure".

In other words, **basic data types provide the "content type" of data, while data structures provide the "organization method" of data**. For example, in the following code, we use the same data structure (array) to store and represent different basic data types, including `int`, `float`, `char`, `bool`, etc.

```
// JavaScript arrays can freely store various basic data types and objects
const array = [0, 0.0, 'a', false];
```

3.3 Number Encoding *

> **Tip**
> In this book, chapters marked with an asterisk * are optional readings. If you are short on time or find them challenging, you may skip these initially and return to them after completing the essential chapters.

3.3.1 Sign-Magnitude, 1's Complement, and 2's Complement

In the table from the previous section, we found that all integer types can represent one more negative number than positive numbers. For example, the `byte` range is $[-128, 127]$. This phenomenon is counterintuitive, and its underlying reason involves knowledge of sign-magnitude, 1's complement, and 2's complement.

First, it should be noted that **numbers are stored in computers in the form of "2's complement"**. Before analyzing the reasons for this, let's first define these three concepts.

- **Sign-magnitude**: We treat the highest bit of the binary representation of a number as the sign bit, where 0 represents a positive number and 1 represents a negative number, and the remaining bits represent the value of the number.
- **1's complement**: The 1's complement of a positive number is the same as its sign-magnitude. For a negative number, the 1's complement is obtained by inverting all bits except the sign bit of its sign-magnitude.
- **2's complement**: The 2's complement of a positive number is the same as its sign-magnitude. For a negative number, the 2's complement is obtained by adding 1 to its 1's complement.

Figure 3-4 shows the conversion methods among sign-magnitude, 1's complement, and 2's complement.

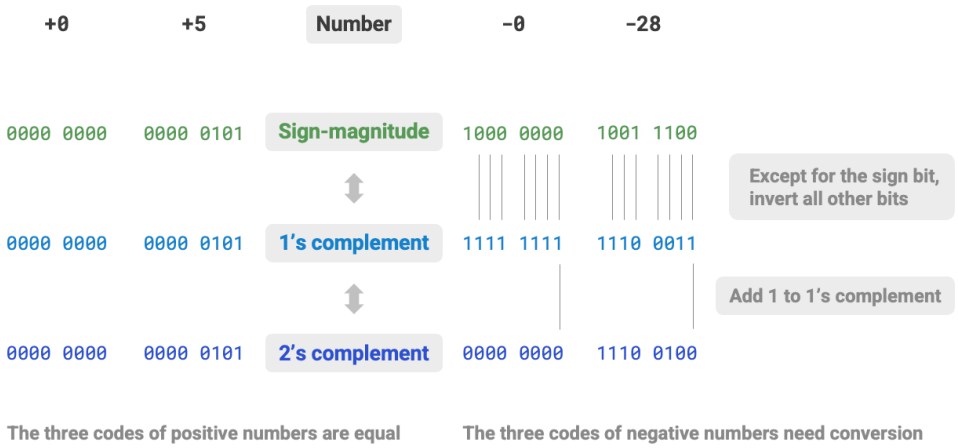


Figure 3-4 Conversions among sign-magnitude, 1's complement, and 2's complement

Sign-magnitude, although the most intuitive, has some limitations. On one hand, **the sign-magnitude of negative numbers cannot be directly used in operations**. For example, calculating $1 + (-2)$ in sign-magnitude yields -3, which is clearly incorrect.

$$
\begin{aligned}
& 1 + (-2) \\
& \rightarrow 0000\ 0001 + 1000\ 0010 \\
& = 1000\ 0011 \\
& \rightarrow -3
\end{aligned}
$$

To solve this problem, computers introduced 1's complement. If we first convert sign-magnitude to 1's complement and calculate $1 + (-2)$ in 1's complement, then convert the result back to sign-magnitude, we can obtain the correct result of -1.

$$
\begin{aligned}
& 1 + (-2) \\
& \rightarrow 0000\ 0001 \text{ (Sign-magnitude)} + 1000\ 0010 \text{ (Sign-magnitude)} \\
& = 0000\ 0001 \text{ (1's complement)} + 1111\ 1101 \text{ (1's complement)} \\
& = 1111\ 1110 \text{ (1's complement)} \\
& = 1000\ 0001 \text{ (Sign-magnitude)} \\
& \rightarrow -1
\end{aligned}
$$

On the other hand, **the sign-magnitude of the number zero has two representations,** $+0$ **and** -0. This means that the number zero corresponds to two different binary encodings, which may cause ambiguity. For example, in conditional judgments, if we don't distinguish between positive zero and negative zero, it may lead to incorrect judgment results. If we want to handle the ambiguity of positive and negative zero, we need to introduce additional judgment operations, which may reduce the computational efficiency of the computer.

$$+0 \rightarrow 0000\ 0000$$
$$-0 \rightarrow 1000\ 0000$$

Like sign-magnitude, 1's complement also has the problem of positive and negative zero ambiguity. Therefore, computers further introduced 2's complement. Let's first observe the conversion process of negative zero from sign-magnitude to 1's complement to 2's complement:

$$\begin{aligned} -0 &\rightarrow 1000\ 0000 \text{ (Sign-magnitude)} \\ &= 1111\ 1111 \text{ (1's complement)} \\ &= 1\ 0000\ 0000 \text{ (2's complement)} \end{aligned}$$

Adding 1 to the 1's complement of negative zero produces a carry, but since the `byte` type has a length of only 8 bits, the 1 that overflows to the 9th bit is discarded. That is to say, **the 2's complement of negative zero is $0000\ 0000$, which is the same as the 2's complement of positive zero**. This means that in 2's complement representation, there is only one zero, and the positive and negative zero ambiguity is thus resolved.

One last question remains: the range of the `byte` type is $[-128, 127]$, and how is the extra negative number -128 obtained? We notice that all integers in the interval $[-127, +127]$ have corresponding sign-magnitude, 1's complement, and 2's complement, and sign-magnitude and 2's complement can be converted to each other.

However, **the 2's complement $1000\ 0000$ is an exception, and it does not have a corresponding sign-magnitude**. According to the conversion method, we get that the sign-magnitude of this 2's complement is $0000\ 0000$. This is clearly contradictory because this sign-magnitude represents the number 0, and its 2's complement should be itself. The computer specifies that this special 2's complement $1000\ 0000$ represents -128. In fact, the result of calculating $(-1) + (-127)$ in 2's complement is -128.

$$\begin{aligned} &(-127) + (-1) \\ &\rightarrow 1111\ 1111 \text{ (Sign-magnitude)} + 1000\ 0001 \text{ (Sign-magnitude)} \\ &= 1000\ 0000 \text{ (1's complement)} + 1111\ 1110 \text{ (1's complement)} \\ &= 1000\ 0001 \text{ (2's complement)} + 1111\ 1111 \text{ (2's complement)} \\ &= 1000\ 0000 \text{ (2's complement)} \\ &\rightarrow -128 \end{aligned}$$

You may have noticed that all the above calculations are addition operations. This hints at an important fact: **the hardware circuits inside computers are mainly designed based on addition operations**. This is because addition operations are simpler to implement in hardware compared to other operations (such as multiplication, division, and subtraction), easier to parallelize, and have faster operation speeds.

Please note that this does not mean that computers can only perform addition. **By combining addition with some basic logical operations, computers can implement various other mathematical operations**.

For example, calculating the subtraction $a - b$ can be converted to calculating the addition $a + (-b)$; calculating multiplication and division can be converted to calculating multiple additions or subtractions.

Now we can summarize the reasons why computers use 2's complement: based on 2's complement representation, computers can use the same circuits and operations to handle the addition of positive and negative numbers, without the need to design special hardware circuits to handle subtraction, and without the need to specially handle the ambiguity problem of positive and negative zero. This greatly simplifies hardware design and improves operational efficiency.

The design of 2's complement is very ingenious. Due to space limitations, we will stop here. Interested readers are encouraged to explore further.

3.3.2 Floating-Point Number Encoding

Careful readers may have noticed: `int` and `float` have the same length, both are 4 bytes, but why does `float` have a much larger range than `int`? This is very counterintuitive because it stands to reason that `float` needs to represent decimals, so the range should be smaller.

In fact, **this is because floating-point number `float` uses a different representation method**. Let's denote a 32-bit binary number as:

$$b_{31}b_{30}b_{29} \ldots b_2b_1b_0$$

According to the IEEE 754 standard, a 32-bit `float` consists of the following three parts.

- Sign bit S: occupies 1 bit, corresponding to b_{31}.
- Exponent bit E: occupies 8 bits, corresponding to $b_{30}b_{29} \ldots b_{23}$.
- Fraction bit N: occupies 23 bits, corresponding to $b_{22}b_{21} \ldots b_0$.

The calculation method for the value corresponding to the binary `float` is:

$$\text{val} = (-1)^{b_{31}} \times 2^{(b_{30}b_{29} \ldots b_{23})_2 - 127} \times (1.b_{22}b_{21} \ldots b_0)_2$$

Converted to decimal, the calculation formula is:

$$\text{val} = (-1)^{\text{S}} \times 2^{\text{E}-127} \times (1 + \text{N})$$

The range of each component is:

$$\text{S} \in \{0, 1\}, \quad \text{E} \in \{1, 2, \ldots, 254\}$$

$$(1 + \text{N}) = (1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}) \subset [1, 2 - 2^{-23}]$$

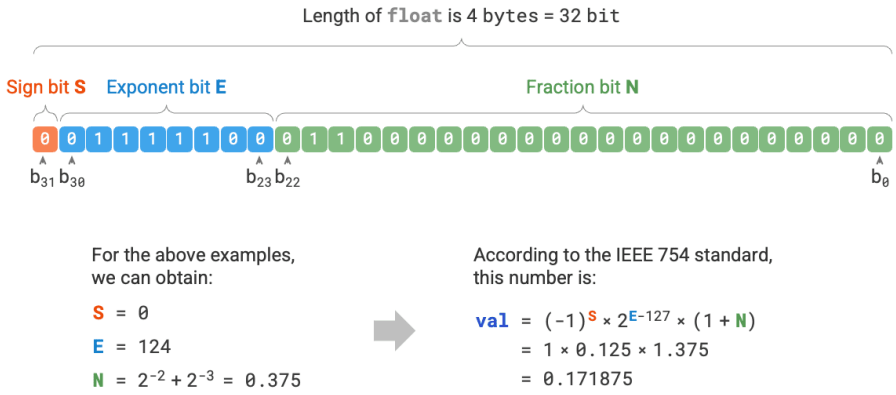


Figure 3-5 Calculation example of float under IEEE 754 standard

Observing Figure 3-5, given example data $\text{S} = 0$, $\text{E} = 124$, $\text{N} = 2^{-2} + 2^{-3} = 0.375$, we have:

$$\text{val} = (-1)^0 \times 2^{124-127} \times (1 + 0.375) = 0.171875$$

Now we can answer the initial question: **the representation of `float` includes an exponent bit, resulting in a range far greater than `int`**. According to the above calculation, the maximum positive number that `float` can represent is $2^{254-127} \times (2 - 2^{-23}) \approx 3.4 \times 10^{38}$, and the minimum negative number can be obtained by switching the sign bit.

Although floating-point number `float` expands the range, its side effect is sacrificing precision. The integer type `int` uses all 32 bits to represent numbers, and the numbers are evenly distributed; however, due to the existence of the exponent bit, the larger the value of floating-point number `float`, the larger the difference between two adjacent numbers tends to be.

As shown in Table 3-2, exponent bits $\text{E} = 0$ and $\text{E} = 255$ have special meanings, **used to represent zero, infinity,** NaN**, etc.**

Table 3-2 Meaning of exponent bits

Exponent Bit E	Fraction Bit $\text{N} = 0$	Fraction Bit $\text{N} \neq 0$	Calculation Formula
0	± 0	Subnormal Number	$(-1)^{\text{S}} \times 2^{-126} \times (0.\text{N})$
$1, 2, \dots, 254$	Normal Number	Normal Number	$(-1)^{\text{S}} \times 2^{(\text{E}-127)} \times (1.\text{N})$
255	$\pm\infty$	NaN	

It is worth noting that subnormal numbers significantly improve the precision of floating-point numbers. The smallest positive normal number is 2^{-126}, and the smallest positive subnormal number is $2^{-126} \times 2^{-23}$.

Double-precision `double` also uses a representation method similar to `float`, which will not be elaborated here.

3.4 Character Encoding *

In computers, all data is stored in binary form, and character `char` is no exception. To represent characters, we need to establish a "character set" that defines a one-to-one correspondence between each character and binary numbers. With a character set, computers can convert binary numbers to characters by looking up the table.

3.4.1 Ascii Character Set

ASCII code is the earliest character set, with the full name American Standard Code for Information Interchange. It uses 7 binary bits (the lower 7 bits of one byte) to represent a character, and can represent a maximum of 128 different characters. As shown in Figure 3-6, ASCII code includes uppercase and lowercase English letters, numbers 0 ~ 9, some punctuation marks, and some control characters (such as newline and tab).

Oct	Binary	Char	Name
0	0000 0000	NUL	Null
1	0000 0001	SOH	Start of heading
2	0000 0010	STX	Start of text
3	0000 0011	ETX	End of text
4	0000 0100	EOT	End of transmission
5	0000 0101	ENQ	Enquiry
6	0000 0110	ACK	Acknowledge
7	0000 0111	BEL	Bell
8	0000 1000	BS	Backspace
9	0000 1001	HT	Horizontal tab
10	0000 1010	LF	NL line feed, new line
11	0000 1011	VT	Vertical tab
12	0000 1100	FF	NP form feed, new page
13	0000 1101	CR	Carriage return
14	0000 1110	SO	Shift out
15	0000 1111	SI	Shift in
16	0001 0000	DLE	Data link escape
17	0001 0001	DC1	Device control 1
18	0001 0010	DC2	Device control 2
19	0001 0011	DC3	Device control 3
20	0001 0100	DC4	Device control 4
21	0001 0101	NAK	Negative acknowledge
22	0001 0110	SYN	Synchronous idle
23	0001 0111	ETB	End of trans. block
24	0001 1000	CAN	Cancel
25	0001 1001	EM	End of medium
26	0001 1010	SUB	Substitute
27	0001 1011	ESC	Escape
28	0001 1100	FS	File separator
29	0001 1101	GS	Group separator
30	0001 1110	RS	Record separator
31	0001 1111	US	Unit separator
32	0010 0000	SP	Space

Oct	Binary	Char
33	0010 0001	!
34	0010 0010	"
35	0010 0011	#
36	0010 0100	$
37	0010 0101	%
38	0010 0110	&
39	0010 0111	'
40	0010 1000	(
41	0010 1001	)
42	0010 1010	*
43	0010 1011	+
44	0010 1100	,
45	0010 1101	-
46	0010 1110	.
47	0010 1111	/
48	0011 0000	0
49	0011 0001	1
50	0011 0010	2
51	0011 0011	3
52	0011 0100	4
53	0011 0101	5
54	0011 0110	6
55	0011 0111	7
56	0011 1000	8
57	0011 1001	9
58	0011 1010	:
59	0011 1011	;
60	0011 1100	<
61	0011 1101	=
62	0011 1110	>
63	0011 1111	?
64	0100 0000	@

Oct	Binary	Char
65	0100 0001	A
66	0100 0010	B
67	0100 0011	C
68	0100 0100	D
69	0100 0101	E
70	0100 0110	F
71	0100 0111	G
72	0100 1000	H
73	0100 1001	I
74	0100 1010	J
75	0100 1011	K
76	0100 1100	L
77	0100 1101	M
78	0100 1110	N
79	0100 1111	O
80	0101 0000	P
81	0101 0001	Q
82	0101 0010	R
83	0101 0011	S
84	0101 0100	T
85	0101 0101	U
86	0101 0110	V
87	0101 0111	W
88	0101 1000	X
89	0101 1001	Y
90	0101 1010	Z
91	0101 1011	[
92	0101 1100	\
93	0101 1101	]
94	0101 1110	^
95	0101 1111	_
96	0110 0000	`

Oct	Binary	Char
97	0110 0001	a
98	0110 0010	b
99	0110 0011	c
100	0110 0100	d
101	0110 0101	e
102	0110 0110	f
103	0110 0111	g
104	0110 1000	h
105	0110 1001	i
106	0110 1010	j
107	0110 1011	k
108	0110 1100	l
109	0110 1101	m
110	0110 1110	n
111	0110 1111	o
112	0111 0000	p
113	0111 0001	q
114	0111 0010	r
115	0111 0011	s
116	0111 0100	t
117	0111 0101	u
118	0111 0110	v
119	0111 0111	w
120	0111 1000	x
121	0111 1001	y
122	0111 1010	z
123	0111 1011	{
124	0111 1100	|
125	0111 1101	}
126	0111 1110	~
127	0111 1111	DEL

Figure 3-6 ASCII code

However, **ASCII code can only represent English**. With the globalization of computers, a character set called EASCII that can represent more languages emerged. It expands from the 7-bit basis of ASCII to 8 bits, and can represent 256 different characters.

Worldwide, a batch of EASCII character sets suitable for different regions have appeared successively. The first 128 characters of these character sets are unified as ASCII code, and the last 128 characters are defined differently to adapt to the needs of different languages.

3.4.2 Gbk Character Set

Later, people found that **EASCII code still cannot meet the character quantity requirements of many languages**. For example, there are nearly one hundred thousand Chinese characters, and several thou-

sand are used daily. In 1980, the China National Standardization Administration released the GB2312 character set, which included 6,763 Chinese characters, basically meeting the needs for computer processing of Chinese characters.

However, GB2312 cannot handle some rare characters and traditional Chinese characters. The GBK character set is an extension based on GB2312, which includes a total of 21,886 Chinese characters. In the GBK encoding scheme, ASCII characters are represented using one byte, and Chinese characters are represented using two bytes.

3.4.3 Unicode Character Set

With the vigorous development of computer technology, character sets and encoding standards flourished, which brought many problems. On the one hand, these character sets generally only define characters for specific languages and cannot work normally in multilingual environments. On the other hand, multiple character set standards exist for the same language, and if two computers use different encoding standards, garbled characters will appear during information transmission.

Researchers of that era thought: **If a sufficiently complete character set is released that includes all languages and symbols in the world, wouldn't it be possible to solve cross-language environment and garbled character problems**? Driven by this idea, a large and comprehensive character set, Unicode, was born.

Unicode is called "统一码" (Unified Code) in Chinese and can theoretically accommodate over one million characters. It is committed to including characters from around the world into a unified character set, providing a universal character set to handle and display various language texts, reducing garbled character problems caused by different encoding standards.

Since its release in 1991, Unicode has continuously expanded to include new languages and characters. As of September 2022, Unicode has included 149,186 characters, including characters, symbols, and even emojis from various languages. In the vast Unicode character set, commonly used characters occupy 2 bytes, and some rare characters occupy 3 bytes or even 4 bytes.

Unicode is a universal character set that essentially assigns a number (called a "code point") to each character, **but it does not specify how to store these character code points in computers**. We can't help but ask: when Unicode code points of multiple lengths appear simultaneously in a text, how does the system parse the characters? For example, given an encoding with a length of 2 bytes, how does the system determine whether it is one 2-byte character or two 1-byte characters?

For the above problem, **a straightforward solution is to store all characters as equal-length encodings**. As shown in Figure 3-7, each character in "Hello" occupies 1 byte, and each character in "算法" (algorithm) occupies 2 bytes. We can encode all characters in "Hello 算法" as 2 bytes in length by padding the high bits with 0. In this way, the system can parse one character every 2 bytes and restore the content of this phrase.

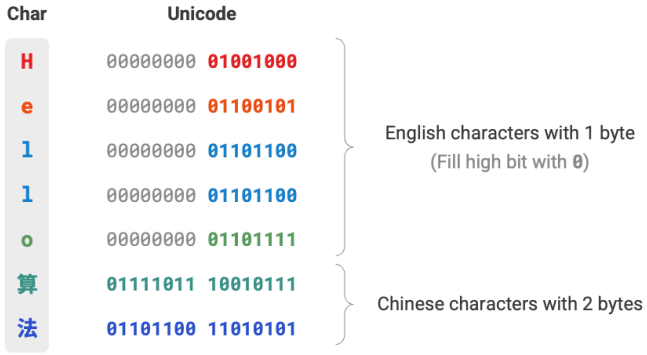


Figure 3-7 Unicode encoding example

However, ASCII code has already proven to us that encoding English only requires 1 byte. If the above scheme is adopted, the size of English text will be twice that under ASCII encoding, which is very wasteful of memory space. Therefore, we need a more efficient Unicode encoding method.

3.4.4 Utf-8 Encoding

Currently, UTF-8 has become the most widely used Unicode encoding method internationally. **It is a variable-length encoding** that uses 1 to 4 bytes to represent a character, depending on the complexity of the character. ASCII characters only require 1 byte, Latin and Greek letters require 2 bytes, commonly used Chinese characters require 3 bytes, and some other rare characters require 4 bytes.

The encoding rules of UTF-8 are not complicated and can be divided into the following two cases.

- For 1-byte characters, set the highest bit to 0, and set the remaining 7 bits to the Unicode code point. It is worth noting that ASCII characters occupy the first 128 code points in the Unicode character set. That is to say, **UTF-8 encoding is backward compatible with ASCII code**. This means we can use UTF-8 to parse very old ASCII code text.
- For characters with a length of n bytes (where $n > 1$), set the highest n bits of the first byte to 1, and set the $(n + 1)$-th bit to 0; starting from the second byte, set the highest 2 bits of each byte to 10; use all remaining bits to fill in the Unicode code point of the character.

Figure 3-8 shows the UTF-8 encoding corresponding to "Hello 算法". It can be observed that since the highest n bits are all set to 1, the system can parse the length of the character as n by reading the number of highest bits that are 1.

But why set the highest 2 bits of all other bytes to 10? In fact, this 10 can serve as a check symbol. Assuming the system starts parsing text from an incorrect byte, the 10 at the beginning of the byte can help the system quickly determine an anomaly.

The reason for using 10 as a check symbol is that under UTF-8 encoding rules, it is impossible for a character's highest two bits to be 10. This conclusion can be proven by contradiction: assuming the highest two bits of a character are 10, it means the length of the character is 1, corresponding to ASCII code. However, the highest bit of ASCII code should be 0, which contradicts the assumption.

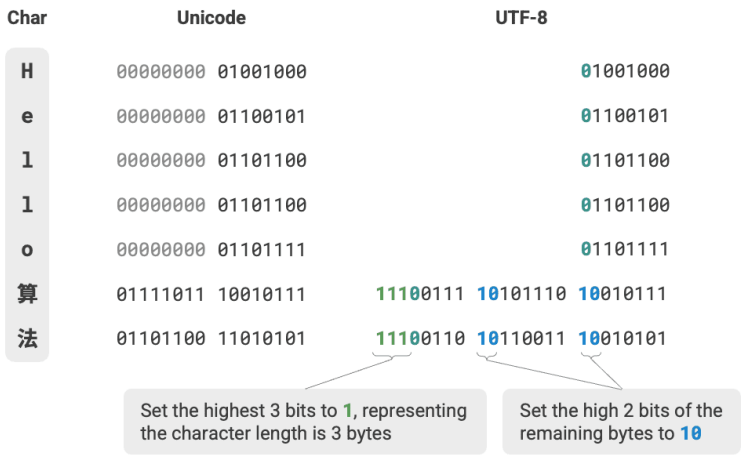


Figure 3-8 UTF-8 encoding example

In addition to UTF-8, common encoding methods also include the following two.

- **UTF-16 encoding**: Uses 2 or 4 bytes to represent a character. All ASCII characters and commonly used non-English characters are represented with 2 bytes; a few characters need to use 4 bytes. For 2-byte characters, UTF-16 encoding is equal to the Unicode code point.
- **UTF-32 encoding**: Every character uses 4 bytes. This means that UTF-32 takes up more space than UTF-8 and UTF-16, especially for text with a high proportion of ASCII characters.

From the perspective of storage space occupation, using UTF-8 to represent English characters is very efficient because it only requires 1 byte; using UTF-16 encoding for some non-English characters (such as Chinese) will be more efficient because it only requires 2 bytes, while UTF-8 may require 3 bytes.

From a compatibility perspective, UTF-8 has the best universality, and many tools and libraries support UTF-8 first.

3.4.5 Character Encoding in Programming Languages

For most past programming languages, strings during program execution use fixed-length encodings such as UTF-16 or UTF-32. Under fixed-length encoding, we can treat strings as arrays for processing, and this approach has the following advantages.

- **Random access**: UTF-16 encoded strings can be easily accessed randomly. UTF-8 is a variable-length encoding. To find the i-th character, we need to traverse from the beginning of the string to the i-th character, which requires $O(n)$ time.
- **Character counting**: Similar to random access, calculating the length of a UTF-16 encoded string is also an $O(1)$ operation. However, calculating the length of a UTF-8 encoded string requires traversing the entire string.
- **String operations**: Many string operations (such as splitting, joining, inserting, deleting, etc.) on UTF-16 encoded strings are easier to perform. Performing these operations on UTF-8 encoded

strings usually requires additional calculations to ensure that invalid UTF-8 encoding is not generated.

In fact, the design of character encoding schemes for programming languages is a very interesting topic involving many factors.

- Java's `String` type uses UTF-16 encoding, with each character occupying 2 bytes. This is because at the beginning of Java language design, people believed that 16 bits were sufficient to represent all possible characters. However, this was an incorrect judgment. Later, the Unicode specification expanded beyond 16 bits, so characters in Java may now be represented by a pair of 16-bit values (called "surrogate pairs").
- The strings of JavaScript and TypeScript use UTF-16 encoding for reasons similar to Java. When Netscape first introduced the JavaScript language in 1995, Unicode was still in its early stages of development, and at that time, using 16-bit encoding was sufficient to represent all Unicode characters.
- C# uses UTF-16 encoding mainly because the .NET platform was designed by Microsoft, and many of Microsoft's technologies (including the Windows operating system) extensively use UTF-16 encoding.

Due to the underestimation of character quantities by the above programming languages, they had to adopt the "surrogate pair" method to represent Unicode characters with lengths exceeding 16 bits. This is a reluctant compromise. On the one hand, in strings containing surrogate pairs, one character may occupy 2 bytes or 4 bytes, thus losing the advantage of fixed-length encoding. On the other hand, handling surrogate pairs requires additional code, which increases the complexity and difficulty of debugging in programming.

For the above reasons, some programming languages have proposed different encoding schemes.

- Python's `str` uses Unicode encoding and adopts a flexible string representation where the stored character length depends on the largest Unicode code point in the string. If all characters in the string are ASCII characters, each character occupies 1 byte; if there are characters exceeding the ASCII range but all within the Basic Multilingual Plane (BMP), each character occupies 2 bytes; if there are characters exceeding the BMP, each character occupies 4 bytes.
- Go language's `string` type uses UTF-8 encoding internally. Go language also provides the `rune` type, which is used to represent a single Unicode code point.
- Rust language's `str` and `String` types use UTF-8 encoding internally. Rust also provides the `char` type for representing a single Unicode code point.

It should be noted that the above discussion is about how strings are stored in programming languages, **which is different from how strings are stored in files or transmitted over networks**. In file storage or network transmission, we usually encode strings into UTF-8 format to achieve optimal compatibility and space efficiency.

3.5 Summary

1. Key Review

- Data structures can be classified from two perspectives: logical structure and physical structure. Logical structure describes the logical relationships between data elements, while physical structure describes how data is stored in computer memory.
- Common logical structures include linear, tree, and network structures. We typically classify data structures as linear (arrays, linked lists, stacks, queues) and non-linear (trees, graphs, heaps) based on their logical structure. The implementation of hash tables may involve both linear and non-linear data structures.
- When a program runs, data is stored in computer memory. Each memory space has a corresponding memory address, and the program accesses data through these memory addresses.
- Physical structures are primarily divided into contiguous space storage (arrays) and dispersed space storage (linked lists). All data structures are implemented using arrays, linked lists, or a combination of both.
- Basic data types in computers include integers `byte`, `short`, `int`, `long`, floating-point numbers `float`, `double`, characters `char`, and booleans `bool`. Their value ranges depend on the size of space they occupy and their representation method.
- Sign-magnitude, 1's complement, and 2's complement are three methods for encoding numbers in computers, and they can be converted into each other. The most significant bit of sign-magnitude is the sign bit, and the remaining bits represent the value of the number.
- Integers are stored in computers in 2's complement form. Under 2's complement representation, computers can treat the addition of positive and negative numbers uniformly, without needing to design special hardware circuits for subtraction, and there is no ambiguity of positive and negative zero.
- The encoding of floating-point numbers consists of 1 sign bit, 8 exponent bits, and 23 fraction bits. Due to the exponent bits, the range of floating-point numbers is much larger than that of integers, at the cost of sacrificing precision.
- ASCII is the earliest English character set, with a length of 1 byte, containing a total of 127 characters. GBK is a commonly used Chinese character set, containing over 20,000 Chinese characters. Unicode is committed to providing a complete character set standard, collecting characters from various languages around the world, thereby solving the garbled text problem caused by inconsistent character encoding methods.
- UTF-8 is the most popular Unicode encoding method, with excellent universality. It is a variable-length encoding method with good scalability, effectively improving storage space efficiency. UTF-16 and UTF-32 are fixed-length encoding methods. When encoding Chinese characters, UTF-16 occupies less space than UTF-8. Programming languages such as Java and C# use UTF-16 encoding by default.

2. Q & A

Q: Why do hash tables contain both linear and non-linear data structures?

The underlying structure of a hash table is an array. To resolve hash collisions, we may use "chaining" (discussed in the subsequent "Hash Collision" section): each bucket in the array points to a linked list, which may be converted to a tree (usually a red-black tree) when the list length exceeds a certain threshold.

From a storage perspective, the underlying structure of a hash table is an array, where each bucket slot may contain a value, a linked list, or a tree. Therefore, hash tables may contain both linear data structures (arrays, linked lists) and non-linear data structures (trees).

Q: Is the length of the `char` type 1 byte?

The length of the `char` type is determined by the encoding method used by the programming language. For example, Java, JavaScript, TypeScript, and C# all use UTF-16 encoding (to store Unicode code points), so the `char` type has a length of 2 bytes.

Q: Is there ambiguity in referring to array-based data structures as "static data structures"? Stacks can also perform "dynamic" operations such as push and pop.

Stacks can indeed implement dynamic data operations, but the data structure is still "static" (fixed length). Although array-based data structures can dynamically add or remove elements, their capacity is fixed. If the data volume exceeds the pre-allocated size, a new larger array needs to be created, and the contents of the old array must be copied to the new array.

Q: When constructing a stack (queue), its size is not specified. Why are they "static data structures"?

In high-level programming languages, we do not need to manually specify the initial capacity of a stack (queue); this work is automatically completed within the class. For example, the initial capacity of Java's `ArrayList` is typically 10. Additionally, the expansion operation is also automatically implemented. See the subsequent "List" section for details.

Q: The method of converting sign-magnitude to 2's complement is "first negate then add 1". So converting 2's complement to sign-magnitude should be the inverse operation "first subtract 1 then negate". However, 2's complement can also be converted to sign-magnitude through "first negate then add 1". Why is this?

This is because the mutual conversion between sign-magnitude and 2's complement is actually the process of computing the "complement". Let us first define the complement: assuming $a + b = c$, then we say that a is the complement of b to c, and conversely, b is the complement of a to c.

Given an $n = 4$ bit binary number 0010, if we treat this number as sign-magnitude (ignoring the sign bit), then its 2's complement can be obtained through "first negate then add 1":

$$0010 \to 1101 \to 1110$$

We find that the sum of sign-magnitude and 2's complement is $0010 + 1110 = 10000$, which means the 2's complement 1110 is the "complement" of sign-magnitude 0010 to 10000. **This means the above "first negate then add 1" is actually the process of computing the complement to** 10000.

So, what is the "complement" of 2's complement 1110 to 10000? We can still use "first negate then add 1" to obtain it:

$$1110 \to 0001 \to 0010$$

In other words, sign-magnitude and 2's complement are each other's "complement" to 10000, so "sign-magnitude to 2's complement" and "2's complement to sign-magnitude" can be implemented using the same operation (first negate then add 1).

Of course, we can also use the inverse operation to find the sign-magnitude of 2's complement 1110, that is, "first subtract 1 then negate":

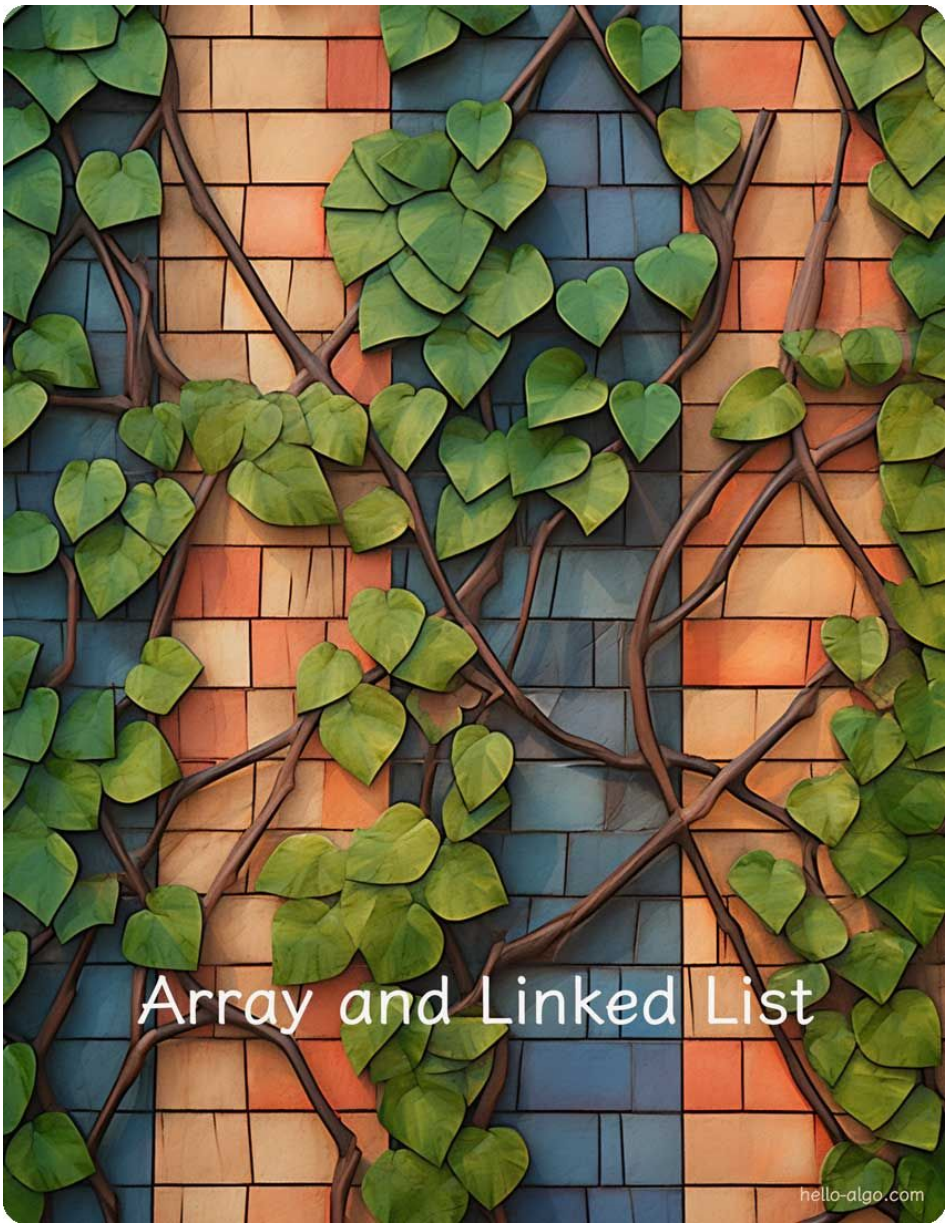
$$1110 \to 1101 \to 0010$$

In summary, both "first negate then add 1" and "first subtract 1 then negate" are computing the complement to 10000, and they are equivalent.

Essentially, the "negate" operation is actually finding the complement to 1111 (because `sign-magnitude + 1's comple` always holds); and adding 1 to the 1's complement yields the 2's complement, which is the complement to 10000.

The above uses $n = 4$ as an example, and it can be generalized to binary numbers of any number of bits.

Chapter 4. Array and Linked List



Abstract

The world of data structures is like a solid brick wall.

Array bricks are neatly arranged, tightly packed one by one. Linked list bricks are scattered everywhere, with connecting vines freely weaving through the gaps between bricks.

4.1 Array

An array is a linear data structure that stores elements of the same type in contiguous memory space. The position of an element in the array is called the element's index. Figure 4-1 illustrates the main concepts and storage method of arrays.

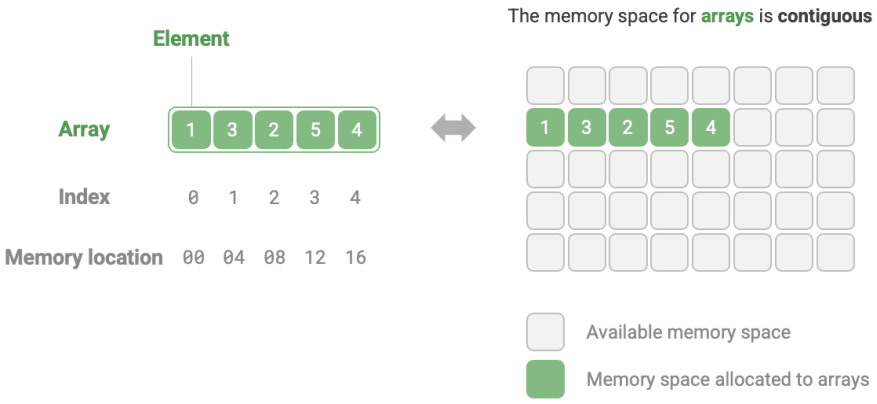


Figure 4-1 Array definition and storage method

4.1.1 Common Array Operations

1. Initializing Arrays

We can choose between two array initialization methods based on our needs: without initial values or with given initial values. When no initial values are specified, most programming languages will initialize array elements to 0:

```
// ═══ File: array.js ═══

/* Initialize array */
var arr = new Array(5).fill(0);
var nums = [1, 3, 2, 5, 4];
```

2. Accessing Elements

Array elements are stored in contiguous memory space, which means calculating the memory address of array elements is very easy. Given the array's memory address (the memory address of the first element) and an element's index, we can use the formula shown in Figure 4-2 to calculate the element's memory address and directly access that element.

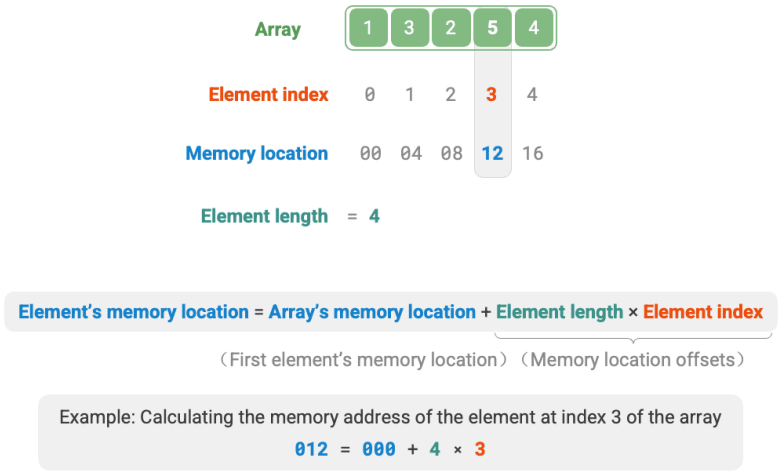


Figure 4-2 Memory address calculation for array elements

Observing Figure 4-2, we find that the first element of an array has an index of 0, which may seem counterintuitive since counting from 1 would be more natural. However, from the perspective of the address calculation formula, **an index is essentially an offset from the memory address**. The address offset of the first element is 0, so it is reasonable for its index to be 0.

Accessing elements in an array is highly efficient; we can randomly access any element in the array in $O(1)$ time.

```
// ═══ File: array.js ═══

/* Random access to element */
function randomAccess(nums) {
    // Randomly select a number in the interval [0, nums.length)
    const random_index = Math.floor(Math.random() * nums.length);
    // Retrieve and return the random element
    const random_num = nums[random_index];
    return random_num;
}
```

3. Inserting Elements

Array elements are stored "tightly adjacent" in memory, with no space between them to store any additional data. As shown in Figure 4-3, if we want to insert an element in the middle of an array, we need to shift all elements after that position backward by one position, and then assign the value to that index.

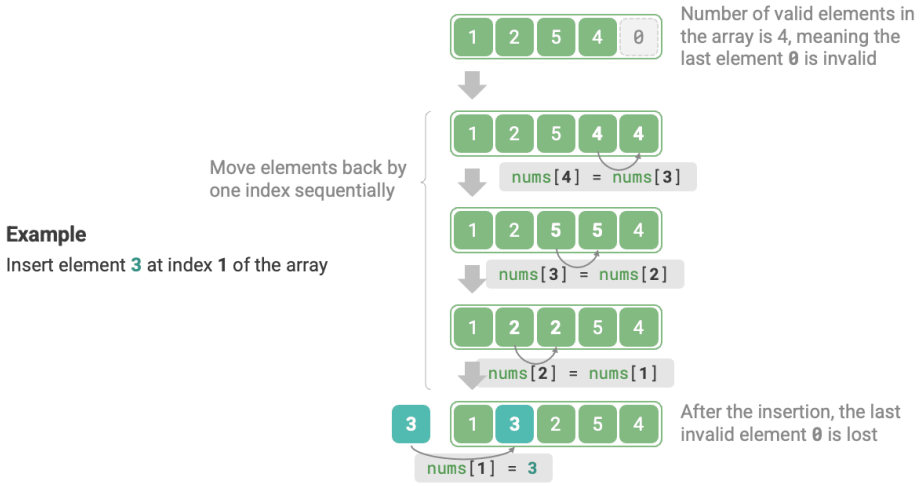


Figure 4-3 Example of inserting an element into an array

It is worth noting that since the length of an array is fixed, inserting an element will inevitably cause the element at the end of the array to be "lost". We will leave the solution to this problem for discussion in the "List" chapter.

```
// ═══ File: array.js ═══

/* Insert element num at index index in the array */
function insert(nums, num, index) {
    // Move all elements at and after index index backward by one position
    for (let i = nums.length - 1; i > index; i--) {
        nums[i] = nums[i - 1];
    }
    // Assign num to the element at index index
    nums[index] = num;
}
```

4. Removing Elements

Similarly, as shown in Figure 4-4, to delete the element at index i, we need to shift all elements after index i forward by one position.

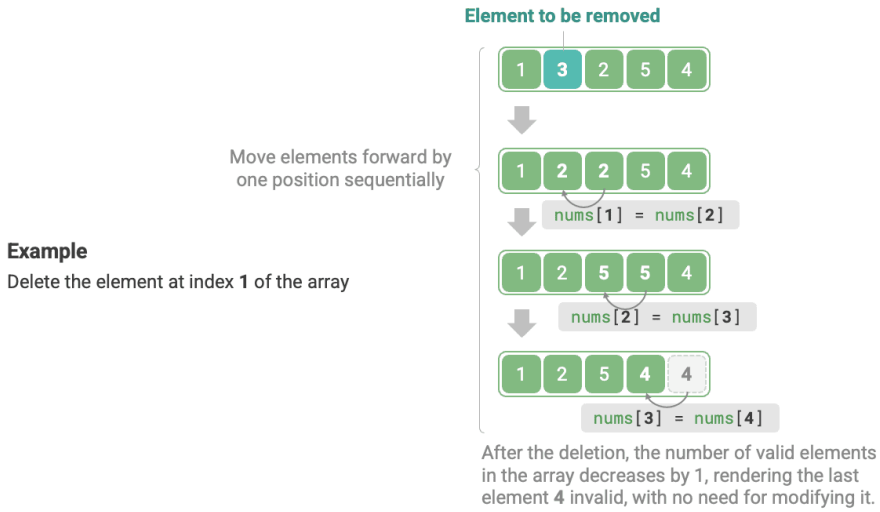


Figure 4-4 Example of removing an element from an array

Note that after the deletion is complete, the original last element becomes "meaningless", so we do not need to specifically modify it.

```
// ═══ File: array.js ═══

/* Remove the element at index index */
function remove(nums, index) {
    // Move all elements after index index forward by one position
    for (let i = index; i < nums.length - 1; i++) {
        nums[i] = nums[i + 1];
    }
}
```

Overall, array insertion and deletion operations have the following drawbacks:

- **High time complexity**: The average time complexity for both insertion and deletion in arrays is $O(n)$, where n is the length of the array.
- **Loss of elements**: Since the length of an array is immutable, after inserting an element, elements that exceed the array's length will be lost.
- **Memory waste**: We can initialize a relatively long array and only use the front portion, so that when inserting data, the lost elements at the end are "meaningless", but this causes some memory space to be wasted.

5. Traversing Arrays

In most programming languages, we can traverse an array either by index or by directly iterating through each element in the array:

```javascript
// ═══ File: array.js ═══

/* Traverse array */
function traverse(nums) {
    let count = 0;
    // Traverse array by index
    for (let i = 0; i < nums.length; i++) {
        count += nums[i];
    }
    // Direct traversal of array elements
    for (const num of nums) {
        count += num;
    }
}
```

6. Finding Elements

Finding a specified element in an array requires traversing the array and checking whether the element value matches in each iteration; if it matches, output the corresponding index.

Since an array is a linear data structure, the above search operation is called a "linear search".

```javascript
// ═══ File: array.js ═══

/* Find the specified element in the array */
function find(nums, target) {
    for (let i = 0; i < nums.length; i++) {
        if (nums[i] === target) return i;
    }
    return -1;
}
```

7. Expanding Arrays

In complex system environments, programs cannot guarantee that the memory space after an array is available, making it unsafe to expand the array's capacity. Therefore, in most programming languages, **the length of an array is immutable**.

If we want to expand an array, we need to create a new, larger array and then copy the original array elements to the new array one by one. This is an $O(n)$ operation, which is very time-consuming when the array is large. The code is shown below:

```javascript
// ═══ File: array.js ═══

/* Extend array length */
// Note: JavaScript's Array is dynamic array, can be directly expanded
// For learning purposes, this function treats Array as fixed-length array
function extend(nums, enlarge) {
    // Initialize an array with extended length
    const res = new Array(nums.length + enlarge).fill(0);
    // Copy all elements from the original array to the new array
    for (let i = 0; i < nums.length; i++) {
        res[i] = nums[i];
```

```
    }
    // Return the extended new array
    return res;
}
```

4.1.2 Advantages and Limitations of Arrays

Arrays are stored in contiguous memory space with elements of the same type. This approach contains rich prior information that the system can use to optimize the efficiency of data structure operations.

- **High space efficiency**: Arrays allocate contiguous memory blocks for data without additional structural overhead.
- **Support for random access**: Arrays allow accessing any element in $O(1)$ time.
- **Cache locality**: When accessing array elements, the computer not only loads the element but also caches the surrounding data, thereby leveraging the cache to improve the execution speed of subsequent operations.

Contiguous space storage is a double-edged sword with the following limitations:

- **Low insertion and deletion efficiency**: When an array has many elements, insertion and deletion operations require shifting a large number of elements.
- **Immutable length**: After an array is initialized, its length is fixed. Expanding the array requires copying all data to a new array, which is very costly.
- **Space waste**: If the allocated size of an array exceeds what is actually needed, the extra space is wasted.

4.1.3 Typical Applications of Arrays

Arrays are a fundamental and common data structure, frequently used in various algorithms and for implementing various complex data structures.

- **Random access**: If we want to randomly sample some items, we can use an array to store them and generate a random sequence to implement random sampling based on indices.
- **Sorting and searching**: Arrays are the most commonly used data structure for sorting and searching algorithms. Quick sort, merge sort, binary search, and others are primarily performed on arrays.
- **Lookup tables**: When we need to quickly find an element or its corresponding relationship, we can use an array as a lookup table. For example, if we want to implement a mapping from characters to ASCII codes, we can use the ASCII code value of a character as an index, with the corresponding element stored at that position in the array.
- **Machine learning**: Neural networks make extensive use of linear algebra operations between vectors, matrices, and tensors, all of which are constructed in the form of arrays. Arrays are the most commonly used data structure in neural network programming.
- **Data structure implementation**: Arrays can be used to implement stacks, queues, hash tables, heaps, graphs, and other data structures. For example, the adjacency matrix representation of a graph is essentially a two-dimensional array.

4.2 Linked List

Memory space is a shared resource for all programs. In a complex system runtime environment, available memory space may be scattered throughout the memory. We know that the memory space for storing an array must be contiguous, and when the array is very large, the memory may not be able to provide such a large contiguous space. This is where the flexibility advantage of linked lists becomes apparent.

A linked list is a linear data structure in which each element is a node object, and the nodes are connected through "references". A reference records the memory address of the next node, through which the next node can be accessed from the current node.

The design of linked lists allows nodes to be stored scattered throughout the memory, and their memory addresses do not need to be contiguous.

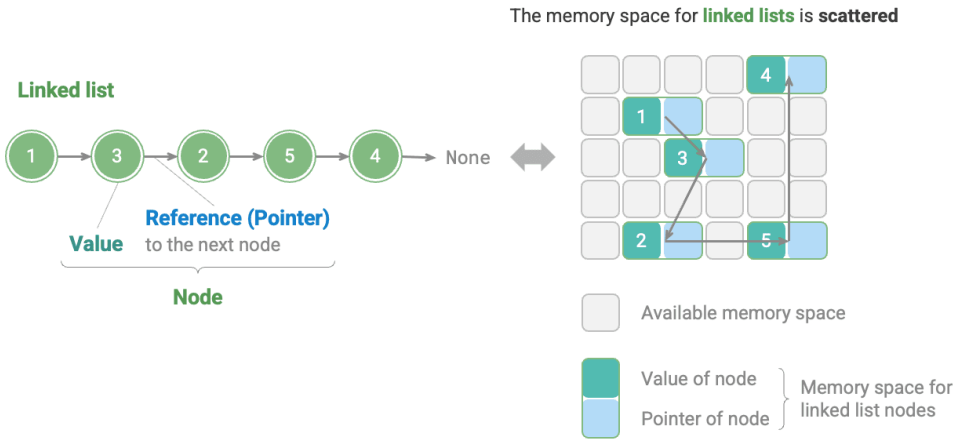


Figure 4-5 Linked list definition and storage method

Observing Figure 4-5, the basic unit of a linked list is a node object. Each node contains two pieces of data: the node's "value" and a "reference" to the next node.

- The first node of a linked list is called the "head node", and the last node is called the "tail node".
- The tail node points to "null", which is denoted as `null`, `nullptr`, and `None` in Java, C++, and Python, respectively.
- In languages that support pointers, such as C, C++, Go, and Rust, the aforementioned "reference" should be replaced with "pointer".

As shown in the following code, a linked list node `ListNode` contains not only a value but also an additional reference (pointer). Therefore, **linked lists occupy more memory space than arrays when storing the same amount of data**.

```
/* Linked list node class */
class ListNode {
```

```
    constructor(val, next) {
        this.val = (val === undefined ? 0 : val);       // Node value
        this.next = (next === undefined ? null : next); // Reference to the next node
    }
}
```

4.2.1 Common Linked List Operations

1. Initializing a Linked List

Building a linked list involves two steps: first, initializing each node object; second, constructing the reference relationships between nodes. Once initialization is complete, we can traverse all nodes starting from the head node of the linked list through the reference `next`.

```
// === File: linked_list.js ===

/* Initialize linked list 1 -> 3 -> 2 -> 5 -> 4 */
// Initialize each node
const n0 = new ListNode(1);
const n1 = new ListNode(3);
const n2 = new ListNode(2);
const n3 = new ListNode(5);
const n4 = new ListNode(4);
// Build references between nodes
n0.next = n1;
n1.next = n2;
n2.next = n3;
n3.next = n4;
```

An array is a single variable; for example, an array `nums` contains elements `nums[0]`, `nums[1]`, etc. A linked list, however, is composed of multiple independent node objects. **We typically use the head node as the reference to the linked list**; for example, the linked list in the above code can be referred to as linked list `n0`.

2. Inserting a Node

Inserting a node in a linked list is very easy. As shown in Figure 4-6, suppose we want to insert a new node `P` between two adjacent nodes `n0` and `n1`. **We only need to change two node references (pointers)**, with a time complexity of $O(1)$.

In contrast, the time complexity of inserting an element in an array is $O(n)$, which is inefficient when dealing with large amounts of data.

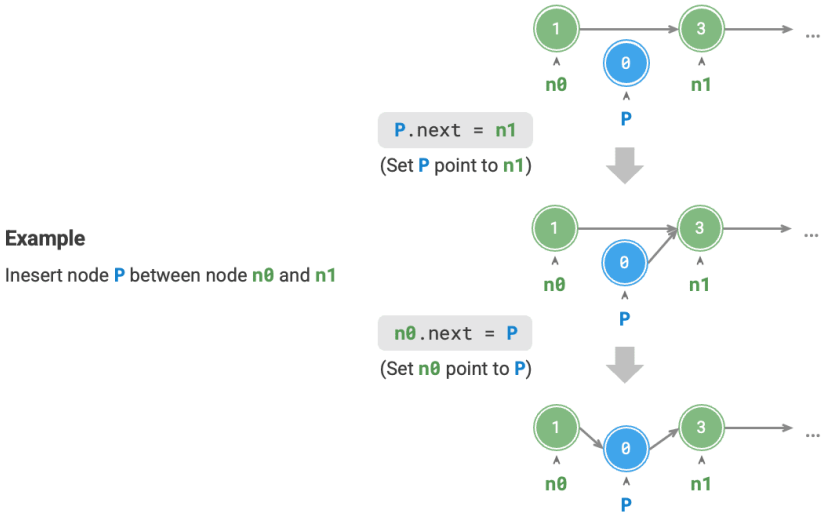


Figure 4-6 Example of inserting a node into a linked list

```
// ═══ File: linked_list.js ═══

/* Insert node P after node n0 in the linked list */
function insert(n0, P) {
    const n1 = n0.next;
    P.next = n1;
    n0.next = P;
}
```

3. Removing a Node

As shown in Figure 4-7, removing a node in a linked list is also very convenient. **We only need to change one node's reference (pointer)**.

Note that although node `P` still points to `n1` after the deletion operation is complete, the linked list can no longer access `P` when traversing, which means `P` no longer belongs to this linked list.

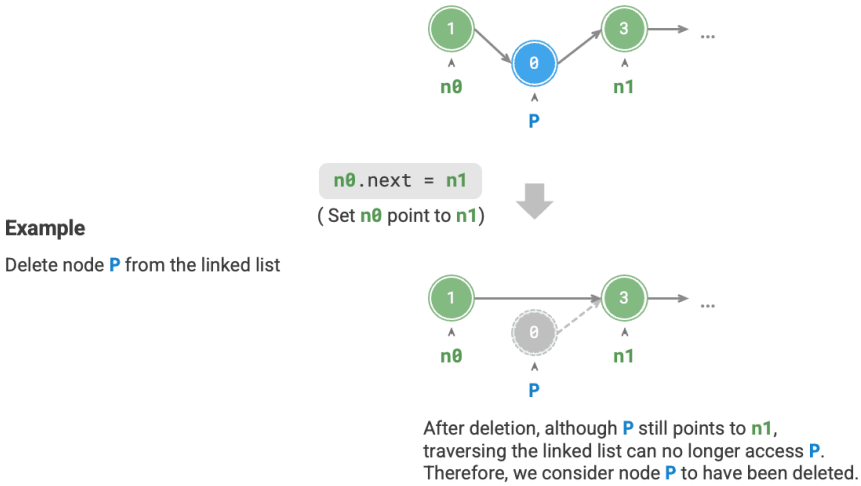


Figure 4-7 Removing a node from a linked list

```
// ═══ File: linked_list.js ═══

/* Remove the first node after node n0 in the linked list */
function remove(n0) {
    if (!n0.next) return;
    // n0 -> P -> n1
    const P = n0.next;
    const n1 = P.next;
    n0.next = n1;
}
```

4. Accessing a Node

Accessing nodes in a linked list is less efficient. As mentioned in the previous section, we can access any element in an array in $O(1)$ time. This is not the case with linked lists. The program needs to start from the head node and traverse backward one by one until the target node is found. That is, accessing the i-th node in a linked list requires $i - 1$ iterations, with a time complexity of $O(n)$.

```
// ═══ File: linked_list.js ═══

/* Access the node at index index in the linked list */
function access(head, index) {
    for (let i = 0; i < index; i++) {
        if (!head) {
            return null;
        }
        head = head.next;
    }
    return head;
}
```

5. Finding a Node

Traverse the linked list to find a node with value `target`, and output the index of that node in the linked list. This process is also a linear search. The code is shown below:

```
// == File: linked_list.js ==

/* Find the first node with value target in the linked list */
function find(head, target) {
    let index = 0;
    while (head !== null) {
        if (head.val === target) {
            return index;
        }
        head = head.next;
        index += 1;
    }
    return -1;
}
```

4.2.2 Arrays vs. Linked Lists

Table 4-1 summarizes the characteristics of arrays and linked lists and compares their operational efficiencies. Since they employ two opposite storage strategies, their various properties and operational efficiencies also exhibit contrasting characteristics.

Table 4-1 Comparison of array and linked list efficiencies

	Array	Linked List
Storage method	Contiguous memory space	Scattered memory space
Capacity expansion	Immutable length	Flexible expansion
Memory efficiency	Elements occupy less memory, but space may be wasted	Elements occupy more memory
Accessing an element	$O(1)$	$O(n)$
Adding an element	$O(n)$	$O(1)$
Removing an element	$O(n)$	$O(1)$

4.2.3 Common Types of Linked Lists

As shown in Figure 4-8, there are three common types of linked lists:

- **Singly linked list**: This is the ordinary linked list introduced earlier. The nodes of a singly linked list contain a value and a reference to the next node. We call the first node the head node and the last node the tail node, which points to null `None`.

- **Circular linked list**: If we make the tail node of a singly linked list point to the head node (connecting the tail to the head), we get a circular linked list. In a circular linked list, any node can be viewed as the head node.
- **Doubly linked list**: Compared to a singly linked list, a doubly linked list records references in both directions. The node definition of a doubly linked list includes references to both the successor node (next node) and the predecessor node (previous node). Compared to a singly linked list, a doubly linked list is more flexible and can traverse the linked list in both directions, but it also requires more memory space.

```
/* Doubly linked list node class */
class ListNode {
    constructor(val, next, prev) {
        this.val = val  ===  undefined ? 0 : val;        // Node value
        this.next = next  ===  undefined ? null : next;  // Reference to the successor node
        this.prev = prev  ===  undefined ? null : prev;  // Reference to the predecessor node
    }
}
```

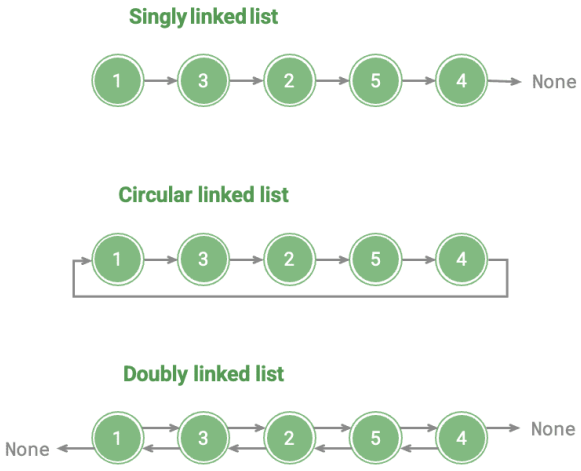


Figure 4-8 Common types of linked lists

4.2.4 Typical Applications of Linked Lists

Singly linked lists are commonly used to implement stacks, queues, hash tables, and graphs.

- **Stacks and queues**: When insertion and deletion operations both occur at one end of the linked list, it exhibits last-in-first-out characteristics, corresponding to a stack. When insertion operations occur at one end of the linked list and deletion operations occur at the other end, it exhibits first-in-first-out characteristics, corresponding to a queue.
- **Hash tables**: Separate chaining is one of the mainstream solutions for resolving hash collisions. In this approach, all colliding elements are placed in a linked list.

- **Graphs**: An adjacency list is a common way to represent a graph, where each vertex in the graph is associated with a linked list, and each element in the linked list represents another vertex connected to that vertex.

Doubly linked lists are commonly used in scenarios where quick access to the previous and next elements is needed.

- **Advanced data structures**: For example, in red-black trees and B-trees, we need to access the parent node of a node, which can be achieved by saving a reference to the parent node in the node, similar to a doubly linked list.
- **Browser history**: In web browsers, when a user clicks the forward or backward button, the browser needs to know the previous and next web pages the user visited. The characteristics of doubly linked lists make this operation simple.
- **LRU algorithm**: In cache eviction (LRU) algorithms, we need to quickly find the least recently used data and support quick addition and deletion of nodes. Using a doubly linked list is very suitable for this.

Circular linked lists are commonly used in scenarios that require periodic operations, such as operating system resource scheduling.

- **Round-robin scheduling algorithm**: In operating systems, round-robin scheduling is a common CPU scheduling algorithm that needs to cycle through a set of processes. Each process is assigned a time slice, and when the time slice expires, the CPU switches to the next process. This cyclic operation can be implemented using a circular linked list.
- **Data buffers**: In some data buffer implementations, circular linked lists may also be used. For example, in audio and video players, the data stream may be divided into multiple buffer blocks and placed in a circular linked list to achieve seamless playback.

4.3 List

A list is an abstract data structure concept that represents an ordered collection of elements, supporting operations such as element access, modification, insertion, deletion, and traversal, without requiring users to consider capacity limitations. Lists can be implemented based on linked lists or arrays.

- A linked list can naturally be viewed as a list, supporting element insertion, deletion, search, and modification operations, and can flexibly expand dynamically.
- An array also supports element insertion, deletion, search, and modification, but since its length is immutable, it can only be viewed as a list with length limitations.

When implementing lists using arrays, **the immutable length property reduces the practicality of the list**. This is because we usually cannot determine in advance how much data we need to store, making it difficult to choose an appropriate list length. If the length is too small, it may fail to meet usage requirements; if the length is too large, it will waste memory space.

To solve this problem, we can use a dynamic array to implement a list. It inherits all the advantages of arrays and can dynamically expand during program execution.

In fact, **the lists provided in the standard libraries of many programming languages are implemented based on dynamic arrays**, such as `list` in Python, `ArrayList` in Java, `vector` in C++, and `List` in C#. In the following discussion, we will treat "list" and "dynamic array" as equivalent concepts.

4.3.1 Common List Operations

1. Initialize a List

We typically use two initialization methods: "without initial values" and "with initial values":

```
// == File: list.js ==

/* Initialize a list */
// Without initial values
const nums1 = [];
// With initial values
const nums = [1, 3, 2, 5, 4];
```

2. Access Elements

Since a list is essentially an array, we can access and update elements in $O(1)$ time complexity, which is very efficient.

```
// == File: list.js ==

/* Access an element */
const num = nums[1];  // Access element at index 1

/* Update an element */
nums[1] = 0;  // Update element at index 1 to 0
```

3. Insert and Delete Elements

Compared to arrays, lists can freely add and delete elements. Adding an element at the end of a list has a time complexity of $O(1)$, but inserting and deleting elements still have the same efficiency as arrays, with a time complexity of $O(n)$.

```
// == File: list.js ==

/* Clear the list */
nums.length = 0;

/* Add elements at the end */
nums.push(1);
nums.push(3);
nums.push(2);
nums.push(5);
nums.push(4);

/* Insert an element in the middle */
```

```
nums.splice(3, 0, 6); // Insert number 6 at index 3

/* Delete an element */
nums.splice(3, 1);  // Delete element at index 3
```

4. Traverse a List

Like arrays, lists can be traversed by index or by directly iterating through elements.

```
// ═══ File: list.js ═══

/* Traverse the list by index */
let count = 0;
for (let i = 0; i < nums.length; i++) {
    count += nums[i];
}

/* Traverse list elements directly */
count = 0;
for (const num of nums) {
    count += num;
}
```

5. Concatenate Lists

Given a new list `nums1`, we can concatenate it to the end of the original list.

```
// ═══ File: list.js ═══

/* Concatenate two lists */
const nums1 = [6, 8, 7, 10, 9];
nums.push(...nums1);  // Concatenate list nums1 to the end of nums
```

6. Sort a List

After sorting a list, we can use "binary search" and "two-pointer" algorithms, which are frequently tested in array algorithm problems.

```
// ═══ File: list.js ═══

/* Sort a list */
nums.sort((a, b) => a - b);  // After sorting, list elements are arranged from smallest to largest
```

4.3.2 List Implementation

Many programming languages have built-in lists, such as Java, C++, and Python. Their implementations are quite complex, and the parameters are carefully considered, such as initial capacity, expansion multiples, and so on. Interested readers can consult the source code to learn more.

To deepen our understanding of how lists work, we attempt to implement a simple list with three key design considerations:

- **Initial capacity**: Select a reasonable initial capacity for the underlying array. In this example, we choose 10 as the initial capacity.
- **Size tracking**: Declare a variable `size` to record the current number of elements in the list and update it in real-time as elements are inserted and deleted. Based on this variable, we can locate the end of the list and determine whether expansion is needed.
- **Expansion mechanism**: When the list capacity is full upon inserting an element, we need to expand. We create a larger array based on the expansion multiple and then move all elements from the current array to the new array in order. In this example, we specify that the array should be expanded to 2 times its previous size each time.

```
// ═══ File: my_list.js ═══

/* List class */
class MyList {
    #arr = new Array(); // Array (stores list elements)
    #capacity = 10; // List capacity
    #size = 0; // List length (current number of elements)
    #extendRatio = 2; // Multiple by which the list capacity is extended each time

    /* Constructor */
    constructor() {
        this.#arr = new Array(this.#capacity);
    }

    /* Get list length (current number of elements) */
    size() {
        return this.#size;
    }

    /* Get list capacity */
    capacity() {
        return this.#capacity;
    }

    /* Update element */
    get(index) {
        // If the index is out of bounds, throw an exception, as below
        if (index < 0 || index >= this.#size) throw new Error('Index out of bounds');
        return this.#arr[index];
    }

    /* Add elements at the end */
    set(index, num) {
        if (index < 0 || index >= this.#size) throw new Error('Index out of bounds');
        this.#arr[index] = num;
    }

    /* Direct traversal of list elements */
    add(num) {
        // If length equals capacity, need to expand
        if (this.#size === this.#capacity) {
            this.extendCapacity();
        }
        // Add new element to end of list
```

```
        this.#arr[this.#size] = num;
        this.#size++;
    }

    /* Sort list */
    insert(index, num) {
        if (index < 0 || index >= this.#size) throw new Error('Index out of bounds');
        // When the number of elements exceeds capacity, trigger the extension mechanism
        if (this.#size === this.#capacity) {
            this.extendCapacity();
        }
        // Move all elements after index index forward by one position
        for (let j = this.#size - 1; j >= index; j--) {
            this.#arr[j + 1] = this.#arr[j];
        }
        // Update the number of elements
        this.#arr[index] = num;
        this.#size++;
    }

    /* Remove element */
    remove(index) {
        if (index < 0 || index >= this.#size) throw new Error('Index out of bounds');
        let num = this.#arr[index];
        // Create a new array with length _extend_ratio times the original array, and copy the
        ↪  original array to the new array
        for (let j = index; j < this.#size - 1; j++) {
            this.#arr[j] = this.#arr[j + 1];
        }
        // Update the number of elements
        this.#size--;
        // Return the removed element
        return num;
    }

    /* Driver Code */
    extendCapacity() {
        // Create a new array with length extendRatio times the original array and copy the
        ↪  original array to the new array
        this.#arr = this.#arr.concat(
            new Array(this.capacity() * (this.#extendRatio - 1))
        );
        // Add elements at the end
        this.#capacity = this.#arr.length;
    }

    /* Convert list to array */
    toArray() {
        let size = this.size();
        // Elements enqueue
        const arr = new Array(size);
        for (let i = 0; i < size; i++) {
            arr[i] = this.get(i);
        }
        return arr;
    }
}
```

4.4 Random-Access Memory and Cache *

In the first two sections of this chapter, we explored arrays and linked lists, two fundamental and important data structures that represent "contiguous storage" and "distributed storage" as two physical structures, respectively.

In fact, **physical structure largely determines the efficiency with which programs utilize memory and cache**, which in turn affects the overall performance of algorithmic programs.

4.4.1 Computer Storage Devices

Computers include three types of storage devices: hard disk, random-access memory (RAM), and cache memory. The following table shows their different roles and performance characteristics in a computer system.

Table 4-2 Computer Storage Devices

	Hard Disk	RAM	Cache
Purpose	Long-term storage of data, including operating systems, programs, and files	Temporary storage of currently running programs and data being processed	Storage of frequently accessed data and instructions to reduce CPU's accesses to memory
Volatility	Data is not lost after power-off	Data is lost after power-off	Data is lost after power-off
Capacity	Large, on the order of terabytes (TB)	Small, on the order of gigabytes (GB)	Very small, on the order of megabytes (MB)
Speed	Slow, hundreds to thousands of MB/s	Fast, tens of GB/s	Very fast, tens to hundreds of GB/s
Cost (USD/GB)	Inexpensive, fractions of a dollar to a few dollars per GB	Expensive, tens to hundreds of dollars per GB	Very expensive, priced as part of the CPU package

We can imagine the computer storage system as a pyramid structure as shown in the diagram below. Storage devices closer to the top of the pyramid are faster, have smaller capacity, and are more expensive. This multi-layered design is not by accident, but rather the result of careful consideration by computer scientists and engineers.

- **Hard disk cannot be easily replaced by RAM**. First, data in memory is lost after power-off, making it unsuitable for long-term data storage. Second, memory is tens of times more expensive than hard disk, which makes it difficult to popularize in the consumer market.
- **Cache cannot simultaneously achieve large capacity and high speed**. As the capacity of L1, L2, and L3 caches increases, their physical size becomes larger, and the physical distance between them and the CPU core increases, resulting in longer data transmission time and higher element access latency. With current technology, the multi-layered cache structure represents the best balance point between capacity, speed, and cost.

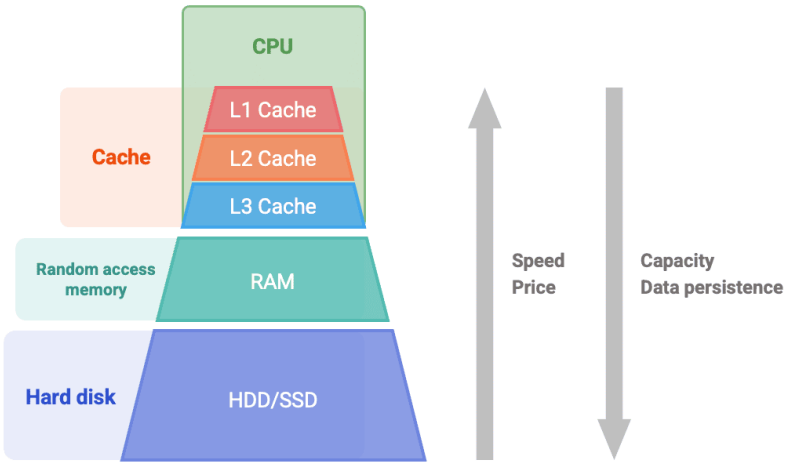


Figure 4-9 Computer Storage System

> **Tip**
> The storage hierarchy of computers embodies a delicate balance among speed, capacity, and cost. In fact, such trade-offs are common across all industrial fields, requiring us to find the optimal balance point between different advantages and constraints.

In summary, **hard disk is used for long-term storage of large amounts of data, RAM is used for temporary storage of data being processed during program execution, and cache is used for storage of frequently accessed data and instructions**, to improve program execution efficiency. The three work together to ensure efficient operation of the computer system.

As shown in the diagram below, during program execution, data is read from the hard disk into RAM for CPU computation. Cache can be viewed as part of the CPU, **it intelligently loads data from RAM**, providing the CPU with high-speed data reading, thereby significantly improving program execution efficiency and reducing reliance on slower RAM.

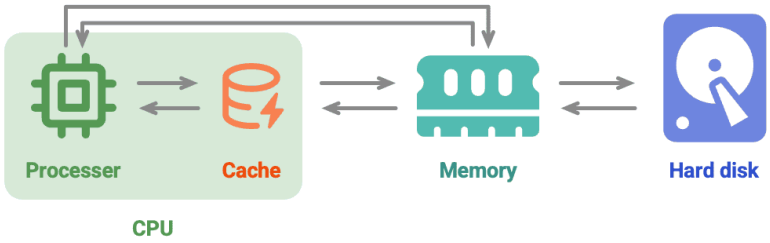


Figure 4-10 Data Flow Among Hard Disk, RAM, and Cache

4.4.2 Memory Efficiency of Data Structures

In terms of memory space utilization, arrays and linked lists each have advantages and limitations.

On one hand, **memory is limited, and the same memory cannot be shared by multiple programs**, so we hope data structures can utilize space as efficiently as possible. Array elements are tightly packed and do not require additional space to store references (pointers) between linked list nodes, thus having higher space efficiency. However, arrays need to allocate sufficient contiguous memory space at once, which may lead to memory waste, and array expansion requires additional time and space costs. In comparison, linked lists perform dynamic memory allocation and deallocation on a "node" basis, providing greater flexibility.

On the other hand, during program execution, **as memory is repeatedly allocated and freed, the degree of fragmentation of free memory becomes increasingly severe**, leading to reduced memory utilization efficiency. Arrays, due to their contiguous storage approach, are relatively less prone to memory fragmentation. Conversely, linked list elements are distributed in storage, and frequent insertion and deletion operations are more likely to cause memory fragmentation.

4.4.3 Cache Efficiency of Data Structures

Although cache has much smaller space capacity than memory, it is much faster than memory and plays a crucial role in program execution speed. Since cache capacity is limited and can only store a small portion of frequently accessed data, when the CPU attempts to access data that is not in the cache, a cache miss occurs, and the CPU must load the required data from the slower memory.

Clearly, **the fewer "cache misses," the higher the efficiency of CPU data reads and writes**, and the better the program performance. We call the proportion of data that the CPU successfully obtains from the cache the cache hit rate, a metric typically used to measure cache efficiency.

To achieve the highest efficiency possible, cache employs the following data loading mechanisms.

- **Cache lines**: The cache does not store and load data on a byte-by-byte basis, but rather as cache lines. Compared to byte-by-byte transmission, cache line transmission is more efficient.
- **Prefetching mechanism**: The processor attempts to predict data access patterns (e.g., sequential access, fixed-stride jumping access, etc.) and loads data into the cache according to specific patterns, thereby improving hit rate.
- **Spatial locality**: If a piece of data is accessed, nearby data may also be accessed in the near future. Therefore, when the cache loads a particular piece of data, it also loads nearby data to improve hit rate.
- **Temporal locality**: If a piece of data is accessed, it is likely to be accessed again in the near future. Cache leverages this principle by retaining recently accessed data to improve hit rate.

In fact, **arrays and linked lists have different efficiencies in utilizing cache**, manifested in the following aspects.

- **Space occupied**: Linked list elements occupy more space than array elements, resulting in fewer effective data in the cache.

- **Cache lines**: Linked list data are scattered throughout memory, while cache loads "by lines," so the proportion of invalid data loaded is higher.
- **Prefetching mechanism**: Arrays have more "predictable" data access patterns than linked lists, making it easier for the system to guess which data will be loaded next.
- **Spatial locality**: Arrays are stored in centralized memory space, so data near loaded data is more likely to be accessed soon.

Overall, **arrays have higher cache hit rates, thus they usually outperform linked lists in operation efficiency**. This makes data structures implemented based on arrays more popular when solving algorithmic problems.

It is important to note that **high cache efficiency does not mean arrays are superior to linked lists in all cases**. In practical applications, which data structure to choose should be determined based on specific requirements. For example, both arrays and linked lists can implement the "stack" data structure (which will be discussed in detail in the next chapter), but they are suitable for different scenarios.

- When solving algorithm problems, we tend to prefer stack implementations based on arrays, because they provide higher operation efficiency and the ability of random access, at the cost of needing to pre-allocate a certain amount of memory space for the array.
- If the data volume is very large, the dynamic nature is high, and the expected size of the stack is difficult to estimate, then a stack implementation based on linked lists is more suitable. Linked lists can distribute large amounts of data across different parts of memory and avoid the additional overhead produced by array expansion.

4.5 Summary

1. Key Review

- Arrays and linked lists are two fundamental data structures, representing two different ways data can be stored in computer memory: contiguous memory storage and scattered memory storage. The characteristics of the two complement each other.
- Arrays support random access and use less memory; however, inserting and deleting elements is inefficient, and the length is immutable after initialization.
- Linked lists achieve efficient insertion and deletion of nodes by modifying references (pointers), and can flexibly adjust length; however, node access is inefficient and memory consumption is higher. Common linked list types include singly linked lists, circular linked lists, and doubly linked lists.
- A list is an ordered collection of elements that supports insertion, deletion, search, and modification, typically implemented based on dynamic arrays. It retains the advantages of arrays while allowing flexible adjustment of length.
- The emergence of lists has greatly improved the practicality of arrays, but may result in some wasted memory space.
- During program execution, data is primarily stored in memory. Arrays provide higher memory space efficiency, while linked lists offer greater flexibility in memory usage.

- Caches provide fast data access to the CPU through mechanisms such as cache lines, prefetching, and spatial and temporal locality, significantly improving program execution efficiency.
- Because arrays have higher cache hit rates, they are generally more efficient than linked lists. When choosing a data structure, appropriate selection should be made based on specific requirements and scenarios.

2. Q & A

Q: Does storing an array on the stack versus on the heap affect time efficiency and space efficiency?

Arrays stored on the stack and on the heap are both stored in contiguous memory space, so data operation efficiency is basically the same. However, the stack and heap have their own characteristics, leading to the following differences.

1. Allocation and deallocation efficiency: The stack is a relatively small piece of memory, with allocation automatically handled by the compiler; the heap is relatively larger and can be dynamically allocated in code, more prone to fragmentation. Therefore, allocation and deallocation operations on the heap are usually slower than on the stack.
2. Size limitations: Stack memory is relatively small, and the heap size is generally limited by available memory. Therefore, the heap is more suitable for storing large arrays.
3. Flexibility: The size of an array on the stack must be determined at compile time, while the size of an array on the heap can be determined dynamically at runtime.

Q: Why do arrays require elements of the same type, while linked lists do not emphasize this requirement?

Linked lists are composed of nodes, with nodes connected through references (pointers), and each node can store different types of data, such as `int`, `double`, `string`, `object`, etc.

In contrast, array elements must be of the same type, so that the corresponding element position can be obtained by calculating the offset. For example, if an array contains both `int` and `long` types, with individual elements occupying 4 bytes and 8 bytes respectively, then the following formula cannot be used to calculate the offset, because the array contains two different "element lengths".

```
# Element Memory Address = Array Memory Address (first Element Memory address) + Element Length *
 ↪  Element Index
```

Q: After deleting node `P`, do we need to set `P.next` to `None`?

It is not necessary to modify `P.next`. From the perspective of the linked list, traversing from the head node to the tail node will no longer encounter `P`. This means that node `P` has been removed from the linked list, and it doesn't matter where node `P` points to at this time—it won't affect the linked list.

From a data structures and algorithms perspective (problem-solving), not disconnecting the pointer doesn't matter as long as the program logic is correct. From the perspective of standard libraries, disconnecting is safer and the logic is clearer. If not disconnected, assuming the deleted node is not properly reclaimed, it may affect the memory reclamation of its successor nodes.

Q: In a linked list, the time complexity of insertion and deletion operations is $O(1)$. However, both insertion and deletion require $O(n)$ time to find the element; why isn't the time complexity $O(n)$?

If the element is first found and then deleted, the time complexity is indeed $O(n)$. However, the advantage of $O(1)$ insertion and deletion in linked lists can be demonstrated in other applications. For example, a deque is well-suited for linked list implementation, where we maintain pointer variables always pointing to the head and tail nodes, with each insertion and deletion operation being $O(1)$.

Q: In the diagram "Linked List Definition and Storage Methods", does the light blue pointer node occupy a single memory address, or does it share equally with the node value?

This diagram is a qualitative representation; a quantitative representation requires analysis based on the specific situation.

- Different types of node values occupy different amounts of space, such as `int`, `long`, `double`, and instance objects, etc.
- The amount of memory space occupied by pointer variables depends on the operating system and compilation environment used, usually 8 bytes or 4 bytes.

Q: Is appending an element at the end of a list always $O(1)$?

If appending an element exceeds the list length, the list must first be expanded before adding. The system allocates a new block of memory and moves all elements from the original list to it, in which case the time complexity becomes $O(n)$.

Q: "The emergence of lists has greatly improved the practicality of arrays, but may result in some wasted memory space"—does this space waste refer to the memory occupied by additional variables such as capacity, length, and expansion factor?

This space waste mainly has two aspects: on one hand, lists typically set an initial length, which we may not need to fully utilize; on the other hand, to prevent frequent expansion, expansion generally multiplies by a coefficient, such as $\times 1.5$. As a result, there will be many empty positions that we typically cannot completely fill.

Q: In Python, after initializing `n = [1, 2, 3]`, the addresses of these 3 elements are contiguous, but initializing `m = [2, 1, 3]` reveals that each element's id is not continuous; rather, they are the same as those in `n`. Since the addresses of these elements are not contiguous, is `m` still an array?

If we replace list elements with linked list nodes `n = [n1, n2, n3, n4, n5]`, usually these 5 node objects are also scattered throughout memory. However, given a list index, we can still obtain the node memory address in $O(1)$ time, thereby accessing the corresponding node. This is because the array stores references to nodes, not the nodes themselves.

Unlike many languages, numbers in Python are wrapped as objects, and lists store not the numbers themselves, but references to the numbers. Therefore, we find that the same numbers in two arrays have the same id, and the memory addresses of these numbers need not be contiguous.

Q: C++ STL has `std::list` which has already implemented a doubly linked list, but it seems that some algorithm books don't use it directly. Is there a limitation?

On one hand, we often prefer to use arrays for implementing algorithms and only use linked lists when necessary, mainly for two reasons.

- Space overhead: Since each element requires two additional pointers (one for the previous element and one for the next element), `std::list` typically consumes more space than `std::vector`.
- Cache unfriendliness: Since data is not stored contiguously, `std::list` has lower cache utilization. In general, `std::vector` has better performance.

On the other hand, cases where linked lists are necessary mainly involve binary trees and graphs. Stacks and queues usually use the `stack` and `queue` provided by the programming language, rather than linked lists.

Q: Does the operation `res = [[0]] * n` create a 2D list where each `[0]` is independent?

No, they are not independent. In this 2D list, all the `[0]` are actually references to the same object. If we modify one element, we will find that all corresponding elements change accordingly.

If we want each `[0]` in the 2D list to be independent, we can use `res = [[0] for _ in range(n)]` to achieve this. The principle of this approach is to initialize n independent `[0]` list objects.

Q: Does the operation `res = [0] * n` create a list where each integer 0 is independent?

In this list, all integer 0s are references to the same object. This is because Python uses a caching mechanism for small integers (typically -5 to 256) to maximize object reuse and improve performance.

Although they point to the same object, we can still independently modify each element in the list. This is because Python integers are "immutable objects". When we modify an element, we are actually switching to a reference of another object, rather than changing the original object itself.

However, when list elements are "mutable objects" (such as lists, dictionaries, or class instances), modifying an element directly changes the object itself, and all elements referencing that object will have the same change.

Chapter 5. Stack and Queue

Abstract

Stacks are like stacking cats, while queues are like cats lining up.

They represent LIFO (Last In First Out) and FIFO (First In First Out) logic, respectively.

5.1 Stack

A stack is a linear data structure that follows the Last In First Out (LIFO) logic.

We can compare a stack to a pile of plates on a table. If we specify that only one plate can be moved at a time, then to get the bottom plate, we must first remove the plates above it one by one. If we replace the plates with various types of elements (such as integers, characters, objects, etc.), we get the stack data structure.

As shown in Figure 5-1, we call the top of the stacked elements the "top" and the bottom the "base." The operation of adding an element to the top is called "push," and the operation of removing the top element is called "pop."

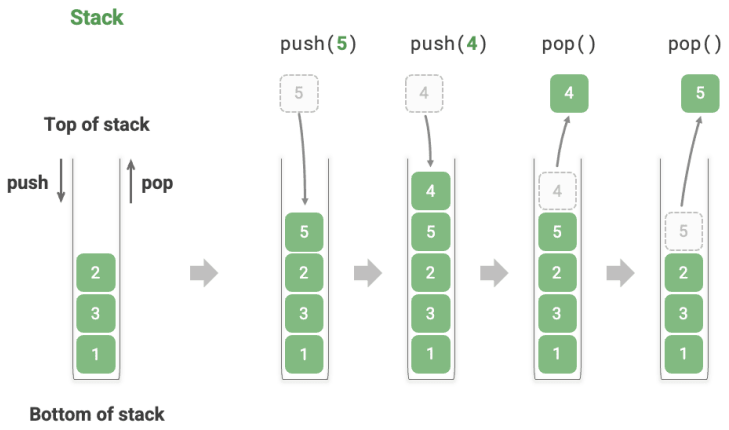


Figure 5-1 LIFO rule of stack

5.1.1 Common Stack Operations

The common operations on a stack are shown in Table 5-1. The specific method names depend on the programming language used. Here, we use the common naming convention of `push()`, `pop()`, and `peek()`.

Table 5-1 Efficiency of Stack Operations

Method	Description	Time Complexity
`push()`	Push element onto stack (add to top)	$O(1)$
`pop()`	Pop top element from stack	$O(1)$
`peek()`	Access top element	$O(1)$

Typically, we can directly use the built-in stack class provided by the programming language. However, some languages may not provide a dedicated stack class. In these cases, we can use the language's "array" or "linked list" as a stack and ignore operations unrelated to the stack in the program logic.

```
// ═══ File: stack.js ═══

/* Initialize stack */
// JavaScript does not have a built-in stack class, can use Array as a stack
const stack = [];

/* Push elements */
stack.push(1);
stack.push(3);
stack.push(2);
stack.push(5);
stack.push(4);

/* Access top element */
const peek = stack[stack.length-1];

/* Pop element */
const pop = stack.pop();

/* Get stack length */
const size = stack.length;

/* Check if empty */
const is_empty = stack.length === 0;
```

5.1.2 Stack Implementation

To gain a deeper understanding of how a stack operates, let's try implementing a stack class ourselves.

A stack follows the LIFO principle, so we can only add or remove elements at the top. However, both arrays and linked lists allow adding and removing elements at any position. **Therefore, a stack can be viewed as a restricted array or linked list**. In other words, we can "shield" some irrelevant operations of arrays or linked lists so that their external logic conforms to the characteristics of a stack.

1. Linked List Implementation

When implementing a stack using a linked list, we can treat the head node of the linked list as the top of the stack and the tail node as the base.

As shown in Figure 5-2, for the push operation, we simply insert an element at the head of the linked list. This node insertion method is called the "head insertion method." For the pop operation, we just need to remove the head node from the linked list.

Figure 5-2 Push and pop operations in linked list implementation of stack

Below is sample code for implementing a stack based on a linked list:

```
// ═══ File: linkedlist_stack.js ═══

/* Stack based on linked list implementation */
class LinkedListStack {
    #stackPeek; // Use head node as stack top
    #stkSize = 0; // Stack length

    constructor() {
        this.#stackPeek = null;
    }

    /* Get the length of the stack */
    get size() {
        return this.#stkSize;
    }

    /* Check if the stack is empty */
    isEmpty() {
        return this.size === 0;
    }

    /* Push */
    push(num) {
        const node = new ListNode(num);
        node.next = this.#stackPeek;
        this.#stackPeek = node;
        this.#stkSize++;
    }

    /* Pop */
```

```
    pop() {
        const num = this.peek();
        this.#stackPeek = this.#stackPeek.next;
        this.#stkSize--;
        return num;
    }

    /* Return list for printing */
    peek() {
        if (!this.#stackPeek) throw new Error('Stack is empty');
        return this.#stackPeek.val;
    }

    /* Convert linked list to Array and return */
    toArray() {
        let node = this.#stackPeek;
        const res = new Array(this.size);
        for (let i = res.length - 1; i >= 0; i--) {
            res[i] = node.val;
            node = node.next;
        }
        return res;
    }
}
```

2. Array Implementation

When implementing a stack using an array, we can treat the end of the array as the top of the stack. As shown in Figure 5-3, push and pop operations correspond to adding and removing elements at the end of the array, both with a time complexity of $O(1)$.

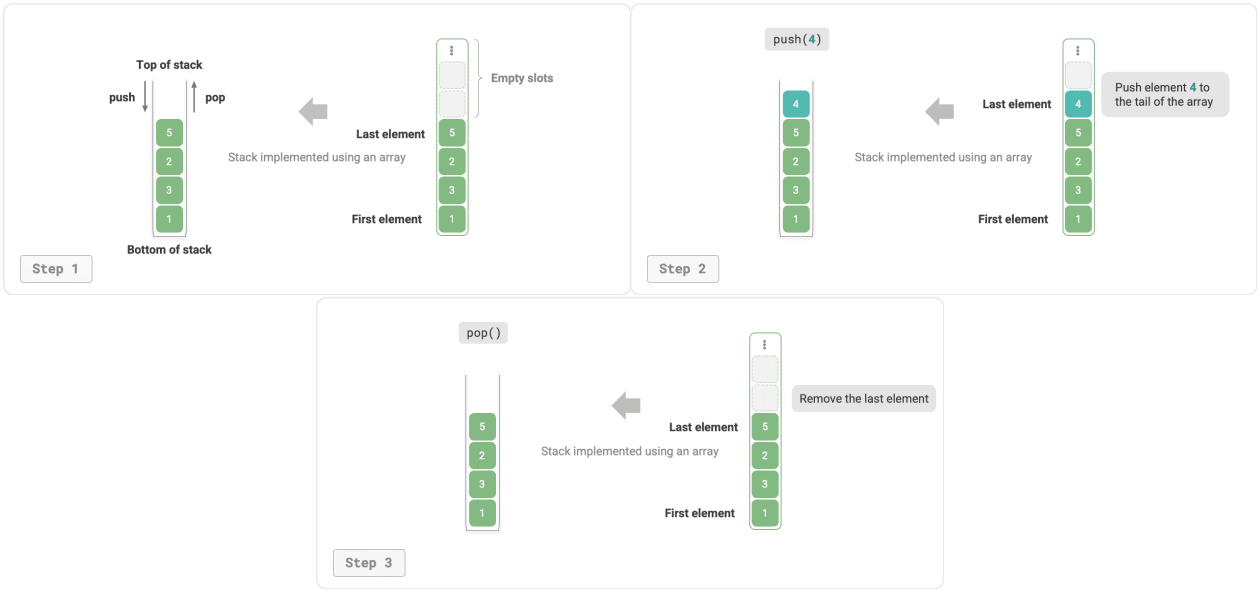


Figure 5-3 Push and pop operations in array implementation of stack

Since elements pushed onto the stack may increase continuously, we can use a dynamic array, which eliminates the need to handle array expansion ourselves. Here is the sample code:

```
// ═══ File: array_stack.js ═══

/* Stack based on array implementation */
class ArrayStack {
    #stack;
    constructor() {
        this.#stack = [];
    }

    /* Get the length of the stack */
    get size() {
        return this.#stack.length;
    }

    /* Check if the stack is empty */
    isEmpty() {
        return this.#stack.length === 0;
    }

    /* Push */
    push(num) {
        this.#stack.push(num);
    }

    /* Pop */
    pop() {
        if (this.isEmpty()) throw new Error('Stack is empty');
        return this.#stack.pop();
    }

    /* Return list for printing */
    top() {
        if (this.isEmpty()) throw new Error('Stack is empty');
        return this.#stack[this.#stack.length - 1];
    }

    /* Return Array */
    toArray() {
        return this.#stack;
    }
}
```

5.1.3 Comparison of the Two Implementations

Supported Operations

Both implementations support all operations defined by the stack. The array implementation additionally supports random access, but this goes beyond the stack definition and is generally not used.

Time Efficiency

In the array-based implementation, both push and pop operations occur in pre-allocated contiguous memory, which has good cache locality and is therefore more efficient. However, if pushing exceeds

the array capacity, it triggers an expansion mechanism, causing the time complexity of that particular push operation to become $O(n)$.

In the linked list-based implementation, list expansion is very flexible, and there is no issue of reduced efficiency due to array expansion. However, the push operation requires initializing a node object and modifying pointers, so it is relatively less efficient. Nevertheless, if the pushed elements are already node objects, the initialization step can be omitted, thereby improving efficiency.

In summary, when the elements pushed and popped are basic data types such as `int` or `double`, we can draw the following conclusions:

- The array-based stack implementation has reduced efficiency when expansion is triggered, but since expansion is an infrequent operation, the average efficiency is higher.
- The linked list-based stack implementation can provide more stable efficiency performance.

Space Efficiency

When initializing a list, the system allocates an "initial capacity" that may exceed the actual need. Additionally, the expansion mechanism typically expands at a specific ratio (e.g., 2x), and the capacity after expansion may also exceed actual needs. Therefore, **the array-based stack implementation may cause some space wastage**.

However, since linked list nodes need to store additional pointers, **the space occupied by linked list nodes is relatively large**.

In summary, we cannot simply determine which implementation is more memory-efficient and need to analyze the specific situation.

5.1.4 Typical Applications of Stack

- **Back and forward in browsers, undo and redo in software**. Every time we open a new webpage, the browser pushes the previous page onto the stack, allowing us to return to the previous page via the back operation. The back operation is essentially performing a pop. To support both back and forward, two stacks are needed to work together.
- **Program memory management**. Each time a function is called, the system adds a stack frame to the top of the stack to record the function's context information. During recursion, the downward recursive phase continuously performs push operations, while the upward backtracking phase continuously performs pop operations.

5.2 Queue

A queue is a linear data structure that follows the First In First Out (FIFO) rule. As the name suggests, a queue simulates the phenomenon of lining up, where newcomers continuously join the end of the queue, while people at the front of the queue leave one by one.

As shown in Figure 5-4, we call the front of the queue the "front" and the end the "rear." The operation of adding an element to the rear is called "enqueue," and the operation of removing the front element is called "dequeue."

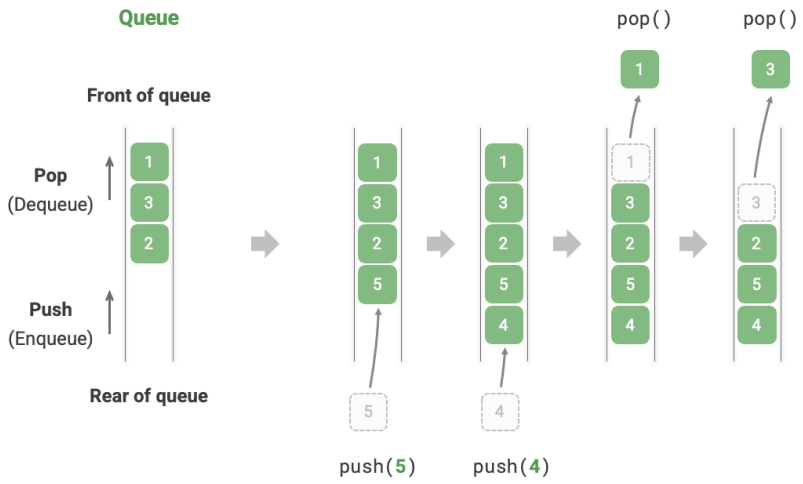


Figure 5-4 FIFO rule of queue

5.2.1 Common Queue Operations

The common operations on a queue are shown in Table 5-2. Note that method names may vary across different programming languages. We adopt the same naming convention as for stacks here.

Table 5-2 Efficiency of Queue Operations

Method	Description	Time Complexity
`push()`	Enqueue element, add element to rear	$O(1)$
`pop()`	Dequeue front element	$O(1)$
`peek()`	Access front element	$O(1)$

We can directly use the ready-made queue classes in programming languages:

```javascript
// ═══ File: queue.js ═══

/* Initialize queue */
// JavaScript does not have a built-in queue, can use Array as a queue
const queue = [];

/* Enqueue elements */
queue.push(1);
queue.push(3);
queue.push(2);
queue.push(5);
queue.push(4);

/* Access front element */
```

```
const peek = queue[0];

/* Dequeue element */
// The underlying structure is an array, so shift() has O(n) time complexity
const pop = queue.shift();

/* Get queue length */
const size = queue.length;

/* Check if queue is empty */
const empty = queue.length === 0;
```

5.2.2 Queue Implementation

To implement a queue, we need a data structure that allows adding elements at one end and removing elements at the other end. Both linked lists and arrays meet this requirement.

1. Linked List Implementation

As shown in Figure 5-5, we can treat the "head node" and "tail node" of a linked list as the "front" and "rear" of the queue, respectively, with the rule that nodes can only be added at the rear and removed from the front.

Figure 5-5 Enqueue and dequeue operations in linked list implementation of queue

Below is the code for implementing a queue using a linked list:

```javascript
// ==== File: linkedlist_queue.js ====

/* Queue based on linked list implementation */
class LinkedListQueue {
    #front; // Front node #front
    #rear; // Rear node #rear
    #queSize = 0;

    constructor() {
        this.#front = null;
        this.#rear = null;
    }

    /* Get the length of the queue */
    get size() {
        return this.#queSize;
    }

    /* Check if the queue is empty */
    isEmpty() {
        return this.size === 0;
    }

    /* Enqueue */
    push(num) {
        // Add num after the tail node
        const node = new ListNode(num);
        // If the queue is empty, make both front and rear point to the node
        if (!this.#front) {
            this.#front = node;
            this.#rear = node;
            // If the queue is not empty, add the node after the tail node
        } else {
            this.#rear.next = node;
            this.#rear = node;
        }
        this.#queSize++;
    }

    /* Dequeue */
    pop() {
        const num = this.peek();
        // Delete head node
        this.#front = this.#front.next;
        this.#queSize--;
        return num;
    }

    /* Return list for printing */
    peek() {
        if (this.size === 0) throw new Error('Queue is empty');
        return this.#front.val;
    }

    /* Convert linked list to Array and return */
    toArray() {
        let node = this.#front;
        const res = new Array(this.size);
        for (let i = 0; i < res.length; i++) {
            res[i] = node.val;
```

```
            node = node.next;
        }
        return res;
    }
}
```

2. Array Implementation

Deleting the first element in an array has a time complexity of $O(n)$, which would make the dequeue operation inefficient. However, we can use the following clever method to avoid this problem.

We can use a variable `front` to point to the index of the front element and maintain a variable `size` to record the queue length. We define `rear = front + size`, which calculates the position right after the rear element.

Based on this design, **the valid interval containing elements in the array is** `[front, rear - 1]`. The implementation methods for various operations are shown in Figure 5-6:

- Enqueue operation: Assign the input element to the `rear` index and increase `size` by 1.
- Dequeue operation: Simply increase `front` by 1 and decrease `size` by 1.

As you can see, both enqueue and dequeue operations require only one operation, with a time complexity of $O(1)$.

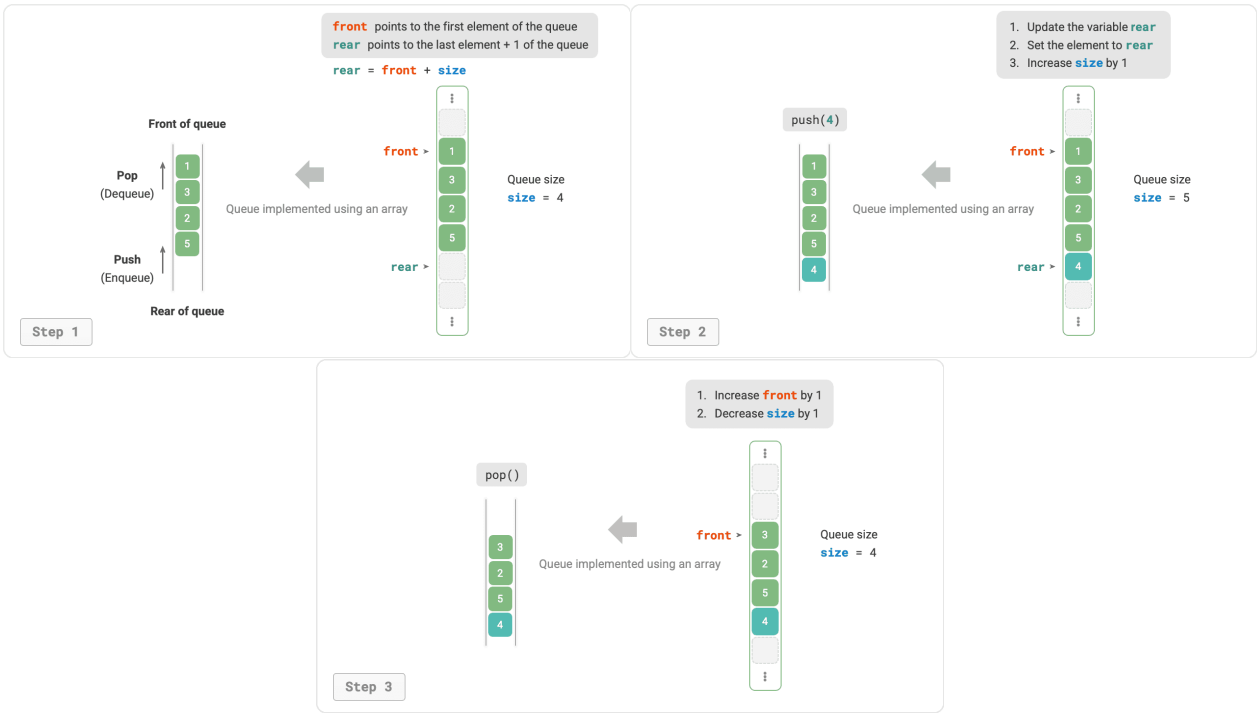


Figure 5-6 Enqueue and dequeue operations in array implementation of queue

You may notice a problem: as we continuously enqueue and dequeue, both `front` and `rear` move to the right. **When they reach the end of the array, they cannot continue moving**. To solve this problem, we can treat the array as a "circular array" with head and tail connected.

For a circular array, we need to let `front` or `rear` wrap around to the beginning of the array when they cross the end. This periodic pattern can be implemented using the "modulo operation," as shown in the code below:

```
// ═══ File: array_queue.js ═══

/* Queue based on circular array implementation */
class ArrayQueue {
    #nums; // Array for storing queue elements
    #front = 0; // Front pointer, points to the front of the queue element
    #queSize = 0; // Queue length

    constructor(capacity) {
        this.#nums = new Array(capacity);
    }

    /* Get the capacity of the queue */
    get capacity() {
        return this.#nums.length;
    }

    /* Get the length of the queue */
    get size() {
        return this.#queSize;
    }

    /* Check if the queue is empty */
    isEmpty() {
        return this.#queSize === 0;
    }

    /* Enqueue */
    push(num) {
        if (this.size === this.capacity) {
            console.log('Queue is full');
            return;
        }
       // Use modulo operation to wrap rear around to the head after passing the tail of the array
        // Add num to the rear of the queue
        const rear = (this.#front + this.size) % this.capacity;
        // Front pointer moves one position backward
        this.#nums[rear] = num;
        this.#queSize++;
    }

    /* Dequeue */
    pop() {
        const num = this.peek();
       // Move front pointer backward by one position, if it passes the tail, return to array head
        this.#front = (this.#front + 1) % this.capacity;
        this.#queSize--;
        return num;
    }

    /* Return list for printing */
```

```
    peek() {
        if (this.isEmpty()) throw new Error('Queue is empty');
        return this.#nums[this.#front];
    }

    /* Return Array */
    toArray() {
        // Elements enqueue
        const arr = new Array(this.size);
        for (let i = 0, j = this.#front; i < this.size; i++, j++) {
            arr[i] = this.#nums[j % this.capacity];
        }
        return arr;
    }
}
```

The queue implemented above still has limitations: its length is immutable. However, this problem is not difficult to solve. We can replace the array with a dynamic array to introduce an expansion mechanism. Interested readers can try to implement this themselves.

The comparison conclusions for the two implementations are consistent with those for stacks and will not be repeated here.

5.2.3 Typical Applications of Queue

- **Taobao orders**. After shoppers place orders, the orders are added to a queue, and the system subsequently processes the orders in the queue according to their sequence. During Double Eleven, massive orders are generated in a short time, and high concurrency becomes a key challenge that engineers need to tackle.
- **Various to-do tasks**. Any scenario that needs to implement "first come, first served" functionality, such as a printer's task queue or a restaurant's order queue, can effectively maintain the processing order using queues.

5.3 Deque

In a queue, we can only remove elements from the front or add elements at the rear. As shown in Figure 5-7, a double-ended queue (deque) provides greater flexibility, allowing the addition or removal of elements at both the front and rear.

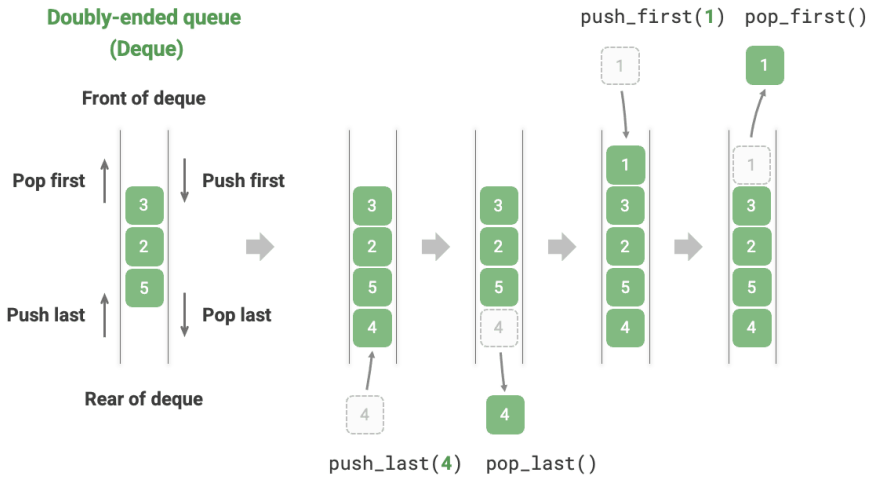


Figure 5-7 Operations of deque

5.3.1 Common Deque Operations

The common operations on a deque are shown in Table 5-3. The specific method names depend on the programming language used.

Table 5-3 Efficiency of Deque Operations

Method	Description	Time Complexity
`push_first()`	Add element to front	$O(1)$
`push_last()`	Add element to rear	$O(1)$
`pop_first()`	Remove front element	$O(1)$
`pop_last()`	Remove rear element	$O(1)$
`peek_first()`	Access front element	$O(1)$
`peek_last()`	Access rear element	$O(1)$

Similarly, we can directly use the deque classes already implemented in programming languages:

```javascript
// ═══ File: deque.js ═══

/* Initialize deque */
// JavaScript does not have a built-in deque, can only use Array as a deque
const deque = [];

/* Enqueue elements */
deque.push(2);
deque.push(5);
```

```
deque.push(4);
// Please note that since it's an array, unshift() has O(n) time complexity
deque.unshift(3);
deque.unshift(1);

/* Access elements */
const peekFirst = deque[0];
const peekLast = deque[deque.length - 1];

/* Dequeue elements */
// Please note that since it's an array, shift() has O(n) time complexity
const popFront = deque.shift();
const popBack = deque.pop();

/* Get deque length */
const size = deque.length;

/* Check if deque is empty */
const isEmpty = size === 0;
```

5.3.2 Deque Implementation *

The implementation of a deque is similar to that of a queue. You can choose either a linked list or an array as the underlying data structure.

1. Doubly Linked List Implementation

Reviewing the previous section, we used a regular singly linked list to implement a queue because it conveniently allows deleting the head node (corresponding to dequeue) and adding new nodes after the tail node (corresponding to enqueue).

For a deque, both the front and rear can perform enqueue and dequeue operations. In other words, a deque needs to implement operations in the opposite direction as well. For this reason, we use a "doubly linked list" as the underlying data structure for the deque.

As shown in Figure 5-8, we treat the head and tail nodes of the doubly linked list as the front and rear of the deque, implementing functionality to add and remove nodes at both ends.

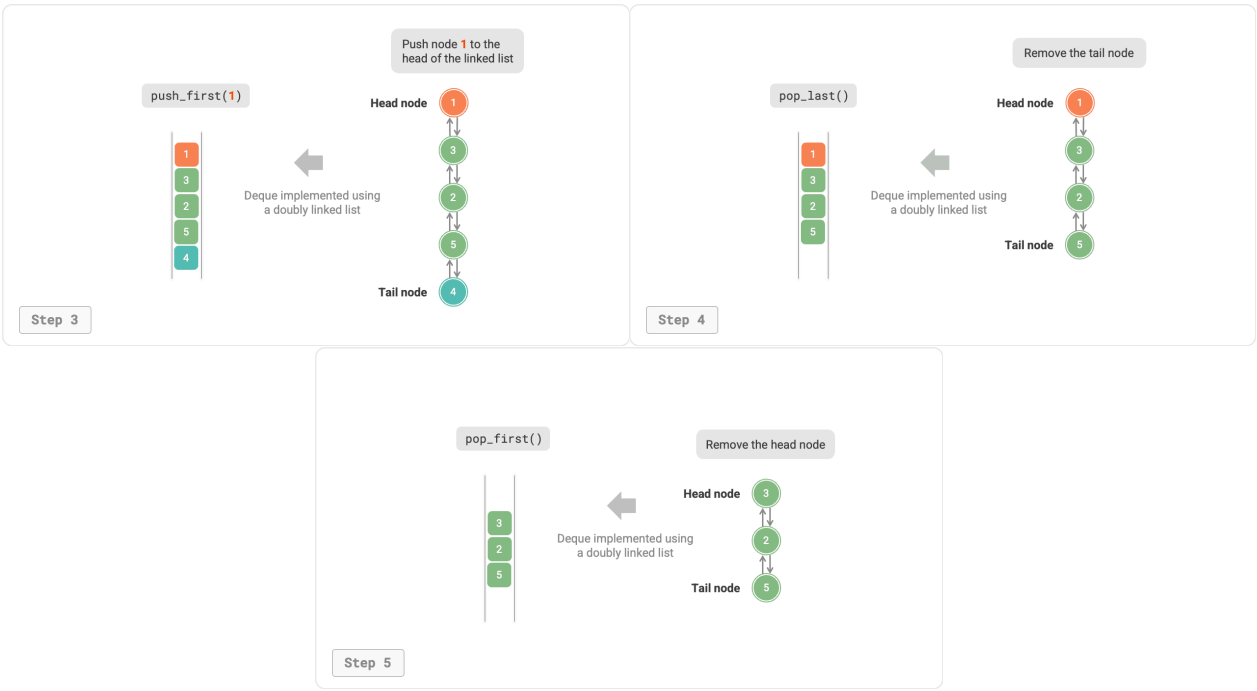


Figure 5-8 Enqueue and dequeue operations in linked list implementation of deque

The implementation code is shown below:

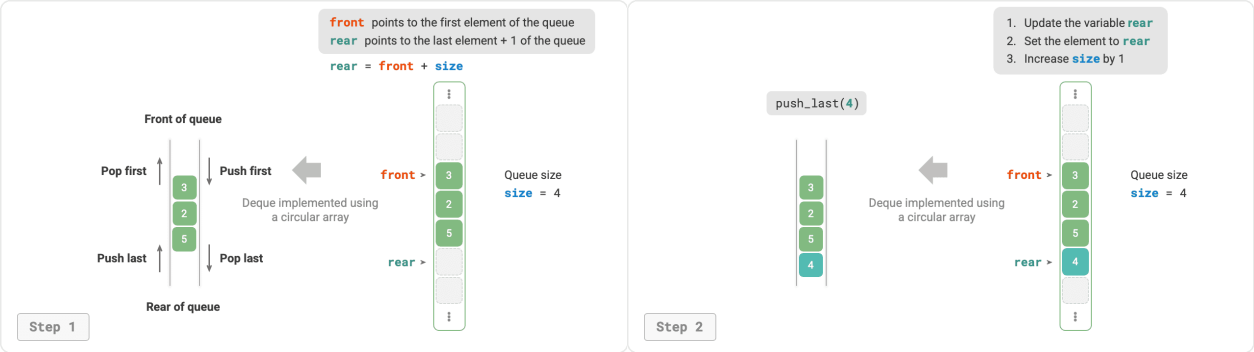
```
// ═══ File: linkedlist_deque.js ═══

/* Doubly linked list node */
class ListNode {
    prev; // Predecessor node reference (pointer)
    next; // Successor node reference (pointer)
    val; // Node value

    constructor(val) {
        this.val = val;
        this.next = null;
        this.prev = null;
    }
}

/* Double-ended queue based on doubly linked list implementation */
class LinkedListDeque {
    #front; // Head node front
    #rear; // Tail node rear
    #queSize; // Length of the double-ended queue

    constructor() {
        this.#front = null;
        this.#rear = null;
        this.#queSize = 0;
    }

    /* Rear of the queue enqueue operation */
    pushLast(val) {
        const node = new ListNode(val);
```

```
        // If the linked list is empty, make both front and rear point to node
        if (this.#queSize === 0) {
            this.#front = node;
            this.#rear = node;
        } else {
            // Add node to the tail of the linked list
            this.#rear.next = node;
            node.prev = this.#rear;
            this.#rear = node; // Update tail node
        }
        this.#queSize++;
    }

    /* Front of the queue enqueue operation */
    pushFirst(val) {
        const node = new ListNode(val);
        // If the linked list is empty, make both front and rear point to node
        if (this.#queSize === 0) {
            this.#front = node;
            this.#rear = node;
        } else {
            // Add node to the head of the linked list
            this.#front.prev = node;
            node.next = this.#front;
            this.#front = node; // Update head node
        }
        this.#queSize++;
    }

    /* Temporarily store tail node value */
    popLast() {
        if (this.#queSize === 0) {
            return null;
        }
        const value = this.#rear.val; // Store tail node value
        // Update tail node
        let temp = this.#rear.prev;
        if (temp !== null) {
            temp.next = null;
            this.#rear.prev = null;
        }
        this.#rear = temp; // Update tail node
        this.#queSize--;
        return value;
    }

    /* Temporarily store head node value */
    popFirst() {
        if (this.#queSize === 0) {
            return null;
        }
        const value = this.#front.val; // Store tail node value
        // Delete head node
        let temp = this.#front.next;
        if (temp !== null) {
            temp.prev = null;
            this.#front.next = null;
        }
        this.#front = temp; // Update head node
        this.#queSize--;
```

```
        return value;
    }

    /* Driver Code */
    peekLast() {
        return this.#queSize === 0 ? null : this.#rear.val;
    }

    /* Return list for printing */
    peekFirst() {
        return this.#queSize === 0 ? null : this.#front.val;
    }

    /* Get the length of the double-ended queue */
    size() {
        return this.#queSize;
    }

    /* Check if the double-ended queue is empty */
    isEmpty() {
        return this.#queSize === 0;
    }

    /* Print deque */
    print() {
        const arr = [];
        let temp = this.#front;
        while (temp !== null) {
            arr.push(temp.val);
            temp = temp.next;
        }
        console.log('[' + arr.join(', ') + ']');
    }
}
```

2. Array Implementation

As shown in Figure 5-9, similar to implementing a queue based on an array, we can also use a circular array to implement a deque.



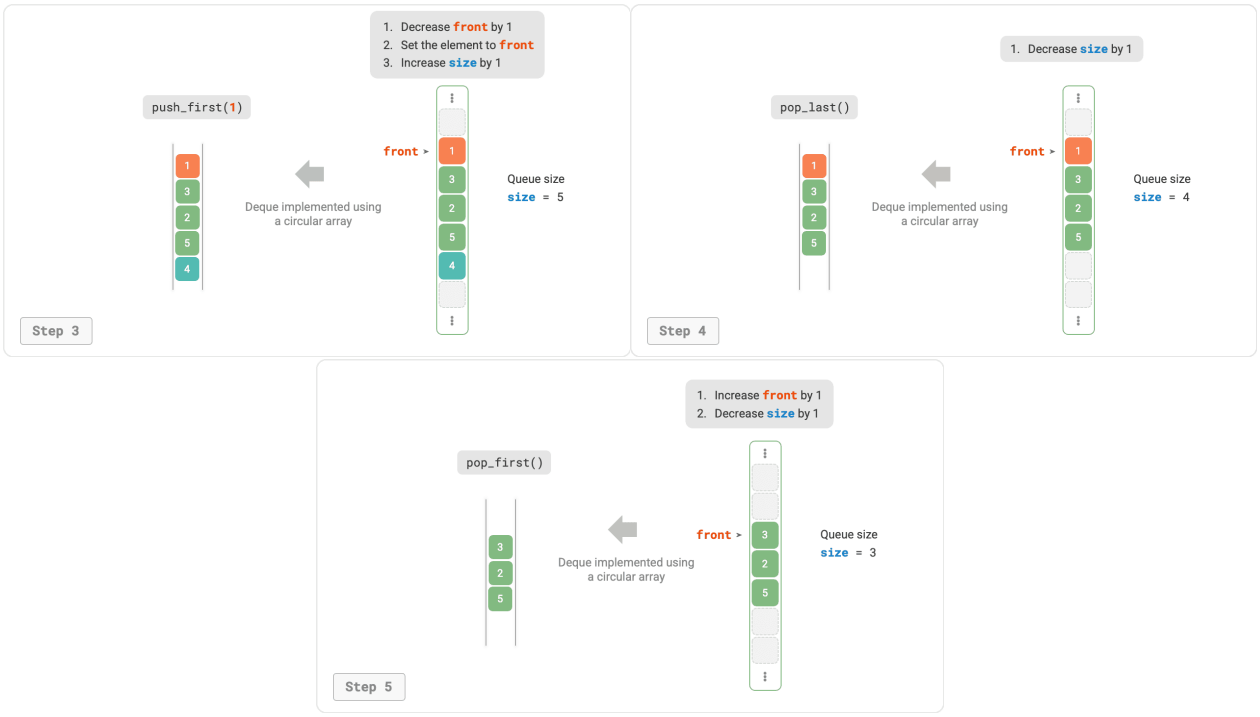


Figure 5-9 Enqueue and dequeue operations in array implementation of deque

Based on the queue implementation, we only need to add methods for "enqueue at front" and "dequeue from rear":

```
// ═══ File: array_deque.js ═══

/* Double-ended queue based on circular array implementation */
class ArrayDeque {
    #nums; // Array for storing double-ended queue elements
    #front; // Front pointer, points to the front of the queue element
    #queSize; // Double-ended queue length

    /* Constructor */
    constructor(capacity) {
        this.#nums = new Array(capacity);
        this.#front = 0;
        this.#queSize = 0;
    }

    /* Get the capacity of the double-ended queue */
    capacity() {
        return this.#nums.length;
    }

    /* Get the length of the double-ended queue */
    size() {
        return this.#queSize;
    }

    /* Check if the double-ended queue is empty */
    isEmpty() {
        return this.#queSize === 0;
```

```
}

/* Calculate circular array index */
index(i) {
    // Use modulo operation to wrap the array head and tail together
    // When i passes the tail of the array, return to the head
    // When i passes the head of the array, return to the tail
    return (i + this.capacity()) % this.capacity();
}

/* Front of the queue enqueue */
pushFirst(num) {
    if (this.#queSize === this.capacity()) {
        console.log('Double-ended queue is full');
        return;
    }
    // Use modulo operation to wrap front around to the tail after passing the head of the
    ↪ array
    // Add num to the front of the queue
    this.#front = this.index(this.#front - 1);
    // Add num to front of queue
    this.#nums[this.#front] = num;
    this.#queSize++;
}

/* Rear of the queue enqueue */
pushLast(num) {
    if (this.#queSize === this.capacity()) {
        console.log('Double-ended queue is full');
        return;
    }
   // Use modulo operation to wrap rear around to the head after passing the tail of the array
    const rear = this.index(this.#front + this.#queSize);
    // Front pointer moves one position backward
    this.#nums[rear] = num;
    this.#queSize++;
}

/* Rear of the queue dequeue */
popFirst() {
    const num = this.peekFirst();
    // Move front pointer backward by one position
    this.#front = this.index(this.#front + 1);
    this.#queSize--;
    return num;
}

/* Access rear of the queue element */
popLast() {
    const num = this.peekLast();
    this.#queSize--;
    return num;
}

/* Return list for printing */
peekFirst() {
    if (this.isEmpty()) throw new Error('The Deque Is Empty.');
    return this.#nums[this.#front];
}
```

```
    /* Driver Code */
    peekLast() {
        if (this.isEmpty()) throw new Error('The Deque Is Empty.');
        // Initialize double-ended queue
        const last = this.index(this.#front + this.#queSize - 1);
        return this.#nums[last];
    }

    /* Return array for printing */
    toArray() {
        // Elements enqueue
        const res = [];
        for (let i = 0, j = this.#front; i < this.#queSize; i++, j++) {
            res[i] = this.#nums[this.index(j)];
        }
        return res;
    }
}
```

5.3.3 Deque Applications

A deque combines the logic of both stacks and queues. **Therefore, it can implement all application scenarios of both, while providing greater flexibility**.

We know that the "undo" function in software is typically implemented using a stack: the system pushes each change operation onto the stack and then implements undo through pop. However, considering system resource limitations, software usually limits the number of undo steps (for example, only allowing 50 steps to be saved). When the stack length exceeds 50, the software needs to perform a deletion operation at the bottom of the stack (front of the queue). **But a stack cannot implement this functionality, so a deque is needed to replace the stack**. Note that the core logic of "undo" still follows the LIFO principle of a stack; it's just that the deque can more flexibly implement some additional logic.

5.4 Summary

1. Key Review

- A stack is a data structure that follows the LIFO principle and can be implemented using arrays or linked lists.
- In terms of time efficiency, the array implementation of a stack has higher average efficiency, but during expansion, the time complexity of a single push operation degrades to $O(n)$. In contrast, the linked list implementation of a stack provides more stable efficiency performance.
- In terms of space efficiency, the array implementation of a stack may lead to some degree of space wastage. However, it should be noted that the memory space occupied by linked list nodes is larger than that of array elements.
- A queue is a data structure that follows the FIFO principle and can also be implemented using arrays or linked lists. The conclusions regarding time efficiency and space efficiency comparisons for queues are similar to those for stacks mentioned above.

- A deque is a queue with greater flexibility that allows adding and removing elements at both ends.

2. Q & A

Q: Is the browser's forward and backward functionality implemented with a doubly linked list?

The forward and backward functionality of a browser is essentially a manifestation of a "stack." When a user visits a new page, that page is added to the top of the stack; when the user clicks the back button, that page is popped from the top of the stack. Using a deque can conveniently implement some additional operations, as mentioned in the "Deque" section.

Q: After popping from the stack, do we need to free the memory of the popped node?

If the popped node will still be needed later, then memory does not need to be freed. If it won't be used afterward, languages like Java and Python have automatic garbage collection, so manual memory deallocation is not required; in C and C++, manual memory deallocation is necessary.

Q: A deque seems like two stacks joined together. What is its purpose?

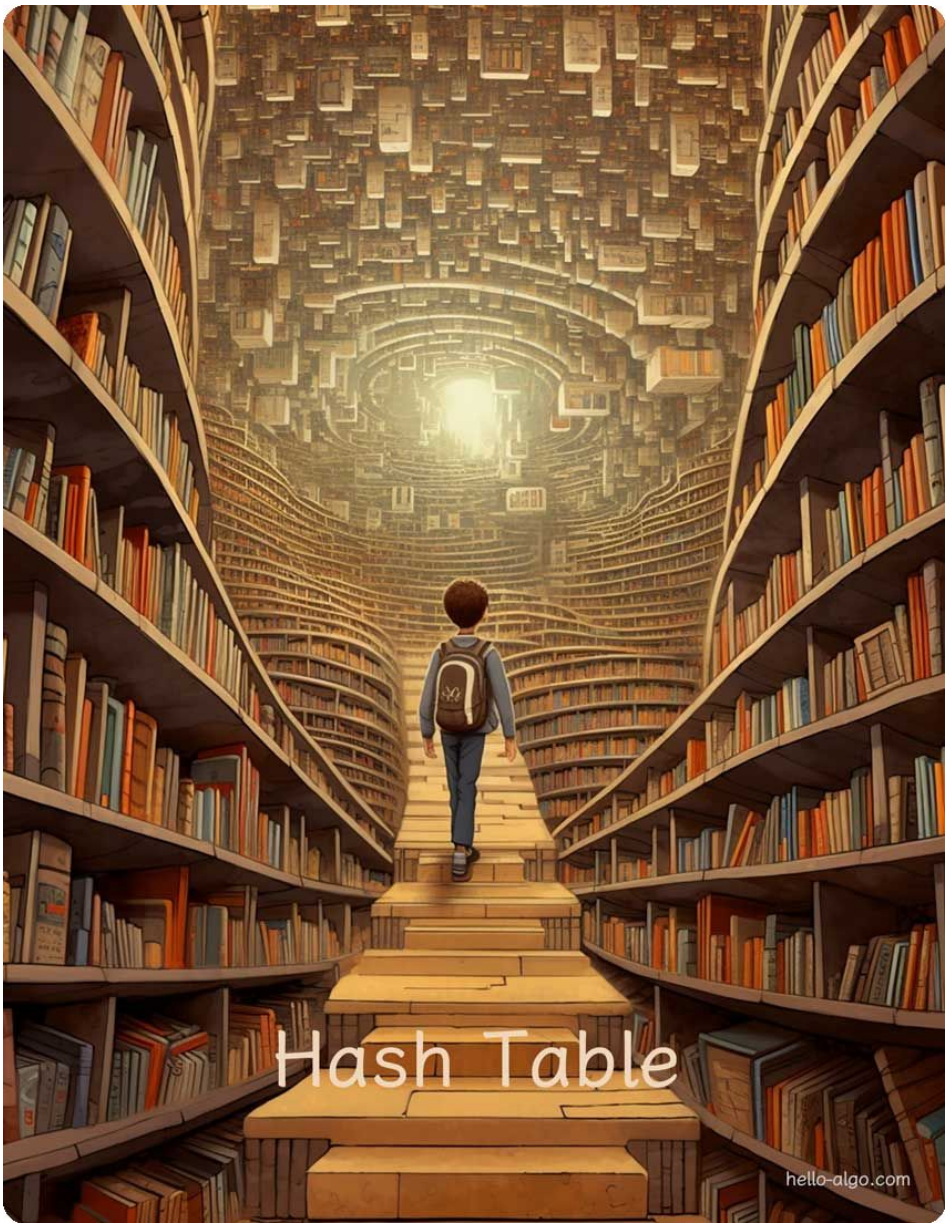
A deque is like a combination of a stack and a queue, or two stacks joined together. It exhibits the logic of both stack and queue, so it can implement all applications of stacks and queues, and is more flexible.

Q: How are undo and redo specifically implemented?

Use two stacks: stack `A` for undo and stack `B` for redo.

1. Whenever the user performs an operation, push this operation onto stack `A` and clear stack `B`.
2. When the user performs "undo," pop the most recent operation from stack `A` and push it onto stack `B`.
3. When the user performs "redo," pop the most recent operation from stack `B` and push it onto stack `A`.

Chapter 6. Hashing



Abstract

In the world of computing, a hash table is like a clever librarian.

They know how to calculate call numbers, enabling them to quickly locate the target book.

6.1 Hash Table

A hash table, also known as a hash map, establishes a mapping between keys `key` and values `value`, enabling efficient element retrieval. Specifically, when we input a key `key` into a hash table, we can retrieve the corresponding value `value` in $O(1)$ time.

As shown in Figure 6-1, given n students, each with two pieces of data: "name" and "student ID". If we want to implement a query function that "inputs a student ID and returns the corresponding name", we can use the hash table shown below.

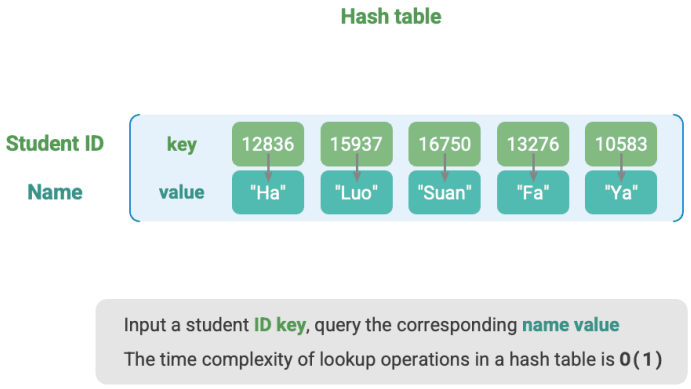


Figure 6-1 Abstract representation of a hash table

In addition to hash tables, arrays and linked lists can also implement query functionality. Their efficiency comparison is shown in the following table.

- **Adding elements**: Simply add elements to the end of the array (linked list), using $O(1)$ time.
- **Querying elements**: Since the array (linked list) is unordered, all elements need to be traversed, using $O(n)$ time.
- **Deleting elements**: The element must first be located, then deleted from the array (linked list), using $O(n)$ time.

Table 6-1 Comparison of element query efficiency

	Array	Linked List	Hash Table
Find element	$O(n)$	$O(n)$	$O(1)$
Add element	$O(1)$	$O(1)$	$O(1)$
Delete element	$O(n)$	$O(n)$	$O(1)$

As observed, **the time complexity for insertion, deletion, search, and modification operations in a hash table is $O(1)$**, which is very efficient.

6.1.1 Common Hash Table Operations

Common operations on hash tables include: initialization, query operations, adding key-value pairs, and deleting key-value pairs. Example code is as follows:

```javascript
// == File: hash_map.js ==

/* Initialize hash table */
const map = new Map();
/* Add operation */
// Add key-value pair (key, value) to hash table
map.set(12836, 'XiaoHa');
map.set(15937, 'XiaoLuo');
map.set(16750, 'XiaoSuan');
map.set(13276, 'XiaoFa');
map.set(10583, 'XiaoYa');

/* Query operation */
// Input key into hash table to get value
let name = map.get(15937);

/* Delete operation */
// Delete key-value pair (key, value) from hash table
map.delete(10583);
```

There are three common ways to traverse a hash table: traversing key-value pairs, traversing keys, and traversing values. Example code is as follows:

```javascript
// == File: hash_map.js ==

/* Traverse hash table */
console.info('\nTraverse key-value pairs Key->Value');
for (const [k, v] of map.entries()) {
    console.info(k + ' -> ' + v);
}
console.info('\nTraverse keys only Key');
for (const k of map.keys()) {
    console.info(k);
}
console.info('\nTraverse values only Value');
for (const v of map.values()) {
    console.info(v);
}
```

6.1.2 Simple Hash Table Implementation

Let's first consider the simplest case: **implementing a hash table using only an array**. In a hash table, each empty position in the array is called a bucket, and each bucket can store a key-value pair. Therefore, the query operation is to find the bucket corresponding to `key` and retrieve the `value` from the bucket.

So how do we locate the corresponding bucket based on `key`? This is achieved through a hash function. The role of the hash function is to map a larger input space to a smaller output space. In a hash table,

the input space is all `keys`, and the output space is all buckets (array indices). In other words, given a `key`, **we can use the hash function to obtain the storage location of the key-value pair corresponding to that `key` in the array**.

When inputting a `key`, the hash function's calculation process consists of the following two steps:

1. Calculate the hash value through a hash algorithm `hash()`.
2. Take the modulo of the hash value by the number of buckets (array length) `capacity` to obtain the bucket (array index) `index` corresponding to that `key`.

```
index = hash(key) % capacity
```

Subsequently, we can use `index` to access the corresponding bucket in the hash table and retrieve the `value`.

Assuming the array length is `capacity = 100` and the hash algorithm is `hash(key) = key`, the hash function becomes `key % 100`. Figure 6-2 shows the working principle of the hash function using `key` as student ID and `value` as name.

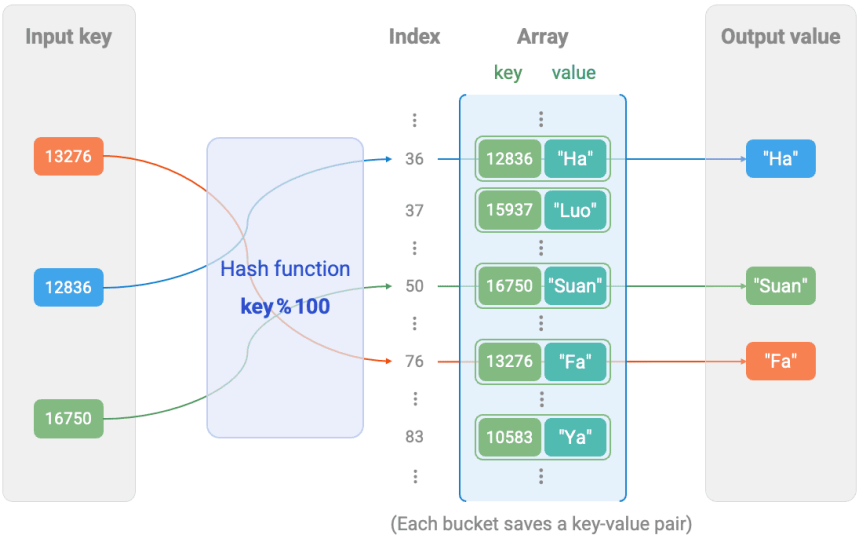


Figure 6-2 Working principle of hash function

The following code implements a simple hash table. Here, we encapsulate `key` and `value` into a class `Pair` to represent a key-value pair.

```
// === File: array_hash_map.js ===

/* Key-value pair Number -> String */
class Pair {
    constructor(key, val) {
        this.key = key;
        this.val = val;
    }
```

```
}

/* Hash table based on array implementation */
class ArrayHashMap {
    #buckets;
    constructor() {
        // Initialize array with 100 buckets
        this.#buckets = new Array(100).fill(null);
    }

    /* Hash function */
    #hashFunc(key) {
        return key % 100;
    }

    /* Query operation */
    get(key) {
        let index = this.#hashFunc(key);
        let pair = this.#buckets[index];
        if (pair === null) return null;
        return pair.val;
    }

    /* Add operation */
    set(key, val) {
        let index = this.#hashFunc(key);
        this.#buckets[index] = new Pair(key, val);
    }

    /* Remove operation */
    delete(key) {
        let index = this.#hashFunc(key);
        // Set to null to represent deletion
        this.#buckets[index] = null;
    }

    /* Get all key-value pairs */
    entries() {
        let arr = [];
        for (let i = 0; i < this.#buckets.length; i++) {
            if (this.#buckets[i]) {
                arr.push(this.#buckets[i]);
            }
        }
        return arr;
    }

    /* Get all keys */
    keys() {
        let arr = [];
        for (let i = 0; i < this.#buckets.length; i++) {
            if (this.#buckets[i]) {
                arr.push(this.#buckets[i].key);
            }
        }
        return arr;
    }

    /* Get all values */
    values() {
```

```
        let arr = [];
        for (let i = 0; i < this.#buckets.length; i++) {
            if (this.#buckets[i]) {
                arr.push(this.#buckets[i].val);
            }
        }
        return arr;
    }

    /* Print hash table */
    print() {
        let pairSet = this.entries();
        for (const pair of pairSet) {
            console.info(`${pair.key} -> ${pair.val}`);
        }
    }
}
```

6.1.3 Hash Collision and Resizing

Fundamentally, the role of a hash function is to map the input space consisting of all `keys` to the output space consisting of all array indices, and the input space is often much larger than the output space. Therefore, **theoretically there must be cases where "multiple inputs correspond to the same output"**.

For the hash function in the above example, when the input `keys` have the same last two digits, the hash function produces the same output. For example, when querying two students with IDs 12836 and 20336, we get:

```
12836 % 100 = 36
20336 % 100 = 36
```

As shown in Figure 6-3, two student IDs point to the same name, which is obviously incorrect. We call this situation where multiple inputs correspond to the same output a hash collision.

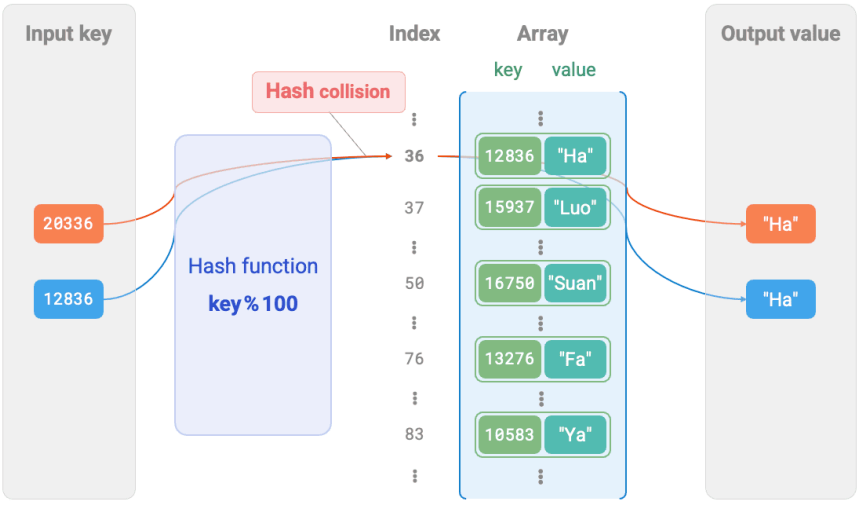


Figure 6-3 Hash collision example

It's easy to see that the larger the hash table capacity n, the lower the probability that multiple `keys` will be assigned to the same bucket, and the fewer collisions. Therefore, **we can reduce hash collisions by expanding the hash table**.

As shown in Figure 6-4, before expansion, the key-value pairs `(136, A)` and `(236, D)` collided, but after expansion, the collision disappears.

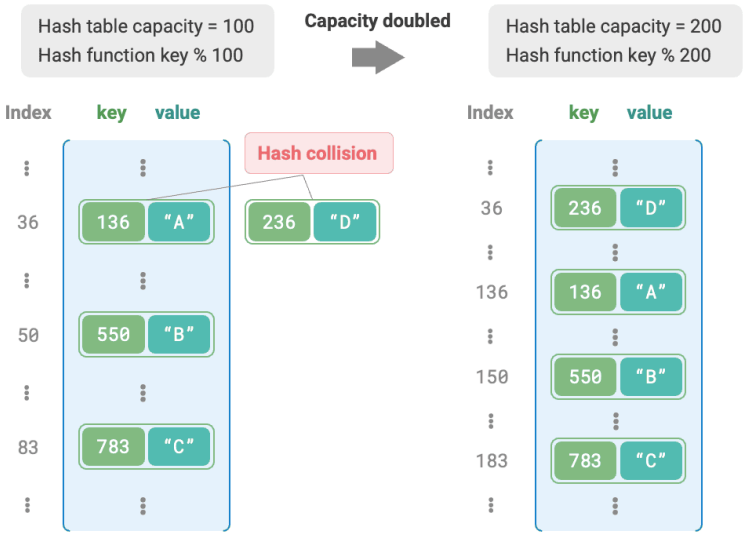


Figure 6-4 Hash table resizing

Similar to array expansion, hash table expansion requires migrating all key-value pairs from the original hash table to the new hash table, which is very time-consuming. Moreover, since the hash table capacity `capacity` changes, we need to recalculate the storage locations of all key-value pairs through the hash function, further increasing the computational overhead of the expansion process. For this reason, programming languages typically reserve a sufficiently large hash table capacity to prevent frequent expansion.

The load factor is an important concept for hash tables. It is defined as the number of elements in the hash table divided by the number of buckets, and is used to measure the severity of hash collisions. **It is also commonly used as a trigger condition for hash table expansion**. For example, in Java, when the load factor exceeds 0.75, the system will expand the hash table to 2 times its original size.

6.2 Hash Collision

The previous section mentioned that, **in most cases, the input space of a hash function is much larger than the output space**, so theoretically, hash collisions are inevitable. For example, if the input space is all integers and the output space is the array capacity size, then multiple integers will inevitably be mapped to the same bucket index.

Hash collisions can lead to incorrect query results, severely impacting the usability of the hash table. To address this issue, whenever a hash collision occurs, we can perform hash table expansion until the collision disappears. This approach is simple, straightforward, and effective, but it is very inefficient because hash table expansion involves a large amount of data migration and hash value recalculation. To improve efficiency, we can adopt the following strategies:

1. Improve the hash table data structure so that **the hash table can function normally when hash collisions occur**.
2. Only expand when necessary, that is, only when hash collisions are severe.

The main methods for improving the structure of hash tables include "separate chaining" and "open addressing".

6.2.1 Separate Chaining

In the original hash table, each bucket can store only one key-value pair. Separate chaining converts a single element into a linked list, treating key-value pairs as linked list nodes and storing all colliding key-value pairs in the same linked list. Figure 6-5 shows an example of a separate chaining hash table.

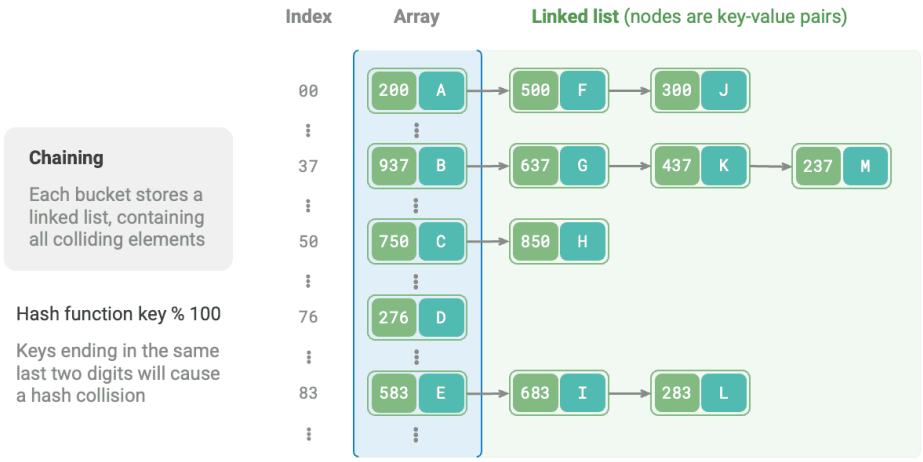


Figure 6-5 Separate chaining hash table

The operations of a hash table implemented with separate chaining have changed as follows:

- **Querying elements**: Input `key`, obtain the bucket index through the hash function, then access the head node of the linked list, then traverse the linked list and compare `key` to find the target key-value pair.
- **Adding elements**: First access the linked list head node through the hash function, then append the node (key-value pair) to the linked list.
- **Deleting elements**: Access the head of the linked list based on the result of the hash function, then traverse the linked list to find the target node and delete it.

Separate chaining has the following limitations:

- **Increased Space Usage**: The linked list contains node pointers, which consume more memory space than arrays.
- **Reduced Query Efficiency**: This is because linear traversal of the linked list is required to find the corresponding element.

The code below provides a simple implementation of a separate chaining hash table, with two things to note:

- Lists (dynamic arrays) are used instead of linked lists to simplify the code. In this setup, the hash table (array) contains multiple buckets, each of which is a list.
- This implementation includes a hash table expansion method. When the load factor exceeds $\frac{2}{3}$, we expand the hash table to 2 times its original size.

```
// ═══ File: hash_map_chaining.js ═══

/* Hash table with separate chaining */
class HashMapChaining {
    #size; // Number of key-value pairs
    #capacity; // Hash table capacity
    #loadThres; // Load factor threshold for triggering expansion
    #extendRatio; // Expansion multiplier
    #buckets; // Bucket array

    /* Constructor */
    constructor() {
        this.#size = 0;
        this.#capacity = 4;
        this.#loadThres = 2.0 / 3.0;
        this.#extendRatio = 2;
        this.#buckets = new Array(this.#capacity).fill(null).map((x) => []);
    }

    /* Hash function */
    #hashFunc(key) {
        return key % this.#capacity;
    }

    /* Load factor */
    #loadFactor() {
        return this.#size / this.#capacity;
    }

    /* Query operation */
    get(key) {
        const index = this.#hashFunc(key);
        const bucket = this.#buckets[index];
        // Traverse bucket, if key is found, return corresponding val
        for (const pair of bucket) {
            if (pair.key === key) {
                return pair.val;
            }
        }
        // If key is not found, return null
        return null;
    }
```

```
    /* Add operation */
    put(key, val) {
        // When load factor exceeds threshold, perform expansion
        if (this.#loadFactor() > this.#loadThres) {
            this.#extend();
        }
        const index = this.#hashFunc(key);
        const bucket = this.#buckets[index];
        // Traverse bucket, if specified key is encountered, update corresponding val and return
        for (const pair of bucket) {
            if (pair.key === key) {
                pair.val = val;
                return;
            }
        }
        // If key does not exist, append key-value pair to the end
        const pair = new Pair(key, val);
        bucket.push(pair);
        this.#size++;
    }

    /* Remove operation */
    remove(key) {
        const index = this.#hashFunc(key);
        let bucket = this.#buckets[index];
        // Traverse bucket and remove key-value pair from it
        for (let i = 0; i < bucket.length; i++) {
            if (bucket[i].key === key) {
                bucket.splice(i, 1);
                this.#size--;
                break;
            }
        }
    }

    /* Expand hash table */
    #extend() {
        // Temporarily store the original hash table
        const bucketsTmp = this.#buckets;
        // Initialize expanded new hash table
        this.#capacity *= this.#extendRatio;
        this.#buckets = new Array(this.#capacity).fill(null).map((x) => []);
        this.#size = 0;
        // Move key-value pairs from original hash table to new hash table
        for (const bucket of bucketsTmp) {
            for (const pair of bucket) {
                this.put(pair.key, pair.val);
            }
        }
    }

    /* Print hash table */
    print() {
        for (const bucket of this.#buckets) {
            let res = [];
            for (const pair of bucket) {
                res.push(pair.key + ' -> ' + pair.val);
            }
            console.log(res);
```

```
        }
    }
}
```

It's worth noting that when the linked list is very long, the query efficiency $O(n)$ is poor. **In this case, the list can be converted to an "AVL tree" or "Red-Black tree"** to optimize the time complexity of the query operation to $O(\log n)$.

6.2.2 Open Addressing

Open addressing does not introduce additional data structures but instead handles hash collisions through "multiple probes". The probing methods mainly include linear probing, quadratic probing, and double hashing.

Let's use linear probing as an example to introduce the mechanism of open addressing hash tables.

1. Linear Probing

Linear probing uses a fixed-step linear search for probing, and its operation method differs from ordinary hash tables.

- **Inserting elements**: Calculate the bucket index using the hash function. If the bucket already contains an element, linearly traverse forward from the conflict position (usually with a step size of 1) until an empty bucket is found, then insert the element.
- **Searching for elements**: If a hash collision is encountered, use the same step size to linearly traverse forward until the corresponding element is found and return `value`; if an empty bucket is encountered, it means the target element is not in the hash table, so return `None`.

Figure 6-6 shows the distribution of key-value pairs in an open addressing (linear probing) hash table. According to this hash function, keys with the same last two digits will be mapped to the same bucket. Through linear probing, they are stored sequentially in that bucket and the buckets below it.

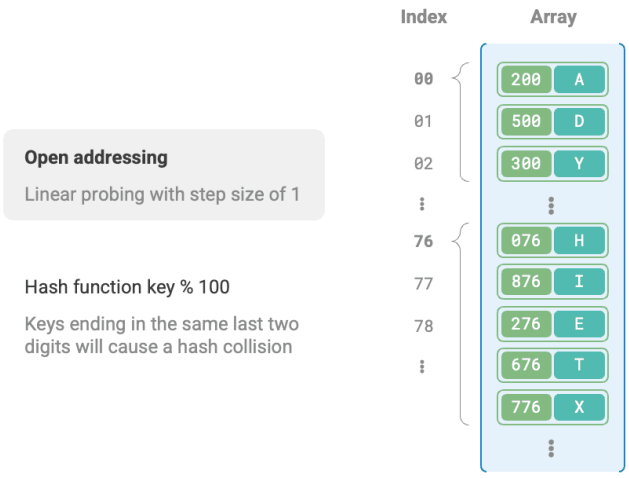


Figure 6-6 Distribution of key-value pairs in open addressing (linear probing) hash table

However, **linear probing is prone to create "clustering"**. Specifically, the longer the continuously occupied positions in the array, the greater the probability of hash collisions occurring in these continuous positions, further promoting clustering growth at that position, forming a vicious cycle, and ultimately leading to degraded efficiency of insertion, deletion, query, and update operations.

It's important to note that **we cannot directly delete elements in an open addressing hash table**. Deleting an element creates an empty bucket `None` in the array. When searching for elements, if linear probing encounters this empty bucket, it will return, making the elements below this empty bucket inaccessible. The program may incorrectly assume these elements do not exist, as shown in Figure 6-7.

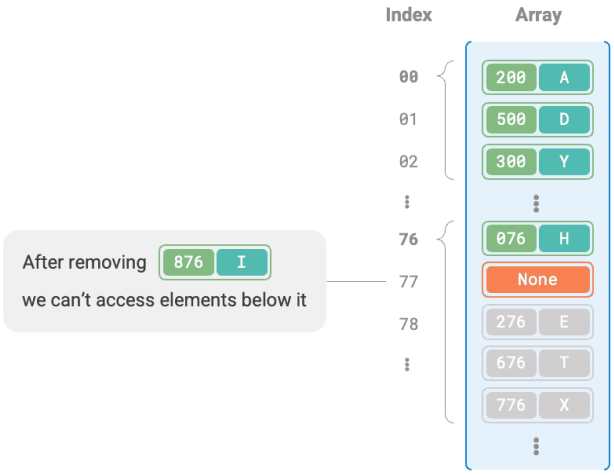


Figure 6-7 Query issues caused by deletion in open addressing

To solve this problem, we can adopt the lazy deletion mechanism: instead of directly removing elements from the hash table, **use a constant `TOMBSTONE` to mark the bucket**. In this mechanism, both `None` and `TOMBSTONE` represent empty buckets and can hold key-value pairs. However, when linear probing encounters `TOMBSTONE`, it should continue traversing since there may still be key-value pairs below it.

However, **lazy deletion may accelerate the performance degradation of the hash table**. Every deletion operation produces a deletion mark, and as `TOMBSTONE` increases, the search time will also increase because linear probing may need to skip multiple `TOMBSTONE` to find the target element.

To address this, consider recording the index of the first encountered `TOMBSTONE` during linear probing and swapping the searched target element with that `TOMBSTONE`. The benefit of doing this is that each time an element is queried or added, the element will be moved to a bucket closer to its ideal position (the starting point of probing), thereby optimizing query efficiency.

The code below implements an open addressing (linear probing) hash table with lazy deletion. To make better use of the hash table space, we treat the hash table as a "circular array". When going beyond the end of the array, we return to the beginning and continue traversing.

```javascript
// === File: hash_map_open_addressing.js ===

/* Hash table with open addressing */
class HashMapOpenAddressing {
    #size; // Number of key-value pairs
    #capacity; // Hash table capacity
    #loadThres; // Load factor threshold for triggering expansion
    #extendRatio; // Expansion multiplier
    #buckets; // Bucket array
    #TOMBSTONE; // Removal marker

    /* Constructor */
    constructor() {
        this.#size = 0; // Number of key-value pairs
        this.#capacity = 4; // Hash table capacity
        this.#loadThres = 2.0 / 3.0; // Load factor threshold for triggering expansion
        this.#extendRatio = 2; // Expansion multiplier
        this.#buckets = Array(this.#capacity).fill(null); // Bucket array
        this.#TOMBSTONE = new Pair(-1, '-1'); // Removal marker
    }

    /* Hash function */
    #hashFunc(key) {
        return key % this.#capacity;
    }

    /* Load factor */
    #loadFactor() {
        return this.#size / this.#capacity;
    }

    /* Search for bucket index corresponding to key */
    #findBucket(key) {
        let index = this.#hashFunc(key);
        let firstTombstone = -1;
        // Linear probing, break when encountering an empty bucket
        while (this.#buckets[index] !== null) {
            // If key is encountered, return the corresponding bucket index
            if (this.#buckets[index].key === key) {
                // If a removal marker was encountered before, move the key-value pair to that
                ↪ index
                if (firstTombstone !== -1) {
                    this.#buckets[firstTombstone] = this.#buckets[index];
                    this.#buckets[index] = this.#TOMBSTONE;
                    return firstTombstone; // Return the moved bucket index
                }
                return index; // Return bucket index
            }
            // Record the first removal marker encountered
            if (
                firstTombstone === -1 &&
                this.#buckets[index] === this.#TOMBSTONE
            ) {
                firstTombstone = index;
            }
            // Calculate bucket index, wrap around to the head if past the tail
            index = (index + 1) % this.#capacity;
        }
        // If key does not exist, return the index for insertion
        return firstTombstone === -1 ? index : firstTombstone;
```

```
}

/* Query operation */
get(key) {
    // Search for bucket index corresponding to key
    const index = this.#findBucket(key);
    // If key-value pair is found, return corresponding val
    if (
        this.#buckets[index] !== null &&
        this.#buckets[index] !== this.#TOMBSTONE
    ) {
        return this.#buckets[index].val;
    }
    // If key-value pair does not exist, return null
    return null;
}

/* Add operation */
put(key, val) {
    // When load factor exceeds threshold, perform expansion
    if (this.#loadFactor() > this.#loadThres) {
        this.#extend();
    }
    // Search for bucket index corresponding to key
    const index = this.#findBucket(key);
    // If key-value pair is found, overwrite val and return
    if (
        this.#buckets[index] !== null &&
        this.#buckets[index] !== this.#TOMBSTONE
    ) {
        this.#buckets[index].val = val;
        return;
    }
    // If key-value pair does not exist, add the key-value pair
    this.#buckets[index] = new Pair(key, val);
    this.#size++;
}

/* Remove operation */
remove(key) {
    // Search for bucket index corresponding to key
    const index = this.#findBucket(key);
    // If key-value pair is found, overwrite it with removal marker
    if (
        this.#buckets[index] !== null &&
        this.#buckets[index] !== this.#TOMBSTONE
    ) {
        this.#buckets[index] = this.#TOMBSTONE;
        this.#size--;
    }
}

/* Expand hash table */
#extend() {
    // Temporarily store the original hash table
    const bucketsTmp = this.#buckets;
    // Initialize expanded new hash table
    this.#capacity *= this.#extendRatio;
    this.#buckets = Array(this.#capacity).fill(null);
    this.#size = 0;
```

```
        // Move key-value pairs from original hash table to new hash table
        for (const pair of bucketsTmp) {
            if (pair !== null && pair !== this.#TOMBSTONE) {
                this.put(pair.key, pair.val);
            }
        }
    }

    /* Print hash table */
    print() {
        for (const pair of this.#buckets) {
            if (pair === null) {
                console.log('null');
            } else if (pair === this.#TOMBSTONE) {
                console.log('TOMBSTONE');
            } else {
                console.log(pair.key + ' -> ' + pair.val);
            }
        }
    }
}
```

2. Quadratic Probing

Quadratic probing is similar to linear probing and is one of the common strategies for open addressing. When a collision occurs, quadratic probing does not simply skip a fixed number of steps but skips a number of steps equal to the "square of the number of probes", i.e., $1, 4, 9, \ldots$ steps.

Quadratic probing has the following advantages:

- Quadratic probing attempts to alleviate the clustering effect of linear probing by skipping distances equal to the square of the probe count.
- Quadratic probing skips larger distances to find empty positions, which helps to distribute data more evenly.

However, quadratic probing is not perfect:

- Clustering still exists, i.e., some positions are more likely to be occupied than others.
- Due to the growth of squares, quadratic probing may not probe the entire hash table, meaning that even if there are empty buckets in the hash table, quadratic probing may not be able to access them.

3. Double Hashing

As the name suggests, the double hashing method uses multiple hash functions $f_1(x)$, $f_2(x)$, $f_3(x)$, $\ldots$ for probing.

- **Inserting elements**: If hash function $f_1(x)$ encounters a conflict, try $f_2(x)$, and so on, until an empty position is found and the element is inserted.

- **Searching for elements**: Search in the same order of hash functions until the target element is found and return it; if an empty position is encountered or all hash functions have been tried, it indicates the element is not in the hash table, then return `None`.

Compared to linear probing, the double hashing method is less prone to clustering, but multiple hash functions introduce additional computational overhead.

> **Tip**
> Please note that open addressing (linear probing, quadratic probing, and double hashing) hash tables all have the problem of "cannot directly delete elements".

6.2.3 Choice of Programming Languages

Different programming languages adopt different hash table implementation strategies. Here are a few examples:

- Python uses open addressing. The `dict` dictionary uses pseudo-random numbers for probing.
- Java uses separate chaining. Since JDK 1.8, when the array length in `HashMap` reaches 64 and the length of a linked list reaches 8, the linked list is converted to a red-black tree to improve search performance.
- Go uses separate chaining. Go stipulates that each bucket can store up to 8 key-value pairs, and if the capacity is exceeded, an overflow bucket is linked; when there are too many overflow buckets, a special equal-capacity expansion operation is performed to ensure performance.

6.3 Hash Algorithm

The previous two sections introduced the working principle of hash tables and the methods to handle hash collisions. However, both open addressing and separate chaining **can only ensure that the hash table functions normally when hash collisions occur, but cannot reduce the frequency of hash collisions**.

If hash collisions occur too frequently, the performance of the hash table will deteriorate drastically. As shown in Figure 6-8, for a separate chaining hash table, in the ideal case, the key-value pairs are evenly distributed across the buckets, achieving optimal query efficiency; in the worst case, all key-value pairs are stored in the same bucket, degrading the time complexity to $O(n)$.

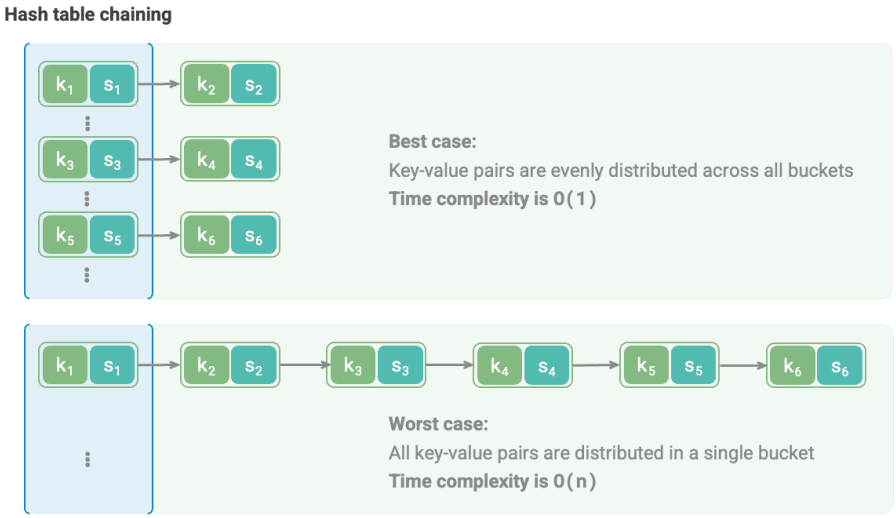


Figure 6-8 Ideal and worst cases of hash collisions

The distribution of key-value pairs is determined by the hash function. Recalling the calculation steps of the hash function, first compute the hash value, then take the modulo by the array length:

```
index = hash(key) % capacity
```

Observing the above formula, when the hash table capacity `capacity` is fixed, **the hash algorithm `hash()` determines the output value**, thereby determining the distribution of key-value pairs in the hash table.

This means that, to reduce the probability of hash collisions, we should focus on the design of the hash algorithm `hash()`.

6.3.1 Goals of Hash Algorithms

To achieve a "fast and stable" hash table data structure, hash algorithms should have the following characteristics:

- **Determinism**: For the same input, the hash algorithm should always produce the same output. Only then can the hash table be reliable.
- **High efficiency**: The process of computing the hash value should be fast enough. The smaller the computational overhead, the more practical the hash table.
- **Uniform distribution**: The hash algorithm should ensure that key-value pairs are evenly distributed in the hash table. The more uniform the distribution, the lower the probability of hash collisions.

In fact, hash algorithms are not only used to implement hash tables but are also widely applied in other fields.

- **Password storage**: To protect the security of user passwords, systems usually do not store the plaintext passwords but rather the hash values of the passwords. When a user enters a password, the system calculates the hash value of the input and compares it with the stored hash value. If they match, the password is considered correct.
- **Data integrity check**: The data sender can calculate the hash value of the data and send it along; the receiver can recalculate the hash value of the received data and compare it with the received hash value. If they match, the data is considered intact.

For cryptographic applications, to prevent reverse engineering such as deducing the original password from the hash value, hash algorithms need higher-level security features.

- **Unidirectionality**: It should be impossible to deduce any information about the input data from the hash value.
- **Collision resistance**: It should be extremely difficult to find two different inputs that produce the same hash value.
- **Avalanche effect**: Minor changes in the input should lead to significant and unpredictable changes in the output.

Note that **"uniform distribution" and "collision resistance" are two independent concepts**. Satisfying uniform distribution does not necessarily mean collision resistance. For example, under random input `key`, the hash function `key % 100` can produce a uniformly distributed output. However, this hash algorithm is too simple, and all `key` with the same last two digits will have the same output, making it easy to deduce a usable `key` from the hash value, thereby cracking the password.

6.3.2 Design of Hash Algorithms

The design of hash algorithms is a complex issue that requires consideration of many factors. However, for some less demanding scenarios, we can also design some simple hash algorithms.

- **Additive hash**: Add up the ASCII codes of each character in the input and use the total sum as the hash value.
- **Multiplicative hash**: Utilize the non-correlation of multiplication, multiplying each round by a constant, accumulating the ASCII codes of each character into the hash value.
- **XOR hash**: Accumulate the hash value by XORing each element of the input data.
- **Rotating hash**: Accumulate the ASCII code of each character into a hash value, performing a rotation operation on the hash value before each accumulation.

```
// ═══ File: simple_hash.js ═══

/* Additive hash */
function addHash(key) {
    let hash = 0;
    const MODULUS = 1000000007;
    for (const c of key) {
        hash = (hash + c.charCodeAt(0)) % MODULUS;
    }
    return hash;
}
```

```
/* Multiplicative hash */
function mulHash(key) {
    let hash = 0;
    const MODULUS = 1000000007;
    for (const c of key) {
        hash = (31 * hash + c.charCodeAt(0)) % MODULUS;
    }
    return hash;
}

/* XOR hash */
function xorHash(key) {
    let hash = 0;
    const MODULUS = 1000000007;
    for (const c of key) {
        hash ^= c.charCodeAt(0);
    }
    return hash % MODULUS;
}

/* Rotational hash */
function rotHash(key) {
    let hash = 0;
    const MODULUS = 1000000007;
    for (const c of key) {
        hash = ((hash << 4) ^ (hash >> 28) ^ c.charCodeAt(0)) % MODULUS;
    }
    return hash;
}
```

It is observed that the last step of each hash algorithm is to take the modulus of the large prime number 1000000007 to ensure that the hash value is within an appropriate range. It is worth pondering why emphasis is placed on modulo a prime number, or what are the disadvantages of modulo a composite number? This is an interesting question.

To conclude: **Using a large prime number as the modulus can maximize the uniform distribution of hash values**. Since a prime number does not share common factors with other numbers, it can reduce the periodic patterns caused by the modulo operation, thus avoiding hash collisions.

For example, suppose we choose the composite number 9 as the modulus, which can be divided by 3, then all `key` divisible by 3 will be mapped to hash values 0, 3, 6.

$$
\begin{aligned}
\text{modulus} &= 9 \\
\text{key} &= \{0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, \dots\} \\
\text{hash} &= \{0, 3, 6, 0, 3, 6, 0, 3, 6, 0, 3, 6, \dots\}
\end{aligned}
$$

If the input `key` happens to have this kind of arithmetic sequence distribution, then the hash values will cluster, thereby exacerbating hash collisions. Now, suppose we replace `modulus` with the prime number 13, since there are no common factors between `key` and `modulus`, the uniformity of the output hash values will be significantly improved.

$$
\begin{aligned}
\text{modulus} &= 13 \\
\text{key} &= \{0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, \dots\} \\
\text{hash} &= \{0, 3, 6, 9, 12, 2, 5, 8, 11, 1, 4, 7, \dots\}
\end{aligned}
$$

It is worth noting that if the `key` is guaranteed to be randomly and uniformly distributed, then choosing a prime number or a composite number as the modulus can both produce uniformly distributed hash values. However, when the distribution of `key` has some periodicity, modulo a composite number is more likely to result in clustering.

In summary, we usually choose a prime number as the modulus, and this prime number should be large enough to eliminate periodic patterns as much as possible, enhancing the robustness of the hash algorithm.

6.3.3 Common Hash Algorithms

It is not hard to see that the simple hash algorithms mentioned above are quite "fragile" and far from reaching the design goals of hash algorithms. For example, since addition and XOR obey the commutative law, additive hash and XOR hash cannot distinguish strings with the same content but in different order, which may exacerbate hash collisions and cause security issues.

In practice, we usually use some standard hash algorithms, such as MD5, SHA-1, SHA-2, and SHA-3. They can map input data of any length to a fixed-length hash value.

Over the past century, hash algorithms have been in a continuous process of upgrading and optimization. Some researchers strive to improve the performance of hash algorithms, while others, including hackers, are dedicated to finding security issues in hash algorithms. Table 6-2 shows hash algorithms commonly used in practical applications.

- MD5 and SHA-1 have been successfully attacked multiple times and are thus abandoned in various security applications.
- SHA-2 series, especially SHA-256, is one of the most secure hash algorithms to date, with no successful attacks reported, hence commonly used in various security applications and protocols.
- SHA-3 has lower implementation costs and higher computational efficiency compared to SHA-2, but its current usage coverage is not as extensive as the SHA-2 series.

Table 6-2 Common hash algorithms

	MD5	SHA-1	SHA-2	SHA-3
Release Year	1992	1995	2002	2008
Output Length	128 bit	160 bit	256/512 bit	224/256/384/512 bit

	MD5	SHA-1	SHA-2	SHA-3
Hash Collisions	Frequent	Frequent	Rare	Rare
Security Level	Low, has been successfully attacked	Low, has been successfully attacked	High	High
Applications	Abandoned, still used for data integrity checks	Abandoned	Cryptocurrency transaction verification, digital signatures, etc.	Can be used to replace SHA-2

Hash Values in Data Structures

We know that the keys in a hash table can be of various data types such as integers, decimals, or strings. Programming languages usually provide built-in hash algorithms for these data types to calculate the bucket indices in the hash table. Taking Python as an example, we can use the `hash()` function to compute the hash values for various data types.

- The hash values of integers and booleans are their own values.
- The calculation of hash values for floating-point numbers and strings is more complex, and interested readers are encouraged to study this on their own.
- The hash value of a tuple is a combination of the hash values of each of its elements, resulting in a single hash value.
- The hash value of an object is generated based on its memory address. By overriding the hash method of an object, hash values can be generated based on content.

> **Tip**
> Be aware that the definition and methods of the built-in hash value calculation functions in different programming languages vary.

```
// ═══ File: built_in_hash.js ═══

// JavaScript does not provide built-in hash code functions
```

In many programming languages, **only immutable objects can serve as the `key` in a hash table**. If we use a list (dynamic array) as a `key`, when the contents of the list change, its hash value also changes, and we would no longer be able to find the original `value` in the hash table.

Although the member variables of a custom object (such as a linked list node) are mutable, it is hashable. **This is because the hash value of an object is usually generated based on its memory address**, and even if the contents of the object change, the memory address remains the same, so the hash value remains unchanged.

You might have noticed that the hash values output in different consoles are different. **This is because the Python interpreter adds a random salt to the string hash function each time it starts up**. This approach effectively prevents HashDoS attacks and enhances the security of the hash algorithm.

6.4 Summary

1. Key Review

- Given an input `key`, a hash table can retrieve the corresponding `value` in $O(1)$ time, which is highly efficient.
- Common hash table operations include querying, adding key-value pairs, deleting key-value pairs, and traversing the hash table.
- The hash function maps a `key` to an array index, allowing access to the corresponding bucket and retrieval of the `value`.
- Two different keys may end up with the same array index after hashing, leading to erroneous query results. This phenomenon is known as hash collision.
- The larger the capacity of the hash table, the lower the probability of hash collisions. Therefore, hash table expansion can mitigate hash collisions. Similar to array expansion, hash table expansion is costly.
- The load factor, defined as the number of elements divided by the number of buckets, reflects the severity of hash collisions and is often used as a condition to trigger hash table expansion.
- Separate chaining addresses hash collisions by converting each element into a linked list, storing all colliding elements in the same linked list. However, excessively long linked lists can reduce query efficiency, which can be improved by converting the linked lists into red-black trees.
- Open addressing handles hash collisions through multiple probing. Linear probing uses a fixed step size but cannot delete elements and is prone to clustering. Double hashing uses multiple hash functions for probing, which reduces clustering compared to linear probing but increases computational overhead.
- Different programming languages adopt various hash table implementations. For example, Java's `HashMap` uses separate chaining, while Python's `dict` employs open addressing.
- In hash tables, we desire hash algorithms with determinism, high efficiency, and uniform distribution. In cryptography, hash algorithms should also possess collision resistance and the avalanche effect.
- Hash algorithms typically use large prime numbers as moduli to maximize the uniform distribution of hash values and reduce hash collisions.
- Common hash algorithms include MD5, SHA-1, SHA-2, and SHA-3. MD5 is often used for file integrity checks, while SHA-2 is commonly used in secure applications and protocols.
- Programming languages usually provide built-in hash algorithms for data types to calculate bucket indices in hash tables. Generally, only immutable objects are hashable.

2. Q & A

Q: When does the time complexity of a hash table degrade to $O(n)$?

The time complexity of a hash table can degrade to $O(n)$ when hash collisions are severe. When the hash function is well-designed, the capacity is set appropriately, and collisions are evenly distributed, the time complexity is $O(1)$. We usually consider the time complexity to be $O(1)$ when using built-in hash tables in programming languages.

Q: Why not use the hash function $f(x) = x$? This would eliminate collisions.

Under the hash function $f(x) = x$, each element corresponds to a unique bucket index, which is equivalent to an array. However, the input space is usually much larger than the output space (array length), so the last step of a hash function is often to take the modulo of the array length. In other words, the goal of a hash table is to map a larger state space to a smaller one while providing $O(1)$ query efficiency.

Q: Why can hash tables be more efficient than arrays, linked lists, or binary trees, even though hash tables are implemented using these structures?

Firstly, hash tables have higher time efficiency but lower space efficiency. A significant portion of memory in hash tables remains unused.

Secondly, hash tables are only more time-efficient in specific use cases. If a feature can be implemented with the same time complexity using an array or a linked list, it's usually faster than using a hash table. This is because the computation of the hash function incurs overhead, making the constant factor in the time complexity larger.

Lastly, the time complexity of hash tables can degrade. For example, in separate chaining, we perform search operations in a linked list or red-black tree, which still risks degrading to $O(n)$ time.

Q: Does double hashing also have the flaw of not being able to delete elements directly? Can space marked as deleted be reused?

Double hashing is a form of open addressing, and all open addressing methods have the drawback of not being able to delete elements directly; they require marking elements as deleted. Marked spaces can be reused. When inserting new elements into the hash table, and the hash function points to a position marked as deleted, that position can be used by the new element. This maintains the probing sequence of the hash table while ensuring efficient use of space.

Q: Why do hash collisions occur during the search process in linear probing?

During the search process, the hash function points to the corresponding bucket and key-value pair. If the `key` doesn't match, it indicates a hash collision. Therefore, linear probing will search downward at a predetermined step size until the correct key-value pair is found or the search fails.

Q: Why can expanding a hash table alleviate hash collisions?

The last step of a hash function often involves taking the modulo of the array length n, to keep the output within the array index range. When expanding, the array length n changes, and the indices corresponding to the keys may also change. Keys that were previously mapped to the same bucket might be distributed across multiple buckets after expansion, thereby mitigating hash collisions.

Chapter 7. Tree

Abstract

Towering trees are full of vitality, with deep roots and lush leaves, spreading branches and flourishing.

They show us the vivid form of divide and conquer in data.

Chapter contents

7.1 Binary Tree

A binary tree is a non-linear data structure that represents the derivation relationship between "ancestors" and "descendants" and embodies the divide-and-conquer logic of "one divides into two". Similar to a linked list, the basic unit of a binary tree is a node, and each node contains a value, a reference to its left child node, and a reference to its right child node.

```
/* Binary tree node */
class TreeNode {
    val; // Node value
    left; // Pointer to left child node
    right; // Pointer to right child node
    constructor(val, left, right) {
        this.val = val === undefined ? 0 : val;
        this.left = left === undefined ? null : left;
        this.right = right === undefined ? null : right;
    }
}
```

Each node has two references (pointers), pointing respectively to the left-child node and right-child node. This node is called the parent node of these two child nodes. When given a node of a binary tree, we call the tree formed by this node's left child and all nodes below it the left subtree of this node. Similarly, the right subtree can be defined.

In a binary tree, except leaf nodes, all other nodes contain child nodes and non-empty subtrees. As shown in Figure 7-1, if "Node 2" is regarded as a parent node, its left and right child nodes are "Node 4" and "Node 5" respectively. The left subtree is formed by "Node 4" and all nodes beneath it, while the right subtree is formed by "Node 5" and all nodes beneath it.

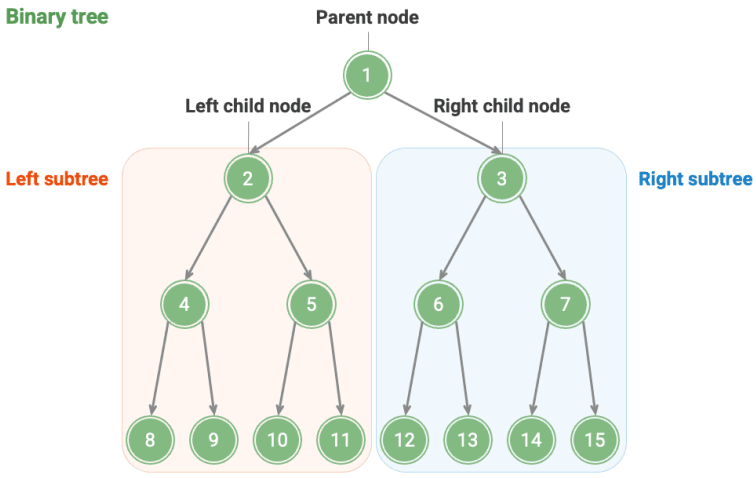


Figure 7-1 Parent Node, child Node, subtree

7.1.1 Common Terminology of Binary Trees

The commonly used terminology of binary trees is shown in Figure 7-2.

- Root node: The node at the top level of a binary tree, which does not have a parent node.
- Leaf node: A node that does not have any child nodes, with both of its pointers pointing to `None`.
- Edge: A line segment that connects two nodes, representing a reference (pointer) between the nodes.
- The level of a node: It increases from top to bottom, with the root node being at level 1.
- The degree of a node: The number of child nodes that a node has. In a binary tree, the degree can be 0, 1, or 2.
- The height of a binary tree: The number of edges from the root node to the farthest leaf node.
- The depth of a node: The number of edges from the root node to the node.
- The height of a node: The number of edges from the farthest leaf node to the node.

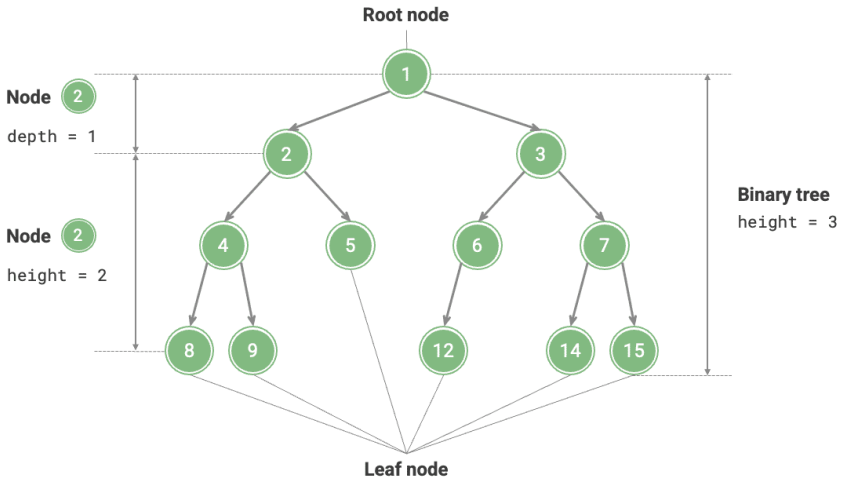


Figure 7-2 Common Terminology of Binary Trees

Tip
Please note that we usually define "height" and "depth" as "the number of edges traversed", but some questions or textbooks may define them as "the number of nodes traversed". In this case, both height and depth need to be incremented by 1.

7.1.2 Basic Operations of Binary Trees

1. Initializing a Binary Tree

Similar to a linked list, the initialization of a binary tree involves first creating the nodes and then establishing the references (pointers) between them.

```
// ═══ File: binary_tree.js ═══

/* Initializing a binary tree */
// Initializing nodes
let n1 = new TreeNode(1),
    n2 = new TreeNode(2),
    n3 = new TreeNode(3),
    n4 = new TreeNode(4),
    n5 = new TreeNode(5);
// Linking references (pointers) between nodes
n1.left = n2;
n1.right = n3;
n2.left = n4;
n2.right = n5;
```

2. Inserting and Removing Nodes

Similar to a linked list, inserting and removing nodes in a binary tree can be achieved by modifying pointers. Figure 7-3 provides an example.

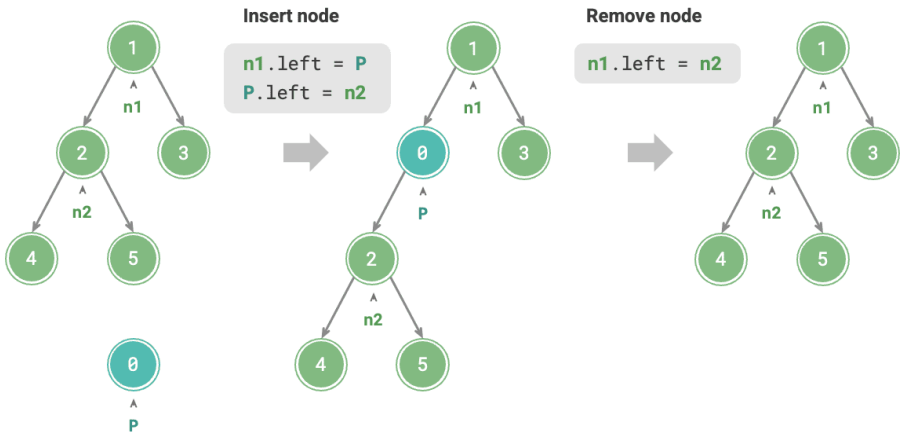


Figure 7-3 Inserting and removing nodes in a binary tree

```
// ═══ File: binary_tree.js ═══

/* Inserting and removing nodes */
let P = new TreeNode(0);
// Inserting node P between n1 and n2
n1.left = P;
P.left = n2;
// Removing node P
n1.left = n2;
```

Tip
It should be noted that inserting nodes may change the original logical structure of the binary tree, while removing nodes typically involves removing the node and all its subtrees. Therefore, in a binary tree, insertion and removal are usually performed through a set of operations to achieve meaningful outcomes.

7.1.3 Common Types of Binary Trees

1. Perfect Binary Tree

As shown in Figure 7-4, a perfect binary tree has all levels completely filled with nodes. In a perfect binary tree, leaf nodes have a degree of 0, while all other nodes have a degree of 2. If the tree height is h, the total number of nodes is $2^{h+1} - 1$, exhibiting a standard exponential relationship that reflects the common phenomenon of cell division in nature.

Tip
Please note that in the Chinese community, a perfect binary tree is often referred to as a full binary tree.

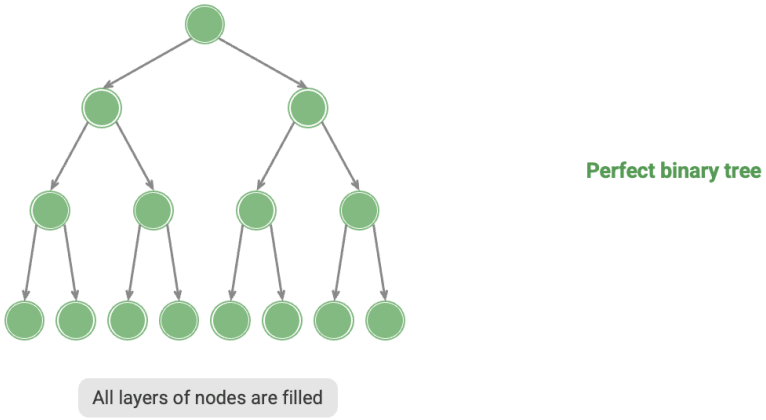


Figure 7-4 Perfect binary tree

2. Complete Binary Tree

As shown in Figure 7-5, a complete binary tree only allows the bottom level to be incompletely filled, and the nodes at the bottom level must be filled continuously from left to right. Note that a perfect binary tree is also a complete binary tree.

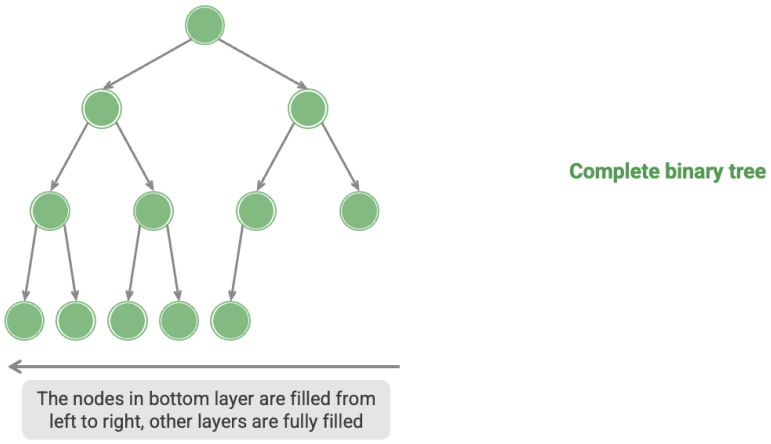


Figure 7-5 Complete binary tree

3. Full Binary Tree

As shown in Figure 7-6, in a full binary tree, all nodes except leaf nodes have two child nodes.

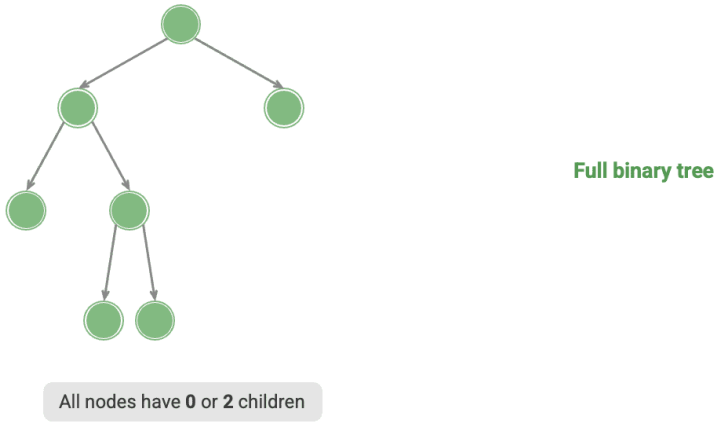


Figure 7-6 Full binary tree

4. Balanced Binary Tree

As shown in Figure 7-7, in a balanced binary tree, the absolute difference between the height of the left and right subtrees of any node does not exceed 1.

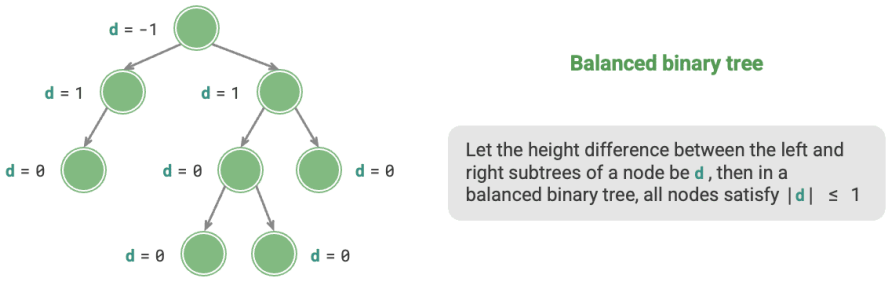


Figure 7-7 Balanced binary tree

7.1.4 Degeneration of Binary Trees

Figure 7-8 shows the ideal and degenerate structures of binary trees. When every level of a binary tree is filled, it reaches the "perfect binary tree" state; when all nodes are biased toward one side, the binary tree degenerates into a "linked list".

- A perfect binary tree is the ideal case, fully leveraging the "divide and conquer" advantage of binary trees.
- A linked list represents the other extreme, where all operations become linear operations with time complexity degrading to $O(n)$.

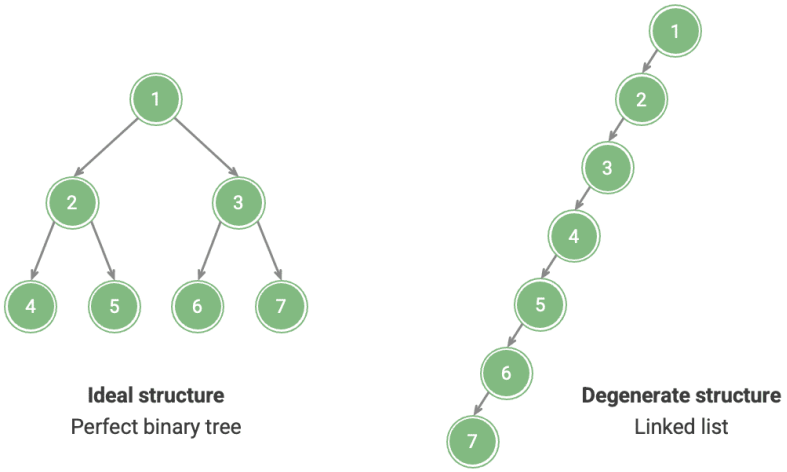


Figure 7-8 The Best and Worst Structures of Binary Trees

As shown in Table 7-1, in the best and worst structures, the binary tree achieves either maximum or minimum values for leaf node count, total number of nodes, and height.

Table 7-1 The Best and Worst Structures of Binary Trees

	Perfect binary tree	Linked list
Number of nodes at level i	2^{i-1}	1
Number of leaf nodes in a tree with height h	2^h	1
Total number of nodes in a tree with height h	$2^{h+1}-1$	$h+1$
Height of a tree with n total nodes	$\log_2(n+1)-1$	$n-1$

7.2 Binary Tree Traversal

From a physical structure perspective, a tree is a data structure based on linked lists. Hence, its traversal method involves accessing nodes one by one through pointers. However, a tree is a non-linear data structure, which makes traversing a tree more complex than traversing a linked list, requiring the assistance of search algorithms.

The common traversal methods for binary trees include level-order traversal, pre-order traversal, in-order traversal, and post-order traversal.

7.2.1 Level-Order Traversal

As shown in Figure 7-9, level-order traversal traverses the binary tree from top to bottom, layer by layer. Within each level, it visits nodes from left to right.

Level-order traversal is essentially breadth-first traversal, also known as breadth-first search (BFS), which embodies a "expanding outward circle by circle" layer-by-layer traversal method.

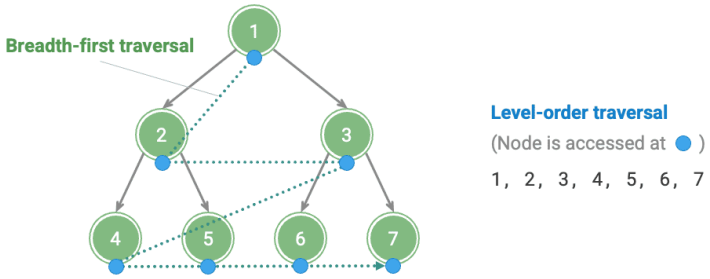


Figure 7-9 Level-order traversal of a binary tree

1. Code Implementation

Breadth-first traversal is typically implemented with the help of a "queue". The queue follows the "first in, first out" rule, while breadth-first traversal follows the "layer-by-layer progression" rule; the underlying ideas of the two are consistent. The implementation code is as follows:

```javascript
// == File: binary_tree_bfs.js ==

/* Level-order traversal */
function levelOrder(root) {
    // Initialize queue, add root node
    const queue = [root];
    // Initialize a list to save the traversal sequence
    const list = [];
    while (queue.length) {
        let node = queue.shift(); // Dequeue
        list.push(node.val); // Save node value
        if (node.left) queue.push(node.left); // Left child node enqueue
        if (node.right) queue.push(node.right); // Right child node enqueue
    }
    return list;
}
```

2. Complexity Analysis

- **Time complexity is $O(n)$**: All nodes are visited once, using $O(n)$ time, where n is the number of nodes.
- **Space complexity is $O(n)$**: In the worst case, i.e., a full binary tree, before traversing to the bottom level, the queue contains at most $(n+1)/2$ nodes simultaneously, occupying $O(n)$ space.

7.2.2 Preorder, Inorder, and Postorder Traversal

Correspondingly, preorder, inorder, and postorder traversals all belong to depth-first traversal, also known as depth-first search (DFS), which embodies a "first go to the end, then backtrack and continue" traversal method.

Figure 7-10 shows how depth-first traversal works on a binary tree. **Depth-first traversal is like "walking" around the perimeter of the entire binary tree**, encountering three positions at each node, corresponding to preorder, inorder, and postorder traversal.

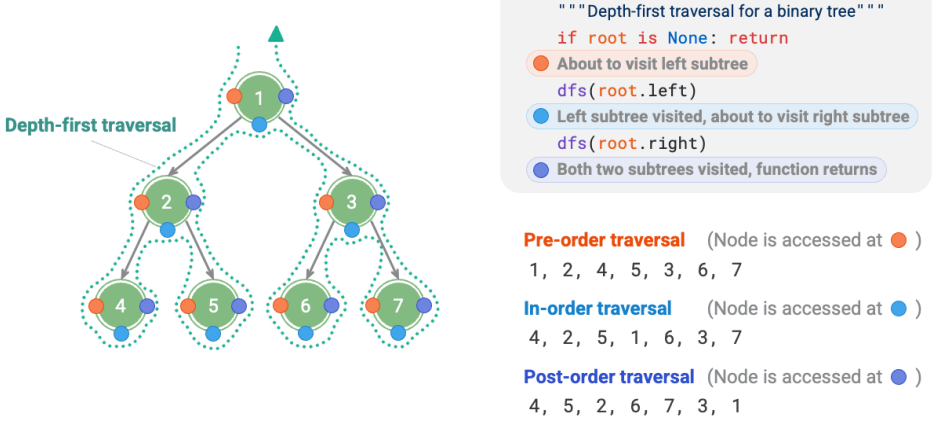


Figure 7-10 Preorder, inorder, and postorder traversal of a binary tree

1. Code Implementation

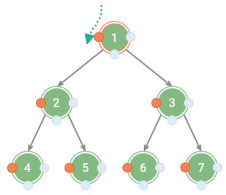
Depth-first search is usually implemented based on recursion:

```
// ═══ File: binary_tree_dfs.js ═══

/* Preorder traversal */
function preOrder(root) {
    if (root === null) return;
    // Visit priority: root node -> left subtree -> right subtree
    list.push(root.val);
    preOrder(root.left);
    preOrder(root.right);
}

/* Inorder traversal */
function inOrder(root) {
    if (root === null) return;
    // Visit priority: left subtree -> root node -> right subtree
    inOrder(root.left);
    list.push(root.val);
    inOrder(root.right);
}

/* Postorder traversal */
function postOrder(root) {
    if (root === null) return;
    // Visit priority: left subtree -> right subtree -> root node
    postOrder(root.left);
    postOrder(root.right);
    list.push(root.val);
}
```

> **Tip**
> Depth-first search can also be implemented based on iteration, interested readers can study this on their own.

Figure 7-11 shows the recursive process of preorder traversal of a binary tree, which can be divided into two opposite parts: "recursion" and "return".

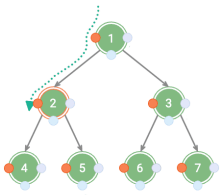
1. "Recursion" means opening a new method, where the program accesses the next node in this process.
2. "Return" means the function returns, indicating that the current node has been fully visited.



Calling
Returning

Pre-order traversal (Node is accessed at ●)
1

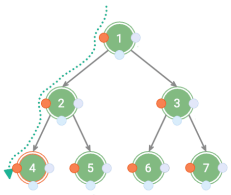
Step 1



Calling
Returning

Pre-order traversal (Node is accessed at ●)
1, 2

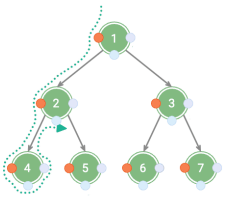
Step 2



Calling
Returning

Pre-order traversal (Node is accessed at ●)
1, 2, 4

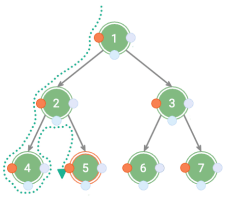
Step 3



Calling
Returning

Pre-order traversal (Node is accessed at ●)
1, 2, 4

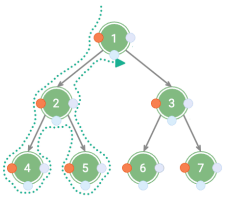
Step 4



Calling
Returning

Pre-order traversal (Node is accessed at ●)
1, 2, 4, 5

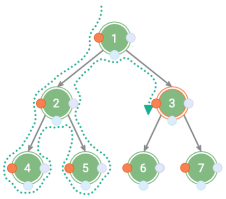
Step 5



Calling
Returning

Pre-order traversal (Node is accessed at ●)
1, 2, 4, 5

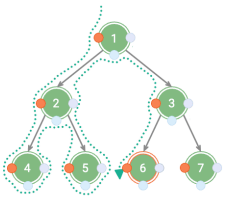
Step 6



Calling
Returning

Pre-order traversal (Node is accessed at ●)
1, 2, 4, 5, 3

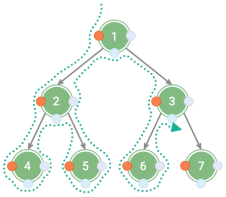
Step 7



Calling
Returning

Pre-order traversal (Node is accessed at ●)
1, 2, 4, 5, 3, 6

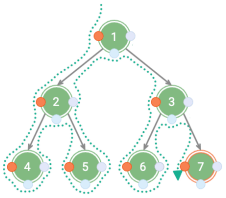
Step 8



Calling
Returning

Pre-order traversal (Node is accessed at ●)
1, 2, 4, 5, 3, 6

Step 9



Calling
Returning

Pre-order traversal (Node is accessed at ●)
1, 2, 4, 5, 3, 6, 7

Step 10

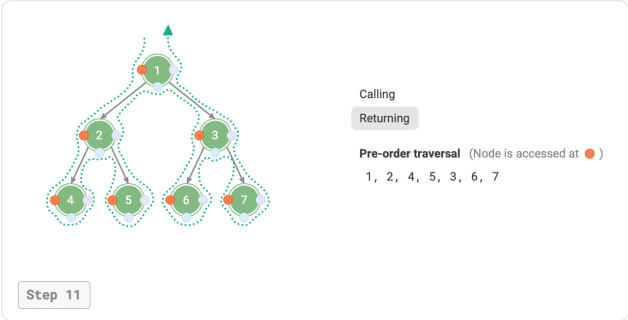


Figure 7-11 The recursive process of preorder traversal

2. Complexity Analysis

- **Time complexity is $O(n)$**: All nodes are visited once, using $O(n)$ time.
- **Space complexity is $O(n)$**: In the worst case, i.e., the tree degenerates into a linked list, the recursion depth reaches n, and the system occupies $O(n)$ stack frame space.

7.3 Array Representation of Binary Trees

Under the linked list representation, the storage unit of a binary tree is a node `TreeNode`, and nodes are connected by pointers. The previous section introduced the basic operations of binary trees under the linked list representation.

So, can we use an array to represent a binary tree? The answer is yes.

7.3.1 Representing Perfect Binary Trees

Let's analyze a simple case first. Given a perfect binary tree, we store all nodes in an array according to the order of level-order traversal, where each node corresponds to a unique array index.

Based on the characteristics of level-order traversal, we can derive a "mapping formula" between parent node index and child node indices: **If a node's index is i, then its left child index is $2i + 1$ and its right child index is $2i + 2$.** Figure 7-12 shows the mapping relationships between various node indices.

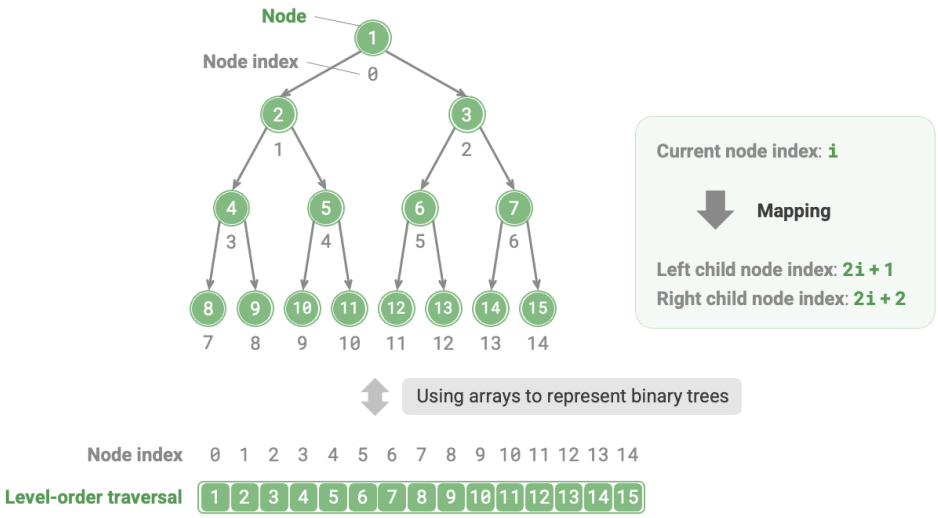


Figure 7-12 Array representation of a perfect binary tree

The mapping formula plays a role similar to the node references (pointers) in linked lists. Given any node in the array, we can access its left (right) child node using the mapping formula.

7.3.2 Representing Any Binary Tree

Perfect binary trees are a special case; in the middle levels of a binary tree, there are typically many `None` values. Since the level-order traversal sequence does not include these `None` values, we cannot infer the number and distribution of `None` values based on this sequence alone. **This means multiple binary tree structures can correspond to the same level-order traversal sequence**.

As shown in Figure 7-13, given a non-perfect binary tree, the above method of array representation fails.

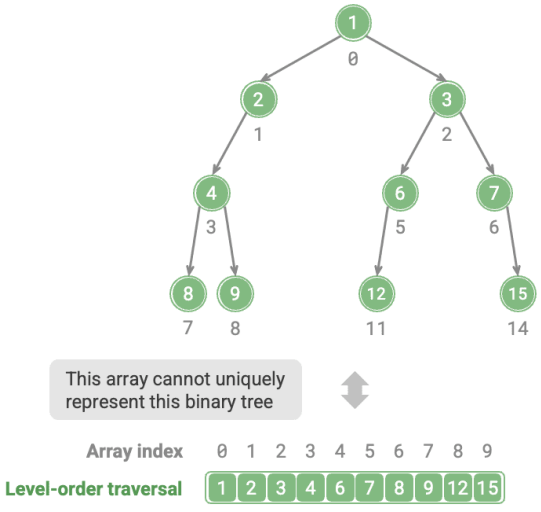


Figure 7-13 Level-order traversal sequence corresponds to multiple binary tree possibilities

To solve this problem, **we can consider explicitly writing out all `None` values in the level-order traversal sequence**. As shown in Figure 7-14, after this treatment, the level-order traversal sequence can uniquely represent a binary tree. Example code is as follows:

```
/* Array representation of a binary tree */
// Using null to represent empty slots
let tree = [1, 2, 3, 4, null, 6, 7, 8, 9, null, null, 12, null, null, 15];
```

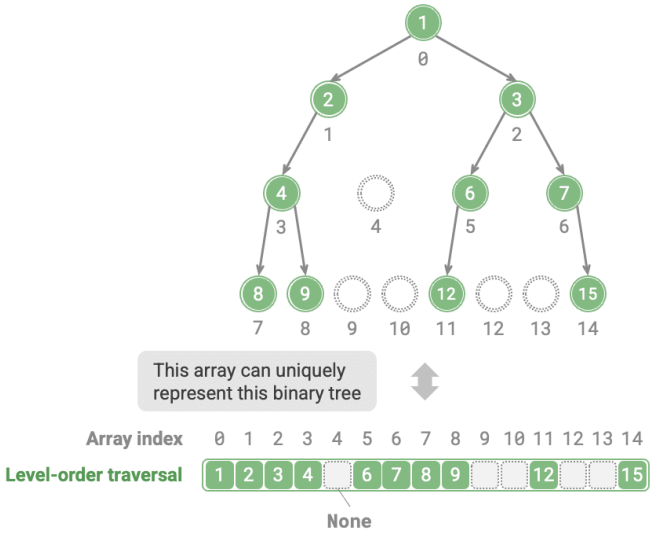


Figure 7-14 Array representation of any type of binary tree

It's worth noting that **complete binary trees are very well-suited for array representation**. Recalling the definition of a complete binary tree, `None` only appears at the bottom level and towards the right, **meaning all `None` values must appear at the end of the level-order traversal sequence**.

This means that when using an array to represent a complete binary tree, it's possible to omit storing all `None` values, which is very convenient. Figure 7-15 gives an example.

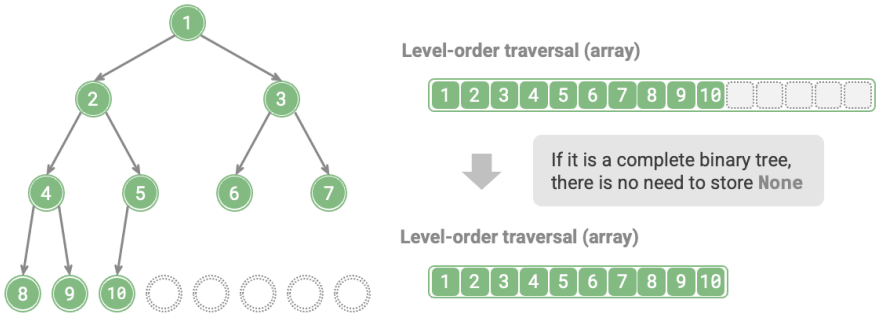


Figure 7-15 Array representation of a complete binary tree

The following code implements a binary tree based on array representation, including the following operations:

- Given a certain node, obtain its value, left (right) child node, and parent node.
- Obtain the preorder, inorder, postorder, and level-order traversal sequences.

```
// ═══ File: array_binary_tree.js ═══

/* Binary tree class represented by array */
class ArrayBinaryTree {
    #tree;

    /* Constructor */
    constructor(arr) {
        this.#tree = arr;
    }

    /* List capacity */
    size() {
        return this.#tree.length;
    }

    /* Get value of node at index i */
    val(i) {
        // If index out of bounds, return null to represent empty position
        if (i < 0 || i >= this.size()) return null;
        return this.#tree[i];
    }

    /* Get index of left child node of node at index i */
    left(i) {
        return 2 * i + 1;
    }

    /* Get index of right child node of node at index i */
    right(i) {
        return 2 * i + 2;
    }

    /* Get index of parent node of node at index i */
    parent(i) {
        return Math.floor((i - 1) / 2); // Floor division
    }

    /* Level-order traversal */
    levelOrder() {
        let res = [];
        // Traverse array directly
        for (let i = 0; i < this.size(); i++) {
            if (this.val(i) !== null) res.push(this.val(i));
        }
        return res;
    }

    /* Depth-first traversal */
    #dfs(i, order, res) {
        // If empty position, return
```

```javascript
        if (this.val(i) === null) return;
        // Preorder traversal
        if (order === 'pre') res.push(this.val(i));
        this.#dfs(this.left(i), order, res);
        // Inorder traversal
        if (order === 'in') res.push(this.val(i));
        this.#dfs(this.right(i), order, res);
        // Postorder traversal
        if (order === 'post') res.push(this.val(i));
    }

    /* Preorder traversal */
    preOrder() {
        const res = [];
        this.#dfs(0, 'pre', res);
        return res;
    }

    /* Inorder traversal */
    inOrder() {
        const res = [];
        this.#dfs(0, 'in', res);
        return res;
    }

    /* Postorder traversal */
    postOrder() {
        const res = [];
        this.#dfs(0, 'post', res);
        return res;
    }
}
```

7.3.3 Advantages and Limitations

The array representation of binary trees has the following advantages:

- Arrays are stored in contiguous memory space, which is cache-friendly, allowing faster access and traversal.
- It does not require storing pointers, which saves space.
- It allows random access to nodes.

However, the array representation also has some limitations:

- Array storage requires contiguous memory space, so it is not suitable for storing trees with a large amount of data.
- Adding or removing nodes requires array insertion and deletion operations, which have lower efficiency.
- When there are many `None` values in the binary tree, the proportion of node data contained in the array is low, leading to lower space utilization.

7.4 Binary Search Tree

As shown in Figure 7-16, a binary search tree satisfies the following conditions.

1. For the root node, the value of all nodes in the left subtree $<$ the value of the root node $<$ the value of all nodes in the right subtree.
2. The left and right subtrees of any node are also binary search trees, i.e., they satisfy condition `1.` as well.

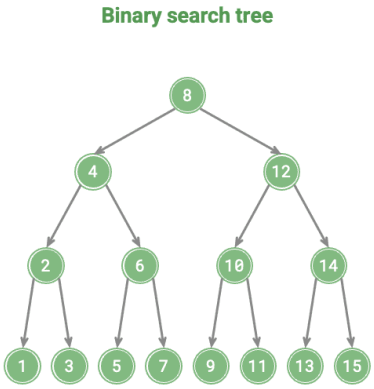


Figure 7-16 Binary search tree

7.4.1 Operations on a Binary Search Tree

We encapsulate the binary search tree as a class `BinarySearchTree` and declare a member variable `root` pointing to the tree's root node.

1. Searching for a Node

Given a target node value `num`, we can search according to the properties of the binary search tree. As shown in Figure 7-17, we declare a node `cur` and start from the binary tree's root node `root`, looping to compare the node value `cur.val` with `num`.

- If `cur.val < num`, it means the target node is in `cur`'s right subtree, thus execute `cur = cur.right`.
- If `cur.val > num`, it means the target node is in `cur`'s left subtree, thus execute `cur = cur.left`.
- If `cur.val = num`, it means the target node is found, exit the loop, and return the node.

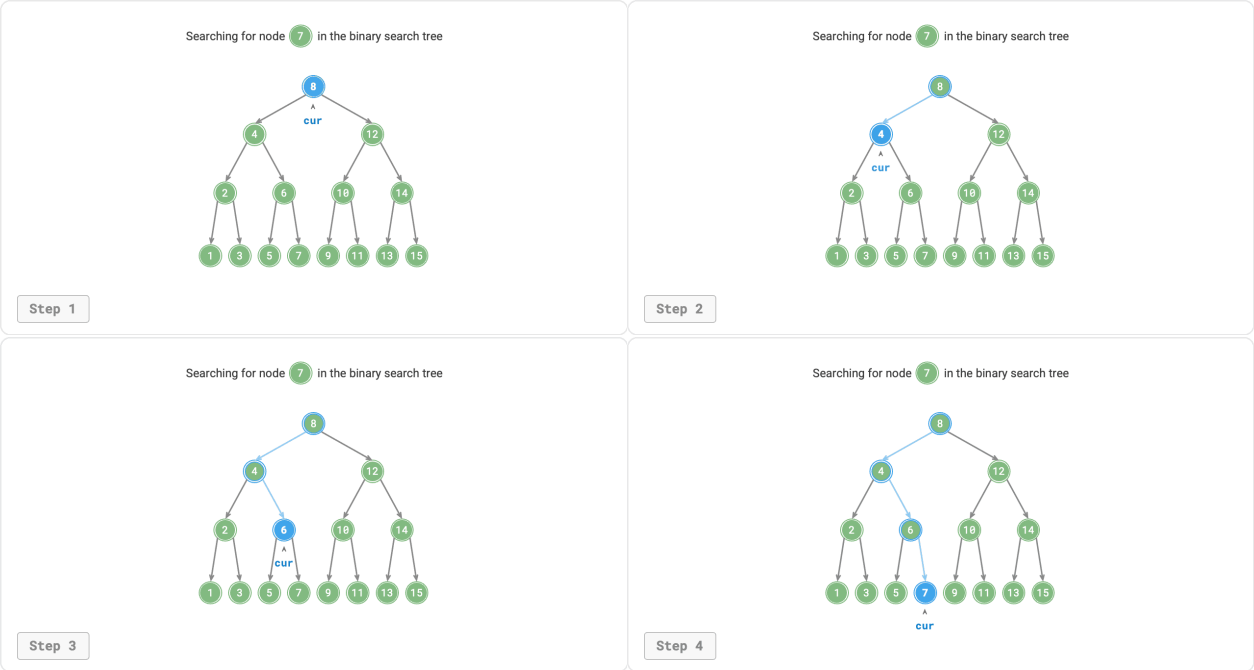


Figure 7-17 Example of searching for a node in a binary search tree

The search operation in a binary search tree works on the same principle as the binary search algorithm, both eliminating half of the cases in each round. The number of loop iterations is at most the height of the binary tree. When the binary tree is balanced, it uses $O(\log n)$ time. The example code is as follows:

```
// ═══ File: binary_search_tree.js ═══

/* Search node */
search(num) {
    let cur = this.root;
    // Loop search, exit after passing leaf node
    while (cur !== null) {
        // Target node is in cur's right subtree
        if (cur.val < num) cur = cur.right;
        // Target node is in cur's left subtree
        else if (cur.val > num) cur = cur.left;
        // Found target node, exit loop
        else break;
    }
    // Return target node
    return cur;
}
```

2. Inserting a Node

Given an element `num` to be inserted, in order to maintain the property of the binary search tree "left subtree < root node < right subtree," the insertion process is as shown in Figure 7-18.

1. **Finding the insertion position**: Similar to the search operation, start from the root node and loop downward searching according to the size relationship between the current node value and `num`, until passing the leaf node (traversing to `None`) and then exit the loop.
2. **Insert the node at that position**: Initialize node `num` and place it at the `None` position.

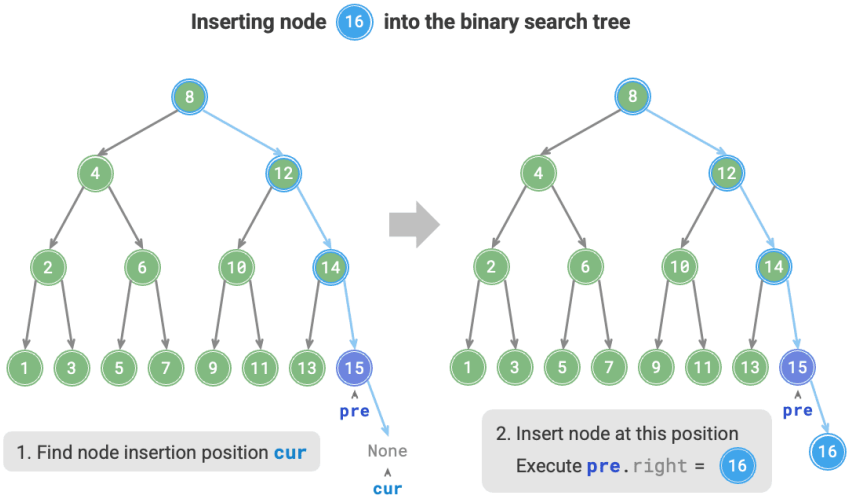


Figure 7-18 Inserting a node into a binary search tree

In the code implementation, note the following two points:

- Binary search trees do not allow duplicate nodes; otherwise, it would violate its definition. Therefore, if the node to be inserted already exists in the tree, the insertion is not performed and it returns directly.
- To implement the node insertion, we need to use node `pre` to save the node from the previous loop iteration. This way, when traversing to `None`, we can obtain its parent node, thereby completing the node insertion operation.

```
// === File: binary_search_tree.js ===

/* Insert node */
insert(num) {
    // If tree is empty, initialize root node
    if (this.root === null) {
        this.root = new TreeNode(num);
        return;
    }
    let cur = this.root,
        pre = null;
    // Loop search, exit after passing leaf node
    while (cur !== null) {
        // Found duplicate node, return directly
        if (cur.val === num) return;
        pre = cur;
        // Insertion position is in cur's right subtree
        if (cur.val < num) cur = cur.right;
```

```
        // Insertion position is in cur's left subtree
        else cur = cur.left;
    }
    // Insert node
    const node = new TreeNode(num);
    if (pre.val < num) pre.right = node;
    else pre.left = node;
}
```

Similar to searching for a node, inserting a node uses $O(\log n)$ time.

3. Removing a Node

First, find the target node in the binary tree, then remove it. Similar to node insertion, we need to ensure that after the removal operation is completed, the binary search tree's property of "left subtree $<$ root node $<$ right subtree" is still maintained. Therefore, depending on the number of child nodes the target node has, we divide it into 0, 1, and 2 three cases, and execute the corresponding node removal operations.

As shown in Figure 7-19, when the degree of the node to be removed is 0, it means the node is a leaf node and can be directly removed.

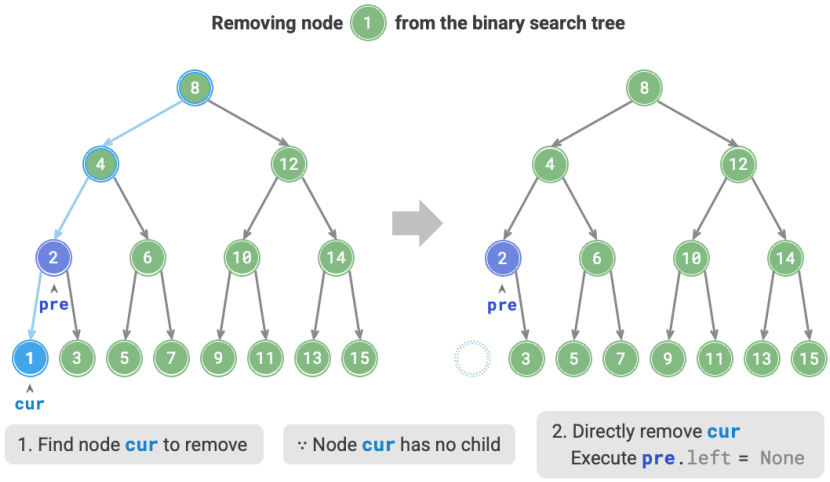


Figure 7-19 Removing a node in a binary search tree (degree 0)

As shown in Figure 7-20, when the degree of the node to be removed is 1, replacing the node to be removed with its child node is sufficient.

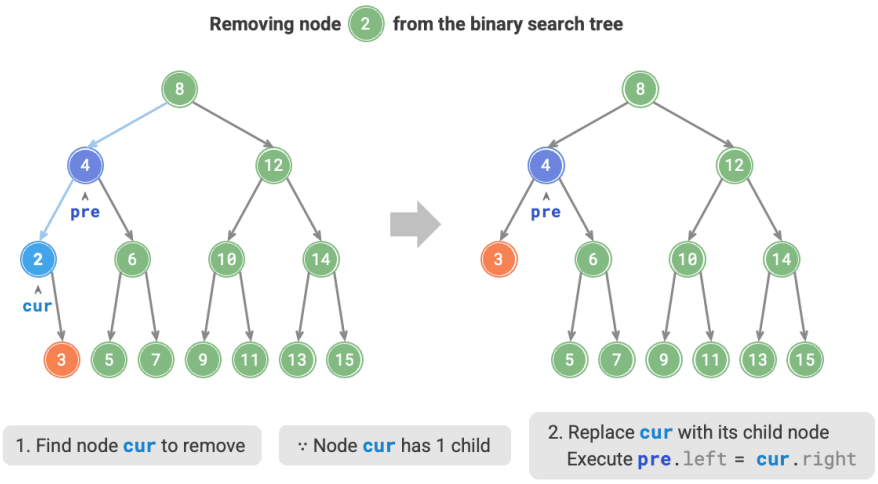
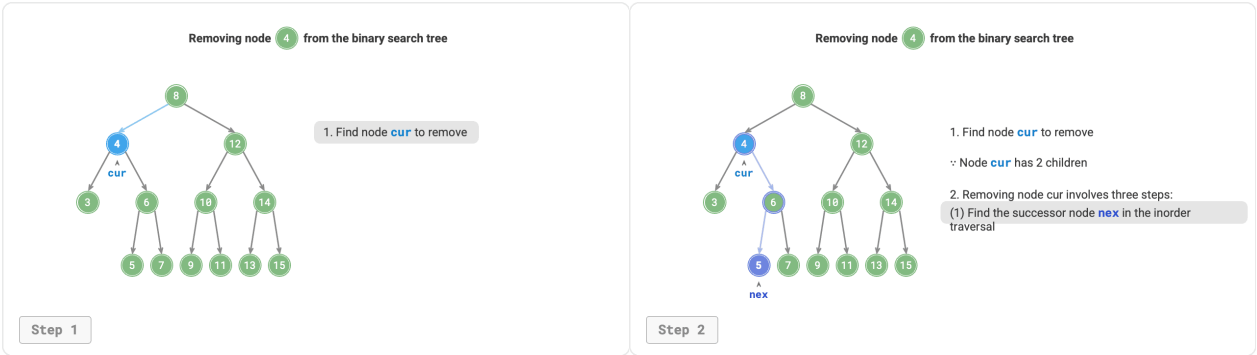


Figure 7-20 Removing a node in a binary search tree (degree 1)

When the degree of the node to be removed is 2, we cannot directly remove it; instead, we need to use a node to replace it. To maintain the binary search tree's property of "left subtree $<$ root node $<$ right subtree," **this node can be either the smallest node in the right subtree or the largest node in the left subtree**.

Assuming we choose the smallest node in the right subtree (the next node in the inorder traversal), the removal process is as shown in Figure 7-21.

1. Find the next node of the node to be removed in the "inorder traversal sequence," denoted as `tmp`.
2. Replace the value of the node to be removed with the value of `tmp`, and recursively remove node `tmp` in the tree.



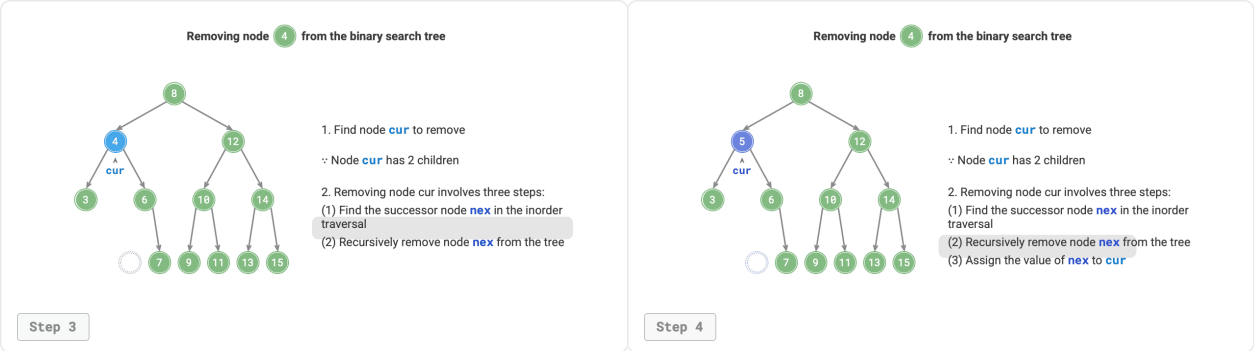


Figure 7-21 Removing a node in a binary search tree (degree 2)

The node removal operation also uses $O(\log n)$ time, where finding the node to be removed requires $O(\log n)$ time, and obtaining the inorder successor node requires $O(\log n)$ time. Example code is as follows:

```
// === File: binary_search_tree.js ===

/* Remove node */
remove(num) {
    // If tree is empty, return directly
    if (this.root === null) return;
    let cur = this.root,
        pre = null;
    // Loop search, exit after passing leaf node
    while (cur !== null) {
        // Found node to delete, exit loop
        if (cur.val === num) break;
        pre = cur;
        // Node to delete is in cur's right subtree
        if (cur.val < num) cur = cur.right;
        // Node to delete is in cur's left subtree
        else cur = cur.left;
    }
    // If no node to delete, return directly
    if (cur === null) return;
    // Number of child nodes = 0 or 1
    if (cur.left === null || cur.right === null) {
        // When number of child nodes = 0 / 1, child = null / that child node
        const child = cur.left !== null ? cur.left : cur.right;
        // Delete node cur
        if (cur !== this.root) {
            if (pre.left === cur) pre.left = child;
            else pre.right = child;
        } else {
            // If deleted node is root node, reassign root node
            this.root = child;
        }
    }
    // Number of child nodes = 2
    else {
        // Get next node of cur in inorder traversal
        let tmp = cur.right;
        while (tmp.left !== null) {
            tmp = tmp.left;
```

```
        }
        // Recursively delete node tmp
        this.remove(tmp.val);
        // Replace cur with tmp
        cur.val = tmp.val;
    }
}
```

4. Inorder Traversal Is Ordered

As shown in Figure 7-22, the inorder traversal of a binary tree follows the "left → root → right" traversal order, while the binary search tree satisfies the "left child node < root node < right child node" size relationship.

This means that when performing an inorder traversal in a binary search tree, the next smallest node is always traversed first, thus yielding an important property: **The inorder traversal sequence of a binary search tree is ascending**.

Using the property of inorder traversal being ascending, we can obtain ordered data in a binary search tree in only $O(n)$ time, without the need for additional sorting operations, which is very efficient.

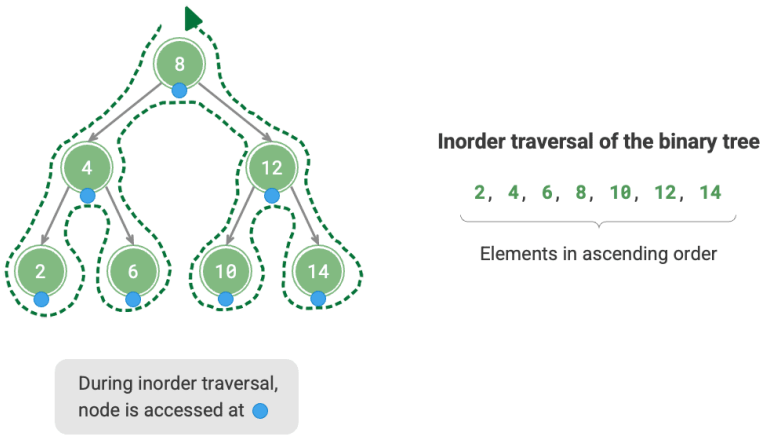


Figure 7-22 Inorder traversal sequence of a binary search tree

7.4.2 Efficiency of Binary Search Trees

Given a set of data, we consider using an array or a binary search tree for storage. Observing Table 7-2, all operations in a binary search tree have logarithmic time complexity, providing stable and efficient performance. Arrays are more efficient than binary search trees only in scenarios with high-frequency additions and low-frequency searches and deletions.

Table 7-2 Efficiency comparison between arrays and search trees

	Unsorted array	Binary search tree
Search element	$O(n)$	$O(\log n)$
Insert element	$O(1)$	$O(\log n)$
Remove element	$O(n)$	$O(\log n)$

In the ideal case, a binary search tree is "balanced," such that any node can be found within $\log n$ loop iterations.

However, if we continuously insert and remove nodes in a binary search tree, it may degenerate into a linked list as shown in Figure 7-23, where the time complexity of various operations also degrades to $O(n)$.

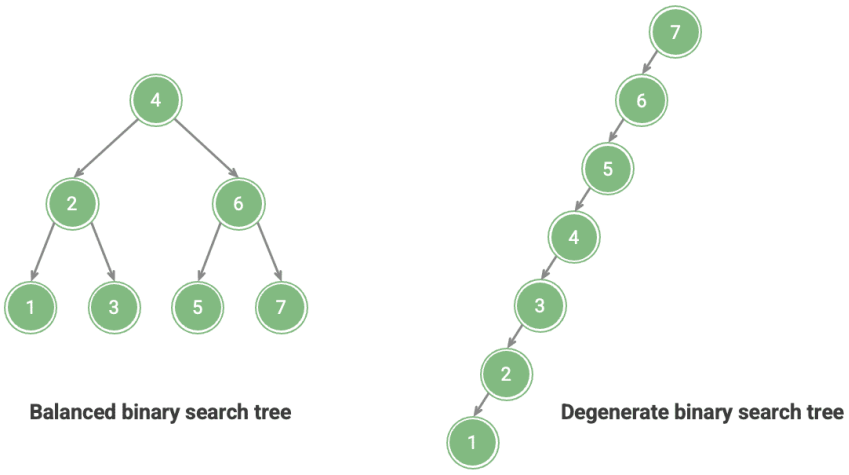


Figure 7-23 Degradation of a binary search tree

7.4.3 Common Applications of Binary Search Trees

- Used as multi-level indexes in systems to implement efficient search, insertion, and removal operations.
- Serves as the underlying data structure for certain search algorithms.
- Used to store data streams to maintain their ordered state.

7.5 Avl Tree *

In the "Binary Search Tree" section, we mentioned that after multiple insertion and removal operations, a binary search tree may degenerate into a linked list. In this case, the time complexity of all operations degrades from $O(\log n)$ to $O(n)$.

As shown in Figure 7-24, after two node removal operations, this binary search tree will degrade into a linked list.

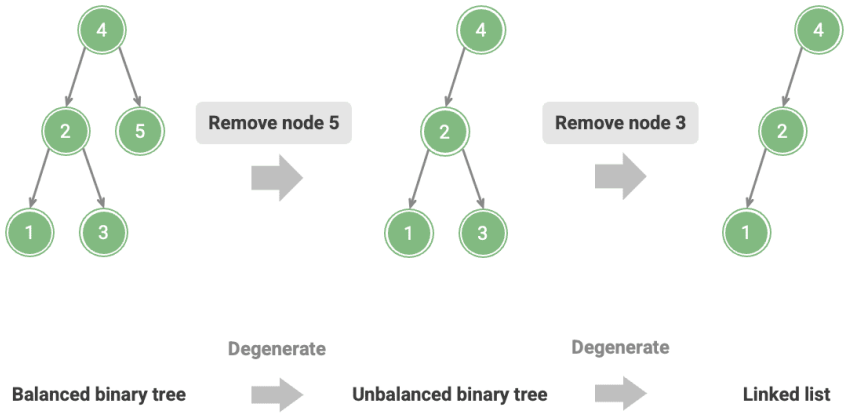


Figure 7-24 Degradation of an AVL tree after removing nodes

For example, in the perfect binary tree shown in Figure 7-25, after inserting two nodes, the tree will lean heavily to the left, and the time complexity of search operations will also degrade.

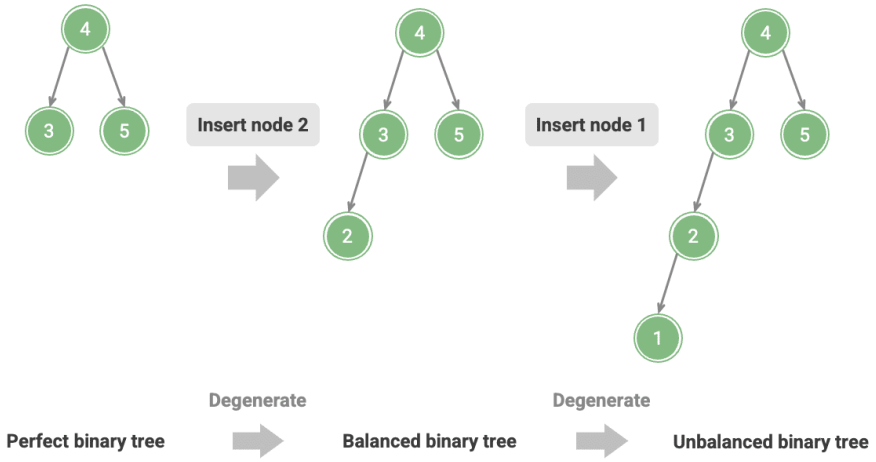


Figure 7-25 Degradation of an AVL tree after inserting nodes

In 1962, G. M. Adelson-Velsky and E. M. Landis proposed the AVL tree in their paper "An algorithm for the organization of information". The paper described in detail a series of operations ensuring that after continuously adding and removing nodes, the AVL tree does not degenerate, thus keeping the time complexity of various operations at the $O(\log n)$ level. In other words, in scenarios requiring frequent insertions, deletions, searches, and modifications, the AVL tree can always maintain efficient data operation performance, making it very valuable in applications.

7.5.1 Common Terminology in Avl Trees

An AVL tree is both a binary search tree and a balanced binary tree, simultaneously satisfying all the properties of these two types of binary trees, hence it is a balanced binary search tree.

1. Node Height

Since the operations related to AVL trees require obtaining node heights, we need to add a `height` variable to the node class:

```
/* AVL tree node */
class TreeNode {
    val; // Node value
    height; // Node height
    left; // Left child pointer
    right; // Right child pointer
    constructor(val, left, right, height) {
        this.val = val === undefined ? 0 : val;
        this.height = height === undefined ? 0 : height;
        this.left = left === undefined ? null : left;
        this.right = right === undefined ? null : right;
    }
}
```

The "node height" refers to the distance from that node to its farthest leaf node, i.e., the number of "edges" passed. It is important to note that the height of a leaf node is 0, and the height of a null node is -1. We will create two utility functions for getting and updating the height of a node:

```
// === File: avl_tree.js ===

/* Get node height */
height(node) {
    // Empty node height is -1, leaf node height is 0
    return node === null ? -1 : node.height;
}

/* Update node height */
#updateHeight(node) {
    // Node height equals the height of the tallest subtree + 1
    node.height =
        Math.max(this.height(node.left), this.height(node.right)) + 1;
}
```

2. Node Balance Factor

The balance factor of a node is defined as the height of the node's left subtree minus the height of its right subtree, and the balance factor of a null node is defined as 0. We also encapsulate the function to obtain the node's balance factor for convenient subsequent use:

```
// == File: avl_tree.js ==

/* Get balance factor */
balanceFactor(node) {
    // Empty node balance factor is 0
    if (node == null) return 0;
    // Node balance factor = left subtree height - right subtree height
    return this.height(node.left) - this.height(node.right);
}
```

Tip

Let the balance factor be f, then the balance factor of any node in an AVL tree satisfies $-1 \leq f \leq 1$.

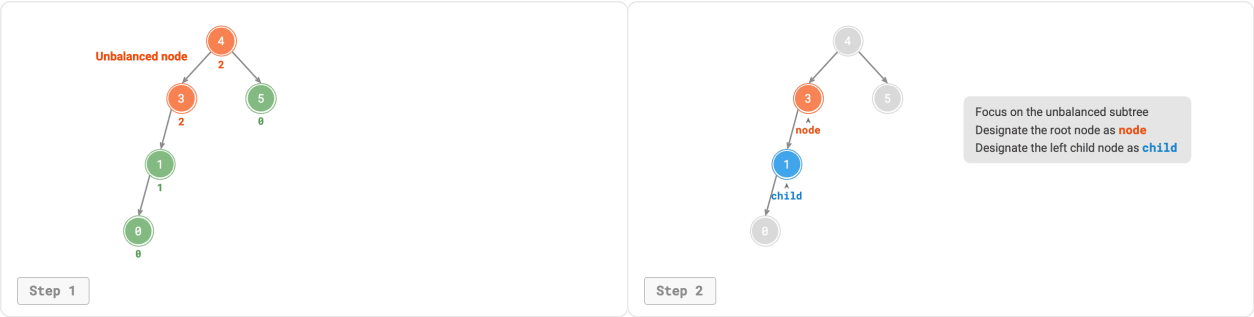
7.5.2 Rotations in Avl Trees

The characteristic of AVL trees lies in the "rotation" operation, which can restore balance to unbalanced nodes without affecting the inorder traversal sequence of the binary tree. In other words, **rotation operations can both maintain the property of a "binary search tree" and make the tree return to a "balanced binary tree"**.

We call nodes with a balance factor absolute value > 1 "unbalanced nodes". Depending on the imbalance situation, rotation operations are divided into four types: right rotation, left rotation, left rotation then right rotation, and right rotation then left rotation. Below we describe these rotation operations in detail.

1. Right Rotation

As shown in Figure 7-26, the value below the node is the balance factor. From bottom to top, the first unbalanced node in the binary tree is "node 3". We focus on the subtree with this unbalanced node as the root, denoting the node as `node` and its left child as `child`, and perform a "right rotation" operation. After the right rotation is completed, the subtree regains balance and still maintains the properties of a binary search tree.



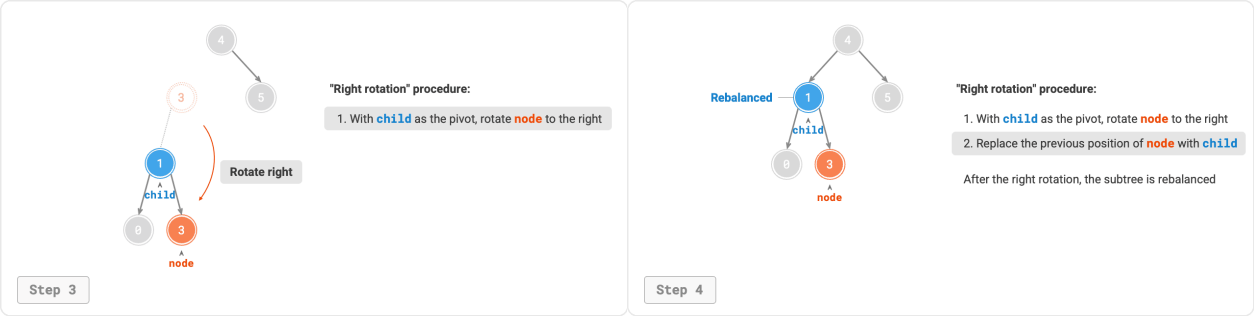


Figure 7-26 Steps of right rotation

As shown in Figure 7-27, when the `child` node has a right child (denoted as `grand_child`), a step needs to be added in the right rotation: set `grand_child` as the left child of `node`.

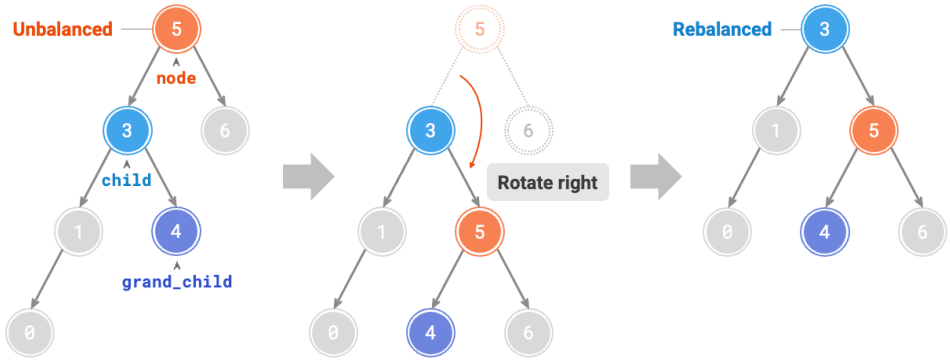


Figure 7-27 Right rotation with grand_child

"Right rotation" is a figurative term; in practice, it is achieved by modifying node pointers, as shown in the following code:

```
// ═══ File: avl_tree.js ═══

/* Right rotation operation */
#rightRotate(node) {
    const child = node.left;
    const grandChild = child.right;
    // Using child as pivot, rotate node to the right
    child.right = node;
    node.left = grandChild;
    // Update node height
    this.#updateHeight(node);
    this.#updateHeight(child);
    // Return root node of subtree after rotation
    return child;
}
```

2. Left Rotation

Correspondingly, if considering the "mirror" of the above unbalanced binary tree, the "left rotation" operation shown in Figure 7-28 needs to be performed.

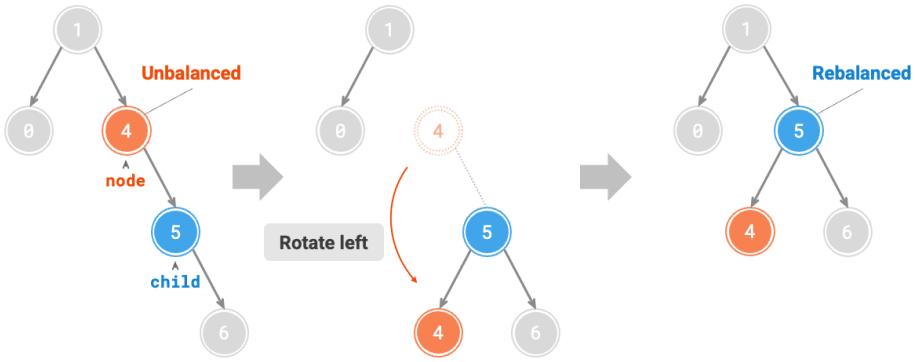


Figure 7-28 Left rotation operation

Similarly, as shown in Figure 7-29, when the `child` node has a left child (denoted as `grand_child`), a step needs to be added in the left rotation: set `grand_child` as the right child of `node`.

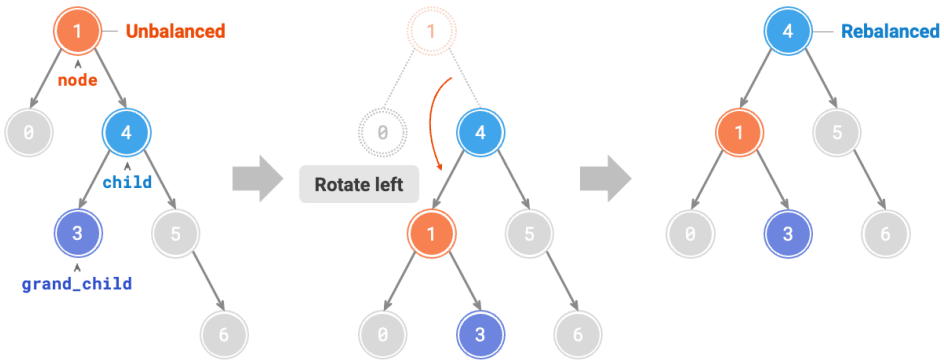


Figure 7-29 Left rotation with grand_child

It can be observed that **right rotation and left rotation operations are mirror symmetric in logic, and the two imbalance cases they solve are also symmetric**. Based on symmetry, we only need to replace all `left` in the right rotation implementation code with `right`, and all `right` with `left`, to obtain the left rotation implementation code:

```
// ═══ File: avl_tree.js ═══

/* Left rotation operation */
#leftRotate(node) {
```

```
    const child = node.right;
    const grandChild = child.left;
    // Using child as pivot, rotate node to the left
    child.left = node;
    node.right = grandChild;
    // Update node height
    this.#updateHeight(node);
    this.#updateHeight(child);
    // Return root node of subtree after rotation
    return child;
}
```

3. Left Rotation Then Right Rotation

For the unbalanced node 3 in Figure 7-30, using either left rotation or right rotation alone cannot restore the subtree to balance. In this case, a "left rotation" needs to be performed on `child` first, followed by a "right rotation" on `node`.

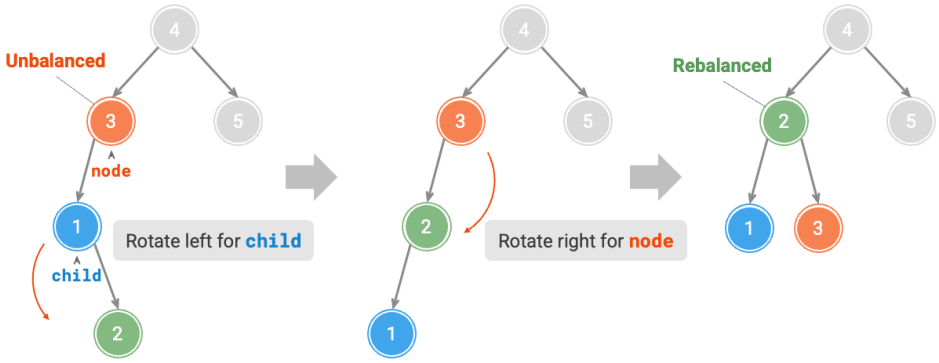


Figure 7-30 Left-right rotation

4. Right Rotation Then Left Rotation

As shown in Figure 7-31, for the mirror case of the above unbalanced binary tree, a "right rotation" needs to be performed on `child` first, then a "left rotation" on `node`.

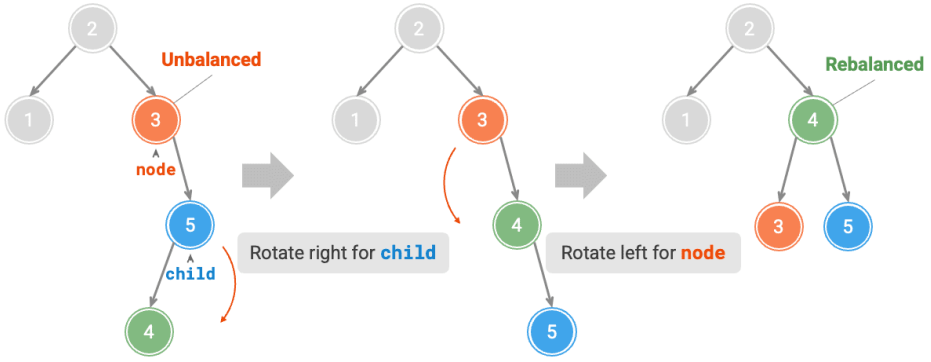


Figure 7-31 Right-left rotation

5. Choice of Rotation

The four imbalances shown in Figure 7-32 correspond one-to-one with the above cases, requiring right rotation, left rotation then right rotation, right rotation then left rotation, and left rotation operations respectively.

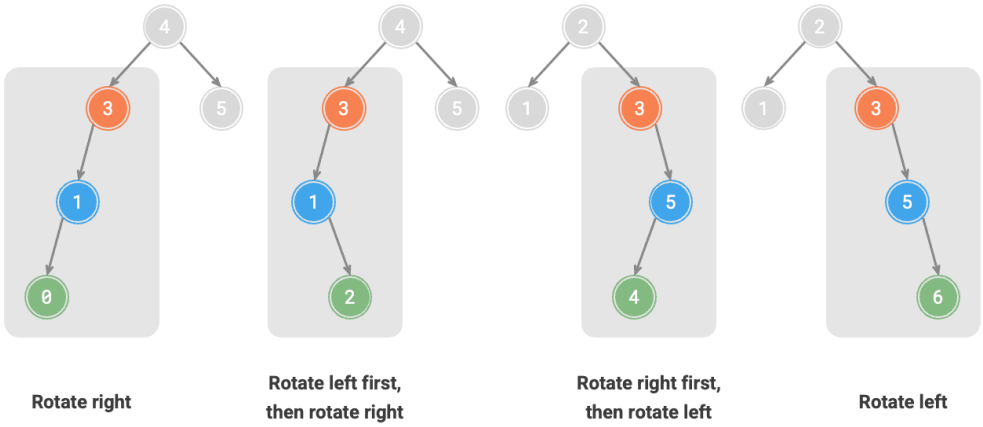


Figure 7-32 The four rotation cases of AVL tree

As shown in Table 7-3, we determine which case the unbalanced node belongs to by judging the signs of the balance factor of the unbalanced node and the balance factor of its taller-side child node.

Table 7-3 Conditions for Choosing Among the Four Rotation Cases

Balance factor of the unbalanced node	Balance factor of the child node	Rotation method to apply
> 1 (left-leaning tree)	≥ 0	Right rotation
> 1 (left-leaning tree)	< 0	Left rotation then right rotation
< -1 (right-leaning tree)	≤ 0	Left rotation
< -1 (right-leaning tree)	> 0	Right rotation then left rotation

For ease of use, we encapsulate the rotation operations into a function. **With this function, we can perform rotations for various imbalance situations, restoring balance to unbalanced nodes**. The code is as follows:

```
// == File: avl_tree.js ==

/* Perform rotation operation to restore balance to this subtree */
#rotate(node) {
    // Get balance factor of node
    const balanceFactor = this.balanceFactor(node);
    // Left-leaning tree
    if (balanceFactor > 1) {
        if (this.balanceFactor(node.left) >= 0) {
            // Right rotation
            return this.#rightRotate(node);
        } else {
            // First left rotation then right rotation
            node.left = this.#leftRotate(node.left);
            return this.#rightRotate(node);
        }
    }
    // Right-leaning tree
    if (balanceFactor < -1) {
        if (this.balanceFactor(node.right) <= 0) {
            // Left rotation
            return this.#leftRotate(node);
        } else {
            // First right rotation then left rotation
            node.right = this.#rightRotate(node.right);
            return this.#leftRotate(node);
        }
    }
    // Balanced tree, no rotation needed, return directly
    return node;
}
```

7.5.3 Common Operations in Avl Trees

1. Node Insertion

The node insertion operation in AVL trees is similar in principle to that in binary search trees. The only difference is that after inserting a node in an AVL tree, a series of unbalanced nodes may appear on the path from that node to the root. Therefore, **we need to start from this node and perform rotation operations from bottom to top, restoring balance to all unbalanced nodes**. The code is as follows:

```
// === File: avl_tree.js ===

/* Insert node */
insert(val) {
    this.root = this.#insertHelper(this.root, val);
}

/* Recursively insert node (helper method) */
#insertHelper(node, val) {
    if (node === null) return new TreeNode(val);
    /* 1. Find insertion position and insert node */
    if (val < node.val) node.left = this.#insertHelper(node.left, val);
    else if (val > node.val)
        node.right = this.#insertHelper(node.right, val);
    else return node; // Duplicate node not inserted, return directly
    this.#updateHeight(node); // Update node height
    /* 2. Perform rotation operation to restore balance to this subtree */
    node = this.#rotate(node);
    // Return root node of subtree
    return node;
}
```

2. Node Removal

Similarly, on the basis of the binary search tree's node removal method, rotation operations need to be performed from bottom to top to restore balance to all unbalanced nodes. The code is as follows:

```
// === File: avl_tree.js ===

/* Remove node */
remove(val) {
    this.root = this.#removeHelper(this.root, val);
}

/* Recursively delete node (helper method) */
#removeHelper(node, val) {
    if (node === null) return null;
    /* 1. Find node and delete */
    if (val < node.val) node.left = this.#removeHelper(node.left, val);
    else if (val > node.val)
        node.right = this.#removeHelper(node.right, val);
    else {
        if (node.left === null || node.right === null) {
            const child = node.left !== null ? node.left : node.right;
            // Number of child nodes = 0, delete node directly and return
            if (child === null) return null;
            // Number of child nodes = 1, delete node directly
            else node = child;
        } else {
            // Number of child nodes = 2, delete the next node in inorder traversal and replace
            ↪ current node with it
            let temp = node.right;
            while (temp.left !== null) {
                temp = temp.left;
            }
            node.right = this.#removeHelper(node.right, temp.val);
            node.val = temp.val;
```

```
        }
    }
    this.#updateHeight(node); // Update node height
    /* 2. Perform rotation operation to restore balance to this subtree */
    node = this.#rotate(node);
    // Return root node of subtree
    return node;
}
```

3. Node Search

The node search operation in AVL trees is consistent with that in binary search trees, and will not be elaborated here.

7.5.4 Typical Applications of Avl Trees

- Organizing and storing large-scale data, suitable for scenarios with high-frequency searches and low-frequency insertions and deletions.
- Used to build index systems in databases.
- Red-black trees are also a common type of balanced binary search tree. Compared to AVL trees, red-black trees have more relaxed balance conditions, require fewer rotation operations for node insertion and deletion, and have higher average efficiency for node addition and deletion operations.

7.6 Summary

1. Key Review

- A binary tree is a non-linear data structure that embodies the divide-and-conquer logic of "one divides into two". Each binary tree node contains a value and two pointers, which respectively point to its left and right child nodes.
- For a certain node in a binary tree, the tree formed by its left (right) child node and all nodes below is called the left (right) subtree of that node.
- Related terminology of binary trees includes root node, leaf node, level, degree, edge, height, and depth.
- The initialization, node insertion, and node removal operations of binary trees are similar to those of linked lists.
- Common types of binary trees include perfect binary trees, complete binary trees, full binary trees, and balanced binary trees. The perfect binary tree is the ideal state, while the linked list is the worst state after degradation.
- A binary tree can be represented using an array by arranging node values and empty slots in level-order traversal sequence, and implementing pointers based on the index mapping relationship between parent and child nodes.

- Level-order traversal of a binary tree is a breadth-first search method, embodying a layer-by-layer traversal approach of "expanding outward circle by circle", typically implemented using a queue.
- Preorder, inorder, and postorder traversals all belong to depth-first search, embodying a traversal approach of "first go to the end, then backtrack and continue", typically implemented using recursion.
- A binary search tree is an efficient data structure for element searching, with search, insertion, and removal operations all having time complexity of $O(\log n)$. When a binary search tree degenerates into a linked list, all time complexities degrade to $O(n)$.
- An AVL tree, also known as a balanced binary search tree, ensures the tree remains balanced after continuous node insertions and removals through rotation operations.
- Rotation operations in AVL trees include right rotation, left rotation, left rotation then right rotation, and right rotation then left rotation. After inserting or removing nodes, AVL trees perform rotation operations from bottom to top to restore the tree to balance.

2. Q & A

Q: For a binary tree with only one node, are both the height of the tree and the depth of the root node 0?

Yes, because height and depth are typically defined as "the number of edges passed."

Q: The insertion and removal in a binary tree are generally accomplished by a set of operations. What does "a set of operations" refer to here? Does it imply releasing the resources of the child nodes?

Taking the binary search tree as an example, the operation of removing a node needs to be handled in three different scenarios, each requiring multiple steps of node operations.

Q: Why does DFS traversal of binary trees have three orders: preorder, inorder, and postorder, and what are their uses?

Similar to forward and reverse traversal of arrays, preorder, inorder, and postorder traversals are three methods of binary tree traversal that allow us to obtain a traversal result in a specific order. For example, in a binary search tree, since nodes satisfy the relationship `left child node value < root node value < right child node value`, we only need to traverse the tree with the priority of "left $\rightarrow$ root $\rightarrow$ right" to obtain an ordered node sequence.

Q: In a right rotation operation handling the relationship between unbalanced nodes `node`, `child`, and `grand_child`, doesn't the connection between `node` and its parent node get lost after the right rotation?

We need to view this problem from a recursive perspective. The right rotation operation `right_rotate(root)` passes in the root node of the subtree and eventually returns the root node of the subtree after rotation with `return child`. The connection between the subtree's root node and its parent node is completed after the function returns, which is not within the maintenance scope of the right rotation operation.

Q: In C++, functions are divided into `private` and `public` sections. What considerations are there for

this? Why are the `height()` function and the `updateHeight()` function placed in `public` and `private`, respectively?

It mainly depends on the method's usage scope. If a method is only used within the class, then it is designed as `private`. For example, calling `updateHeight()` alone by the user makes no sense, as it is only a step in insertion or removal operations. However, `height()` is used to access node height, similar to `vector.size()`, so it is set to `public` for ease of use.

Q: How do you build a binary search tree from a set of input data? Is the choice of root node very important?

Yes, the method for building a tree is provided in the `build_tree()` method in the binary search tree code. As for the choice of root node, we typically sort the input data, then select the middle element as the root node, and recursively build the left and right subtrees. This approach maximizes the tree's balance.

Q: In Java, do you always have to use the `equals()` method for string comparison?

In Java, for primitive data types, `==` is used to compare whether the values of two variables are equal. For reference types, the working principles of the two symbols are different.

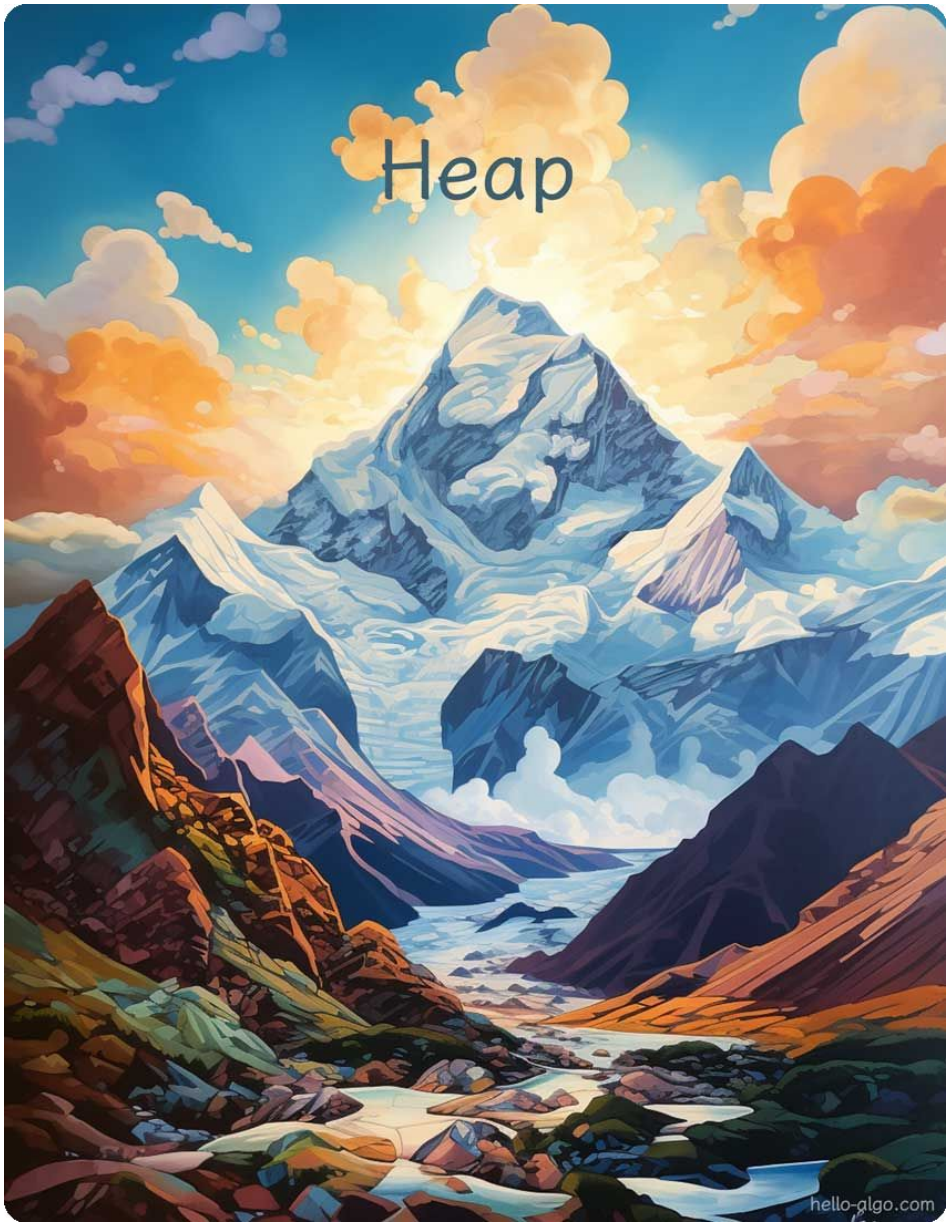
- `==`: Used to compare whether two variables point to the same object, i.e., whether their positions in memory are the same.
- `equals()`: Used to compare whether the values of two objects are equal.

Therefore, if we want to compare values, we should use `equals()`. However, strings initialized via `String a = "hi"; String b = "hi";` are stored in the string constant pool and point to the same object, so `a == b` can also be used to compare the contents of the two strings.

Q: Before reaching the bottom level, is the number of nodes in the queue 2^h in breadth-first traversal?

Yes, for example, a full binary tree with height $h = 2$ has a total of $n = 7$ nodes, then the bottom level has $4 = 2^h = (n + 1)/2$ nodes.

Chapter 8. Heap



Abstract

Heaps are like mountain peaks, layered and undulating, each with its unique form.

The peaks rise and fall at varying heights, yet the tallest peak always catches the eye first.

8.1 Heap

A heap is a complete binary tree that satisfies specific conditions and can be mainly categorized into two types, as shown in Figure 8-1.

- min heap: The value of any node $\leq$ the values of its child nodes.
- max heap: The value of any node $\geq$ the values of its child nodes.

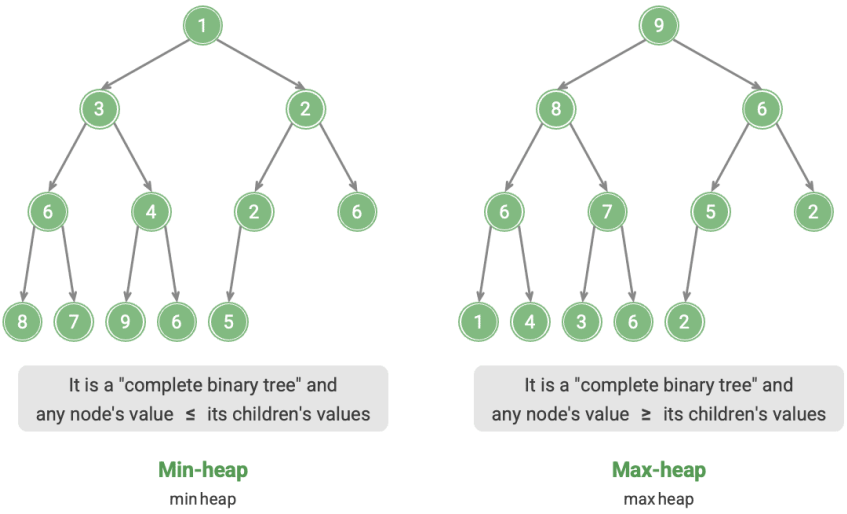


Figure 8-1 Min heap and max heap

As a special case of a complete binary tree, heaps have the following characteristics.

- The bottom layer nodes are filled from left to right, and nodes in other layers are fully filled.
- We call the root node of the binary tree the "heap top" and the bottom-rightmost node the "heap bottom."
- For max heaps (min heaps), the value of the heap top element (root node) is the largest (smallest).

8.1.1 Common Heap Operations

It should be noted that many programming languages provide a priority queue, which is an abstract data structure defined as a queue with priority sorting.

In fact, **heaps are typically used to implement priority queues, with max heaps corresponding to priority queues where elements are dequeued in descending order**. From a usage perspective, we can regard "priority queue" and "heap" as equivalent data structures. Therefore, this book does not make a special distinction between the two and uniformly refers to them as "heap."

Common heap operations are shown in Table 8-1, and method names need to be determined based on the programming language.

Table 8-1 Efficiency of Heap Operations

Method name	Description	Time complexity
`push()`	Insert an element into the heap	$O(\log n)$
`pop()`	Remove the heap top element	$O(\log n)$
`peek()`	Access the heap top element (max/min value for max/min heap)	$O(1)$
`size()`	Get the number of elements in the heap	$O(1)$
`isEmpty()`	Check if the heap is empty	$O(1)$

In practical applications, we can directly use the heap class (or priority queue class) provided by programming languages.

Similar to "ascending order" and "descending order" in sorting algorithms, we can implement conversion between "min heap" and "max heap" by setting a `flag` or modifying the `Comparator`. The code is as follows:

```
// ═══ File: heap.js ═══

// JavaScript does not provide a built-in Heap class
```

8.1.2 Implementation of the Heap

The following implementation is of a max heap. To convert it to a min heap, simply invert all size logic comparisons (for example, replace $\geq$ with $\leq$). Interested readers are encouraged to implement this on their own.

1. Heap Storage and Representation

As mentioned in the "Binary Tree" chapter, complete binary trees are well-suited for array representation. Since heaps are a type of complete binary tree, **we will use arrays to store heaps**.

When representing a binary tree with an array, elements represent node values, and indexes represent node positions in the binary tree. **Node pointers are implemented through index mapping formulas**.

As shown in Figure 8-2, given an index i, the index of its left child is $2i + 1$, the index of its right child is $2i + 2$, and the index of its parent is $(i - 1)/2$ (floor division). When an index is out of bounds, it indicates a null node or that the node does not exist.

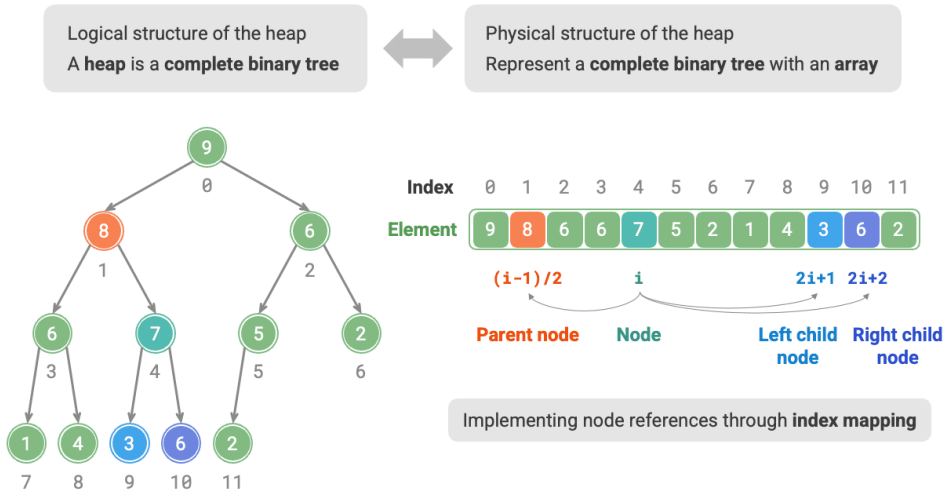


Figure 8-2 Representation and storage of heaps

We can encapsulate the index mapping formula into functions for convenient subsequent use:

```
// ═══ File: my_heap.js ═══

/* Get index of left child node */
#left(i) {
    return 2 * i + 1;
}

/* Get index of right child node */
#right(i) {
    return 2 * i + 2;
}

/* Get index of parent node */
#parent(i) {
    return Math.floor((i - 1) / 2); // Floor division
}
```

2. Accessing the Heap Top Element

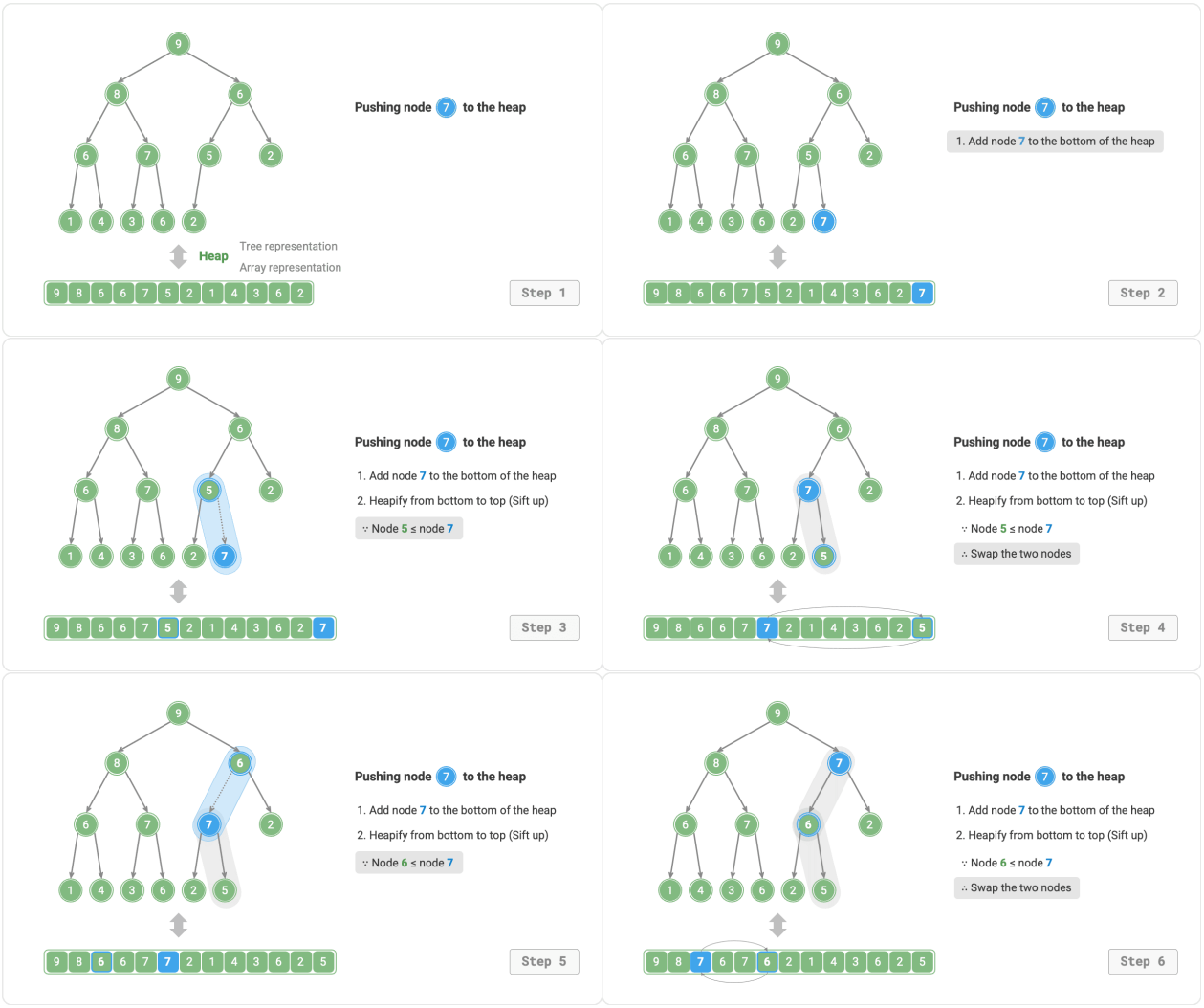
The heap top element is the root node of the binary tree, which is also the first element of the list:

```
// ═══ File: my_heap.js ═══

/* Access top element */
peek() {
    return this.#maxHeap[0];
}
```

3. Inserting an Element Into the Heap

Given an element `val`, we first add it to the bottom of the heap. After addition, since `val` may be larger than other elements in the heap, the heap's property may be violated. **Therefore, it's necessary to repair the path from the inserted node to the root node**. This operation is called heapify.

Starting from the inserted node, **perform heapify from bottom to top**. As shown in Figure 8-3, we compare the inserted node with its parent node, and if the inserted node is larger, swap them. Then continue this operation, repairing nodes in the heap from bottom to top until we pass the root node or encounter a node that does not need swapping.



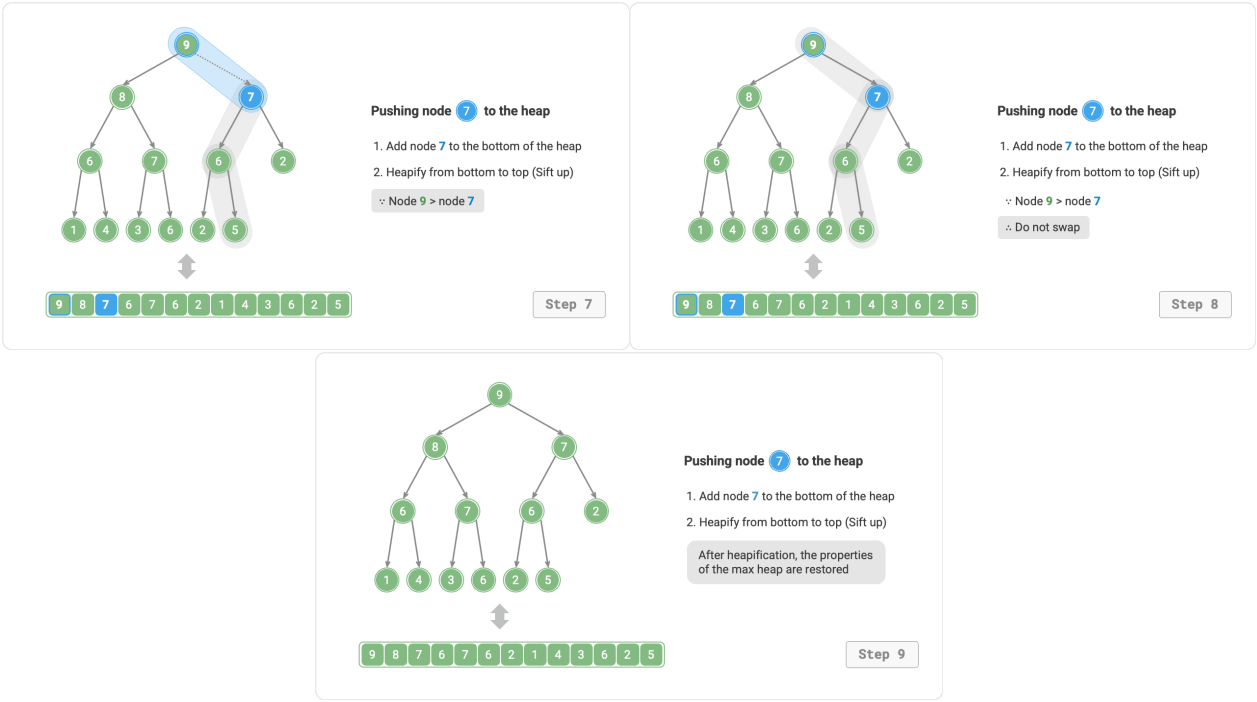


Figure 8-3 Steps of inserting an element into the heap

Given a total of n nodes, the tree height is $O(\log n)$. Thus, the number of loop iterations in the heapify operation is at most $O(\log n)$, **making the time complexity of the element insertion operation** $O(\log n)$. The code is as follows:

```
// === File: my_heap.js ===

/* Element enters heap */
push(val) {
    // Add node
    this.#maxHeap.push(val);
    // Heapify from bottom to top
    this.#siftUp(this.size() - 1);
}

/* Starting from node i, heapify from bottom to top */
#siftUp(i) {
    while (true) {
        // Get parent node of node i
        const p = this.#parent(i);
        // When "crossing root node" or "node needs no repair", end heapify
        if (p < 0 || this.#maxHeap[i] <= this.#maxHeap[p]) break;
        // Swap two nodes
        this.#swap(i, p);
        // Loop upward heapify
        i = p;
    }
}
```

4. Removing the Heap Top Element

The heap top element is the root node of the binary tree, which is the first element of the list. If we directly remove the first element from the list, all node indexes in the binary tree would change, making subsequent repair with heapify difficult. To minimize changes in element indexes, we use the following steps.

1. Swap the heap top element with the heap bottom element (swap the root node with the rightmost leaf node).
2. After swapping, remove the heap bottom from the list (note that since we've swapped, we're actually removing the original heap top element).
3. Starting from the root node, **perform heapify from top to bottom**.

As shown in Figure 8-4, **the direction of "top-to-bottom heapify" is opposite to "bottom-to-top heapify"**. We compare the root node's value with its two children and swap it with the largest child. Then loop this operation until we pass a leaf node or encounter a node that doesn't need swapping.

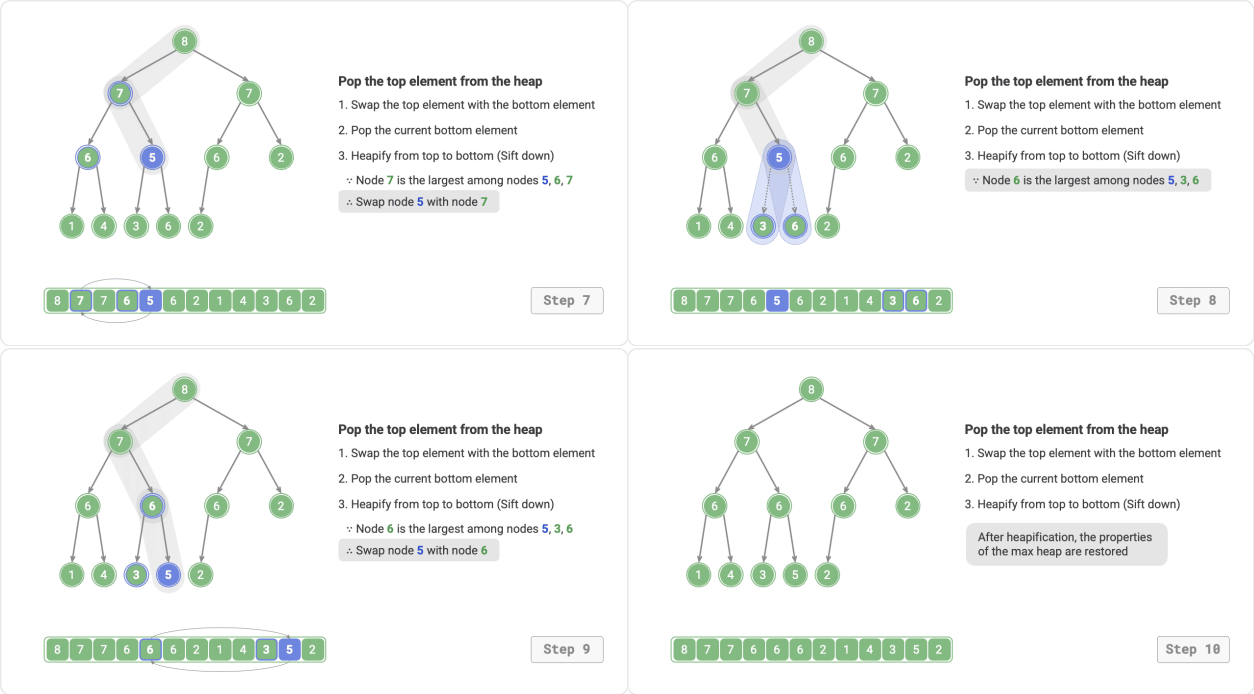


Figure 8-4 Steps of removing the heap top element

Similar to the element insertion operation, the time complexity of the heap top element removal operation is also $O(\log n)$. The code is as follows:

```
// ═══ File: my_heap.js ═══

/* Element exits heap */
pop() {
    // Handle empty case
    if (this.isEmpty()) throw new Error('Heap is empty');
    // Delete node
    this.#swap(0, this.size() - 1);
    // Remove node
    const val = this.#maxHeap.pop();
    // Return top element
    this.#siftDown(0);
    // Return heap top element
    return val;
}

/* Starting from node i, heapify from top to bottom */
#siftDown(i) {
    while (true) {
        // If node i is largest or indices l, r are out of bounds, no need to continue heapify,
        ↪ break
        const l = this.#left(i),
            r = this.#right(i);
        let ma = i;
        if (l < this.size() && this.#maxHeap[l] > this.#maxHeap[ma]) ma = l;
        if (r < this.size() && this.#maxHeap[r] > this.#maxHeap[ma]) ma = r;
        // Swap two nodes
        if (ma == i) break;
```

```
        // Swap two nodes
        this.#swap(i, ma);
        // Loop downwards heapification
        i = ma;
    }
}
```

8.1.3 Common Applications of Heaps

- **Priority queue**: Heaps are typically the preferred data structure for implementing priority queues, with both enqueue and dequeue operations having a time complexity of $O(\log n)$, and the heap construction operation having $O(n)$, all of which are highly efficient.
- **Heap sort**: Given a set of data, we can build a heap with them and then continuously perform element removal operations to obtain sorted data. However, we usually use a more elegant approach to implement heap sort, as detailed in the "Heap Sort" chapter.
- **Getting the largest k elements**: This is a classic algorithm problem and also a typical application, such as selecting the top 10 trending news for Weibo hot search, selecting the top 10 best-selling products, etc.

8.2 Heap Construction Operation

In some cases, we want to build a heap using all elements of a list, and this process is called "heap construction operation."

8.2.1 Implementing with Element Insertion

We first create an empty heap, then iterate through the list, performing the "element insertion operation" on each element in sequence. This means adding the element to the bottom of the heap and then performing "bottom-to-top" heapify on that element.

Each time an element is inserted into the heap, the heap's length increases by one. Since nodes are added to the binary tree sequentially from top to bottom, the heap is constructed "from top to bottom."

Given n elements, each element's insertion operation takes $O(\log n)$ time, so the time complexity of this heap construction method is $O(n \log n)$.

8.2.2 Implementing Through Heapify Traversal

In fact, we can implement a more efficient heap construction method in two steps.

1. Add all elements of the list as-is to the heap, at which point the heap property is not yet satisfied.
2. Traverse the heap in reverse order (reverse of level-order traversal), performing "top-to-bottom heapify" on each non-leaf node in sequence.

After heapifying a node, the subtree rooted at that node becomes a valid sub-heap. Since we traverse in reverse order, the heap is constructed "from bottom to top."

The reason for choosing reverse order traversal is that it ensures the subtree below the current node is already a valid sub-heap, making the heapification of the current node effective.

It's worth noting that **since leaf nodes have no children, they are naturally valid sub-heaps and do not require heapification**. As shown in the code below, the last non-leaf node is the parent of the last node; we start from it and traverse in reverse order to perform heapification:

```
// ═══ File: my_heap.js ═══

/* Constructor, build empty heap or build heap from input list */
constructor(nums) {
    // Add list elements to heap as is
    this.#maxHeap = nums === undefined ? [] : [...nums];
    // Heapify all nodes except leaf nodes
    for (let i = this.#parent(this.size() - 1); i >= 0; i--) {
        this.#siftDown(i);
    }
}
```

8.2.3 Complexity Analysis

Next, let's attempt to derive the time complexity of this second heap construction method.

- Assuming the complete binary tree has n nodes, then the number of leaf nodes is $(n+1)/2$, where $/$ is floor division. Therefore, the number of nodes that need heapification is $(n-1)/2$.
- In the top-to-bottom heapify process, each node is heapified at most to the leaf nodes, so the maximum number of iterations is the binary tree height $\log n$.

Multiplying these two together, we get a time complexity of $O(n \log n)$ for the heap construction process. **However, this estimate is not accurate because it doesn't account for the property that binary trees have far more nodes at lower levels than at upper levels**.

Let's perform a more accurate calculation. To reduce calculation difficulty, assume a "perfect binary tree" with n nodes and height h; this assumption does not affect the correctness of the result.

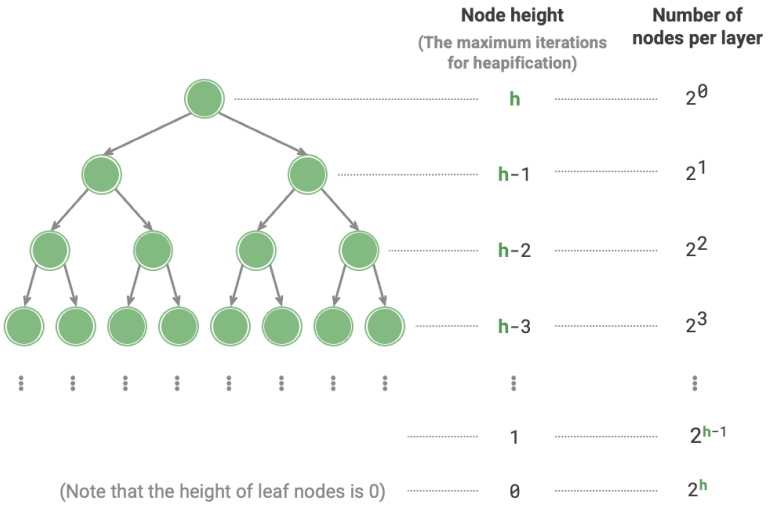


Figure 8-5 Node count at each level of a perfect binary tree

As shown in Figure 8-5, the maximum number of iterations for a node's "top-to-bottom heapify" equals the distance from that node to the leaf nodes, which is precisely the "node height." Therefore, we can sum the "number of nodes $\times$ node height" at each level to **obtain the total number of heapify iterations for all nodes**.

$$T(h) = 2^0 h + 2^1(h-1) + 2^2(h-2) + \cdots + 2^{(h-1)} \times 1$$

To simplify the above expression, we need to use sequence knowledge from high school. First, multiply $T(h)$ by 2 to get:

$$\begin{aligned} T(h) &= 2^0 h + 2^1(h-1) + 2^2(h-2) + \cdots + 2^{h-1} \times 1 \\ 2T(h) &= 2^1 h + 2^2(h-1) + 2^3(h-2) + \cdots + 2^h \times 1 \end{aligned}$$

Using the method of differences, subtract the first equation $T(h)$ from the second equation $2T(h)$ to get:

$$2T(h) - T(h) = T(h) = -2^0 h + 2^1 + 2^2 + \cdots + 2^{h-1} + 2^h$$

Observing the above expression, we find that $T(h)$ is a geometric series, which can be calculated directly using the sum formula, yielding a time complexity of:

$$\begin{aligned} T(h) &= 2\frac{1-2^h}{1-2} - h \\ &= 2^{h+1} - h - 2 \\ &= O(2^h) \end{aligned}$$

Furthermore, a perfect binary tree with height h has $n = 2^{h+1} - 1$ nodes, so the complexity is $O(2^h) = O(n)$. This derivation shows that **the time complexity of building a heap from an input list is $O(n)$, which is highly efficient**.

8.3 Top-K Problem

Question
Given an unordered array `nums` of length n, return the largest k elements in the array.

For this problem, we'll first introduce two solutions with relatively straightforward approaches, then introduce a more efficient heap-based solution.

8.3.1 Method 1: Iterative Selection

We can perform k rounds of traversal as shown in Figure 8-6, extracting the 1^{st}, 2^{nd}, ..., k^{th} largest elements in each round, with a time complexity of $O(nk)$.

This method is only suitable when $k \ll n$, because when k is close to n, the time complexity approaches $O(n^2)$, which is very time-consuming.

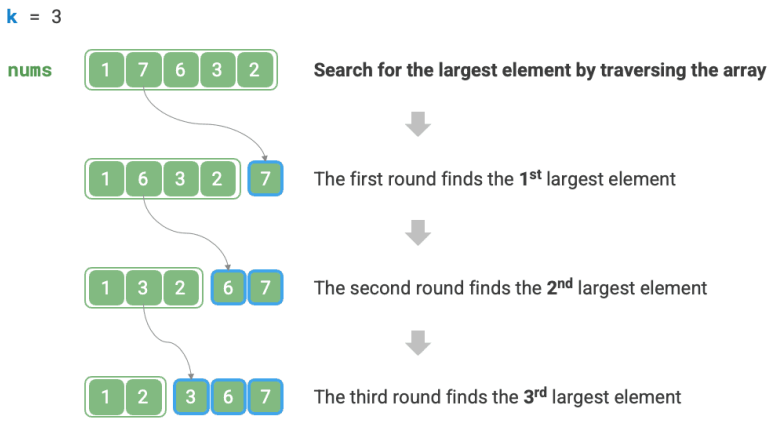


Figure 8-6 Traversing to find the largest k elements

Tip
When $k = n$, we can obtain a complete sorted sequence, which is equivalent to the "selection sort" algorithm.

8.3.2 Method 2: Sorting

As shown in Figure 8-7, we can first sort the array `nums`, then return the rightmost k elements, with a time complexity of $O(n \log n)$.

Clearly, this method "overachieves" the task, as we only need to find the largest k elements, without needing to sort the other elements.

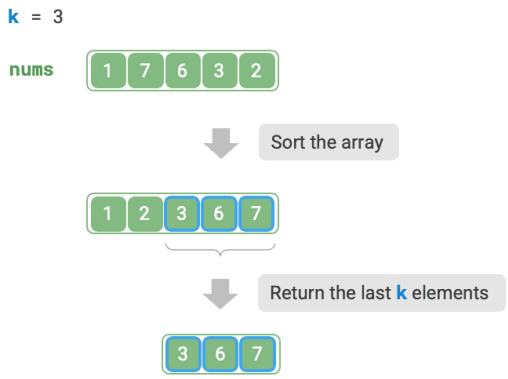


Figure 8-7 Sorting to find the largest k elements

8.3.3 Method 3: Heap

We can solve the Top-k problem more efficiently using heaps, with the process shown in Figure 8-8.

1. Initialize a min heap, where the heap top element is the smallest.
2. First, insert the first k elements of the array into the heap in sequence.
3. Starting from the $(k+1)^{th}$ element, if the current element is greater than the heap top element, remove the heap top element and insert the current element into the heap.
4. After traversal is complete, the heap contains the largest k elements.

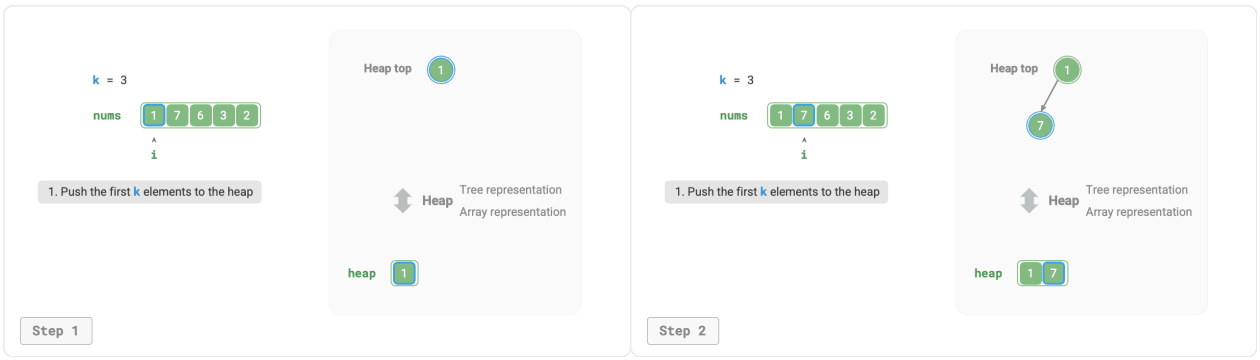


Figure 8-8 Finding the largest k elements using a heap

Example code is as follows:

```
// ═══ File: top_k.js ═══

/* Element enters heap */
function pushMinHeap(maxHeap, val) {
```

```
    // Negate element
    maxHeap.push(-val);
}

/* Element exits heap */
function popMinHeap(maxHeap) {
    // Negate element
    return -maxHeap.pop();
}

/* Access top element */
function peekMinHeap(maxHeap) {
    // Negate element
    return -maxHeap.peek();
}

/* Extract elements from heap */
function getMinHeap(maxHeap) {
    // Negate element
    return maxHeap.getMaxHeap().map((num) => -num);
}

/* Find the largest k elements in array based on heap */
function topKHeap(nums, k) {
    // Python's heapq module implements min heap by default
    // Note: We negate all heap elements to simulate min heap using max heap
    const maxHeap = new MaxHeap([]);
    // Enter the first k elements of array into heap
    for (let i = 0; i < k; i++) {
        pushMinHeap(maxHeap, nums[i]);
    }
    // Starting from the (k+1)th element, maintain heap length as k
    for (let i = k; i < nums.length; i++) {
        // If current element is greater than top element, top element exits heap, current
        ↪  element enters heap
        if (nums[i] > peekMinHeap(maxHeap)) {
            popMinHeap(maxHeap);
            pushMinHeap(maxHeap, nums[i]);
        }
    }
    // Return elements in heap
    return getMinHeap(maxHeap);
}
```

A total of n rounds of heap insertions and removals are performed, with the heap's maximum length being k, so the time complexity is $O(n \log k)$. This method is very efficient; when k is small, the time complexity approaches $O(n)$; when k is large, the time complexity does not exceed $O(n \log n)$.

Additionally, this method is suitable for dynamic data stream scenarios. By continuously adding data, we can maintain the elements in the heap, thus achieving dynamic updates of the largest k elements.

8.4 Summary

1. Key Review

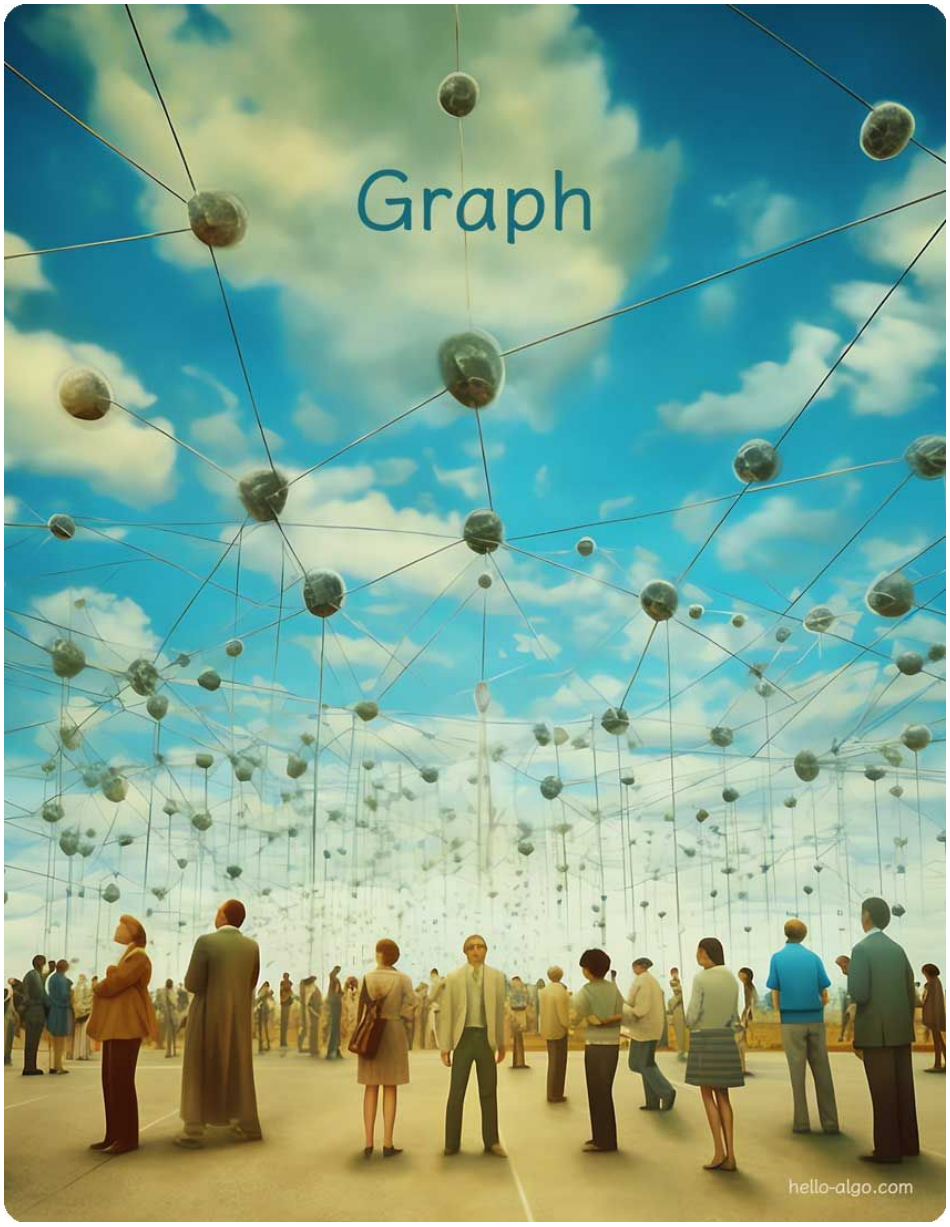
- A heap is a complete binary tree that can be categorized as a max heap or min heap based on its property. The heap top element of a max heap (min heap) is the largest (smallest).
- A priority queue is defined as a queue with priority sorting, typically implemented using heaps.
- Common heap operations and their corresponding time complexities include: element insertion $O(\log n)$, heap top element removal $O(\log n)$, and accessing the heap top element $O(1)$.
- Complete binary trees are well-suited for array representation, so we typically use arrays to store heaps.
- Heapify operations are used to maintain the heap property and are employed in both element insertion and removal operations.
- The time complexity of building a heap with n input elements can be optimized to $O(n)$, which is highly efficient.
- Top-k is a classic algorithm problem that can be efficiently solved using the heap data structure, with a time complexity of $O(n \log k)$.

2. Q & A

Q: Are the "heap" in data structures and the "heap" in memory management the same concept?

The two are not the same concept; they just happen to share the name "heap." The heap in computer system memory is part of dynamic memory allocation, where programs can use it to store data during runtime. Programs can request a certain amount of heap memory to store complex structures such as objects and arrays. When this data is no longer needed, the program needs to release this memory to prevent memory leaks. Compared to stack memory, heap memory management and usage require more caution, as improper use can lead to issues such as memory leaks and dangling pointers.

Chapter 9. Graph



Abstract

In the journey of life, we are like nodes, connected by countless invisible edges.
Each encounter and parting leaves a unique mark on this vast network graph.

9.1 Graph

A graph is a nonlinear data structure consisting of vertices and edges. We can abstractly represent a graph G as a set of vertices V and a set of edges E. The following example shows a graph containing 5 vertices and 7 edges.

$$
\begin{aligned}
V &= \{1, 2, 3, 4, 5\} \\
E &= \{(1, 2), (1, 3), (1, 5), (2, 3), (2, 4), (2, 5), (4, 5)\} \\
G &= \{V, E\}
\end{aligned}
$$

If we view vertices as nodes and edges as references (pointers) connecting the nodes, we can see graphs as a data structure extended from linked lists. As shown in Figure 9-1, **compared to linear relationships (linked lists) and divide-and-conquer relationships (trees), network relationships (graphs) have a higher degree of freedom and are therefore more complex**.

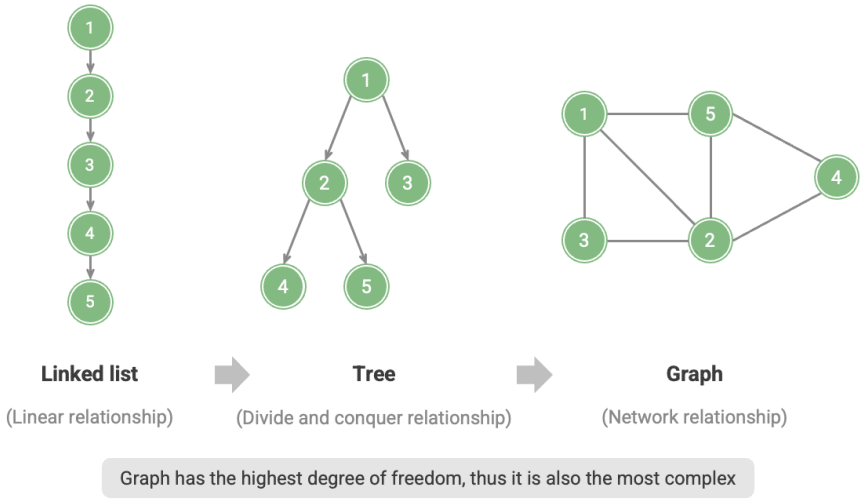


Figure 9-1 Relationships among linked lists, trees, and graphs

9.1.1 Common Types and Terminology of Graphs

Graphs can be divided into undirected graphs and directed graphs based on whether edges have direction, as shown in Figure 9-2.

- In undirected graphs, edges represent a "bidirectional" connection between two vertices, such as the "friend relationship" on WeChat or QQ.
- In directed graphs, edges have directionality, meaning edges $A \to B$ and $A \leftarrow B$ are independent of each other, such as the "follow" and "be followed" relationships on Weibo or TikTok.

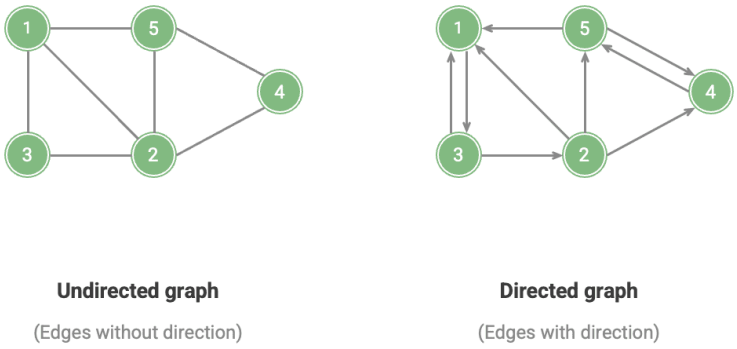


Figure 9-2 Directed and undirected graphs

Graphs can be divided into connected graphs and disconnected graphs based on whether all vertices are connected, as shown in Figure 9-3.

- For connected graphs, starting from any vertex, all other vertices can be reached.
- For disconnected graphs, starting from a certain vertex, at least one vertex cannot be reached.

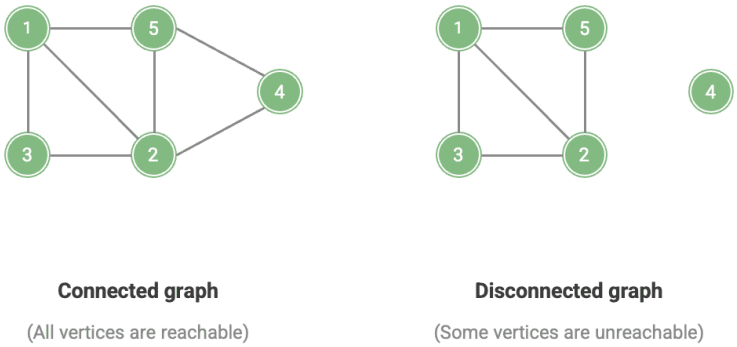


Figure 9-3 Connected and disconnected graphs

We can also add a "weight" variable to edges, resulting in weighted graphs as shown in Figure 9-4. For example, in mobile games like "Honor of Kings", the system calculates the "intimacy" between players based on their shared game time, and such intimacy networks can be represented using weighted graphs.

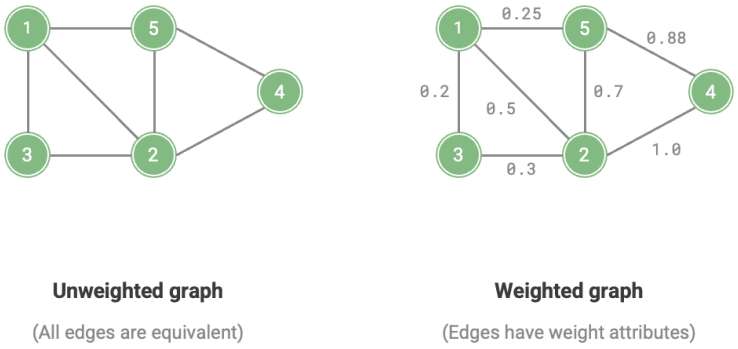


Figure 9-4 Weighted and unweighted graphs

Graph data structures include the following commonly used terms.

- Adjacency: When two vertices are connected by an edge, these two vertices are said to be "adjacent". In Figure 9-4, the adjacent vertices of vertex 1 are vertices 2, 3, and 5.
- Path: The sequence of edges from vertex A to vertex B is called a "path" from A to B. In Figure 9-4, the edge sequence 1-5-2-4 is a path from vertex 1 to vertex 4.
- Degree: The number of edges a vertex has. For directed graphs, in-degree indicates how many edges point to the vertex, and out-degree indicates how many edges point out from the vertex.

9.1.2 Representation of Graphs

Common representations of graphs include "adjacency matrices" and "adjacency lists". The following uses undirected graphs as examples.

1. Adjacency Matrix

Given a graph with n vertices, an adjacency matrix uses an $n \times n$ matrix to represent the graph, where each row (column) represents a vertex, and matrix elements represent edges, using 1 or 0 to indicate whether an edge exists between two vertices.

As shown in Figure 9-5, let the adjacency matrix be M and the vertex list be V. Then matrix element $M[i, j] = 1$ indicates that an edge exists between vertex $V[i]$ and vertex $V[j]$, whereas $M[i, j] = 0$ indicates no edge between the two vertices.

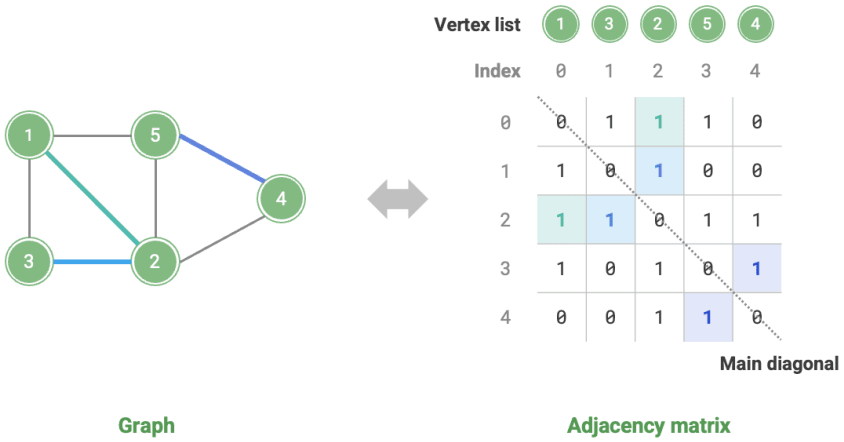


Figure 9-5 Adjacency matrix representation of a graph

Adjacency matrices have the following properties.

- In simple graphs, vertices cannot connect to themselves, so the elements on the main diagonal of the adjacency matrix are meaningless.
- For undirected graphs, edges in both directions are equivalent, so the adjacency matrix is symmetric about the main diagonal.
- Replacing the elements of the adjacency matrix from 1 and 0 to weights allows representation of weighted graphs.

When using adjacency matrices to represent graphs, we can directly access matrix elements to obtain edges, resulting in highly efficient addition, deletion, lookup, and modification operations, all with a time complexity of $O(1)$. However, the space complexity of the matrix is $O(n^2)$, which consumes significant memory.

2. Adjacency List

An adjacency list uses n linked lists to represent a graph, with linked list nodes representing vertices. The i-th linked list corresponds to vertex i and stores all adjacent vertices of that vertex (vertices connected to that vertex). Figure 9-6 shows an example of a graph stored using an adjacency list.

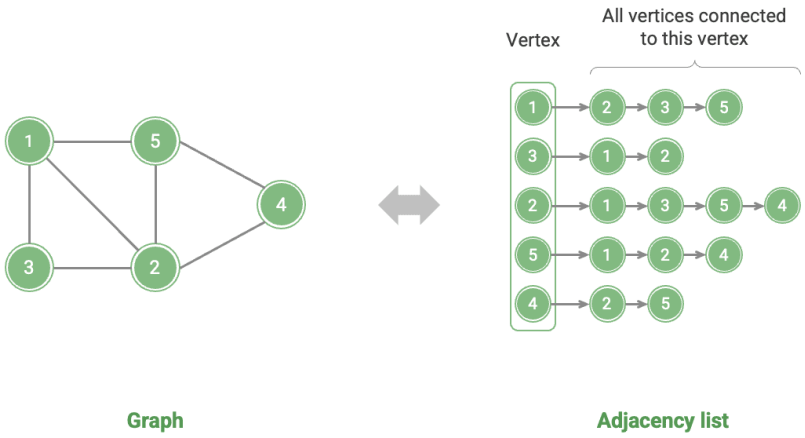


Figure 9-6 Adjacency list representation of a graph

Adjacency lists only store edges that actually exist, and the total number of edges is typically much less than n^2, making them more space-efficient. However, finding edges in an adjacency list requires traversing the linked list, so its time efficiency is inferior to that of adjacency matrices.

Observing Figure 9-6, **the structure of adjacency lists is very similar to "chaining" in hash tables, so we can adopt similar methods to optimize efficiency**. For example, when linked lists are long, they can be converted to AVL trees or red-black trees, thereby optimizing time efficiency from $O(n)$ to $O(\log n)$; linked lists can also be converted to hash tables, thereby reducing time complexity to $O(1)$.

9.1.3 Common Applications of Graphs

As shown in Table 9-1, many real-world systems can be modeled using graphs, and corresponding problems can be reduced to graph computation problems.

Table 9-1 Common graphs in real life

	Vertices	Edges	Graph Computation Problem
Social network	Users	Friend relationships	Potential friend recommendation
Subway lines	Stations	Connectivity between stations	Shortest route recommendation
Solar system	Celestial bodies	Gravitational forces between celestial bodies	Planetary orbit calculation

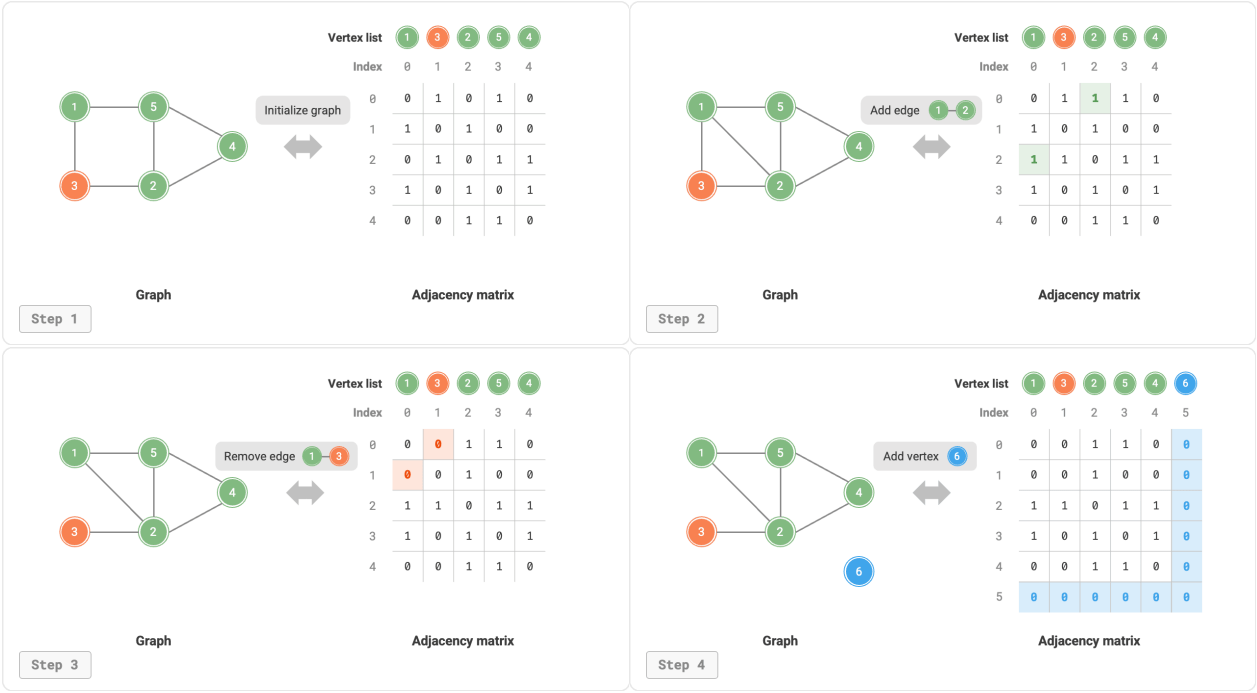
9.2 Basic Operations on Graphs

Basic operations on graphs can be divided into operations on "edges" and operations on "vertices". Under the two representation methods of "adjacency matrix" and "adjacency list", the implementation methods differ.

9.2.1 Implementation Based on Adjacency Matrix

Given an undirected graph with n vertices, the various operations are implemented as shown in Figure 9-7.

- **Adding or removing an edge**: Directly modify the specified edge in the adjacency matrix, using $O(1)$ time. Since it is an undirected graph, both directions of the edge need to be updated simultaneously.
- **Adding a vertex**: Add a row and a column at the end of the adjacency matrix and fill them all with 0s, using $O(n)$ time.
- **Removing a vertex**: Delete a row and a column in the adjacency matrix. The worst case occurs when removing the first row and column, requiring $(n-1)^2$ elements to be "moved up and to the left", thus using $O(n^2)$ time.
- **Initialization**: Pass in n vertices, initialize a vertex list `vertices` of length n, using $O(n)$ time; initialize an adjacency matrix `adjMat` of size $n \times n$, using $O(n^2)$ time.



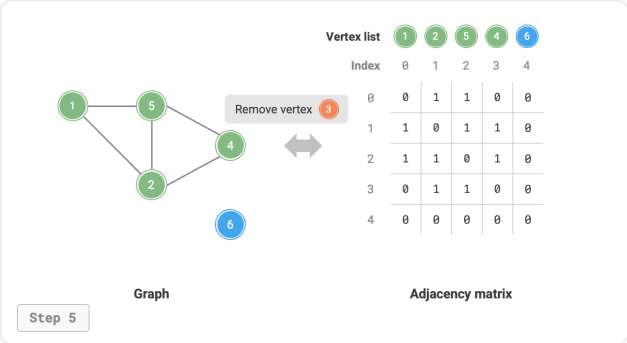


Figure 9-7 Initialization, adding and removing edges, adding and removing vertices in adjacency matrix

The following is the implementation code for graphs represented using an adjacency matrix:

```
// ═══ File: graph_adjacency_matrix.js ═══

/* Undirected graph class based on adjacency matrix */
class GraphAdjMat {
    vertices; // Vertex list, where the element represents the "vertex value" and the index
    ↪ represents the "vertex index"
    adjMat; // Adjacency matrix, where the row and column indices correspond to the "vertex index"

    /* Constructor */
    constructor(vertices, edges) {
        this.vertices = [];
        this.adjMat = [];
        // Add vertex
        for (const val of vertices) {
            this.addVertex(val);
        }
        // Add edge
        // Note that the edges elements represent vertex indices, i.e., corresponding to the
        ↪ vertices element indices
        for (const e of edges) {
            this.addEdge(e[0], e[1]);
        }
    }

    /* Get the number of vertices */
    size() {
        return this.vertices.length;
    }

    /* Add vertex */
    addVertex(val) {
        const n = this.size();
        // Add the value of the new vertex to the vertex list
        this.vertices.push(val);
        // Add a row to the adjacency matrix
        const newRow = [];
        for (let j = 0; j < n; j++) {
            newRow.push(0);
        }
        this.adjMat.push(newRow);
```

```
        // Add a column to the adjacency matrix
        for (const row of this.adjMat) {
            row.push(0);
        }
    }

    /* Remove vertex */
    removeVertex(index) {
        if (index >= this.size()) {
            throw new RangeError('Index Out Of Bounds Exception');
        }
        // Remove the vertex at index from the vertex list
        this.vertices.splice(index, 1);

        // Remove the row at index from the adjacency matrix
        this.adjMat.splice(index, 1);
        // Remove the column at index from the adjacency matrix
        for (const row of this.adjMat) {
            row.splice(index, 1);
        }
    }

    /* Add edge */
    // Parameters i, j correspond to the vertices element indices
    addEdge(i, j) {
        // Handle index out of bounds and equality
        if (i < 0 || j < 0 || i >= this.size() || j >= this.size() || i === j) {
            throw new RangeError('Index Out Of Bounds Exception');
        }
        // In undirected graph, adjacency matrix is symmetric about main diagonal, i.e.,
        ↪ satisfies (i, j) === (j, i)
        this.adjMat[i][j] = 1;
        this.adjMat[j][i] = 1;
    }

    /* Remove edge */
    // Parameters i, j correspond to the vertices element indices
    removeEdge(i, j) {
        // Handle index out of bounds and equality
        if (i < 0 || j < 0 || i >= this.size() || j >= this.size() || i === j) {
            throw new RangeError('Index Out Of Bounds Exception');
        }
        this.adjMat[i][j] = 0;
        this.adjMat[j][i] = 0;
    }

    /* Print adjacency matrix */
    print() {
        console.log('Vertex list = ', this.vertices);
        console.log('Adjacency matrix =', this.adjMat);
    }
}
```

9.2.2 Implementation Based on Adjacency List

Given an undirected graph with a total of n vertices and m edges, the various operations can be implemented as shown in Figure 9-8.

- **Adding an edge**: Add the edge at the end of the corresponding vertex's linked list, using $O(1)$ time. Since it is an undirected graph, edges in both directions need to be added simultaneously.
- **Removing an edge**: Find and remove the specified edge in the corresponding vertex's linked list, using $O(m)$ time. In an undirected graph, edges in both directions need to be removed simultaneously.
- **Adding a vertex**: Add a linked list in the adjacency list and set the new vertex as the head node of the list, using $O(1)$ time.
- **Removing a vertex**: Traverse the entire adjacency list and remove all edges containing the specified vertex, using $O(n + m)$ time.
- **Initialization**: Create n vertices and $2m$ edges in the adjacency list, using $O(n + m)$ time.

Figure 9-8 Initialization, adding and removing edges, adding and removing vertices in adjacency list

The following is the adjacency list code implementation. Compared to Figure 9-8, the actual code has the following differences.

- For convenience in adding and removing vertices, and to simplify the code, we use lists (dynamic arrays) instead of linked lists.

- A hash table is used to store the adjacency list, where `key` is the vertex instance and `value` is the list (linked list) of adjacent vertices for that vertex.

Additionally, we use the `Vertex` class to represent vertices in the adjacency list. The reason for this is: if we used list indices to distinguish different vertices as with adjacency matrices, then to delete the vertex at index i, we would need to traverse the entire adjacency list and decrement all indices greater than i by 1, which is very inefficient. However, if each vertex is a unique `Vertex` instance, deleting a vertex does not require modifying other vertices.

```
// ═══ File: graph_adjacency_list.js ═══

/* Undirected graph class based on adjacency list */
class GraphAdjList {
    // Adjacency list, key: vertex, value: all adjacent vertices of that vertex
    adjList;

    /* Constructor */
    constructor(edges) {
        this.adjList = new Map();
        // Add all vertices and edges
        for (const edge of edges) {
            this.addVertex(edge[0]);
            this.addVertex(edge[1]);
            this.addEdge(edge[0], edge[1]);
        }
    }

    /* Get the number of vertices */
    size() {
        return this.adjList.size;
    }

    /* Add edge */
    addEdge(vet1, vet2) {
        if (
            !this.adjList.has(vet1) ||
            !this.adjList.has(vet2) ||
            vet1 === vet2
        ) {
            throw new Error('Illegal Argument Exception');
        }
        // Add edge vet1 - vet2
        this.adjList.get(vet1).push(vet2);
        this.adjList.get(vet2).push(vet1);
    }

    /* Remove edge */
    removeEdge(vet1, vet2) {
        if (
            !this.adjList.has(vet1) ||
            !this.adjList.has(vet2) ||
            vet1 === vet2 ||
            this.adjList.get(vet1).indexOf(vet2) === -1
        ) {
            throw new Error('Illegal Argument Exception');
        }
        // Remove edge vet1 - vet2
        this.adjList.get(vet1).splice(this.adjList.get(vet1).indexOf(vet2), 1);
```

```
        this.adjList.get(vet2).splice(this.adjList.get(vet2).indexOf(vet1), 1);
    }

    /* Add vertex */
    addVertex(vet) {
        if (this.adjList.has(vet)) return;
        // Add a new linked list in the adjacency list
        this.adjList.set(vet, []);
    }

    /* Remove vertex */
    removeVertex(vet) {
        if (!this.adjList.has(vet)) {
            throw new Error('Illegal Argument Exception');
        }
        // Remove the linked list corresponding to vertex vet in the adjacency list
        this.adjList.delete(vet);
        // Traverse the linked lists of other vertices and remove all edges containing vet
        for (const set of this.adjList.values()) {
            const index = set.indexOf(vet);
            if (index > -1) {
                set.splice(index, 1);
            }
        }
    }

    /* Print adjacency list */
    print() {
        console.log('Adjacency list =');
        for (const [key, value] of this.adjList) {
            const tmp = [];
            for (const vertex of value) {
                tmp.push(vertex.val);
            }
            console.log(key.val + ': ' + tmp.join());
        }
    }
}
```

9.2.3 Efficiency Comparison

Assuming the graph has n vertices and m edges, Table 9-2 compares the time efficiency and space efficiency of adjacency matrices and adjacency lists. Note that the adjacency list (linked list) corresponds to the implementation in this text, while the adjacency list (hash table) refers specifically to the implementation where all linked lists are replaced with hash tables.

Table 9-2 Comparison of adjacency matrix and adjacency list

	Adjacency matrix	Adjacency list (linked list)	Adjacency list (hash table)
Determine adjacency	$O(1)$	$O(n)$	$O(1)$
Add an edge	$O(1)$	$O(1)$	$O(1)$

	Adjacency matrix	Adjacency list (linked list)	Adjacency list (hash table)
Remove an edge	$O(1)$	$O(n)$	$O(1)$
Add a vertex	$O(n)$	$O(1)$	$O(1)$
Remove a vertex	$O(n^2)$	$O(n+m)$	$O(n)$
Memory space usage	$O(n^2)$	$O(n+m)$	$O(n+m)$

Observing Table 9-2, it appears that the adjacency list (hash table) has the best time efficiency and space efficiency. However, in practice, operating on edges in the adjacency matrix is more efficient, requiring only a single array access or assignment operation. Overall, adjacency matrices embody the principle of "trading space for time", while adjacency lists embody "trading time for space".

9.3 Graph Traversal

Trees represent "one-to-many" relationships, while graphs have a higher degree of freedom and can represent any "many-to-many" relationships. Therefore, we can view trees as a special case of graphs. Clearly, **tree traversal operations are also a special case of graph traversal operations**.

Both graphs and trees require the application of search algorithms to implement traversal operations. Graph traversal methods can also be divided into two types: breadth-first traversal and depth-first traversal.

9.3.1 Breadth-First Search

Breadth-first search is a near-to-far traversal method that, starting from a certain node, always prioritizes visiting the nearest vertices and expands outward layer by layer. As shown in Figure 9-9, starting from the top-left vertex, first traverse all adjacent vertices of that vertex, then traverse all adjacent vertices of the next vertex, and so on, until all vertices have been visited.

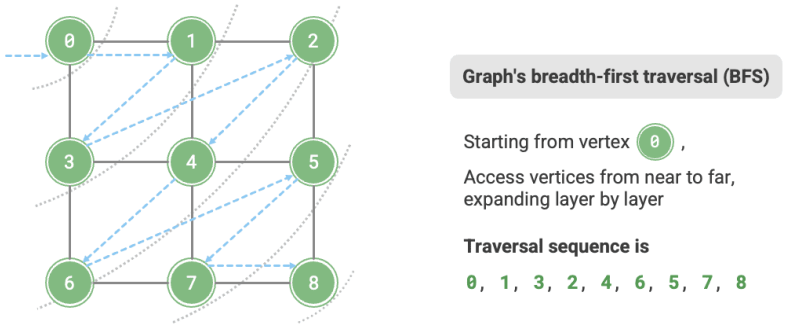


Figure 9-9 Breadth-first search of a graph

1. Algorithm Implementation

BFS is typically implemented with the help of a queue, as shown in the code below. The queue has a "first in, first out" property, which aligns with the BFS idea of "near to far".

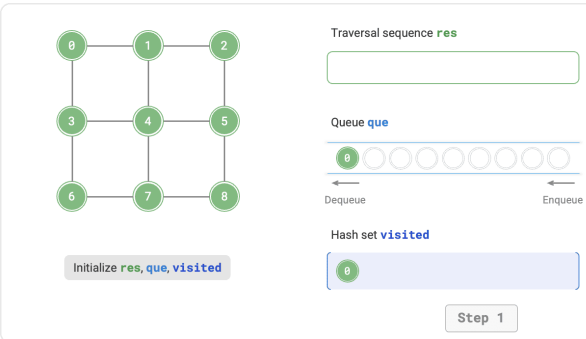
1. Add the starting vertex `startVet` to the queue and begin the loop.
2. In each iteration of the loop, pop the vertex at the front of the queue and record it as visited, then add all adjacent vertices of that vertex to the back of the queue.
3. Repeat step `2.` until all vertices have been visited.

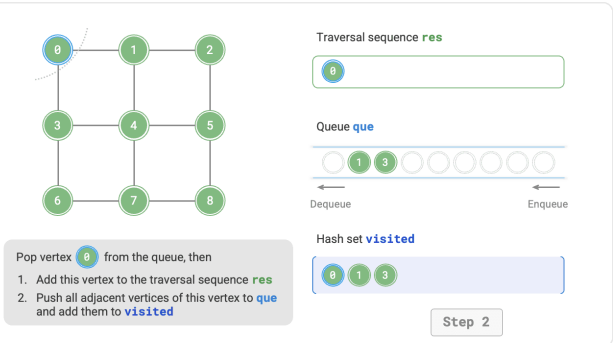
To prevent revisiting vertices, we use a hash set `visited` to record which nodes have been visited.

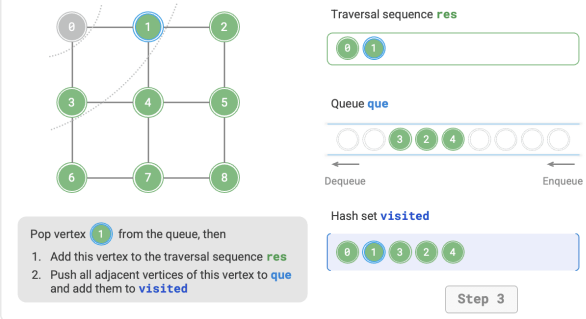
> **Tip**
> A hash set can be viewed as a hash table that stores only `key` without storing `value`. It can perform addition, deletion, lookup, and modification operations on `key` in $O(1)$ time complexity. Based on the uniqueness of `key`, hash sets are typically used for data deduplication and similar scenarios.

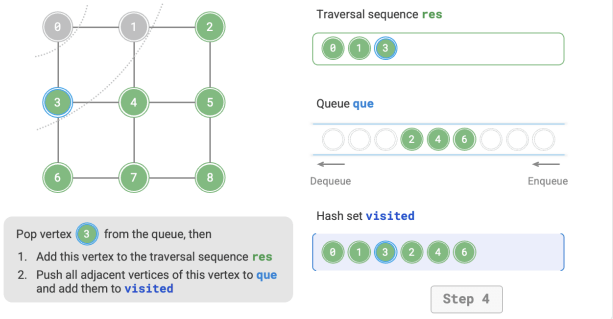
```
// ═══ File: graph_bfs.js ═══

/* Breadth-first traversal */
// Use adjacency list to represent the graph, in order to obtain all adjacent vertices of a
↪  specified vertex
function graphBFS(graph, startVet) {
    // Vertex traversal sequence
    const res = [];
    // Hash set for recording vertices that have been visited
    const visited = new Set();
    visited.add(startVet);
    // Queue used to implement BFS
    const que = [startVet];
    // Starting from vertex vet, loop until all vertices are visited
    while (que.length) {
        const vet = que.shift(); // Dequeue the front vertex
        res.push(vet); // Record visited vertex
        // Traverse all adjacent vertices of this vertex
        for (const adjVet of graph.adjList.get(vet) ?? []) {
            if (visited.has(adjVet)) {
                continue; // Skip vertices that have been visited
            }
            que.push(adjVet); // Only enqueue unvisited vertices
            visited.add(adjVet); // Mark this vertex as visited
        }
    }
    // Return vertex traversal sequence
    return res;
}
```

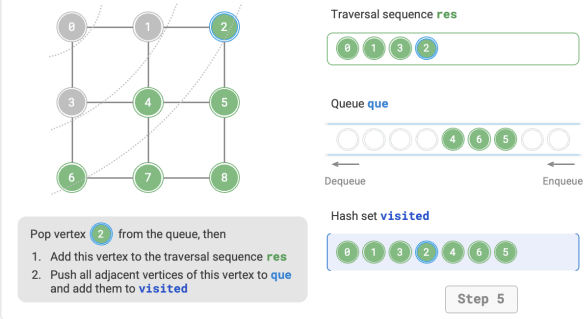
The code is relatively abstract; it is recommended to refer to Figure 9-10 to deepen understanding.

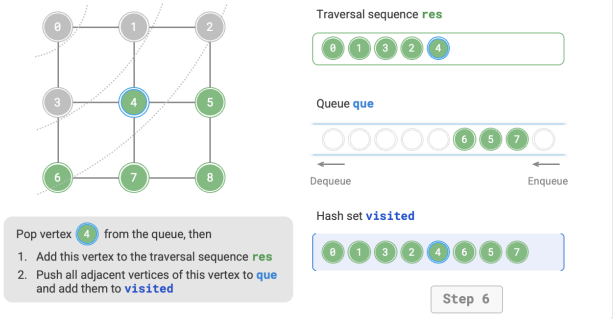

Traversal sequence res
Queue que
Dequeue
Enqueue
Hash set visited
Initialize res, que, visited
Step 1

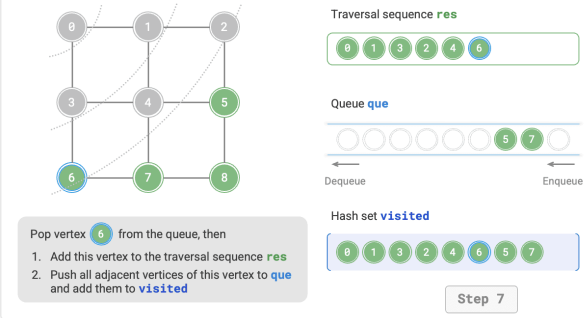

Traversal sequence res
Queue que
Dequeue
Enqueue
Hash set visited
Pop vertex 0 from the queue, then
1. Add this vertex to the traversal sequence res
2. Push all adjacent vertices of this vertex to que and add them to visited
Step 2

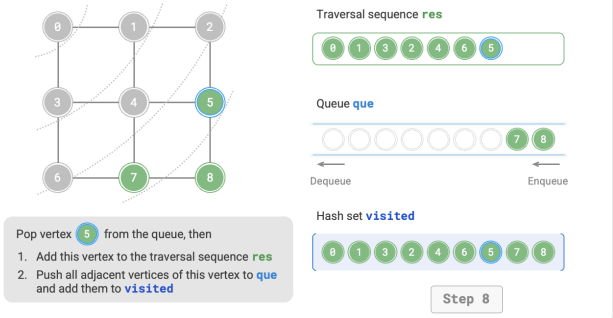

Traversal sequence res
Queue que
Dequeue
Enqueue
Hash set visited
Pop vertex 1 from the queue, then
1. Add this vertex to the traversal sequence res
2. Push all adjacent vertices of this vertex to que and add them to visited
Step 3

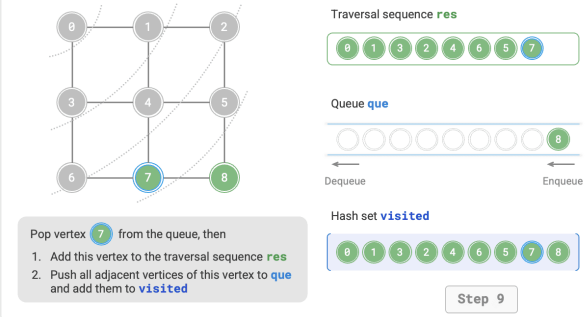

Traversal sequence res
Queue que
Dequeue
Enqueue
Hash set visited
Pop vertex 3 from the queue, then
1. Add this vertex to the traversal sequence res
2. Push all adjacent vertices of this vertex to que and add them to visited
Step 4

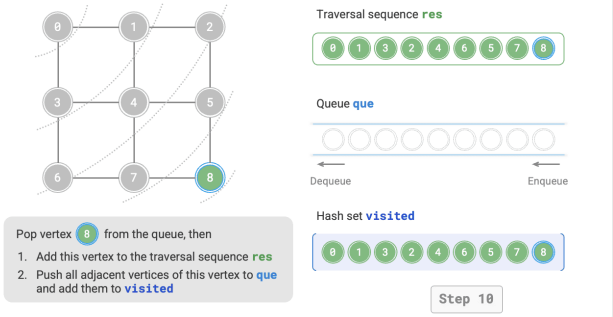

Traversal sequence res
Queue que
Dequeue
Enqueue
Hash set visited
Pop vertex 2 from the queue, then
1. Add this vertex to the traversal sequence res
2. Push all adjacent vertices of this vertex to que and add them to visited
Step 5


Traversal sequence res
Queue que
Dequeue
Enqueue
Hash set visited
Pop vertex 4 from the queue, then
1. Add this vertex to the traversal sequence res
2. Push all adjacent vertices of this vertex to que and add them to visited
Step 6


Traversal sequence res
Queue que
Dequeue
Enqueue
Hash set visited
Pop vertex 6 from the queue, then
1. Add this vertex to the traversal sequence res
2. Push all adjacent vertices of this vertex to que and add them to visited
Step 7


Traversal sequence res
Queue que
Dequeue
Enqueue
Hash set visited
Pop vertex 5 from the queue, then
1. Add this vertex to the traversal sequence res
2. Push all adjacent vertices of this vertex to que and add them to visited
Step 8


Traversal sequence res
Queue que
Dequeue
Enqueue
Hash set visited
Pop vertex 7 from the queue, then
1. Add this vertex to the traversal sequence res
2. Push all adjacent vertices of this vertex to que and add them to visited
Step 9


Traversal sequence res
Queue que
Dequeue
Enqueue
Hash set visited
Pop vertex 8 from the queue, then
1. Add this vertex to the traversal sequence res
2. Push all adjacent vertices of this vertex to que and add them to visited
Step 10

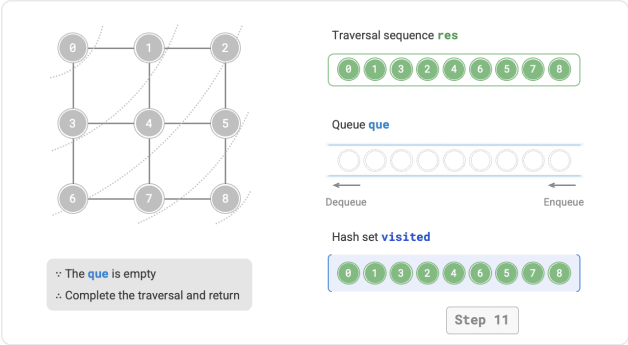


Figure 9-10 Steps of breadth-first search of a graph

Is the breadth-first traversal sequence unique?

Not unique. Breadth-first search only requires traversing in a "near to far" order, **and the traversal order of vertices at the same distance can be arbitrarily shuffled**. Taking Figure 9-10 as an example, the visit order of vertices 1 and 3 can be swapped, as can the visit order of vertices 2, 4, and 6.

2. Complexity Analysis

Time complexity: All vertices will be enqueued and dequeued once, using $O(|V|)$ time; in the process of traversing adjacent vertices, since it is an undirected graph, all edges will be visited 2 times, using $O(2|E|)$ time; overall using $O(|V| + |E|)$ time.

Space complexity: The list `res`, hash set `visited`, and queue `que` can contain at most $|V|$ vertices, using $O(|V|)$ space.

9.3.2 Depth-First Search

Depth-first search is a traversal method that prioritizes going as far as possible, then backtracks when no path remains. As shown in Figure 9-11, starting from the top-left vertex, visit an adjacent vertex of the current vertex, continuing until reaching a dead end, then return and continue going as far as possible before returning again, and so on, until all vertices have been traversed.

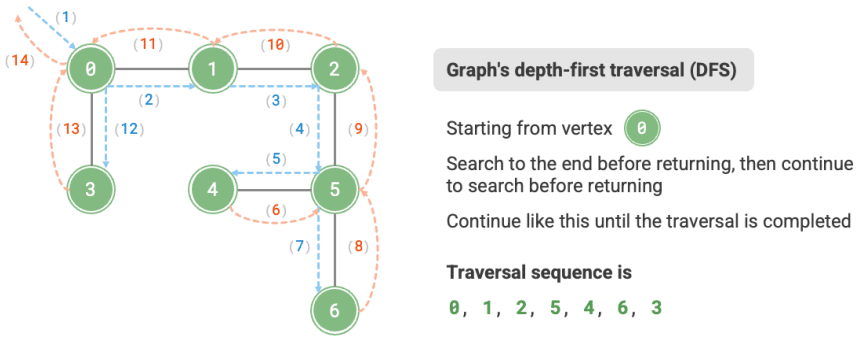


Figure 9-11 Depth-first search of a graph

1. Algorithm Implementation

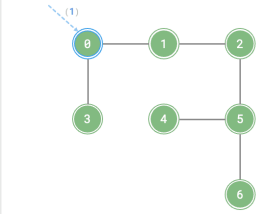
This "go as far as possible then return" algorithm paradigm is typically implemented using recursion. Similar to breadth-first search, in depth-first search we also need a hash set `visited` to record visited vertices and avoid revisiting.

```
// ═══ File: graph_dfs.js ═══

/* Depth-first traversal */
// Use adjacency list to represent the graph, in order to obtain all adjacent vertices of a
↪ specified vertex
function dfs(graph, visited, res, vet) {
    res.push(vet); // Record visited vertex
    visited.add(vet); // Mark this vertex as visited
    // Traverse all adjacent vertices of this vertex
    for (const adjVet of graph.adjList.get(vet)) {
        if (visited.has(adjVet)) {
            continue; // Skip vertices that have been visited
        }
        // Recursively visit adjacent vertices
        dfs(graph, visited, res, adjVet);
    }
}

/* Depth-first traversal */
// Use adjacency list to represent the graph, in order to obtain all adjacent vertices of a
↪ specified vertex
function graphDFS(graph, startVet) {
    // Vertex traversal sequence
    const res = [];
    // Hash set for recording vertices that have been visited
    const visited = new Set();
    dfs(graph, visited, res, startVet);
    return res;
}
```

The algorithm flow of depth-first search is shown in Figure 9-12.

- **Straight dashed lines represent downward recursion**, indicating that a new recursive method has been initiated to visit a new vertex.
- **Curved dashed lines represent upward backtracking**, indicating that this recursive method has returned to the position where it was initiated.

To deepen understanding, it is recommended to combine Figure 9-12 with the code to mentally simulate (or draw out) the entire DFS process, including when each recursive method is initiated and when it returns.



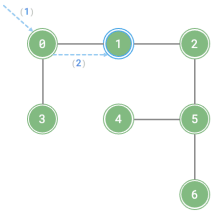
Traversal sequence `res`

Hash set `visited`

Calling stage:

Visit vertex 0

Step 1



Traversal sequence `res`

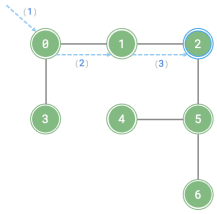
Hash set `visited`

Calling stage:

Traverse adjacent vertices of vertex 0

Recursively visit vertex 1

Step 2



Traversal sequence `res`

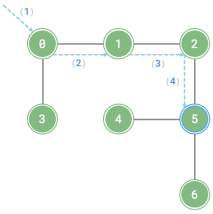
Hash set `visited`

Calling stage:

Traverse adjacent vertices of vertex 1

Recursively visit vertex 2

Step 3



Traversal sequence `res`

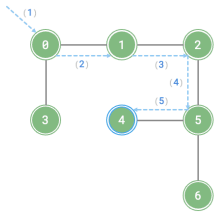
Hash set `visited`

Calling stage:

Traverse adjacent vertices of vertex 2

Recursively visit vertex 5

Step 4



Traversal sequence `res`

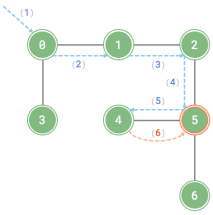
Hash set `visited`

Calling stage:

Traverse adjacent vertices of vertex 5

Recursively visit vertex 4

Step 5



Traversal sequence `res`

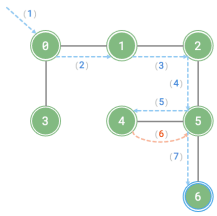
Hash set `visited`

Returning stage:

All adjacent vertices of vertex 4 have been visited

Thus, the method returns, backtracking to vertex 5

Step 6



Traversal sequence `res`

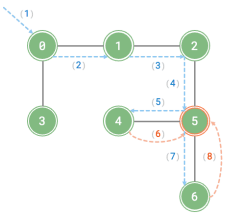
Hash set `visited`

Calling stage:

Traverse adjacent vertices of vertex 5

Recursively visit vertex 6

Step 7



Traversal sequence `res`

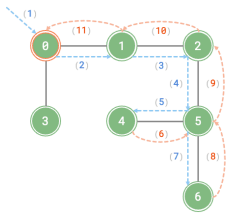
Hash set `visited`

Returning stage:

All adjacent vertices of vertex 6 have been visited

Thus, the method returns, backtracking to vertex 5

Step 8



Traversal sequence `res`

Hash set `visited`

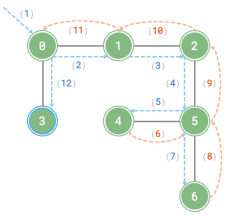
Returning stage:

All adjacent vertices of vertex 6 have been visited

Thus, the method returns, backtracking to vertex 5

Continue returning, eventually backtracking to vertex 0

Step 9



Traversal sequence `res`

Hash set `visited`

Calling stage:

Traverse adjacent vertices of vertex 0

Recursively visit vertex 3

Step 10

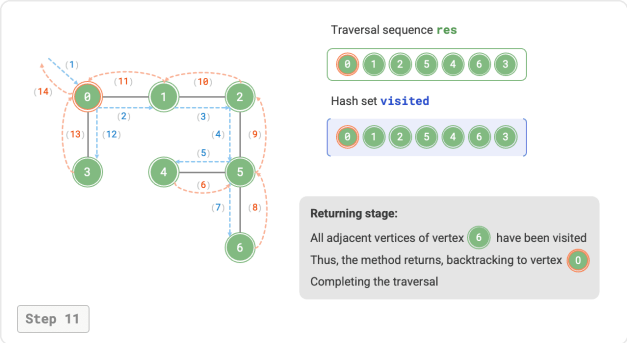


Figure 9-12 Steps of depth-first search of a graph

Is the depth-first traversal sequence unique?

Similar to breadth-first search, the order of depth-first traversal sequences is also not unique. Given a certain vertex, exploring in any direction first is valid, meaning the order of adjacent vertices can be arbitrarily shuffled, all being depth-first search.

Taking tree traversal as an example, "root $\rightarrow$ left $\rightarrow$ right", "left $\rightarrow$ root $\rightarrow$ right", and "left $\rightarrow$ right $\rightarrow$ root" correspond to pre-order, in-order, and post-order traversals, respectively. They represent three different traversal priorities, yet all three belong to depth-first search.

2. Complexity Analysis

Time complexity: All vertices will be visited 1 time, using $O(|V|)$ time; all edges will be visited 2 times, using $O(2|E|)$ time; overall using $O(|V| + |E|)$ time.

Space complexity: The list `res` and hash set `visited` can contain at most $|V|$ vertices, and the maximum recursion depth is $|V|$, therefore using $O(|V|)$ space.

9.4 Summary

1. Key Review

- Graphs consist of vertices and edges and can be represented as a set of vertices and a set of edges.
- Compared to linear relationships (linked lists) and divide-and-conquer relationships (trees), network relationships (graphs) have a higher degree of freedom and are therefore more complex.
- Directed graphs have edges with directionality, connected graphs have all vertices reachable from any vertex, and weighted graphs have edges that each contain a weight variable.
- Adjacency matrices use matrices to represent graphs, where each row (column) represents a vertex, and matrix elements represent edges, using 1 or 0 to indicate whether two vertices have an edge or not. Adjacency matrices are highly efficient for addition, deletion, lookup, and modification operations, but consume significant space.

- Adjacency lists use multiple linked lists to represent graphs, where the i-th linked list corresponds to vertex i and stores all adjacent vertices of that vertex. Adjacency lists are more space-efficient than adjacency matrices, but have lower time efficiency because they require traversing linked lists to find edges.
- When linked lists in adjacency lists become too long, they can be converted to red-black trees or hash tables, thereby improving lookup efficiency.
- From an algorithmic perspective, adjacency matrices embody "trading space for time", while adjacency lists embody "trading time for space".
- Graphs can be used to model various real-world systems, such as social networks and subway lines.
- Trees are a special case of graphs, and tree traversal is a special case of graph traversal.
- Breadth-first search of graphs is a near-to-far, layer-by-layer expansion search method, typically implemented using a queue.
- Depth-first search of graphs is a search method that prioritizes going as far as possible and backtracks when no path remains, commonly implemented using recursion.

2. Q & A

Q: Is a path defined as a sequence of vertices or a sequence of edges?

The definitions in different language versions of Wikipedia are inconsistent: the English version states "a path is a sequence of edges", while the Chinese version states "a path is a sequence of vertices". The following is the original English text: In graph theory, a path in a graph is a finite or infinite sequence of edges which joins a sequence of vertices.

In this text, a path is viewed as a sequence of edges, not a sequence of vertices. This is because there may be multiple edges connecting two vertices, in which case each edge corresponds to a path.

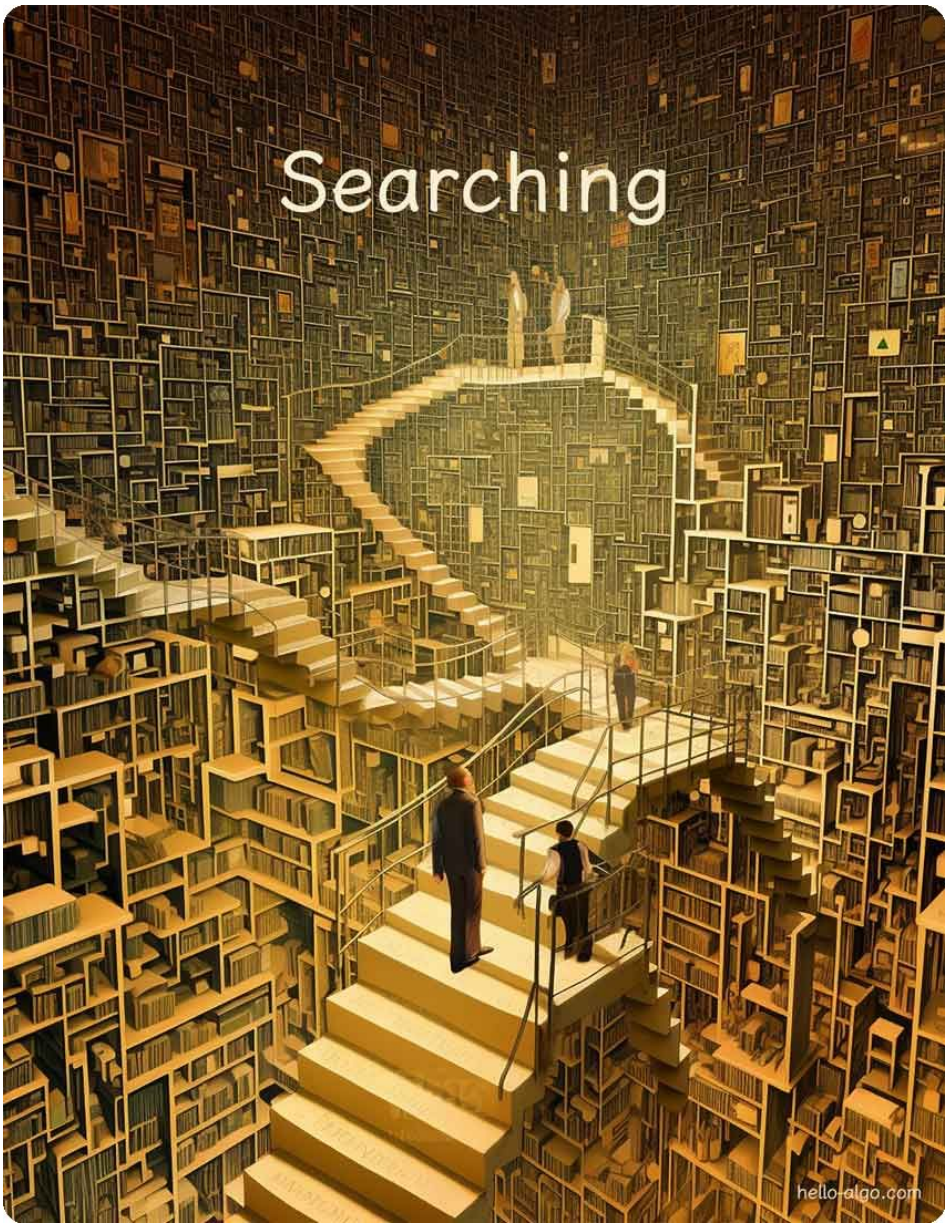
Q: In a disconnected graph, will there be unreachable vertices?

In a disconnected graph, starting from a certain vertex, at least one vertex cannot be reached. Traversing a disconnected graph requires setting multiple starting points to traverse all connected components of the graph.

Q: In an adjacency list, is there a requirement for the order of "all vertices connected to that vertex"?

It can be in any order. However, in practical applications, it may be necessary to sort according to specified rules, such as the order in which vertices were added, or the order of vertex values, which helps quickly find vertices "with certain extreme values".

Chapter 10. Searching



Abstract

Searching is an adventure into the unknown, where we may need to traverse every corner of the mysterious space, or we may be able to quickly lock onto the target.

In this journey of discovery, each exploration may yield an unexpected answer.

10.1 Binary Search

Binary search is an efficient searching algorithm based on the divide-and-conquer strategy. It leverages the orderliness of data to reduce the search range by half in each round until the target element is found or the search interval becomes empty.

Question

Given an array `nums` of length n with elements arranged in ascending order and no duplicates, search for and return the index of element `target` in the array. If the array does not contain the element, return -1. An example is shown in Figure 10-1.

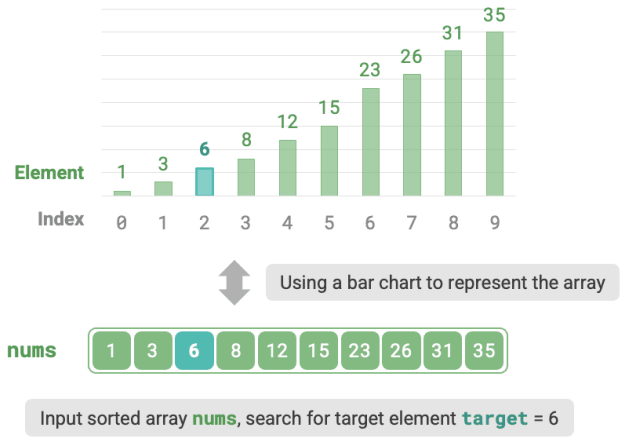


Figure 10-1 Binary search example data

As shown in Figure 10-2, we first initialize pointers $i = 0$ and $j = n - 1$, pointing to the first and last elements of the array respectively, representing the search interval $[0, n - 1]$. Note that square brackets denote a closed interval, which includes the boundary values themselves.

Next, perform the following two steps in a loop:

1. Calculate the midpoint index $m = \lfloor (i + j)/2 \rfloor$, where $\lfloor \ \rfloor$ denotes the floor operation.
2. Compare `nums[m]` and `target`, which results in three cases:
    1. When `nums[m] < target`, it indicates that `target` is in the interval $[m + 1, j]$, so execute $i = m + 1$.
    2. When `nums[m] > target`, it indicates that `target` is in the interval $[i, m - 1]$, so execute $j = m - 1$.
    3. When `nums[m] = target`, it indicates that `target` has been found, so return index m.

If the array does not contain the target element, the search interval will eventually shrink to empty. In this case, return -1.

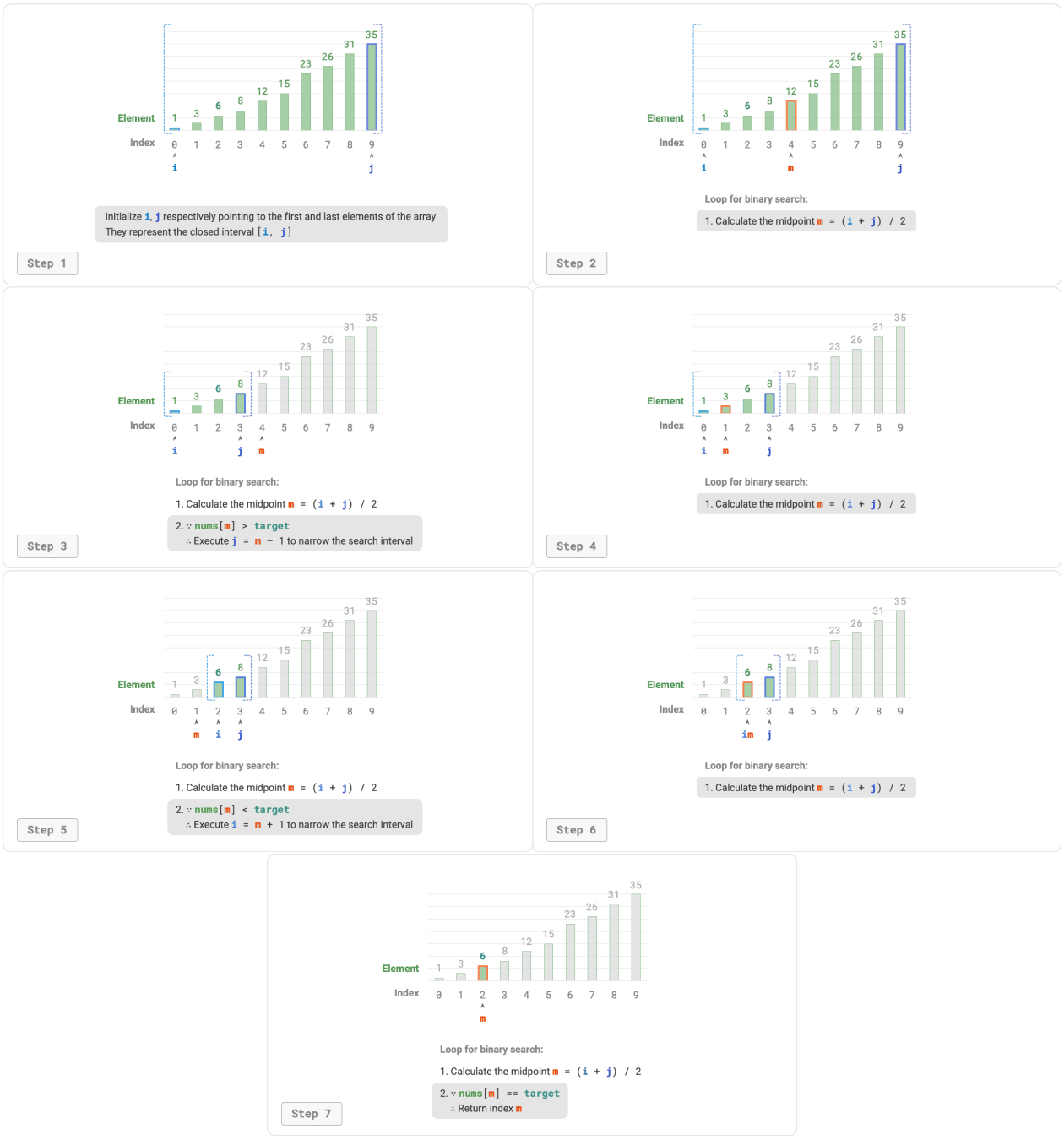


Figure 10-2 Binary search process

It's worth noting that since both i and j are of `int` type, $i + j$ **may exceed the range of the** `int` **type**. To avoid large number overflow, we typically use the formula $m = \lfloor i + (j - i)/2 \rfloor$ to calculate the midpoint.

The code is shown below:

```
// === File: binary_search.js ===
```

```javascript
/* Binary search (closed interval on both sides) */
function binarySearch(nums, target) {
    // Initialize closed interval [0, n-1], i.e., i, j point to the first and last elements of
    ↪ the array
    let i = 0,
        j = nums.length - 1;
    // Loop, exit when the search interval is empty (empty when i > j)
    while (i <= j) {
        // Calculate midpoint index m, use parseInt() to round down
        const m = parseInt(i + (j - i) / 2);
        if (nums[m] < target)
            // This means target is in the interval [m+1, j]
            i = m + 1;
        else if (nums[m] > target)
            // This means target is in the interval [i, m-1]
            j = m - 1;
        else return m; // Found the target element, return its index
    }
    // Target element not found, return -1
    return -1;
}
```

Time complexity is $O(\log n)$: In the binary loop, the interval is reduced by half each round, so the number of loops is $\log_2 n$.

Space complexity is $O(1)$: Pointers i and j use constant-size space.

10.1.1 Interval Representation Methods

In addition to the closed interval mentioned above, another common interval representation is the "left-closed right-open" interval, defined as $[0, n)$, meaning the left boundary includes itself while the right boundary does not. Under this representation, the interval $[i, j)$ is empty when $i = j$.

We can implement a binary search algorithm with the same functionality based on this representation:

```javascript
// === File: binary_search.js ===

/* Binary search (left-closed right-open interval) */
function binarySearchLCRO(nums, target) {
    // Initialize left-closed right-open interval [0, n), i.e., i, j point to the first element
    ↪ and last element+1
    let i = 0,
        j = nums.length;
    // Loop, exit when the search interval is empty (empty when i = j)
    while (i < j) {
        // Calculate midpoint index m, use parseInt() to round down
        const m = parseInt(i + (j - i) / 2);
        if (nums[m] < target)
            // This means target is in the interval [m+1, j)
            i = m + 1;
        else if (nums[m] > target)
            // This means target is in the interval [i, m)
            j = m;
        // Found the target element, return its index
```

```
        else return m;
    }
    // Target element not found, return -1
    return -1;
}
```

As shown in Figure 10-3, under the two interval representations, the initialization, loop condition, and interval narrowing operations of the binary search algorithm are all different.

Since both the left and right boundaries in the "closed interval" representation are defined as closed, the operations to narrow the interval through pointers i and j are also symmetric. This makes it less error-prone, **so the "closed interval" approach is generally recommended**.

Figure 10-3 Two interval definitions

10.1.2 Advantages and Limitations

Binary search performs well in both time and space aspects.

- Binary search has high time efficiency. With large data volumes, the logarithmic time complexity has significant advantages. For example, when the data size $n = 2^{20}$, linear search requires $2^{20} = 1048576$ loop rounds, while binary search only needs $\log_2 2^{20} = 20$ rounds.
- Binary search requires no extra space. Compared to searching algorithms that require additional space (such as hash-based search), binary search is more space-efficient.

However, binary search is not suitable for all situations, mainly for the following reasons:

- Binary search is only applicable to sorted data. If the input data is unsorted, sorting specifically to use binary search would be counterproductive, as sorting algorithms typically have a time complexity of $O(n \log n)$, which is higher than both linear search and binary search. For scenarios with frequent element insertions, maintaining array orderliness requires inserting elements at specific positions with a time complexity of $O(n)$, which is also very expensive.

- Binary search is only applicable to arrays. Binary search requires jump-style (non-contiguous) element access, and jump-style access has low efficiency in linked lists, making it unsuitable for linked lists or data structures based on linked list implementations.
- For small data volumes, linear search performs better. In linear search, each round requires only 1 comparison operation; while in binary search, it requires 1 addition, 1 division, 1-3 comparison operations, and 1 addition (subtraction), totaling 4-6 unit operations. Therefore, when the data volume n is small, linear search is actually faster than binary search.

10.2 Binary Search Insertion Point

Binary search can not only be used to search for target elements but also to solve many variant problems, such as searching for the insertion position of a target element.

10.2.1 Case Without Duplicate Elements

Question

Given a sorted array `nums` of length n and an element `target`, where the array contains no duplicate elements. Insert `target` into the array `nums` while maintaining its sorted order. If the array already contains the element `target`, insert it to its left. Return the index of `target` in the array after insertion. An example is shown in Figure 10-4.

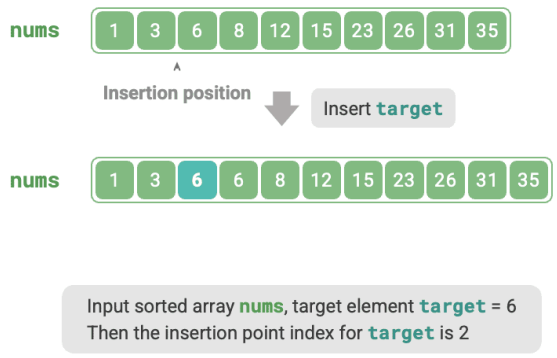


Figure 10-4 Binary search insertion point example data

If we want to reuse the binary search code from the previous section, we need to answer the following two questions.

Question 1: When the array contains `target`, is the insertion point index the same as that element's index?

The problem requires inserting `target` to the left of equal elements, which means the newly inserted `target` replaces the position of the original `target`. In other words, **when the array contains `target`, the insertion point index is the index of that `target`**.

Question 2: When the array does not contain `target`, what is the insertion point index?

Further consider the binary search process: When `nums[m] < target`, i moves, which means pointer i is approaching elements greater than or equal to `target`. Similarly, pointer j is always approaching elements less than or equal to `target`.

Therefore, when the binary search ends, we must have: i points to the first element greater than `target`, and j points to the first element less than `target`. **It's easy to see that when the array does not contain `target`, the insertion index is i**. The code is shown below:

```
// ═══ File: binary_search_insertion.js ═══

/* Binary search for insertion point (no duplicate elements) */
function binarySearchInsertionSimple(nums, target) {
    let i = 0,
        j = nums.length - 1; // Initialize closed interval [0, n-1]
    while (i <= j) {
        const m = Math.floor(i + (j - i) / 2); // Calculate midpoint index m, use Math.floor() to
        ↪ round down
        if (nums[m] < target) {
            i = m + 1; // target is in the interval [m+1, j]
        } else if (nums[m] > target) {
            j = m - 1; // target is in the interval [i, m-1]
        } else {
            return m; // Found target, return insertion point m
        }
    }
    // Target not found, return insertion point i
    return i;
}
```

10.2.2 Case with Duplicate Elements

> **Question**
> Based on the previous problem, assume the array may contain duplicate elements, with everything else remaining the same.

Suppose there are multiple `target` elements in the array. Ordinary binary search can only return the index of one `target`, **and cannot determine how many `target` elements are to the left and right of that element**.

The problem requires inserting the target element at the leftmost position, **so we need to find the index of the leftmost `target` in the array**. Initially, consider implementing this through the steps shown in Figure 10-5:

1. Perform binary search to obtain the index of any `target`, denoted as k.
2. Starting from index k, perform linear traversal to the left, and return when the leftmost `target` is found.

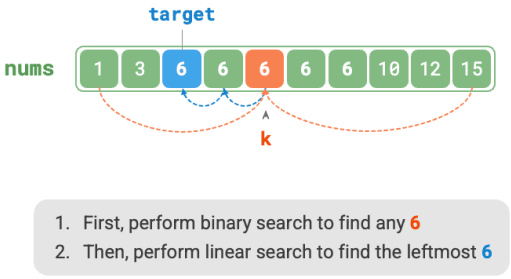


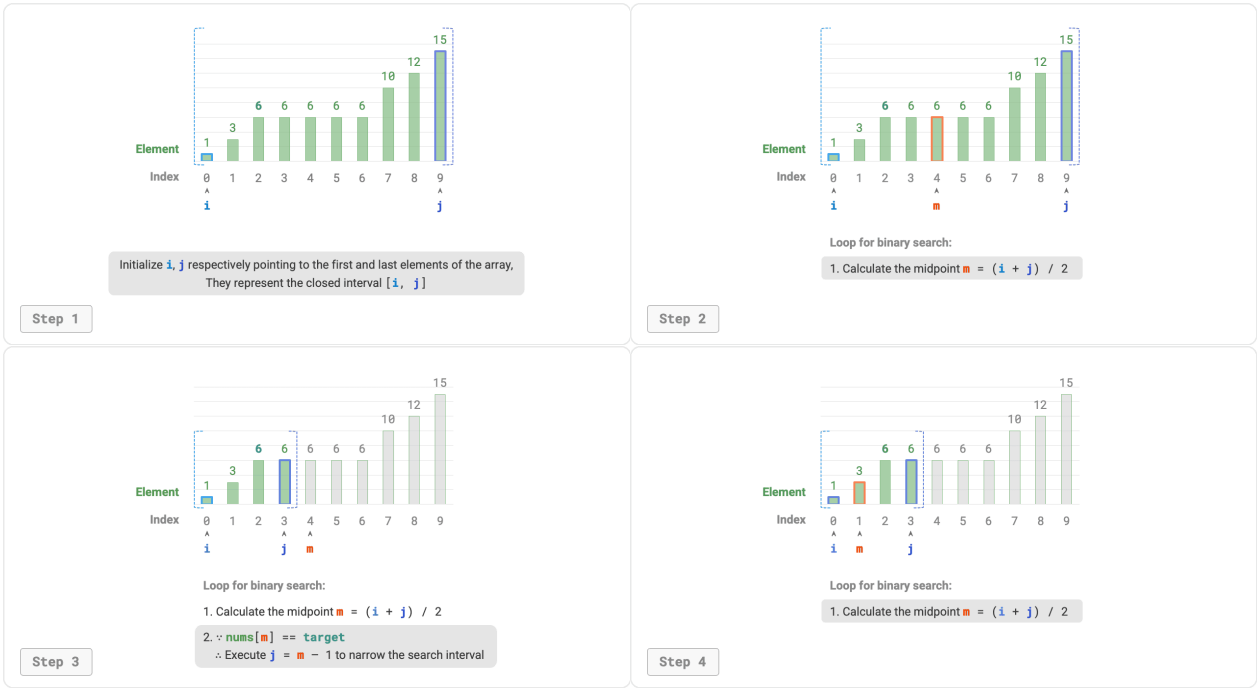
Figure 10-5 Linear search for insertion point of duplicate elements

Although this method works, it includes linear search, resulting in a time complexity of $O(n)$. When the array contains many duplicate `target` elements, this method is very inefficient.

Now consider extending the binary search code. As shown in Figure 10-6, the overall process remains unchanged: calculate the midpoint index m in each round, then compare `target` with `nums[m]`, divided into the following cases:

- When `nums[m] < target` or `nums[m] > target`, it means `target` has not been found yet, so use the ordinary binary search interval narrowing operation to **make pointers i and j approach `target`**.
- When `nums[m] == target`, it means elements less than `target` are in the interval $[i, m-1]$, so use $j = m - 1$ to narrow the interval, thereby **making pointer j approach elements less than `target`**.

After the loop completes, i points to the leftmost `target`, and j points to the first element less than `target`, **so index i is the insertion point**.



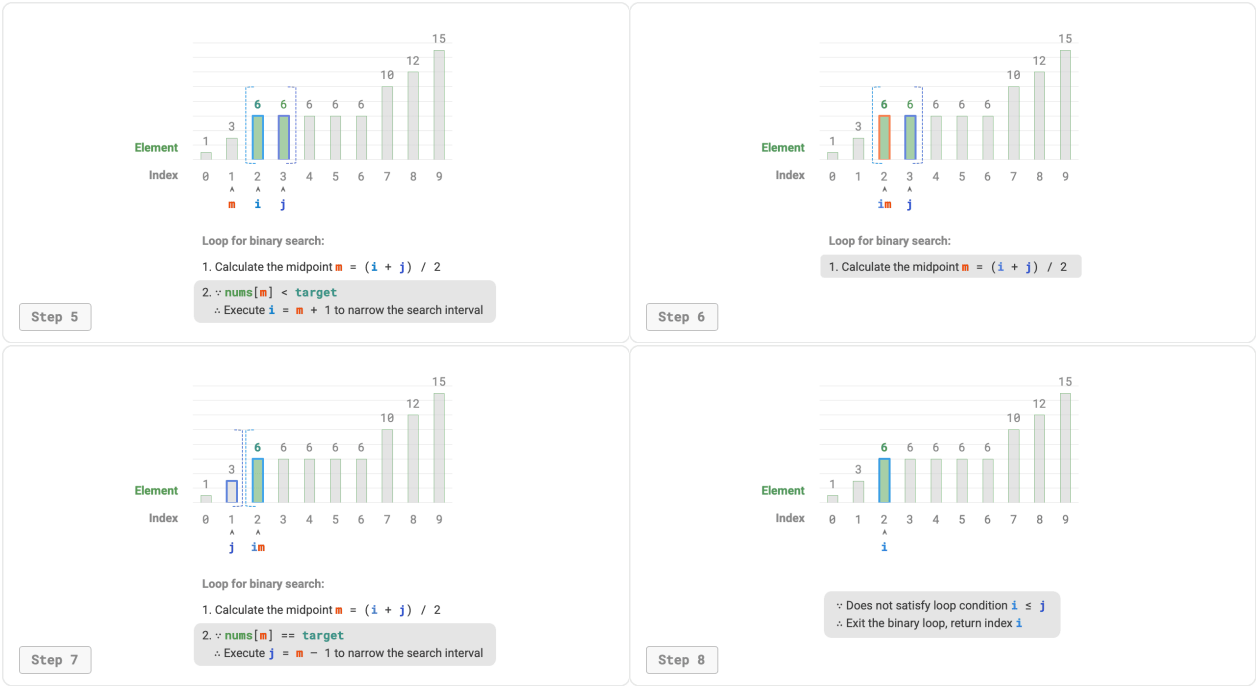


Figure 10-6 Steps for binary search insertion point of duplicate elements

Observe the following code: the operations for branches `nums[m] > target` and `nums[m] == target` are the same, so the two can be merged.

Even so, we can still keep the conditional branches expanded, as the logic is clearer and more readable.

```
// === File: binary_search_insertion.js ===

/* Binary search for insertion point (with duplicate elements) */
function binarySearchInsertion(nums, target) {
    let i = 0,
        j = nums.length - 1; // Initialize closed interval [0, n-1]
    while (i <= j) {
        const m = Math.floor(i + (j - i) / 2); // Calculate midpoint index m, use Math.floor() to
         ↪  round down
        if (nums[m] < target) {
            i = m + 1; // target is in the interval [m+1, j]
        } else if (nums[m] > target) {
            j = m - 1; // target is in the interval [i, m-1]
        } else {
            j = m - 1; // The first element less than target is in the interval [i, m-1]
        }
    }
    // Return insertion point i
    return i;
}
```

Tip

The code in this section all uses the "closed interval" approach. Interested readers can implement the "left-closed right-open" approach themselves.

Overall, binary search is simply about setting search targets for pointers i and j separately. The target could be a specific element (such as `target`) or a range of elements (such as elements less than `target`).

Through continuous binary iterations, both pointers i and j gradually approach their preset targets. Ultimately, they either successfully find the answer or stop after crossing the boundaries.

10.3 Binary Search Edge Cases

10.3.1 Finding the Left Boundary

> **Question**
> Given a sorted array `nums` of length n that may contain duplicate elements, return the index of the leftmost element `target` in the array. If the array does not contain the element, return -1.

Recall the method for finding the insertion point with binary search. After the search completes, i points to the leftmost `target`, **so finding the insertion point is essentially finding the index of the leftmost** `target`.

Consider implementing the left boundary search using the insertion point finding function. Note that the array may not contain `target`, which could result in the following two cases:

- The insertion point index i is out of bounds.
- The element `nums[i]` is not equal to `target`.

When either of these situations occurs, simply return -1. The code is shown below:

```javascript
// === File: binary_search_edge.js ===

/* Binary search for the leftmost target */
function binarySearchLeftEdge(nums, target) {
    // Equivalent to finding the insertion point of target
    const i = binarySearchInsertion(nums, target);
    // Target not found, return -1
    if (i === nums.length || nums[i] !== target) {
        return -1;
    }
    // Found target, return index i
    return i;
}
```

10.3.2 Finding the Right Boundary

So how do we find the rightmost `target`? The most direct approach is to modify the code and replace the pointer shrinking operation in the `nums[m] == target` case. The code is omitted here; interested readers can implement it themselves.

Below we introduce two more clever methods.

1. Reusing Left Boundary Search

In fact, we can use the function for finding the leftmost element to find the rightmost element. The specific method is: **Convert finding the rightmost `target` into finding the leftmost `target + 1`**.

As shown in Figure 10-7, after the search completes, pointer i points to the leftmost `target + 1` (if it exists), while j points to the rightmost `target`, **so we can simply return j**.

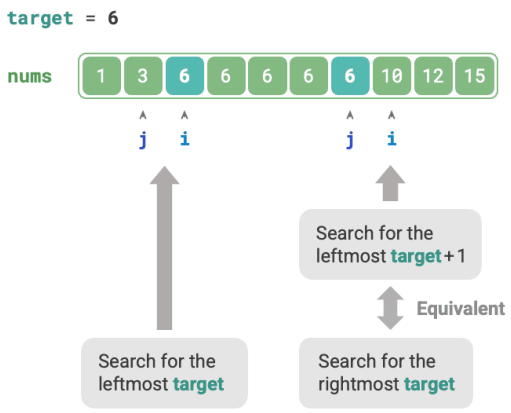


Figure 10-7 Converting right boundary search to left boundary search

Note that the returned insertion point is i, so we need to subtract 1 from it to obtain j:

```
// ═══ File: binary_search_edge.js ═══

/* Binary search for the rightmost target */
function binarySearchRightEdge(nums, target) {
    // Convert to finding the leftmost target + 1
    const i = binarySearchInsertion(nums, target + 1);
    // j points to the rightmost target, i points to the first element greater than target
    const j = i - 1;
    // Target not found, return -1
    if (j === -1 || nums[j] !== target) {
        return -1;
    }
    // Found target, return index j
    return j;
}
```

2. Converting to Element Search

We know that when the array does not contain `target`, i and j will eventually point to the first elements greater than and less than `target`, respectively.

Therefore, as shown in Figure 10-8, we can construct an element that does not exist in the array to find the left and right boundaries.

- Finding the leftmost `target`: Can be converted to finding `target - 0.5` and returning pointer i.
- Finding the rightmost `target`: Can be converted to finding `target + 0.5` and returning pointer j.

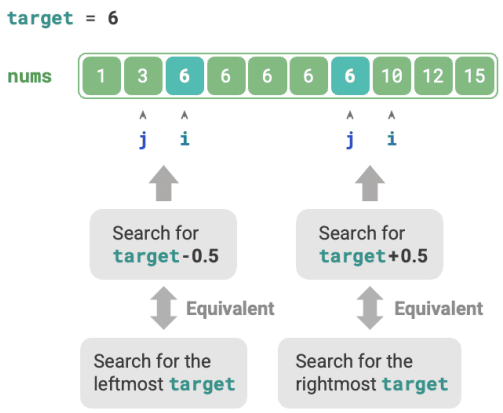


Figure 10-8 Converting boundary search to element search

The code is omitted here, but the following two points are worth noting:

- Since the given array does not contain decimals, we don't need to worry about how to handle equal cases.
- Because this method introduces decimals, the variable `target` in the function needs to be changed to a floating-point type (Python does not require this change).

10.4 Hash Optimization Strategy

In algorithm problems, **we often reduce the time complexity of algorithms by replacing linear search with hash-based search**. Let's use an algorithm problem to deepen our understanding.

> **Question**
> Given an integer array `nums` and a target element `target`, search for two elements in the array whose "sum" equals `target`, and return their array indices. Any solution will do.

10.4.1 Linear Search: Trading Time for Space

Consider directly traversing all possible combinations. As shown in Figure 10-9, we open a two-layer loop and judge in each round whether the sum of two integers equals `target`. If so, return their indices.

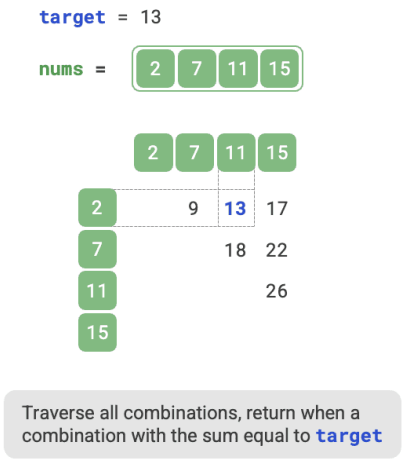


Figure 10-9 Linear search solution for two sum

The code is shown below:

```
// ═══ File: two_sum.js ═══

/* Method 1: Brute force enumeration */
function twoSumBruteForce(nums, target) {
    const n = nums.length;
    // Two nested loops, time complexity is O(n^2)
    for (let i = 0; i < n; i++) {
        for (let j = i + 1; j < n; j++) {
            if (nums[i] + nums[j] === target) {
                return [i, j];
            }
        }
    }
    return [];
}
```

This method has a time complexity of $O(n^2)$ and a space complexity of $O(1)$, which is very time-consuming with large data volumes.

10.4.2 Hash-Based Search: Trading Space for Time

Consider using a hash table where key-value pairs are array elements and element indices respectively. Loop through the array, performing the steps shown in Figure 10-10 in each round:

1. Check if the number `target - nums[i]` is in the hash table. If so, directly return the indices of these two elements.
2. Add the key-value pair `nums[i]` and index `i` to the hash table.

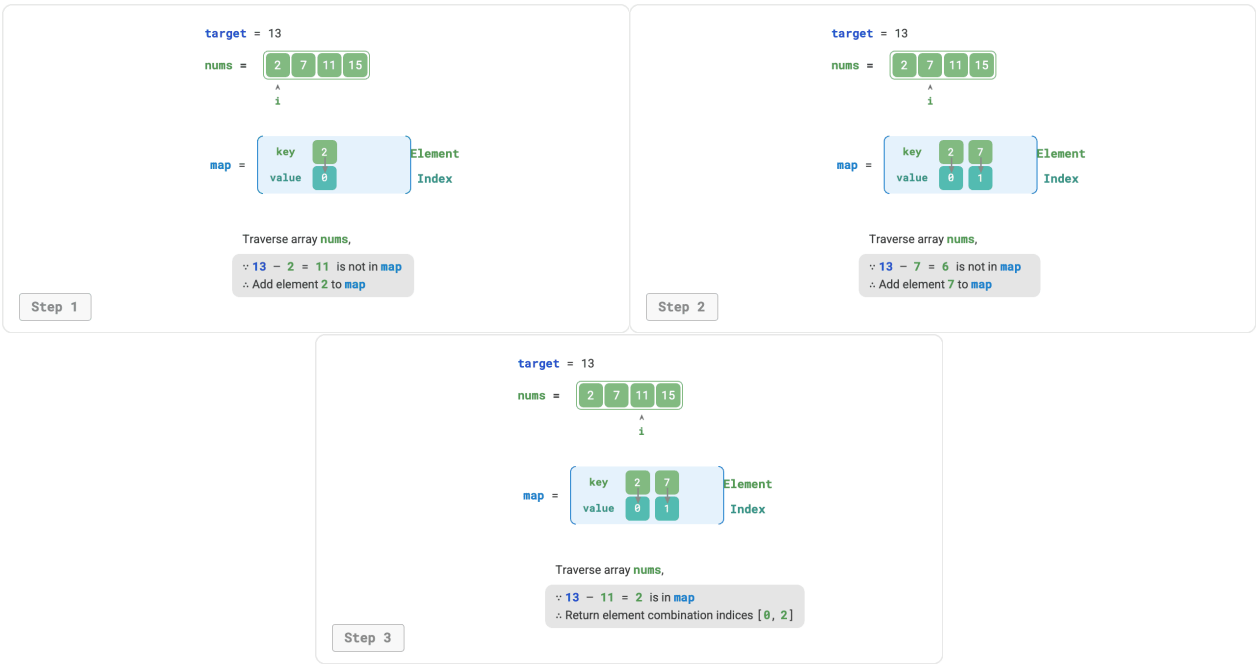


Figure 10-10 Hash table solution for two sum

The implementation code is shown below, requiring only a single loop:

```
// ═══ File: two_sum.js ═══

/* Method 2: Auxiliary hash table */
function twoSumHashTable(nums, target) {
    // Auxiliary hash table, space complexity is O(n)
    let m = {};
    // Single loop, time complexity is O(n)
    for (let i = 0; i < nums.length; i++) {
        if (m[target - nums[i]] !== undefined) {
            return [m[target - nums[i]], i];
        } else {
            m[nums[i]] = i;
        }
    }
    return [];
}
```

This method reduces the time complexity from $O(n^2)$ to $O(n)$ through hash-based search, greatly improving runtime efficiency.

Since an additional hash table needs to be maintained, the space complexity is $O(n)$. **Nevertheless, this method achieves a more balanced overall time-space efficiency, making it the optimal solution for this problem**.

10.5 Searching Algorithms Revisited

Searching algorithms are used to search for one or a group of elements that meet specific conditions in data structures (such as arrays, linked lists, trees, or graphs).

Searching algorithms can be divided into the following two categories based on their implementation approach:

- **Locating target elements by traversing the data structure**, such as traversing arrays, linked lists, trees, and graphs.
- **Achieving efficient element search by utilizing data organization structure or prior information contained in the data**, such as binary search, hash-based search, and binary search tree search.

It's not hard to see that these topics have all been covered in previous chapters, so searching algorithms are not unfamiliar to us. In this section, we will approach from a more systematic perspective and re-examine searching algorithms.

10.5.1 Brute-Force Search

Brute-force search locates target elements by traversing each element of the data structure.

- "Linear search" is applicable to linear data structures such as arrays and linked lists. It starts from one end of the data structure and accesses elements one by one until the target element is found or the other end is reached without finding the target element.
- "Breadth-first search" and "depth-first search" are two traversal strategies for graphs and trees. Breadth-first search starts from the initial node and searches layer by layer, visiting nodes from near to far. Depth-first search starts from the initial node, follows a path to the end, then backtracks and tries other paths until the entire data structure is traversed.

The advantage of brute-force search is that it is simple and has good generality, **requiring no data preprocessing or additional data structures**.

However, **the time complexity of such algorithms is** $O(n)$, where n is the number of elements, so performance is poor when dealing with large amounts of data.

10.5.2 Adaptive Search

Adaptive search utilizes the unique properties of data (such as orderliness) to optimize the search process, thereby locating target elements more efficiently.

- "Binary search" uses the orderliness of data to achieve efficient searching, applicable only to arrays.
- "Hash-based search" uses hash tables to establish key-value pair mappings between search data and target data, thereby achieving query operations.
- "Tree search" in specific tree structures (such as binary search trees), quickly eliminates nodes based on comparing node values to locate target elements.

The advantage of such algorithms is high efficiency, **with time complexity reaching $O(\log n)$ or even $O(1)$**.

However, **using these algorithms often requires data preprocessing**. For example, binary search requires pre-sorting the array, while hash-based search and tree search both require additional data structures, and maintaining these data structures also requires extra time and space overhead.

> **Tip**
> Adaptive search algorithms are often called lookup algorithms, **mainly used to quickly retrieve target elements in specific data structures**.

10.5.3 Search Method Selection

Given a dataset of size n, we can use linear search, binary search, tree search, hash-based search, and other methods to search for the target element. The working principles of each method are shown in Figure 10-11.

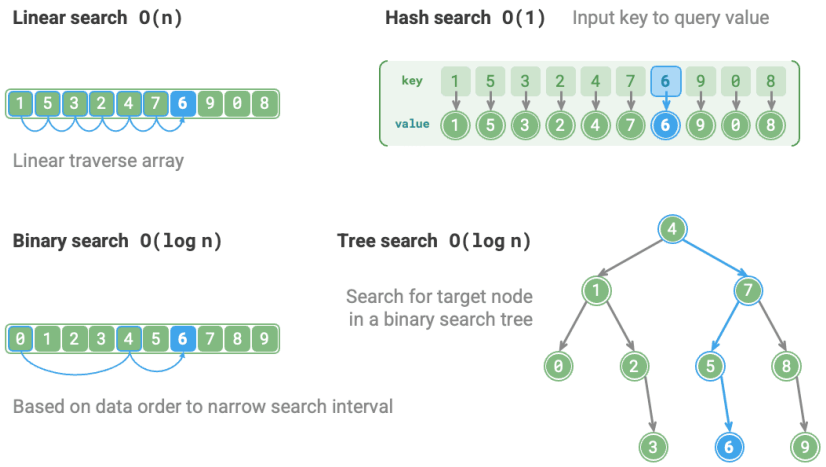


Figure 10-11 Multiple search strategies

The operational efficiency and characteristics of the above methods are as follows:

Table 10-1 Comparison of search algorithm efficiency

	Linear search	Binary search	Tree search	Hash-based search
Search element	$O(n)$	$O(\log n)$	$O(\log n)$	$O(1)$
Insert element	$O(1)$	$O(n)$	$O(\log n)$	$O(1)$

	Linear search	Binary search	Tree search	Hash-based search
Delete element	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$
Extra space	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Data preprocessing	/	Sorting $O(n \log n)$	Tree building $O(n \log n)$	Hash table building $O(n)$
Data ordered	Unordered	Ordered	Ordered	Unordered

The choice of search algorithm also depends on data volume, search performance requirements, data query and update frequency, etc.

Linear search

- Good generality, requiring no data preprocessing operations. If we only need to query the data once, the data preprocessing time for the other three methods would be longer than linear search.
- Suitable for small data volumes, where time complexity has less impact on efficiency.
- Suitable for scenarios with high data update frequency, as this method does not require any additional data maintenance.

Binary search

- Suitable for large data volumes with stable efficiency performance, worst-case time complexity of $O(\log n)$.
- Data volume cannot be too large, as storing arrays requires contiguous memory space.
- Not suitable for scenarios with frequent data insertion and deletion, as maintaining a sorted array has high overhead.

Hash-based search

- Suitable for scenarios with high query performance requirements, with an average time complexity of $O(1)$.
- Not suitable for scenarios requiring ordered data or range searches, as hash tables cannot maintain data orderliness.
- High dependence on hash functions and hash collision handling strategies, with significant risk of performance degradation.
- Not suitable for excessively large data volumes, as hash tables require extra space to minimize collisions and thus provide good query performance.

Tree search

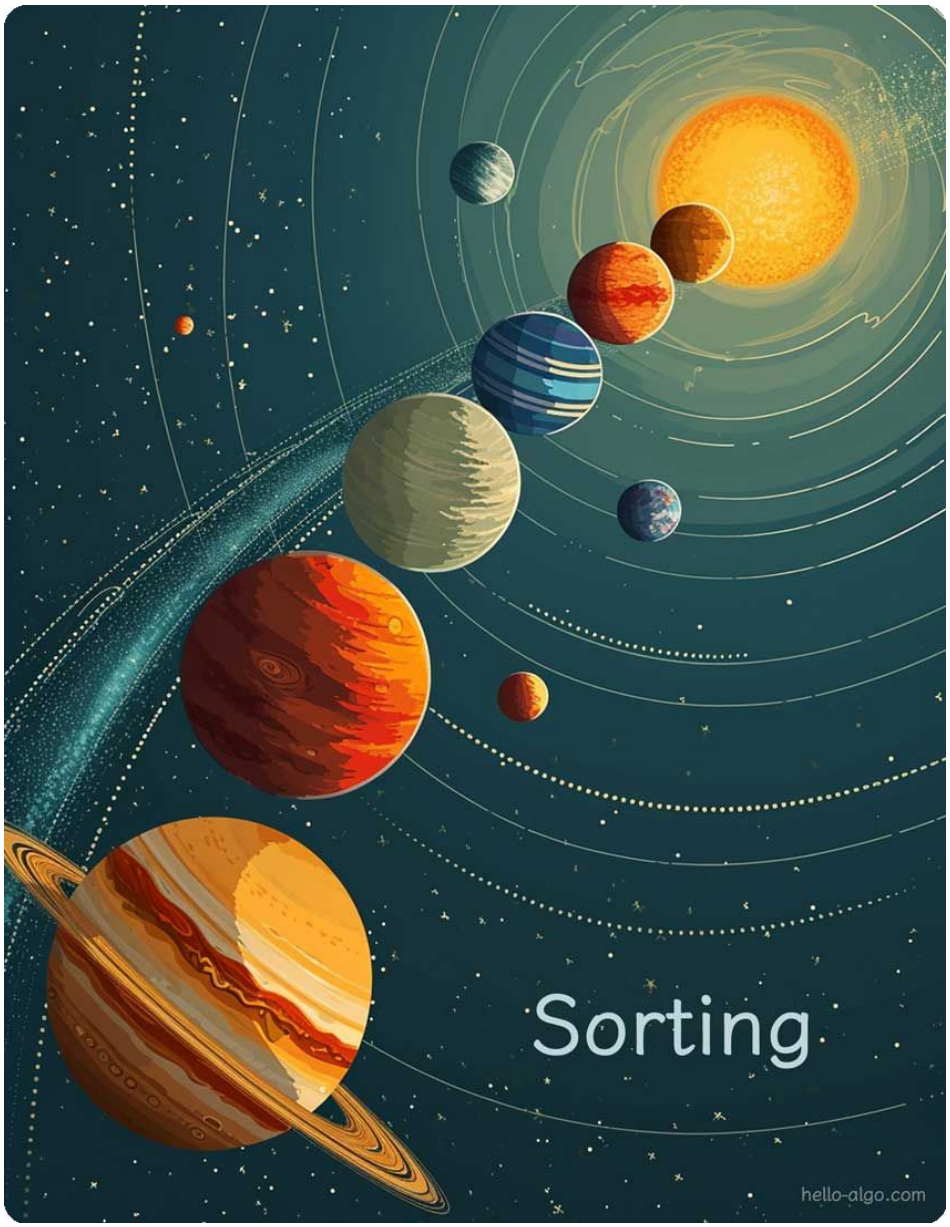
- Suitable for massive data, as tree nodes are stored dispersedly in memory.
- Suitable for scenarios requiring maintained ordered data or range searches.
- During continuous node insertion and deletion, binary search trees may become skewed, degrading time complexity to $O(n)$.
- If using AVL trees or red-black trees, all operations can run stably at $O(\log n)$ efficiency, but operations to maintain tree balance add extra overhead.

10.6 Summary

1. Key Review

- Binary search relies on data orderliness and progressively reduces the search interval by half through loops. It requires input data to be sorted and is only applicable to arrays or data structures based on array implementations.
- Brute-force search locates data by traversing the data structure. Linear search is applicable to arrays and linked lists, while breadth-first search and depth-first search are applicable to graphs and trees. Such algorithms have good generality and require no data preprocessing, but have a relatively high time complexity of $O(n)$.
- Hash-based search, tree search, and binary search are efficient search methods that can quickly locate target elements in specific data structures. Such algorithms are highly efficient with time complexity reaching $O(\log n)$ or even $O(1)$, but typically require additional data structures.
- In practice, we need to analyze factors such as data scale, search performance requirements, and data query and update frequency to choose the appropriate search method.
- Linear search is suitable for small-scale or frequently updated data; binary search is suitable for large-scale, sorted data; hash-based search is suitable for data with high query efficiency requirements and no need for range queries; tree search is suitable for large-scale dynamic data that needs to maintain order and support range queries.
- Replacing linear search with hash-based search is a commonly used strategy to optimize runtime, reducing time complexity from $O(n)$ to $O(1)$.

Chapter 11. Sorting



Abstract

Sorting is like a magic key that transforms chaos into order, enabling us to understand and process data more efficiently.

Whether it's simple ascending order or complex categorized arrangements, sorting demonstrates the harmonious beauty of data.

11.1 Sorting Algorithm

Sorting algorithm (sorting algorithm) is used to arrange a group of data in a specific order. Sorting algorithms have extensive applications because ordered data can usually be searched, analyzed, and processed more efficiently.

As shown in Figure 11-1, data types in sorting algorithms can be integers, floating-point numbers, characters, or strings, etc. The sorting criterion can be set according to requirements, such as numerical size, character ASCII code order, or custom rules.

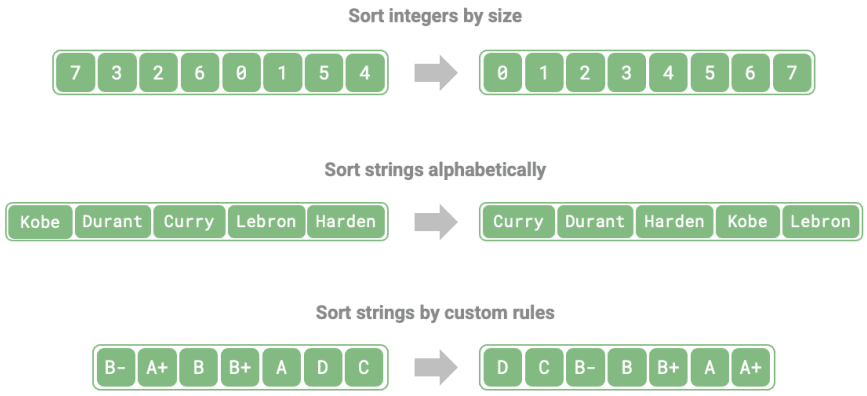


Figure 11-1 Data type and criterion examples

11.1.1 Evaluation Dimensions

Execution efficiency: We expect the time complexity of sorting algorithms to be as low as possible, with a smaller total number of operations (reducing the constant factor in time complexity). For large data volumes, execution efficiency is particularly important.

In-place property: As the name implies, in-place sorting achieves sorting by operating directly on the original array without requiring additional auxiliary arrays, thus saving memory. Typically, in-place sorting involves fewer data movement operations and runs faster.

Stability: Stable sorting ensures that the relative order of equal elements in the array does not change after sorting is completed.

Stable sorting is a necessary condition for multi-level sorting scenarios. Suppose we have a table storing student information, where column 1 and column 2 are name and age, respectively. In this case, unstable sorting may cause the ordered nature of the input data to be lost:

```
# Input Data Is Sorted by Name
# (name, age)
  ('A', 19)
  ('B', 18)
  ('C', 21)
```

```
  ('D', 19)
  ('E', 23)

# Assuming We Use an Unstable Sorting Algorithm to Sort the List by Age,
# In the Result, the Relative Positions of ('D', 19) and ('A', 19) Are Changed,
# And the Property That the Input Data Is Sorted by Name Is Lost
  ('B', 18)
  ('D', 19)
  ('A', 19)
  ('C', 21)
  ('E', 23)
```

Adaptability: Adaptive sorting can utilize the existing order information in the input data to reduce the amount of computation, achieving better time efficiency. The best-case time complexity of adaptive sorting algorithms is typically better than the average time complexity.

Comparison-based or not: Comparison-based sorting relies on comparison operators ($<$, $=$, $>$) to determine the relative order of elements, thereby sorting the entire array, with a theoretical optimal time complexity of $O(n \log n)$. Non-comparison sorting does not use comparison operators and can achieve a time complexity of $O(n)$, but its versatility is relatively limited.

11.1.2 Ideal Sorting Algorithm

Fast execution, in-place, stable, adaptive, good versatility. Clearly, no sorting algorithm has been discovered to date that combines all of these characteristics. Therefore, when selecting a sorting algorithm, it is necessary to decide based on the specific characteristics of the data and the requirements of the problem.

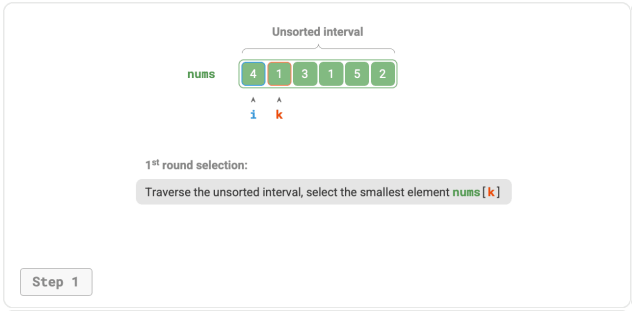
Next, we will learn about various sorting algorithms together and analyze the advantages and disadvantages of each sorting algorithm based on the above evaluation dimensions.

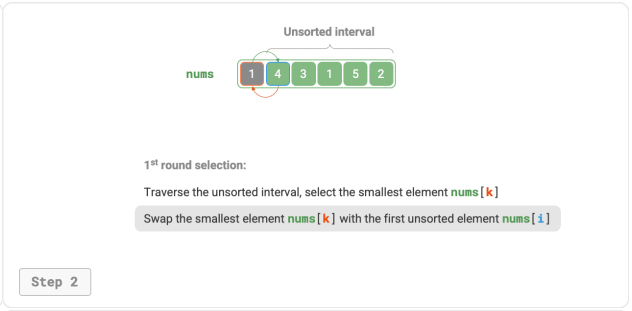
11.2 Selection Sort

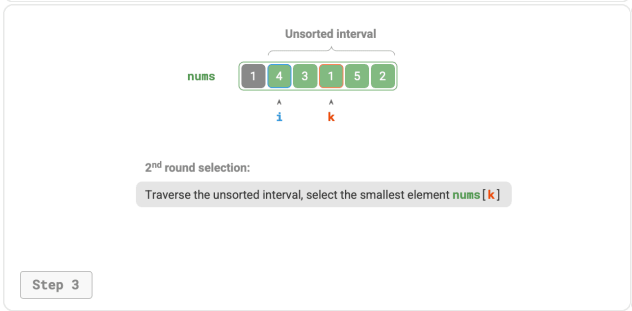
Selection sort (selection sort) works very simply: it opens a loop, and in each round, selects the smallest element from the unsorted interval and places it at the end of the sorted interval.

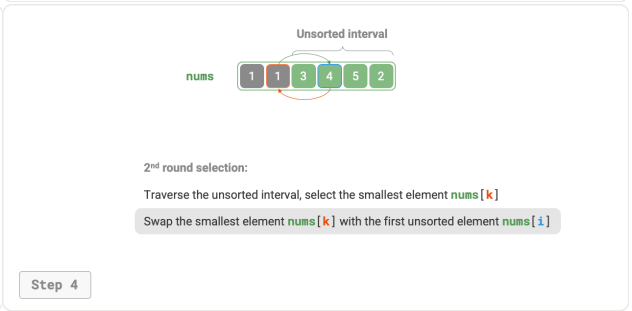
Assume the array has length n. The algorithm flow of selection sort is shown in Figure 11-2.

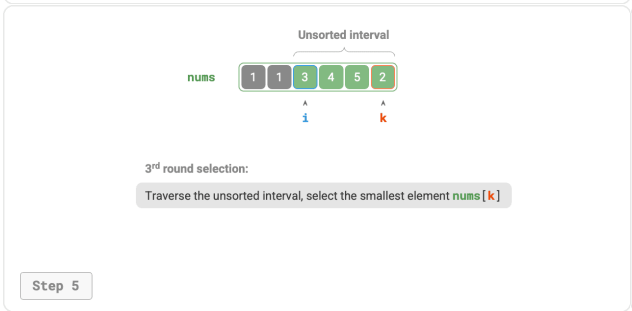
1. Initially, all elements are unsorted, i.e., the unsorted (index) interval is $[0, n-1]$.
2. Select the smallest element in the interval $[0, n-1]$ and swap it with the element at index 0. After completion, the first element of the array is sorted.
3. Select the smallest element in the interval $[1, n-1]$ and swap it with the element at index 1. After completion, the first 2 elements of the array are sorted.
4. And so on. After $n-1$ rounds of selection and swapping, the first $n-1$ elements of the array are sorted.
5. The only remaining element must be the largest element, requiring no sorting, so the array sorting is complete.

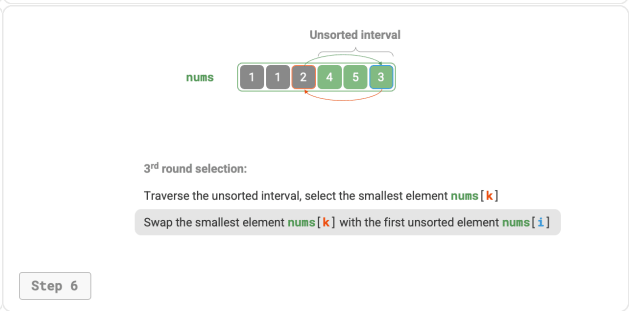

Unsorted interval
nums
4 1 3 1 5 2
i k
1st round selection:
Traverse the unsorted interval, select the smallest element nums[k]
Step 1

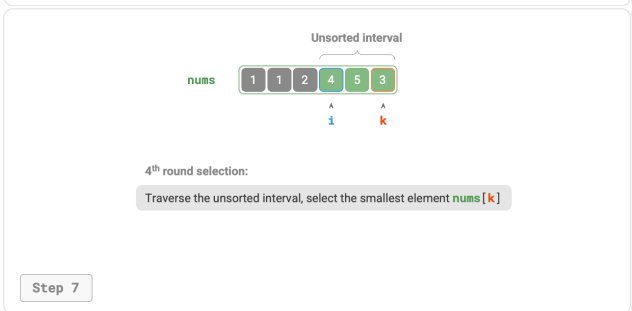

Unsorted interval
nums
1 4 3 1 5 2
1st round selection:
Traverse the unsorted interval, select the smallest element nums[k]
Swap the smallest element nums[k] with the first unsorted element nums[i]
Step 2

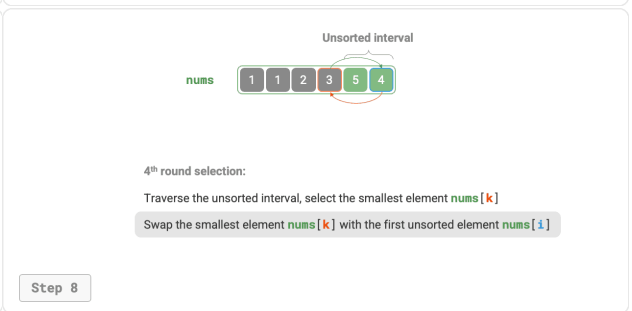

Unsorted interval
nums
1 4 3 1 5 2
i k
2nd round selection:
Traverse the unsorted interval, select the smallest element nums[k]
Step 3

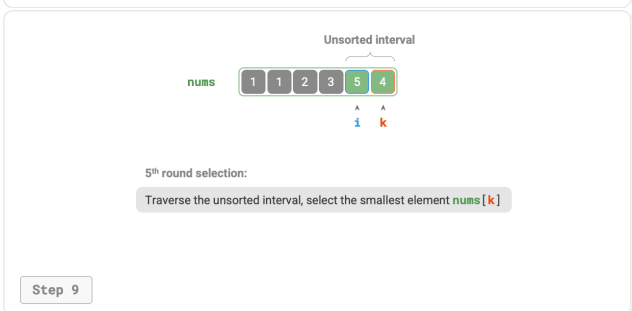

Unsorted interval
nums
1 1 3 4 5 2
2nd round selection:
Traverse the unsorted interval, select the smallest element nums[k]
Swap the smallest element nums[k] with the first unsorted element nums[i]
Step 4

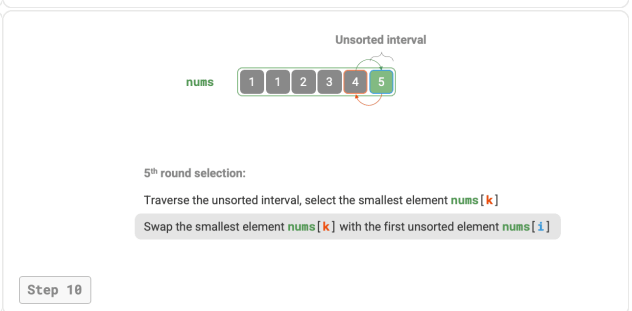

Unsorted interval
nums
1 1 3 4 5 2
i k
3rd round selection:
Traverse the unsorted interval, select the smallest element nums[k]
Step 5


Unsorted interval
nums
1 1 2 4 5 3
3rd round selection:
Traverse the unsorted interval, select the smallest element nums[k]
Swap the smallest element nums[k] with the first unsorted element nums[i]
Step 6


Unsorted interval
nums
1 1 2 4 5 3
i k
4th round selection:
Traverse the unsorted interval, select the smallest element nums[k]
Step 7


Unsorted interval
nums
1 1 2 3 5 4
4th round selection:
Traverse the unsorted interval, select the smallest element nums[k]
Swap the smallest element nums[k] with the first unsorted element nums[i]
Step 8


Unsorted interval
nums
1 1 2 3 5 4
i k
5th round selection:
Traverse the unsorted interval, select the smallest element nums[k]
Step 9


Unsorted interval
nums
1 1 2 3 4 5
5th round selection:
Traverse the unsorted interval, select the smallest element nums[k]
Swap the smallest element nums[k] with the first unsorted element nums[i]
Step 10

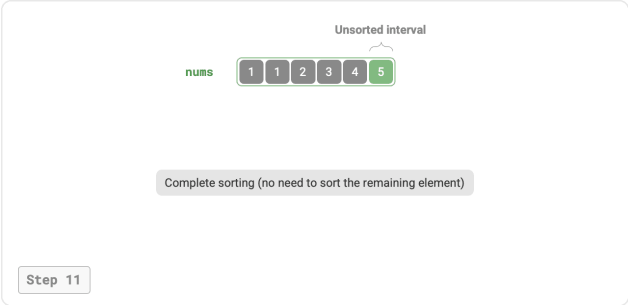


Figure 11-2 Selection sort steps

In the code, we use k to record the smallest element within the unsorted interval:

```
// ═══ File: selection_sort.js ═══

/* Selection sort */
function selectionSort(nums) {
    let n = nums.length;
    // Outer loop: unsorted interval is [i, n-1]
    for (let i = 0; i < n - 1; i++) {
        // Inner loop: find the smallest element within the unsorted interval
        let k = i;
        for (let j = i + 1; j < n; j++) {
            if (nums[j] < nums[k]) {
                k = j; // Record the index of the smallest element
            }
        }
        // Swap the smallest element with the first element of the unsorted interval
        [nums[i], nums[k]] = [nums[k], nums[i]];
    }
}
```

11.2.1 Algorithm Characteristics

- **Time complexity of $O(n^2)$, non-adaptive sorting**: The outer loop has $n - 1$ rounds in total. The length of the unsorted interval in the first round is n, and the length of the unsorted interval in the last round is 2. That is, each round of the outer loop contains n, $n - 1$, ..., 3, 2 inner loop iterations, summing to $\frac{(n-1)(n+2)}{2}$.
- **Space complexity of $O(1)$, in-place sorting**: Pointers i and j use a constant amount of extra space.
- **Non-stable sorting**: As shown in Figure 11-3, element `nums[i]` may be swapped to the right of an element equal to it, causing a change in their relative order.

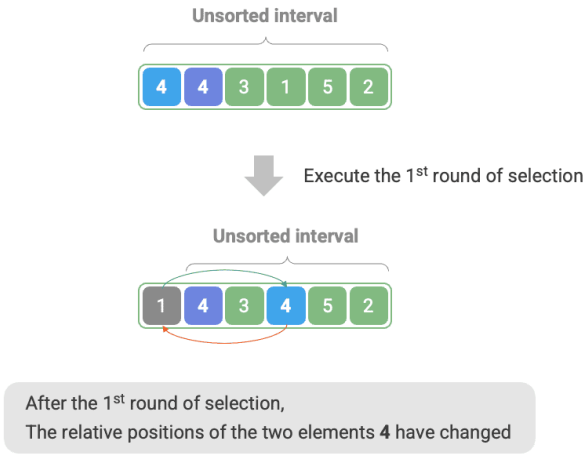
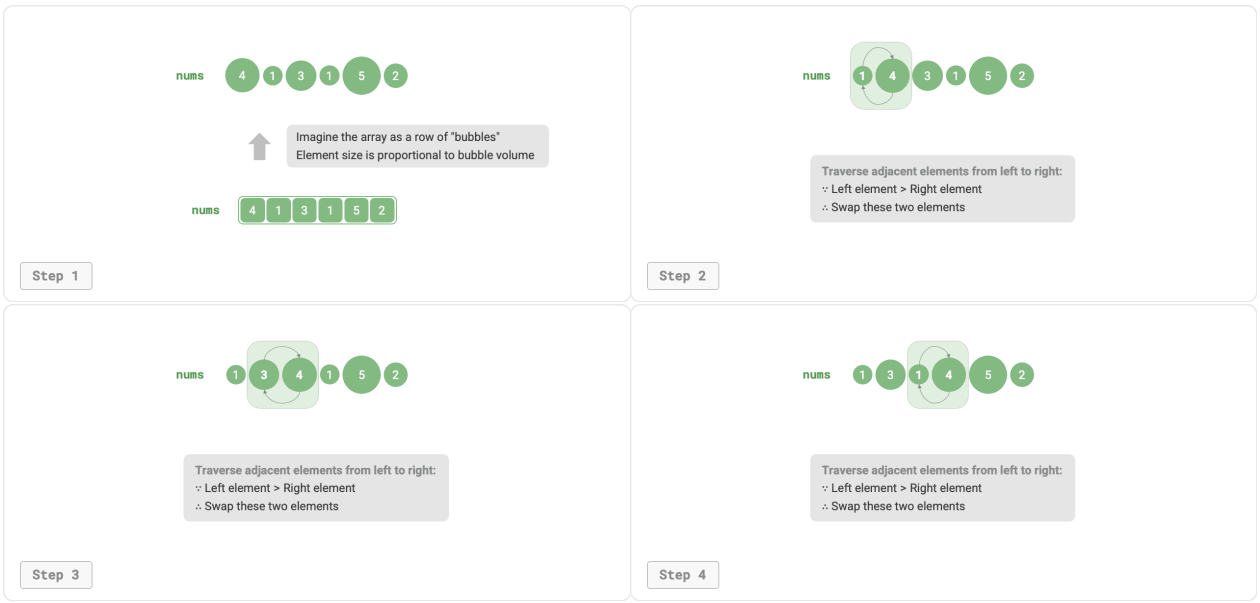


Figure 11-3 Selection sort non-stability example

11.3 Bubble Sort

Bubble sort (bubble sort) achieves sorting by continuously comparing and swapping adjacent elements. This process is like bubbles rising from the bottom to the top, hence the name bubble sort.

As shown in Figure 11-4, the bubbling process can be simulated using element swap operations: starting from the leftmost end of the array and traversing to the right, compare the size of adjacent elements, and if "left element > right element", swap them. After completing the traversal, the largest element will be moved to the rightmost end of the array.



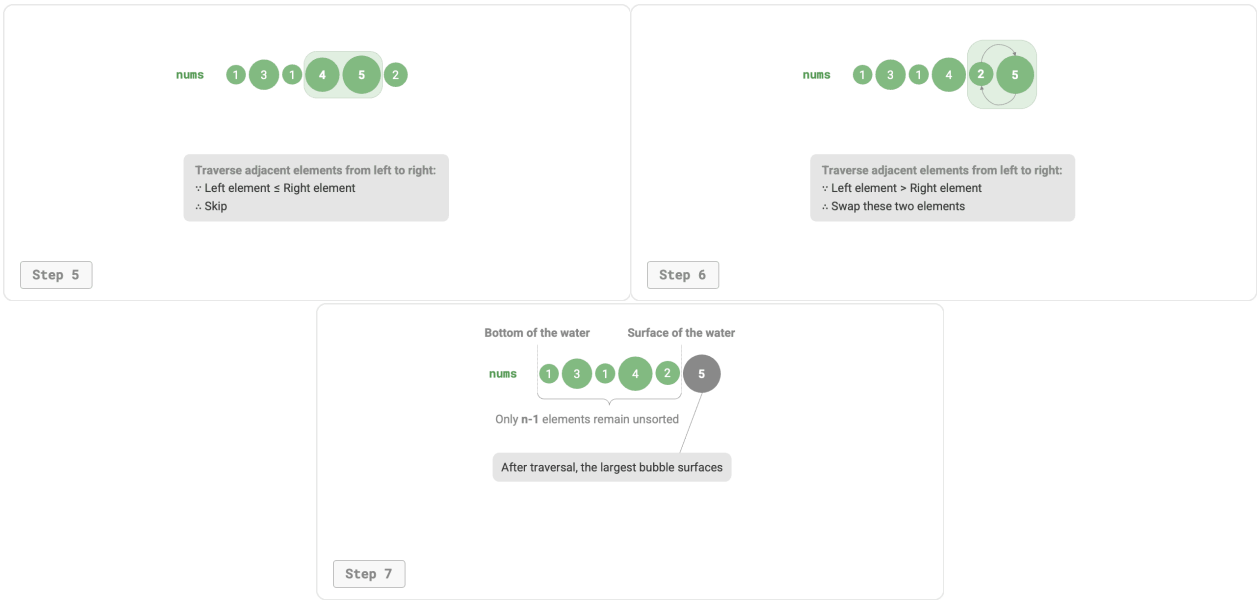


Figure 11-4 Simulating bubble using element swap operation

11.3.1 Algorithm Flow

Assume the array has length n. The steps of bubble sort are shown in Figure 11-5.

1. First, perform "bubbling" on n elements, **swapping the largest element of the array to its correct position**.
2. Next, perform "bubbling" on the remaining $n - 1$ elements, **swapping the second largest element to its correct position**.
3. And so on. After $n - 1$ rounds of "bubbling", **the largest $n - 1$ elements have all been swapped to their correct positions**.
4. The only remaining element must be the smallest element, requiring no sorting, so the array sorting is complete.

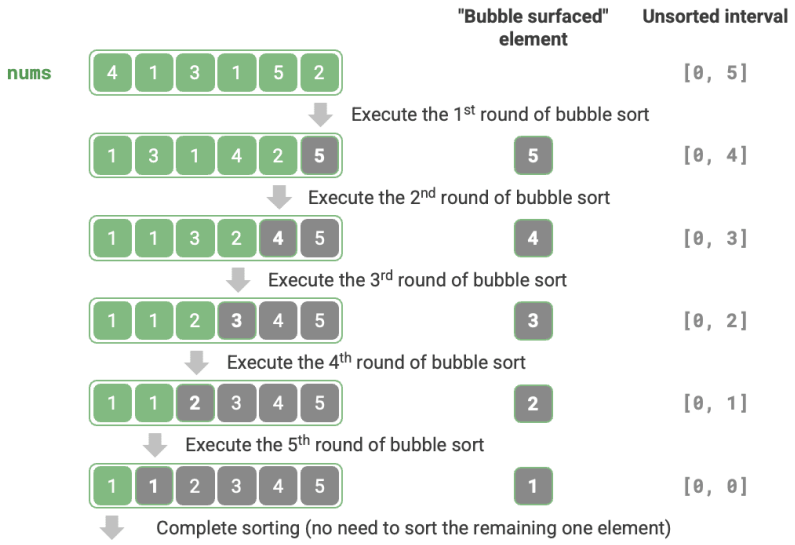


Figure 11-5 Bubble sort flow

Example code is as follows:

```
// == File: bubble_sort.js ==

/* Bubble sort */
function bubbleSort(nums) {
    // Outer loop: unsorted range is [0, i]
    for (let i = nums.length - 1; i > 0; i--) {
        // Inner loop: swap the largest element in the unsorted range [0, i] to the rightmost end
        ↪ of that range
        for (let j = 0; j < i; j++) {
            if (nums[j] > nums[j + 1]) {
                // Swap nums[j] and nums[j + 1]
                let tmp = nums[j];
                nums[j] = nums[j + 1];
                nums[j + 1] = tmp;
            }
        }
    }
}
```

11.3.2 Efficiency Optimization

We notice that if no swap operations are performed during a certain round of "bubbling", it means the array has already completed sorting and can directly return the result. Therefore, we can add a flag `flag` to monitor this situation and return immediately once it occurs.

After optimization, the worst-case time complexity and average time complexity of bubble sort remain $O(n^2)$; but when the input array is completely ordered, the best-case time complexity can reach $O(n)$.

```
// ═══ File: bubble_sort.js ═══

/* Bubble sort (flag optimization) */
function bubbleSortWithFlag(nums) {
    // Outer loop: unsorted range is [0, i]
    for (let i = nums.length - 1; i > 0; i--) {
        let flag = false; // Initialize flag
        // Inner loop: swap the largest element in the unsorted range [0, i] to the rightmost end
        ↪ of that range
        for (let j = 0; j < i; j++) {
            if (nums[j] > nums[j + 1]) {
                // Swap nums[j] and nums[j + 1]
                let tmp = nums[j];
                nums[j] = nums[j + 1];
                nums[j + 1] = tmp;
                flag = true; // Record element swap
            }
        }
        if (!flag) break; // No elements were swapped in this round of "bubbling", exit directly
    }
}
```

11.3.3 Algorithm Characteristics

- **Time complexity of $O(n^2)$, adaptive sorting**: The array lengths traversed in each round of "bubbling" are $n-1, n-2, \ldots, 2, 1$, totaling $(n-1)n/2$. After introducing the `flag` optimization, the best-case time complexity can reach $O(n)$.
- **Space complexity of $O(1)$, in-place sorting**: Pointers i and j use a constant amount of extra space.
- **Stable sorting**: Since equal elements are not swapped during "bubbling".

11.4 Insertion Sort

Insertion sort (insertion sort) is a simple sorting algorithm that works very similarly to the process of manually organizing a deck of cards.

Specifically, we select a base element from the unsorted interval, compare the element with elements in the sorted interval to its left one by one, and insert the element into the correct position.

Figure 11-6 shows the operation flow of inserting an element into the array. Let the base element be `base`. We need to move all elements from the target index to `base` one position to the right, and then assign `base` to the target index.

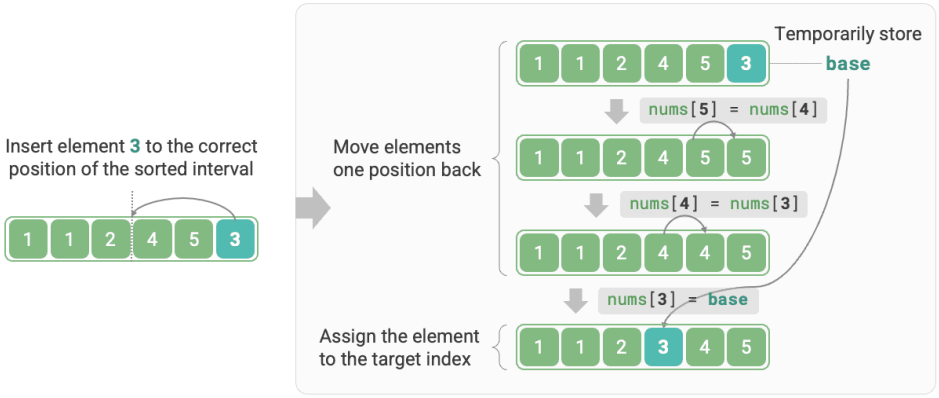


Figure 11-6 Single insertion operation

11.4.1 Algorithm Flow

The overall flow of insertion sort is shown in Figure 11-7.

1. Initially, the first element of the array has completed sorting.
2. Select the second element of the array as `base`, and after inserting it into the correct position, **the first 2 elements of the array are sorted**.
3. Select the third element as `base`, and after inserting it into the correct position, **the first 3 elements of the array are sorted**.
4. And so on. In the last round, select the last element as `base`, and after inserting it into the correct position, **all elements are sorted**.

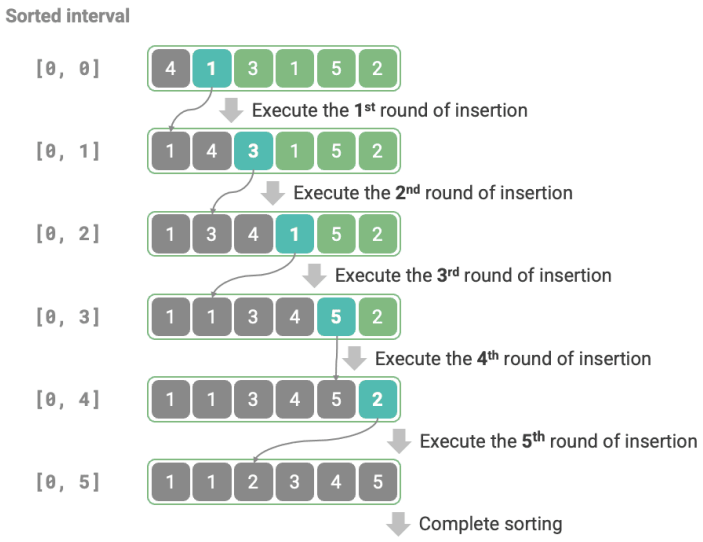


Figure 11-7 Insertion sort flow

Example code is as follows:

```
// ═══ File: insertion_sort.js ═══

/* Insertion sort */
function insertionSort(nums) {
    // Outer loop: sorted interval is [0, i-1]
    for (let i = 1; i < nums.length; i++) {
        let base = nums[i],
            j = i - 1;
        // Inner loop: insert base into the correct position within the sorted interval [0, i-1]
        while (j >= 0 && nums[j] > base) {
            nums[j + 1] = nums[j]; // Move nums[j] to the right by one position
            j--;
        }
        nums[j + 1] = base; // Assign base to the correct position
    }
}
```

11.4.2 Algorithm Characteristics

- **Time complexity of $O(n^2)$, adaptive sorting**: In the worst case, each insertion operation requires loops of $n-1$, $n-2$, ..., 2, 1, summing to $(n-1)n/2$, so the time complexity is $O(n^2)$. When encountering ordered data, the insertion operation will terminate early. When the input array is completely ordered, insertion sort achieves the best-case time complexity of $O(n)$.
- **Space complexity of $O(1)$, in-place sorting**: Pointers i and j use a constant amount of extra space.
- **Stable sorting**: During the insertion operation process, we insert elements to the right of equal elements, without changing their order.

11.4.3 Advantages of Insertion Sort

The time complexity of insertion sort is $O(n^2)$, while the time complexity of quick sort, which we will learn about next, is $O(n \log n)$. Although insertion sort has a higher time complexity, **insertion sort is usually faster for smaller data volumes**.

This conclusion is similar to the applicable situations of linear search and binary search. Algorithms like quick sort with $O(n \log n)$ complexity are sorting algorithms based on divide-and-conquer strategy and often contain more unit computation operations. When the data volume is small, n^2 and $n \log n$ are numerically close, and complexity does not dominate; the number of unit operations per round plays a decisive role.

In fact, the built-in sorting functions in many programming languages (such as Java) adopt insertion sort. The general approach is: for long arrays, use sorting algorithms based on divide-and-conquer strategy, such as quick sort; for short arrays, directly use insertion sort.

Although bubble sort, selection sort, and insertion sort all have a time complexity of $O(n^2)$, in actual situations, **insertion sort is used significantly more frequently than bubble sort and selection sort**, mainly for the following reasons.

- Bubble sort is based on element swapping, requiring the use of a temporary variable, involving 3 unit operations; insertion sort is based on element assignment, requiring only 1 unit operation. Therefore, **the computational overhead of bubble sort is usually higher than that of insertion sort**.
- Selection sort has a time complexity of $O(n^2)$ in any case. **If given a set of partially ordered data, insertion sort is usually more efficient than selection sort**.
- Selection sort is unstable and cannot be applied to multi-level sorting.

11.5 Quick Sort

Quick sort (quick sort) is a sorting algorithm based on the divide-and-conquer strategy, which operates efficiently and is widely applied.

The core operation of quick sort is "sentinel partitioning", which aims to: select a certain element in the array as the "pivot", move all elements smaller than the pivot to its left, and move elements larger than the pivot to its right. Specifically, the flow of sentinel partitioning is shown in Figure 11-8.

1. Select the leftmost element of the array as the pivot, and initialize two pointers `i` and `j` pointing to the two ends of the array.
2. Set up a loop in which `i` (`j`) is used in each round to find the first element larger (smaller) than the pivot, and then swap these two elements.
3. Loop through step `2.` until `i` and `j` meet, and finally swap the pivot to the boundary line of the two sub-arrays.

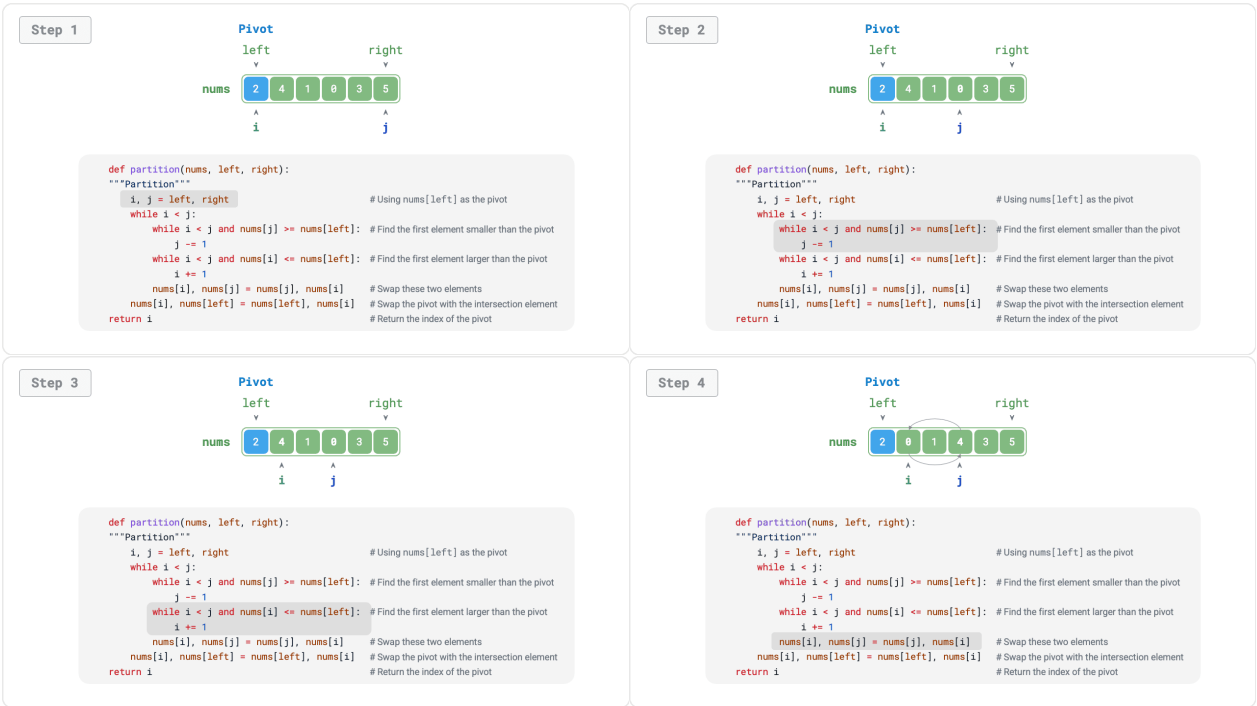


Figure 11-8 Sentinel partitioning steps

After sentinel partitioning is complete, the original array is divided into three parts: left sub-array, pivot, right sub-array, satisfying "any element in left sub-array $\leq$ pivot $\leq$ any element in right sub-array". Therefore, we next only need to sort these two sub-arrays.

> **Divide-and-conquer strategy of quick sort**
>
> The essence of sentinel partitioning is to simplify the sorting problem of a longer array into the sorting problems of two shorter arrays.

```
// ═══ File: quick_sort.js ═══

/* Swap elements */
swap(nums, i, j) {
    let tmp = nums[i];
    nums[i] = nums[j];
    nums[j] = tmp;
}
```

```
/* Sentinel partition */
partition(nums, left, right) {
    // Use nums[left] as the pivot
    let i = left,
        j = right;
    while (i < j) {
        while (i < j && nums[j] >= nums[left]) {
            j -= 1; // Search from right to left for the first element smaller than the pivot
        }
        while (i < j && nums[i] <= nums[left]) {
            i += 1; // Search from left to right for the first element greater than the pivot
        }
        // Swap elements
        this.swap(nums, i, j); // Swap these two elements
    }
    this.swap(nums, i, left); // Swap the pivot to the boundary between the two subarrays
    return i; // Return the index of the pivot
}
```

11.5.1 Algorithm Flow

The overall flow of quick sort is shown in Figure 11-9.

1. First, perform one "sentinel partitioning" on the original array to obtain the unsorted left sub-array and right sub-array.
2. Then, recursively perform "sentinel partitioning" on the left sub-array and right sub-array respectively.
3. Continue recursively until the sub-array length is 1, at which point sorting of the entire array is complete.

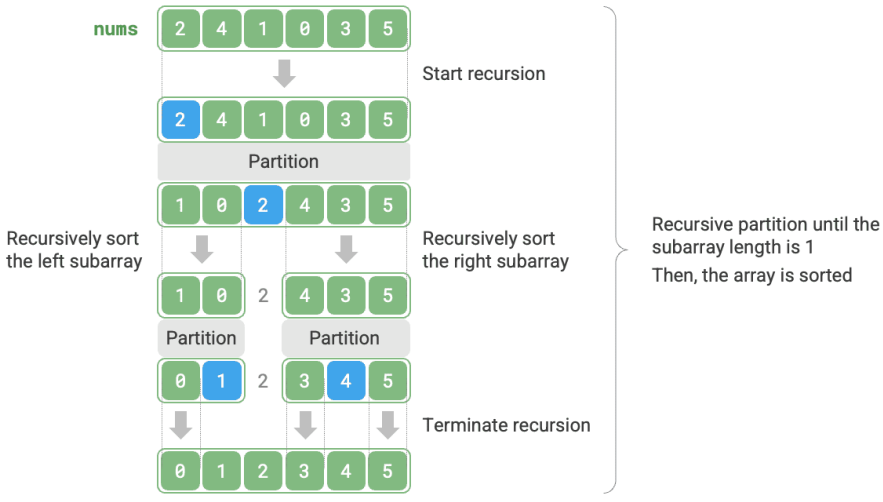


Figure 11-9 Quick sort flow

```
// === File: quick_sort.js ===

/* Quick sort */
quickSort(nums, left, right) {
    // Terminate recursion when subarray length is 1
    if (left >= right) return;
    // Sentinel partition
    const pivot = this.partition(nums, left, right);
    // Recursively process the left subarray and right subarray
    this.quickSort(nums, left, pivot - 1);
    this.quickSort(nums, pivot + 1, right);
}
```

11.5.2 Algorithm Characteristics

- **Time complexity of $O(n \log n)$, non-adaptive sorting**: In the average case, the number of recursive levels of sentinel partitioning is $\log n$, and the total number of loops at each level is n, using $O(n \log n)$ time overall. In the worst case, each round of sentinel partitioning divides an array of length n into two sub-arrays of length 0 and $n - 1$, at which point the number of recursive levels reaches n, the number of loops at each level is n, and the total time used is $O(n^2)$.
- **Space complexity of $O(n)$, in-place sorting**: In the case where the input array is completely reversed, the worst recursive depth reaches n, using $O(n)$ stack frame space. The sorting operation is performed on the original array without the aid of an additional array.
- **Non-stable sorting**: In the last step of sentinel partitioning, the pivot may be swapped to the right of equal elements.

11.5.3 Why Is Quick Sort Fast

From the name, we can see that quick sort should have certain advantages in terms of efficiency. Although the average time complexity of quick sort is the same as "merge sort" and "heap sort", quick sort is usually more efficient, mainly for the following reasons.

- **The probability of the worst case occurring is very low**: Although the worst-case time complexity of quick sort is $O(n^2)$, which is not as stable as merge sort, in the vast majority of cases, quick sort can run with a time complexity of $O(n \log n)$.
- **High cache utilization**: When performing sentinel partitioning operations, the system can load the entire sub-array into the cache, so element access efficiency is relatively high. Algorithms like "heap sort" require jump-style access to elements, thus lacking this characteristic.
- **Small constant coefficient of complexity**: Among the three algorithms mentioned above, quick sort has the smallest total number of operations such as comparisons, assignments, and swaps. This is similar to the reason why "insertion sort" is faster than "bubble sort".

11.5.4 Pivot Optimization

Quick sort may have reduced time efficiency for certain inputs. Take an extreme example: suppose the input array is completely reversed. Since we select the leftmost element as the pivot, after sentinel

partitioning is complete, the pivot is swapped to the rightmost end of the array, causing the left sub-array length to be $n-1$ and the right sub-array length to be 0. If we recurse down like this, each round of sentinel partitioning will have a sub-array length of 0, the divide-and-conquer strategy fails, and quick sort degrades to a form approximate to "bubble sort".

To avoid this situation as much as possible, **we can optimize the pivot selection strategy in sentinel partitioning**. For example, we can randomly select an element as the pivot. However, if luck is not good and we select a non-ideal pivot every time, efficiency is still not satisfactory.

It should be noted that programming languages usually generate "pseudo-random numbers". If we construct a specific test case for a pseudo-random number sequence, the efficiency of quick sort may still degrade.

For further improvement, we can select three candidate elements in the array (usually the first, last, and middle elements of the array), **and use the median of these three candidate elements as the pivot**. In this way, the probability that the pivot is "neither too small nor too large" will be greatly increased. Of course, we can also select more candidate elements to further improve the robustness of the algorithm. After adopting this method, the probability of time complexity degrading to $O(n^2)$ is greatly reduced.

Example code is as follows:

```
// == File: quick_sort.js ==

/* Select the median of three candidate elements */
medianThree(nums, left, mid, right) {
    let l = nums[left],
        m = nums[mid],
        r = nums[right];
    // m is between l and r
    if ((l <= m && m <= r) || (r <= m && m <= l)) return mid;
    // l is between m and r
    if ((m <= l && l <= r) || (r <= l && l <= m)) return left;
    return right;
}

/* Sentinel partition (median of three) */
partition(nums, left, right) {
    // Select the median of three candidate elements
    let med = this.medianThree(
        nums,
        left,
        Math.floor((left + right) / 2),
        right
    );
    // Swap the median to the array's leftmost position
    this.swap(nums, left, med);
    // Use nums[left] as the pivot
    let i = left,
        j = right;
    while (i < j) {
        while (i < j && nums[j] >= nums[left]) j--; // Search from right to left for the first
        ↪  element smaller than the pivot
        while (i < j && nums[i] <= nums[left]) i++; // Search from left to right for the first
        ↪  element greater than the pivot
        this.swap(nums, i, j); // Swap these two elements
    }
```

```
    this.swap(nums, i, left); // Swap the pivot to the boundary between the two subarrays
    return i; // Return the index of the pivot
}
```

11.5.5 Recursive Depth Optimization

For certain inputs, quick sort may occupy more space. Taking a completely ordered input array as an example, let the length of the sub-array in recursion be m. Each round of sentinel partitioning will produce a left sub-array of length 0 and a right sub-array of length $m - 1$, which means that the problem scale reduced per recursive call is very small (only one element is reduced), and the height of the recursion tree will reach $n - 1$, at which point $O(n)$ size of stack frame space is required.

To prevent the accumulation of stack frame space, we can compare the lengths of the two sub-arrays after each round of sentinel sorting is complete, **and only recurse on the shorter sub-array**. Since the length of the shorter sub-array will not exceed $n/2$, this method can ensure that the recursion depth does not exceed $\log n$, thus optimizing the worst-case space complexity to $O(\log n)$. The code is as follows:

```
// ═══ File: quick_sort.js ═══

/* Quick sort (recursion depth optimization) */
quickSort(nums, left, right) {
    // Terminate when subarray length is 1
    while (left < right) {
        // Sentinel partition operation
        let pivot = this.partition(nums, left, right);
        // Perform quick sort on the shorter of the two subarrays
        if (pivot - left < right - pivot) {
            this.quickSort(nums, left, pivot - 1); // Recursively sort the left subarray
            left = pivot + 1; // Remaining unsorted interval is [pivot + 1, right]
        } else {
            this.quickSort(nums, pivot + 1, right); // Recursively sort the right subarray
            right = pivot - 1; // Remaining unsorted interval is [left, pivot - 1]
        }
    }
}
```

11.6 Merge Sort

Merge sort (merge sort) is a sorting algorithm based on the divide-and-conquer strategy, which includes the "divide" and "merge" phases shown in Figure 11-10.

1. **Divide phase**: Recursively split the array from the midpoint, transforming the sorting problem of a long array into the sorting problems of shorter arrays.
2. **Merge phase**: When the sub-array length is 1, terminate the division and start merging, continuously merging two shorter sorted arrays into one longer sorted array until the process is complete.

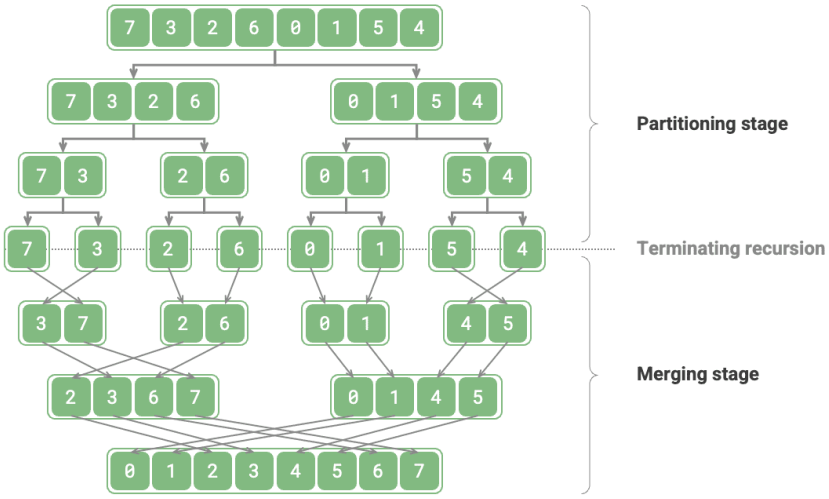


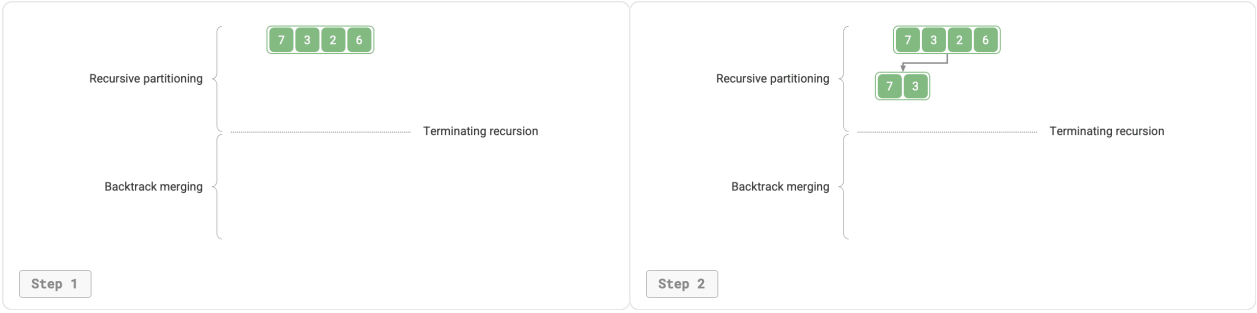
Figure 11-10 Divide and merge phases of merge sort

11.6.1 Algorithm Flow

As shown in Figure 11-11, the "divide phase" recursively splits the array from the midpoint into two sub-arrays from top to bottom.

1. Calculate the array midpoint `mid`, recursively divide the left sub-array (interval `[left, mid]`) and right sub-array (interval `[mid + 1, right]`).
2. Recursively execute step `1.` until the sub-array interval length is 1, then terminate.

The "merge phase" merges the left sub-array and right sub-array into a sorted array from bottom to top. Note that merging starts from sub-arrays of length 1, and each sub-array in the merge phase is sorted.



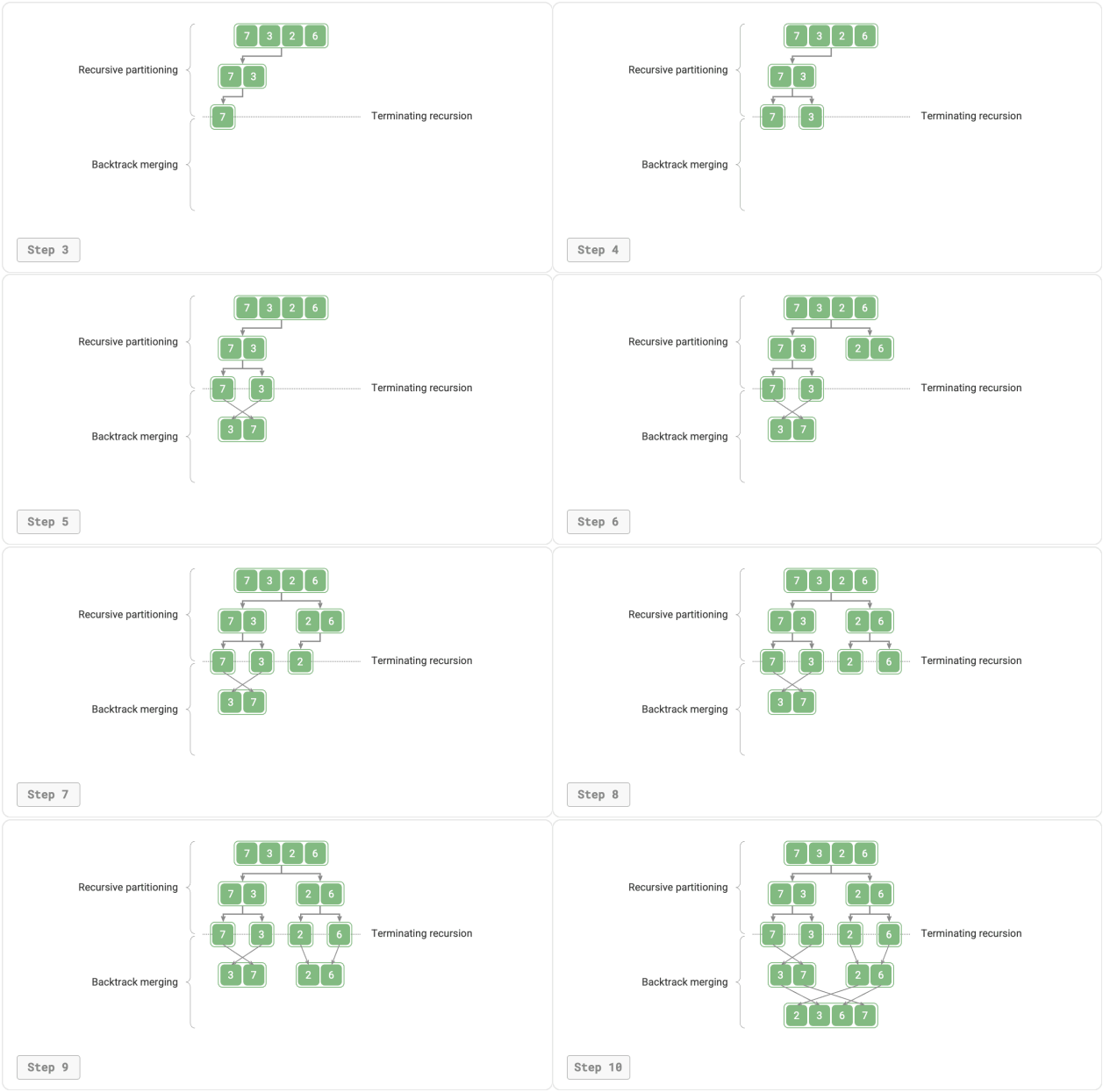


Figure 11-11 Merge sort steps

It can be observed that the recursive order of merge sort is consistent with the post-order traversal of a binary tree.

- **Post-order traversal**: First recursively traverse the left subtree, then recursively traverse the right subtree, and finally process the root node.
- **Merge sort**: First recursively process the left sub-array, then recursively process the right sub-array, and finally perform the merge.

The implementation of merge sort is shown in the code below. Note that the interval to be merged in `nums` is `[left, right]`, while the corresponding interval in `tmp` is `[0, right - left]`.

```javascript
// === File: merge_sort.js ===

/* Merge left subarray and right subarray */
function merge(nums, left, mid, right) {
    // Left subarray interval is [left, mid], right subarray interval is [mid+1, right]
    // Create a temporary array tmp to store the merged results
    const tmp = new Array(right - left + 1);
    // Initialize the start indices of the left and right subarrays
    let i = left,
        j = mid + 1,
        k = 0;
    // While both subarrays still have elements, compare and copy the smaller element into the
    ↪ temporary array
    while (i <= mid && j <= right) {
        if (nums[i] <= nums[j]) {
            tmp[k++] = nums[i++];
        } else {
            tmp[k++] = nums[j++];
        }
    }
    // Copy the remaining elements of the left and right subarrays into the temporary array
    while (i <= mid) {
        tmp[k++] = nums[i++];
    }
    while (j <= right) {
        tmp[k++] = nums[j++];
    }
    // Copy the elements from the temporary array tmp back to the original array nums at the
    ↪ corresponding interval
    for (k = 0; k < tmp.length; k++) {
        nums[left + k] = tmp[k];
    }
}

/* Merge sort */
function mergeSort(nums, left, right) {
    // Termination condition
    if (left >= right) return; // Terminate recursion when subarray length is 1
    // Divide and conquer stage
    let mid = Math.floor(left + (right - left) / 2); // Calculate midpoint
    mergeSort(nums, left, mid); // Recursively process the left subarray
    mergeSort(nums, mid + 1, right); // Recursively process the right subarray
    // Merge stage
    merge(nums, left, mid, right);
}
```

11.6.2 Algorithm Characteristics

- **Time complexity of $O(n \log n)$, non-adaptive sorting**: The division produces a recursion tree of height $\log n$, and the total number of merge operations at each level is n, so the overall time complexity is $O(n \log n)$.
- **Space complexity of $O(n)$, non-in-place sorting**: The recursion depth is $\log n$, using $O(\log n)$ size of stack frame space. The merge operation requires the aid of an auxiliary array, using $O(n)$ size of additional space.
- **Stable sorting**: In the merge process, the order of equal elements remains unchanged.

11.6.3 Linked List Sorting

For linked lists, merge sort has significant advantages over other sorting algorithms, **and can optimize the space complexity of linked list sorting tasks to** $O(1)$.

- **Divide phase**: "Iteration" can be used instead of "recursion" to implement linked list division work, thus saving the stack frame space used by recursion.
- **Merge phase**: In linked lists, node insertion and deletion operations can be achieved by just changing references (pointers), so there is no need to create additional linked lists during the merge phase (merging two short ordered linked lists into one long ordered linked list).

The specific implementation details are quite complex, and interested readers can consult related materials for learning.

11.7 Heap Sort

Tip
Before reading this section, please ensure you have completed the "Heap" chapter.

Heap sort (heap sort) is an efficient sorting algorithm based on the heap data structure. We can use the "build heap operation" and "element out-heap operation" that we have already learned to implement heap sort.

1. Input the array and build a min-heap, at which point the smallest element is at the heap top.
2. Continuously perform the out-heap operation, record the out-heap elements in sequence, and an ascending sorted sequence can be obtained.

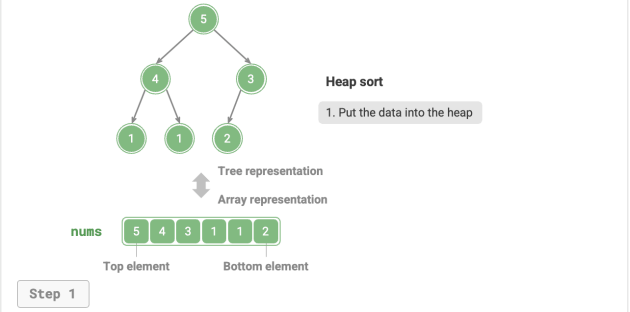
Although the above method is feasible, it requires an additional array to save the popped elements, which is quite wasteful of space. In practice, we usually use a more elegant implementation method.

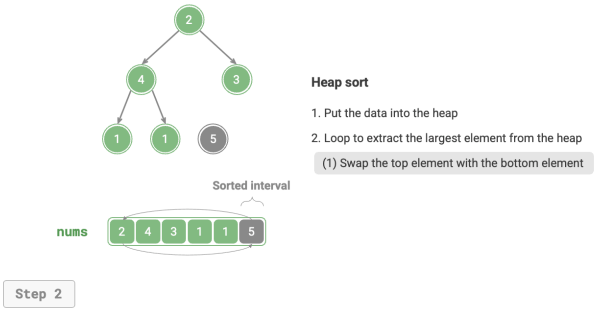
11.7.1 Algorithm Flow

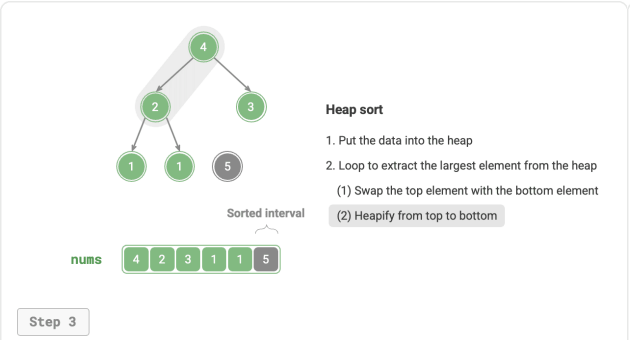
Assume the array length is n. The flow of heap sort is shown in Figure 11-12.

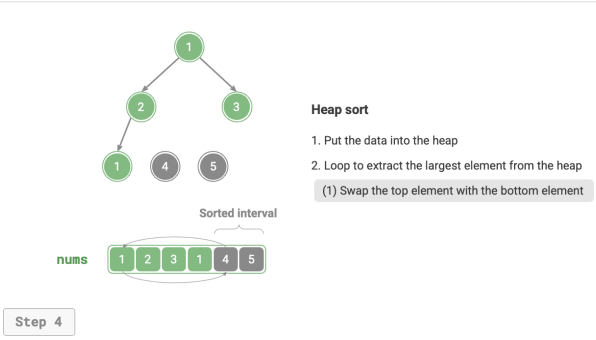
1. Input the array and build a max-heap. After completion, the largest element is at the heap top.
2. Swap the heap top element (first element) with the heap bottom element (last element). After the swap is complete, reduce the heap length by 1 and increase the count of sorted elements by 1.
3. Starting from the heap top element, perform top-to-bottom heapify operation (sift down). After heapify is complete, the heap property is restored.
4. Loop through steps `2.` and `3.` After looping $n - 1$ rounds, the array sorting can be completed.

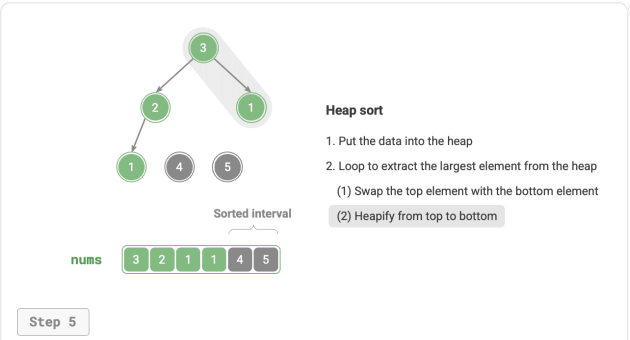
Tip
In fact, the element out-heap operation also includes steps `2.` and `3.`, with just an additional step to pop the element.

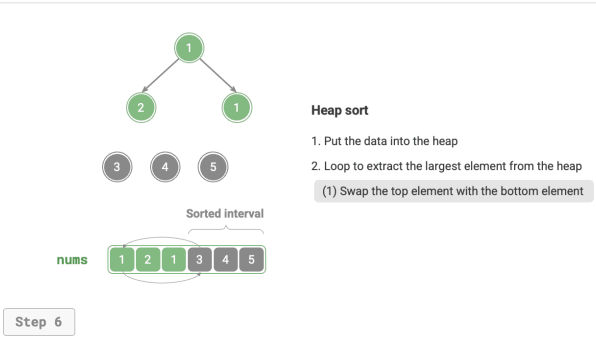

Heap sort
1. Put the data into the heap
Tree representation
Array representation
nums
Top element
Bottom element
Step 1

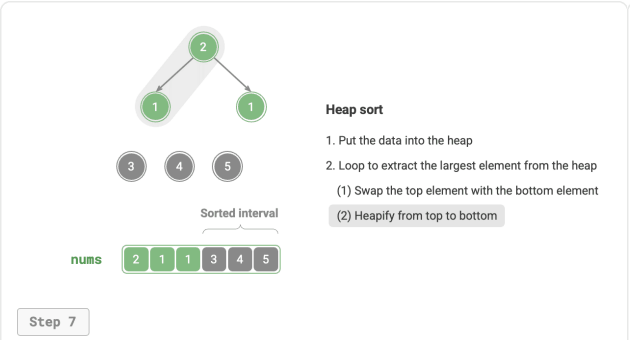

Heap sort
1. Put the data into the heap
2. Loop to extract the largest element from the heap
(1) Swap the top element with the bottom element
Sorted interval
nums
Step 2

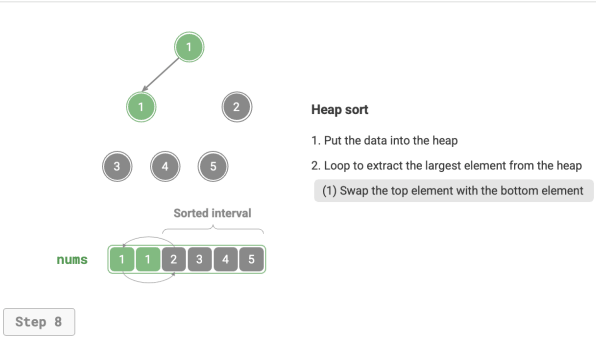

Heap sort
1. Put the data into the heap
2. Loop to extract the largest element from the heap
(1) Swap the top element with the bottom element
(2) Heapify from top to bottom
Sorted interval
nums
Step 3

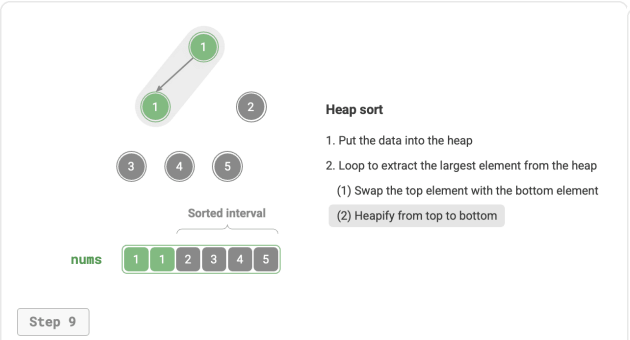

Heap sort
1. Put the data into the heap
2. Loop to extract the largest element from the heap
(1) Swap the top element with the bottom element
Sorted interval
nums
Step 4

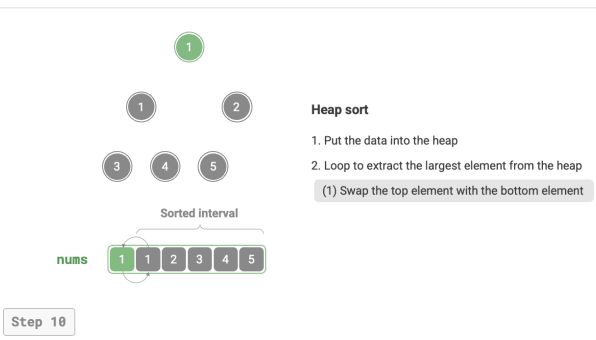

Heap sort
1. Put the data into the heap
2. Loop to extract the largest element from the heap
(1) Swap the top element with the bottom element
(2) Heapify from top to bottom
Sorted interval
nums
Step 5


Heap sort
1. Put the data into the heap
2. Loop to extract the largest element from the heap
(1) Swap the top element with the bottom element
Sorted interval
nums
Step 6


Heap sort
1. Put the data into the heap
2. Loop to extract the largest element from the heap
(1) Swap the top element with the bottom element
(2) Heapify from top to bottom
Sorted interval
nums
Step 7


Heap sort
1. Put the data into the heap
2. Loop to extract the largest element from the heap
(1) Swap the top element with the bottom element
Sorted interval
nums
Step 8


Heap sort
1. Put the data into the heap
2. Loop to extract the largest element from the heap
(1) Swap the top element with the bottom element
(2) Heapify from top to bottom
Sorted interval
nums
Step 9


Heap sort
1. Put the data into the heap
2. Loop to extract the largest element from the heap
(1) Swap the top element with the bottom element
Sorted interval
nums
Step 10

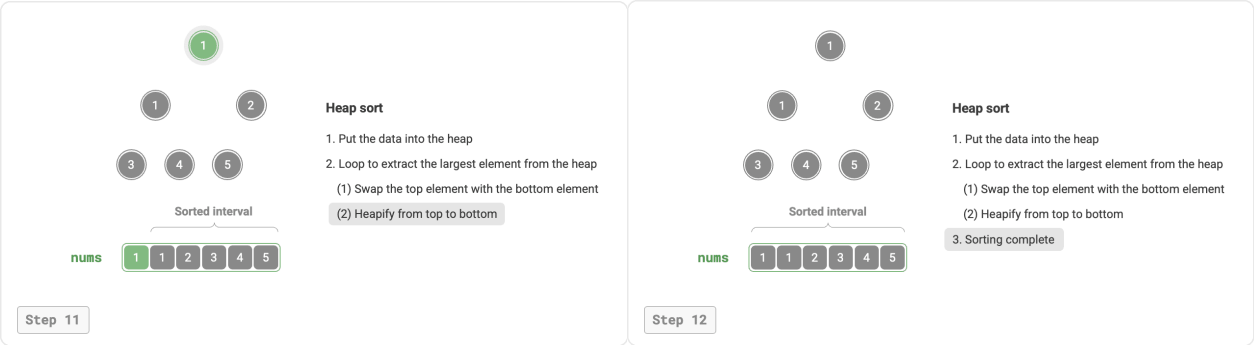


Figure 11-12 Heap sort steps

In the code implementation, we use the same top-to-bottom heapify function `sift_down()` from the "Heap" chapter. It is worth noting that since the heap length will decrease as the largest element is extracted, we need to add a length parameter n to the `sift_down()` function to specify the current effective length of the heap. The code is as follows:

```
// ═══ File: heap_sort.js ═══

/* Heap length is n, start heapifying node i, from top to bottom */
function siftDown(nums, n, i) {
    while (true) {
        // If node i is largest or indices l, r are out of bounds, no need to continue heapify,
        ↪  break
        let l = 2 * i + 1;
        let r = 2 * i + 2;
        let ma = i;
        if (l < n && nums[l] > nums[ma]) {
            ma = l;
        }
        if (r < n && nums[r] > nums[ma]) {
            ma = r;
        }
        // Swap two nodes
        if (ma === i) {
            break;
        }
        // Swap two nodes
        [nums[i], nums[ma]] = [nums[ma], nums[i]];
        // Loop downwards heapification
        i = ma;
    }
}

/* Heap sort */
function heapSort(nums) {
    // Build heap operation: heapify all nodes except leaves
    for (let i = Math.floor(nums.length / 2) - 1; i >= 0; i--) {
        siftDown(nums, nums.length, i);
    }
    // Extract the largest element from the heap and repeat for n-1 rounds
    for (let i = nums.length - 1; i > 0; i--) {
        // Delete node
        [nums[0], nums[i]] = [nums[i], nums[0]];
```

```
        // Start heapifying the root node, from top to bottom
        siftDown(nums, i, 0);
    }
}
```

11.7.2 Algorithm Characteristics

- **Time complexity of $O(n \log n)$, non-adaptive sorting**: The build heap operation uses $O(n)$ time. Extracting the largest element from the heap has a time complexity of $O(\log n)$, looping a total of $n - 1$ rounds.
- **Space complexity of $O(1)$, in-place sorting**: A few pointer variables use $O(1)$ space. Element swapping and heapify operations are both performed on the original array.
- **Non-stable sorting**: When swapping the heap top element and heap bottom element, the relative positions of equal elements may change.

11.8 Bucket Sort

The several sorting algorithms mentioned earlier all belong to "comparison-based sorting algorithms", which achieve sorting by comparing the size of elements. The time complexity of such sorting algorithms cannot exceed $O(n \log n)$. Next, we will explore several "non-comparison sorting algorithms", whose time complexity can reach linear order.

Bucket sort (bucket sort) is a typical application of the divide-and-conquer strategy. It works by setting up buckets with size order, each bucket corresponding to a data range, evenly distributing data to each bucket; then, sorting within each bucket separately; finally, merging all data in the order of the buckets.

11.8.1 Algorithm Flow

Consider an array of length n, whose elements are floating-point numbers in the range $[0, 1)$. The flow of bucket sort is shown in Figure 11-13.

1. Initialize k buckets and distribute the n elements into the k buckets.
2. Sort each bucket separately (here we use the built-in sorting function of the programming language).
3. Merge the results in order from smallest to largest bucket.

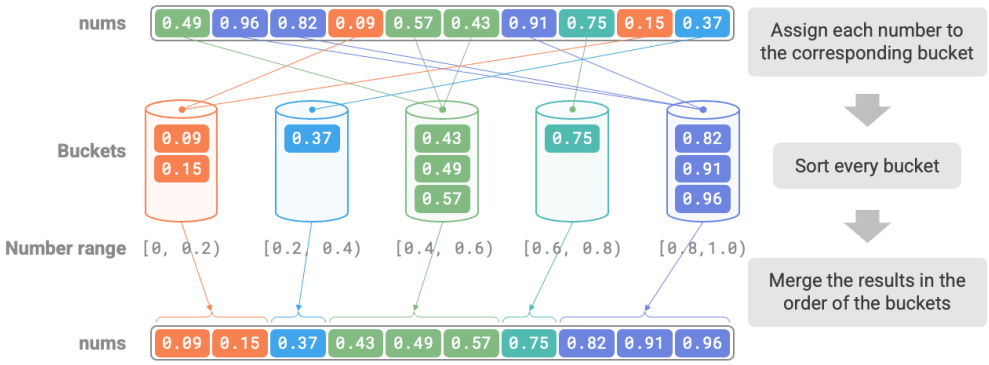


Figure 11-13 Bucket sort algorithm flow

The code is as follows:

```
// ═══ File: bucket_sort.js ═══

/* Bucket sort */
function bucketSort(nums) {
    // Initialize k = n/2 buckets, expected to allocate 2 elements per bucket
    const k = nums.length / 2;
    const buckets = [];
    for (let i = 0; i < k; i++) {
        buckets.push([]);
    }
    // 1. Distribute array elements into various buckets
    for (const num of nums) {
        // Input data range is [0, 1), use num * k to map to index range [0, k-1]
        const i = Math.floor(num * k);
        // Add num to bucket i
        buckets[i].push(num);
    }
    // 2. Sort each bucket
    for (const bucket of buckets) {
        // Use built-in sorting function, can also replace with other sorting algorithms
        bucket.sort((a, b) => a - b);
    }
    // 3. Traverse buckets to merge results
    let i = 0;
    for (const bucket of buckets) {
        for (const num of bucket) {
            nums[i++] = num;
        }
    }
}
```

11.8.2 Algorithm Characteristics

Bucket sort is suitable for processing very large data volumes. For example, if the input data contains 1 million elements and system memory cannot load all the data at once, the data can be divided into 1000 buckets, each bucket sorted separately, and then the results merged.

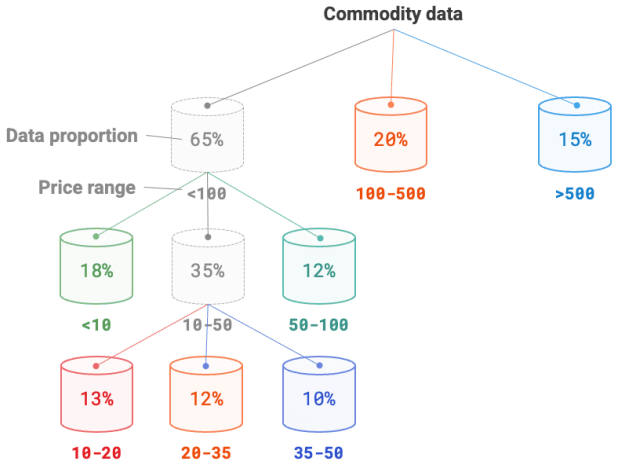
- **Time complexity of $O(n + k)$**: Assuming the elements are evenly distributed among the buckets, then the number of elements in each bucket is $\frac{n}{k}$. Assuming sorting a single bucket uses $O(\frac{n}{k} \log \frac{n}{k})$ time, then sorting all buckets uses $O(n \log \frac{n}{k})$ time. **When the number of buckets k is relatively large, the time complexity approaches $O(n)$**. Merging results requires traversing all buckets and elements, taking $O(n + k)$ time. In the worst case, all data is distributed into one bucket, and sorting that bucket uses $O(n^2)$ time.
- **Space complexity of $O(n + k)$, non-in-place sorting**: Additional space is required for k buckets and a total of n elements.
- Whether bucket sort is stable depends on whether the algorithm for sorting elements within buckets is stable.

11.8.3 How to Achieve Even Distribution

Theoretically, bucket sort can achieve $O(n)$ time complexity. **The key is to evenly distribute elements to each bucket**, because real data is often not evenly distributed. For example, if we want to evenly distribute all products on Taobao into 10 buckets by price range, there may be very many products below 100 yuan and very few above 1000 yuan. If the price intervals are evenly divided into 10, the difference in the number of products in each bucket will be very large.

To achieve even distribution, we can first set an approximate dividing line to roughly divide the data into 3 buckets. **After distribution is complete, continue dividing buckets with more products into 3 buckets until the number of elements in all buckets is roughly equal**.

As shown in Figure 11-14, this method essentially creates a recursion tree, with the goal of making the values of leaf nodes as even as possible. Of course, it is not necessary to divide the data into 3 buckets every round; the specific division method can be flexibly chosen according to data characteristics.



Data division solution

Price range	Data proportion
<10	18%
10-20	13%
20-35	12%
35-50	10%
50-100	12%
100-500	20%
>500	15%

Figure 11-14 Recursively dividing buckets

If we know the probability distribution of product prices in advance, **we can set the price dividing line for each bucket based on the data probability distribution**. It is worth noting that the data distribution

does not necessarily need to be specifically calculated, but can also be approximated using a certain probability model based on data characteristics.

As shown in Figure 11-15, we assume that product prices follow a normal distribution, which allows us to reasonably set price intervals to evenly distribute products to each bucket.

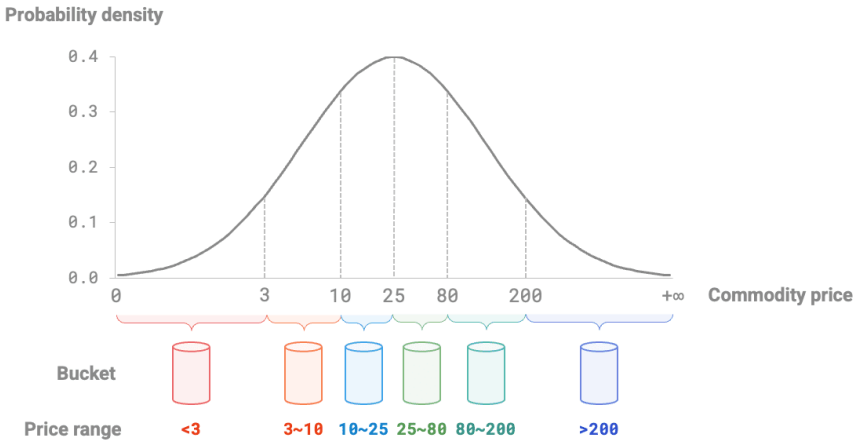


Figure 11-15 Dividing buckets based on probability distribution

11.9 Counting Sort

Counting sort (counting sort) achieves sorting by counting the number of elements, typically applied to integer arrays.

11.9.1 Simple Implementation

Let's start with a simple example. Given an array `nums` of length n, where the elements are all "non-negative integers", the overall flow of counting sort is shown in Figure 11-16.

1. Traverse the array to find the largest number, denoted as m, and then create an auxiliary array `counter` of length $m + 1$.
2. **Use `counter` to count the number of occurrences of each number in `nums`**, where `counter[num]` corresponds to the number of occurrences of the number `num`. The counting method is simple: just traverse `nums` (let the current number be `num`), and increase `counter[num]` by 1 in each round.
3. **Since each index of `counter` is naturally ordered, this is equivalent to all numbers being sorted**. Next, we traverse `counter` and fill in `nums` in ascending order based on the number of occurrences of each number.

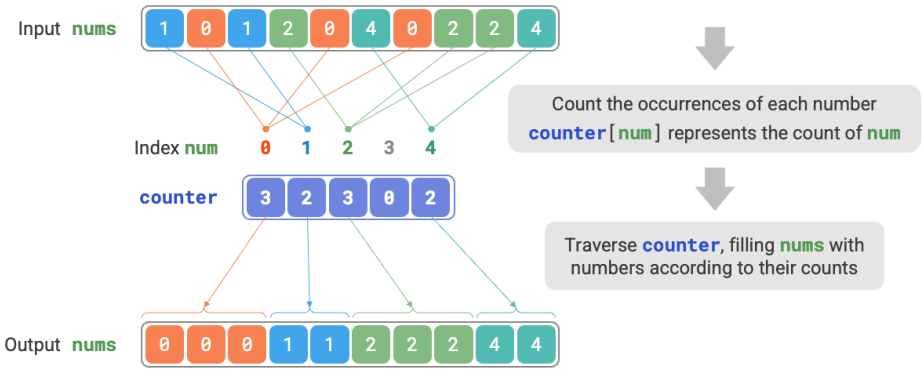


Figure 11-16 Counting sort flow

The code is as follows:

```
// ═══ File: counting_sort.js ═══

/* Counting sort */
// Simple implementation, cannot be used for sorting objects
function countingSortNaive(nums) {
    // 1. Count the maximum element m in the array
    let m = Math.max(...nums);
    // 2. Count the occurrence of each number
    // counter[num] represents the occurrence of num
    const counter = new Array(m + 1).fill(0);
    for (const num of nums) {
        counter[num]++;
    }
    // 3. Traverse counter, filling each element back into the original array nums
    let i = 0;
    for (let num = 0; num < m + 1; num++) {
        for (let j = 0; j < counter[num]; j++, i++) {
            nums[i] = num;
        }
    }
}
```

Connection between counting sort and bucket sort

From the perspective of bucket sort, we can regard each index of the counting array `counter` in counting sort as a bucket, and the process of counting quantities as distributing each element to the corresponding bucket. Essentially, counting sort is a special case of bucket sort for integer data.

11.9.2 Complete Implementation

Observant readers may have noticed that **if the input data is objects, step `3.` above becomes invalid**. Suppose the input data is product objects, and we want to sort the products by price (a member variable of the class), but the above algorithm can only give the sorting result of prices.

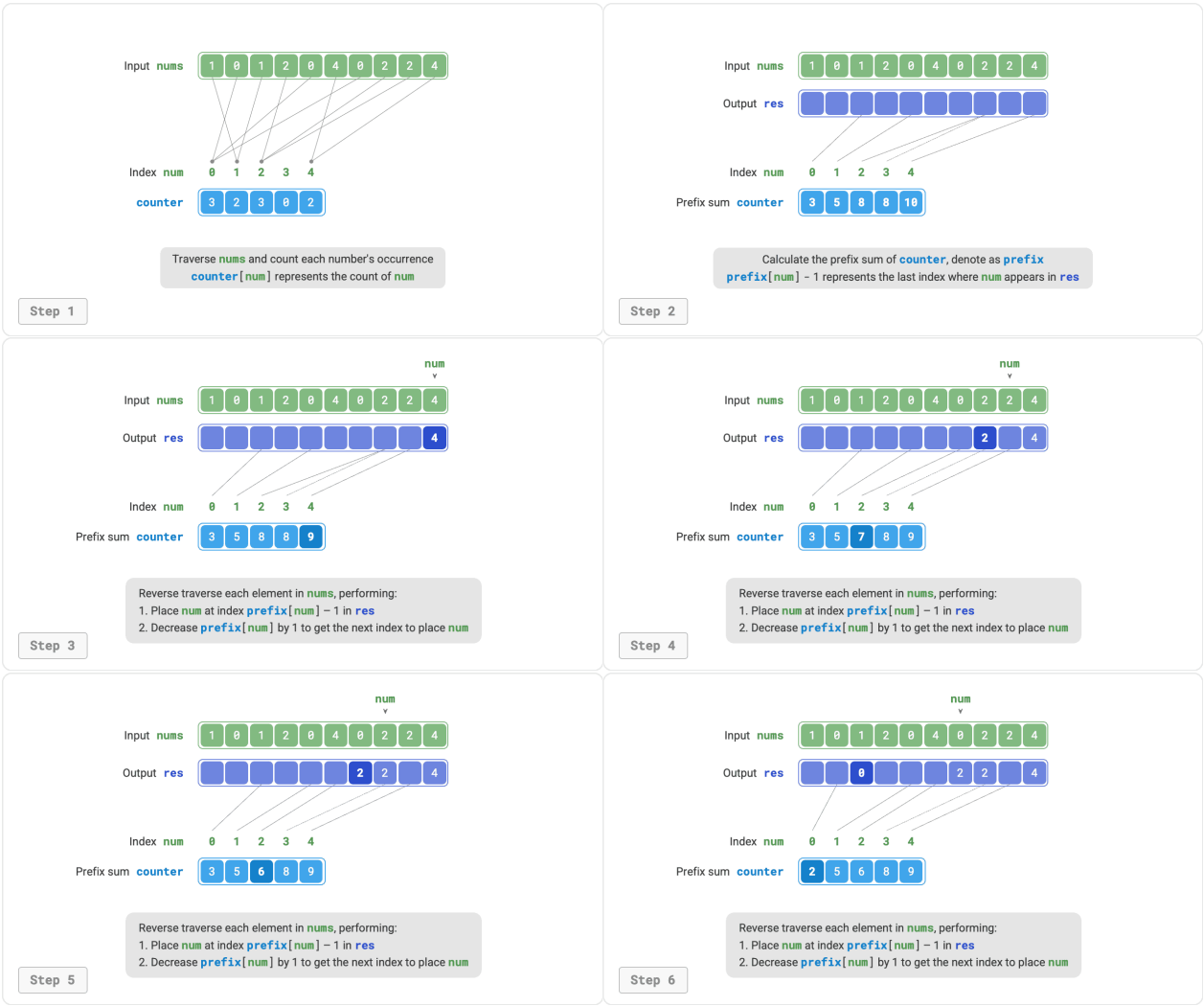
So how can we obtain the sorting result of the original data? We first calculate the "prefix sum" of `counter`. As the name suggests, the prefix sum at index `i`, `prefix[i]`, equals the sum of the first `i` elements of the array:

$$\text{prefix}[i] = \sum_{j=0}^{i} \text{counter[j]}$$

The prefix sum has a clear meaning: `prefix[num] - 1` represents the index of the last occurrence of element `num` in the result array `res`. This information is very critical because it tells us where each element should appear in the result array. Next, we traverse each element `num` of the original array `nums` in reverse order, performing the following two steps in each iteration.

1. Fill `num` into the array `res` at index `prefix[num] - 1`.
2. Decrease the prefix sum `prefix[num]` by 1 to get the index for the next placement of `num`.

After the traversal is complete, the array `res` contains the sorted result, and finally `res` is used to overwrite the original array `nums`. The complete counting sort flow is shown in Figure 11-17.



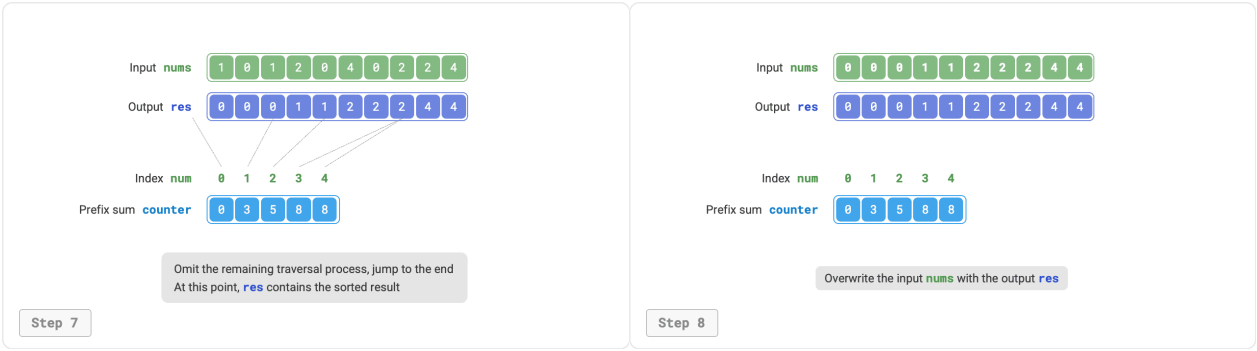


Figure 11-17 Counting sort steps

The implementation code of counting sort is as follows:

```
// ═══ File: counting_sort.js ═══

/* Counting sort */
// Complete implementation, can sort objects and is a stable sort
function countingSort(nums) {
    // 1. Count the maximum element m in the array
    let m = Math.max(...nums);
    // 2. Count the occurrence of each number
    // counter[num] represents the occurrence of num
    const counter = new Array(m + 1).fill(0);
    for (const num of nums) {
        counter[num]++;
    }
    // 3. Calculate the prefix sum of counter, converting "occurrence count" to "tail index"
    // counter[num]-1 is the last index where num appears in res
    for (let i = 0; i < m; i++) {
        counter[i + 1] += counter[i];
    }
    // 4. Traverse nums in reverse order, placing each element into the result array res
    // Initialize the array res to record results
    const n = nums.length;
    const res = new Array(n);
    for (let i = n - 1; i >= 0; i--) {
        const num = nums[i];
        res[counter[num] - 1] = num; // Place num at the corresponding index
        counter[num]--; // Decrement the prefix sum by 1, getting the next index to place num
    }
    // Use result array res to overwrite the original array nums
    for (let i = 0; i < n; i++) {
        nums[i] = res[i];
    }
}
```

11.9.3 Algorithm Characteristics

- **Time complexity of $O(n + m)$, non-adaptive sorting**: Involves traversing `nums` and traversing `counter`, both using linear time. Generally, $n \gg m$, and time complexity tends toward $O(n)$.
- **Space complexity of $O(n + m)$, non-in-place sorting**: Uses arrays `res` and `counter` of lengths n and m respectively.

- **Stable sorting**: Since elements are filled into `res` in a "right-to-left" order, traversing `nums` in reverse can avoid changing the relative positions of equal elements, thereby achieving stable sorting. In fact, traversing `nums` in forward order can also yield correct sorting results, but the result would be unstable.

11.9.4 Limitations

By this point, you might think counting sort is very clever, as it can achieve efficient sorting just by counting quantities. However, the prerequisites for using counting sort are relatively strict.

Counting sort is only suitable for non-negative integers. If you want to apply it to other types of data, you need to ensure that the data can be converted to non-negative integers without changing the relative size relationships between elements. For example, for an integer array containing negative numbers, you can first add a constant to all numbers to convert them all to positive numbers, and then convert them back after sorting is complete.

Counting sort is suitable for situations where the data volume is large but the data range is small. For example, in the above example, m cannot be too large, otherwise it will occupy too much space. And when $n \ll m$, counting sort uses $O(m)$ time, which may be slower than $O(n \log n)$ sorting algorithms.

11.10 Radix Sort

The previous section introduced counting sort, which is suitable for situations where the data volume n is large but the data range m is small. Suppose we need to sort $n = 10^6$ student IDs, and the student ID is an 8-digit number, which means the data range $m = 10^8$ is very large. Using counting sort would require allocating a large amount of memory space, whereas radix sort can avoid this situation.

Radix sort (radix sort) has a core idea consistent with counting sort, which also achieves sorting by counting quantities. Building on this, radix sort utilizes the progressive relationship between the digits of numbers, sorting each digit in turn to obtain the final sorting result.

11.10.1 Algorithm Flow

Taking student ID data as an example, assume the lowest digit is the 1st digit and the highest digit is the 8th digit. The flow of radix sort is shown in Figure 11-18.

1. Initialize the digit $k = 1$.
2. Perform "counting sort" on the kth digit of the student IDs. After completion, the data will be sorted from smallest to largest according to the kth digit.
3. Increase k by 1, then return to step `2.` and continue iterating until all digits are sorted, at which point the process ends.

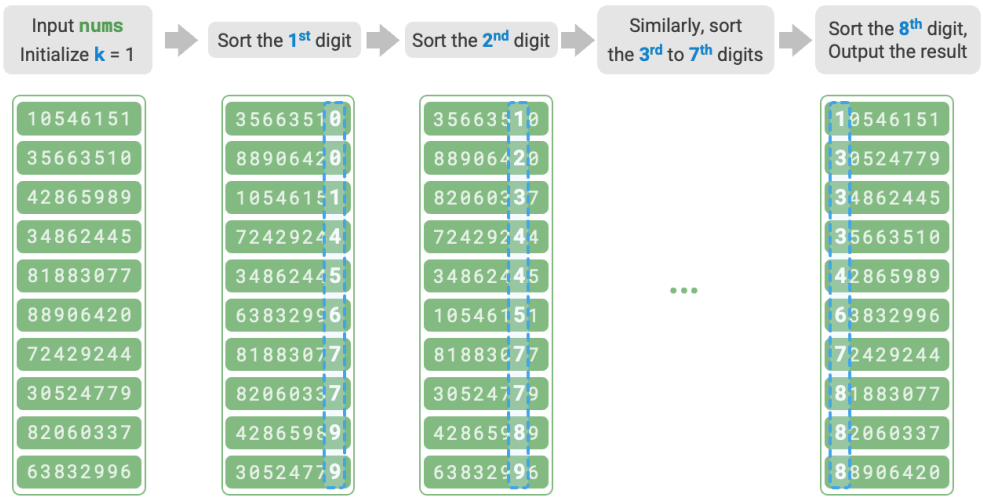


Figure 11-18 Radix sort algorithm flow

Below we analyze the code implementation. For a d-base number x, to get its kth digit x_k, the following calculation formula can be used:

$$x_k = \lfloor \frac{x}{d^{k-1}} \rfloor \bmod d$$

Where $\lfloor a \rfloor$ denotes rounding down the floating-point number a, and $\bmod d$ denotes taking the modulo (remainder) with respect to d. For student ID data, $d = 10$ and $k \in [1, 8]$.

Additionally, we need to slightly modify the counting sort code to make it sort based on the kth digit of the number:

```
// == File: radix_sort.js ==

/* Get the k-th digit of element num, where exp = 10^(k-1) */
function digit(num, exp) {
    // Passing exp instead of k can avoid repeated expensive exponentiation here
    return Math.floor(num / exp) % 10;
}

/* Counting sort (based on nums k-th digit) */
function countingSortDigit(nums, exp) {
    // Decimal digit range is 0~9, therefore need a bucket array of length 10
    const counter = new Array(10).fill(0);
    const n = nums.length;
    // Count the occurrence of digits 0~9
    for (let i = 0; i < n; i++) {
        const d = digit(nums[i], exp); // Get the k-th digit of nums[i], noted as d
        counter[d]++; // Count the occurrence of digit d
    }
    // Calculate prefix sum, converting "occurrence count" into "array index"
    for (let i = 1; i < 10; i++) {
        counter[i] += counter[i - 1];
    }
```

```javascript
    // Traverse in reverse, based on bucket statistics, place each element into res
    const res = new Array(n).fill(0);
    for (let i = n - 1; i >= 0; i--) {
        const d = digit(nums[i], exp);
        const j = counter[d] - 1; // Get the index j for d in the array
        res[j] = nums[i]; // Place the current element at index j
        counter[d]--; // Decrease the count of d by 1
    }
    // Use result to overwrite the original array nums
    for (let i = 0; i < n; i++) {
        nums[i] = res[i];
    }
}

/* Radix sort */
function radixSort(nums) {
    // Get the maximum element of the array, used to determine the maximum number of digits
    let m = Math.max(... nums);
    // Traverse from the lowest to the highest digit
    for (let exp = 1; exp <= m; exp *= 10) {
        // Perform counting sort on the k-th digit of array elements
        // k = 1 -> exp = 1
        // k = 2 -> exp = 10
        // i.e., exp = 10^(k-1)
        countingSortDigit(nums, exp);
    }
}
```

Why start sorting from the lowest digit?

In successive sorting rounds, the result of a later round will override the result of an earlier round. For example, if the first round result is $a < b$, while the second round result is $a > b$, then the second round's result will replace the first round's result. Since higher-order digits have higher priority than lower-order digits, we should sort the lower digits first and then sort the higher digits.

11.10.2 Algorithm Characteristics

Compared to counting sort, radix sort is suitable for larger numerical ranges, **but the prerequisite is that the data must be representable in a fixed number of digits, and the number of digits should not be too large**. For example, floating-point numbers are not suitable for radix sort because their number of digits k may be too large, potentially leading to time complexity $O(nk) \gg O(n^2)$.

- **Time complexity of $O(nk)$, non-adaptive sorting**: Let the data volume be n, the data be in base d, and the maximum number of digits be k. Then performing counting sort on a certain digit uses $O(n+d)$ time, and sorting all k digits uses $O((n+d)k)$ time. Typically, both d and k are relatively small, and the time complexity approaches $O(n)$.
- **Space complexity of $O(n+d)$, non-in-place sorting**: Same as counting sort, radix sort requires auxiliary arrays `res` and `counter` of lengths n and d.
- **Stable sorting**: When counting sort is stable, radix sort is also stable; when counting sort is unstable, radix sort cannot guarantee obtaining correct sorting results.

11.11 Summary

1. Key Review

- Bubble sort achieves sorting by swapping adjacent elements. By adding a flag to enable early return, we can optimize the best-case time complexity of bubble sort to $O(n)$.
- Insertion sort completes sorting by inserting elements from the unsorted interval into the correct position in the sorted interval each round. Although the time complexity of insertion sort is $O(n^2)$, it is very popular in small data volume sorting tasks because it involves relatively few unit operations.
- Quick sort is implemented based on sentinel partitioning operations. In sentinel partitioning, it is possible to select the worst pivot every time, causing the time complexity to degrade to $O(n^2)$. Introducing median pivot or random pivot can reduce the probability of such degradation. By preferentially recursing on the shorter sub-interval, the recursion depth can be effectively reduced, optimizing the space complexity to $O(\log n)$.
- Merge sort includes two phases: divide and merge, which typically embody the divide-and-conquer strategy. In merge sort, sorting an array requires creating auxiliary arrays, with a space complexity of $O(n)$; however, the space complexity of sorting a linked list can be optimized to $O(1)$.
- Bucket sort consists of three steps: distributing data into buckets, sorting within buckets, and merging results. It also embodies the divide-and-conquer strategy and is suitable for very large data volumes. The key to bucket sort is distributing data evenly.
- Counting sort is a special case of bucket sort, which achieves sorting by counting the number of occurrences of data. Counting sort is suitable for situations where the data volume is large but the data range is limited, and requires that data can be converted to positive integers.
- Radix sort achieves data sorting by sorting digit by digit, requiring that data can be represented as fixed-digit numbers.
- Overall, we hope to find a sorting algorithm that is efficient, stable, in-place, and adaptive, with good versatility. However, just like other data structures and algorithms, no sorting algorithm has been found so far that simultaneously possesses all these characteristics. In practical applications, we need to select the appropriate sorting algorithm based on the specific characteristics of the data.
- Figure 11-19 compares mainstream sorting algorithms in terms of efficiency, stability, in-place property, and adaptability.

	Time complexity			Space complexity	Stability	In-place	Adaptivity	Comparison
	Best	Average	Worst	Worst				
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	Unstable	In-place	Non-adaptive	Comparison-based
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Stable	In-place	Adaptive	Comparison-based
Insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Stable	In-place	Adaptive	Comparison-based
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	Unstable	In-place	Non-adaptive	Comparison-based
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Stable	Not in-place	Non-adaptive	Comparison-based
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	Unstable	In-place	Non-adaptive	Comparison-based
Bucket sort	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Stable	Not in-place	Non-adaptive	Non-comparison
Counting sort	$O(n + m)$	$O(n + m)$	$O(n + m)$	$O(n + m)$	Stable	Not in-place	Non-adaptive	Non-comparison
Radic sort	$O(n k)$	$O(n k)$	$O(n k)$	$O(n + b)$	Stable	Not in-place	Non-adaptive	Non-comparison

Poor Fair Good

- **n** is the size of the data
- In bucket sort, **k** is the number of buckets
- In counting sort, **m** is the data range
- In radix sort, **k** is the maximum number of digits, data in **b** base

Figure 11-19 Sorting algorithm comparison

2. Q & A

Q: In what situations is the stability of sorting algorithms necessary?

In reality, we may sort based on a certain attribute of objects. For example, students have two attributes: name and height. We want to implement multi-level sorting: first sort by name to get `(A, 180) (B, 185) (C, 170) (D, 170)`; then sort by height. Because the sorting algorithm is unstable, we may get `(D, 170) (C, 170) (A, 180) (B, 185)`.

It can be seen that the positions of students D and C have been swapped, and the orderliness of names has been disrupted, which is something we don't want to see.

Q: Can the order of "searching from right to left" and "searching from left to right" in sentinel partitioning be swapped?

No. When we use the leftmost element as the pivot, we must first "search from right to left" and then "search from left to right". This conclusion is somewhat counterintuitive; let's analyze the reason.

The last step of sentinel partitioning `partition()` is to swap `nums[left]` and `nums[i]`. After the swap is complete, the elements to the left of the pivot are all `<=` the pivot, **which requires that `nums[left] >= nums[i]` must hold before the last swap**. Suppose we first "search from left to right", then if we cannot find an element larger than the pivot, **we will exit the loop when `i == j`, at which point it may be that `nums[j] == nums[i] > nums[left]`**. In other words, the last swap operation will swap an element larger than the pivot to the leftmost end of the array, causing sentinel partitioning to fail.

For example, given the array `[0, 0, 0, 0, 1]`, if we first "search from left to right", the array after sentinel partitioning is `[1, 0, 0, 0, 0]`, which is incorrect.

Thinking deeper, if we select `nums[right]` as the pivot, then it's exactly the opposite - we must first "search from left to right".

Q: Regarding the optimization of recursion depth in quick sort, why can selecting the shorter array ensure that the recursion depth does not exceed $\log n$?

The recursion depth is the number of currently unreturned recursive methods. Each round of sentinel partitioning divides the original array into two sub-arrays. After recursion depth optimization, the length of the sub-array to be recursively processed is at most half of the original array length. Assuming the worst case is always half the length, the final recursion depth will be $\log n$.

Reviewing the original quick sort, we may continuously recurse on the longer array. In the worst case, it would be n, $n-1$, ..., 2, 1, with a recursion depth of n. Recursion depth optimization can avoid this situation.

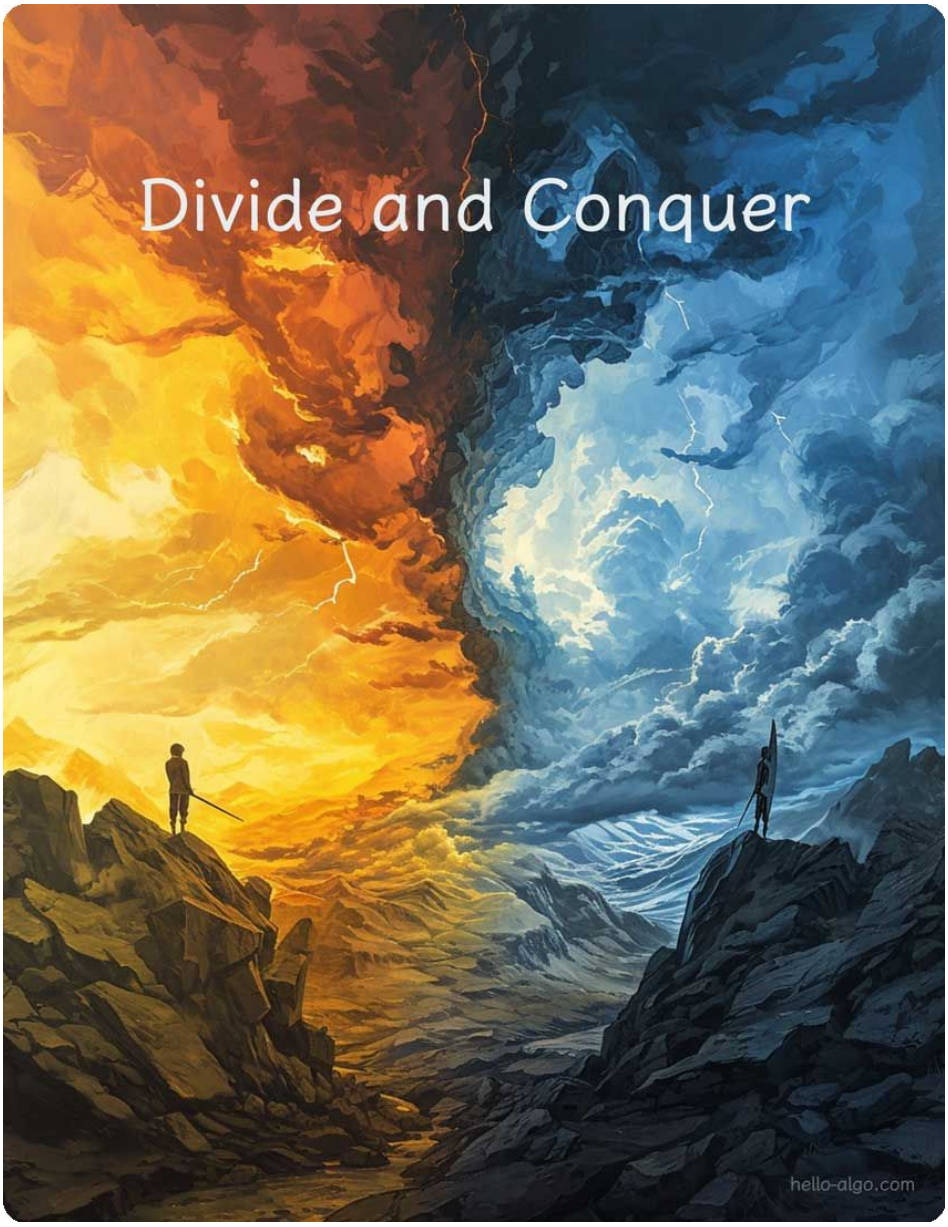
Q: When all elements in the array are equal, is the time complexity of quick sort $O(n^2)$? How should this degenerate case be handled?

Yes. For this situation, consider partitioning the array into three parts through sentinel partitioning: less than, equal to, and greater than the pivot. Only recursively process the less than and greater than parts. Under this method, an array where all input elements are equal can complete sorting in just one round of sentinel partitioning.

Q: Why is the worst-case time complexity of bucket sort $O(n^2)$?

In the worst case, all elements are distributed into the same bucket. If we use an $O(n^2)$ algorithm to sort these elements, the time complexity will be $O(n^2)$.

Chapter 12. Divide and Conquer



Abstract

Difficult problems are decomposed layer by layer, with each decomposition making them simpler.

Divide and conquer reveals an important truth: start with simplicity, and nothing remains complex.

12.1 Divide and Conquer Algorithms

Divide and conquer is a very important and common algorithm strategy. Divide and conquer is typically implemented based on recursion, consisting of two steps: "divide" and "conquer".

1. **Divide (partition phase)**: Recursively divide the original problem into two or more subproblems until the smallest subproblem is reached.
2. **Conquer (merge phase)**: Starting from the smallest subproblems with known solutions, merge the solutions of subproblems from bottom to top to construct the solution to the original problem.

As shown in Figure 12-1, "merge sort" is one of the typical applications of the divide and conquer strategy.

1. **Divide**: Recursively divide the original array (original problem) into two subarrays (subproblems) until the subarray has only one element (smallest subproblem).
2. **Conquer**: Merge the sorted subarrays (solutions to subproblems) from bottom to top to obtain a sorted original array (solution to the original problem).

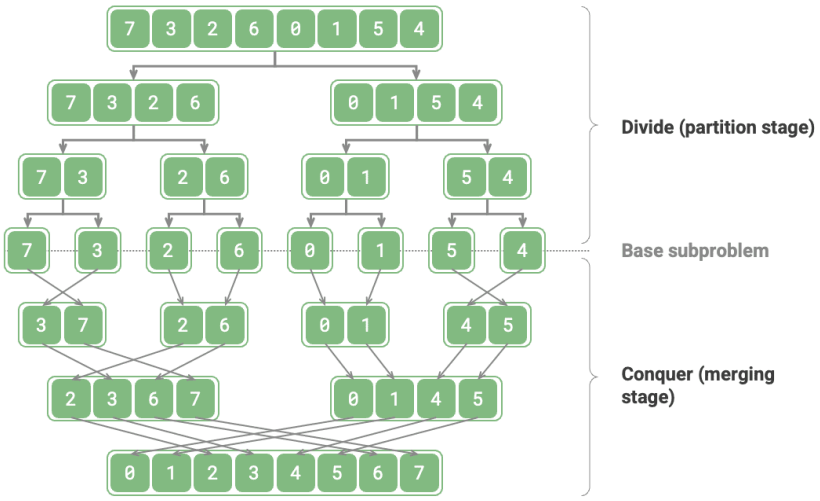


Figure 12-1 Divide and conquer strategy of merge sort

12.1.1 How to Determine Divide and Conquer Problems

Whether a problem is suitable for solving with divide and conquer can usually be determined based on the following criteria.

1. **The problem can be decomposed**: The original problem can be divided into smaller, similar subproblems, and can be recursively divided in the same way.
2. **Subproblems are independent**: There is no overlap between subproblems, they are independent of each other and can be solved independently.

3. **Solutions of subproblems can be merged**: The solution to the original problem is obtained by merging the solutions of subproblems.

Clearly, merge sort satisfies these three criteria.

1. **The problem can be decomposed**: Recursively divide the array (original problem) into two subarrays (subproblems).
2. **Subproblems are independent**: Each subarray can be sorted independently (subproblems can be solved independently).
3. **Solutions of subproblems can be merged**: Two sorted subarrays (solutions of subproblems) can be merged into one sorted array (solution of the original problem).

12.1.2 Improving Efficiency Through Divide and Conquer

Divide and conquer can not only effectively solve algorithmic problems but often also improve algorithm efficiency. In sorting algorithms, quick sort, merge sort, and heap sort are faster than selection, bubble, and insertion sort because they apply the divide and conquer strategy.

This raises the question: **Why can divide and conquer improve algorithm efficiency, and what is the underlying logic**? In other words, why is dividing a large problem into multiple subproblems, solving the subproblems, and merging their solutions more efficient than directly solving the original problem? This question can be discussed from two aspects: operation count and parallel computation.

1. Operation Count Optimization

Taking "bubble sort" as an example, processing an array of length n requires $O(n^2)$ time. Suppose we divide the array into two subarrays from the midpoint as shown in Figure 12-2, the division requires $O(n)$ time, sorting each subarray requires $O((n/2)^2)$ time, and merging the two subarrays requires $O(n)$ time, resulting in an overall time complexity of:

$$O(n + (\frac{n}{2})^2 \times 2 + n) = O(\frac{n^2}{2} + 2n)$$

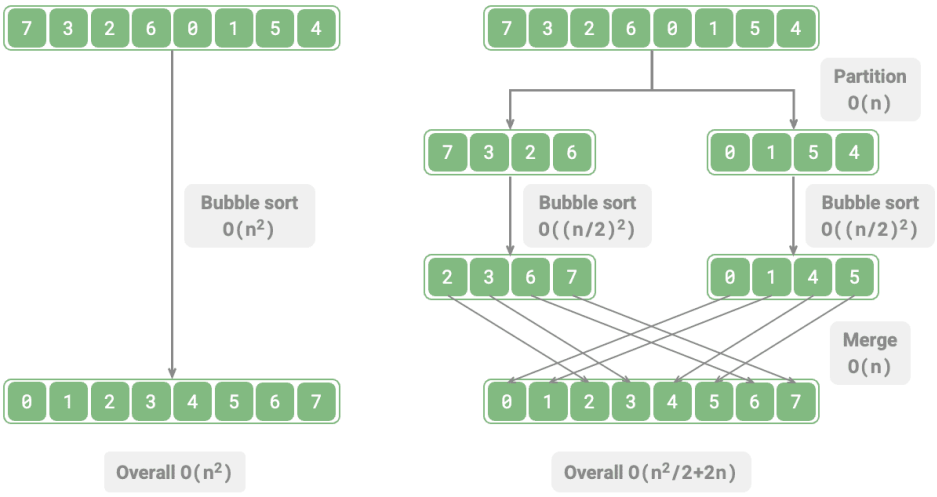


Figure 12-2 Bubble sort before and after array division

Next, we compute the following inequality, where the left and right sides represent the total number of operations before and after division, respectively:

$$
\begin{aligned}
n^2 &> \frac{n^2}{2} + 2n \\
n^2 - \frac{n^2}{2} - 2n &> 0 \\
n(n-4) &> 0
\end{aligned}
$$

This means that when $n > 4$, the number of operations after division is smaller, and sorting efficiency should be higher. Note that the time complexity after division is still quadratic $O(n^2)$, but the constant term in the complexity has become smaller.

Going further, **what if we continuously divide the subarrays from their midpoints into two subarrays** until the subarrays have only one element? This approach is actually "merge sort", with a time complexity of $O(n \log n)$.

Thinking further, **what if we set multiple division points** and evenly divide the original array into k subarrays? This situation is very similar to "bucket sort", which is well-suited for sorting massive amounts of data, with a theoretical time complexity of $O(n + k)$.

2. Parallel Computation Optimization

We know that the subproblems generated by divide and conquer are independent of each other, **so they can typically be solved in parallel**. This means divide and conquer can not only reduce the time complexity of algorithms, **but also benefits from parallel optimization by operating systems**.

Parallel optimization is particularly effective in multi-core or multi-processor environments, as the system can simultaneously handle multiple subproblems, making fuller use of computing resources and significantly reducing overall runtime.

For example, in the "bucket sort" shown in Figure 12-3, we evenly distribute massive data into various buckets, and the sorting tasks for all buckets can be distributed to various computing units. After completion, the results are merged.

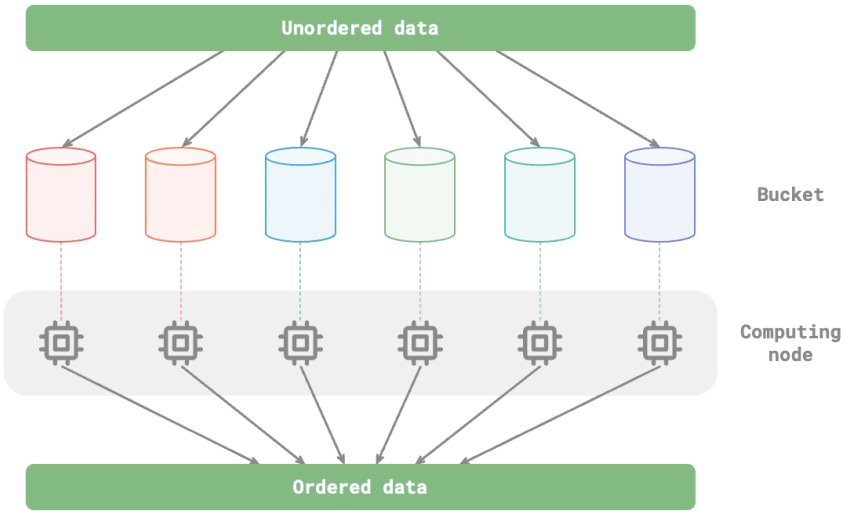


Figure 12-3 Parallel computation in bucket sort

12.1.3 Common Applications of Divide and Conquer

On one hand, divide and conquer can be used to solve many classic algorithmic problems.

- **Finding the closest pair of points**: This algorithm first divides the point set into two parts, then finds the closest pair of points in each part separately, and finally finds the closest pair of points that spans both parts.
- **Large integer multiplication**: For example, the Karatsuba algorithm, which decomposes large integer multiplication into several smaller integer multiplications and additions.
- **Matrix multiplication**: For example, the Strassen algorithm, which decomposes large matrix multiplication into multiple small matrix multiplications and additions.
- **Hanota problem**: The hanota problem can be solved through recursion, which is a typical application of the divide and conquer strategy.
- **Solving inversion pairs**: In a sequence, if a preceding number is greater than a following number, these two numbers form an inversion pair. Solving the inversion pair problem can utilize the divide and conquer approach with the help of merge sort.

On the other hand, divide and conquer is widely applied in the design of algorithms and data structures.

- **Binary search**: Binary search divides a sorted array into two parts from the midpoint index, then decides which half to eliminate based on the comparison result between the target value and the middle element value, and performs the same binary operation on the remaining interval.
- **Merge sort**: Already introduced at the beginning of this section, no further elaboration needed.
- **Quick sort**: Quick sort selects a pivot value, then divides the array into two subarrays, one with elements smaller than the pivot and the other with elements larger than the pivot, then performs the same division operation on these two parts until the subarrays have only one element.
- **Bucket sort**: The basic idea of bucket sort is to scatter data into multiple buckets, then sort the elements within each bucket, and finally extract the elements from each bucket in sequence to obtain a sorted array.
- **Trees**: For example, binary search trees, AVL trees, red-black trees, B-trees, B+ trees, etc. Their search, insertion, and deletion operations can all be viewed as applications of the divide and conquer strategy.
- **Heaps**: A heap is a special complete binary tree, and its various operations, such as insertion, deletion, and heapify, actually imply the divide and conquer idea.
- **Hash tables**: Although hash tables do not directly apply divide and conquer, some hash collision resolution solutions indirectly apply the divide and conquer strategy. For example, long linked lists in chaining may be converted to red-black trees to improve query efficiency.

It can be seen that **divide and conquer is a "subtly pervasive" algorithmic idea**, embedded in various algorithms and data structures.

12.2 Divide and Conquer Search Strategy

We have already learned that search algorithms are divided into two major categories.

- **Brute-force search**: Implemented by traversing the data structure, with a time complexity of $O(n)$.
- **Adaptive search**: Utilizes unique data organization forms or prior information, with time complexity reaching $O(\log n)$ or even $O(1)$.

In fact, **search algorithms with time complexity of $O(\log n)$ are typically implemented based on the divide and conquer strategy**, such as binary search and trees.

- Each step of binary search divides the problem (searching for a target element in an array) into a smaller problem (searching for the target element in half of the array), continuing until the array is empty or the target element is found.
- Trees are representative of the divide and conquer idea. In data structures such as binary search trees, AVL trees, and heaps, the time complexity of various operations is $O(\log n)$.

The divide and conquer strategy of binary search is as follows.

- **The problem can be decomposed**: Binary search recursively decomposes the original problem (searching in an array) into subproblems (searching in half of the array), achieved by comparing the middle element with the target element.

- **Subproblems are independent**: In binary search, each round only processes one subproblem, which is not affected by other subproblems.
- **Solutions of subproblems do not need to be merged**: Binary search aims to find a specific element, so there is no need to merge the solutions of subproblems. When a subproblem is solved, the original problem is also solved.

Divide and conquer can improve search efficiency because brute-force search can only eliminate one option per round, **while divide and conquer search can eliminate half of the options per round**.

1. Implementing Binary Search Based on Divide and Conquer

In previous sections, binary search was implemented based on iteration. Now we implement it based on divide and conquer (recursion).

> **Question**
> Given a sorted array `nums` of length n, where all elements are unique, find the element `target`.

From a divide and conquer perspective, we denote the subproblem corresponding to the search interval $[i, j]$ as $f(i, j)$.

Starting from the original problem $f(0, n-1)$, perform binary search through the following steps.

1. Calculate the midpoint m of the search interval $[i, j]$, and use it to eliminate half of the search interval.
2. Recursively solve the subproblem reduced by half in size, which could be $f(i, m-1)$ or $f(m+1, j)$.
3. Repeat steps `1.` and `2.` until `target` is found or the interval is empty and return.

Figure 12-4 shows the divide and conquer process of binary search for element 6 in an array.

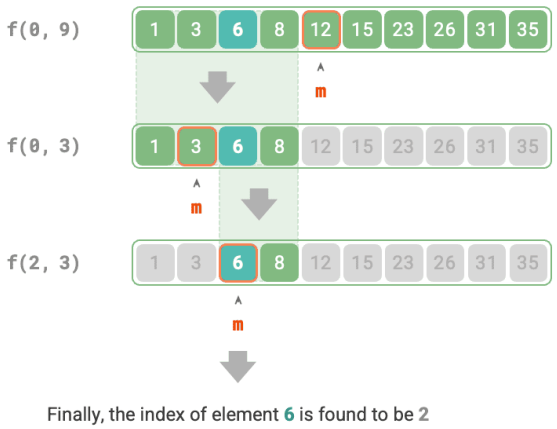


Figure 12-4 Divide and conquer process of binary search

In the implementation code, we declare a recursive function `dfs()` to solve the problem $f(i, j)$:

```javascript
// === File: binary_search_recur.js ===

/* Binary search: problem f(i, j) */
function dfs(nums, target, i, j) {
    // If the interval is empty, it means there is no target element, return -1
    if (i > j) {
        return -1;
    }
    // Calculate the midpoint index m
    const m = i + ((j - i) >> 1);
    if (nums[m] < target) {
        // Recursion subproblem f(m+1, j)
        return dfs(nums, target, m + 1, j);
    } else if (nums[m] > target) {
        // Recursion subproblem f(i, m-1)
        return dfs(nums, target, i, m - 1);
    } else {
        // Found the target element, return its index
        return m;
    }
}

/* Binary search */
function binarySearch(nums, target) {
    const n = nums.length;
    // Solve the problem f(0, n-1)
    return dfs(nums, target, 0, n - 1);
}
```

12.3 Building a Binary Tree Problem

Question

Given the preorder traversal `preorder` and inorder traversal `inorder` of a binary tree, construct the binary tree and return the root node of the binary tree. Assume there are no duplicate node values in the binary tree (as shown in Figure 12-5).

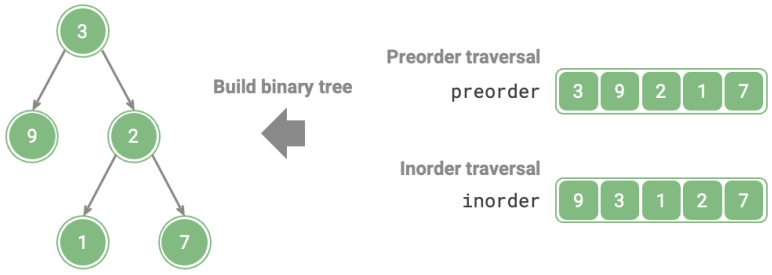


Figure 12-5 Example data for building a binary tree

1. Determining If It Is a Divide and Conquer Problem

The original problem is defined as constructing a binary tree from `preorder` and `inorder`, which is a typical divide and conquer problem.

- **The problem can be decomposed**: From a divide and conquer perspective, we can divide the original problem into two subproblems: constructing the left subtree and constructing the right subtree, plus one operation: initializing the root node. For each subtree (subproblem), we can still reuse the above division method, dividing it into smaller subtrees (subproblems) until the smallest subproblem (empty subtree) is reached.
- **Subproblems are independent**: The left and right subtrees are independent of each other; there is no overlap between them. When constructing the left subtree, we only need to focus on the parts of the inorder and preorder traversals corresponding to the left subtree. The same applies to the right subtree.
- **Solutions of subproblems can be merged**: Once we have the left and right subtrees (solutions of subproblems), we can link them to the root node to obtain the solution to the original problem.

2. How to Divide Subtrees

Based on the above analysis, this problem can be solved using divide and conquer, **but how do we divide the left and right subtrees through the preorder traversal `preorder` and inorder traversal `inorder`**?

According to the definition, both `preorder` and `inorder` can be divided into three parts.

- Preorder traversal: `[ Root Node | Left Subtree | Right Subtree ]`, for example, the tree in Figure 12-5 corresponds to `[ 3 | 9 | 2 1 7 ]`.
- Inorder traversal: `[ Left Subtree | Root Node | Right Subtree ]`, for example, the tree in Figure 12-5 corresponds to `[ 9 | 3 | 1 2 7 ]`.

Using the data from the figure above as an example, we can obtain the division results through the steps shown in Figure 12-6.

1. The first element 3 in the preorder traversal is the value of the root node.
2. Find the index of root node 3 in `inorder`, and use this index to divide `inorder` into `[ 9 | 3 | 1 2 7 ]`.
3. Based on the division result of `inorder`, it is easy to determine that the left and right subtrees have 1 and 3 nodes respectively, allowing us to divide `preorder` into `[ 3 | 9 | 2 1 7 ]`.

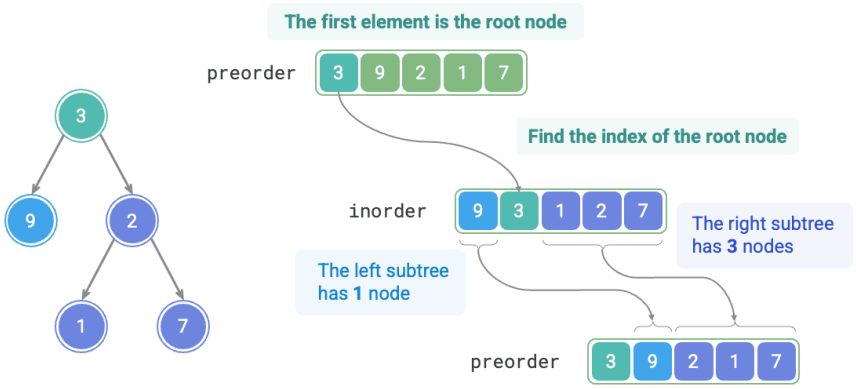


Figure 12-6 Dividing subtrees in preorder and inorder traversals

3. Describing Subtree Intervals Based on Variables

Based on the above division method, **we have obtained the index intervals of the root node, left subtree, and right subtree in `preorder` and `inorder`**. To describe these index intervals, we need to use several pointer variables.

- Denote the index of the current tree's root node in `preorder` as i.
- Denote the index of the current tree's root node in `inorder` as m.
- Denote the index interval of the current tree in `inorder` as $[l, r]$.

As shown in Table 12-1, through these variables we can represent the index of the root node in `preorder` and the index intervals of the subtrees in `inorder`.

Table 12-1 Indices of root node and subtrees in preorder and inorder traversals

	Root node index in `preorder`	Subtree index interval in `inorder`
Current tree	i	$[l, r]$
Left subtree	$i + 1$	$[l, m - 1]$
Right subtree	$i + 1 + (m - l)$	$[m + 1, r]$

Please note that $(m - l)$ in the right subtree root node index means "the number of nodes in the left subtree". It is recommended to understand this in conjunction with Figure 12-7.

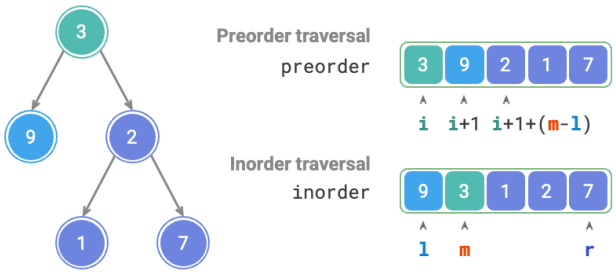


Figure 12-7 Index interval representation of root node and left and right subtrees

4. Code Implementation

To improve the efficiency of querying m, we use a hash table `hmap` to store the mapping from elements in the `inorder` array to their indices:

```
// ═══ File: build_tree.js ═══

/* Build binary tree: divide and conquer */
function dfs(preorder, inorderMap, i, l, r) {
    // Terminate when the subtree interval is empty
    if (r - l < 0) return null;
    // Initialize the root node
    const root = new TreeNode(preorder[i]);
    // Query m to divide the left and right subtrees
    const m = inorderMap.get(preorder[i]);
    // Subproblem: build the left subtree
    root.left = dfs(preorder, inorderMap, i + 1, l, m - 1);
    // Subproblem: build the right subtree
    root.right = dfs(preorder, inorderMap, i + 1 + m - l, m + 1, r);
    // Return the root node
    return root;
}

/* Build binary tree */
function buildTree(preorder, inorder) {
    // Initialize hash map, storing the mapping from inorder elements to indices
    let inorderMap = new Map();
    for (let i = 0; i < inorder.length; i++) {
        inorderMap.set(inorder[i], i);
    }
    const root = dfs(preorder, inorderMap, 0, 0, inorder.length - 1);
    return root;
}
```

Figure 12-8 shows the recursive process of building the binary tree. Each node is established during the downward "recursion" process, while each edge (reference) is established during the upward "return" process.

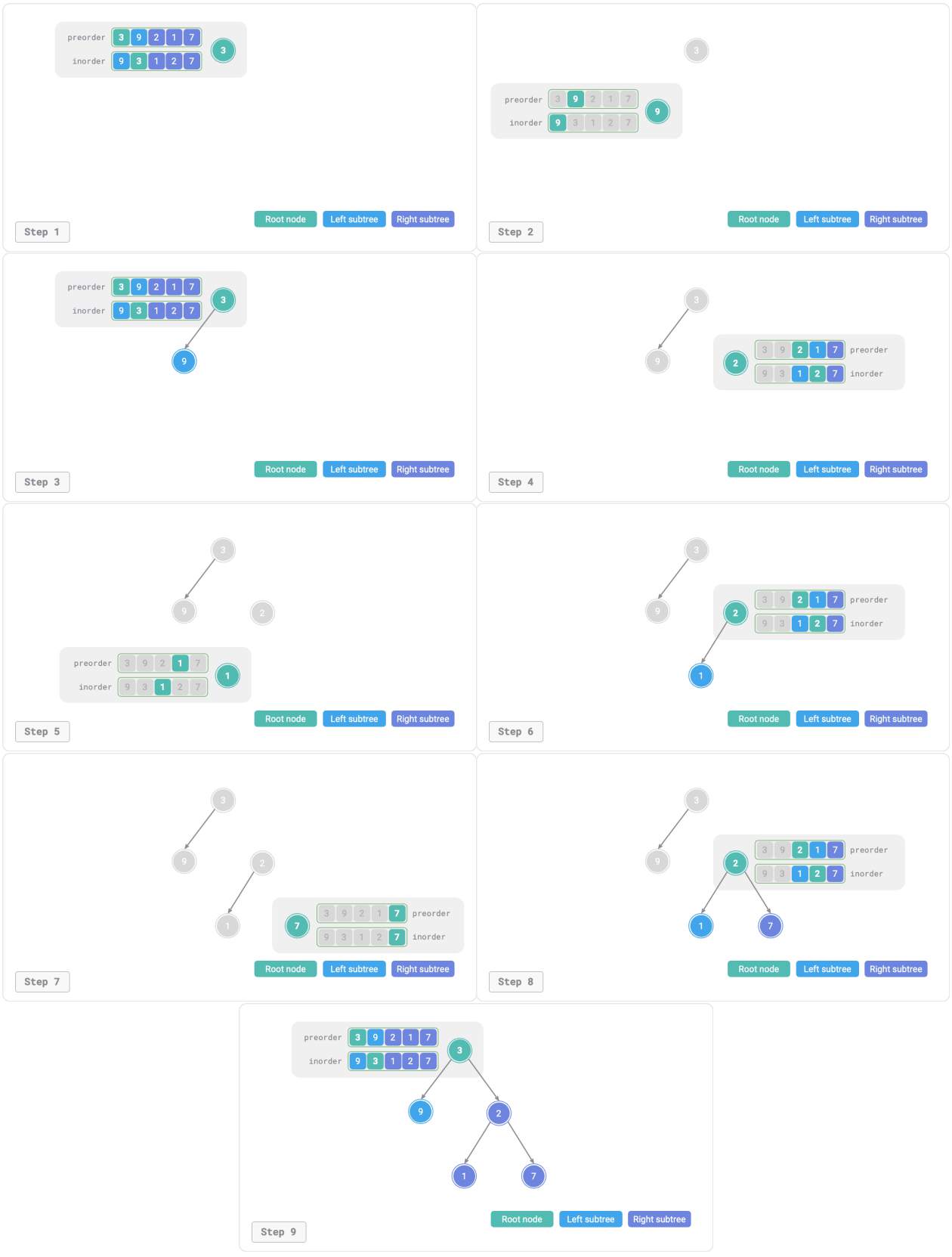


Figure 12-8 Recursive process of building a binary tree

The division results of the preorder traversal `preorder` and inorder traversal `inorder` within each recursive function are shown in Figure 12-9.

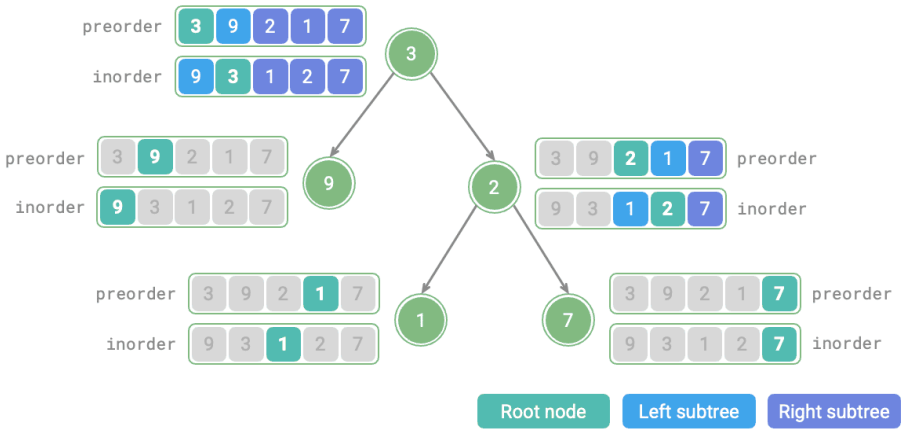


Figure 12-9 Division results in each recursive function

Let the number of nodes in the tree be n. Initializing each node (executing one recursive function `dfs()`) takes $O(1)$ time. **Therefore, the overall time complexity is $O(n)$.**

The hash table stores the mapping from `inorder` elements to their indices, with a space complexity of $O(n)$. In the worst case, when the binary tree degenerates into a linked list, the recursion depth reaches n, using $O(n)$ stack frame space. **Therefore, the overall space complexity is $O(n)$.**

12.4 Hanota Problem

In merge sort and building binary trees, we decompose the original problem into two subproblems, each half the size of the original problem. However, for the hanota problem, we adopt a different decomposition strategy.

Question

Given three pillars, denoted as `A`, `B`, and `C`. Initially, pillar `A` has n discs stacked on it, arranged from top to bottom in ascending order of size. Our task is to move these n discs to pillar `C` while maintaining their original order (as shown in Figure 12-10). The following rules must be followed when moving the discs.

1. A disc can only be taken from the top of one pillar and placed on top of another pillar.
2. Only one disc can be moved at a time.
3. A smaller disc must always be on top of a larger disc.

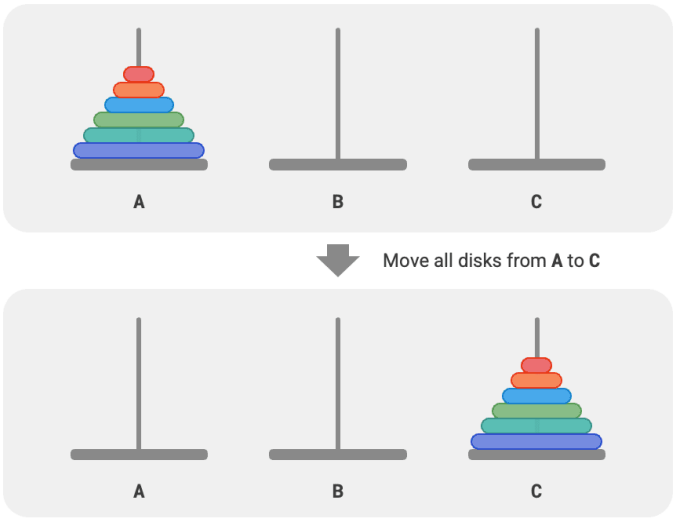


Figure 12-10 Example of the hanota problem

We denote the hanota problem of size i as $f(i)$. For example, $f(3)$ represents moving 3 discs from `A` to `C`.

1. Considering the Base Cases

As shown in Figure 12-11, for problem $f(1)$, when there is only one disc, we can move it directly from `A` to `C`.

Figure 12-11 Solution for a problem of size 1

As shown in Figure 12-12, for problem $f(2)$, when there are two discs, **since we must always keep the smaller disc on top of the larger disc, we need to use `B` to assist in the move.**

1. First, move the smaller disc from `A` to `B`.
2. Then move the larger disc from `A` to `C`.
3. Finally, move the smaller disc from `B` to `C`.

Figure 12-12 Solution for a problem of size 2

The process of solving problem $f(2)$ can be summarized as: **moving two discs from `A` to `C` with the help of `B`**. Here, `C` is called the target pillar, and `B` is called the buffer pillar.

2. Subproblem Decomposition

For problem $f(3)$, when there are three discs, the situation becomes slightly more complex.

Since we already know the solutions to $f(1)$ and $f(2)$, we can think from a divide and conquer perspective, **treating the top two discs on `A` as a whole**, and execute the steps shown in Figure 12-13. This successfully moves the three discs from `A` to `C`.

1. Let `B` be the target pillar and `C` be the buffer pillar, and move two discs from `A` to `B`.
2. Move the remaining disc from `A` directly to `C`.
3. Let `C` be the target pillar and `A` be the buffer pillar, and move two discs from `B` to `C`.

Figure 12-13 Solution for a problem of size 3

Essentially, **we divide problem $f(3)$ into two subproblems $f(2)$ and one subproblem $f(1)$**. By solving these three subproblems in order, the original problem is solved. This shows that the subproblems are independent and their solutions can be merged.

From this, we can summarize the divide and conquer strategy for solving the hanota problem shown in Figure 12-14: divide the original problem $f(n)$ into two subproblems $f(n-1)$ and one subproblem $f(1)$, and solve these three subproblems in the following order.

1. Move $n-1$ discs from `A` to `B` with the help of `C`.
2. Move the remaining 1 disc directly from `A` to `C`.
3. Move $n-1$ discs from `B` to `C` with the help of `A`.

For these two subproblems $f(n-1)$, **we can recursively divide them in the same way** until reaching the smallest subproblem $f(1)$. The solution to $f(1)$ is known and requires only one move operation.

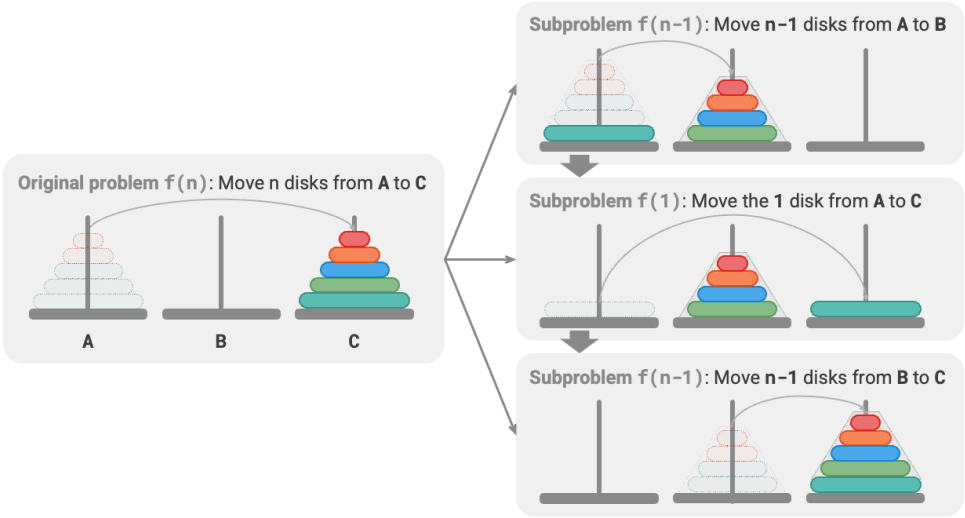


Figure 12-14 Divide and conquer strategy for solving the hanota problem

3. Code Implementation

In the code, we declare a recursive function `dfs(i, src, buf, tar)`, whose purpose is to move the top i discs from pillar `src` to target pillar `tar` with the help of buffer pillar `buf`:

```javascript
// === File: hanota.js ===

/* Move a disk */
function move(src, tar) {
    // Take out a disk from the top of src
    const pan = src.pop();
    // Place the disk on top of tar
    tar.push(pan);
}

/* Solve the Tower of Hanoi problem f(i) */
function dfs(i, src, buf, tar) {
    // If there is only one disk left in src, move it directly to tar
    if (i === 1) {
        move(src, tar);
        return;
    }
    // Subproblem f(i-1): move the top i-1 disks from src to buf using tar
    dfs(i - 1, src, tar, buf);
    // Subproblem f(1): move the remaining disk from src to tar
    move(src, tar);
    // Subproblem f(i-1): move the top i-1 disks from buf to tar using src
    dfs(i - 1, buf, src, tar);
}

/* Solve the Tower of Hanoi problem */
function solveHanota(A, B, C) {
    const n = A.length;
    // Move the top n disks from A to C using B
    dfs(n, A, B, C);
}
```

As shown in Figure 12-15, the hanota problem forms a recursion tree of height n, where each node represents a subproblem corresponding to an invocation of the `dfs()` function, **therefore the time complexity is $O(2^n)$ and the space complexity is $O(n)$**.

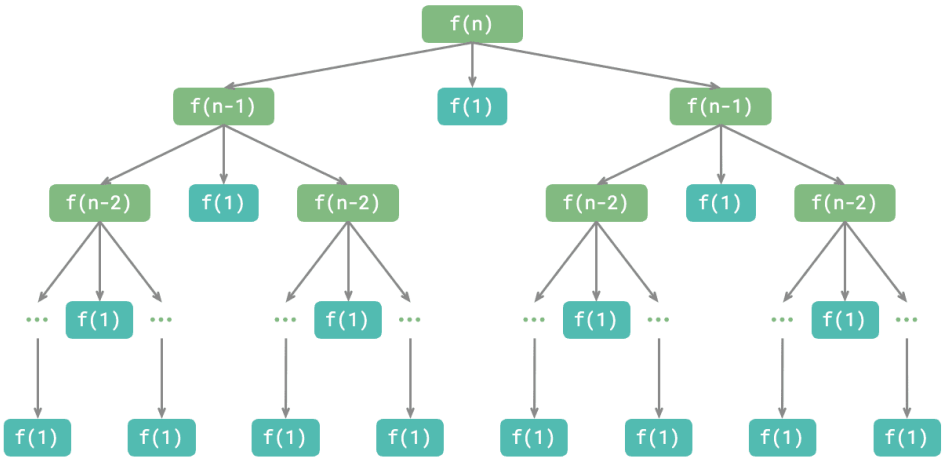


Figure 12-15 Recursion tree of the hanota problem

Quote

The hanota problem originates from an ancient legend. In a temple in ancient India, monks had three tall diamond pillars and 64 golden discs of different sizes. The monks continuously moved the discs, believing that when the last disc was correctly placed, the world would come to an end.

However, even if the monks moved one disc per second, it would take approximately $2^{64} \approx 1.84 \times 10^{19}$ seconds, which is about 5850 billion years, far exceeding current estimates of the age of the universe. Therefore, if this legend is true, we should not need to worry about the end of the world.

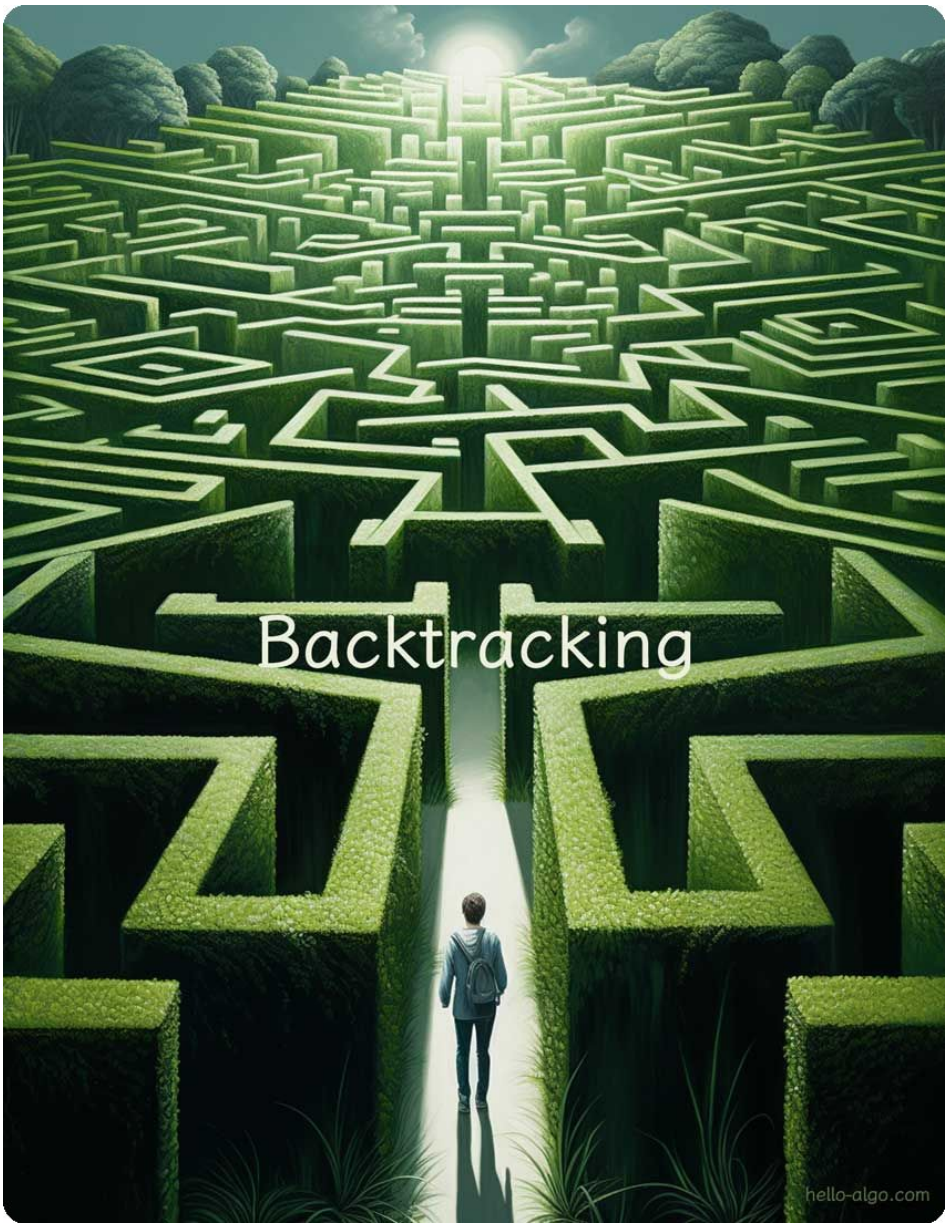
12.5 Summary

1. Key Review

- Divide and conquer is a common algorithm design strategy, consisting of two phases: divide (partition) and conquer (merge), typically implemented based on recursion.
- The criteria for determining whether a problem is a divide and conquer problem include: whether the problem can be decomposed, whether subproblems are independent, and whether subproblems can be merged.
- Merge sort is a typical application of the divide and conquer strategy. It recursively divides an array into two equal-length subarrays until only one element remains, then merges them layer by layer to complete the sorting.
- Introducing the divide and conquer strategy can often improve algorithm efficiency. On one hand, the divide and conquer strategy reduces the number of operations; on the other hand, it facilitates parallel optimization of the system after division.

- Divide and conquer can both solve many algorithmic problems and is widely applied in data structure and algorithm design, appearing everywhere.
- Compared to brute-force search, adaptive search is more efficient. Search algorithms with time complexity of $O(\log n)$ are typically implemented based on the divide and conquer strategy.
- Binary search is another typical application of divide and conquer. It does not include the step of merging solutions of subproblems. We can implement binary search through recursive divide and conquer.
- In the problem of building a binary tree, building the tree (original problem) can be divided into building the left subtree and right subtree (subproblems), which can be achieved by dividing the index intervals of the preorder and inorder traversals.
- In the hanota problem, a problem of size n can be divided into two subproblems of size $n-1$ and one subproblem of size 1. After solving these three subproblems in order, the original problem is solved.

Chapter 13. Backtracking



Abstract

We are like explorers in a maze, and may encounter difficulties on the path forward.

The power of backtracking allows us to start over, keep trying, and eventually find the exit leading to light.

13.1 Backtracking Algorithm

The backtracking algorithm is a method for solving problems through exhaustive search. Its core idea is to start from an initial state and exhaustively search all possible solutions. When a correct solution is found, it is recorded. This process continues until a solution is found or all possible choices have been tried without finding a solution.

The backtracking algorithm typically employs "depth-first search" to traverse the solution space. In the "Binary Tree" chapter, we mentioned that preorder, inorder, and postorder traversals all belong to depth-first search. Next, we will construct a backtracking problem using preorder traversal to progressively understand how the backtracking algorithm works.

Example 1
Given a binary tree, search and record all nodes with value 7, and return a list of these nodes.

For this problem, we perform a preorder traversal of the tree and check whether the current node's value is 7. If it is, we add the node to the result list `res`. The relevant implementation is shown in the following figure and code:

```javascript
// ═══ File: preorder_traversal_i_compact.js ═══

/* Preorder traversal: Example 1 */
function preOrder(root, res) {
    if (root === null) {
        return;
    }
    if (root.val === 7) {
        // Record solution
        res.push(root);
    }
    preOrder(root.left, res);
    preOrder(root.right, res);
}
```

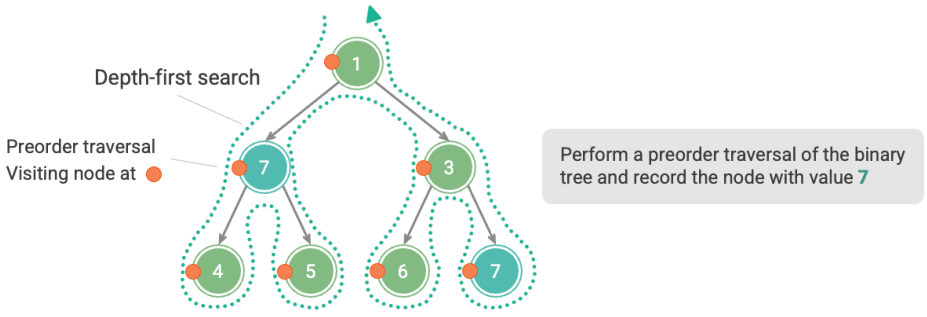


Figure 13-1 Search for nodes in preorder traversal

13.1.1 Attempt and Backtrack

The reason it is called a backtracking algorithm is that it employs "attempt" and "backtrack" strategies when searching the solution space. When the algorithm encounters a state where it cannot continue forward or cannot find a solution that satisfies the constraints, it will undo the previous choice, return to a previous state, and try other possible choices.

For Example 1, visiting each node represents an "attempt", while skipping over a leaf node or a function `return` from the parent node represents a "backtrack".

It is worth noting that **backtracking is not limited to function returns alone**. To illustrate this, let's extend Example 1 slightly.

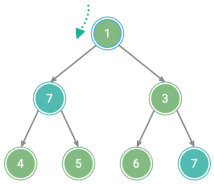
> **Example 2**
> In a binary tree, search all nodes with value 7, **and return the paths from the root node to these nodes**.

Based on the code from Example 1, we need to use a list `path` to record the visited node path. When we reach a node with value 7, we copy `path` and add it to the result list `res`. After traversal is complete, `res` contains all the solutions. The code is as follows:

```javascript
// === File: preorder_traversal_ii_compact.js ===

/* Preorder traversal: Example 2 */
function preOrder(root, path, res) {
    if (root === null) {
        return;
    }
    // Attempt
    path.push(root);
    if (root.val === 7) {
        // Record solution
        res.push([...path]);
    }
    preOrder(root.left, path, res);
    preOrder(root.right, path, res);
    // Backtrack
    path.pop();
}
```

In each "attempt", we record the path by adding the current node to `path`; before "backtracking", we need to remove the node from `path`, **to restore the state before this attempt**.

Observing the process shown in the following figure, **we can understand attempt and backtrack as "advance" and "undo"**, two operations that are the reverse of each other.



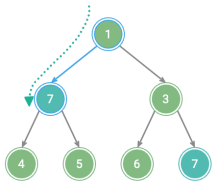
Attempt: Recursive traversal, update state

Backtrack: Restore state, function returns

`path = [1]`

`res = []`

Step 1



Attempt: Recursive traversal, update state

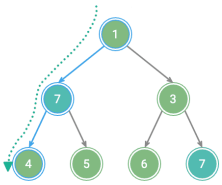
Backtrack: Restore state, function returns

`path = [1, 7]`

Encounter node 7, add the path to the result

`res = [[1, 7]]`

Step 2



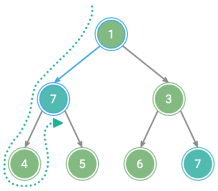
Attempt: Recursive traversal, update state

Backtrack: Restore state, function returns

`path = [1, 7, 4]`

`res = [[1, 7]]`

Step 3



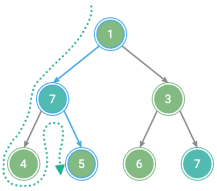
Attempt: Recursive traversal, update state

Backtrack: Restore state, function returns

`path = [1, 7]`

`res = [[1, 7]]`

Step 4



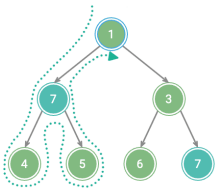
Attempt: Recursive traversal, update state

Backtrack: Restore state, function returns

`path = [1, 7, 5]`

`res = [[1, 7]]`

Step 5



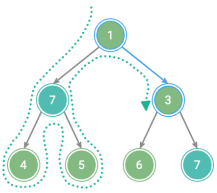
Attempt: Recursive traversal, update state

Backtrack: Restore state, function returns

`path = [1]`

`res = [[1, 7]]`

Step 6



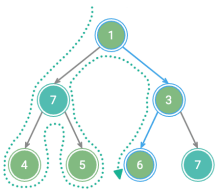
Attempt: Recursive traversal, update state

Backtrack: Restore state, function returns

`path = [1, 3]`

`res = [[1, 7]]`

Step 7



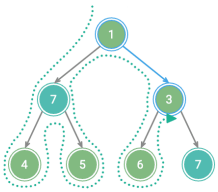
Attempt: Recursive traversal, update state

Backtrack: Restore state, function returns

`path = [1, 3, 6]`

`res = [[1, 7]]`

Step 8



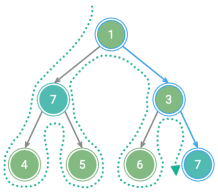
Attempt: Recursive traversal, update state

Backtrack: Restore state, function returns

`path = [1, 3]`

`res = [[1, 7]]`

Step 9



Attempt: Recursive traversal, update state

Backtrack: Restore state, function returns

`path = [1, 3, 7]`

Encounter node 7, add the path to the result

`res = [[1, 7], [1, 3, 7]]`

Step 10

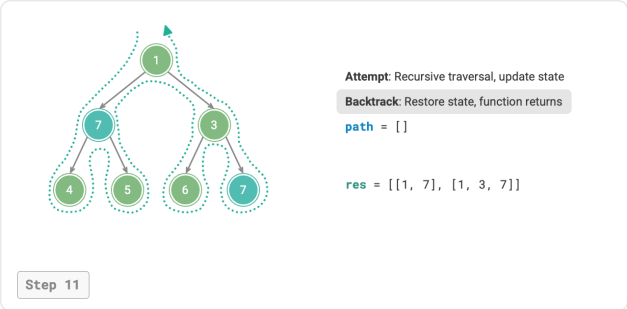


Figure 13-2 Attempt and backtrack

13.1.2 Pruning

Complex backtracking problems usually contain one or more constraints. **Constraints can typically be used for "pruning"**.

> **Example 3**
> In a binary tree, search all nodes with value 7 and return the paths from the root node to these nodes, **but require that the paths do not contain nodes with value** 3.

To satisfy the above constraints, **we need to add pruning operations**: during the search process, if we encounter a node with value 3, we return early and do not continue searching. The code is as follows:

```
// === File: preorder_traversal_iii_compact.js ===

/* Preorder traversal: Example 3 */
function preOrder(root, path, res) {
    // Pruning
    if (root === null || root.val === 3) {
        return;
    }
    // Attempt
    path.push(root);
    if (root.val === 7) {
        // Record solution
        res.push([...path]);
    }
    preOrder(root.left, path, res);
    preOrder(root.right, path, res);
    // Backtrack
    path.pop();
}
```

"Pruning" is a vivid term. As shown in the following figure, during the search process, **we "prune" search branches that do not satisfy the constraints**, avoiding many meaningless attempts and thus improving search efficiency.

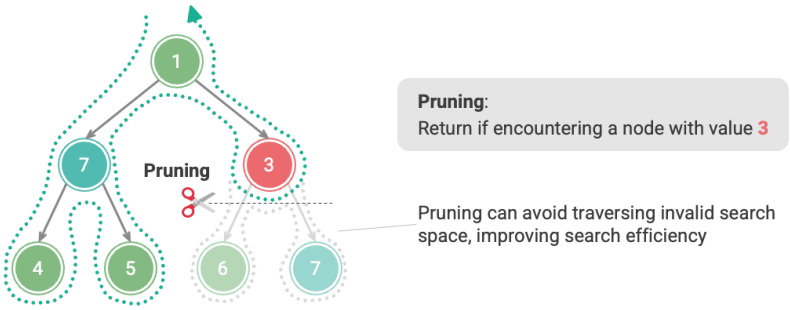


Figure 13-3 Pruning according to constraints

13.1.3 Framework Code

Next, we attempt to extract the main framework of backtracking's "attempt, backtrack, and pruning", to improve code generality.

In the following framework code, `state` represents the current state of the problem, and `choices` represents the choices available in the current state:

```
/* Backtracking algorithm framework */
function backtrack(state, choices, res) {
    // Check if it is a solution
    if (isSolution(state)) {
        // Record the solution
        recordSolution(state, res);
        // Stop searching
        return;
    }
    // Traverse all choices
    for (let choice of choices) {
        // Pruning: check if the choice is valid
        if (isValid(state, choice)) {
            // Attempt: make a choice and update the state
            makeChoice(state, choice);
            backtrack(state, choices, res);
            // Backtrack: undo the choice and restore to the previous state
            undoChoice(state, choice);
        }
    }
}
```

Next, we solve Example 3 based on the framework code. The state `state` is the node traversal path, the choices `choices` are the left and right child nodes of the current node, and the result `res` is a list of paths:

```
// ═══ File: preorder_traversal_iii_template.js ═══

/* Check if the current state is a solution */
function isSolution(state) {
```

```
    return state && state[state.length - 1]?.val === 7;
}

/* Record solution */
function recordSolution(state, res) {
    res.push([...state]);
}

/* Check if the choice is valid under the current state */
function isValid(state, choice) {
    return choice !== null && choice.val !== 3;
}

/* Update state */
function makeChoice(state, choice) {
    state.push(choice);
}

/* Restore state */
function undoChoice(state) {
    state.pop();
}

/* Backtracking algorithm: Example 3 */
function backtrack(state, choices, res) {
    // Check if it is a solution
    if (isSolution(state)) {
        // Record solution
        recordSolution(state, res);
    }
    // Traverse all choices
    for (const choice of choices) {
        // Pruning: check if the choice is valid
        if (isValid(state, choice)) {
            // Attempt: make choice, update state
            makeChoice(state, choice);
            // Proceed to the next round of selection
            backtrack(state, [choice.left, choice.right], res);
            // Backtrack: undo choice, restore to previous state
            undoChoice(state);
        }
    }
}
```

As per the problem statement, we should continue searching after finding a node with value 7. **Therefore, we need to remove the `return` statement after recording the solution**. The following figure compares the search process with and without the `return` statement.

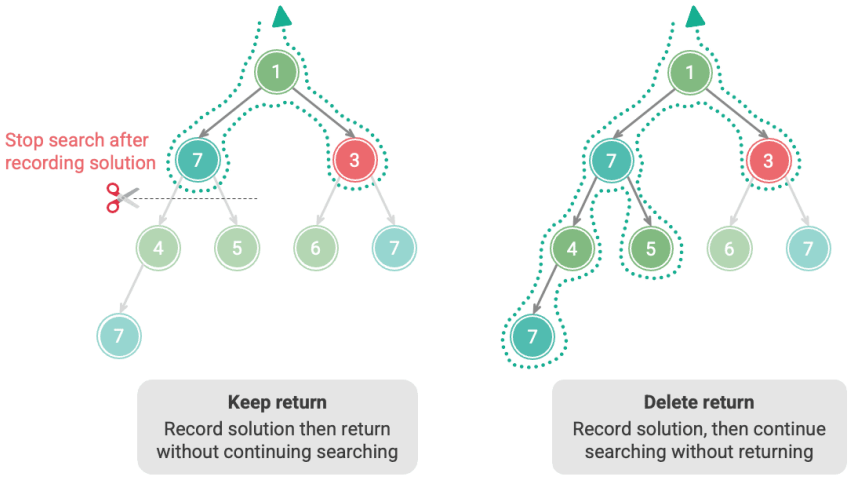


Figure 13-4 Comparison of search process with and without return statement

Compared to code based on preorder traversal, code based on the backtracking algorithm framework appears more verbose, but has better generality. In fact, **many backtracking problems can be solved within this framework**. We only need to define `state` and `choices` for the specific problem and implement each method in the framework.

13.1.4 Common Terminology

To analyze algorithmic problems more clearly, we summarize the meanings of common terminology used in backtracking algorithms and provide corresponding examples from Example 3, as shown in the following table.

Table 13-1 Common Backtracking Algorithm Terminology

Term	Definition	Example 3
Solution (solution)	A solution is an answer that satisfies the specific conditions of a problem; there may be one or more solutions	All paths from root to nodes with value 7 that satisfy the constraint
Constraint (constraint)	A constraint is a condition in the problem that limits the feasibility of solutions, typically used for pruning	Paths do not contain nodes with value 3
State (state)	State represents the situation of a problem at a certain moment, including the choices already made	The currently visited node path, i.e., the `path` list of nodes
Attempt (attempt)	An attempt is the process of exploring the solution space according to available choices, including making choices, updating state, and checking if it is a solution	Recursively visit left (right) child nodes, add nodes to `path`, check if node value is 7

Term	Definition	Example 3
Backtrack (backtracking)	Backtracking refers to undoing previous choices and returning to a previous state when encountering a state that does not satisfy constraints	Stop searching when passing over leaf nodes, ending node visits, or encountering nodes with value 3; function returns
Pruning (pruning)	Pruning is a method of avoiding meaningless search paths according to problem characteristics and constraints, which can improve search efficiency	When encountering a node with value 3, do not continue searching

Tip

The concepts of problem, solution, state, etc. are universal and are involved in divide-and-conquer, backtracking, dynamic programming, greedy and other algorithms.

13.1.5 Advantages and Limitations

The backtracking algorithm is essentially a depth-first search algorithm that tries all possible solutions until it finds one that satisfies the conditions. The advantage of this approach is that it can find all possible solutions, and with reasonable pruning operations, it achieves high efficiency.

However, when dealing with large-scale or complex problems, **the running efficiency of the backtracking algorithm may be unacceptable**.

- **Time**: The backtracking algorithm usually needs to traverse all possibilities in the solution space, and the time complexity can reach exponential or factorial order.
- **Space**: During recursive calls, the current state needs to be saved (such as paths, auxiliary variables used for pruning, etc.), and when the depth is large, the space requirement can become very large.

Nevertheless, **the backtracking algorithm is still the best solution for certain search problems and constraint satisfaction problems**. For these problems, since we cannot predict which choices will generate valid solutions, we must traverse all possible choices. In this case, **the key is how to optimize efficiency**. There are two common efficiency optimization methods.

- **Pruning**: Avoid searching paths that are guaranteed not to produce solutions, thereby saving time and space.
- **Heuristic search**: Introduce certain strategies or estimation values during the search process to prioritize searching paths that are most likely to produce valid solutions.

13.1.6 Typical Backtracking Examples

The backtracking algorithm can be used to solve many search problems, constraint satisfaction problems, and combinatorial optimization problems.

Search problems: The goal of these problems is to find solutions that satisfy specific conditions.

- Permutation problem: Given a set, find all possible permutations and combinations.
- Subset sum problem: Given a set and a target sum, find all subsets in the set whose elements sum to the target.
- Tower of Hanoi: Given three pegs and a series of disks of different sizes, move all disks from one peg to another, moving only one disk at a time, and never placing a larger disk on a smaller disk.

Constraint satisfaction problems: The goal of these problems is to find solutions that satisfy all constraints.

- N-Queens: Place n queens on an $n \times n$ chessboard such that they do not attack each other.
- Sudoku: Fill numbers 1 to 9 in a 9×9 grid such that each row, column, and 3×3 subgrid contains no repeated digits.
- Graph coloring: Given an undirected graph, color each vertex with the minimum number of colors such that adjacent vertices have different colors.

Combinatorial optimization problems: The goal of these problems is to find an optimal solution that satisfies certain conditions in a combinatorial space.

- 0-1 Knapsack: Given a set of items and a knapsack, each item has a value and weight. Under the knapsack capacity constraint, select items to maximize total value.
- Traveling Salesman Problem: Starting from a point in a graph, visit all other points exactly once and return to the starting point, finding the shortest path.
- Maximum Clique: Given an undirected graph, find the largest complete subgraph, i.e., a subgraph where any two vertices are connected by an edge.

Note that for many combinatorial optimization problems, backtracking is not the optimal solution.

- The 0-1 Knapsack problem is usually solved using dynamic programming to achieve higher time efficiency.
- The Traveling Salesman Problem is a famous NP-Hard problem; common solutions include genetic algorithms and ant colony algorithms.
- The Maximum Clique problem is a classical problem in graph theory and can be solved using heuristic algorithms such as greedy algorithms.

13.2 Permutations Problem

The permutations problem is a classic application of backtracking algorithms. It is defined as finding all possible arrangements of elements in a given collection (such as an array or string).

Table 13-2 shows several example datasets, including input arrays and their corresponding permutations.

Table 13-2 Permutations Examples

Input Array	All Permutations
$[1]$	$[1]$
$[1, 2]$	$[1, 2], [2, 1]$
$[1, 2, 3]$	$[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]$

13.2.1 Case with Distinct Elements

Question
Given an integer array with no duplicate elements, return all possible permutations.

From the perspective of backtracking algorithms, **we can imagine the process of generating permutations as the result of a series of choices**. Suppose the input array is $[1, 2, 3]$. If we first choose 1, then choose 3, and finally choose 2, we obtain the permutation $[1, 3, 2]$. Backtracking means undoing a choice and then trying other choices.

From the perspective of backtracking code, the candidate set `choices` consists of all elements in the input array, and the state `state` is the elements that have been chosen so far. Note that each element can only be chosen once, **therefore all elements in `state` should be unique**.

As shown in Figure 13-5, we can unfold the search process into a recursion tree, where each node in the tree represents the current state `state`. Starting from the root node, after three rounds of choices, we reach a leaf node, and each leaf node corresponds to a permutation.

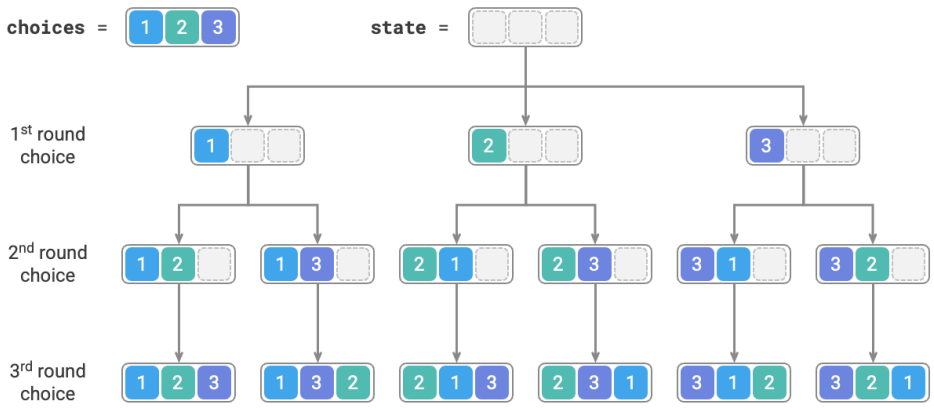


Figure 13-5 Recursion tree of permutations

1. Pruning Duplicate Choices

To ensure that each element is chosen only once, we consider introducing a boolean array `selected`, where `selected[i]` indicates whether `choices[i]` has been chosen. We implement the following pruning operation based on it.

- After making a choice `choice[i]`, we set `selected[i]` to True, indicating that it has been chosen.
- When traversing the candidate list `choices`, we skip all nodes that have been chosen, which is pruning.

As shown in Figure 13-6, suppose we choose 1 in the first round, 3 in the second round, and 2 in the third round. Then we need to prune the branch of element 1 in the second round and prune the branches of elements 1 and 3 in the third round.

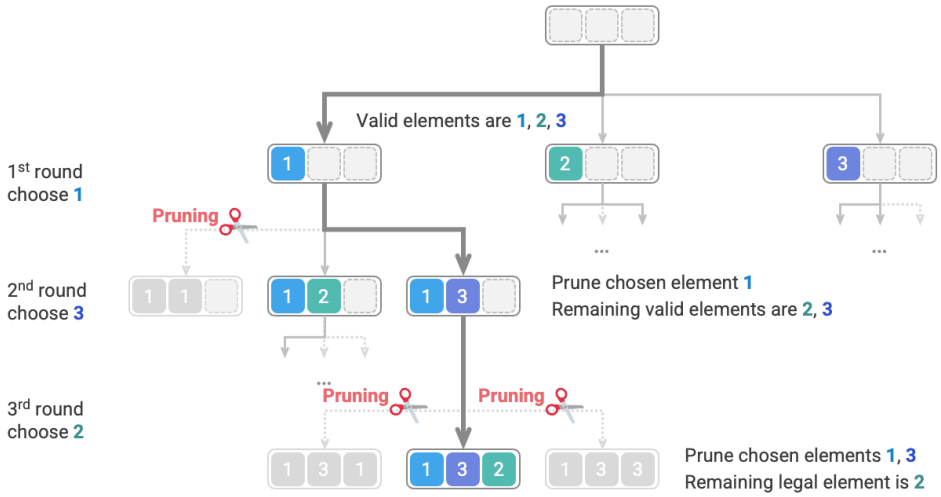


Figure 13-6 Pruning example of permutations

Observing the above figure, we find that this pruning operation reduces the search space size from $O(n^n)$ to $O(n!)$.

2. Code Implementation

After understanding the above information, we can fill in the blanks in the template code. To shorten the overall code, we do not implement each function in the template separately, but instead unfold them in the `backtrack()` function:

```
// ═══ File: permutations_i.js ═══

/* Backtracking algorithm: Permutations I */
function backtrack(state, choices, selected, res) {
    // When the state length equals the number of elements, record the solution
    if (state.length === choices.length) {
        res.push([...state]);
        return;
    }
    // Traverse all choices
    choices.forEach((choice, i) => {
        // Pruning: do not allow repeated selection of elements
        if (!selected[i]) {
            // Attempt: make choice, update state
```

```
            selected[i] = true;
            state.push(choice);
            // Proceed to the next round of selection
            backtrack(state, choices, selected, res);
            // Backtrack: undo choice, restore to previous state
            selected[i] = false;
            state.pop();
        }
    });
}

/* Permutations I */
function permutationsI(nums) {
    const res = [];
    backtrack([], nums, Array(nums.length).fill(false), res);
    return res;
}
```

13.2.2 Case with Duplicate Elements

> **Question**
> Given an integer array that **may contain duplicate elements**, return all unique permutations.

Suppose the input array is $[1, 1, 2]$. To distinguish the two duplicate elements 1, we denote the second 1 as $\hat{1}$.

As shown in Figure 13-7, the method described above generates permutations where half are duplicates.

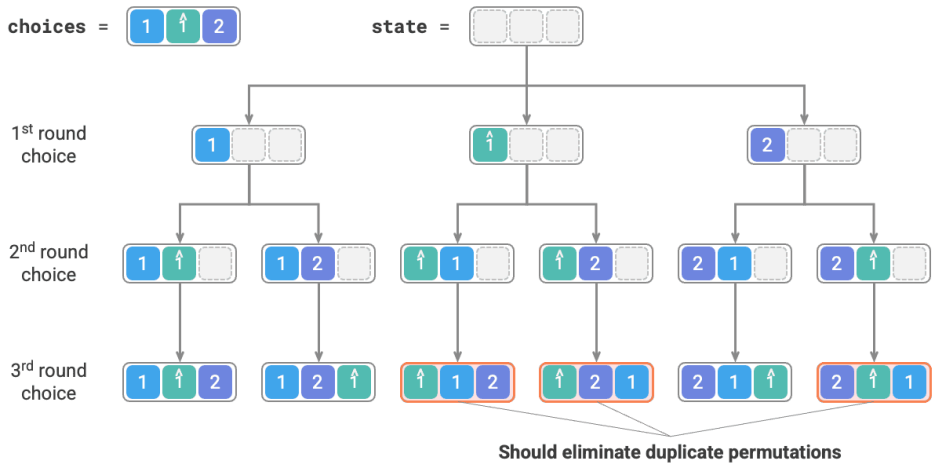


Figure 13-7 Duplicate permutations

So how do we remove duplicate permutations? The most direct approach is to use a hash set to directly deduplicate the permutation results. However, this is not elegant because **the search branches that**

generate duplicate permutations are unnecessary and should be identified and pruned early, which can further improve algorithm efficiency.

1. Pruning Duplicate Elements

Observe Figure 13-8. In the first round, choosing 1 or choosing $\hat{1}$ is equivalent. All permutations generated under these two choices are duplicates. Therefore, we should prune $\hat{1}$.

Similarly, after choosing 2 in the first round, the 1 and $\hat{1}$ in the second round also produce duplicate branches, so the second round's $\hat{1}$ should also be pruned.

Essentially, **our goal is to ensure that multiple equal elements are chosen only once in a certain round of choices**.

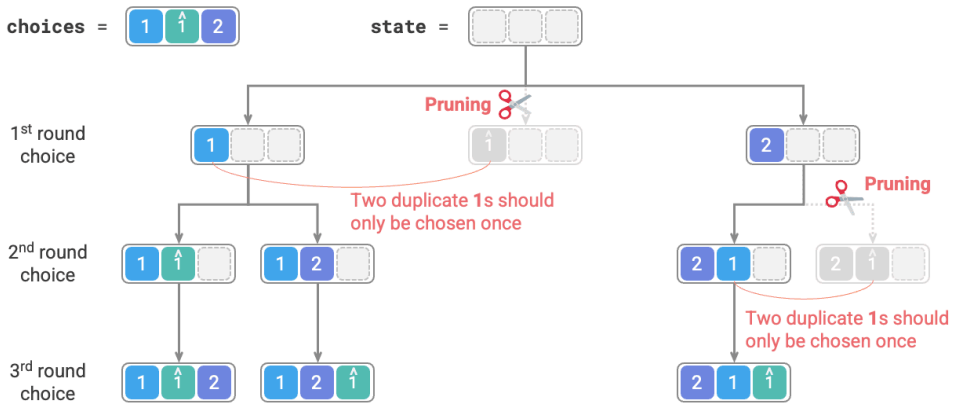


Figure 13-8 Pruning duplicate permutations

2. Code Implementation

Building on the code from the previous problem, we consider opening a hash set `duplicated` in each round of choices to record which elements have been tried in this round, and prune duplicate elements:

```
// ═══ File: permutations_ii.js ═══

/* Backtracking algorithm: Permutations II */
function backtrack(state, choices, selected, res) {
    // When the state length equals the number of elements, record the solution
    if (state.length === choices.length) {
        res.push([...state]);
        return;
    }
    // Traverse all choices
    const duplicated = new Set();
    choices.forEach((choice, i) => {
        // Pruning: do not allow repeated selection of elements and do not allow repeated
        ↪ selection of equal elements
```

```
        if (!selected[i] && !duplicated.has(choice)) {
            // Attempt: make choice, update state
            duplicated.add(choice); // Record the selected element value
            selected[i] = true;
            state.push(choice);
            // Proceed to the next round of selection
            backtrack(state, choices, selected, res);
            // Backtrack: undo choice, restore to previous state
            selected[i] = false;
            state.pop();
        }
    });
}

/* Permutations II */
function permutationsII(nums) {
    const res = [];
    backtrack([], nums, Array(nums.length).fill(false), res);
    return res;
}
```

Assuming elements are pairwise distinct, there are $n!$ (factorial) permutations of n elements. When recording results, we need to copy a list of length n, using $O(n)$ time. **Therefore, the time complexity is $O(n! \cdot n)$.**

The maximum recursion depth is n, using $O(n)$ stack frame space. `selected` uses $O(n)$ space. At most n `duplicated` sets exist simultaneously, using $O(n^2)$ space. **Therefore, the space complexity is $O(n^2)$.**

3. Comparison of Two Pruning Methods

Note that although both `selected` and `duplicated` are used for pruning, they have different objectives.

- **Pruning duplicate choices**: There is only one `selected` throughout the entire search process. It records which elements are included in the current state, and its purpose is to prevent an element from appearing repeatedly in `state`.
- **Pruning duplicate elements**: Each round of choices (each `backtrack` function call) contains a `duplicated` set. It records which elements have been chosen in this round's iteration (the `for` loop), and its purpose is to ensure that equal elements are chosen only once.

Figure 13-9 shows the effective scope of the two pruning conditions. Note that each node in the tree represents a choice, and the nodes on the path from the root to a leaf node form a permutation.

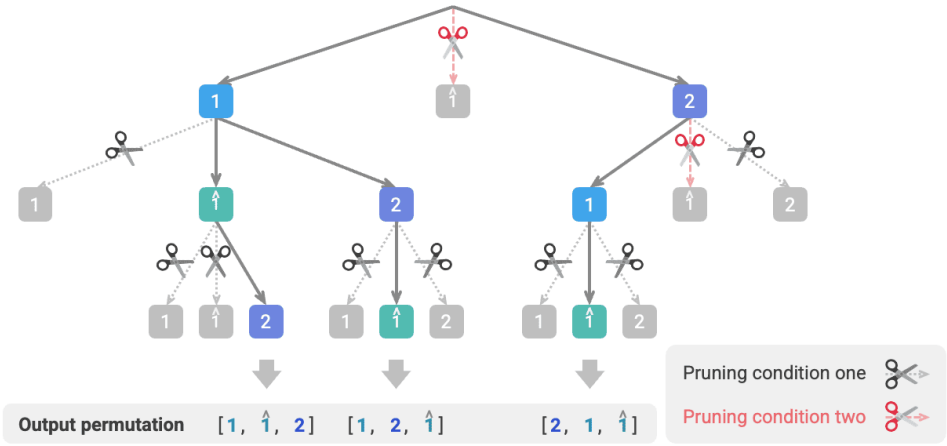


Figure 13-9 Effective scope of two pruning conditions

13.3 Subset-Sum Problem

13.3.1 Without Duplicate Elements

Question

Given a positive integer array `nums` and a target positive integer `target`, find all possible combinations where the sum of elements in the combination equals `target`. The given array has no duplicate elements, and each element can be selected multiple times. Return these combinations in list form, where the list should not contain duplicate combinations.

For example, given the set $\{3, 4, 5\}$ and target integer 9, the solutions are $\{3, 3, 3\}, \{4, 5\}$. Note the following two points:

- Elements in the input set can be selected repeatedly without limit.
- Subsets do not distinguish element order; for example, $\{4, 5\}$ and $\{5, 4\}$ are the same subset.

1. Reference to Full Permutation Solution

Similar to the full permutation problem, we can imagine the process of generating subsets as a series of choices, and update the "sum of elements" in real-time during the selection process. When the sum equals `target`, we record the subset to the result list.

Unlike the full permutation problem, **elements in this problem's set can be selected unlimited times**, so we do not need to use a `selected` boolean list to track whether an element has been selected. We can make minor modifications to the full permutation code and initially obtain the solution:

```javascript
// === File: subset_sum_i_naive.js ===

/* Backtracking algorithm: Subset sum I */
function backtrack(state, target, total, choices, res) {
    // When the subset sum equals target, record the solution
    if (total === target) {
        res.push([...state]);
        return;
    }
    // Traverse all choices
    for (let i = 0; i < choices.length; i++) {
        // Pruning: if the subset sum exceeds target, skip this choice
        if (total + choices[i] > target) {
            continue;
        }
        // Attempt: make choice, update element sum total
        state.push(choices[i]);
        // Proceed to the next round of selection
        backtrack(state, target, total + choices[i], choices, res);
        // Backtrack: undo choice, restore to previous state
        state.pop();
    }
}

/* Solve subset sum I (including duplicate subsets) */
function subsetSumINaive(nums, target) {
    const state = []; // State (subset)
    const total = 0; // Subset sum
    const res = []; // Result list (subset list)
    backtrack(state, target, total, nums, res);
    return res;
}
```

When we input array $[3, 4, 5]$ and target element 9 to the above code, the output is $[3, 3, 3]$, $[4, 5]$, $[5, 4]$. **Although we successfully find all subsets that sum to 9, there are duplicate subsets $[4, 5]$ and $[5, 4]$.**

This is because the search process distinguishes the order of selections, but subsets do not distinguish selection order. As shown in Figure 13-10, selecting 4 first and then 5 versus selecting 5 first and then 4 are different branches, but they correspond to the same subset.

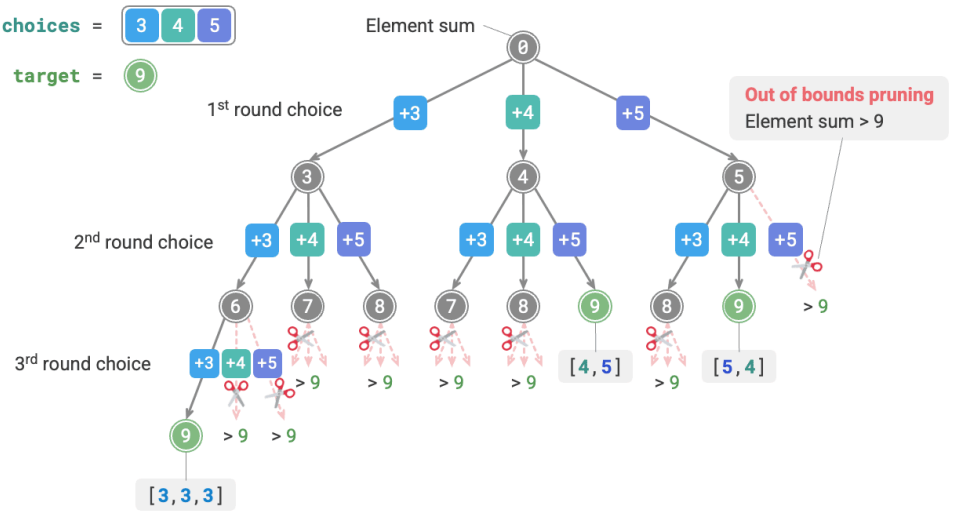


Figure 13-10 Subset search and boundary pruning

To eliminate duplicate subsets, **one straightforward idea is to deduplicate the result list**. However, this approach is very inefficient for two reasons:

- When there are many array elements, especially when `target` is large, the search process generates many duplicate subsets.
- Comparing subsets (arrays) is very time-consuming, requiring sorting the arrays first, then comparing each element in them.

2. Pruning Duplicate Subsets

We consider deduplication through pruning during the search process. Observing Figure 13-11, duplicate subsets occur when array elements are selected in different orders, as in the following cases:

1. When the first and second rounds select 3 and 4 respectively, all subsets containing these two elements are generated, denoted as $[3, 4, \dots]$.
2. Afterward, when the first round selects 4, **the second round should skip** 3, because the subset $[4, 3, \dots]$ generated by this choice is completely duplicate with the subset generated in step `1.`

In the search process, each level's choices are tried from left to right, so the rightmost branches are pruned more.

1. The first two rounds select 3 and 5, generating subset $[3, 5, \dots]$.
2. The first two rounds select 4 and 5, generating subset $[4, 5, \dots]$.
3. If the first round selects 5, **the second round should skip** 3 **and** 4, because subsets $[5, 3, \dots]$ and $[5, 4, \dots]$ are completely duplicate with the subsets described in steps `1.` and `2.`

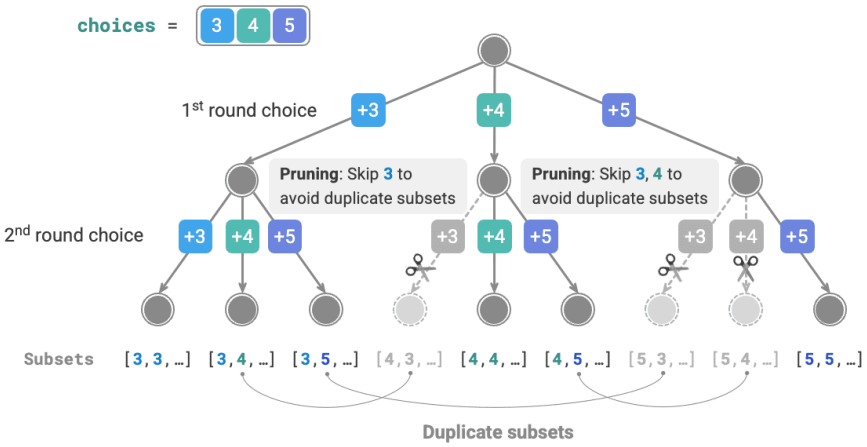


Figure 13-11 Different selection orders leading to duplicate subsets

In summary, given an input array $[x_1, x_2, \dots, x_n]$, let the selection sequence in the search process be $[x_{i_1}, x_{i_2}, \dots, x_{i_m}]$. This selection sequence must satisfy $i_1 \leq i_2 \leq \cdots \leq i_m$; **any selection sequence that does not satisfy this condition will cause duplicates and should be pruned**.

3. Code Implementation

To implement this pruning, we initialize a variable `start` to indicate the starting point of traversal. **After making choice x_i, set the next round to start traversal from index i.** This ensures that the selection sequence satisfies $i_1 \leq i_2 \leq \cdots \leq i_m$, guaranteeing subset uniqueness.

In addition, we have made the following two optimizations to the code:

- Before starting the search, first sort the array `nums`. When traversing all choices, **end the loop immediately when the subset sum exceeds `target`**, because subsequent elements are larger, and their subset sums must exceed `target`.
- Omit the element sum variable `total` and **use subtraction on `target` to track the sum of elements**. Record the solution when `target` equals 0.

```
// ═══ File: subset_sum_i.js ═══

/* Backtracking algorithm: Subset sum I */
function backtrack(state, target, choices, start, res) {
    // When the subset sum equals target, record the solution
    if (target === 0) {
        res.push([...state]);
        return;
    }
    // Traverse all choices
    // Pruning 2: start traversing from start to avoid generating duplicate subsets
    for (let i = start; i < choices.length; i++) {
        // Pruning 1: if the subset sum exceeds target, end the loop directly
        // This is because the array is sorted, and later elements are larger, so the subset sum
        ↪ will definitely exceed target
```

```
        if (target - choices[i] < 0) {
            break;
        }
        // Attempt: make choice, update target, start
        state.push(choices[i]);
        // Proceed to the next round of selection
        backtrack(state, target - choices[i], choices, i, res);
        // Backtrack: undo choice, restore to previous state
        state.pop();
    }
}

/* Solve subset sum I */
function subsetSumI(nums, target) {
    const state = []; // State (subset)
    nums.sort((a, b) => a - b); // Sort nums
    const start = 0; // Start point for traversal
    const res = []; // Result list (subset list)
    backtrack(state, target, nums, start, res);
    return res;
}
```

Figure 13-12 shows the complete backtracking process when array $[3, 4, 5]$ and target element 9 are input to the above code.

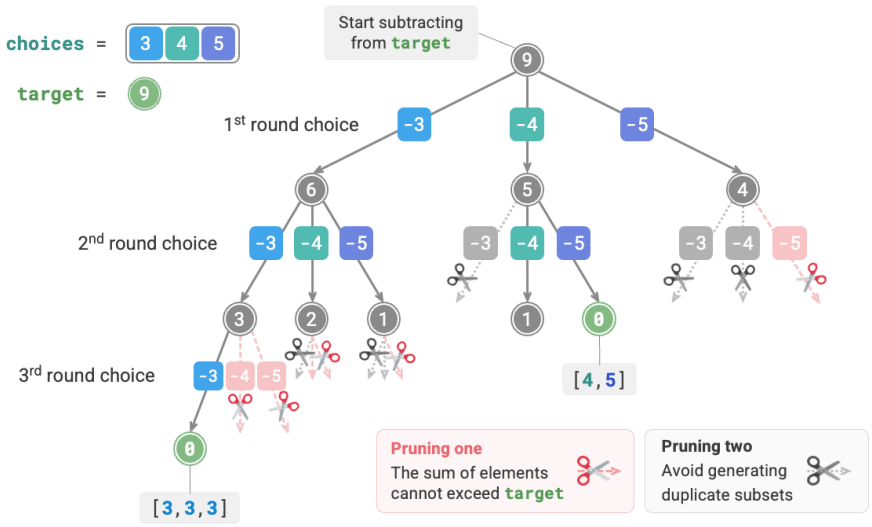


Figure 13-12 Subset-sum I backtracking process

13.3.2 With Duplicate Elements in Array

Question

Given a positive integer array `nums` and a target positive integer `target`, find all possible combinations where the sum of elements in the combination equals `target`. **The given array may contain duplicate elements, and each element can be selected at most once**. Return these combinations in list form, where the list should not contain duplicate combinations.

Compared to the previous problem, **the input array in this problem may contain duplicate elements**, which introduces new challenges. For example, given array $[4, \hat{4}, 5]$ and target element 9, the output of the existing code is $[4, 5]$, $[\hat{4}, 5]$, which contains duplicate subsets.

The reason for this duplication is that equal elements are selected multiple times in a certain round. In Figure 13-13, the first round has three choices, two of which are 4, creating two duplicate search branches that output duplicate subsets. Similarly, the two 4's in the second round also produce duplicate subsets.

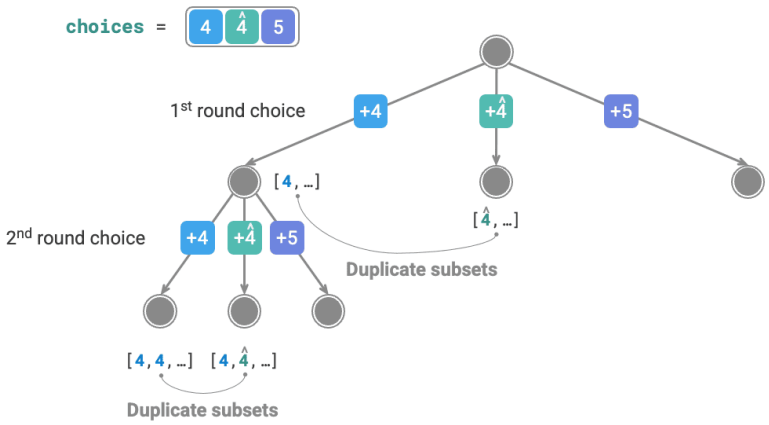


Figure 13-13 Duplicate subsets caused by equal elements

1. Pruning Equal Elements

To solve this problem, **we need to limit equal elements to be selected only once in each round**. The implementation is quite clever: since the array is already sorted, equal elements are adjacent. This means that in a certain round of selection, if the current element equals the element to its left, it means this element has already been selected, so we skip the current element directly.

At the same time, **this problem specifies that each array element can only be selected once**. Fortunately, we can also use the variable `start` to satisfy this constraint: after making choice x_i, set the next round to start traversal from index $i + 1$ onwards. This both eliminates duplicate subsets and avoids selecting elements multiple times.

2. Code Implementation

```javascript
// === File: subset_sum_ii.js ===

/* Backtracking algorithm: Subset sum II */
function backtrack(state, target, choices, start, res) {
    // When the subset sum equals target, record the solution
    if (target === 0) {
        res.push([...state]);
        return;
    }
    // Traverse all choices
    // Pruning 2: start traversing from start to avoid generating duplicate subsets
    // Pruning 3: start traversing from start to avoid repeatedly selecting the same element
    for (let i = start; i < choices.length; i++) {
        // Pruning 1: if the subset sum exceeds target, end the loop directly
        // This is because the array is sorted, and later elements are larger, so the subset sum
        ↪ will definitely exceed target
        if (target - choices[i] < 0) {
            break;
        }
        // Pruning 4: if this element equals the left element, it means this search branch is
        ↪ duplicate, skip it directly
        if (i > start && choices[i] === choices[i - 1]) {
            continue;
        }
        // Attempt: make choice, update target, start
        state.push(choices[i]);
        // Proceed to the next round of selection
        backtrack(state, target - choices[i], choices, i + 1, res);
        // Backtrack: undo choice, restore to previous state
        state.pop();
    }
}

/* Solve subset sum II */
function subsetSumII(nums, target) {
    const state = []; // State (subset)
    nums.sort((a, b) => a - b); // Sort nums
    const start = 0; // Start point for traversal
    const res = []; // Result list (subset list)
    backtrack(state, target, nums, start, res);
    return res;
}
```

Figure 13-14 shows the backtracking process for array $[4, 4, 5]$ and target element 9, which includes four types of pruning operations. Combine the illustration with the code comments to understand the entire search process and how each pruning operation works.

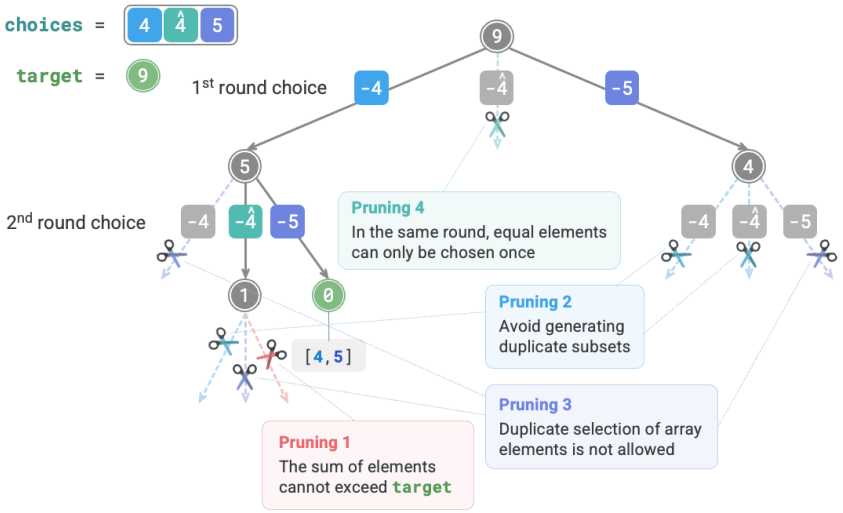


Figure 13-14 Subset-sum II backtracking process

13.4 N-Queens Problem

Question

According to the rules of chess, a queen can attack pieces that share the same row, column, or diagonal line. Given n queens and an $n \times n$ chessboard, find a placement scheme such that no two queens can attack each other.

As shown in Figure 13-15, when $n = 4$, there are two solutions that can be found. From the perspective of the backtracking algorithm, an $n \times n$ chessboard has n^2 squares, which provide all the choices `choices`. During the process of placing queens one by one, the chessboard state changes continuously, and the chessboard at each moment represents the state `state`.

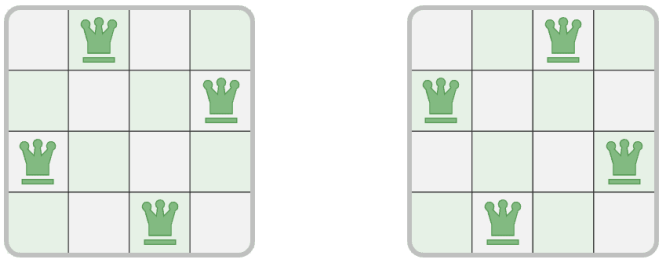


Figure 13-15 Solution to the 4-queens problem

Figure 13-16 illustrates the three constraints of this problem: **multiple queens cannot be in the same row, the same column, or on the same diagonal**. It is worth noting that diagonals are divided into two types: the main diagonal `\` and the anti-diagonal `/`.

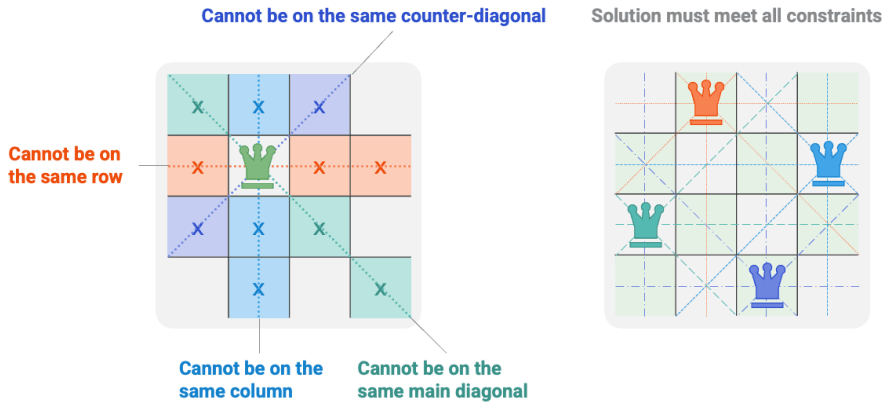


Figure 13-16 Constraints of the n-queens problem

1. Row-By-Row Placement Strategy

Since both the number of queens and the number of rows on the chessboard are n, we can easily derive a conclusion: **each row of the chessboard allows and only allows exactly one queen to be placed**.

This means we can adopt a row-by-row placement strategy: starting from the first row, place one queen in each row until the last row is completed.

Figure 13-17 shows the row-by-row placement process for the 4-queens problem. Due to space limitations, the figure only expands one search branch of the first row, and all schemes that do not satisfy the column constraint and diagonal constraints are pruned.

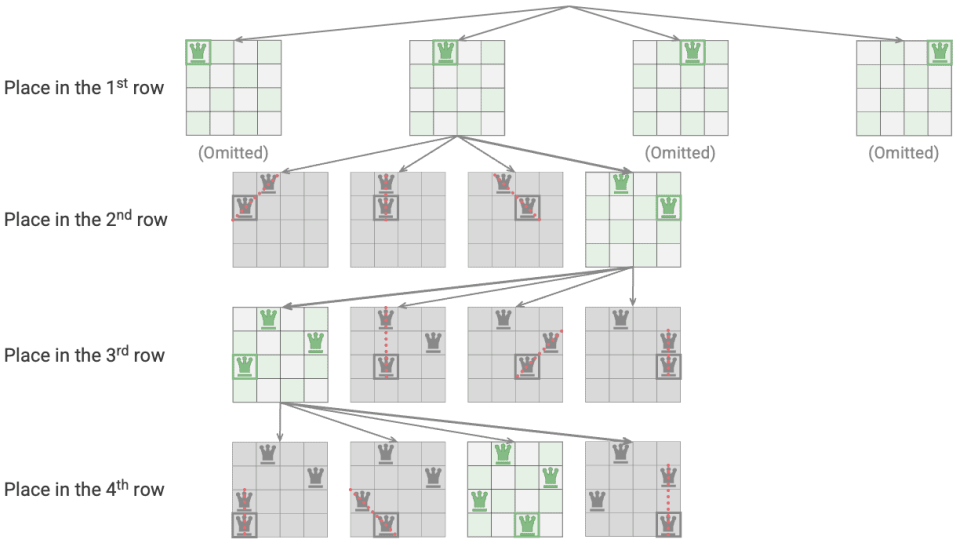


Figure 13-17 Row-by-row placement strategy

Essentially, **the row-by-row placement strategy serves a pruning function**, as it avoids all search branches where multiple queens appear in the same row.

2. Column and Diagonal Pruning

To satisfy the column constraint, we can use a boolean array `cols` of length n to record whether each column has a queen. Before each placement decision, we use `cols` to prune columns that already have queens, and dynamically update the state of `cols` during backtracking.

> **Tip**
> Please note that the origin of the matrix is located in the upper-left corner, where the row index increases from top to bottom, and the column index increases from left to right.

So how do we handle diagonal constraints? Consider a square on the chessboard with row and column indices (row, col). If we select a specific main diagonal in the matrix, we find that all squares on that diagonal have the same difference between their row and column indices, **meaning that $row - col$ is a constant value for all squares on the main diagonal**.

In other words, if two squares satisfy $row_1 - col_1 = row_2 - col_2$, they must be on the same main diagonal. Using this pattern, we can use the array `diags1` shown in Figure 13-18 to record whether there is a queen on each main diagonal.

Similarly, **for all squares on an anti-diagonal, the sum $row + col$ is a constant value**. We can likewise use the array `diags2` to handle anti-diagonal constraints.

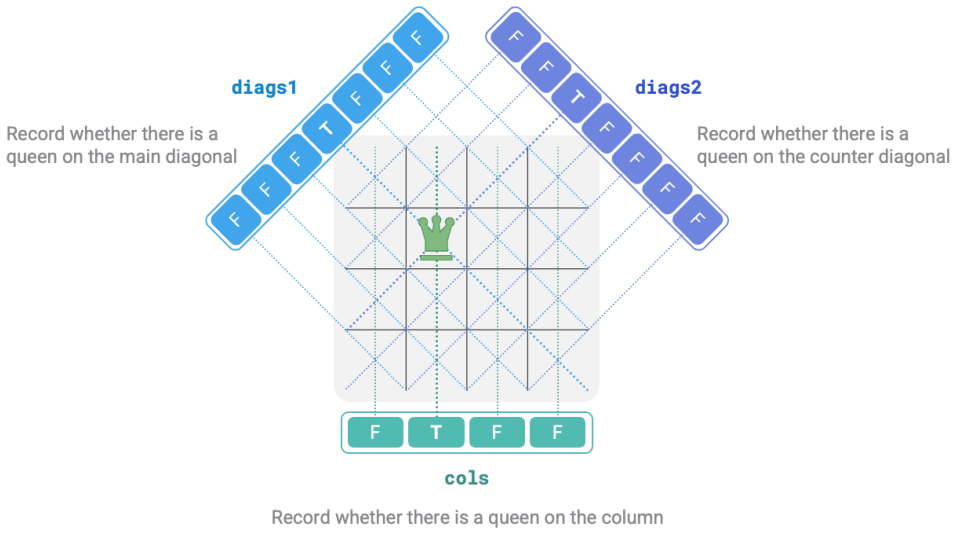


Figure 13-18 Handling column and diagonal constraints

3. Code Implementation

Please note that in an n-dimensional square matrix, the range of $row - col$ is $[-n + 1, n - 1]$, and the range of $row + col$ is $[0, 2n - 2]$. Therefore, the number of both main diagonals and anti-diagonals is $2n - 1$, meaning the length of both arrays `diags1` and `diags2` is $2n - 1$.

```javascript
// === File: n_queens.js ===

/* Backtracking algorithm: N queens */
function backtrack(row, n, state, res, cols, diags1, diags2) {
    // When all rows are placed, record the solution
    if (row === n) {
        res.push(state.map((row) => row.slice()));
        return;
    }
    // Traverse all columns
    for (let col = 0; col < n; col++) {
        // Calculate the main diagonal and anti-diagonal corresponding to this cell
        const diag1 = row - col + n - 1;
        const diag2 = row + col;
        // Pruning: do not allow queens to exist in the column, main diagonal, and anti-diagonal
        ↪ of this cell
        if (!cols[col] && !diags1[diag1] && !diags2[diag2]) {
            // Attempt: place the queen in this cell
            state[row][col] = 'Q';
            cols[col] = diags1[diag1] = diags2[diag2] = true;
            // Place the next row
            backtrack(row + 1, n, state, res, cols, diags1, diags2);
            // Backtrack: restore this cell to an empty cell
            state[row][col] = '#';
            cols[col] = diags1[diag1] = diags2[diag2] = false;
        }
    }
}

/* Solve N queens */
function nQueens(n) {
    // Initialize an n*n chessboard, where 'Q' represents a queen and '#' represents an empty cell
    const state = Array.from({ length: n }, () => Array(n).fill('#'));
    const cols = Array(n).fill(false); // Record whether there is a queen in the column
    const diags1 = Array(2 * n - 1).fill(false); // Record whether there is a queen on the main
    ↪ diagonal
    const diags2 = Array(2 * n - 1).fill(false); // Record whether there is a queen on the
    ↪ anti-diagonal
    const res = [];

    backtrack(0, n, state, res, cols, diags1, diags2);
    return res;
}
```

Placing n queens row by row, considering the column constraint, from the first row to the last row there are $n, n-1, \ldots, 2, 1$ choices, using $O(n!)$ time. When recording a solution, it is necessary to copy the matrix `state` and add it to `res`, and the copy operation uses $O(n^2)$ time. Therefore, **the overall time complexity is** $O(n! \cdot n^2)$. In practice, pruning based on diagonal constraints can also significantly reduce the search space, so the search efficiency is often better than the time complexity mentioned above.

The array `state` uses $O(n^2)$ space, and the arrays `cols`, `diags1`, and `diags2` each use $O(n)$ space. The maximum recursion depth is n, using $O(n)$ stack frame space. Therefore, **the space complexity is** $O(n^2)$.

13.5 Summary

1. Key Review

- The backtracking algorithm is fundamentally an exhaustive search method. It finds solutions that meet specified conditions by performing a depth-first traversal of the solution space. During the search process, when a solution satisfying the conditions is found, it is recorded. The search ends either after finding all solutions or when the traversal is complete.
- The backtracking algorithm search process consists of two parts: attempting and backtracking. It tries various choices through depth-first search. When encountering situations that violate constraints, it reverts the previous choice, returns to the previous state, and continues exploring other options. Attempting and backtracking are operations in opposite directions.
- Backtracking problems typically contain multiple constraints, which can be utilized to implement pruning operations. Pruning can terminate unnecessary search branches early, significantly improving search efficiency.
- The backtracking algorithm is primarily used to solve search problems and constraint satisfaction problems. While combinatorial optimization problems can be solved with backtracking, there are often more efficient or better-performing solutions available.
- The permutation problem aims to find all possible permutations of elements in a given set. We use an array to record whether each element has been selected, thereby pruning search branches that attempt to select the same element repeatedly, ensuring each element is selected exactly once.
- In the permutation problem, if the set contains duplicate elements, the final result will contain duplicate permutations. We need to impose a constraint so that equal elements can only be selected once per round, which is typically achieved using a hash set.
- The subset-sum problem aims to find all subsets of a given set that sum to a target value. Since the set is unordered but the search process outputs results in all orders, duplicate subsets are generated. We sort the data before backtracking and use a variable to indicate the starting point of each round's traversal, thereby pruning search branches that generate duplicate subsets.
- For the subset-sum problem, equal elements in the array produce duplicate sets. We leverage the precondition that the array is sorted by checking whether adjacent elements are equal to implement pruning, ensuring that equal elements can only be selected once per round.
- The n queens problem aims to find placements of n queens on an $n \times n$ chessboard such that no two queens can attack each other. The constraints of this problem include row constraints, column constraints, and main and anti-diagonal constraints. To satisfy row constraints, we adopt a row-by-row placement strategy, ensuring exactly one queen is placed in each row.
- The handling of column constraints and diagonal constraints is similar. For column constraints, we use an array to record whether each column has a queen, thereby indicating whether a selected cell is valid. For diagonal constraints, we use two arrays to separately record whether queens exist on each main or anti-diagonal. The challenge lies in finding the row-column index pattern that characterizes cells on the same main (anti-)diagonal.

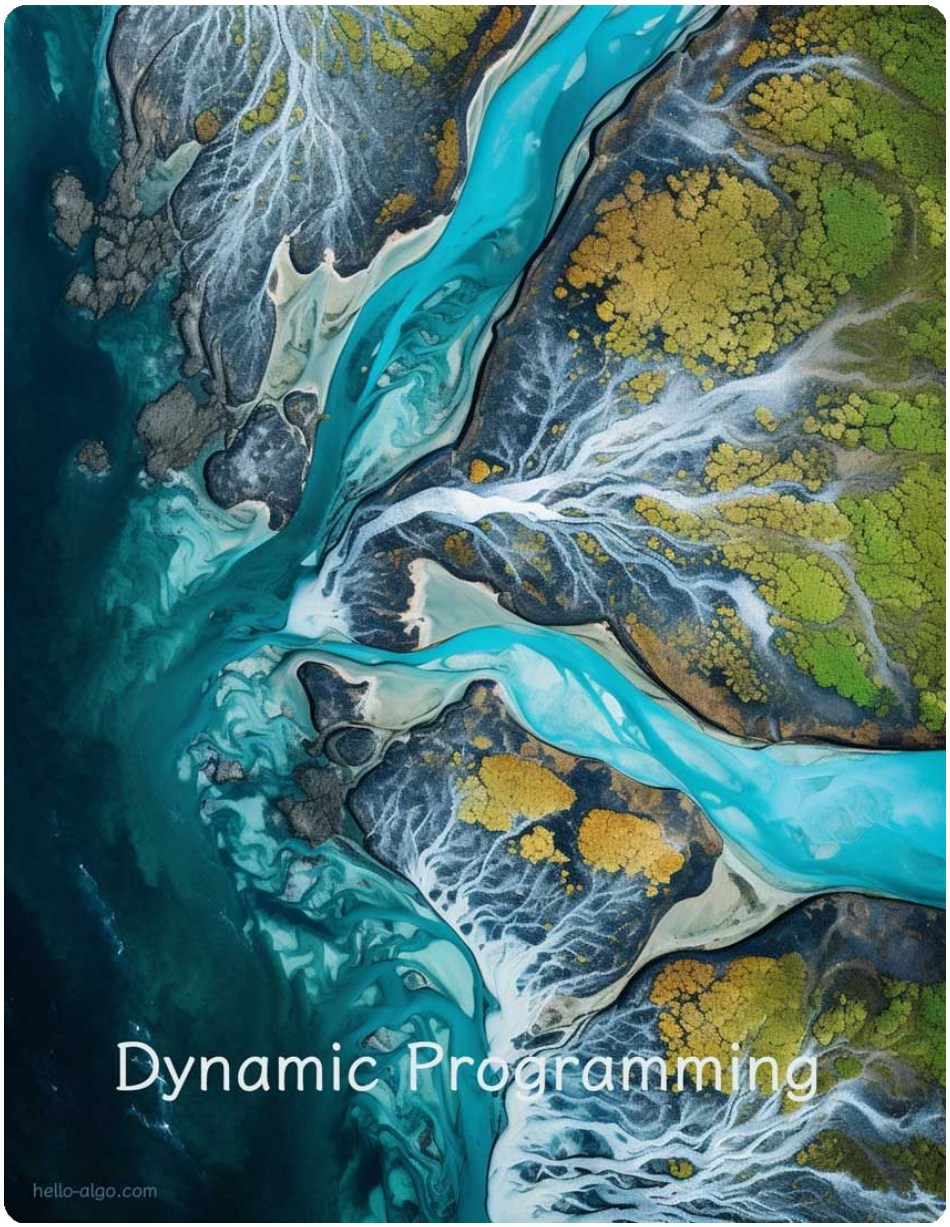
2. Q & A

Q: How should we understand the relationship between backtracking and recursion?

Overall, backtracking is an "algorithm strategy", while recursion is more like a "tool".

- The backtracking algorithm is typically implemented based on recursion. However, backtracking is one application scenario of recursion and represents the application of recursion in search problems.
- The structure of recursion embodies the "subproblem decomposition" problem-solving paradigm, commonly used to solve problems involving divide-and-conquer, backtracking, and dynamic programming (memoized recursion).

Chapter 14. Dynamic Programming



Abstract

Streams converge into rivers, rivers converge into the sea.

Dynamic programming gathers solutions to small problems into answers to large problems, step by step guiding us to the shore of problem-solving.

14.1 Introduction to Dynamic Programming

Dynamic programming is an important algorithmic paradigm that decomposes a problem into a series of smaller subproblems and avoids redundant computation by storing the solutions to subproblems, thereby significantly improving time efficiency.

In this section, we start with a classic example, first presenting its brute force backtracking solution, observing the overlapping subproblems within it, and then gradually deriving a more efficient dynamic programming solution.

Climbing stairs
Given a staircase with n steps, where you can climb 1 or 2 steps at a time, how many different ways are there to reach the top?

As shown in Figure 14-1, for a 3-step staircase, there are 3 different ways to reach the top.

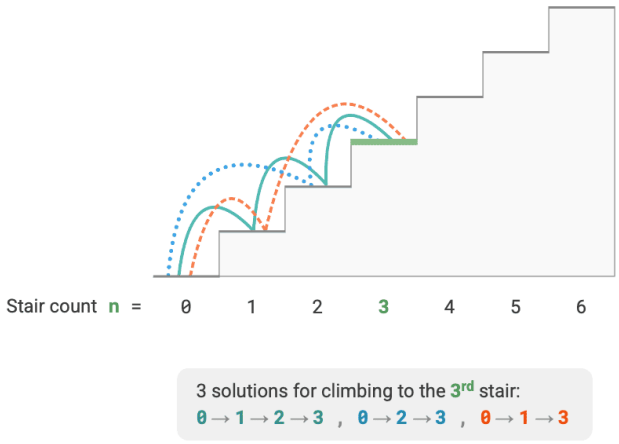


Figure 14-1 Number of ways to reach the 3rd step

The goal of this problem is to find the number of ways, **we can consider using backtracking to enumerate all possibilities**. Specifically, imagine climbing stairs as a multi-round selection process: starting from the ground, choosing to go up 1 or 2 steps in each round, incrementing the count by 1 whenever the top of the stairs is reached, and pruning when exceeding the top. The code is as follows:

```
// ═══ File: climbing_stairs_backtrack.js ═══

/* Backtracking */
function backtrack(choices, state, n, res) {
    // When climbing to the n-th stair, add 1 to the solution count
    if (state === n) res.set(0, res.get(0) + 1);
    // Traverse all choices
    for (const choice of choices) {
        // Pruning: not allowed to go beyond the n-th stair
        if (state + choice > n) continue;
```

```
        // Attempt: make choice, update state
        backtrack(choices, state + choice, n, res);
        // Backtrack
    }
}

/* Climbing stairs: Backtracking */
function climbingStairsBacktrack(n) {
    const choices = [1, 2]; // Can choose to climb up 1 or 2 stairs
    const state = 0; // Start climbing from the 0-th stair
    const res = new Map();
    res.set(0, 0); // Use res[0] to record the solution count
    backtrack(choices, state, n, res);
    return res.get(0);
}
```

14.1.1 Method 1: Brute Force Search

Backtracking algorithms typically do not explicitly decompose problems, but rather treat solving the problem as a series of decision steps, searching for all possible solutions through trial and pruning.

We can try to analyze this problem from the perspective of problem decomposition. Let the number of ways to climb to the i-th step be $dp[i]$, then $dp[i]$ is the original problem, and its subproblems include:

$$dp[i-1], dp[i-2], \dots, dp[2], dp[1]$$

Since we can only go up 1 or 2 steps in each round, when we stand on the i-th step, we could only have been on the $i-1$-th or $i-2$-th step in the previous round. In other words, we can only reach the i-th step from the $i-1$-th or $i-2$-th step.

This leads to an important conclusion: **the number of ways to climb to the $i-1$-th step plus the number of ways to climb to the $i-2$-th step equals the number of ways to climb to the i-th step**. The formula is as follows:

$$dp[i] = dp[i-1] + dp[i-2]$$

This means that in the stair climbing problem, there exists a recurrence relation among the subproblems, **the solution to the original problem can be constructed from the solutions to the subproblems**. Figure 14-2 illustrates this recurrence relation.

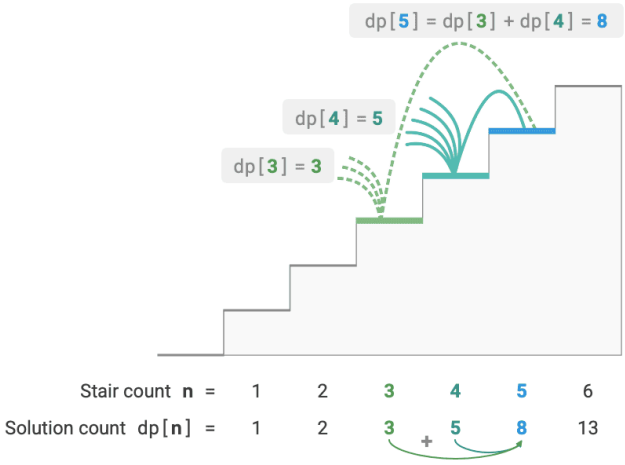


Figure 14-2 Recurrence relation for the number of ways

We can obtain a brute force search solution based on the recurrence formula. Starting from $dp[n]$, **recursively decompose a larger problem into the sum of two smaller problems**, until reaching the smallest subproblems $dp[1]$ and $dp[2]$ and returning. Among them, the solutions to the smallest subproblems are known, namely $dp[1] = 1$ and $dp[2] = 2$, representing 1 and 2 ways to climb to the 1st and 2nd steps, respectively.

Observe the following code, which, like standard backtracking code, belongs to depth-first search but is more concise:

```
// ═══ File: climbing_stairs_dfs.js ═══

/* Search */
function dfs(i) {
    // Known dp[1] and dp[2], return them
    if (i === 1 || i === 2) return i;
    // dp[i] = dp[i-1] + dp[i-2]
    const count = dfs(i - 1) + dfs(i - 2);
    return count;
}

/* Climbing stairs: Search */
function climbingStairsDFS(n) {
    return dfs(n);
}
```

Figure 14-3 shows the recursion tree formed by brute force search. For the problem $dp[n]$, the depth of its recursion tree is n, with a time complexity of $O(2^n)$. Exponential order represents explosive growth; if we input a relatively large n, we will fall into a long wait.

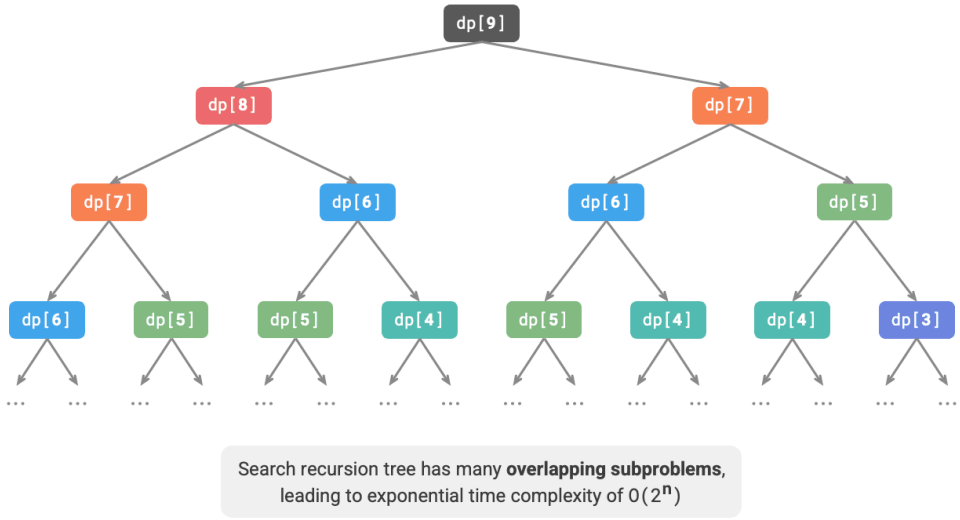


Figure 14-3 Recursion tree for climbing stairs

Observing the above figure, **the exponential time complexity is caused by "overlapping subproblems"**. For example, $dp[9]$ is decomposed into $dp[8]$ and $dp[7]$, and $dp[8]$ is decomposed into $dp[7]$ and $dp[6]$, both of which contain the subproblem $dp[7]$.

And so on, subproblems contain smaller overlapping subproblems, ad infinitum. The vast majority of computational resources are wasted on these overlapping subproblems.

14.1.2 Method 2: Memoization

To improve algorithm efficiency, **we want all overlapping subproblems to be computed only once**. For this purpose, we declare an array `mem` to record the solution to each subproblem and prune overlapping subproblems during the search process.

1. When computing $dp[i]$ for the first time, we record it in `mem[i]` for later use.
2. When we need to compute $dp[i]$ again, we can directly retrieve the result from `mem[i]`, thereby avoiding redundant computation of that subproblem.

The code is as follows:

```
// ═══ File: climbing_stairs_dfs_mem.js ═══

/* Memoization search */
function dfs(i, mem) {
    // Known dp[1] and dp[2], return them
    if (i === 1 || i === 2) return i;
    // If record dp[i] exists, return it directly
    if (mem[i] !== -1) return mem[i];
    // dp[i] = dp[i-1] + dp[i-2]
    const count = dfs(i - 1, mem) + dfs(i - 2, mem);
    // Record dp[i]
    mem[i] = count;
```

```
    return count;
}

/* Climbing stairs: Memoization search */
function climbingStairsDFSMem(n) {
    // mem[i] records the total number of solutions to climb to the i-th stair, -1 means no record
    const mem = new Array(n + 1).fill(-1);
    return dfs(n, mem);
}
```

Observe Figure 14-4, **after memoization, all overlapping subproblems only need to be computed once, optimizing the time complexity to** $O(n)$, which is a tremendous leap.

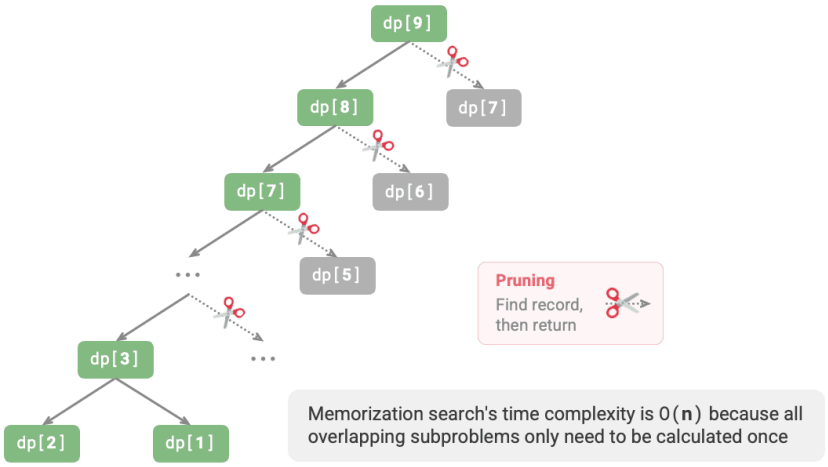


Figure 14-4 Recursion tree with memoization

14.1.3 Method 3: Dynamic Programming

Memoization is a "top-down" method: we start from the original problem (root node), recursively decompose larger subproblems into smaller ones, until reaching the smallest known subproblems (leaf nodes). Afterward, by backtracking, we collect the solutions to the subproblems layer by layer to construct the solution to the original problem.

In contrast, **dynamic programming is a "bottom-up" method**: starting from the solutions to the smallest subproblems, iteratively constructing solutions to larger subproblems until obtaining the solution to the original problem.

Since dynamic programming does not include a backtracking process, it only requires loop iteration for implementation and does not need recursion. In the following code, we initialize an array `dp` to store the solutions to subproblems, which serves the same recording function as the array `mem` in memoization:

```javascript
// === File: climbing_stairs_dp.js ===

/* Climbing stairs: Dynamic programming */
function climbingStairsDP(n) {
    if (n === 1 || n === 2) return n;
    // Initialize dp table, used to store solutions to subproblems
    const dp = new Array(n + 1).fill(-1);
    // Initial state: preset the solution to the smallest subproblem
    dp[1] = 1;
    dp[2] = 2;
    // State transition: gradually solve larger subproblems from smaller ones
    for (let i = 3; i <= n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }
    return dp[n];
}
```

Figure 14-5 simulates the execution process of the above code.

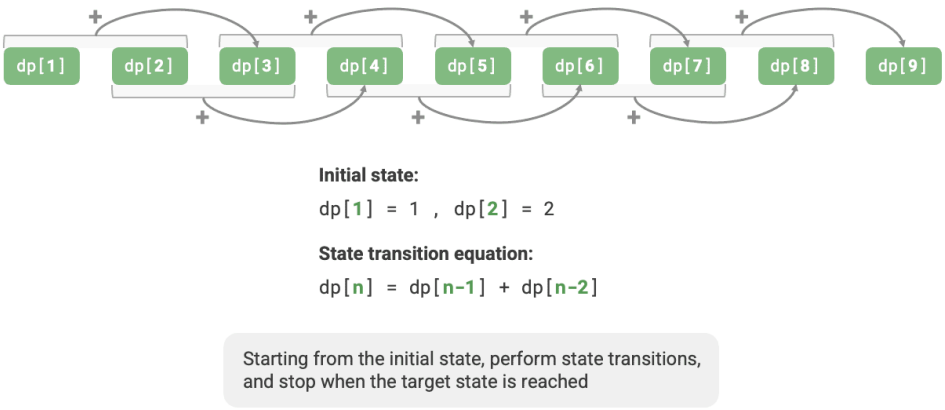


Figure 14-5 Dynamic programming process for climbing stairs

Like backtracking algorithms, dynamic programming also uses the "state" concept to represent specific stages of problem solving, with each state corresponding to a subproblem and its corresponding local optimal solution. For example, the state in the stair climbing problem is defined as the current stair step number i.

Based on the above content, we can summarize the commonly used terminology in dynamic programming.

- The array `dp` is called the dp table, where $dp[i]$ represents the solution to the subproblem corresponding to state i.
- The states corresponding to the smallest subproblems (the 1st and 2nd steps) are called initial states.
- The recurrence formula $dp[i] = dp[i-1] + dp[i-2]$ is called the state transition equation.

14.1.4 Space Optimization

Observant readers may have noticed that **since $dp[i]$ is only related to $dp[i-1]$ and $dp[i-2]$, we do not need to use an array `dp` to store the solutions to all subproblems**, but can simply use two variables to roll forward. The code is as follows:

```
// === File: climbing_stairs_dp.js ===

/* Climbing stairs: Space-optimized dynamic programming */
function climbingStairsDPComp(n) {
    if (n === 1 || n === 2) return n;
    let a = 1,
        b = 2;
    for (let i = 3; i <= n; i++) {
        const tmp = b;
        b = a + b;
        a = tmp;
    }
    return b;
}
```

Observing the above code, since the space occupied by the array `dp` is saved, the space complexity is reduced from $O(n)$ to $O(1)$.

In dynamic programming problems, the current state often depends only on a limited number of preceding states, allowing us to retain only the necessary states and save memory space through "dimension reduction". **This space optimization technique is called "rolling variable" or "rolling array"**.

14.2 Characteristics of Dynamic Programming Problems

In the previous section, we learned how dynamic programming solves the original problem by decomposing it into subproblems. In fact, subproblem decomposition is a general algorithmic approach, with different emphases in divide and conquer, dynamic programming, and backtracking.

- Divide and conquer algorithms recursively divide the original problem into multiple independent subproblems until the smallest subproblems are reached, and merge the solutions to the subproblems during backtracking to ultimately obtain the solution to the original problem.
- Dynamic programming also recursively decomposes problems, but the main difference from divide and conquer algorithms is that subproblems in dynamic programming are interdependent, and many overlapping subproblems appear during the decomposition process.
- Backtracking algorithms enumerate all possible solutions through trial and error, and avoid unnecessary search branches through pruning. The solution to the original problem consists of a series of decision steps, and we can regard the subsequence before each decision step as a subproblem.

In fact, dynamic programming is commonly used to solve optimization problems, which not only contain overlapping subproblems but also have two other major characteristics: optimal substructure and no aftereffects.

14.2.1 Optimal Substructure

We make a slight modification to the stair climbing problem to make it more suitable for demonstrating the concept of optimal substructure.

> **Climbing stairs with minimum cost**
> Given a staircase, where you can climb 1 or 2 steps at a time, and each step has a non-negative integer representing the cost you need to pay at that step. Given a non-negative integer array $cost$, where $cost[i]$ represents the cost at the i-th step, and $cost[0]$ is the ground (starting point). What is the minimum cost required to reach the top?

As shown in Figure 14-6, if the costs of the 1st, 2nd, and 3rd steps are 1, 10, and 1 respectively, then climbing from the ground to the 3rd step requires a minimum cost of 2.

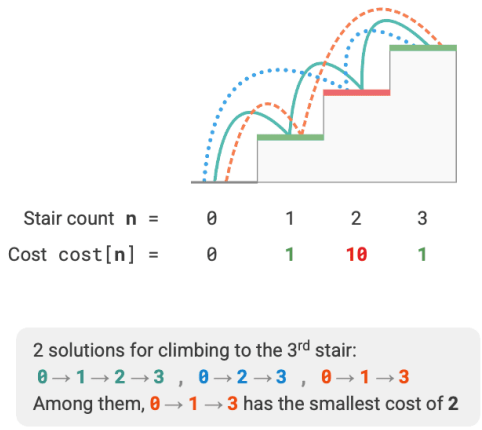


Figure 14-6 Minimum cost to climb to the 3rd step

Let $dp[i]$ be the accumulated cost of climbing to the i-th step. Since the i-th step can only come from the $i-1$-th or $i-2$-th step, $dp[i]$ can only equal $dp[i-1]+cost[i]$ or $dp[i-2]+cost[i]$. To minimize the cost, we should choose the smaller of the two:

$$dp[i] = \min(dp[i-1], dp[i-2]) + cost[i]$$

This leads us to the meaning of optimal substructure: **the optimal solution to the original problem is constructed from the optimal solutions to the subproblems**.

This problem clearly has optimal substructure: we select the better one from the optimal solutions to the two subproblems $dp[i-1]$ and $dp[i-2]$, and use it to construct the optimal solution to the original problem $dp[i]$.

So, does the stair climbing problem from the previous section have optimal substructure? Its goal is to find the number of ways, which seems to be a counting problem, but if we change the question: “Find

the maximum number of ways". We surprisingly discover that **although the problem before and after modification are equivalent, the optimal substructure has emerged**: the maximum number of ways for the n-th step equals the sum of the maximum number of ways for the $n-1$-th and $n-2$-th steps. Therefore, the interpretation of optimal substructure is quite flexible and will have different meanings in different problems.

According to the state transition equation and the initial states $dp[1] = cost[1]$ and $dp[2] = cost[2]$, we can obtain the dynamic programming code:

```javascript
// === File: min_cost_climbing_stairs_dp.js ===

/* Minimum cost climbing stairs: Dynamic programming */
function minCostClimbingStairsDP(cost) {
    const n = cost.length - 1;
    if (n === 1 || n === 2) {
        return cost[n];
    }
    // Initialize dp table, used to store solutions to subproblems
    const dp = new Array(n + 1);
    // Initial state: preset the solution to the smallest subproblem
    dp[1] = cost[1];
    dp[2] = cost[2];
    // State transition: gradually solve larger subproblems from smaller ones
    for (let i = 3; i <= n; i++) {
        dp[i] = Math.min(dp[i - 1], dp[i - 2]) + cost[i];
    }
    return dp[n];
}
```

Figure 14-7 shows the dynamic programming process for the above code.

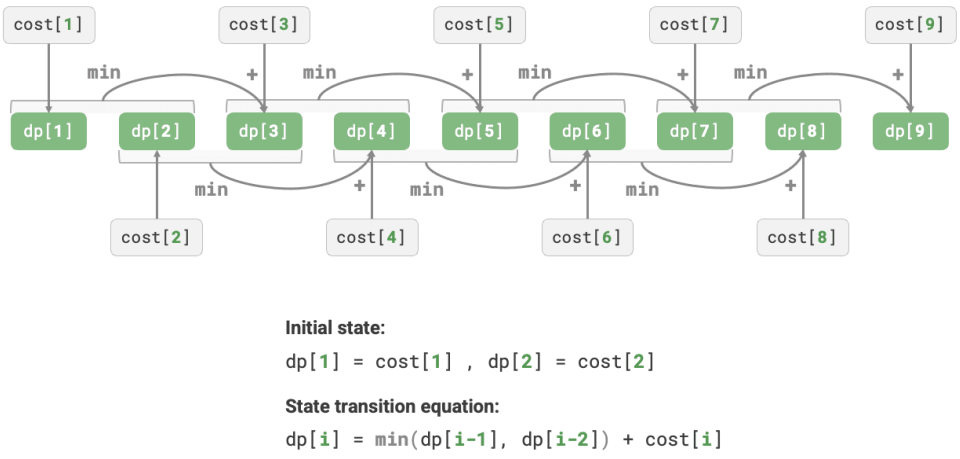


Figure 14-7 Dynamic programming process for climbing stairs with minimum cost

This problem can also be space-optimized, compressing from one dimension to zero, reducing the space complexity from $O(n)$ to $O(1)$:

```javascript
// ═══ File: min_cost_climbing_stairs_dp.js ═══

/* Minimum cost climbing stairs: Space-optimized dynamic programming */
function minCostClimbingStairsDPComp(cost) {
    const n = cost.length - 1;
    if (n === 1 || n === 2) {
        return cost[n];
    }
    let a = cost[1],
        b = cost[2];
    for (let i = 3; i <= n; i++) {
        const tmp = b;
        b = Math.min(a, tmp) + cost[i];
        a = tmp;
    }
    return b;
}
```

14.2.2 No Aftereffects

No aftereffects is one of the important characteristics that enable dynamic programming to solve problems effectively. Its definition is: **given a certain state, its future development is only related to the current state and has nothing to do with all past states**.

Taking the stair climbing problem as an example, given state i, it will develop into states $i+1$ and $i+2$, corresponding to jumping 1 step and jumping 2 steps, respectively. When making these two choices, we do not need to consider the states before state i, as they have no effect on the future of state i.

However, if we add a constraint to the stair climbing problem, the situation changes.

> **Climbing stairs with constraint**
> Given a staircase with n steps, where you can climb 1 or 2 steps at a time, **but you cannot jump 1 step in two consecutive rounds**. How many ways are there to climb to the top?

As shown in Figure 14-8, there are only 2 feasible ways to climb to the 3rd step. The way of jumping 1 step three consecutive times does not satisfy the constraint and is therefore discarded.

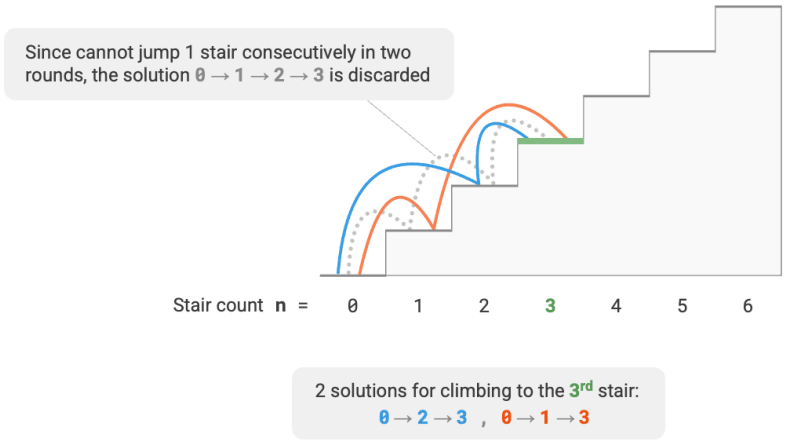


Figure 14-8 Number of ways to climb to the 3rd step with constraint

In this problem, if the previous round was a jump of 1 step, then the next round must jump 2 steps. This means that **the next choice cannot be determined solely by the current state (current stair step number), but also depends on the previous state (the stair step number from the previous round)**.

It is not difficult to see that this problem no longer satisfies no aftereffects, and the state transition equation $dp[i] = dp[i-1] + dp[i-2]$ also fails, because $dp[i-1]$ represents jumping 1 step in this round, but it includes many solutions where "the previous round was a jump of 1 step", which cannot be directly counted in $dp[i]$ to satisfy the constraint.

For this reason, we need to expand the state definition: **state $[i, j]$ represents being on the i-th step with the previous round having jumped j steps**, where $j \in \{1, 2\}$. This state definition effectively distinguishes whether the previous round was a jump of 1 step or 2 steps, allowing us to determine where the current state came from.

- When the previous round jumped 1 step, the round before that could only choose to jump 2 steps, i.e., $dp[i, 1]$ can only be transferred from $dp[i-1, 2]$.
- When the previous round jumped 2 steps, the round before that could choose to jump 1 step or 2 steps, i.e., $dp[i, 2]$ can be transferred from $dp[i-2, 1]$ or $dp[i-2, 2]$.

As shown in Figure 14-9, under this definition, $dp[i, j]$ represents the number of ways for state $[i, j]$. The state transition equation is then:

$$
\begin{cases}
dp[i, 1] = dp[i-1, 2] \\
dp[i, 2] = dp[i-2, 1] + dp[i-2, 2]
\end{cases}
$$

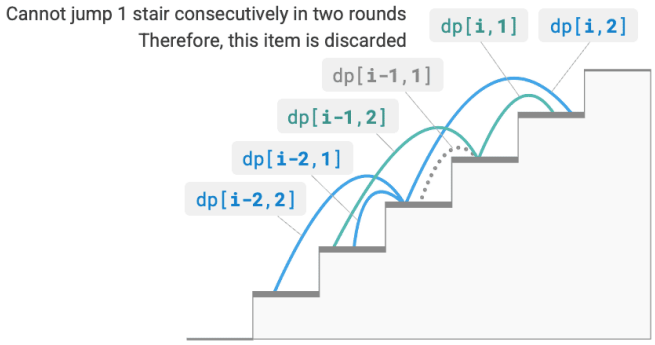


Figure 14-9 Recurrence relation considering constraints

Finally, return $dp[n, 1] + dp[n, 2]$, where the sum of the two represents the total number of ways to climb to the n-th step:

```
// === File: climbing_stairs_constraint_dp.js ===

/* Climbing stairs with constraint: Dynamic programming */
function climbingStairsConstraintDP(n) {
    if (n === 1 || n === 2) {
        return 1;
    }
    // Initialize dp table, used to store solutions to subproblems
    const dp = Array.from(new Array(n + 1), () => new Array(3));
    // Initial state: preset the solution to the smallest subproblem
    dp[1][1] = 1;
    dp[1][2] = 0;
    dp[2][1] = 0;
    dp[2][2] = 1;
    // State transition: gradually solve larger subproblems from smaller ones
    for (let i = 3; i <= n; i++) {
        dp[i][1] = dp[i - 1][2];
        dp[i][2] = dp[i - 2][1] + dp[i - 2][2];
    }
    return dp[n][1] + dp[n][2];
}
```

In the above case, since we only need to consider one more preceding state, we can still make the problem satisfy no aftereffects by expanding the state definition. However, some problems have very severe "aftereffects".

Climbing stairs with obstacle generation

Given a staircase with n steps, where you can climb 1 or 2 steps at a time. **It is stipulated that when climbing to the i-th step, the system will automatically place an obstacle on the $2i$-th step, and thereafter no round is allowed to jump to the $2i$-th step**. For example, if the first two rounds jump to the 2nd and 3rd steps, then afterwards you cannot jump to the 4th and 6th steps. How many ways are there to climb to the top?

In this problem, the next jump depends on all past states, because each jump places obstacles on higher steps, affecting future jumps. For such problems, dynamic programming is often difficult to solve.

In fact, many complex combinatorial optimization problems (such as the traveling salesman problem) do not satisfy no aftereffects. For such problems, we usually choose to use other methods, such as heuristic search, genetic algorithms, reinforcement learning, etc., to obtain usable local optimal solutions within a limited time.

14.3 Dynamic Programming Problem-Solving Approach

The previous two sections introduced the main characteristics of dynamic programming problems. Next, let us explore two more practical issues together.

1. How to determine whether a problem is a dynamic programming problem?
2. What is the complete process for solving a dynamic programming problem, and where should we start?

14.3.1 Problem Determination

Generally speaking, if a problem contains overlapping subproblems, optimal substructure, and satisfies no aftereffects, then it is usually suitable for solving with dynamic programming. However, it is difficult to directly extract these characteristics from the problem description. Therefore, we usually relax the conditions and **first observe whether the problem is suitable for solving with backtracking (exhaustive search)**.

Problems suitable for solving with backtracking usually satisfy the "decision tree model", which means the problem can be described using a tree structure, where each node represents a decision and each path represents a sequence of decisions.

In other words, if a problem contains an explicit concept of decisions, and the solution is generated through a series of decisions, then it satisfies the decision tree model and can usually be solved using backtracking.

On this basis, dynamic programming problems also have some "bonus points" for determination.

- The problem contains descriptions such as maximum (minimum) or most (least), indicating optimization.
- The problem's state can be represented using a list, multi-dimensional matrix, or tree, and a state has a recurrence relation with its surrounding states.

Correspondingly, there are also some "penalty points".

- The goal of the problem is to find all possible solutions, rather than finding the optimal solution.
- The problem description has obvious permutation and combination characteristics, requiring the return of specific multiple solutions.

If a problem satisfies the decision tree model and has relatively obvious "bonus points", we can assume it is a dynamic programming problem and verify it during the solving process.

14.3.2 Problem-Solving Steps

The problem-solving process for dynamic programming varies depending on the nature and difficulty of the problem, but generally follows these steps: describe decisions, define states, establish the dp table, derive state transition equations, determine boundary conditions, etc.

To illustrate the problem-solving steps more vividly, we use a classic problem "minimum path sum" as an example.

> **Question**
> Given an $n \times m$ two-dimensional grid `grid`, where each cell in the grid contains a non-negative integer representing the cost of that cell. A robot starts from the top-left cell and can only move down or right at each step until reaching the bottom-right cell. Return the minimum path sum from the top-left to the bottom-right.

Figure 14-10 shows an example where the minimum path sum for the given grid is 13.

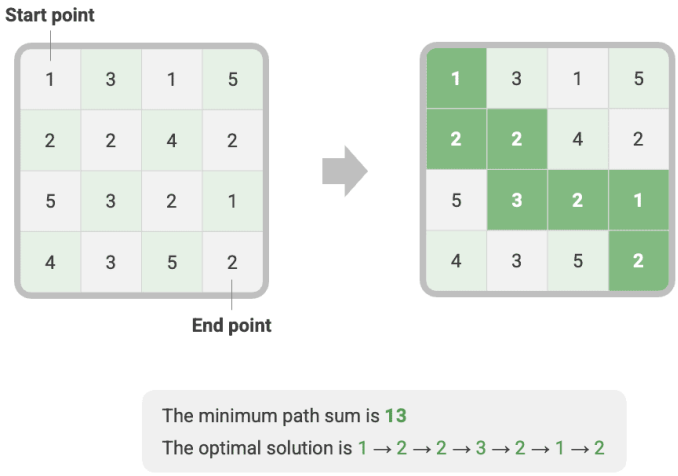


Figure 14-10 Minimum path sum example data

Step 1: Think about the decisions in each round, define the state, and thus obtain the dp table

The decision in each round of this problem is to move one step down or right from the current cell. Let the row and column indices of the current cell be $[i, j]$. After moving down or right, the indices become $[i + 1, j]$ or $[i, j + 1]$. Therefore, the state should include two variables, the row index and column index, denoted as $[i, j]$.

State $[i, j]$ corresponds to the subproblem: **the minimum path sum from the starting point** $[0, 0]$ **to** $[i, j]$, denoted as $dp[i, j]$.

From this, we obtain the two-dimensional dp matrix shown in Figure 14-11, whose size is the same as the input grid $grid$.

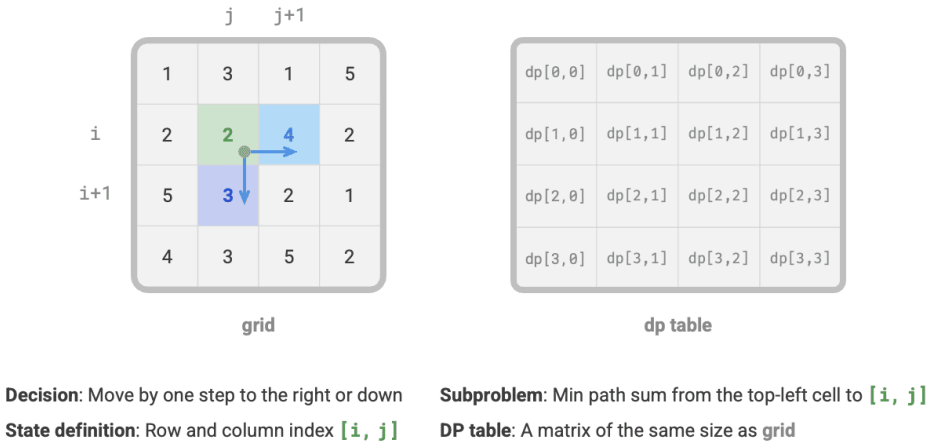


Figure 14-11 State definition and dp table

Note

The dynamic programming and backtracking processes can be described as a sequence of decisions, and the state consists of all decision variables. It should contain all variables describing the progress of problem-solving, and should contain sufficient information to derive the next state.

Each state corresponds to a subproblem, and we define a dp table to store the solutions to all subproblems. Each independent variable of the state is a dimension of the dp table. Essentially, the dp table is a mapping between states and solutions to subproblems.

Step 2: Identify the optimal substructure, and then derive the state transition equation

For state $[i, j]$, it can only be transferred from the cell above $[i-1, j]$ or the cell to the left $[i, j-1]$. Therefore, the optimal substructure is: the minimum path sum to reach $[i, j]$ is determined by the smaller of the minimum path sums of $[i, j-1]$ and $[i-1, j]$.

Based on the above analysis, the state transition equation shown in Figure 14-12 can be derived:

$$dp[i, j] = \min(dp[i-1, j], dp[i, j-1]) + grid[i, j]$$

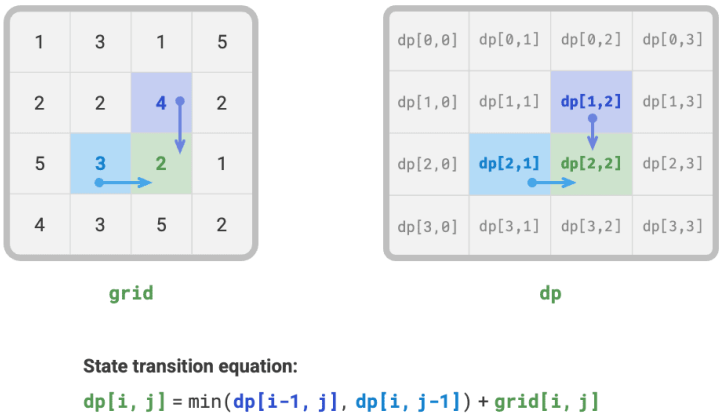


Figure 14-12 Optimal substructure and state transition equation

Note

Based on the defined dp table, think about the relationship between the original problem and subproblems, and find the method to construct the optimal solution to the original problem from the optimal solutions to the subproblems, which is the optimal substructure.

Once we identify the optimal substructure, we can use it to construct the state transition equation.

Step 3: Determine boundary conditions and state transition order

In this problem, states in the first row can only come from the state to their left, and states in the first column can only come from the state above them. Therefore, the first row $i = 0$ and first column $j = 0$ are boundary conditions.

As shown in Figure 14-13, since each cell is transferred from the cell to its left and the cell above it, we use loops to traverse the matrix, with the outer loop traversing rows and the inner loop traversing columns.

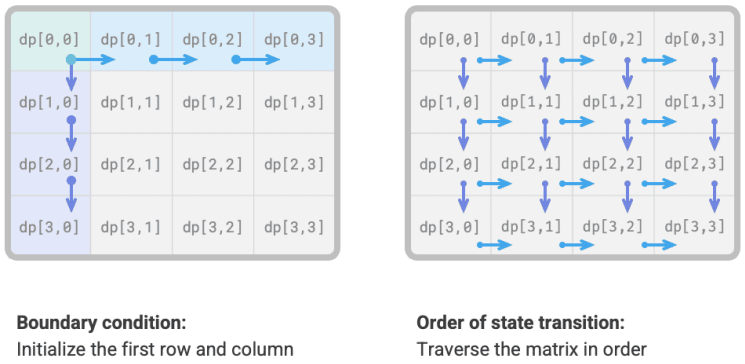


Figure 14-13 Boundary conditions and state transition order

Note

Boundary conditions in dynamic programming are used to initialize the dp table, and in search are used for pruning.

The core of state transition order is to ensure that when computing the solution to the current problem, all the smaller subproblems it depends on have already been computed correctly.

Based on the above analysis, we can directly write the dynamic programming code. However, subproblem decomposition is a top-down approach, so implementing in the order "brute force search $\rightarrow$ memoization $\rightarrow$ dynamic programming" is more aligned with thinking habits.

1. Method 1: Brute Force Search

Starting from state $[i, j]$, continuously decompose into smaller states $[i-1, j]$ and $[i, j-1]$. The recursive function includes the following elements.

- **Recursive parameters**: state $[i, j]$.
- **Return value**: minimum path sum from $[0, 0]$ to $[i, j]$, which is $dp[i, j]$.
- **Termination condition**: when $i = 0$ and $j = 0$, return cost $grid[0, 0]$.
- **Pruning**: when $i < 0$ or $j < 0$, the index is out of bounds, return cost $+\infty$, representing infeasibility.

The implementation code is as follows:

```
// === File: min_path_sum.js ===

/* Minimum path sum: Brute-force search */
function minPathSumDFS(grid, i, j) {
    // If it's the top-left cell, terminate the search
    if (i === 0 && j === 0) {
        return grid[0][0];
    }
    // If row or column index is out of bounds, return +∞ cost
    if (i < 0 || j < 0) {
        return Infinity;
    }
    // Calculate the minimum path cost from top-left to (i-1, j) and (i, j-1)
    const up = minPathSumDFS(grid, i - 1, j);
    const left = minPathSumDFS(grid, i, j - 1);
    // Return the minimum path cost from top-left to (i, j)
    return Math.min(left, up) + grid[i][j];
}
```

Figure 14-14 shows the recursion tree rooted at $dp[2, 1]$, which includes some overlapping subproblems whose number will increase sharply as the size of grid `grid` grows.

Essentially, the reason for overlapping subproblems is: **there are multiple paths from the top-left corner to reach a certain cell**.

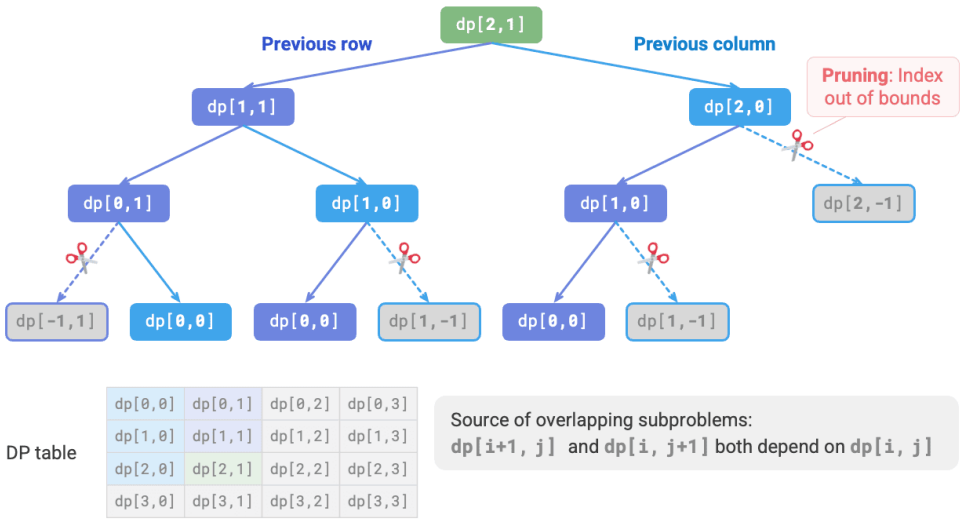


Figure 14-14 Brute force search recursion tree

Each state has two choices, down and right, so the total number of steps from the top-left corner to the bottom-right corner is $m + n - 2$, giving a worst-case time complexity of $O(2^{m+n})$, where n and m are the number of rows and columns of the grid, respectively. Note that this calculation does not account for situations near the grid boundaries, where only one choice remains when reaching the grid boundary, so the actual number of paths will be somewhat less.

2. Method 2: Memoization

We introduce a memo list `mem` of the same size as grid `grid` to record the solutions to subproblems and prune overlapping subproblems:

```
// ═══ File: min_path_sum.js ═══

/* Minimum path sum: Memoization search */
function minPathSumDFSMem(grid, mem, i, j) {
    // If it's the top-left cell, terminate the search
    if (i === 0 && j === 0) {
        return grid[0][0];
    }
    // If row or column index is out of bounds, return +∞ cost
    if (i < 0 || j < 0) {
        return Infinity;
    }
    // If there's a record, return it directly
    if (mem[i][j] !== -1) {
        return mem[i][j];
    }
    // Minimum path cost for left and upper cells
    const up = minPathSumDFSMem(grid, mem, i - 1, j);
    const left = minPathSumDFSMem(grid, mem, i, j - 1);
    // Record and return the minimum path cost from top-left to (i, j)
    mem[i][j] = Math.min(left, up) + grid[i][j];
```

```
    return mem[i][j];
}
```

As shown in Figure 14-15, after introducing memoization, all subproblem solutions only need to be computed once, so the time complexity depends on the total number of states, which is the grid size $O(nm)$.

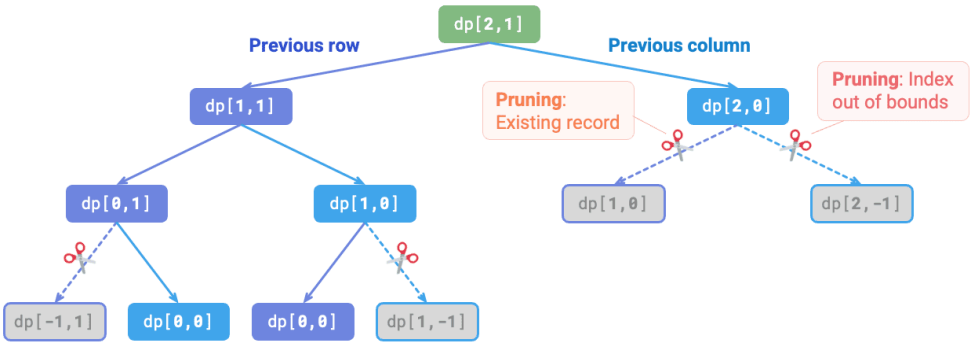


Figure 14-15 Memoization recursion tree

3. Method 3: Dynamic Programming

Implement the dynamic programming solution based on iteration, as shown in the code below:

```
// ═══ File: min_path_sum.js ═══

/* Minimum path sum: Dynamic programming */
function minPathSumDP(grid) {
    const n = grid.length,
        m = grid[0].length;
    // Initialize dp table
    const dp = Array.from({ length: n }, () =>
        Array.from({ length: m }, () => 0)
    );
    dp[0][0] = grid[0][0];
    // State transition: first row
    for (let j = 1; j < m; j++) {
        dp[0][j] = dp[0][j - 1] + grid[0][j];
    }
    // State transition: first column
    for (let i = 1; i < n; i++) {
        dp[i][0] = dp[i - 1][0] + grid[i][0];
    }
    // State transition: rest of the rows and columns
    for (let i = 1; i < n; i++) {
        for (let j = 1; j < m; j++) {
            dp[i][j] = Math.min(dp[i][j - 1], dp[i - 1][j]) + grid[i][j];
        }
    }
    return dp[n - 1][m - 1];
}
```

Figure 14-16 shows the state transition process for minimum path sum, which traverses the entire grid, **thus the time complexity is $O(nm)$**.

The array `dp` has size $n \times m$, **thus the space complexity is $O(nm)$**.

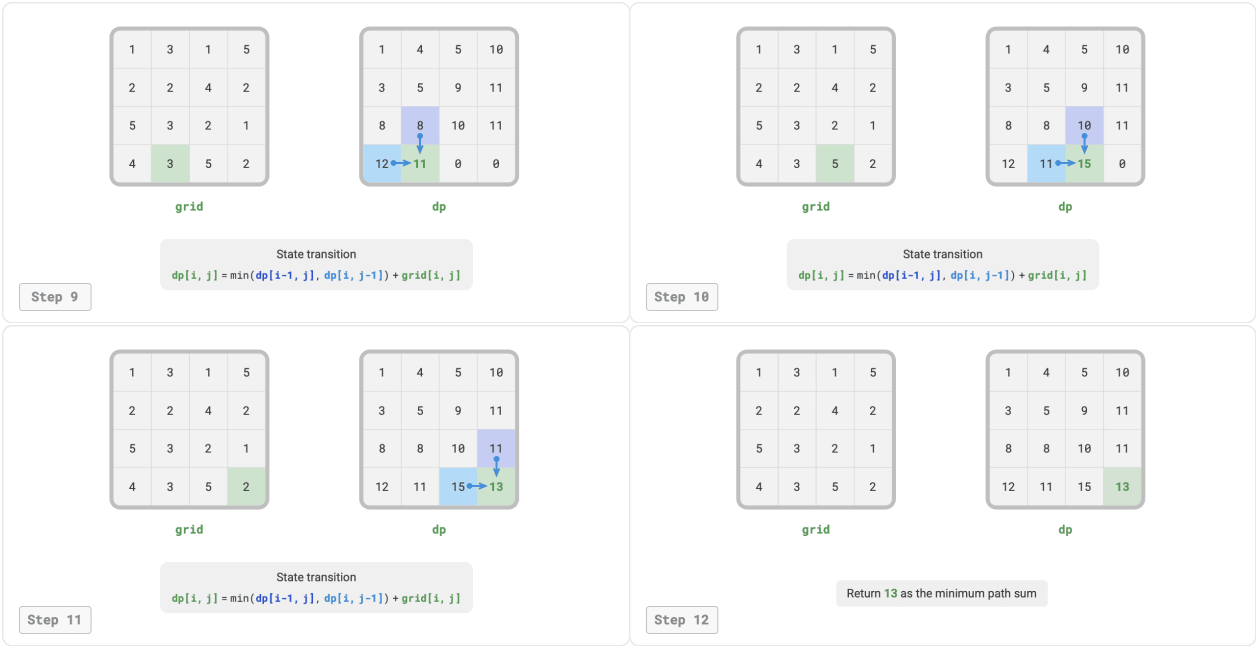


Figure 14-16 Dynamic programming process for minimum path sum

4. Space Optimization

Since each cell is only related to the cell to its left and the cell above it, we can use a single-row array to implement the dp table.

Note that since the array `dp` can only represent the state of one row, we cannot initialize the first column state in advance, but rather update it when traversing each row:

```
// ═══ File: min_path_sum.js ═══

/* Minimum path sum: Space-optimized dynamic programming */
function minPathSumDPComp(grid) {
    const n = grid.length,
        m = grid[0].length;
    // Initialize dp table
    const dp = new Array(m);
    // State transition: first row
    dp[0] = grid[0][0];
    for (let j = 1; j < m; j++) {
        dp[j] = dp[j - 1] + grid[0][j];
    }
    // State transition: rest of the rows
    for (let i = 1; i < n; i++) {
        // State transition: first column
        dp[0] = dp[0] + grid[i][0];
        // State transition: rest of the columns
        for (let j = 1; j < m; j++) {
            dp[j] = Math.min(dp[j - 1], dp[j]) + grid[i][j];
        }
    }
    return dp[m - 1];
}
```

14.4 0-1 Knapsack Problem

The knapsack problem is an excellent introductory problem for dynamic programming and is one of the most common problem forms in dynamic programming. It has many variants, such as the 0-1 knapsack problem, the unbounded knapsack problem, and the multiple knapsack problem.

In this section, we will first solve the most common 0-1 knapsack problem.

> **Question**
> Given n items, where the weight of the i-th item is $wgt[i-1]$ and its value is $val[i-1]$, and a knapsack with capacity cap. Each item can only be selected once. What is the maximum value that can be placed in the knapsack within the capacity limit?

Observe Figure 14-17. Since item number i starts counting from 1 and array indices start from 0, item i corresponds to weight $wgt[i-1]$ and value $val[i-1]$.

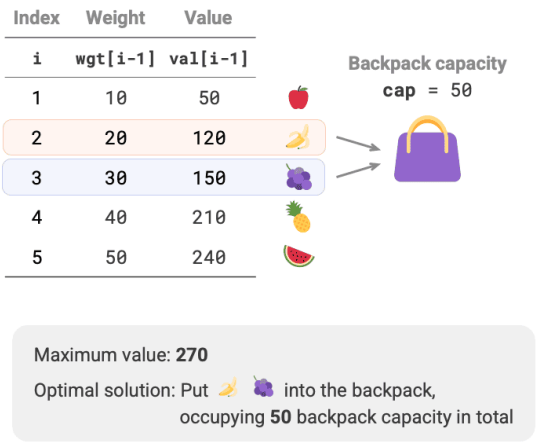


Figure 14-17 Example data for 0-1 knapsack

We can view the 0-1 knapsack problem as a process consisting of n rounds of decisions, where for each item there are two decisions: not putting it in and putting it in, thus the problem satisfies the decision tree model.

The goal of this problem is to find "the maximum value that can be placed in the knapsack within the capacity limit", so it is more likely to be a dynamic programming problem.

Step 1: Think about the decisions in each round, define the state, and thus obtain the dp table

For each item, if not placed in the knapsack, the knapsack capacity remains unchanged; if placed in, the knapsack capacity decreases. From this, we can derive the state definition: current item number i and knapsack capacity c, denoted as $[i, c]$.

State $[i, c]$ corresponds to the subproblem: **the maximum value among the first i items in a knapsack of capacity c**, denoted as $dp[i, c]$.

What we need to find is $dp[n, cap]$, so we need a two-dimensional dp table of size $(n+1) \times (cap+1)$.

Step 2: Identify the optimal substructure, and then derive the state transition equation

After making the decision for item i, what remains is the subproblem of the first $i-1$ items, which can be divided into the following two cases.

- **Not putting item** i: The knapsack capacity remains unchanged, and the state changes to $[i-1, c]$.
- **Putting item** i: The knapsack capacity decreases by $wgt[i-1]$, the value increases by $val[i-1]$, and the state changes to $[i-1, c-wgt[i-1]]$.

The above analysis reveals the optimal substructure of this problem: **the maximum value** $dp[i, c]$ **equals the larger value between not putting item** i **and putting item** i. From this, the state transition equation can be derived:

$$dp[i, c] = \max(dp[i-1, c], dp[i-1, c-wgt[i-1]] + val[i-1])$$

Note that if the weight of the current item $wgt[i-1]$ exceeds the remaining knapsack capacity c, then the only option is not to put it in the knapsack.

Step 3: Determine boundary conditions and state transition order

When there are no items or the knapsack capacity is 0, the maximum value is 0, i.e., the first column $dp[i, 0]$ and the first row $dp[0, c]$ are both equal to 0.

The current state $[i, c]$ is transferred from the state above $[i-1, c]$ and the state in the upper-left $[i-1, c-wgt[i-1]]$, so the entire dp table is traversed in order through two nested loops.

Based on the above analysis, we will next implement the brute force search, memoization, and dynamic programming solutions in order.

1. Method 1: Brute Force Search

The search code includes the following elements.

- **Recursive parameters**: state $[i, c]$.
- **Return value**: solution to the subproblem $dp[i, c]$.
- **Termination condition**: when the item number is out of bounds $i = 0$ or the remaining knapsack capacity is 0, terminate recursion and return value 0.
- **Pruning**: if the weight of the current item exceeds the remaining knapsack capacity, only the option of not putting it in is available.

```
// === File: knapsack.js ===

/* 0-1 knapsack: Brute-force search */
function knapsackDFS(wgt, val, i, c) {
    // If all items have been selected or knapsack has no remaining capacity, return value 0
    if (i === 0 || c === 0) {
        return 0;
    }
    // If exceeds knapsack capacity, can only choose not to put it in
```

```
    if (wgt[i - 1] > c) {
        return knapsackDFS(wgt, val, i - 1, c);
    }
    // Calculate the maximum value of not putting in and putting in item i
    const no = knapsackDFS(wgt, val, i - 1, c);
    const yes = knapsackDFS(wgt, val, i - 1, c - wgt[i - 1]) + val[i - 1];
    // Return the larger value of the two options
    return Math.max(no, yes);
}
```

As shown in Figure 14-18, since each item generates two search branches of not selecting and selecting, the time complexity is $O(2^n)$.

Observing the recursion tree, it is easy to see overlapping subproblems, such as $dp[1, 10]$. When there are many items, large knapsack capacity, and especially many items with the same weight, the number of overlapping subproblems will increase significantly.

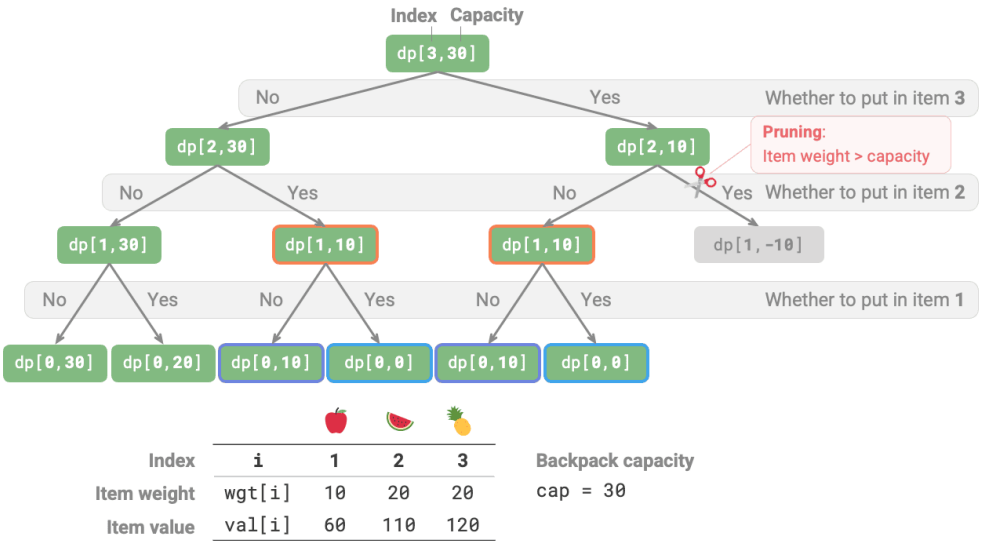


Figure 14-18 Brute force search recursion tree for 0-1 knapsack problem

2. Method 2: Memoization

To ensure that overlapping subproblems are only computed once, we use a memo list `mem` to record the solutions to subproblems, where `mem[i][c]` corresponds to $dp[i, c]$.

After introducing memoization, **the time complexity depends on the number of subproblems**, which is $O(n \times cap)$. The implementation code is as follows:

```
// === File: knapsack.js ===

/* 0-1 knapsack: Memoization search */
function knapsackDFSMem(wgt, val, mem, i, c) {
    // If all items have been selected or knapsack has no remaining capacity, return value 0
```

```
    if (i === 0 || c === 0) {
        return 0;
    }
    // If there's a record, return it directly
    if (mem[i][c] !== -1) {
        return mem[i][c];
    }
    // If exceeds knapsack capacity, can only choose not to put it in
    if (wgt[i - 1] > c) {
        return knapsackDFSMem(wgt, val, mem, i - 1, c);
    }
    // Calculate the maximum value of not putting in and putting in item i
    const no = knapsackDFSMem(wgt, val, mem, i - 1, c);
    const yes =
        knapsackDFSMem(wgt, val, mem, i - 1, c - wgt[i - 1]) + val[i - 1];
    // Record and return the larger value of the two options
    mem[i][c] = Math.max(no, yes);
    return mem[i][c];
}
```

Figure 14-19 shows the search branches pruned in memoization.

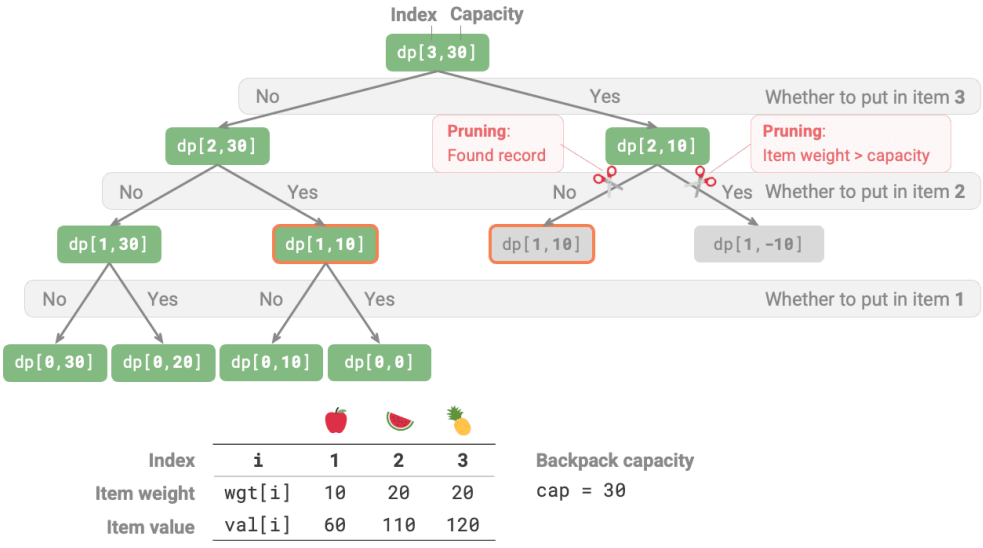


Figure 14-19 Memoization recursion tree for 0-1 knapsack problem

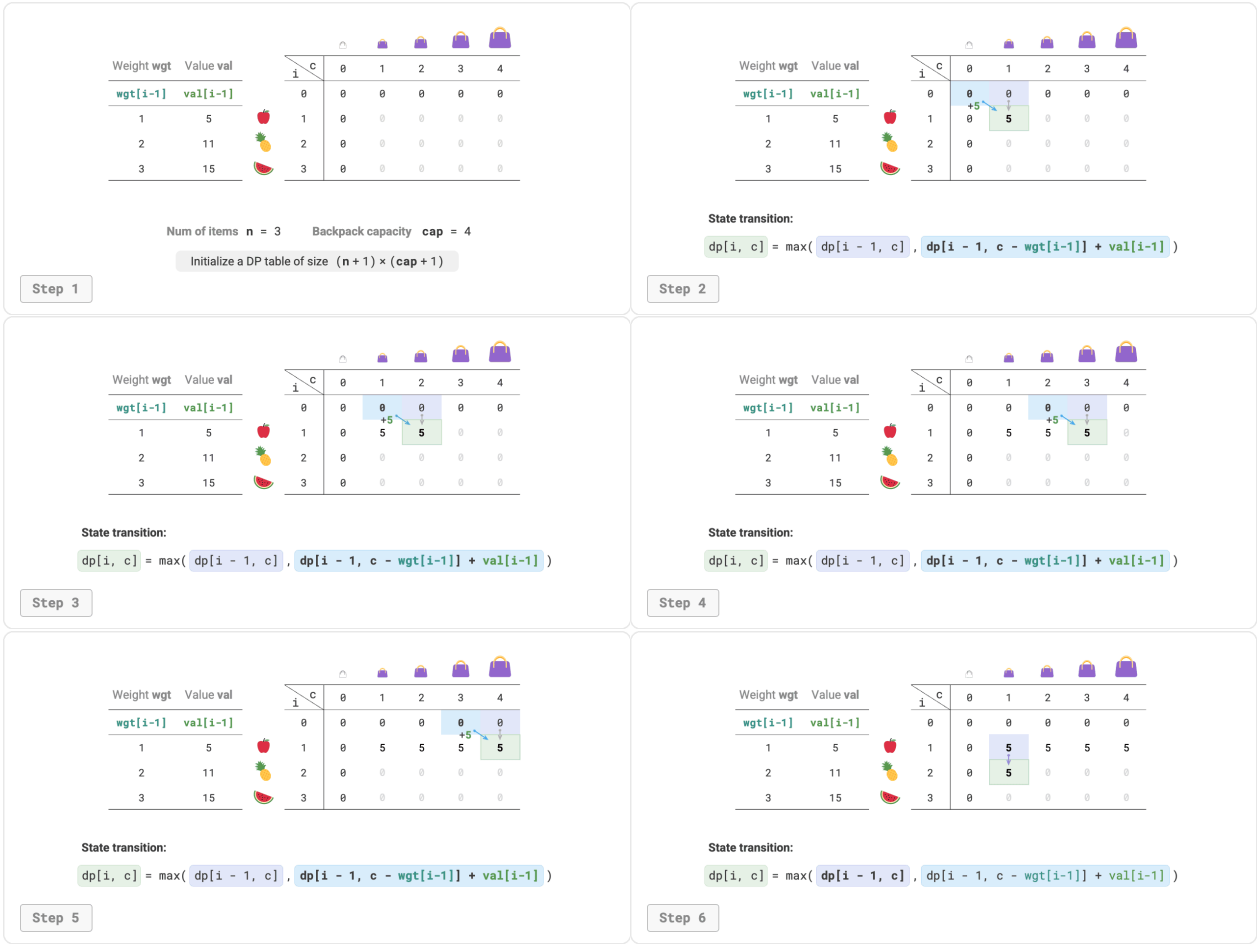
3. Method 3: Dynamic Programming

Dynamic programming is essentially the process of filling the dp table during state transitions. The code is as follows:

```
// === File: knapsack.js ===

/* 0-1 knapsack: Dynamic programming */
function knapsackDP(wgt, val, cap) {
    const n = wgt.length;
```

```
    // Initialize dp table
    const dp = Array(n + 1)
        .fill(0)
        .map(() => Array(cap + 1).fill(0));
    // State transition
    for (let i = 1; i <= n; i++) {
        for (let c = 1; c <= cap; c++) {
            if (wgt[i - 1] > c) {
                // If exceeds knapsack capacity, don't select item i
                dp[i][c] = dp[i - 1][c];
            } else {
                // The larger value between not selecting and selecting item i
                dp[i][c] = Math.max(
                    dp[i - 1][c],
                    dp[i - 1][c - wgt[i - 1]] + val[i - 1]
                );
            }
        }
    }
    return dp[n][cap];
}
```

As shown in Figure 14-20, both time complexity and space complexity are determined by the size of the array `dp`, which is $O(n \times cap)$.



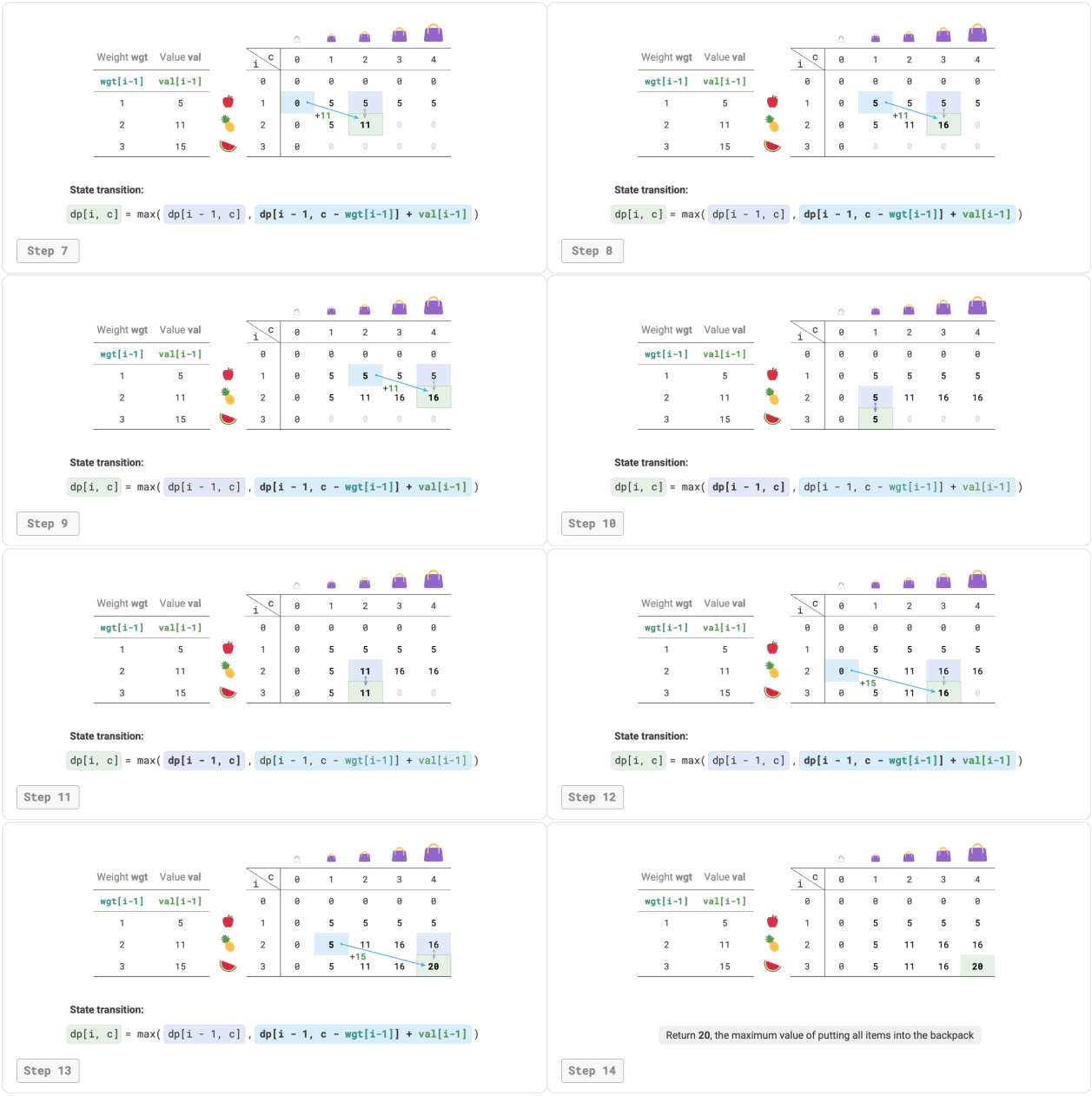


Figure 14-20 Dynamic programming process for 0-1 knapsack problem

4. Space Optimization

Since each state is only related to the state in the row above it, we can use two arrays rolling forward to reduce the space complexity from $O(n^2)$ to $O(n)$.

Further thinking, can we achieve space optimization using just one array? Observing, we can see that each state is transferred from the cell directly above or the cell in the upper-left. If there is only one array, when we start traversing row i, that array still stores the state of row $i-1$.

- If using forward traversal, then when traversing to $dp[i, j]$, the values in the upper-left $dp[i-1, 1]$ ~ $dp[i-1, j-1]$ may have already been overwritten, thus preventing correct state transition.
- If using reverse traversal, there will be no overwriting issue, and state transition can proceed correctly.

Figure 14-21 shows the transition process from row $i = 1$ to row $i = 2$ using a single array. Please consider the difference between forward and reverse traversal.

Figure 14-21 Space-optimized dynamic programming process for 0-1 knapsack

In the code implementation, we simply need to delete the first dimension i of the array `dp` and change the inner loop to reverse traversal:

```
// ═══ File: knapsack.js ═══

/* 0-1 knapsack: Space-optimized dynamic programming */
function knapsackDPComp(wgt, val, cap) {
    const n = wgt.length;
    // Initialize dp table
    const dp = Array(cap + 1).fill(0);
    // State transition
    for (let i = 1; i <= n; i++) {
```

```
        // Traverse in reverse order
        for (let c = cap; c >= 1; c--) {
            if (wgt[i - 1] <= c) {
                // The larger value between not selecting and selecting item i
                dp[c] = Math.max(dp[c], dp[c - wgt[i - 1]] + val[i - 1]);
            }
        }
    }
    return dp[cap];
}
```

14.5 Unbounded Knapsack Problem

In this section, we first solve another common knapsack problem: the unbounded knapsack, and then explore a special case of it: the coin change problem.

14.5.1 Unbounded Knapsack Problem

Question

Given n items, where the weight of the i-th item is $wgt[i-1]$ and its value is $val[i-1]$, and a knapsack with capacity cap. **Each item can be selected multiple times**. What is the maximum value that can be placed in the knapsack within the capacity limit? An example is shown in Figure 14-22.

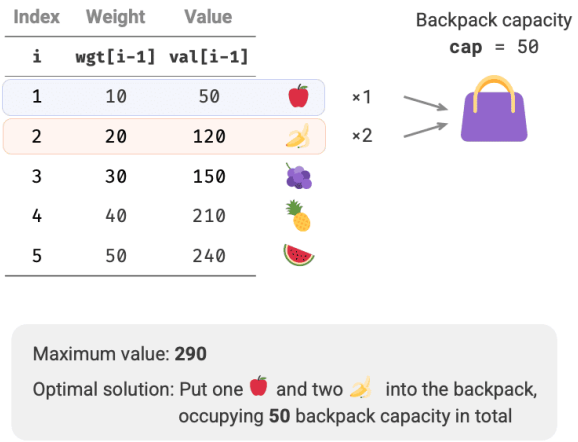


Figure 14-22 Example data for unbounded knapsack problem

1. Dynamic Programming Approach

The unbounded knapsack problem is very similar to the 0-1 knapsack problem, **differing only in that there is no limit on the number of times an item can be selected**.

- In the 0-1 knapsack problem, there is only one of each type of item, so after placing item i in the knapsack, we can only choose from the first $i-1$ items.
- In the unbounded knapsack problem, the quantity of each type of item is unlimited, so after placing item i in the knapsack, **we can still choose from the first i items**.

Under the rules of the unbounded knapsack problem, the changes in state $[i, c]$ are divided into two cases.

- **Not putting item i**: Same as the 0-1 knapsack problem, transfer to $[i-1, c]$.
- **Putting item i**: Different from the 0-1 knapsack problem, transfer to $[i, c-wgt[i-1]]$.

Thus, the state transition equation becomes:

$$dp[i,c] = \max(dp[i-1,c], dp[i,c-wgt[i-1]] + val[i-1])$$

2. Code Implementation

Comparing the code for the two problems, there is one change in state transition from $i-1$ to i, with everything else identical:

```
// ═══ File: unbounded_knapsack.js ═══

/* Unbounded knapsack: Dynamic programming */
function unboundedKnapsackDP(wgt, val, cap) {
    const n = wgt.length;
    // Initialize dp table
    const dp = Array.from({ length: n + 1 }, () =>
        Array.from({ length: cap + 1 }, () => 0)
    );
    // State transition
    for (let i = 1; i <= n; i++) {
        for (let c = 1; c <= cap; c++) {
            if (wgt[i - 1] > c) {
                // If exceeds knapsack capacity, don't select item i
                dp[i][c] = dp[i - 1][c];
            } else {
                // The larger value between not selecting and selecting item i
                dp[i][c] = Math.max(
                    dp[i - 1][c],
                    dp[i][c - wgt[i - 1]] + val[i - 1]
                );
            }
        }
    }
    return dp[n][cap];
}
```

3. Space Optimization

Since the current state is transferred from states on the left and above, **after space optimization, each row in the dp table should be traversed in forward order**.

This traversal order is exactly opposite to the 0-1 knapsack. Please refer to Figure 14-23 to understand the difference between the two.

Figure 14-23 Space-optimized dynamic programming process for unbounded knapsack problem

The code implementation is relatively simple, just delete the first dimension of the array `dp`:

```
// ═══ File: unbounded_knapsack.js ═══

/* Unbounded knapsack: Space-optimized dynamic programming */
function unboundedKnapsackDPComp(wgt, val, cap) {
    const n = wgt.length;
    // Initialize dp table
    const dp = Array.from({ length: cap + 1 }, () => 0);
    // State transition
    for (let i = 1; i <= n; i++) {
        for (let c = 1; c <= cap; c++) {
            if (wgt[i - 1] > c) {
                // If exceeds knapsack capacity, don't select item i
                dp[c] = dp[c];
            } else {
                // The larger value between not selecting and selecting item i
                dp[c] = Math.max(dp[c], dp[c - wgt[i - 1]] + val[i - 1]);
            }
```

```
        }
    }
    return dp[cap];
}
```

14.5.2 Coin Change Problem

The knapsack problem represents a large class of dynamic programming problems and has many variants, such as the coin change problem.

Question

Given n types of coins, where the denomination of the i-th type of coin is $coins[i-1]$, and the target amount is amt. **Each type of coin can be selected multiple times**. What is the minimum number of coins needed to make up the target amount? If it is impossible to make up the target amount, return -1. An example is shown in Figure 14-24.

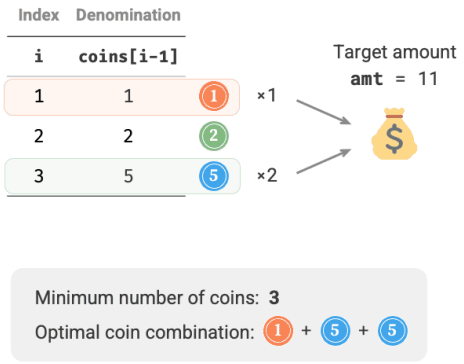


Figure 14-24 Example data for coin change problem

1. Dynamic Programming Approach

The coin change problem can be viewed as a special case of the unbounded knapsack problem, with the following connections and differences.

- The two problems can be converted to each other: "item" corresponds to "coin", "item weight" corresponds to "coin denomination", and "knapsack capacity" corresponds to "target amount".
- The optimization goals are opposite: the unbounded knapsack problem aims to maximize item value, while the coin change problem aims to minimize the number of coins.
- The unbounded knapsack problem seeks solutions "not exceeding" the knapsack capacity, while the coin change problem seeks solutions that "exactly" make up the target amount.

Step 1: Think about the decisions in each round, define the state, and thus obtain the dp table

State $[i, a]$ corresponds to the subproblem: **the minimum number of coins among the first i types of coins that can make up amount a**, denoted as $dp[i, a]$.

The two-dimensional dp table has size $(n + 1) \times (amt + 1)$.

Step 2: Identify the optimal substructure, and then derive the state transition equation

This problem differs from the unbounded knapsack problem in the following two aspects regarding the state transition equation.

- This problem seeks the minimum value, so the operator $\max()$ needs to be changed to $\min()$.
- The optimization target is the number of coins rather than item value, so when a coin is selected, simply execute $+1$.

$$dp[i, a] = \min(dp[i-1, a], dp[i, a - coins[i-1]] + 1)$$

Step 3: Determine boundary conditions and state transition order

When the target amount is 0, the minimum number of coins needed to make it up is 0, so all $dp[i, 0]$ in the first column equal 0.

When there are no coins, **it is impossible to make up any amount** > 0, which is an invalid solution. To enable the $\min()$ function in the state transition equation to identify and filter out invalid solutions, we consider using $+\infty$ to represent them, i.e., set all $dp[0, a]$ in the first row to $+\infty$.

2. Code Implementation

Most programming languages do not provide a $+\infty$ variable, and can only use the maximum value of integer type `int` as a substitute. However, this can lead to large number overflow: the $+1$ operation in the state transition equation may cause overflow.

For this reason, we use the number $amt + 1$ to represent invalid solutions, because the maximum number of coins needed to make up amt is at most amt. Before returning, check whether $dp[n, amt]$ equals $amt + 1$; if so, return -1, indicating that the target amount cannot be made up. The code is as follows:

```
// ═══ File: coin_change.js ═══

/* Coin change: Dynamic programming */
function coinChangeDP(coins, amt) {
    const n = coins.length;
    const MAX = amt + 1;
    // Initialize dp table
    const dp = Array.from({ length: n + 1 }, () =>
        Array.from({ length: amt + 1 }, () => 0)
    );
    // State transition: first row and first column
    for (let a = 1; a <= amt; a++) {
        dp[0][a] = MAX;
    }
    // State transition: rest of the rows and columns
    for (let i = 1; i <= n; i++) {
```

```
        for (let a = 1; a <= amt; a++) {
            if (coins[i - 1] > a) {
                // If exceeds target amount, don't select coin i
                dp[i][a] = dp[i - 1][a];
            } else {
                // The smaller value between not selecting and selecting coin i
                dp[i][a] = Math.min(dp[i - 1][a], dp[i][a - coins[i - 1]] + 1);
            }
        }
    }
    return dp[n][amt] !== MAX ? dp[n][amt] : -1;
}
```

Figure 14-25 shows the dynamic programming process for coin change, which is very similar to the unbounded knapsack problem.

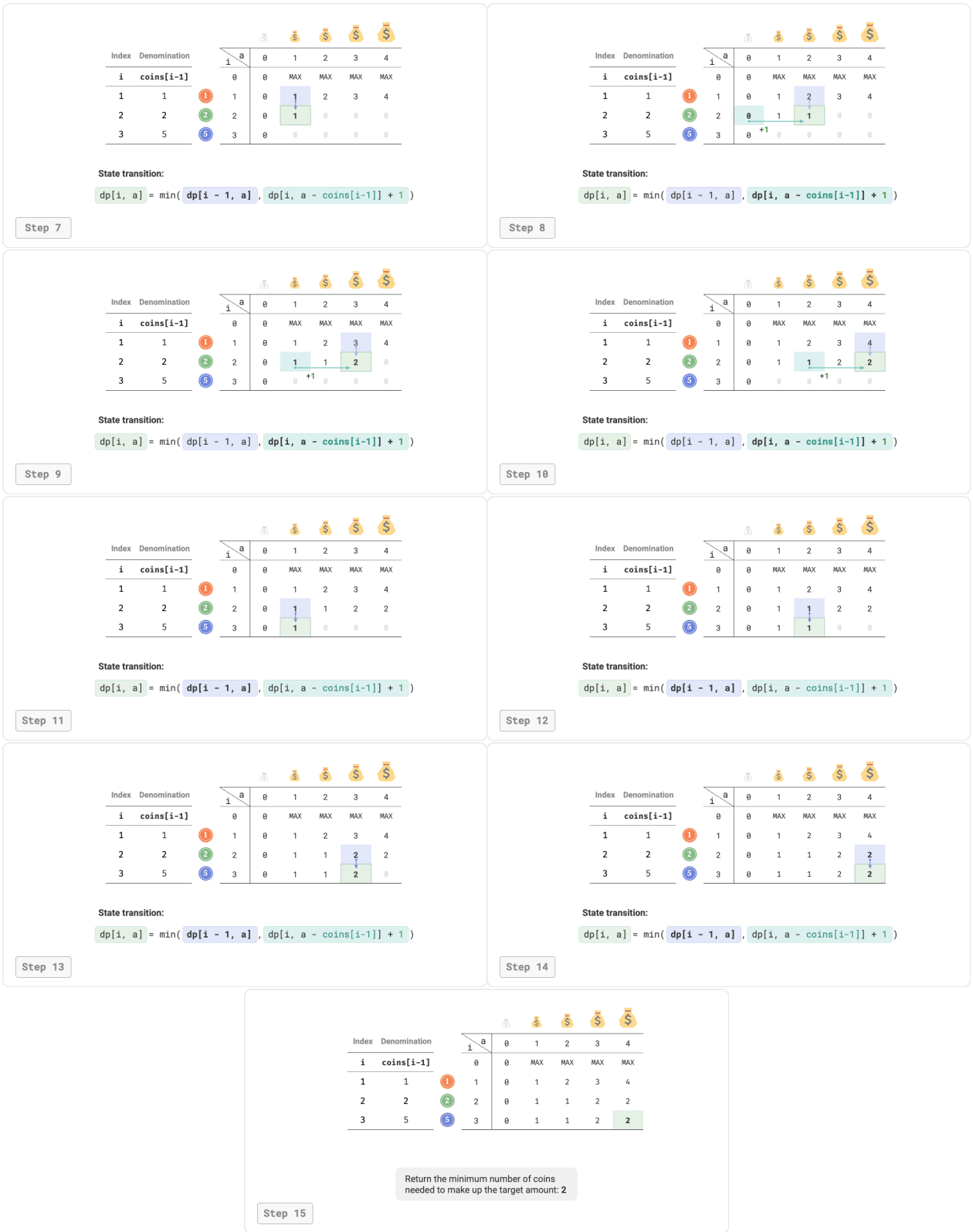


Figure 14-25 Dynamic programming process for coin change problem

3. Space Optimization

The space optimization for the coin change problem is handled in the same way as the unbounded knapsack problem:

```
// ═══ File: coin_change.js ═══

/* Coin change: Space-optimized dynamic programming */
function coinChangeDPComp(coins, amt) {
    const n = coins.length;
    const MAX = amt + 1;
    // Initialize dp table
    const dp = Array.from({ length: amt + 1 }, () => MAX);
    dp[0] = 0;
    // State transition
    for (let i = 1; i <= n; i++) {
        for (let a = 1; a <= amt; a++) {
            if (coins[i - 1] > a) {
                // If exceeds target amount, don't select coin i
                dp[a] = dp[a];
            } else {
                // The smaller value between not selecting and selecting coin i
                dp[a] = Math.min(dp[a], dp[a - coins[i - 1]] + 1);
            }
        }
    }
    return dp[amt] !== MAX ? dp[amt] : -1;
}
```

14.5.3 Coin Change Problem Ii

Question

Given n types of coins, where the denomination of the i-th type of coin is $coins[i-1]$, and the target amount is amt. Each type of coin can be selected multiple times. **What is the number of coin combinations that can make up the target amount?** An example is shown in Figure 14-26.

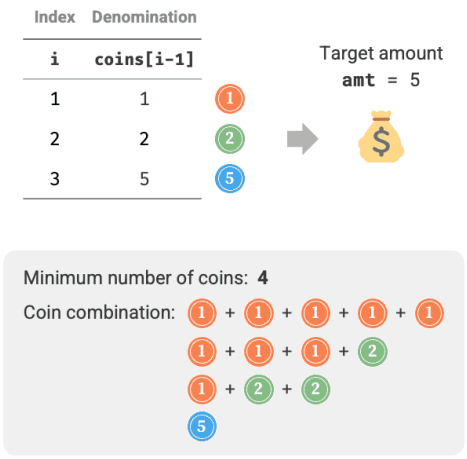


Figure 14-26 Example data for coin change problem II

1. Dynamic Programming Approach

Compared to the previous problem, this problem's goal is to find the number of combinations, so the subproblem becomes: **the number of combinations among the first i types of coins that can make up amount a**. The dp table remains a two-dimensional matrix of size $(n+1) \times (amt+1)$.

The number of combinations for the current state equals the sum of the combinations from not selecting the current coin and selecting the current coin. The state transition equation is:

$$dp[i, a] = dp[i-1, a] + dp[i, a - coins[i-1]]$$

When the target amount is 0, no coins need to be selected to make up the target amount, so all $dp[i, 0]$ in the first column should be initialized to 1. When there are no coins, it is impossible to make up any amount > 0, so all $dp[0, a]$ in the first row equal 0.

2. Code Implementation

```javascript
// ═══ File: coin_change_ii.js ═══

/* Coin change II: Dynamic programming */
function coinChangeIIDP(coins, amt) {
    const n = coins.length;
    // Initialize dp table
    const dp = Array.from({ length: n + 1 }, () =>
        Array.from({ length: amt + 1 }, () => 0)
    );
    // Initialize first column
    for (let i = 0; i <= n; i++) {
        dp[i][0] = 1;
    }
    // State transition
    for (let i = 1; i <= n; i++) {
        for (let a = 1; a <= amt; a++) {
            if (coins[i - 1] > a) {
                // If exceeds target amount, don't select coin i
                dp[i][a] = dp[i - 1][a];
            } else {
                // Sum of the two options: not selecting and selecting coin i
                dp[i][a] = dp[i - 1][a] + dp[i][a - coins[i - 1]];
            }
        }
    }
    return dp[n][amt];
}
```

3. Space Optimization

The space optimization is handled in the same way, just delete the coin dimension:

```
// == File: coin_change_ii.js ==

/* Coin change II: Space-optimized dynamic programming */
function coinChangeIIDPComp(coins, amt) {
    const n = coins.length;
    // Initialize dp table
    const dp = Array.from({ length: amt + 1 }, () => 0);
    dp[0] = 1;
    // State transition
    for (let i = 1; i <= n; i++) {
        for (let a = 1; a <= amt; a++) {
            if (coins[i - 1] > a) {
                // If exceeds target amount, don't select coin i
                dp[a] = dp[a];
            } else {
                // Sum of the two options: not selecting and selecting coin i
                dp[a] = dp[a] + dp[a - coins[i - 1]];
            }
        }
    }
    return dp[amt];
}
```

14.6 Edit Distance Problem

Edit distance, also known as Levenshtein distance, refers to the minimum number of edits required to transform one string into another, commonly used in information retrieval and natural language processing to measure the similarity between two sequences.

Question

Given two strings s and t, return the minimum number of edits required to transform s into t. You can perform three types of edit operations on a string: insert a character, delete a character, or replace a character with any other character.

As shown in Figure 14-27, transforming `kitten` into `sitting` requires 3 edits, including 2 replacements and 1 insertion; transforming `hello` into `algo` requires 3 steps, including 2 replacements and 1 deletion.

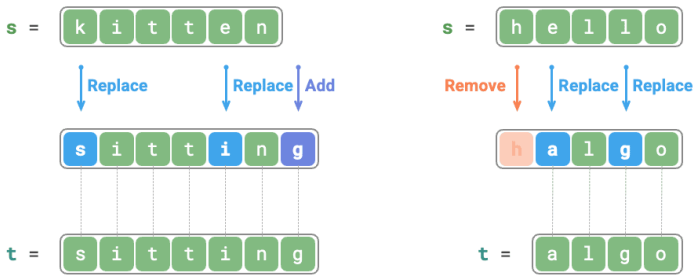


Figure 14-27 Example data for edit distance

The edit distance problem can be naturally explained using the decision tree model. Strings correspond to tree nodes, and a round of decision (one edit operation) corresponds to an edge of the tree.

As shown in Figure 14-28, without restricting operations, each node can branch into many edges, with each edge corresponding to one operation, meaning there are many possible paths to transform `hello` into `algo`.

From the perspective of the decision tree, the goal of this problem is to find the shortest path between node `hello` and node `algo`.

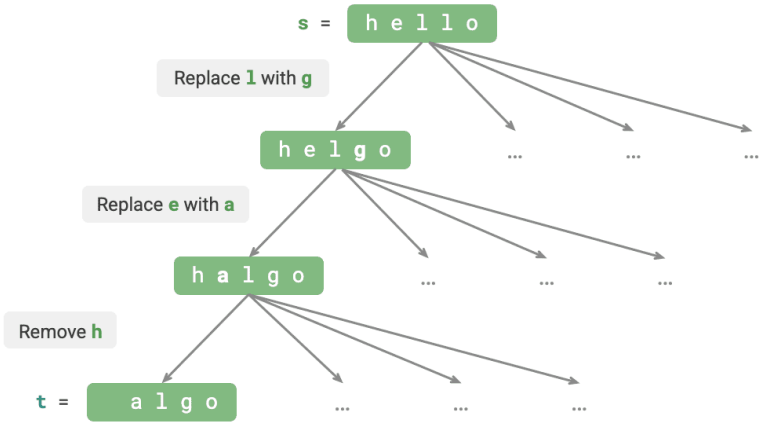


Figure 14-28 Representing edit distance problem based on decision tree model

1. Dynamic Programming Approach

Step 1: Think about the decisions in each round, define the state, and thus obtain the dp table

Each round of decision involves performing one edit operation on string s.

We want the problem scale to gradually decrease during the editing process, which allows us to construct subproblems. Let the lengths of strings s and t be n and m respectively. We first consider the tail characters of the two strings, $s[n-1]$ and $t[m-1]$.

- If $s[n-1]$ and $t[m-1]$ are the same, we can skip them and directly consider $s[n-2]$ and $t[m-2]$.
- If $s[n-1]$ and $t[m-1]$ are different, we need to perform one edit on s (insert, delete, or replace) to make the tail characters of the two strings the same, allowing us to skip them and consider a smaller-scale problem.

In other words, each round of decision (edit operation) we make on string s will change the remaining characters to be matched in s and t. Therefore, the state is the i-th and j-th characters currently being considered in s and t, denoted as $[i, j]$.

State $[i, j]$ corresponds to the subproblem: **the minimum number of edits required to change the first i characters of s into the first j characters of t.**

From this, we obtain a two-dimensional dp table of size $(i+1) \times (j+1)$.

Step 2: Identify the optimal substructure, and then derive the state transition equation

Consider subproblem $dp[i, j]$, where the tail characters of the corresponding two strings are $s[i-1]$ and $t[j-1]$, which can be divided into the three cases shown in Figure 14-29 based on different edit operations.

1. Insert $t[j-1]$ after $s[i-1]$, then the remaining subproblem is $dp[i, j-1]$.
2. Delete $s[i-1]$, then the remaining subproblem is $dp[i-1, j]$.
3. Replace $s[i-1]$ with $t[j-1]$, then the remaining subproblem is $dp[i-1, j-1]$.

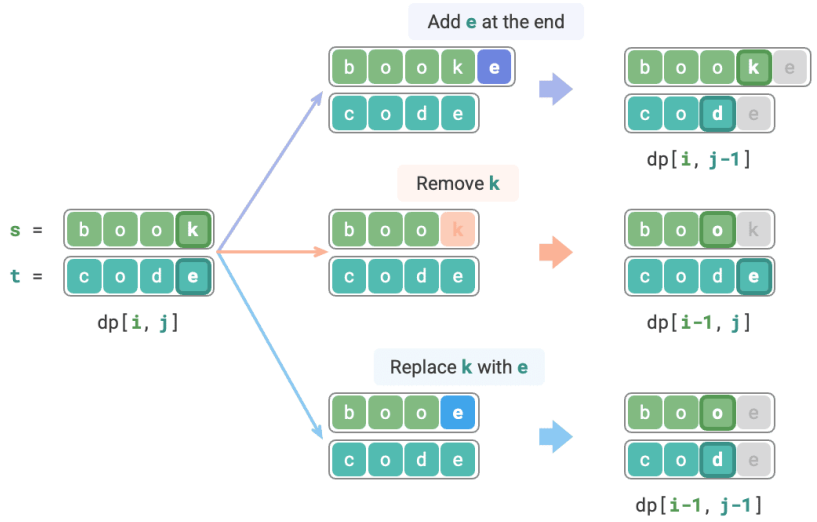


Figure 14-29 State transition for edit distance

Based on the above analysis, the optimal substructure can be obtained: the minimum number of edits for $dp[i, j]$ equals the minimum among the minimum edit steps of $dp[i, j-1]$, $dp[i-1, j]$, and $dp[i-1, j-1]$, plus the edit step 1 for this time. The corresponding state transition equation is:

$$dp[i, j] = \min(dp[i, j-1], dp[i-1, j], dp[i-1, j-1]) + 1$$

Please note that **when $s[i-1]$ and $t[j-1]$ are the same, no edit is required for the current character**, in which case the state transition equation is:

$$dp[i, j] = dp[i-1, j-1]$$

Step 3: Determine boundary conditions and state transition order

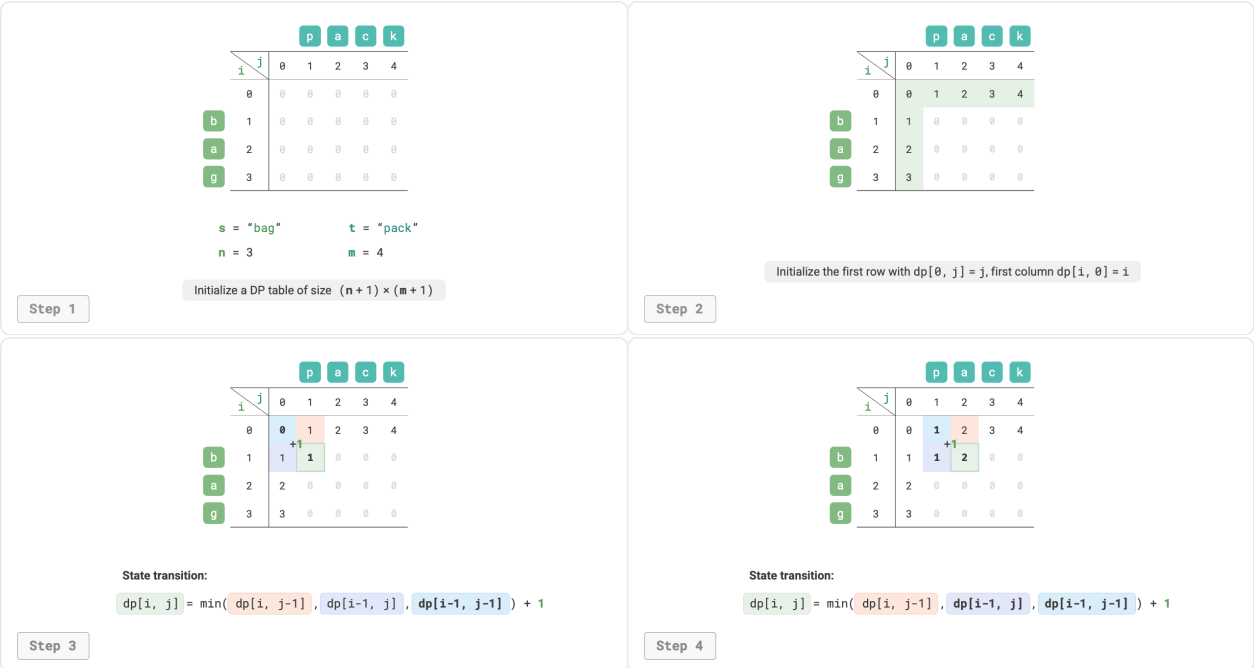
When both strings are empty, the number of edit steps is 0, i.e., $dp[0, 0] = 0$. When s is empty but t is not, the minimum number of edit steps equals the length of t, i.e., the first row $dp[0, j] = j$. When s is not empty but t is empty, the minimum number of edit steps equals the length of s, i.e., the first column $dp[i, 0] = i$.

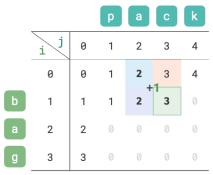
Observing the state transition equation, the solution $dp[i, j]$ depends on solutions to the left, above, and upper-left, so the entire dp table can be traversed in order through two nested loops.

2. Code Implementation

```
// ═══ File: edit_distance.js ═══

/* Edit distance: Dynamic programming */
function editDistanceDP(s, t) {
    const n = s.length,
        m = t.length;
    const dp = Array.from({ length: n + 1 }, () => new Array(m + 1).fill(0));
    // State transition: first row and first column
    for (let i = 1; i <= n; i++) {
        dp[i][0] = i;
    }
    for (let j = 1; j <= m; j++) {
        dp[0][j] = j;
    }
    // State transition: rest of the rows and columns
    for (let i = 1; i <= n; i++) {
        for (let j = 1; j <= m; j++) {
            if (s.charAt(i - 1) === t.charAt(j - 1)) {
                // If two characters are equal, skip both characters
                dp[i][j] = dp[i - 1][j - 1];
            } else {
                // Minimum edit steps = minimum edit steps of insert, delete, replace + 1
                dp[i][j] =
                    Math.min(dp[i][j - 1], dp[i - 1][j], dp[i - 1][j - 1]) + 1;
            }
        }
    }
    return dp[n][m];
}
```

As shown in Figure 14-30, the state transition process for the edit distance problem is very similar to the knapsack problem and can both be viewed as the process of filling a two-dimensional grid.

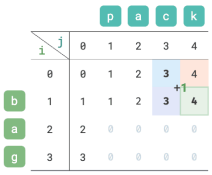




State transition:

dp[i, j] = min(dp[i, j-1] , **dp[i-1, j]** , **dp[i-1, j-1]**) + 1

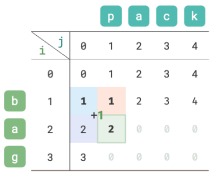
Step 5



State transition:

dp[i, j] = min(dp[i, j-1] , **dp[i-1, j]** , **dp[i-1, j-1]**) + 1

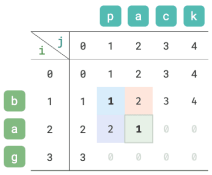
Step 6



State transition:

dp[i, j] = min(**dp[i, j-1]** , dp[i-1, j] , **dp[i-1, j-1]**) + 1

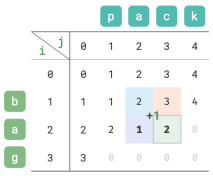
Step 7



State transition:

∵ s[i-1] = t[i-1] = 'a' ∴ dp[i, j] = **dp[i-1, j-1]**

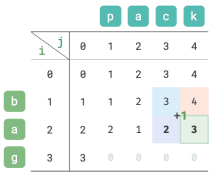
Step 8



State transition:

dp[i, j] = min(dp[i, j-1] , **dp[i-1, j]** , dp[i-1, j-1]) + 1

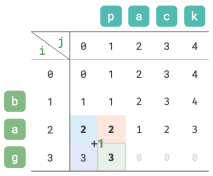
Step 9



State transition:

dp[i, j] = min(dp[i, j-1] , **dp[i-1, j]** , dp[i-1, j-1]) + 1

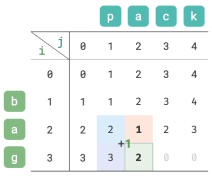
Step 10



State transition:

dp[i, j] = min(**dp[i, j-1]** , dp[i-1, j] , **dp[i-1, j-1]**) + 1

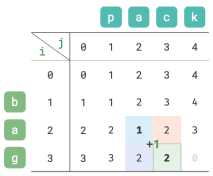
Step 11



State transition:

dp[i, j] = min(**dp[i, j-1]** , dp[i-1, j] , dp[i-1, j-1]) + 1

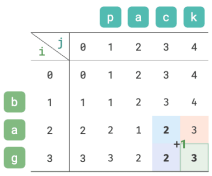
Step 12



State transition:

dp[i, j] = min(dp[i, j-1] , dp[i-1, j] , **dp[i-1, j-1]**) + 1

Step 13



State transition:

dp[i, j] = min(dp[i, j-1] , **dp[i-1, j]** , **dp[i-1, j-1]**) + 1

Step 14

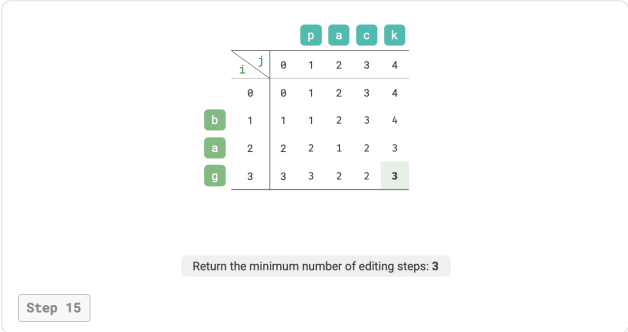


Figure 14-30 Dynamic programming process for edit distance

3. Space Optimization

Since $dp[i, j]$ is transferred from the solutions above $dp[i-1, j]$, to the left $dp[i, j-1]$, and to the upper-left $dp[i-1, j-1]$, forward traversal will lose the upper-left solution $dp[i-1, j-1]$, and reverse traversal cannot build $dp[i, j-1]$ in advance, so neither traversal order is feasible.

For this reason, we can use a variable `leftup` to temporarily store the upper-left solution $dp[i-1, j-1]$, so we only need to consider the solutions to the left and above. This situation is the same as the unbounded knapsack problem, allowing for forward traversal. The code is as follows:

```
// === File: edit_distance.js ===

/* Edit distance: Space-optimized dynamic programming */
function editDistanceDPComp(s, t) {
    const n = s.length,
        m = t.length;
    const dp = new Array(m + 1).fill(0);
    // State transition: first row
    for (let j = 1; j <= m; j++) {
        dp[j] = j;
    }
    // State transition: rest of the rows
    for (let i = 1; i <= n; i++) {
        // State transition: first column
        let leftup = dp[0]; // Temporarily store dp[i-1, j-1]
        dp[0] = i;
        // State transition: rest of the columns
        for (let j = 1; j <= m; j++) {
            const temp = dp[j];
            if (s.charAt(i - 1) === t.charAt(j - 1)) {
                // If two characters are equal, skip both characters
                dp[j] = leftup;
            } else {
                // Minimum edit steps = minimum edit steps of insert, delete, replace + 1
                dp[j] = Math.min(dp[j - 1], dp[j], leftup) + 1;
            }
            leftup = temp; // Update for next round's dp[i-1, j-1]
        }
    }
    return dp[m];
}
```

14.7 Summary

1. Key Review

- Dynamic programming decomposes problems and avoids redundant computation by storing the solutions to subproblems, thereby significantly improving computational efficiency.
- Without considering time constraints, all dynamic programming problems can be solved using backtracking (brute force search), but the recursion tree contains a large number of overlapping subproblems, resulting in extremely low efficiency. By introducing a memo list, we can store the solutions to all computed subproblems, ensuring that overlapping subproblems are only computed once.
- Memoization is a top-down recursive solution, while the corresponding dynamic programming is a bottom-up iterative solution, similar to "filling in a table". Since the current state only depends on certain local states, we can eliminate one dimension of the dp table to reduce space complexity.
- Subproblem decomposition is a general algorithmic approach, with different properties in divide and conquer, dynamic programming, and backtracking.
- Dynamic programming problems have three major characteristics: overlapping subproblems, optimal substructure, and no aftereffects.
- If the optimal solution to the original problem can be constructed from the optimal solutions to the subproblems, then it has optimal substructure.
- No aftereffects means that for a given state, its future development is only related to that state and has nothing to do with all past states. Many combinatorial optimization problems do not have no aftereffects and cannot be quickly solved using dynamic programming.

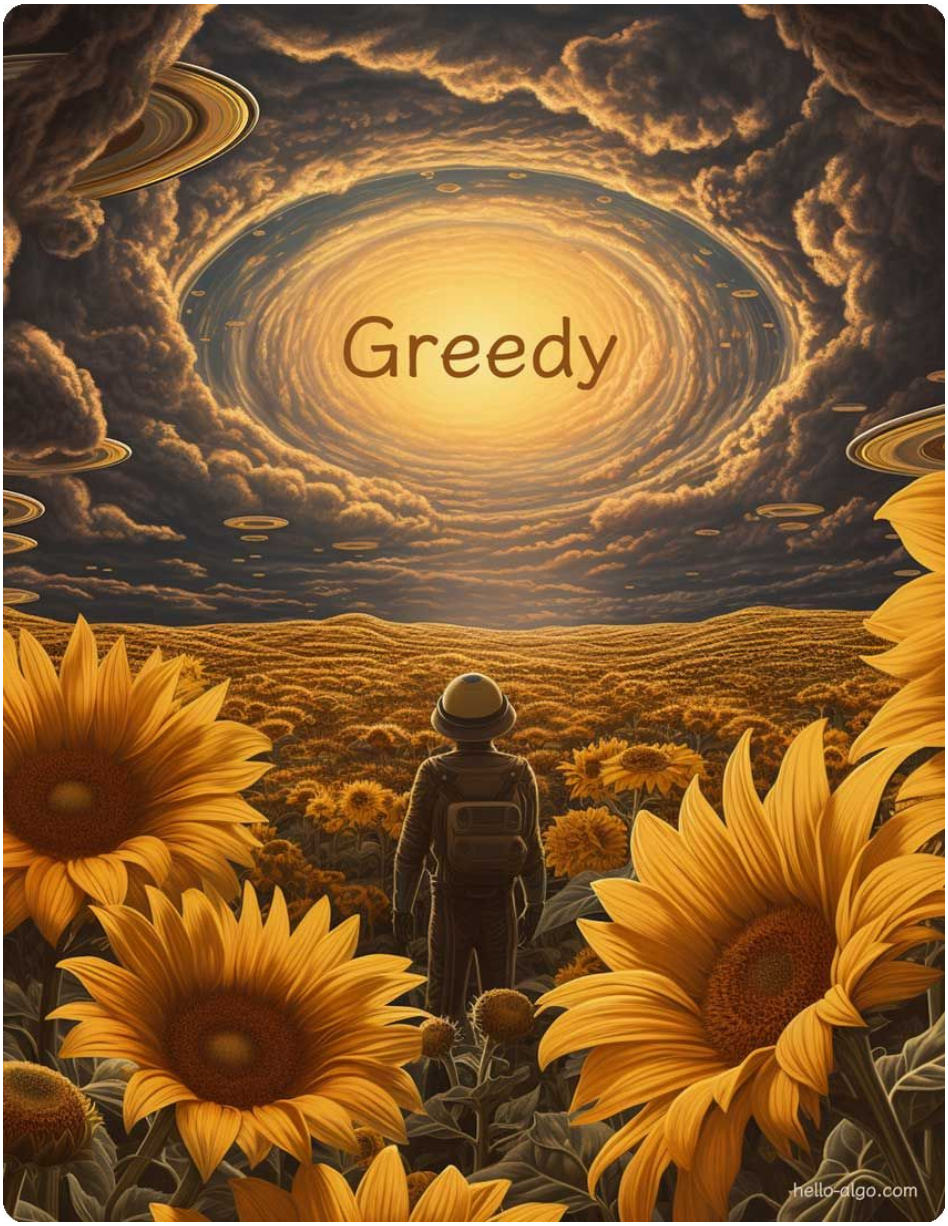
Knapsack problem

- The knapsack problem is one of the most typical dynamic programming problems, with variants such as the 0-1 knapsack, unbounded knapsack, and multiple knapsack.
- The state definition for the 0-1 knapsack is the maximum value among the first i items in a knapsack of capacity c. Based on the two decisions of not putting an item in the knapsack and putting it in, the optimal substructure can be identified and the state transition equation constructed. In space optimization, since each state depends on the state directly above and to the upper-left, the list needs to be traversed in reverse order to avoid overwriting the upper-left state.
- The unbounded knapsack problem has no limit on the selection quantity of each type of item, so the state transition for choosing to put in an item differs from the 0-1 knapsack problem. Since the state depends on the state directly above and directly to the left, space optimization should use forward traversal.
- The coin change problem is a variant of the unbounded knapsack problem. It changes from seeking the "maximum" value to seeking the "minimum" number of coins, so $\max()$ in the state transition equation should be changed to $\min()$. It changes from seeking "not exceeding" the knapsack capacity to seeking "exactly" making up the target amount, so $amt + 1$ is used to represent the invalid solution of "unable to make up the target amount".
- Coin change problem II changes from seeking the "minimum number of coins" to seeking the "number of coin combinations", so the state transition equation correspondingly changes from $\min()$ to a summation operator.

Edit distance problem

- Edit distance (Levenshtein distance) is used to measure the similarity between two strings, defined as the minimum number of edit steps from one string to another, with edit operations including insert, delete, and replace.
- The state definition for the edit distance problem is the minimum number of edit steps required to change the first i characters of s into the first j characters of t. When $s[i] \neq t[j]$, there are three decisions: insert, delete, replace, each with corresponding remaining subproblems. From this, the optimal substructure can be identified and the state transition equation constructed. When $s[i] = t[j]$, no edit is required for the current character.
- In edit distance, the state depends on the state directly above, directly to the left, and to the upper-left, so after space optimization, neither forward nor reverse traversal can correctly perform state transitions. For this reason, we use a variable to temporarily store the upper-left state, thus transforming to a situation equivalent to the unbounded knapsack problem, allowing for forward traversal after space optimization.

Chapter 15. Greedy



Abstract

Sunflowers turn toward the sun, constantly pursuing the maximum potential for their own growth.

Through rounds of simple choices, greedy strategies gradually lead to the best answer.

15.1 Greedy Algorithm

Greedy algorithm is a common algorithm for solving optimization problems. Its basic idea is to make the seemingly best choice at each decision stage of the problem, that is, to greedily make locally optimal decisions in hopes of obtaining a globally optimal solution. Greedy algorithms are simple and efficient, and are widely applied in many practical problems.

Greedy algorithms and dynamic programming are both commonly used to solve optimization problems. They share some similarities, such as both relying on the optimal substructure property, but they work differently.

- Dynamic programming considers all previous decisions when making the current decision, and uses solutions to past subproblems to construct the solution to the current subproblem.
- Greedy algorithms do not consider past decisions, but instead make greedy choices moving forward, continually reducing the problem size until the problem is solved.

We will first understand how greedy algorithms work through the example problem "coin change". This problem has already been introduced in the "Complete Knapsack Problem" chapter, so I believe you are not unfamiliar with it.

> **Question**
> Given n types of coins, where the denomination of the i-th type of coin is $coins[i-1]$, and the target amount is amt, with each type of coin available for repeated selection, what is the minimum number of coins needed to make up the target amount? If it is impossible to make up the target amount, return -1.

The greedy strategy adopted for this problem is shown in Figure 15-1. Given a target amount, **we greedily select the coin that is not greater than and closest to it**, and continuously repeat this step until the target amount is reached.

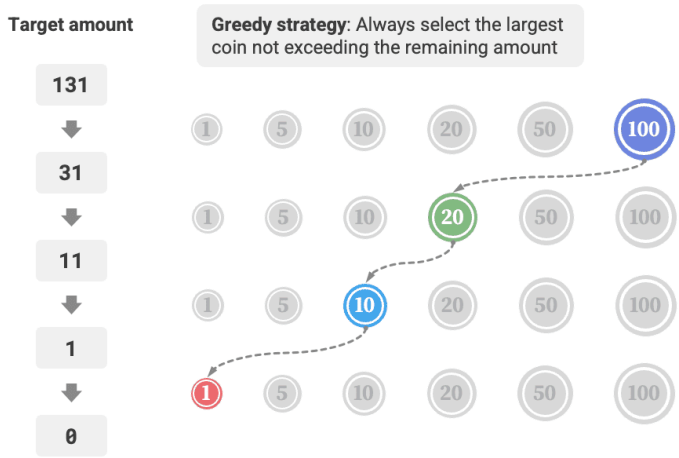


Figure 15-1 Greedy strategy for coin change

The implementation code is as follows:

```
// ═══ File: coin_change_greedy.js ═══

/* Coin change: Greedy algorithm */
function coinChangeGreedy(coins, amt) {
    // Assume coins array is sorted
    let i = coins.length - 1;
    let count = 0;
    // Loop to make greedy choices until no remaining amount
    while (amt > 0) {
        // Find the coin that is less than and closest to the remaining amount
        while (i > 0 && coins[i] > amt) {
            i--;
        }
        // Choose coins[i]
        amt -= coins[i];
        count++;
    }
    // If no feasible solution is found, return -1
    return amt === 0 ? count : -1;
}
```

You might exclaim: So clean! The greedy algorithm solves the coin change problem in about ten lines of code.

15.1.1 Advantages and Limitations of Greedy Algorithms

Greedy algorithms are not only straightforward and simple to implement, but are also usually very efficient. In the code above, if the smallest coin denomination is $\min(coins)$, the greedy choice loops at most $amt / \min(coins)$ times, giving a time complexity of $O(amt / \min(coins))$. This is an order of magnitude smaller than the time complexity of the dynamic programming solution $O(n \times amt)$.

However, **for certain coin denomination combinations, greedy algorithms cannot find the optimal solution.** Figure 15-2 provides two examples.

- **Positive example** $coins = [1, 5, 10, 20, 50, 100]$: With this coin combination, given any amt, the greedy algorithm can find the optimal solution.
- **Negative example** $coins = [1, 20, 50]$: Suppose $amt = 60$, the greedy algorithm can only find the combination $50 + 1 \times 10$, totaling 11 coins, but dynamic programming can find the optimal solution $20 + 20 + 20$, requiring only 3 coins.
- **Negative example** $coins = [1, 49, 50]$: Suppose $amt = 98$, the greedy algorithm can only find the combination $50 + 1 \times 48$, totaling 49 coins, but dynamic programming can find the optimal solution $49 + 49$, requiring only 2 coins.

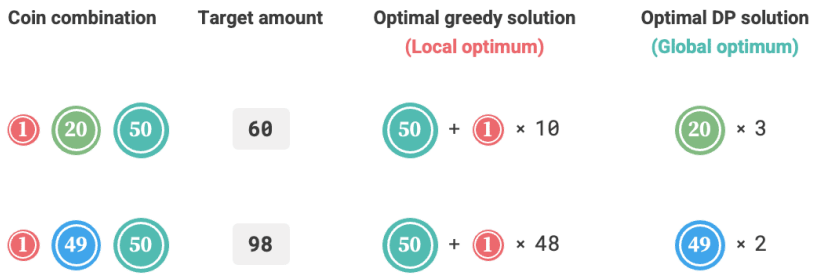


Figure 15-2 Examples where greedy algorithms cannot find the optimal solution

In other words, for the coin change problem, greedy algorithms cannot guarantee finding the global optimal solution, and may even find very poor solutions. It is better suited for solving with dynamic programming.

Generally, the applicability of greedy algorithms falls into the following two situations.

1. **Can guarantee finding the optimal solution**: In this situation, greedy algorithms are often the best choice, because they tend to be more efficient than backtracking and dynamic programming.
2. **Can find an approximate optimal solution**: Greedy algorithms are also applicable in this situation. For many complex problems, finding the global optimal solution is very difficult, and being able to find a suboptimal solution with high efficiency is also very good.

15.1.2 Characteristics of Greedy Algorithms

So the question arises: what kind of problems are suitable for solving with greedy algorithms? Or in other words, under what conditions can greedy algorithms guarantee finding the optimal solution?

Compared to dynamic programming, the conditions for using greedy algorithms are stricter, mainly focusing on two properties of the problem.

- **Greedy choice property**: Only when locally optimal choices can always lead to a globally optimal solution can greedy algorithms guarantee obtaining the optimal solution.
- **Optimal substructure**: The optimal solution to the original problem contains the optimal solutions to subproblems.

Optimal substructure has already been introduced in the "Dynamic Programming" chapter, so we won't elaborate on it here. It's worth noting that the optimal substructure of some problems is not obvious, but they can still be solved using greedy algorithms.

We mainly explore methods for determining the greedy choice property. Although its description seems relatively simple, **in practice, for many problems, proving the greedy choice property is not easy**.

For example, in the coin change problem, although we can easily provide counterexamples to disprove the greedy choice property, proving it is quite difficult. If asked: **what conditions must a coin combination satisfy to be solvable using a greedy algorithm**? We often can only rely on intuition or examples to give an ambiguous answer, and find it difficult to provide a rigorous mathematical proof.

Quote

There is a paper that presents an algorithm with $O(n^3)$ time complexity for determining whether a coin combination can use a greedy algorithm to find the optimal solution for any amount. Pearson, D. A polynomial-time algorithm for the change-making problem[J]. Operations Research Letters, 2005, 33(3): 231-234.

15.1.3 Steps for Solving Problems with Greedy Algorithms

The problem-solving process for greedy problems can generally be divided into the following three steps.

1. **Problem analysis**: Sort out and understand the problem characteristics, including state definition, optimization objectives, and constraints, etc. This step is also involved in backtracking and dynamic programming.
2. **Determine the greedy strategy**: Determine how to make greedy choices at each step. This strategy should be able to reduce the problem size at each step, ultimately solving the entire problem.
3. **Correctness proof**: It is usually necessary to prove that the problem has both greedy choice property and optimal substructure. This step may require mathematical proofs, such as mathematical induction or proof by contradiction.

Determining the greedy strategy is the core step in solving the problem, but it may not be easy to implement, mainly for the following reasons.

- **Greedy strategies differ greatly between different problems**. For many problems, the greedy strategy is relatively straightforward, and we can derive it through some general thinking and attempts. However, for some complex problems, the greedy strategy may be very elusive, which really tests one's problem-solving experience and algorithmic ability.
- **Some greedy strategies are highly misleading**. When we confidently design a greedy strategy, write the solution code and submit it for testing, we may find that some test cases cannot pass. This is because the designed greedy strategy is only "partially correct", as exemplified by the coin change problem discussed above.

To ensure correctness, we should rigorously mathematically prove the greedy strategy, **usually using proof by contradiction or mathematical induction**.

However, correctness proofs may also not be easy. If we have no clue, we usually choose to debug the code based on test cases, step by step modifying and verifying the greedy strategy.

15.1.4 Typical Problems Solved by Greedy Algorithms

Greedy algorithms are often applied to optimization problems that satisfy greedy choice property and optimal substructure. Below are some typical greedy algorithm problems.

- **Coin change problem**: With certain coin combinations, greedy algorithms can always obtain the optimal solution.

- **Interval scheduling problem**: Suppose you have some tasks, each taking place during a period of time, and your goal is to complete as many tasks as possible. If you always choose the task that ends earliest, then the greedy algorithm can obtain the optimal solution.
- **Fractional knapsack problem**: Given a set of items and a carrying capacity, your goal is to select a set of items such that the total weight does not exceed the carrying capacity and the total value is maximized. If you always choose the item with the highest value-to-weight ratio (value / weight), then the greedy algorithm can obtain the optimal solution in some cases.
- **Stock trading problem**: Given a set of historical stock prices, you can make multiple trades, but if you already hold stocks, you cannot buy again before selling, and the goal is to obtain the maximum profit.
- **Huffman coding**: Huffman coding is a greedy algorithm used for lossless data compression. By constructing a Huffman tree and always merging the two nodes with the lowest frequency, the resulting Huffman tree has the minimum weighted path length (encoding length).
- **Dijkstra's algorithm**: It is a greedy algorithm for solving the shortest path problem from a given source vertex to all other vertices.

15.2 Fractional Knapsack Problem

Question

Given n items, where the weight of the i-th item is $wgt[i-1]$ and its value is $val[i-1]$, and a knapsack with capacity cap. Each item can be selected only once, **but a portion of an item can be selected, with the value calculated based on the proportion of weight selected**, what is the maximum value of items in the knapsack under the limited capacity? An example is shown in Figure 15-3.

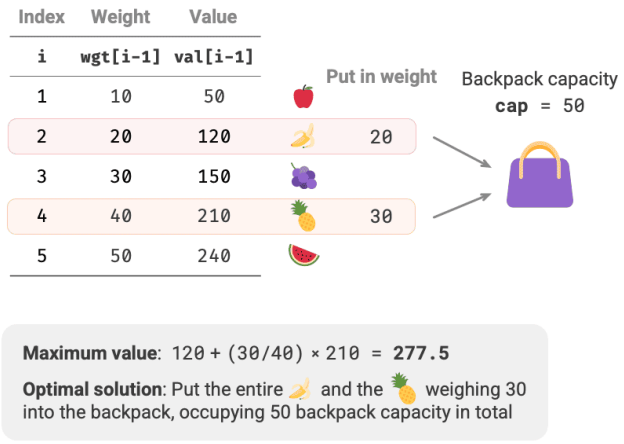


Figure 15-3 Example data for the fractional knapsack problem

The fractional knapsack problem is very similar overall to the 0-1 knapsack problem, with states including the current item i and capacity c, and the goal being to maximize value under the limited knapsack capacity.

The difference is that this problem allows selecting only a portion of an item. As shown in Figure 15-4, **we can arbitrarily split items and calculate the corresponding value based on the weight proportion**.

1. For item i, its value per unit weight is $val[i-1]/wgt[i-1]$, referred to as unit value.
2. Suppose we put a portion of item i with weight w into the knapsack, then the value added to the knapsack is $w \times val[i-1]/wgt[i-1]$.

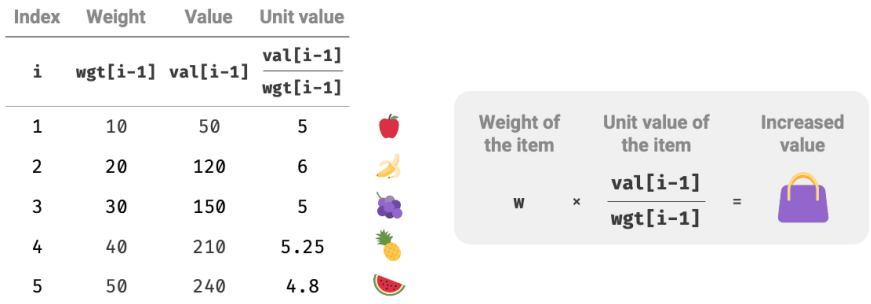


Figure 15-4 Value of items per unit weight

1. Greedy Strategy Determination

Maximizing the total value of items in the knapsack **is essentially maximizing the value per unit weight of items**. From this, we can derive the greedy strategy shown in Figure 15-5.

1. Sort items by unit value from high to low.
2. Iterate through all items, **greedily selecting the item with the highest unit value in each round**.
3. If the remaining knapsack capacity is insufficient, use a portion of the current item to fill the knapsack.

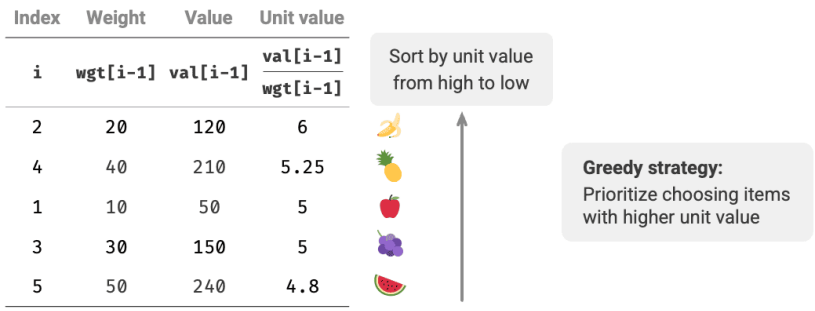


Figure 15-5 Greedy strategy for the fractional knapsack problem

2. Code Implementation

We created an `Item` class to facilitate sorting items by unit value. We loop to make greedy selections, breaking when the knapsack is full and returning the solution:

```javascript
// == File: fractional_knapsack.js ==

/* Item */
class Item {
    constructor(w, v) {
        this.w = w; // Item weight
        this.v = v; // Item value
    }
}

/* Fractional knapsack: Greedy algorithm */
function fractionalKnapsack(wgt, val, cap) {
    // Create item list with two attributes: weight, value
    const items = wgt.map((w, i) => new Item(w, val[i]));
    // Sort by unit value item.v / item.w from high to low
    items.sort((a, b) => b.v / b.w - a.v / a.w);
    // Loop for greedy selection
    let res = 0;
    for (const item of items) {
        if (item.w <= cap) {
            // If remaining capacity is sufficient, put the entire current item into the knapsack
            res += item.v;
            cap -= item.w;
        } else {
            // If remaining capacity is insufficient, put part of the current item into the
            ↪ knapsack
            res += (item.v / item.w) * cap;
            // No remaining capacity, so break out of the loop
            break;
        }
    }
    return res;
}
```

The time complexity of built-in sorting algorithms is usually $O(\log n)$, and the space complexity is usually $O(\log n)$ or $O(n)$, depending on the specific implementation of the programming language.

Apart from sorting, in the worst case the entire item list needs to be traversed, **therefore the time complexity is** $O(n)$, where n is the number of items.

Since an `Item` object list is initialized, **the space complexity is** $O(n)$.

3. Correctness Proof

Using proof by contradiction. Suppose item x has the highest unit value, and some algorithm yields a maximum value of `res`, but this solution does not include item x.

Now remove a unit weight of any item from the knapsack and replace it with a unit weight of item x. Since item x has the highest unit value, the total value after replacement will definitely be greater

than `res`. **This contradicts the assumption that `res` is the optimal solution, proving that the optimal solution must include item x.**

For other items in this solution, we can also construct the above contradiction. In summary, **items with greater unit value are always better choices**, which proves that the greedy strategy is effective.

As shown in Figure 15-6, if we view item weight and item unit value as the horizontal and vertical axes of a two-dimensional chart respectively, then the fractional knapsack problem can be transformed into "finding the maximum area enclosed within a limited horizontal axis range". This analogy can help us understand the effectiveness of the greedy strategy from a geometric perspective.

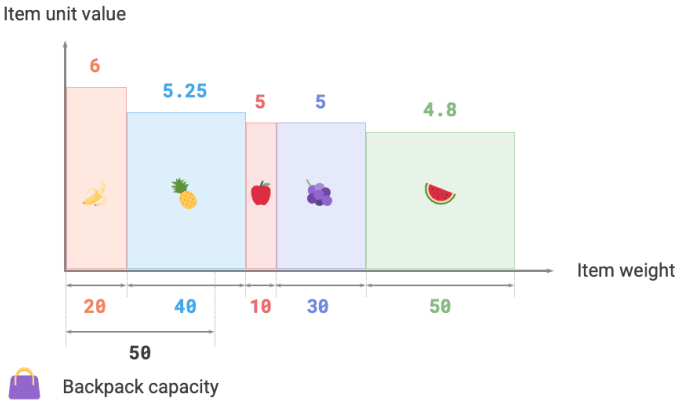


Figure 15-6 Geometric representation of the fractional knapsack problem

15.3 Max Capacity Problem

Question

Input an array ht, where each element represents the height of a vertical partition. Any two partitions in the array, along with the space between them, can form a container.

The capacity of the container equals the product of height and width (area), where the height is determined by the shorter partition, and the width is the difference in array indices between the two partitions.

Please select two partitions in the array such that the capacity of the formed container is maximized, and return the maximum capacity. An example is shown in Figure 15-7.

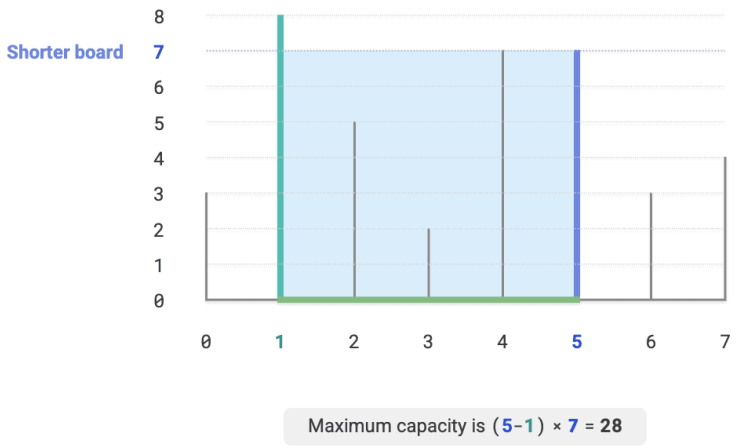


Figure 15-7 Example data for the max capacity problem

The container is formed by any two partitions, **therefore the state of this problem is the indices of two partitions, denoted as** $[i, j]$.

According to the problem description, capacity equals height multiplied by width, where height is determined by the shorter partition, and width is the difference in array indices between the two partitions. Let the capacity be $cap[i, j]$, then the calculation formula is:

$$cap[i, j] = \min(ht[i], ht[j]) \times (j - i)$$

Let the array length be n, then the number of combinations of two partitions (total number of states) is $C_n^2 = \frac{n(n-1)}{2}$. Most directly, **we can exhaustively enumerate all states** to find the maximum capacity, with time complexity $O(n^2)$.

1. Greedy Strategy Determination

This problem has a more efficient solution. As shown in Figure 15-8, select a state $[i, j]$ where index $i < j$ and height $ht[i] < ht[j]$, meaning i is the short partition and j is the long partition.

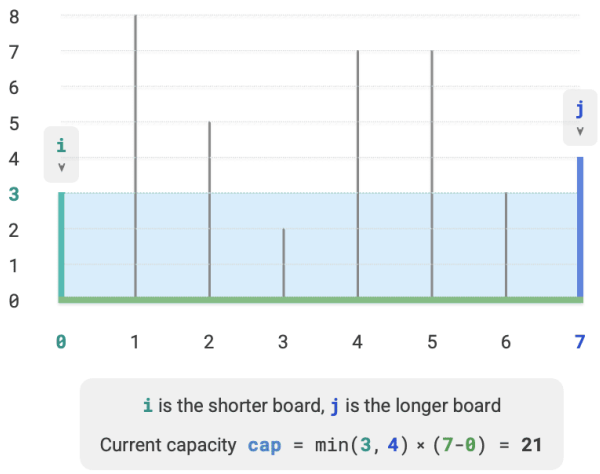


Figure 15-8 Initial state

As shown in Figure 15-9, **if we now move the long partition j closer to the short partition i, the capacity will definitely decrease**.

This is because after moving the long partition j, the width $j - i$ definitely decreases; and since height is determined by the short partition, the height can only remain unchanged (i is still the short partition) or decrease (the moved j becomes the short partition).

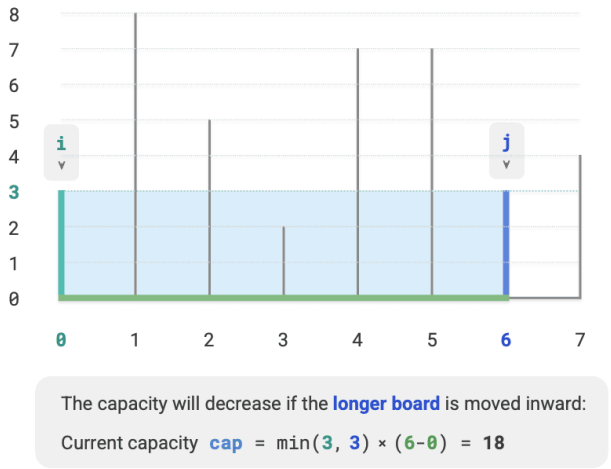


Figure 15-9 State after moving the long partition inward

Conversely, **we can only possibly increase capacity by contracting the short partition i inward**. Because although width will definitely decrease, **height may increase** (the moved short partition i may become taller). For example, in Figure 15-10, the area increases after moving the short partition.

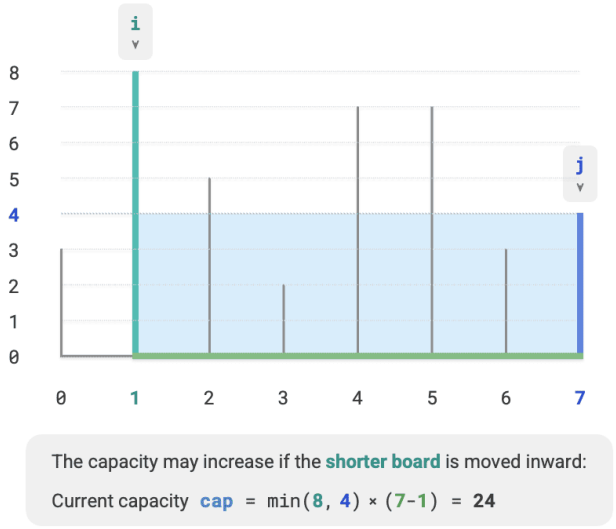


Figure 15-10 State after moving the short partition inward

From this we can derive the greedy strategy for this problem: initialize two pointers at both ends of the container, and in each round contract the pointer corresponding to the short partition inward, until the two pointers meet.

Figure 15-11 shows the execution process of the greedy strategy.

1. In the initial state, pointers i and j are at both ends of the array.
2. Calculate the capacity of the current state $cap[i, j]$, and update the maximum capacity.
3. Compare the heights of partition i and partition j, and move the short partition inward by one position.
4. Loop through steps `2.` and `3.` until i and j meet.

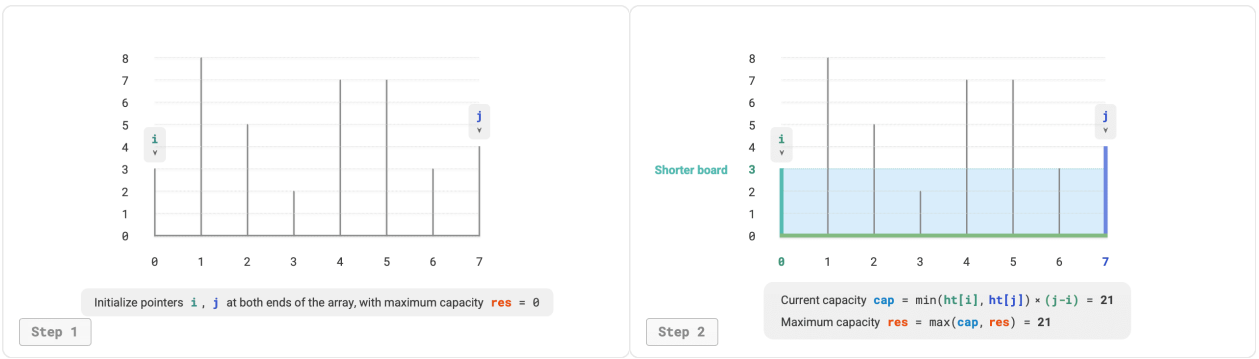


Figure 15-11 Greedy process for the max capacity problem

2. Code Implementation

The code loops at most n rounds, **therefore the time complexity is $O(n)$**.

Variables i, j, and res use a constant amount of extra space, **therefore the space complexity is $O(1)$**.

```javascript
// == File: max_capacity.js ==

/* Max capacity: Greedy algorithm */
function maxCapacity(ht) {
    // Initialize i, j to be at both ends of the array
    let i = 0,
        j = ht.length - 1;
    // Initial max capacity is 0
    let res = 0;
    // Loop for greedy selection until the two boards meet
    while (i < j) {
        // Update max capacity
        const cap = Math.min(ht[i], ht[j]) * (j - i);
        res = Math.max(res, cap);
        // Move the shorter board inward
        if (ht[i] < ht[j]) {
            i += 1;
        } else {
            j -= 1;
        }
    }
    return res;
}
```

3. Correctness Proof

The reason greedy is faster than exhaustive enumeration is that each round of greedy selection "skips" some states.

For example, in state $cap[i, j]$ where i is the short partition and j is the long partition, if we greedily move the short partition i inward by one position, the states shown in Figure 15-12 will be "skipped". **This means that the capacities of these states cannot be verified later**.

$$cap[i, i+1], cap[i, i+2], \dots, cap[i, j-2], cap[i, j-1]$$

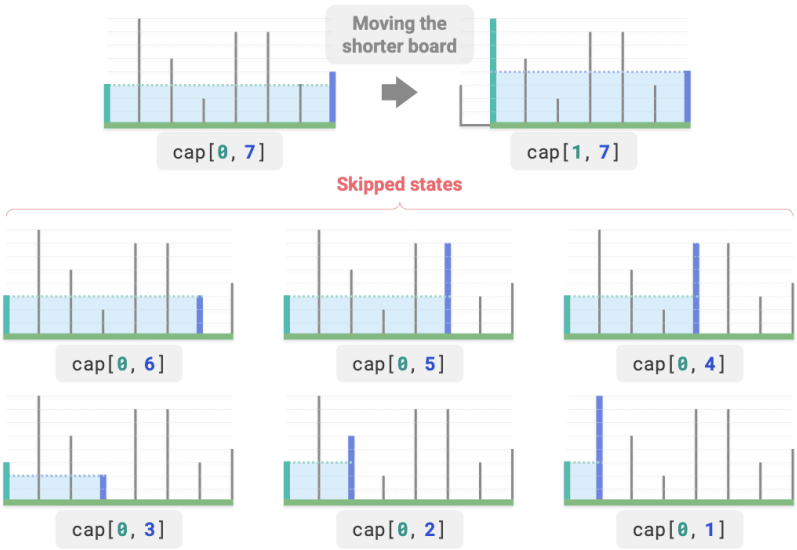


Figure 15-12 States skipped by moving the short partition

Observing carefully, **these skipped states are actually all the states obtained by moving the long partition j inward**. We have already proven that moving the long partition inward will definitely decrease capacity. That is, the skipped states cannot possibly be the optimal solution, **skipping them will not cause us to miss the optimal solution**.

The above analysis shows that the operation of moving the short partition is "safe", and the greedy strategy is effective.

15.4 Max Product Cutting Problem

Question

Given a positive integer n, split it into the sum of at least two positive integers, and find the maximum product of all integers after splitting, as shown in Figure 15-13.

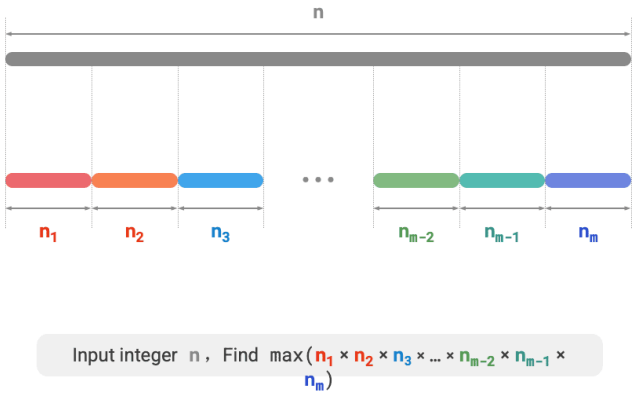


Figure 15-13 Problem definition of max product cutting

Suppose we split n into m integer factors, where the i-th factor is denoted as n_i, that is

$$n = \sum_{i=1}^{m} n_i$$

The goal of this problem is to find the maximum product of all integer factors, namely

$$\max(\prod_{i=1}^{m} n_i)$$

We need to think about: how large should the splitting count m be, and what should each n_i be?

1. Greedy Strategy Determination

Based on experience, the product of two integers is often greater than their sum. Suppose we split out a factor of 2 from n, then their product is $2(n-2)$. We compare this product with n:

$$
\begin{aligned}
2(n-2) &\geq n \\
2n-n-4 &\geq 0 \\
n &\geq 4
\end{aligned}
$$

As shown in Figure 15-14, when $n \geq 4$, splitting out a 2 will increase the product, **which indicates that integers greater than or equal to 4 should all be split**.

Greedy strategy one: If the splitting scheme includes factors ≥ 4, then they should continue to be split. The final splitting scheme should only contain factors 1, 2, and 3.

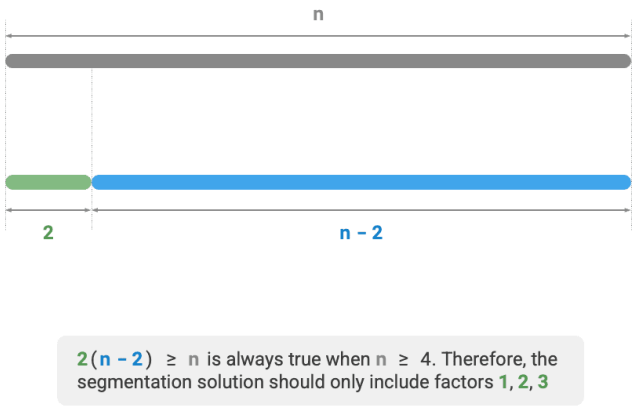


Figure 15-14 Splitting causes product to increase

Next, consider which factor is optimal. Among the three factors 1, 2, and 3, clearly 1 is the worst, because $1 \times (n-1) < n$ always holds, meaning splitting out 1 will actually decrease the product.

As shown in Figure 15-15, when $n = 6$, we have $3 \times 3 > 2 \times 2 \times 2$. **This means that splitting out 3 is better than splitting out 2.**

Greedy strategy two: In the splitting scheme, there should be at most two 2s. Because three 2s can always be replaced by two 3s to obtain a larger product.

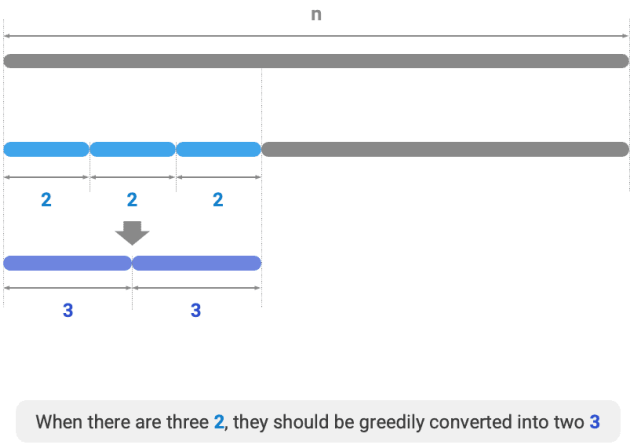


Figure 15-15 Optimal splitting factor

In summary, the following greedy strategies can be derived.

1. Input integer n, continuously split out factor 3 until the remainder is 0, 1, or 2.
2. When the remainder is 0, it means n is a multiple of 3, so no further action is needed.
3. When the remainder is 2, do not continue splitting, keep it.
4. When the remainder is 1, since $2 \times 2 > 1 \times 3$, the last 3 should be replaced with 2.

2. Code Implementation

As shown in Figure 15-16, we don't need to use loops to split the integer, but can use integer division to get the count of 3s as a, and modulo operation to get the remainder as b, at which point we have:

$$n = 3a + b$$

Please note that for the edge case of $n \leq 3$, a 1 must be split out, with product $1 \times (n - 1)$.

```
// ═══ File: max_product_cutting.js ═══

/* Max product cutting: Greedy algorithm */
function maxProductCutting(n) {
    // When n <= 3, must cut out a 1
    if (n <= 3) {
        return 1 * (n - 1);
    }
    // Greedily cut out 3, a is the number of 3s, b is the remainder
    let a = Math.floor(n / 3);
    let b = n % 3;
    if (b === 1) {
        // When the remainder is 1, convert a pair of 1 * 3 to 2 * 2
        return Math.pow(3, a - 1) * 2 * 2;
    }
    if (b === 2) {
```

```
        // When the remainder is 2, do nothing
        return Math.pow(3, a) * 2;
    }
    // When the remainder is 0, do nothing
    return Math.pow(3, a);
}
```

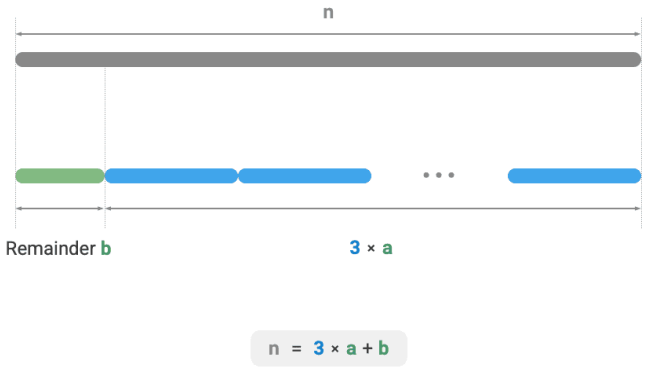


Figure 15-16 Calculation method for max product cutting

The time complexity depends on the implementation of the exponentiation operation in the programming language. Taking Python as an example, there are three commonly used power calculation functions.

- Both the operator `**` and the function `pow()` have time complexity $O(\log\ a)$.
- The function `math.pow()` internally calls the C library's `pow()` function, which performs floating-point exponentiation, with time complexity $O(1)$.

Variables a and b use a constant amount of extra space, **therefore the space complexity is** $O(1)$.

3. Correctness Proof

Using proof by contradiction, only analyzing the case where $n \geq 4$.

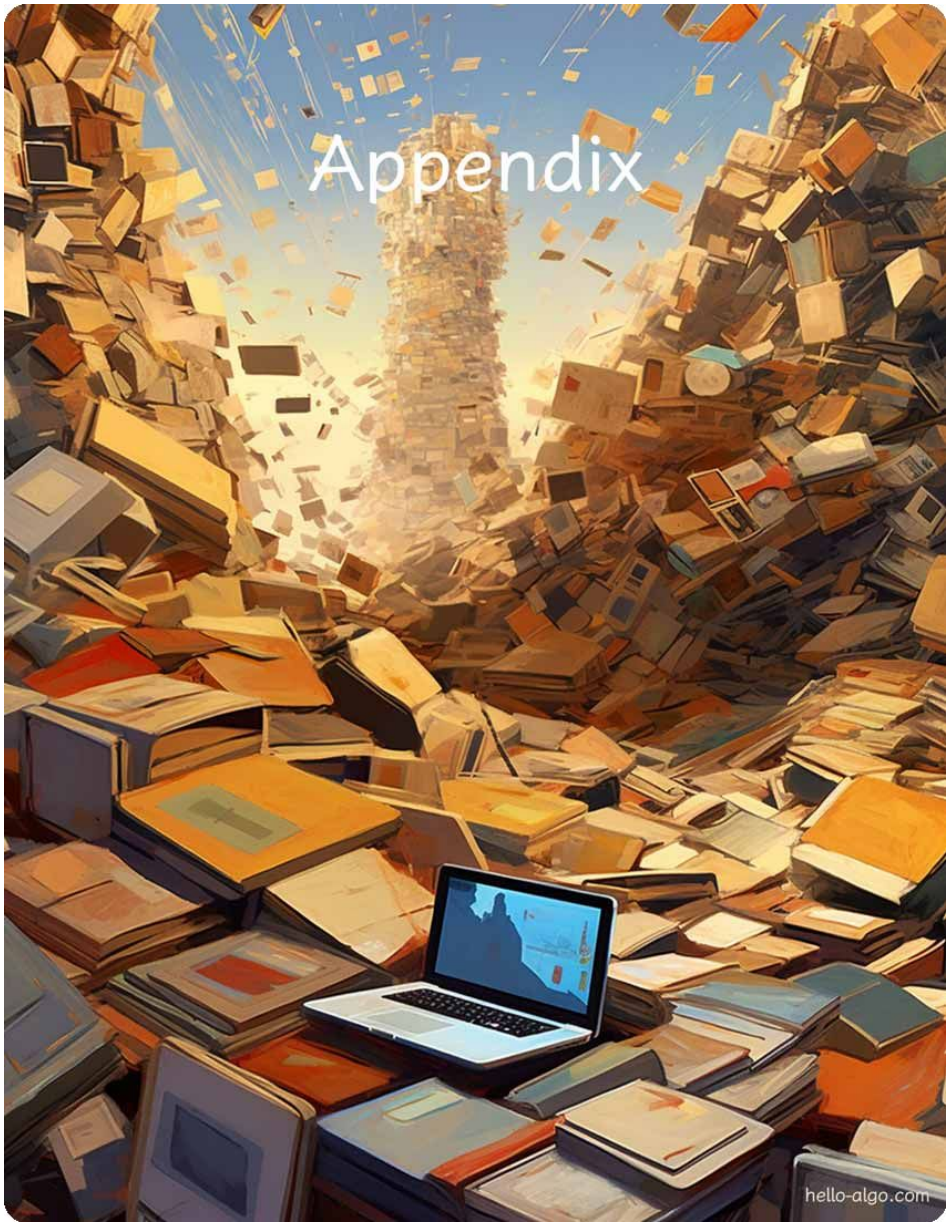
1. **All factors** ≤ 3: Suppose the optimal splitting scheme includes a factor $x \geq 4$, then it can definitely continue to be split into $2(x-2)$ to obtain a larger (or equal) product. This contradicts the assumption.
2. **The splitting scheme does not contain** 1: Suppose the optimal splitting scheme includes a factor of 1, then it can definitely be merged into another factor to obtain a larger product. This contradicts the assumption.
3. **The splitting scheme contains at most two** 2**s**: Suppose the optimal splitting scheme includes three 2s, then they can definitely be replaced by two 3s for a larger product. This contradicts the assumption.

15.5 Summary

1. Key Review

- Greedy algorithms are typically used to solve optimization problems. The principle is to make locally optimal decisions at each decision stage in hopes of obtaining a globally optimal solution.
- Greedy algorithms iteratively make one greedy choice after another, transforming the problem into a smaller subproblem in each round, until the problem is solved.
- Greedy algorithms are not only simple to implement, but also have high problem-solving efficiency. Compared to dynamic programming, greedy algorithms typically have lower time complexity.
- In the coin change problem, for certain coin combinations, greedy algorithms can guarantee finding the optimal solution; for other coin combinations, however, greedy algorithms may find very poor solutions.
- Problems suitable for solving with greedy algorithms have two major properties: greedy choice property and optimal substructure. The greedy choice property represents the effectiveness of the greedy strategy.
- For some complex problems, proving the greedy choice property is not simple. Relatively speaking, disproving it is easier, such as in the coin change problem.
- Solving greedy problems mainly consists of three steps: problem analysis, determining the greedy strategy, and correctness proof. Among these, determining the greedy strategy is the core step, and correctness proof is often the difficult point.
- The fractional knapsack problem, based on the 0-1 knapsack problem, allows selecting a portion of items, and therefore can be solved using greedy algorithms. The correctness of the greedy strategy can be proven using proof by contradiction.
- The max capacity problem can be solved using exhaustive enumeration with time complexity $O(n^2)$. By designing a greedy strategy to move the short partition inward in each round, the time complexity can be optimized to $O(n)$.
- In the max product cutting problem, we successively derive two greedy strategies: integers ≥ 4 should all continue to be split, and the optimal splitting factor is 3. The code includes exponentiation operations, and the time complexity depends on the implementation method of exponentiation, typically being $O(1)$ or $O(\log n)$.

Chapter 16. Appendix



16.1 Programming Environment Installation

16.1.1 Installing Ide

We recommend using the open-source and lightweight VS Code as the local integrated development environment (IDE). Visit the VS Code official website, and download and install the appropriate version of VS Code according to your operating system.

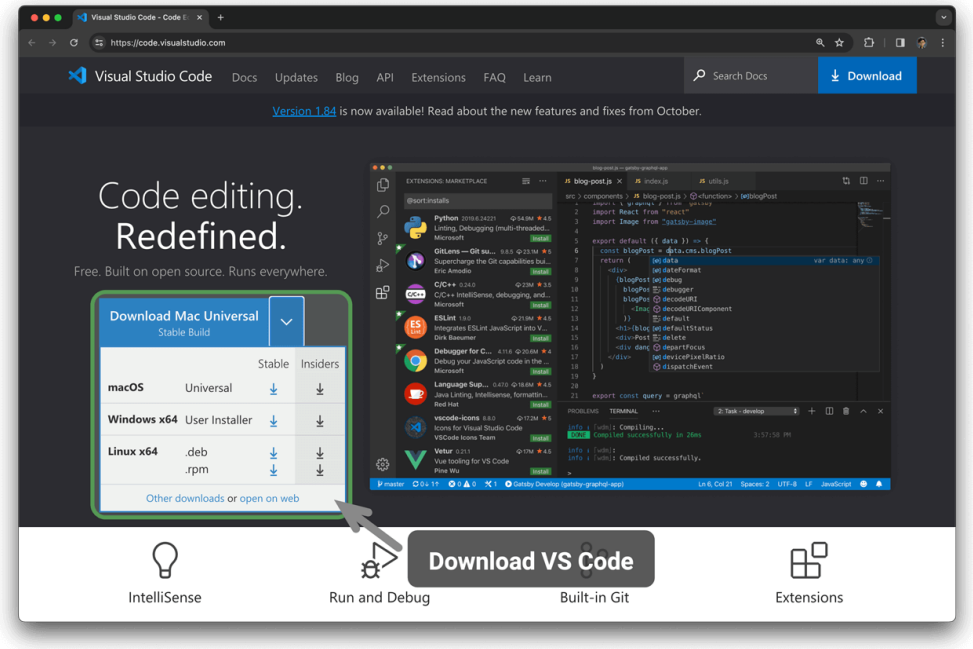


Figure 16-1 Download VS Code from the Official Website

VS Code has a powerful ecosystem of extensions that supports running and debugging most programming languages. For example, after installing the “Python Extension Pack” extension, you can debug Python code. The installation steps are shown in the following figure.

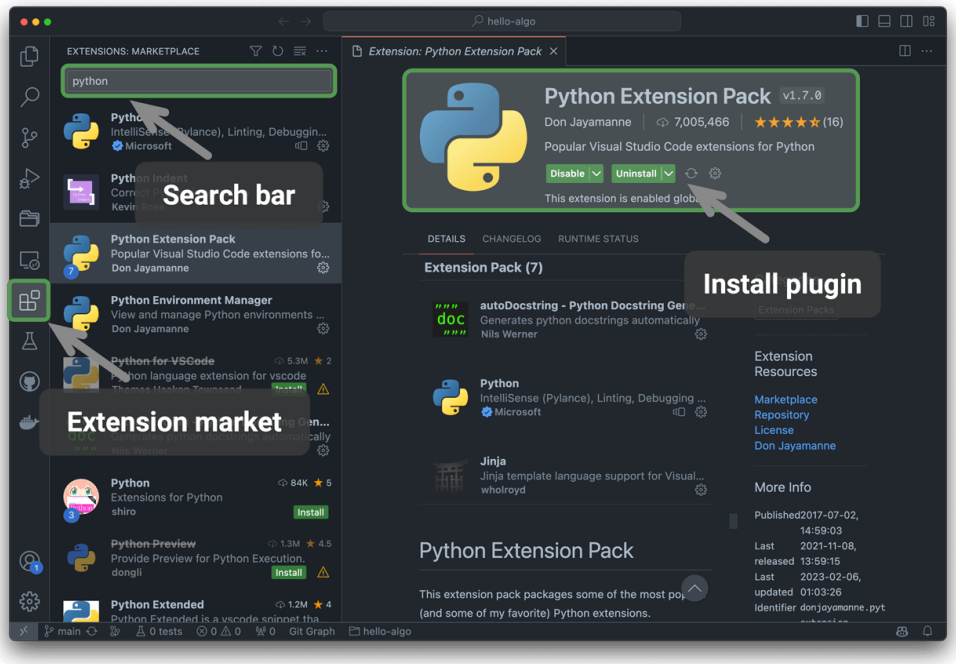


Figure 16-2 Install VS Code Extensions

16.1.2 Installing Language Environments

1. Python Environment

1. Download and install Miniconda3, which requires Python 3.10 or newer.
2. Search for `python` in the VS Code extension marketplace and install the Python Extension Pack.
3. (Optional) Enter `pip install black` on the command line to install the code formatter.

2. C/c++ Environment

1. Windows systems need to install MinGW (configuration tutorial); macOS comes with Clang built-in and does not require installation.
2. Search for `c++` in the VS Code extension marketplace and install the C/C++ Extension Pack.
3. (Optional) Open the Settings page, search for the `Clang_format_fallback Style` code formatting option, and set it to `{ BasedOnStyle: Microsoft, BreakBeforeBraces: Attach }`.

3. Java Environment

1. Download and install OpenJDK (version must be > JDK 9).
2. Search for `java` in the VS Code extension marketplace and install the Extension Pack for Java.

4. C# Environment

1. Download and install .Net 8.0.
2. Search for `C# Dev Kit` in the VS Code extension marketplace and install C# Dev Kit (configuration tutorial).
3. You can also use Visual Studio (installation tutorial).

5. Go Environment

1. Download and install Go.
2. Search for `go` in the VS Code extension marketplace and install Go.
3. Press `Ctrl + Shift + P` to open the command palette, type `go`, select `Go: Install/Update Tools`, check all options and install.

6. Swift Environment

1. Download and install Swift.
2. Search for `swift` in the VS Code extension marketplace and install Swift for Visual Studio Code.

7. Javascript Environment

1. Download and install Node.js.
2. (Optional) Search for `Prettier` in the VS Code extension marketplace and install the code formatter.

8. Typescript Environment

1. Follow the same installation steps as the JavaScript environment.
2. Install TypeScript Execute (tsx).
3. Search for `typescript` in the VS Code extension marketplace and install Pretty TypeScript Errors.

9. Dart Environment

1. Download and install Dart.
2. Search for `dart` in the VS Code extension marketplace and install Dart.

10. Rust Environment

1. Download and install Rust.
2. Search for `rust` in the VS Code extension marketplace and install rust-analyzer.

16.2 Contributing Together

Due to limited capacity, there may be inevitable omissions and errors in this book. We appreciate your understanding and are grateful for your help in correcting them. If you discover typos, broken links, missing content, ambiguous wording, unclear explanations, or structural issues, please help us make corrections to provide readers with higher-quality learning resources.

The GitHub IDs of all contributors will be displayed on the homepage of the book repository, the web version, and the PDF version to acknowledge their selfless contributions to the open source community.

The Charm of Open Source
The interval between two printings of a physical book is often quite long, making content updates very inconvenient.
In this open source book, the time for content updates has been shortened to just days or even hours.

1. Minor Content Adjustments

As shown in Figure 16-3, there is an "edit icon" in the top-right corner of each page. You can modify text or code by following these steps.

1. Click the "edit icon". If you encounter a prompt asking you to "Fork this repository", please approve the operation.
2. Modify the content of the Markdown source file, verify the correctness of the content, and maintain consistent formatting as much as possible.
3. Fill in a description of your changes at the bottom of the page, then click the "Propose file change" button. After the page transitions, click the "Create pull request" button to submit your pull request.

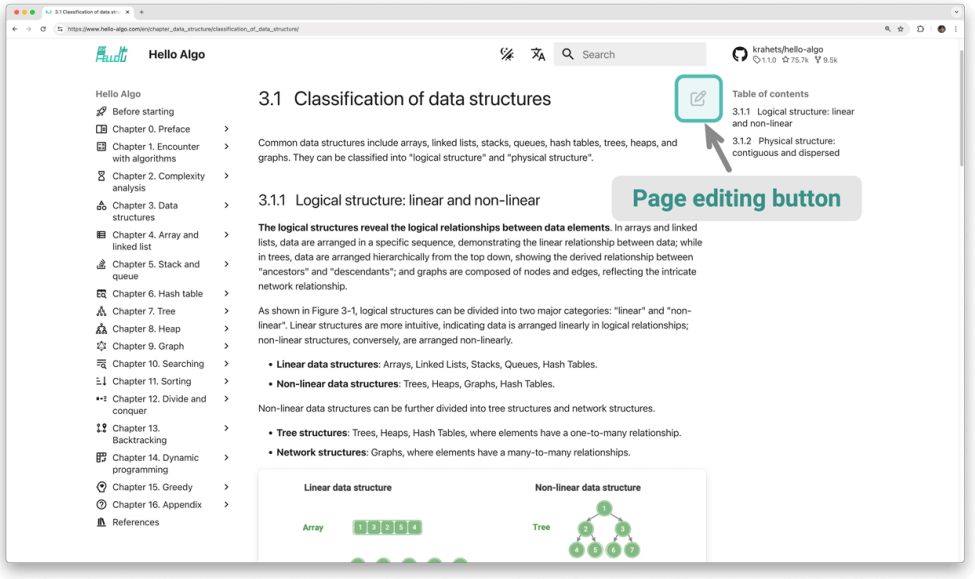


Figure 16-3 Page edit button

Images cannot be directly modified. Please describe the issue by creating a new Issue or leaving a comment. We will promptly redraw and replace the images.

2. Content Creation

If you are interested in contributing to this open source project, including translating code into other programming languages or expanding article content, you will need to follow the Pull Request workflow below.

1. Log in to GitHub and Fork the book's code repository to your personal account.
2. Enter your forked repository webpage and use the `git clone` command to clone the repository to your local machine.
3. Create content locally and conduct comprehensive tests to verify code correctness.
4. Commit your local changes and push them to the remote repository.
5. Refresh the repository webpage and click the "Create pull request" button to submit your pull request.

3. Docker Deployment

From the root directory of `hello-algo`, run the following Docker script to access the project at `http://localhost:8000`:

```
docker-compose up -d
```

Use the following command to remove the deployment:

```
docker-compose down
```

16.3 Terminology Table

The following table lists important terms that appear in this book. It is worth noting the following points:

- We recommend remembering the English names of terms to help with reading English literature.
- Some terms have different names in Simplified Chinese and Traditional Chinese.

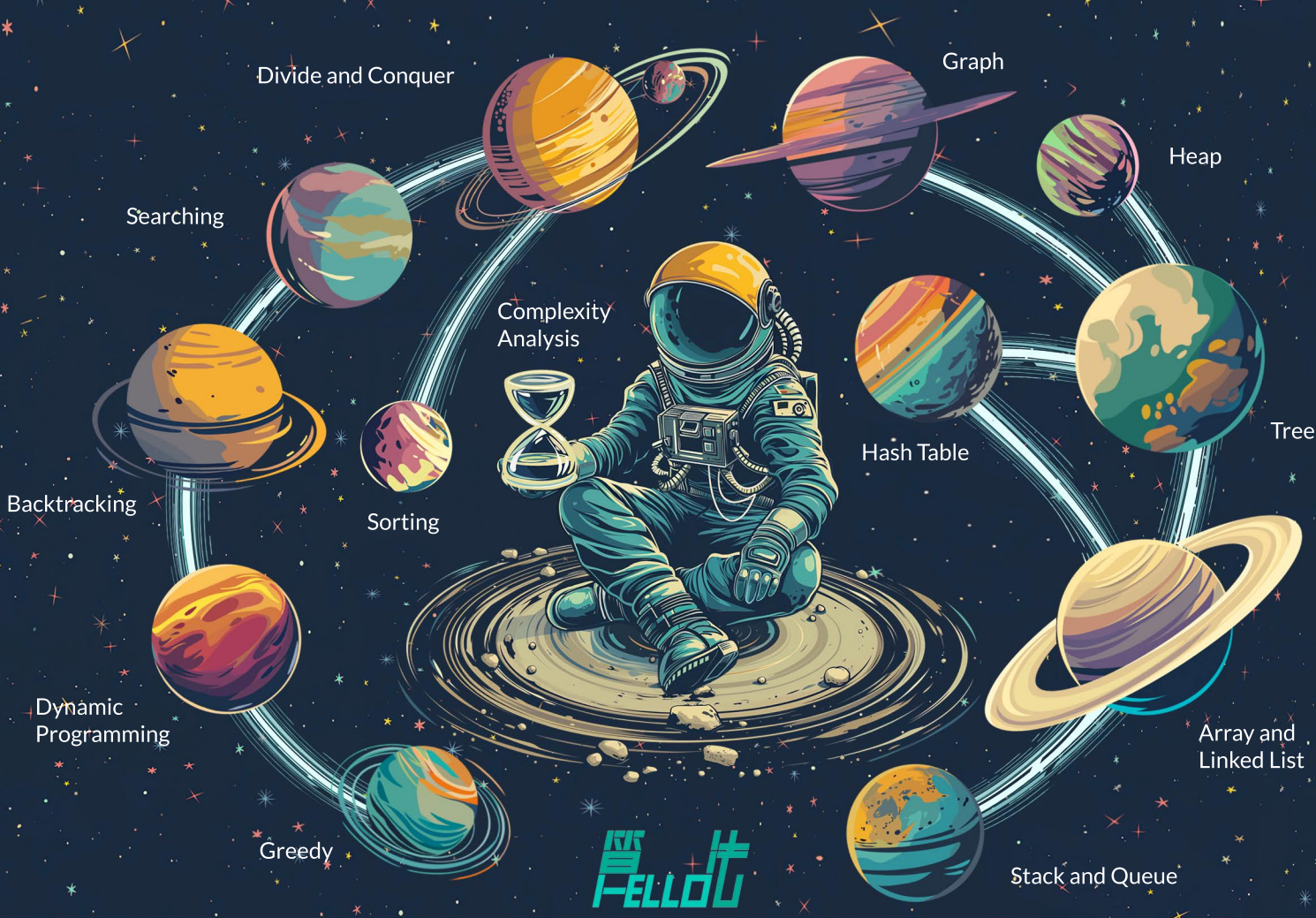
Table 16-1 Important Terms in Data Structures and Algorithms

English	Simplified Chinese	Traditional Chinese
algorithm	算法	演算法
data structure	数据结构	資料結構
code	代码	程式碼
file	文件	檔案
function	函数	函式
method	方法	方法
variable	变量	變數
asymptotic complexity analysis	渐近复杂度分析	漸近複雜度分析
time complexity	时间复杂度	時間複雜度
space complexity	空间复杂度	空間複雜度
loop	循环	迴圈
iteration	迭代	迭代
recursion	递归	遞迴
tail recursion	尾递归	尾遞迴
recursion tree	递归树	遞迴樹
big-O notation	大 O 记号	大 O 記號
asymptotic upper bound	渐近上界	漸近上界
sign-magnitude	原码	原碼
1’s complement	反码	一補數
2’s complement	补码	二補數
array	数组	陣列
index	索引	索引
linked list	链表	鏈結串列

English	Simplified Chinese	Traditional Chinese
linked list node, list node	链表节点	鏈結串列節點
head node	头节点	頭節點
tail node	尾节点	尾節點
list	列表	串列
dynamic array	动态数组	動態陣列
hard disk	硬盘	硬碟
random-access memory (RAM)	内存	記憶體
cache memory	缓存	快取
cache miss	缓存未命中	快取未命中
cache hit rate	缓存命中率	快取命中率
stack	栈	堆疊
top of the stack	栈顶	堆疊頂
bottom of the stack	栈底	堆疊底
queue	队列	佇列
double-ended queue	双向队列	雙向佇列
front of the queue	队首	佇列首
rear of the queue	队尾	佇列尾
hash table	哈希表	雜湊表
hash set	哈希集合	雜湊集合
bucket	桶	桶
hash function	哈希函数	雜湊函式
hash collision	哈希冲突	雜湊衝突
load factor	负载因子	負載因子
separate chaining	链式地址	鏈結位址
open addressing	开放寻址	開放定址
linear probing	线性探测	線性探查
lazy deletion	懒删除	懶刪除
binary tree	二叉树	二元樹
tree node	树节点	樹節點
left-child node	左子节点	左子節點
right-child node	右子节点	右子節點
parent node	父节点	父節點
left subtree	左子树	左子樹
right subtree	右子树	右子樹
root node	根节点	根節點

English	Simplified Chinese	Traditional Chinese
leaf node	叶节点	葉節點
edge	边	邊
level	层	層
degree	度	度
height	高度	高度
depth	深度	深度
perfect binary tree	完美二叉树	完美二元樹
complete binary tree	完全二叉树	完全二元樹
full binary tree	完满二叉树	完滿二元樹
balanced binary tree	平衡二叉树	平衡二元樹
binary search tree	二叉搜索树	二元搜尋樹
AVL tree	AVL 树	AVL 樹
red-black tree	红黑树	紅黑樹
level-order traversal	层序遍历	層序走訪
breadth-first traversal	广度优先遍历	廣度優先走訪
depth-first traversal	深度优先遍历	深度優先走訪
binary search tree	二叉搜索树	二元搜尋樹
balanced binary search tree	平衡二叉搜索树	平衡二元搜尋樹
balance factor	平衡因子	平衡因子
heap	堆	堆積
max heap	大顶堆	大頂堆積
min heap	小顶堆	小頂堆積
priority queue	优先队列	優先佇列
heapify	堆化	堆積化
top-k problem	Top-k 问题	Top-k 問題
graph	图	圖
vertex	顶点	頂點
undirected graph	无向图	無向圖
directed graph	有向图	有向圖
connected graph	连通图	連通圖
disconnected graph	非连通图	非連通圖
weighted graph	有权图	有權圖
adjacency	邻接	鄰接
path	路径	路徑
in-degree	入度	入度

English	Simplified Chinese	Traditional Chinese
out-degree	出度	出度
adjacency matrix	邻接矩阵	鄰接矩陣
adjacency list	邻接表	鄰接表
breadth-first search	广度优先搜索	廣度優先搜尋
depth-first search	深度优先搜索	深度優先搜尋
binary search	二分查找	二分搜尋
searching algorithm	搜索算法	搜尋演算法
sorting algorithm	排序算法	排序演算法
selection sort	选择排序	選擇排序
bubble sort	冒泡排序	泡沫排序
insertion sort	插入排序	插入排序
quick sort	快速排序	快速排序
merge sort	归并排序	合併排序
heap sort	堆排序	堆積排序
bucket sort	桶排序	桶排序
counting sort	计数排序	計數排序
radix sort	基数排序	基數排序
divide and conquer	分治	分治
hanota problem	汉诺塔问题	河內塔問題
backtracking algorithm	回溯算法	回溯演算法
constraint	约束	約束
solution	解	解
state	状态	狀態
pruning	剪枝	剪枝
permutations problem	全排列问题	全排列問題
subset-sum problem	子集和问题	子集合問題
n-queens problem	n 皇后问题	n 皇后問題
dynamic programming	动态规划	動態規劃
initial state	初始状态	初始狀態
state-transition equation	状态转移方程	狀態轉移方程
knapsack problem	背包问题	背包問題
edit distance problem	编辑距离问题	編輯距離問題
greedy algorithm	贪心算法	貪婪演算法



“Among the universe's 200 billion galaxies, encountering you, a shining star, is this book's great fortune.”