4. Algorithmic Thinking

So if an algorithm is an idealized recipe, a program is the detailed set of instructions for a cooking robot preparing a month of meals for an army while under enemy attack.

Brian Kernighan, 2017

Algorithmic thinking is concerned with solving problems **smartly**, by designing and using algorithms. We look at the world through an algorithmic lens.

A problem is specified by rigorously specifying the input and the desired output. An algorithm is a set of rules specifying the sequences of computational steps for solving a specific problem. That is, for any given input data, the algorithm produces the desired output data. Thus, an algorithm is specified as follows:

A Specific Algorithm
- **Input**: specifying the given input data.
- **Output**: specifying the desired output data.
- **Steps**: specifying the sequence of computational steps.

What exactly does *smart* mean in solving problems *smartly*? The following four characteristics of algorithmic thinking are noteworthy. Discussing these four characteristics constitutes the main contents of this chapter.

- A smart way to **define** algorithms. Donald Knuth gives a five-point definition of algorithms. Here, smartness manifests as simplicity. This definition captures the essence of algorithms, is extremely simple, yet universally applicable. Also, the simple definition makes it easy to check if a sequence of steps is an algorithm.
- A smart way to **measure** algorithms. We use asymptotic notations and asymptotic analysis methods to measure and analyze the time and space complexities of algorithms. This asymptotic way avoids many irrelevant details and idiosyncrasies. It also reveals an important division of the hardness of computational problems: the tractable (called P) and the intractable (called NP).
- Smart paradigms to **design** algorithms. We discuss several representative paradigms to reveal concrete skills and crafts, including divide-and-conquer, dynamic programing and greedy paradigms. They help design clever and much faster algorithms.
- Smart variations to **adapt** for problem nuances. Here, smartness manifests as flexibility. Problem nuances are utilized to increase algorithmic efficiency.

4.1. What Are Algorithms

We first discuss the algorithm definition and how to measure algorithms. The bubble sort algorithm is used as an illustrative example. In sections 4.2 and 4.3, we introduce the design and analysis of some representative algorithms.

4.1.1. Knuth's Characterization of Algorithm

In his seminal work *The Art of Computer Programming*, Donald Knuth proposed a five-point definition of algorithm, which has been widely accepted and used.

Definition. An **algorithm** is a finite set of rules specifying sequences of computational steps for solving a given problem, with the following five properties.

- *Finiteness*. An algorithm must always terminate after a finite number of steps.
- *Definiteness*. Each step of an algorithm must be precisely defined, that is, the actions to be carried out must be rigorously and unambiguously specified.
- *Input*. An algorithm has zero or more inputs, given before the algorithm begins or during the algorithm's execution.
- *Output*. An algorithm has one or more outputs, which relate to the inputs.
- *Effectiveness*. Every operation of an algorithm must be sufficiently rudimentary, such that in principle, the operation can be done by a human using paper and pencil, in finite time.

From the algorithmic lens, a problem is often specified as follows: design an algorithm according to Knuth's definition, such that for any given input data, it produces the desired output data. The algorithm is specified as follows, where the Steps part must satisfy the five properties in Knuth's definition.

- **Input**: specifying the given input data.
- **Output**: specifying the desired output data.
- **Steps**: specifying one or more sequences of computational steps.

Students can use a programming language to specify an algorithm. In fact, such a specification is more than a specification, but also an implementation of the algorithm, because the program can be compiled and automatically executed on a computer.

However, the chapter quotation tells us that an algorithm is not the same as a program. The quicksort algorithm was discovered before the invention of the Go programming language. Many algorithms were designed and used long before the modern computer era.

Algorithms represent essential ideas of programs. They are sufficiently detailed (Knuth's five points) to ensure that they are step-by-step procedures, but ignore many syntactic and semantic details of any particular programming language. In the design and analysis of algorithms, people often use **pseudocode**, i.e., some form of high-level natural language mixing mathematic notations, to specify an algorithm. This chapter follows this practice.

Example 30. Algorithm versus non-algorithm

Consider the problem of finding a common divisor of two positive integers *x* and *y*. The problem is easily specified:

- **Input**: Two positive integers *x* and *y*.
- **Output**: A positive integer *z* such that $x \% z = 0$ and $y \% z = 0$.

For instance, for input numbers *x*=36 and *y*=24, a desired output is 3. Indeed, the positive integer 3 is a common divisor, since 24%3=0 and 36%3=0.

One may devise many sequences of computational steps to solve this problem. However, a sequence of computational steps is not necessarily an algorithm. An algorithm must satisfy Knuth's five properties. Let us contrast two specifications.

The first specification (CD1) randomly picks a positive integer *z* and checks to see if it is a common devisor of *x* and *y*.

CD1: Randomly pick and check.
- **Input**: Two positive integers *x* and *y*.
- **Output**: A positive integer *z* such that $x \% z = 0$ and $y \% z = 0$.
- **Steps**:

```
while true
  randomly pick a positive integer $z$
  if ($x \% z == 0$) and ($y \% z == 0$) then halt
```

However, CD1 is not an algorithm because it violates some of the five properties.

- It may never stop, violating the finiteness property.
- The step "randomly picking a positive integer" is not sufficiently rigorous or unambiguous. Out of the set of infinitely many positive integers, what is the meaning of "randomly picking"? It violates the definiteness property.

The second specification (CD2) is a revised version of **Euclid's algorithm**. The idea is to repetitively replace the larger of *x* and *y* by *y* and $x \% y$, till $y = 0$.

CD2: Euclid's algorithm.
- **Input**: Two positive integers *x* and *y* such $x > y$.
- **Output**: A positive integer *z* such that $x \% z = 0$ and $y \% z = 0$.
- **Steps**:

```
while $y \neq 0$
  $x, y = y, x \% y$
$z = x$
```

CD2 is indeed an algorithm. In fact, it does more than finding a common divisor, but finding the **greatest common divisor** of *x* and *y*, i.e., gcd(*x*, *y*). We leave it as exercises to show that CD2 indeed satisfies Knuth's five properties, and the algorithm finds gcd(*x*, *y*)=12, given two inputs *x*=36 and *y*=24.

☷

4.1.2. The Sorting Problem and the Bubble Sort Algorithm

The sorting problem is a classic problem in computer science. The purpose of sorting is to adjust a sequence of "out-of-order" numbers into an ordered sequence of numbers. For simplicity, we assume that all positive integers are stored in an array, these integers have different values, and we need to sort these positive integers from small to large. More formally, the sorting problem is:

The sorting problem

- **Input**: a sequence $< a_1, a_2, \dots, a_n >$ of n positive integers.
- **Output**: a reordered sequence $< a'_1, a'_2, \dots, a'_n >$ such that $a'_1 \le a'_2 \le \cdots \le a'_n$.

People have developed various algorithms to solve the sorting problem, such as bubble sort, insertion sort, quicksort, merge sort, heap sort, etc. They vary in simplicity, efficiency, and suitability to different application scenarios. They also provide rich examples for the design of algorithms. In this section, we discuss the bubble sort algorithm as an example to appreciate how an algorithm works. In Section 4.2, we will discuss insertion sort and merge sort to show the power of the divide-and-conquer strategy. In Section 4.3, we will further introduce the quicksort algorithm which is more sophisticated.

Example 31. The bubble sort algorithm

The name "bubble sort" comes from the fact that large numbers will gradually bubble up to the top of the sequence through comparison and exchange operations, just like bubbles rising from the bottom in a water tank. The algorithm follows.

```
Input: An array A of length n to be sorted, e.g., A=[6, 2, 4, 1, 5, 9].
Output: A sorted array A, e.g., A=[1, 2, 4, 5, 6, 9].
Steps:
  for i = 1 to n-1                  // for each round
      for j = 1 to n-i              // compare every adjacent pair
          if A [j]> A [j + 1] then exchange A [j] with A [j + 1];
```

Fig. 4.1 The bubble sort algorithm

The idea of bubble sort is very simple. In each round, compare every adjacent pair of numbers from left to right, and exchange the two numbers of a pair if the larger one is on the left side of the smaller one. After one round, the largest number will be moved to the rightmost position. We then go to the next round and compare-and-exchange every pair of numbers from left to right again.

For input A=[6, 2, 4, 1, 5, 9], the algorithm's sequence of execution steps is shown in Table 4.1. The output is A=[1, 2, 4, 5, 6, 9].

Table 4.1 Bubble sort [6, 2, 4, 1, 5, 9] into [1, 2, 4, 5, 6, 9]

Outer loop	Inner loop	State before	State after
First round	1st comparison	6, 2, 4, 1, 5, 9	2, 6, 4, 1, 5, 9
	6>2, exchange		
	2nd comparison	2, 6, 4, 1, 5, 9	2, 4, 6, 1, 5, 9
	6>4, exchange		
	3rd comparison	2, 4, 6, 1, 5, 9	2, 4, 1, 6, 5, 9
	6>1, exchange		
	4th comparison	2, 4, 1, 6, 5, 9	2, 4, 1, 5, 6, 9
	6>5, exchange		
	5th comparison	2, 4, 1, 5, 6, 9	2, 4, 1, 5, 6, 9
	6<9, no exchange		
Second round	1st comparison	2, 4, 1, 5, 6, 9	2, 4, 1, 5, 6, 9
	2<4, no exchange		
	2nd comparison	2, 4, 1, 5, 6, 9	2, 1, 4, 5, 6, 9
	4>1, exchange		
	3rd comparison	2, 1, 4, 5, 6, 9	2, 1, 4, 5, 6, 9
	4<5, no exchange		
	4th comparison	2, 1, 4, 5, 6, 9	2, 1, 4, 5, 6, 9
	5<6, no exchange		
Third round	1st comparison	2, 1, 4, 5, 6, 9	1, 2, 4, 5, 6, 9
	2>1, exchange		
	2nd comparison	1, 2, 4, 5, 6, 9	1, 2, 4, 5, 6, 9
	2<4, no exchange		
	3rd comparison	1, 2, 4, 5, 6, 9	1, 2, 4, 5, 6, 9
	4<5, no exchange		
Fourth round	1st comparison	1, 2, 4, 5, 6, 9	1, 2, 4, 5, 6, 9
	1<2, no exchange		
	2nd comparison	1, 2, 4, 5, 6, 9	1, 2, 4, 5, 6, 9
	2<4, no exchange		
Fifth round	1st comparison	1, 2, 4, 5, 6, 9	1, 2, 4, 5, 6, 9
	1<2, no exchange		

Let us take another look at the bubble sort algorithm from the viewpoint of Knuth's characterization. It is indeed an algorithm according to Knuth's definition. The description of the algorithm defines a finite set of rules for specifying the sequence of computational steps to solve the sorting problem.

This specification satisfies the five properties in Knuth's definition of algorithms.

- *Finiteness*. In the bubble sort algorithm, the outer loop needs to be executed $n-1$ times; for the i-th round, the inner loop contains $(n-i)$ comparisons and at most $(n-i)$ exchange. Therefore, the algorithm must terminate within $\sum_{i=1}^{n-1}(n-i) = \frac{n(n-1)}{2}$ steps.
- *Definiteness*. The meaning of each step in the bubble sort algorithm is very clear.
- *Input*. There are two inputs. One is the array A to be sorted, and the other is the length *n* of the array.
- *Output*. The output is the sorted array A, which share space with the input.
- *Effectiveness*. The basic operations of bubble sort are comparison and exchange. Both operations are sufficiently rudimentary. People can use pen and paper to achieve these operations accurately.

The bubble sort algorithm is inefficient, requiring roughly $n^2/\,2$ comparison operations. However, the algorithm has the obvious advantage of simplicity. It consists of a straightforward double loop and an easy-to-understand loop body. In addition, it has the *robustness* advantage: in the case when a small number of errors of comparison operations may occur, the resulting output will still be a mostly sorted sequence, since the algorithm does comparisons only on adjacent numbers.

☷

4.1.3. Asymptotic Notations

It is always important to know if an algorithm is efficient. Given a problem or an algorithm, how much resource (such as execution time or storage space) is theoretically required? For example, in the bubble sort algorithm in Example 31, for n positive integers, the algorithm requires $n(n-1)/\,2$ comparisons and at most $n(n-1)/\,2$ exchange steps. Usually, we do not need to know the exact number or quantity of resource required. We can say the bubble sort algorithm requires roughly n^2 steps, or more professionally, we say the time complexity of bubble sort algorithm is $O(n^2)$. Here, $O(n^2)$ is the **asymptotic notation** of "exactly $n(n-1)/\,2$ comparisons and at most $n(n-1)/\,2$ exchange steps".

We usually use the asymptotic notations such as $O(\cdot), o(\cdot), \Omega(\cdot)$ to describe the efficiency of the algorithms or problems. The following is the formal definition of asymptotic notations.

Definition: let $f, g\colon \mathbb{N} \rightarrow \mathbb{N}$ be two functions, where $\mathbb{N}$ is the set of natural numbers.

- $f(n) = O\big(g(n)\big)$ if $\exists$ constant $c > 0,\ \forall\, n \in \mathbb{N}, f(n) \le cg(n)$.
- $f(n) = o\big(g(n)\big)$ if $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = 0$.
- $f(n) = \Omega\big(g(n)\big)$ if $\exists$ constant $c > 0, \forall n \in \mathbb{N}, f(n) \ge cg(n)$.
- $f(n) = \Theta\big(g(n)\big)$ if $f(n) = O\big(g(n)\big)$ and $f(n) = \Omega\big(g(n)\big)$.

Intuitively, these notations have the following *asymptotic* meanings:

- The big-O notation denotes that g is an *upper bound* of *f*;
- The small-o notation denotes that g is a *strict upper bound* of *f*;
- The Ω notation denotes that g is a lower bound of *f*; and
- The Θ notation denotes that g is the same order of *f*.

It is a good learning practice to compare these notations in one place with some concrete values, to see their differences. For instance, given $f(n) = n^{1.58}$ and $g(n) = n^2$, we have the equalities and inequalities shown in Table 4.2.

Table 4.2 Equalities and inequalities regarding o, O, Ω, Θ notations

Notation	Equalities and Inequalities		
o	$n^{1.58} = \mathrm{o}(n^2)$	$n^{1.58} \neq \mathrm{o}(n^{1.58})$	$n^2 \neq \mathrm{o}(n^{1.58})$
O	$n^{1.58} = \mathrm{O}(n^2)$	$n^{1.58} = \mathrm{O}(n^{1.58})$	$n^2 \neq \mathrm{O}(n^{1.58})$
Ω	$n^{1.58} \neq \Omega(n^2)$	$n^{1.58} = \Omega(n^{1.58})$	$n^2 = \Omega(n^{1.58})$
Θ	$n^{1.58} \neq \Theta(n^2)$	$n^{1.58} = \Theta(n^{1.58})$	$n^2 \neq \Theta(n^{1.58})$

The above table reveals something interesting regarding equality when expressing asymptotic values: the commutativity law and the transitivity law of ordinary math do not hold anymore. It is correct to write $n^{1.58} = \mathrm{O}(n^2)$, but incorrect to write $\mathrm{O}(n^2) = n^{1.58}$ or $n^2 = \mathrm{O}(n^{1.58})$. In fact, the following equalities hold. However, $\mathrm{O}(n^4) \neq \mathrm{O}(n^{1.6})$.

$$n^{1.58} = \mathrm{O}(n^4),$$
$$n^{1.58} = \mathrm{O}(n^3),$$
$$n^{1.58} = \mathrm{O}(n^2 + n - 3),$$
$$n^{1.58} = \mathrm{O}(n^{1.6}).$$

The meaning of the equal sign (=) has changed with asymptotic notations. It becomes *single-direction equality*. The equation $n^{1.58} = \mathrm{O}(n^2)$ means that the right side value n^2 is an upper bound of the left side value $n^{1.58}$.

The introduction of the asymptotic notations helps us focus on the dominant term when n becomes large. Though $n^2 = O(2n^2 + 3n - 4)$ is correct according to the definition of big-O notation, it is strange to use the notations in this way. Usually, we will say $2n^2 + 3n - 4 = O(n^2)$ where $O(n^2)$ represents the main term of function $2n^2 + 3n - 4$ and it helps us focus on how fast the function grows with the input size n. Let us analyze the bubble sort algorithm as an example.

Example 32. Time complexity of the bubble sort algorithm

To sort n positive integers, we know the bubble sort algorithm requires exactly $n(n-1)/\,2$ comparisons but we don't know how many exchange steps we need. The number of exchange steps depends on the input sequence. For example, if the input sequence is $< 1,2,\ldots,n >$, no exchange steps are needed. If the input sequence is $< n, n-1,\ldots,1 >$, the algorithm performs $n(n-1)/\ 2$ exchange steps.

Furthermore, the running time of each comparison or exchange step depends on the physical device which executes this algorithm. So, the exactly running time depends on many factors and is difficult to estimated. However, we can always assume the running time of one step (either comparison or exchange) is bounded by some constant which is independent of n for any physical device.

By using the asymptotic notations, we can safely say that the running time of the bubble sort algorithm is $O(n^2)$. On the other hand, it is also $\Omega(n^2)$ since we need $n(n-1)/\ 2$ comparison steps. Thus, the time complexity of the bubble sort algorithm is $\Theta(n^2)$.

The asymptotic notations help us ignore some details of the running process of the algorithm and focus on the dominant term in the running time. It shows that, when the input size grows, the running time of bubble sort algorithm grows quadratically, but neither linearly nor exponentially.

☷

4.2. Divide-and-Conquer Algorithms

Divide-and-conquer is an algorithm design paradigm based on the idea that we recursively break down a problem into two or more subproblems of the similar type, until these subproblems become simple enough to be solved directly. In this section, we will use several examples to illustrate the idea of divide and conquer method. We firstly focus on how different division methods affect the performance of the algorithm. Section 4.2.1 and 4.2.2 consider the sort problem again. They provide two different ways to split the original problem into subproblems, and show different performance correspondingly. In Section 4.2.3, we show an interesting example which illustrates that equal division is not always the best idea. Then, in Section 4.2.4 and 4.2.5, we will learn how to efficiently combine the results of subproblems so as to obtain the result of the original problem. These two examples illustrate that the construction of the subproblems and the combination process should closely bound together. Finally, in section 4.2.6, we summarize the key points in the divide-and-conquer methodology.

4.2.1. The Insertion Sort Algorithm

In the sorting problem, we want to order the sequence with n integers from small to large. One natural idea is to firstly sort the first $n-1$ integers and then insert the n-th integer into the proper position of a sorted sequence. How can we sort the first $n-1$ integers? Well, this is a subproblem with smaller size. This leads to the idea of the insertion sort algorithm.

```
Input: An array A of length $n$ to be sorted, e.g., A=[6, 2, 4, 1, 5, 9]
Output: A sorted array A, e.g., A=[1, 2, 4, 5, 6, 9].
Steps:
    for i = 1 to n-1
            j=i+1;
            while (j>1) and (A[j-1]>A[j])
                   exchange A[j] with A[j-1];
                   j=j-1;
```

Fig. 4.2 The insertion sort algorithm

In each round, the first i integers of the sequence are already in order. Our task is to insert the $(i + 1)$-th integer into the proper position. Table 4.3 shows the detailed process for the example input [6, 2, 4, 1, 5, 9].

Table 4.3 The sequence of steps for insertion sorting [6, 2, 4, 1, 5, 9] into [1, 2, 4, 5, 6, 9]

Outer loop	Inner loop	State before	State after
First round	1st comparison 6>2, exchange	6, 2	2, 6
Second round	1st comparison 6>4, exchange	2, 6, 4	2, 4, 6
	2nd comparison 2<4, no exchange	2, 4, 6	2, 4, 6
Third round	1st comparison 6>1, exchange	2, 4, 6, 1	2, 4, 1, 6
	2nd comparison 4>1, exchange	2, 4, 1, 6	2, 1, 4, 6
	3rd comparison 2>1, exchange	2, 1, 4, 6	1, 2, 4, 6
Fourth round	1st comparison 6>5, exchange	1, 2, 4, 6, 5	1, 2, 4, 5, 6
	2nd comparison 4<5, no exchange	1, 2, 4, 5, 6	1, 2, 4, 5, 6
Fifth round	1st comparison 6<9, no exchange	1, 2, 4, 5, 6, 9	1, 2, 4, 5, 6, 9

The insertion sort algorithm is not a typical example of divide-and-conquer method. But it illustrates the idea of subproblem. For the sequence of n unsorted integers, we divide it into two subproblems: one with the first $n-1$ integers, and the other with the last integer. Suppose we can solve two subproblems while the second one is trivial; we only need to insert the last integer into the sorted sequence.

We can revise the insertion sort algorithm in the following way to emphasize the idea of subproblems.

```
InsertionSort(n)   //sort sequence A[1] to A[n]
   if (n==1) then return;
   InsertionSort(n-1);        //solve the sub-problem with A[1] to A[n-1]
   j=n;
   while (j>1) and (A[j-1]>A[j])
         exchange A[j] with A[j-1];
         j=j-1;
```

Fig. 4.3 The insertion sort algorithm (revision)

Now let us consider the time complexity of insertion sort algorithm. Let $T(n)$ denote the time complexity of the insertion sort algorithm for n unsorted integers. We have

$$\begin{aligned} T(1) &= 0; \\ T(n) &= T(n-1) + \text{time for insertion} \\ &\le T(n-1) + cn \end{aligned}$$

for some constant c. Thus, we have $T(n) = O(n^2)$.

4.2.2. The Merge Sort Algorithm

In the insertion sort algorithm just discussed above, we divide the original problem into two unequal subproblems, where one subproblem contains $n-1$ integers and the other subproblem contains only one integer.

Fig. 4.4 shows another sorting algorithm called the merge sort algorithm. Here, we divide the original problem into two subproblems, which deal with almost equal number of integers. That is, the sorting problem of MergeSort([A[1],…,A[n]]) is first divided into two subproblems: MergeSort([A[1],…,A[n/2]]) and MergeSort([A[n/2+1],…,A[n]]). Then we merge the results B and C of the two subproblems, which are each a sorted sequence of integers.

```
MergeSort([A[1],…,A[n]])  //sort sequence A[1] to A[n]
  if (n==1) then return [A[1]];
  B=MergeSort([A[1],…,A[n/2]]);
  C=MergeSort([A[n/2+1],…,A[n]]);
  return merge(B, C);
```

Fig. 4.4 The merge sort algorithm

How can we merge two integer sequences B and C? If B and C are two arbitrary sequences of integers, the merging problem is as difficult as the original sorting problem. But remember that, we already know an important fact: B and C are two sorted sequences of integers.

Fig 4.5 shows one of the ways to merge two sorted sequences. Table 4.4 shows the example process for merging two sorted sequences B=[2, 4, 6] and C=[1, 5, 9].

```
merge(B, C)  // merge two sorted sequences
  while (B is not empty) and (C is not empty)
        b = first integer in B;
        c = first integer in C;
        if (b<c) then
                append A with b;
                delete b from B;
        else
                append A with c;
                delete c from C;
  while (B is not empty)
        b = first integer in B;
        append A with b;
        delete b from B;
  while (C is not empty)
        c = first integer in C;
        append A with c;
        delete c from C;
  return A;
```

Fig. 4.5 The merge function in the merge sort algorithm

Table 4.4 The sequence of steps for the merge function

Comparison	State before	State after
1st comparison	A:	A: 1
2>1, delete 1 from C	B: 2, 4, 6	B: 2, 4, 6
	C: 1, 5, 9	C: 5, 9
2nd comparison	A: 1	A: 1, 2
2<5, delete 2 from B	B: 2, 4, 6	B: 4, 6
	C: 5, 9	C: 5, 9
3rd comparison	A: 1, 2	A: 1, 2, 4
4<5, delete 4 from B	B: 4, 6	B: 6
	C: 5, 9	C: 5, 9
4th comparison	A: 1, 2, 4	A: 1, 2, 4, 5
6>5, delete 5 from C	B: 6	B: 6
	C: 5, 9	C: 9
5th comparison	A: 1, 2, 4, 5	A: 1, 2, 4, 5, 6
6<9, delete 6 from B	B: 6	B:
	C: 9	C: 9
no comparison	A: 1, 2, 4, 5, 6	A: 1, 2, 4, 5, 6, 9
delete 9 from C	B:	B:
	C: 9	C:

We are now ready to discuss the main part of the merge sort algorithm in details. Note that when we write down B = MergeSort([A[1],…,A[n/2]]), we recursively call the merge sort algorithm for a smaller subproblem with input sequence A[1],…,A[n/2]. In this subproblem, we will again divide it into 2 sub-subproblems: A[1],…,A[n/4] and A[n/4+1],…,A[n/2], and recursively solve the sub-subproblems with the merge sort algorithm. The recursion will end if the size of unsorted sequence is 1 which is the trivial case. Figure 4.6 illustrated an example for the input sequence [6, 2, 4, 5, 1, 7, 9].

Finally, let us consider the time complexity of the merge sort algorithm. Let $T(n)$ denote the time complexity for n unsorted integers. Similar to the insertion sort algorithm, we have

$$\begin{aligned} T(1) &= 0; \\ T(n) &= T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \text{time for merge function} \\ &\leq 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + cn \end{aligned}$$

for some constant c. Thus, we have $T(n) = O(n \log n)$ which is more efficient than the insertion sort algorithm.

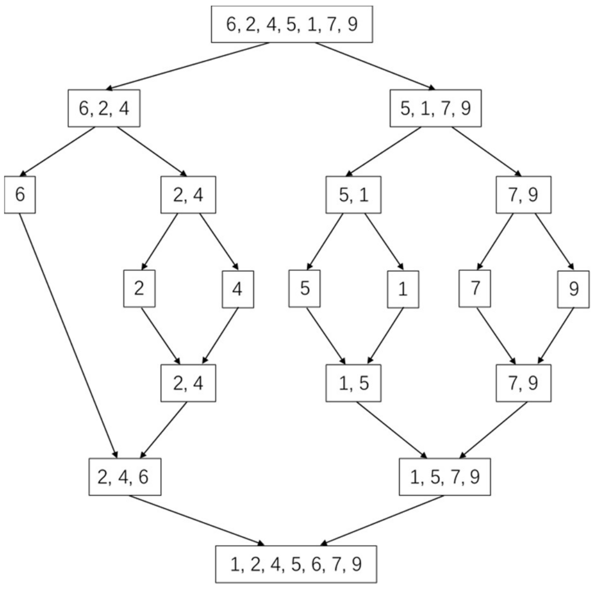


Fig. 4.6 The process of merge sort algorithm with input [6, 2, 4, 5, 1, 7, 9]

The framework of both the insertion sort algorithm and the merge sort algorithm is the same. The cost of "insertion of the last element" in the insertion sort algorithm and the cost of "merge function" in the merge sort algorithm are also roughly the same, where each requires at most $n - 1$ comparisons. The main difference is the sizes of the two subproblems. In the insertion sort algorithm, the sizes are 1 vs. $n - 1$, while in the merge sort algorithm, the sizes are $n/2$ vs. $n/2$. This reduces the time complexity from $O(n^2)$ to $O(n \log n)$ for the sorting problem. It illustrates that when we want to use the divide-and-conquer method to solve a problem, it is important to smartly divide the original problem into subproblems. Often, it is smart to divide a problem into two subproblems of almost equal sizes.

4.2.3. Single Factor Optimization

In this section, we will discuss an interesting problem called single factor optimization. One illusion of the people who just learn the divide-and-conquer method is that they may blindly believe the power of equal division, like the one we did in the merge sort algorithm. This section illustrates that it is not always the case.

The single factor optimization problem considers a univariate function f defined in the interval [a, b]. Assume that f satisfies the following single-peak condition: f firstly (strictly) monotonically increases and then (strictly) monotonically decreases. How can we quickly find the point *x* that maximize $f(x)$?

Well, if function f has good properties, we might compute the maximum directly. For example, if we know the explicit representation of the function f, we can calculate the zero point of its derivative. But in this section, we assume f is implicitly accessed by an oracle such that the only allowed operation is that given x, the oracle will return the value $f(x)$. Our goal is to minimize the number of oracle queries.

Generally, in order to facilitate computer processing, we need to transfer the problem into a discrete version. Suppose we discretize the interval $[a, b]$ into n points, and the function f is represented by an array A: A[1], A[2],…, A[n]. The choice of n depends on the precision we want to achieve. In this way, the problem can be described as the following searching problem:

The single factor optimization problem

- **Input**: array A[1], A[2], …, A[n] such that $\exists i, 1 \leq i \leq n$, A[1]<A[2]<···<A[i], and A[i]>A[i+1]>···>A[n].
- **Output**: i and A[i].

The simplest way to solve this problem is to query the array A one by one. The worst case needs n queries. This method can be used to find the maximum value of any array and obliviously does not take full advantage of the "single-peak" property.

A natural idea is to search from the middle. We select to query $A[\frac{n}{2}]$ and $A[\frac{n}{2}+1]$. There are several cases:

1) If $A\left[\frac{n}{2}\right] > A[\frac{n}{2}+1]$, then according to the property of the function, we know that the maximum value of the function is in the interval $[1, \frac{n}{2}]$, so we can discard the interval $[\frac{n}{2}+1, n]$;
2) If $A\left[\frac{n}{2}\right] < A[\frac{n}{2}+1]$, similar to 1), we can determine that the maximum value of the function is in the interval $[\frac{n}{2}+1, n]$, so we can discard the interval $[1, \frac{n}{2}]$;
3) The case $A\left[\frac{n}{2}\right] = A[\frac{n}{2}+1]$ is impossible.

In either case, we have reduced the search interval from $[1, n]$ by half. In the new search interval, which could be $[1, \frac{n}{2}]$, or $[\frac{n}{2}+1, n]$, the function f still satisfies the property of single-peak condition. We can recursively call this algorithm to continue searching for the maximum point of the function.

The algorithm we described above can be formalized into the algorithm shown as follows.

```
Algorithm 1: find the maximum based on equal division
    Input: A[1], A[2],…, A[n] which satisfies single-peak property
    Output: the maximum value in the array A[i], i

    begin=1; end=n;
    While (end-begin>1) do
        mid = (end-begin)/2;
        If (A[mid]<A[mid+1]) then
            begin=mid+1;
        Else
            end=mid;
    If (A[begin]<A[end]) then
        Return A[end], end
    Else
        Return A[begin], begin
```

Fig. 4.7 The single-factor optimization (binary search)

Let's take a look at the efficiency of this algorithm. We use $T(n)$ to denote the number of queries required by an algorithm on an input of length n. In the first step of the algorithm, we need to query twice: the function values $f(\lfloor\frac{n}{2}\rfloor), f(\left\lfloor\frac{n}{2}\right\rfloor + 1)$. In the second step of the algorithm, we reduce the problem with the original input size n to an input size of $\lfloor\frac{n}{2}\rfloor$ or $(n - \lfloor\frac{n}{2}\rfloor)$ subproblem. The setting of this subproblem is exactly the same as the original problem, except that the scale is smaller than the original problem. Therefore, if the algorithm is called recursively, the required number of queries is $T(\left\lfloor\frac{n}{2}\right\rfloor)$ or $T(n - \left\lfloor\frac{n}{2}\right\rfloor)$. Combining these two steps, we can get:

$$T\ (n) \leq \max\left\{T\ \left(\left\lfloor\frac{n}{2}\right\rfloor\right), T\ \left(n - \left\lfloor\frac{n}{2}\right\rfloor\right)\right\}\ + 2 \leq T\left(\left\lfloor\frac{n+1}{2}\right\rfloor\right) + 2$$

In addition, we know that the initial value T (1) = 1. Thus, we have

$$T(n) \leq 2\lceil\log n\rceil + 1$$

In the above algorithm, we make two queries each time and reduce the length of the interval by half. Is it possible to further improve this algorithm? It seems to be the most economical to shrink the interval by half each time, because if we divide the interval into two parts, the length of one part will always be at least the half. Can we reduce the number of queries in each round? This is possible, if we can reuse the query results which we have obtained before.

Below we give another more efficient algorithm for the above problem. The idea of Algorithm 2 is basically the same as that of Algorithm 1. The key difference lies in the selection of cut points. In our algorithm and analysis, the constant $(\sqrt{5} - 1)/2 \approx 0.618$ which is actually the golden section ratio is frequently used. For the convenience of writing, we set $\alpha = (\sqrt{5} - 1)\ /\ 2$. The block diagram of Algorithm 2 is shown as follows:

```
Algorithm 2: find the maximum based on the golden section method
Input: $A[1], A[2], \cdots, A[n]$
Output: the maximum value in the array
Steps:
 $begin \leftarrow 1$, $end \leftarrow n$
 While $end - begin > 1$ do
         $x_1 \leftarrow \lfloor \alpha \cdot begin + (1-\alpha) \cdot end \rfloor, x_2 \leftarrow \lfloor (1-\alpha) \cdot begin + \alpha \cdot end \rfloor$
         If $A[x_1] \leq A[x_2]$ then
             $begin \leftarrow x_1$
         Else
             $end \leftarrow x_2$
 If $A[begin] < A[end]$ then
         Return $A[end], end$
 Else
         Return $A[begin], begin$
```

Fig. 4.8 The single-factor optimization (golden section method)

Now, let us analyze the performance of Algorithm 2. In order to simplify the analysis, we ignore all rounding symbols. By querying $x_1 = (1-\alpha)\,n$ and $x_2 = \alpha n$, we reduce the problem to a subproblem $A[1, \ldots, \alpha n]$ or $A[(1-\alpha)\,n, \ldots, n]$. In either case, the scale of the new subproblem is αn. It seems that it is worse than binary search. However, notice that for the new subproblem, one of the two points we need to query is already known! Taking $A[1, \ldots, \alpha n]$ as an example, according to the steps of the algorithm, the two points we need to query are $y_1 = \alpha(1-\alpha)n$ and $y_2 = \alpha^2 n$. Note that α is the golden section ratio which is the solution of the equation $x^2 + x - 1 = 0$. After simple calculation, we have $y_2 = x_1$, which means that we do not need to query the value of point y_2 because we already know the value of point x_1. Similarly, for the subproblem $A[(1-\alpha)\,n, \ldots, n]$, we can know that the first branch point required by the algorithm is exactly x_2. In either case, we only need to query ONE new point, so we get the following recursion:

$$\begin{cases} \mathrm{T}(n) = \mathrm{T}(\alpha n) + 1, \\ \quad \mathrm{T}(1) = 1. \end{cases}$$

By solving this recursion, we have

$$\mathrm{T}(n) = \log_{(1+\alpha)} n + 1$$

Comparing the two algorithms, we can see that the performance of Algorithm 2 is better than that of Algorithm 1 (because $2\log_2 n = \log_{\sqrt{2}} n > \log_{(1+\alpha)} n$). To solve the same problem, when we use different methods to design our algorithm, its performance is different. We always hope to be able to design the best algorithm, that is, the most efficient algorithm to solve the problem.

In single-factor optimization, our intuition is to reduce the query number in each round. We come up with the brilliant idea that we can reuse the query in the previous round. In order to use such an idea, we modify the division method by using golden section. This tells us: the division method and the combination method are interdependent, and no division method is universal.

There is a final remark of the single-factor optimization problem. In all of our discussion, our objection is to minimize the number of oracle queries. Since in most real scenarios, one oracle query is much more expensive than the comparison operations or assignment operations in the algorithm, it is natural to ignore the cost of the other operations. But, if the running time of one oracle query is the same as that of one comparison operation or one assignment operation, is algorithm 2 still better than algorithm 1?

4.2.4. Integer Multiplication

In this section, we will discuss the integer multiplication problem. We will show if we apply divide-and-conquer method mechanically, we will not enhance performance. Thus, we need to think about a cleverer way to solve the problem.

Firstly, let us describe the integer multiplication problem:

The integer multiplication problem

- Input: $X = x_n x_{n-1} \dots x_1$, $Y = y_n y_{n-1} \dots y_1$.
- Output: $Z = X \times Y = XY$

Calculating the multiplication of two numbers is a problem we often encounter in our daily life. We give an example to show how to calculate the multiplication of two 3-digit numbers 123 × 321 by hand.

$$\begin{array}{r} 123 \\ \times 321 \\ \hline 123 \\ 246 \\ 369 \\ \hline 39483 \end{array}$$

For two n-digit numbers (imagine that n is very large, for example, $n = 10^{12}$), if we use a similar method to calculate the multiplication, we need n^2 multiplications and about n^2 additions of 1-digit operation. Let us see if we can reduce the total number of calculations by adopting the divide-and-conquer approach.

Write the input X and Y as follows:

$$X = X_1 \times 10^{n/2} + X_2, Y = Y_1 \times 10^{n/2} + Y_2,$$

where the length of X_1, X_2, Y_1, Y_2 is $n/2$. What we need to calculate is:

$$Z = XY = X_1Y_1 \times 10^n + (X_1Y_2 + X_2Y_1) \times 10^{n/2} + X_2Y_2.$$

The naïve idea is to call the algorithm recursively to calculate $X_1Y_1, X_1Y_2, X_2Y_1, X_2Y_2$. Based on this idea, we need to multiply two (n/2)-digits numbers 4 times in total, and also need up to 3 times n-digits addition, so we get

$$\begin{cases} \mathrm{T(n)} = 4\mathrm{T}(n/2) + 3n, \\ \qquad \mathrm{T}(1) = 1. \end{cases}$$

It is easy to solve the recursion and obtain $\mathrm{T}(n) = \mathrm{O}(n^2)$, where there is no substantial improvement over the previous natural algorithm.

Now let us change the way of thinking. The idea is: what we need is $X_1Y_2 + X_2Y_1$ instead of X_1Y_2 and X_2Y_1.

Notice that $X_1Y_1 + X_1Y_2 + X_2Y_1 + X_2Y_2 = (X_1 + X_2)(Y_1 + Y_2)$. So, by calculating $X_1Y_1, X_2Y_2, (X_1 + X_2)(Y_1 + Y_2)$, and then use

$$X_1Y_2 + X_2Y_1 = (X_1 + X_2)(Y_1 + Y_2) - X_1Y_1 - X_2Y_2,$$

we can obtain $X_1Y_2 + X_2Y_1$. By this method, we need to multiply two (n/2)-digits numbers 3 times in total, and also need n-digits addition 6 times: $X_1 + X_2, Y_1 + Y_2, (X_1 + X_2)(Y_1 + Y_2) - X_1Y_1 - X_2Y_2$, and

$$Z = X_1Y_1 \times 10^n + (X_1Y_2 + X_2Y_1) \times 10^{n/2} + X_2Y_2.$$

Thus, we have

$$\begin{cases} \mathrm{T}(n) = 3\mathrm{T}(n/2) + 6n, \\ \qquad \mathrm{T}(1) = 1. \end{cases}$$

By solving this recursion, we have $\mathrm{T}(n) = cn^{\log_2 3} + O(n) \sim n^{1.59}$, which is a great improvement compared to the naïve algorithm with time complexity $O(n^2)$. This example tells us when designing divide-and-conquer algorithms, it is important to make the number of subproblems of recursive calls as small as possible.

4.2.5. Matrix Multiplication

Matrix multiplication is a natural extension of integer multiplication. We want to further illustrate the idea on how to minimize the number of subproblems.

The matrix multiplication problem

- **Input**: two $n \times n$ matrices $\mathrm{A} = [a_{i,j}]$, $\mathrm{B} = [b_{i,j}]$.
- Output: $\mathrm{C} = \mathrm{AB}$.

According to the definition of matrix multiplication, we know

$$c_{i,j} = a_{i,1}b_{1,j} + a_{i,2}b_{2,j} + \cdots + a_{i,n}b_{n,j}.$$

If we use the natural method to compute each $c_{i,j}$ directly, we need $\mathrm{O}(n^3)$ multiplications and $\mathrm{O}(n^3)$ additions in total. Let use divide A, B and C into four $(n/2 \times n/2)$ sub-matrices:

$$A = \begin{bmatrix} \mathrm{A}_{1,1} & \mathrm{A}_{1,2} \\ \mathrm{A}_{2,1} & \mathrm{A}_{2,2} \end{bmatrix}, \quad B = \begin{bmatrix} \mathrm{B}_{1,1} & \mathrm{B}_{1,2} \\ \mathrm{B}_{2,1} & \mathrm{B}_{2,2} \end{bmatrix}, \quad C = \begin{bmatrix} \mathrm{C}_{1,1} & \mathrm{C}_{1,2} \\ \mathrm{C}_{2,1} & \mathrm{C}_{2,2} \end{bmatrix}.$$

Then, we have

$$\begin{aligned} \mathrm{C}_{1,1} &= \mathrm{A}_{1,1}\mathrm{B}_{1,1} + \mathrm{A}_{1,2}\mathrm{B}_{2,1} \\ \mathrm{C}_{1,2} &= \mathrm{A}_{1,1}\mathrm{B}_{1,2} + \mathrm{A}_{1,2}\mathrm{B}_{2,2} \\ \mathrm{C}_{2,1} &= \mathrm{A}_{2,1}\mathrm{B}_{1,1} + \mathrm{A}_{2,2}\mathrm{B}_{2,1} \\ \mathrm{C}_{2,2} &= \mathrm{A}_{2,1}\mathrm{B}_{1,2} + \mathrm{A}_{2,2}\mathrm{B}_{2,2} \end{aligned}$$

If we directly call the subroutine to compute $C_{1,1}, C_{1,2}, C_{2,1}, C_{2,2}$, we need 8 calls of the subproblem of the multiplication of $n/2 \times n/2$ submatrices. In addition, we need 4 times addition of $n/2 \times n/2$ matrices. Thus, the recursion is

$$T(n) = 8T\left(\frac{n}{2}\right) + 4\left(\frac{n}{2}\right)^2$$

The final complexity obtained by solving this recursion is still $O(n^3)$. Applying the previous ideas about n-digit multiplication, we need to reduce the number of subroutine calls through appropriate addition and subtraction. How can we achieve this? It is more difficult than the integer multiplication problem. The following solution is proposed by Volker Strassen (1936-).

Example 33. Strassen's algorithm for matrix multiplication

Define the following 7 matrices:

$$\begin{aligned}
M_1 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}),\\
M_2 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}),\\
M_3 &= (A_{1,1} - A_{2,1})(B_{1,1} + B_{1,2}),\\
M_4 &= (A_{1,1} + A_{1,2})B_{2,2},\\
M_5 &= A_{1,1}(B_{1,2} - B_{2,2}),\\
M_6 &= A_{2,2}(B_{2,1} - B_{1,1}),\\
M_7 &= (A_{2,1} + A_{2,2})B_{1,1}.
\end{aligned}$$

We can make an observation: the matrices we need to compute, e.g., $C_{1,1}, C_{1,2}, C_{2,1}, C_{2,2}$, can be calculated by using $M_1, \ldots, M_7$ and some addition or subtraction operations. The detailed method is as follows:

$$\begin{aligned}
C_{1,1} &= M_1 + M_2 - M_4 + M_6,\\
C_{1,2} &= M_4 + M_5,\\
C_{2,1} &= M_6 + M_7,\\
C_{2,2} &= M_2 - M_3 + M_5 - M_7.
\end{aligned}$$

Let's take a look at the performance of Strassen algorithm. First of all, Strassen algorithm needs to call a total of 7 sub-matrix multiplications. In addition, the algorithm also needs to add $n/2 \times n/2$ matrices several times, so we have the following recursion:

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2).$$

Here we do not accurately calculate the total number of times required for the addition or subtraction. Instead, we use the $O(\cdot)$ notation to hide the constant. In fact, this constant does not affect the magnitude of T(n). No matter what the constant is, the solution is $T(n) = O(n^{\log 7}) \approx O(n^{2.81})$.

☷

The algorithm proposed by Strassen in 1969 is the first algorithm about matrix multiplication that can beat the conventional $O(n^3)$ algorithm. Since then, the upper bound of the complexity of matrix multiplication has been continuously improved: the algorithm complexity proposed by Pan in 1978 is $O(n^{2.796})$, by Bini et al. in 1979 is $O(n^{2.78})$, by Schönhage in 1981 is $O(n^{2.548})$, by Romani in 1982 is $O(n^{2.517})$, by Strassen in 1986 is $O(n^{2.479})$. At present, the best matrix multiplication algorithm was proposed by Coppersmith and Winograd in 1987. The complexity of the algorithm was $O\left(n^{2.376}\right)$ when originally proposed. Recently, the analysis of the original algorithm has been continuously improved by Stothers, Williams, Le Gall and others. The algorithm complexity is reduced to $O(n^{2.3729})$. Whether there is a matrix multiplication algorithm close to $O(n^2)$ complexity is an important unsolved problem in the field of algorithms.

4.2.6. Summarization

In this section, we discuss many examples with the help of divide-and-conquer methodology. In general, divide-and-conquer comes from the idea that when you want to solve a complicated problem, try to transfer it into some easier problem. There are two features of the problems which can be solved with the help of divide-and-conquer method. Firstly, the problem with extremely small size is straightforward to solve, for example, the sort problem with only 2 elements. This will be served as the basis of the recursive process. Secondly, we can solve the general problem with the help of the problem with smaller size. This part is the art in the divide-and-conquer method. There is no universal way of construction for every scenario, and we need to observe the specialty for each problem ourselves. However, there are two things which are usually important in the design of divide-and-conquer algorithms.

Firstly, it is usually better to use smaller number of subproblems to solve the original problem. In many examples, such as integer multiplication and matrix multiplication, we are trying to reduce the number of subproblems and we show with smaller number of subproblems, we dramatically improve the performance of the algorithms, even if we slightly increase the time complexity for each round. But please do not go to the other extreme. For example, in the integer multiplication problem, the natural idea gives us $\mathrm{T(n)} = 4\mathrm{T}(n/2) + 3n$. If we modify it into $\mathrm{T(n)} = 3\mathrm{T}(n/2) + 6n$ as illustrated in section 4.2.4, it improves the performance. But if we modify it into $\mathrm{T(n)} = 3\mathrm{T}(n/2) + O(n^2)$, the performance will be $\mathrm{T(n)} = \mathrm{O(n^2)}$. If $\mathrm{T(n)} = 3\mathrm{T}(n/2) + O(n^3)$, the performance will be even worse than the natural algorithm. In other words, in the design of divide-and-conquer algorithms, we need to balance all parts.

Secondly, it is usually better to partition the original problem into subproblems. For example, in the sort algorithm, we partition the whole unsorted set into two disjoint subsets and recursively sort them. It makes no sense if the subproblems have overlapping elements. Partition, in some sense, can make the size of subproblems as small as possible. Thus, it is useful for better performance. But in the example of single factor optimization, we also see that this is not always the case. In such an example, although two possible subproblems $[1, \ldots, \alpha n]$ and $[(1-\alpha)n, n]$

are overlapping, it is faster than the natural halving method. In the integer multiplication and the matrix multiplication problems, it is even hard to distinguish which idea is a partition. When dividing the problem into subproblems, the partition method is usually a good start point since it makes the size of subproblems small, but the size of subproblems is not the only factor in the divide-and-conquer method, and we need to balance all parts. On the other hand, if there are too many overlaps between different subproblems, some other methods may be more powerful than divide-and-conquer. See Section 4.3.1 for an example.

4.3. Other Examples of Interesting Algorithms

In the previous section, we focused on the algorithms based on the divide-and-conquer method. We also learn how to analyze the algorithm complexity through recursion expression. In this section we will see some other examples of algorithms. The first example is using dynamic programming to compute Fibonacci number. We will see how this method can eliminate duplicated computation. The second example is the stable matching problem, which uses a kind of "greedy" algorithm method, different from the divide-and-conquer paradigm. While the correctness of a divide-and-conquer algorithm is usually straightforward, the correctness of a greedy solution needs to be proved. The final example is the quicksort algorithm for the sorting problem. The quicksort algorithm is not a deterministic algorithm, that is, the algorithm will toss some coins to decide the next step during the process of running. We will see how to analyze the complexity of such algorithms.

4.3.1. Dynamic Programming

A divide-and-conquer algorithm solves a problem by dividing the problem into *independent* subproblems, and then combining their solutions. The key character here is that the subproblems are independent, meaning that they do not overlap. That is, the subproblems do not share subproblems and do not solve the shared subproblems multiple times.

What if the subproblems do overlap? An algorithm paradigm called dynamic programming specifically addresses such concerns. A dynamic programming algorithm divides the problem into potentially overlapping subproblems and combines their solutions. The key character is that the solution to every shared subproblem is memorized and reused, avoiding computing its solution multiple times.

There are two approaches to implementing memorization in dynamic programming algorithms, the top-down approach and the bottom-up approach. We analyze the example of computing a small Fibonacci number F(5) to see how the two approaches in dynamic programming work, as shown in Example 34.

Example 34. Eliminate redundant computation by dynamic programing

We analyze the behaviors of two dynamic programming algorithms computing a small Fibonacci number F(5), against the recursive program fib-5.go. The details of the fib-5.go program and the fib.dp-5.go program using the top-down approach are shown in Fig. 4.10. Some diagnostic print statements are added to print out intermediate results, to reveal the behaviors of these programs.

The recursive program fib-5.go does a lot of redundant, unnecessary computations. This becomes immediately clear when we look at Fig. 4.9, which shows the tree of recursive calls to fibonacci(n), denoted as F(5), (F4), F(3), F(2), F(1), and F(0). The circled numbers, ① , ② , …, ⑮ , show the order as to how the calls are made. The program calls fibonacci(n) 15 times. It first calls F(5), then F(4), and finally F(0). Note that F(0) is called 3 times, F(1) 5 times, F(2) 3 times, F(3) 2 times. Altogether, 9 unnecessary computations are performed.

The fib.dp-5.go program uses the top-down approach of dynamic programming to compute F(5). It is similar to the recursive program fib-5.go, but results of F(n) are stored in a 6-element array mem. Every element mem[i] is initialized to -1, to denote that this element has not been computed yet. When the program calls fibonacci(n), the code first checks if mem[n] is -1. If it is not, the function call immediately returns with the already computed value mem[n], without going further to do unnecessary computation.

The diagnostic printout should show that when running the fib.dp-5.go program, fibonacci(n) is called only 9 times. The calling order is F(5), (F4), F(3), F(2), F(1), F(0), F(1), F(2), F(3). Furthermore, the last three calls F(1), F(2), F(3) are returned immediately, without doing unnecessary computation. Thus, the fib.dp-5.go program only performs 6 necessary computations F(5), (F4), F(3), F(2), F(1), F(0).

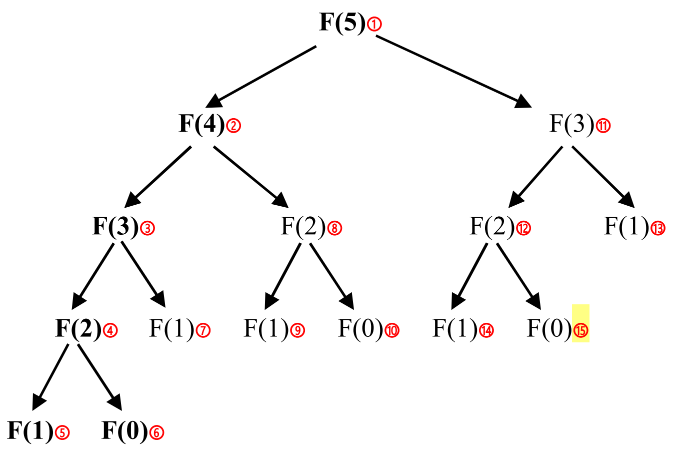


Fig. 4.9 The sequence of fibonacci(n) calls, where ① , ② , ⋯ denote the order.

```
package main
import "fmt"
func main() {
  fmt.Println("F(5)=", fibonacci(5))
}
func fibonacci(n int) int {
  fmt.Println("F(",n,")")
  if n == 0 || n == 1 {
    return n
  }
  return fibonacci(n-1)+fibonacci(n-2)
}
```

(a) Recursive program fib-5.go

```
package main
import "fmt"
var mem [6]int
func main() {
        for i := 0; i < 6; i++ { mem[i] = -1 }
        fmt.Println("F(5)=", fibonacci(5))
}
func fibonacci(n int) int {
        fmt.Println("F(",n,")")
        if mem[n] != -1 {
                fmt.Println("Immediate Return: F(",n,")=",mem[n])
                return mem[n]
        }
        if n == 0 || n == 1 {
                mem[n] = n
                fmt.Println("Return: F(",n,")=",mem[n])
                return mem[n]
        }
        mem[n] = fibonacci(n-1) + fibonacci(n-2)
        fmt.Println("Return: F(",n,")=",mem[n])
        return mem[n]
}
```

(b) Dynamic programming program fib.dp-5.go

Fig. 4.10 Two programs to compute Fibonacci number F(5)

To compute Fibonacci number F(n), it is easy to find that we need to call F(i) for all $i < n$. The bottom-up approach takes advantage of this fact and prepares the small Fibonacci number before a call. It starts at the smallest subproblems F(0) and F(1), memorize their solutions, and combines their solutions into the solution of the subproblem next level up, e.g., F(2)=F(1)+F(0). This iterative process continues, to obtain F(3)=F(2)+F(1), F(4)=F(3)+F(2), until the topmost solution F(5) is obtained. When we compute some Fibonacci number, for example F(3), we do not need to worry whether the smaller numbers F(2) and F(1) have been computed or not. Thus, we do not need -1 to represent unfinished work, as what we did in fib.dp-5.go. The bottom-up program fib.dp.bu.go is shown below. In this code, we do not even store all numbers we compute, since we only need F(n-1) and F(n-2) to computer F(n).

```
package main        // program fib.dp.bu.go
import "fmt"
func main() {
   fmt.Println("F(5)=", fibonacci(5))
}
func fibonacci(n int) int {
   a :=0
   b :=1
   for i :=1; i < n+1; i++ {
      a = a + b
      a, b = b, a
   }
   return a
}
```

The bottom-up approach often results in a simpler, iterative program. However, students may find either the top-down or the bottom-up approach more intuitive and easier to use. For instance, for the problem of finding a shortest path in a graph, many students prefer the top-down approach.

☷

4.3.2. (***) The Greedy Strategy

This example comes from economics. The 2012 Nobel Prize in Economics was awarded to mathematical economists Alvin Roth and Lloyd Shapley in recognition of their outstanding contributions to "the theory of stable distribution and its market design practice". The stable matching problem is one of the starting point of this research area.

Consider the following scenario. Suppose n boys $\mathrm{M}_1, \mathrm{M}_2, \dots, \mathrm{M}_n$ and n girls $\mathrm{W}_1, \mathrm{W}_2, \dots, \mathrm{W}_n$ participate in a dance together, and each of them hopes to find a suitable partner to dance. For each girl W_i, according to her own criteria, there is a ranking for the n boys. The boy in the higher rank indicates that W_i thinks he is more suitable than the boy in the lower rank. Similarly, for each boy M_j, there will also be a ranking for all n girls. Suppose that at the beginning of the dance, they arbitrarily formed n pairs of dance partners and began to dance the first dance. In this process, if there exists a pair of boys M_i and girls W_j, they are not each other 's partners, but each of them feels that the other one is better than their current partner, then when the next song starts, they will choose the other as their partner instead of the current partner. We call such a pair (girls, boys) an unstable pair. If there is an unstable pair in the matching, we call such a matching unstable, otherwise we call such a matching stable. The question now is whether these n girls and these n boys can form n pairs of stable partners together?

This problem is called the **stable matching problem**. Below we give a more rigorous mathematical description of this problem:

We can use two $n \times n$ matrices to represent our input. Matrix W represents the preference matrix for girls. Each row is a permutation of $\{1,2,\dots,n\}$, and the i-th row means that the ranking of boys for the girl W_i. Matrix M represents the preference matrix of boys, and the i-th row represents the ranking of M_i for all girls. An example is given in the table below.

W_1	M_2	M_1	M_3
W_2	M_1	M_2	M_3
W_3	M_2	M_3	M_1

M_1	W_2	W_1	W_3
M_2	W_1	W_2	W_3
M_3	W_3	W_1	W_2

Fig. 4.11 Matrix W and matrix M in an example of the stable matching problem

The stable matching problem:

Is there a matching between boys and girls $\{(W_{i_1}, M_{j_1}), (W_{i_2}, M_{j_2}), \dots, (W_{i_n}, M_{j_n})\}$, where $\{i_1, \dots, i_n\}$ and $\{j_1, \dots, j_n\}$ are two permutations of $\{1,2,\dots,n\}$, satisfying that there is no pair (W_{i_k}, M_{j_ℓ}), where $k \neq \ell$, such that M_{j_ℓ} ranks higher than M_{j_k} in the order of W_{i_k}, and W_{i_k} ranks higher than W_{i_ℓ} in the order of M_{j_ℓ}.

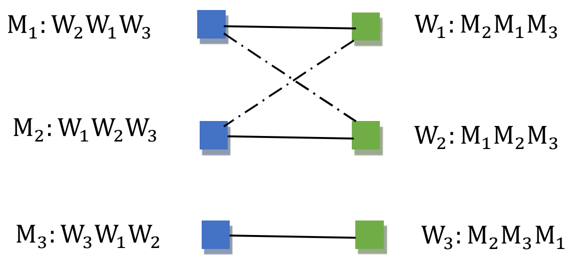


Fig. 4.12 Example of stable matching and unstable matching

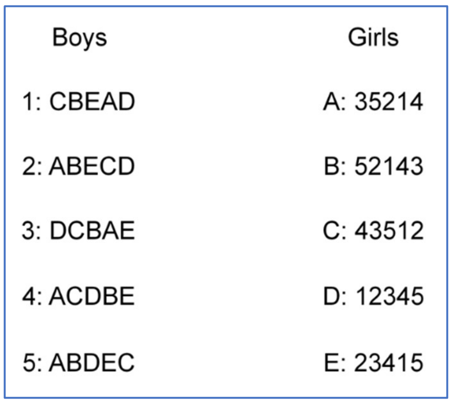
For example, in the example in Fig 4.12, $\{(\mathrm{W}_1, \mathrm{M}_1), (\mathrm{W}_2, \mathrm{M}_2), (\mathrm{W}_3, \mathrm{M}_3)\}$ is an unstable matching. Let us examine girl W_1 and boy M_2. W_1's current partner is M_1, while in the ranking of W_1, M_2 ranks higher than M_1. At the same time, for the boy M_2's current partner W_2, W_1 is also ranked higher than W_2. It can be verified that the matching $\{(\mathrm{W}_1, \mathrm{M}_2), (\mathrm{W}_2, \mathrm{M}_1), (\mathrm{W}_3, \mathrm{M}_3)\}$ is a stable matching. Note that for unstable boy and girl pairs, both parties must feel that the other is better. If only one party feels that the other is better, this does not constitute an unstable pair. For example, consider the pair M_2 and W_3 here. Although according to the order of W_3, M_2 is better than its current partner for M_3, in the view of M_2, his partner W_2 is better than W_3. Thus he will not agree to change the partner.

This problem has important applications in economics. Mathematical economists Gale and Shapley first proposed and studied this problem. They proved that regardless of the preference ranking of each boy and girl, a stable matching always exists. In fact, they have given an algorithm to find such a matching. This algorithm is called the Gale-Shapley algorithm today, described as follows.

The algorithm is divided into several rounds. In the first round, each boy selects the girl who has the highest ranking in his preference order and invites the girl to dance. For any girl W who receives the invitation, choose the best boy among the inviters and become his "temporary" dance partner, and for all other inviters, refuse the invitation. We change the status of W to be "not free". As long as there are "free" girls, the algorithm performs the following steps:

In the new round, for each boy who is unmatched in the previous round, he will select the highest ranked girl who has not been invited by him according to his preference order and send her an invitation, regardless of whether this girl currently has a "temporary" dance partner. On the girl's side, if some girl has "temporary" dance partner, she pretends to receive the invitation from her partner in this round. Then, for each girl who receives at least one invitation in this round, choose the best boy among the inviters and become his "temporary" dance partner, and for all other inviters, refuse the invitation. We change the status of this girl to be "not free". The algorithm re-examines whether there are "free" girls, and start a new round if there are any "free" girls.

We give an example with 5 boys and 5 girls to illustrate how Gale-Shapley algorithm works. In the example shown in Fig. 4.13, boys are represented by numbers, and girls are represented by capital letters. On the left side of the figure, the letter string next to the number indicates the ranking of the boy's preference for all girls. On the right side of the figure, a string of numbers next to the letter indicates the ranking of girl's preferences. The rankings on both sides are arranged from most like to least like. For instance, the entry 1:CBEAD indicates that Boy 1 likes C the most and D the least.

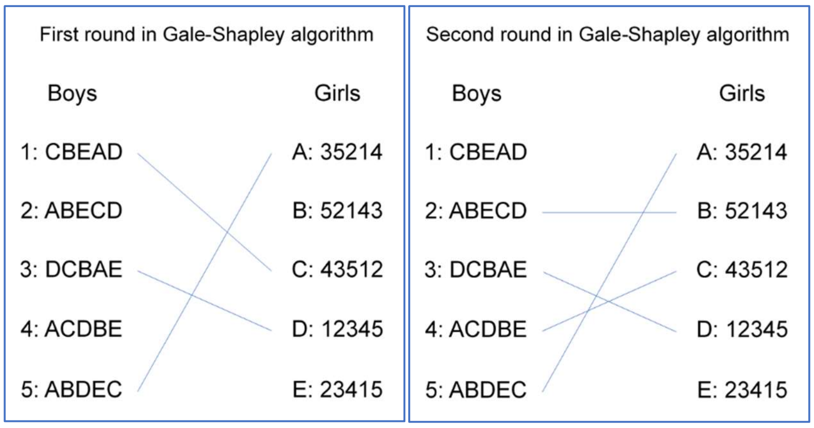


Boys	Girls
1: CBEAD	A: 35214
2: ABECD	B: 52143
3: DCBAE	C: 43512
4: ACDBE	D: 12345
5: ABDEC	E: 23415

Fig. 4.13 Example of the Gale-Shapley algorithm

In the first round, each boy will propose to the girl he likes most. Girl C will receive invitation from boy 1, and she becomes his "temporary" dance partner. Girl A will receive invitation from boy 2,4,5 and she will become the "temporary" dance partner of boy 5 according to her preference. Girl D will receive invitation from boy 3, and she becomes his "temporary" dance partner. In the end of this end, girl B and E are still free while boy 2 and 4 do not have dance partner.

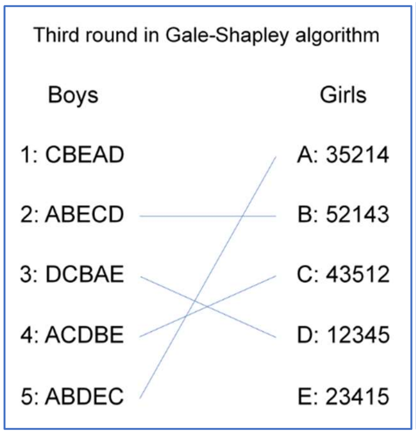
In the second round, boy 2 will propose to girl B and boy 4 will propose to girl C. For girl B, she only receives the invitation from boy 2, thus she becomes his "temporary" dance partner. But for girl C, she is currently the partner of boy 1 and receives a new invitation from boy 4. She will compare these two boys according to her preference, and becomes the partner of boy 4. In the end of this round, boy 1 becomes unmatched.

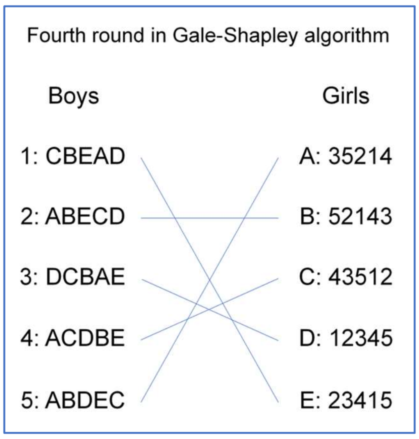


In the third round, boy 1 will propose to girl B. But girl B thinks her current partner boy 2 is better than boy 1, so she will refuse the invitation.

In the fourth round, boy 1 will propose to girl E. Since girl E is free, she will accept the invitation. Now, all girls become "not free", thus, the algorithm terminates.

The reader can verify that {(1, E), (2, B), (3, D), (4, C), (5, A)} is indeed a stable matching.





Now, let us firstly discuss the correctness of Gale-Shapley algorithm. It is not always obvious whether an algorithm correctly solves the problem we require, especially for some complex algorithms. However, it is quite important to strictly prove the correctness of any algorithm, otherwise, there is no guarantee for the output. The previous algorithms based on divide-and-conquer method are relatively simple, and the correctness of the algorithm is self-evident, so we omitted the proof of the correctness. But for the Gale-Shapley algorithm, the correctness is not obvious. It is even not obvious why the algorithm will eventually terminate. From a mathematical point of view, "stable matching must exist" is not a clearly established proposition.

Before proving the correctness of Gale-Shapley algorithm, we first observe some simple properties of this algorithm:

1. Every boy invites a girl at most once;
2. Every girl keeps the status "not free" since she was first invited;
3. Every girl has at most one dance partner during the process of the algorithm;
4. Every boy has at most one dance partner during the process of the algorithm;
5. Every unmatched boy will continue to invite until it matches or he has invited all girls;

The following lemma shows the correctness of the Gale-Shapley algorithm

Lemma 1: Gale-Shapley algorithm stops after $O(n^2)$ rounds, and after it stops, it will output a matching.

Proof: According to property 1, we know that each boy invites n times at most, so the total number of invitations is at most n^2. In each round, there are at least one invitation. Thus, the algorithm will terminate after at most n^2 rounds.

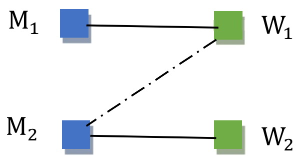
When the algorithm stops, if all girls are "not free", according to property 3 and 4, all girls and boys are matched. Thus it forms a matching.

It seems that we have already finished the proof. However, there are some subtlety in the algorithm. In the algorithm, in each round, we say "for each unmatched boy, he will select the highest ranked girl who has not been invited by him and send the invitation". But, does it possible that for some unmatched boy, he has already invited all girls? We will show this case is impossible.

We show it by contradiction. Suppose for some boy M, in the beginning of some round T, he is unmatched, but he has already invited all girls. According to property 2, after a girl receives her first invitation, her status will always be "not free". Since this boy M has invited all girls, the status of all girls should be "not free" in the beginning of round T. Thus, the algorithm should stop in the previous round. Contradiction. We finish the proof. ■

Lemma 2: The output by the Gale-Shapley algorithm is a stable matching.

Proof: We will still prove it by contradiction. Suppose the final matching output by the Gale-Shapley algorithm is $\{(\mathrm{W}_1, \mathrm{M}_1), (\mathrm{W}_2, \mathrm{M}_2), \dots, (\mathrm{W}_\mathrm{n}, \mathrm{M}_\mathrm{n})\}$. Without loss of generality, let us assume that the unstable pair in the final matching is (W_1, M_2). This pair is unstable means W_1 prefers M_2 to M_1, and M_2 prefers W_1 to W_2 (see figure below).



We consider two cases:

Case 1: M_2 has never invited W_1. Since M_2 and W_2 are finally together, it means that M_2 invites W_2 in some round. On the other hand, since M_2 prefers W_1 to W_2, M_2 must invite W_1 before he invites W_2. Contradiction.

Case 2: M_2 has invited W_1 in some round. Since M_1 and W_1 are finally together and M_2 has invited W_1 in some round, it means W_1 refuses the invitation of M_2 in some round due to she receives invitation from some better boy. For every girl, she will refuse the invitation or change partner only if some better boy sends her invitation, so it means in the girls' view, their partners becomes better and better. Since the final partner of W_1 is M_1, girl W_1 prefers M_1 to M_2. Contradiction.

Therefore, in each case, we will reach contradiction. We finish the proof. ■

Question: in your opinion, is this algorithm beneficial for boys or girls?

4.3.3. The Randomization Strategy

In the previous section, we have already introduced the sorting problem and discussed three sorting algorithms: the bubble sort algorithm, the insertion sort algorithm and the merge sort algorithm. In addition to these sorting algorithms, there are many different sorting algorithms. In this section, we will introduce a sorting algorithm commonly used in our computers: the **quicksort** algorithm. The difference is that we will use randomized process in the algorithm, and we will show the power of randomization.

The core idea of the quicksort algorithm is similar to the merge sort algorithm: call itself recursively to sort the subproblems. In the merge sort algorithm, we firstly solve the subproblems and then try to merge the results into a whole ordered set. But in the quicksort algorithm, we will carefully divide the original problem into subproblems, and after solving the subproblem, the merge process becomes trivial. How can we achieve this? Suppose the original array is A. The key idea is to divide A into two subsets A_1 and A_2 where all elements in A_1 are smaller than all elements in A_2, then the merge process will be trivial.

```
QuickSort(A, $p$, $r$)
  If $p < r$
    1. $q$ = Partition(A, $p$, $r$)
    2. QuickSort(A, $p$, $q$-1)
    3. QuickSort(A, $q$+1, $r$)
```

The above is the pseudocode of the quicksort algorithm, which sorts the p-th element to the r-th element in the array A. It needs to call a Partition subroutine.

The Partition (A, p, r) subroutine uniformly and randomly extracts an element x from the array A[p,…, r], and then adjusts the array A[p,… r] so that the numbers larger than x are arranged on the right side of x, and the numbers smaller than x are arranged on the left side of x. Note that the numbers on the right side are not sorted, and the same goes for the left side. Partition(A, p, r) finally returns the position q of x in the array.

After the operation of the Partition() subroutine, we know that any number on the left side of x must be smaller than the one on the right side, so we only need to sort the elements A[p,…, q-1] on the left side of x and the elements A[q + 1, ..., r] on the right side of x, separately. We can do so by recursively call QuickSort() for the two subproblems.

Figure 4.14 shows an example of the specific implementation of the quicksort algorithm. The elements which are randomly selected each time are marked in red in the figure.

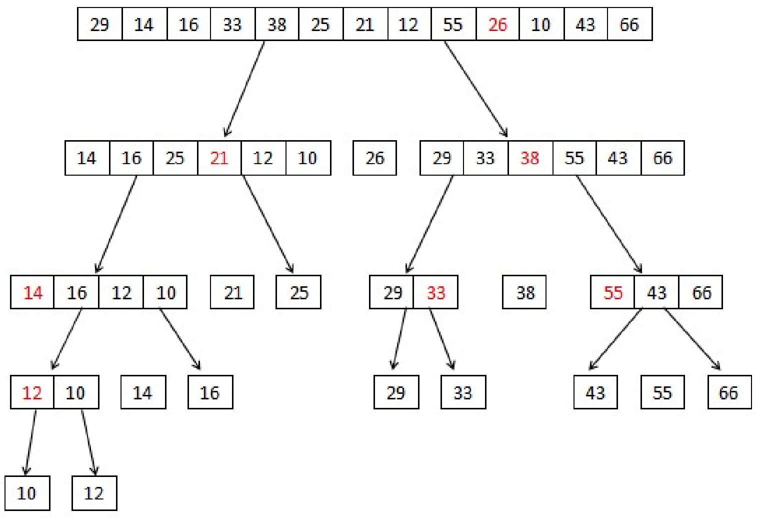


Fig. 4.14 An example run of the quicksort algorithm

Part of the Go code to implement the above quicksort algorithm is shown below, for students who want more details. Note that initially, the input data is stored in a slice variable A.

```
func quicksort(A []int) {
        if len(A) < 2 { return }
        lowerA, upperA := partition(A)
        quicksort(lowerA)
        quicksort(upperA)
}
func partition(A []int) ([]int, []int) {                // return two slices as output
        pivotIndex := rand.Intn(len(A))                 // randomly select a pivot
        pivotValue := A[pivotIndex]
        lower := 0
        A[pivotIndex], A[len(A)-1] = A[len(A)-1], A[pivotIndex]
        for i:= 0; i<len(A); i++ {
                if (A[i]<pivotValue) {
                        A[lower], A[i] = A[i], A[lower]
                        lower++
                }
        }
        A[lower], A[len(A)-1] = A[len(A)-1], A[lower]
        return A[0:lower], A[lower+1:len(A)]
}
```

Finally, let us analyze the performance of the quicksort algorithm. Because the Partition() subroutine called in the quicksort algorithm selects the elements randomly, the running time of the quicksort algorithm is not deterministic but is a random variable. We use $T(n)$ to represent the time required by the quicksort algorithm to sort n unordered numbers, thus $T(n)$ is a random variable.

At best, if each time the Partition() subroutine happens to divide the array into two equal parts, the total sorting time will be very fast. We use $T^{best}(n)$ to represent the time of the algorithm in this lucky case. Then we have

$$T^{best}(n) = 2T^{best}\left(\frac{n}{2}\right) + n.$$

By solving the recursion, we have $T^{best}(n) = O(n \log n)$.

However, if the length of each part is very uneven, then the algorithm will be very slow. For example, in the extreme cases, the algorithm will always choose the largest elements each time. In this case, one of the parts after the partition is the empty set, while the other part contains $n - 1$ element. We use $T^{worst}(n)$ to represent the time of the algorithm in this unlucky case. Then we have

$$T^{worst}(n) = T^{worst}(n-1) + n.$$

By solving the recursion, we have $T^{worst}(n) = O(n^2)$.

So which one is better to represent the performance of the quicksort algorithm? Usually, we use neither **best case** analysis nor **worst case** analysis. Instead, our goal is to analyze the **average case** running time, i.e., the expectation of $T(n)$.

Since we select an element uniformly and randomly every time, the probability of selecting any element is $1/n$. Assuming that the element x selected by the algorithm is ranked i among all the elements of the array, then there are $(i-1)$ elements smaller than x where these elements will be ranked on the left of x and the time required to recursively call the QuickSort() algorithm is $T(i-1)$. Similarly, $(n-i)$ elements are larger than x and will be sorted to the right of x, and the time required to recursively call QuickSort() algorithm is $T(n-i)$. Therefore, the expectation of total sorting time $T(n)$ is

$\mathbb{E}\big(\mathrm{T}(n)\big) = \frac{1}{n}\sum_{i=1}^{n} \mathbb{E}(\mathrm{T}(i-1) + \mathrm{T}(n-i) + n - 1).$

The last item $(n-1)$ is due to the need to compare x with all other elements. After simplifying the above formula, we can get

$\mathbb{E}\big(\mathrm{T}(n)\big) \leq \frac{2}{n}\sum_{i=1}^{n-1} \mathbb{E}(\mathrm{T}(i)) + (n-1).$

By solving the recursion, we have $\mathbb{E}\big(\mathrm{T}(n)\big) = O(n \log n)$. In other words, the expected running time of the quicksort algorithm is $O(n \log n)$, which is significantly faster than the $O(n^2)$ time required by the bubble sort algorithm and insertion sort algorithm in the previous section.

4.3.4. (***) Search Algorithms

When computer science is used to solve a real-world problem, it often happens that the given problem has different nuances, which can be utilized to design different algorithms. As a concrete example, we discuss the search problem by contrasting three algorithms to look up an item in a dictionary. The three algorithms have different time complexities of $O(n)$, $O(\log n)$, and $O(1)$.

A **dictionary** is a set of *n* records, where each record consists of a key-value pair <*key*, *value*>. A **search** operation lookup(*key*) returns the *value* of the key-value pair with a matching *key*.

A dictionary of students of computer science, who are actually pioneers of computer science quoted in the book, is shown in Table 4.5. There are *n*=15 records, each denoting a pioneer. The key is the full name of a student, and the value is the country of birth of the same student, both of which are strings.

Table 4.5 The dictionary StudentsMap for linear search

Key	Value
Berners-Lee, Tim	UK
Wu, Wenjun	China
Godel, Kurt	Austria
Turing, Alan	UK
Knuth, Donald	USA
Leibniz, Gottfried	Germany
Von Neumann, John	Hungary
Amdahl, Gene	USA
Yao, Andrew	China
Moore, Gordon	USA
Yang, Xiong	China
Karp, Richard	USA
Boole, George	UK
Makimoto, Tsugio	Japan
Torvalds, Linus	Finland

22. Linear search in O(n) time

The first algorithm is **linear search**, which searches the dictionary record-by-record and is implemented by the following linear.search.go program. A data structure called *map* is used to represent a dictionary, where the keyword *range* says that the for loop iterates record-by-record for studentsMap. Thus, lookup("Knuth, Donald") needs 5 iterations and returns "USA", and lookup("Babayan, Boris") needs 15 iterations and returns "not found".

```
package main          // The linear.search.go program
import "fmt"
var studentsMap = map[string]string{
    "Berners-Lee, Tim": "UK", "Wu, Wenjun": "China",
    "Godel, Kurt": "Austria", "Turing, Alan": "UK",
    "Knuth, Donald": "USA", "Leibniz, Gottfried": "Germany",
    "Von Neumann, John": "Hungary", "Amdahl, Gene": "USA",
    "Yao, Andrew": "China", "Moore, Gordon": "USA",
    "Yang, Xiong": "China", "Karp, Richard": "USA",
    "Boole, George": "UK", "Makimoto, Tsugio": "Japan",
    "Torvalds, Linus": "Finland",
}
func lookup(studentName string) string {
    for key, value := range studentsMap {
            if studentName == key { return value }
    }
    return "not found"
}
func main() {
    s, t := "Knuth, Donald", "Babayan, Boris"
    fmt.Println(s, "is from", lookup(s))
    fmt.Println(t, "is", lookup(t))
}
```

23. Binary search in $O(\log n)$ time

A more efficient algorithm is **binary search**, which applies to a pre-sorted dictionary, as shown in Table 4.6. The main idea is to narrow down the search space by half with each iteration. Looking up a record with an input key *K* in a sorted *n*-element array *A* needs only $O(\log n)$ iterations. The algorithm goes as follows:

```
Initially: low, high = 0, n-1                    // indices = [0, n-1]
while indices not empty
        mid = (low+high)/2
        if K == A[mid].key then return A[mid].value
        if K < A[mid].key then high=mid-1    // indices = [low, mid-1]
        else low=mid+1                       // indices = [mid+1, high]
return "not found"
```

Binary search has the following notable differences from linear search.

- The dictionary is stored in an array with explicit index. The array is presorted by Key. That is, if j>i, then A[j].key > A[i].key. For instance, 14>8 and A[14].key > A[8].key, because A[14].key = "Yao, Andrew", A[8].key="Moore, Gordon".
- In each iteration, the input *K* is compared to the *Key* of the middle element. If there is a match, the binary search algorithm returns the *Value* of the middle element and stops. If there is a mismatch, the algorithm cuts the search space by half (by adjusting low or high) and goes to the next iteration.
- If no value is returned after all iterations, the input key *K* does not match the key of any of the array elements. The algorithm outputs "not found".

Table 4.6 Initial configuration of the search space for binary search

	Index	Key	Value
low	0	Amdahl, Gene	USA
	1	Berners-Lee, Tim	UK
	2	Boole, George	UK
	3	Godel, Kurt	Austria
	4	Karp, Richard	USA
	5	Knuth, Donald	USA
	6	Leibniz, Gottfried	Germany
mid →	7	Makimoto, Tsugio	Japan
	8	Moore, Gordon	USA
	9	Torvalds, Linus	Finland
	10	Turing, Alan	UK
	11	Wu, Wenjun	China
	12	Von Neumann, John	Hungary
	13	Yang, Xiong	China
high	14	Yao, Andrew	China

For instance, to look up "Knuth, Donald", the search space evolves as follows. At the first iteration, "Knuth, Donald"<"Makimoto, Tsugio", update *high* to 6.
In Iteration 2: "Knuth, Donald">"Godel, Kurt", update *low* to 4

	Index	Key	Value
low	0	Amdahl, Gene	USA
	1	Berners-Lee, Tim	UK
	2	Boole, George	UK
mid →	3	Godel, Kurt	Austria
	4	Karp, Richard	USA
	5	Knuth, Donald	USA
high	6	Leibniz, Gottfried	Germany

In Iteration 3: "Knuth, Donald"="Knuth, Donald", found; return "USA"

	Index	Key	Value
low	4	Karp, Richard	USA
mid →	5	Knuth, Donald	USA
high	6	Leibniz, Gottfried	Germany

It takes 3 iterations to look up "Knuth, Donald" and outputs "USA". What if the input key K does not match any key of the array? Then $\log n$ iterations are needed. For instance, it takes $\log(15) = 4$ iterations to look up "Babayan, Boris" and outputs "not found".
When implementing the binary search algorithm in a Go program, we need to pay special attention to making sure that the given dictionary is a presorted array. That is, the Key field is ordered. We use a *struct* type to define an array element.

```
const n = 15
var studentsArray = [n] struct {
    key         string          //studentName
    value       string          //studentCountry
}
```

Variables key and value are ASCII strings. "Gödel, Kurt" must be written as "Godel, Kurt", since ö is not an ASCII character. "von Neumann, John" must be written as "Von Neumann, John", since 'v' has a large ASCII encoding than capitals.

The complete binary.search.go program follows.

```
package main
import "fmt"
const n = 15
var studentsArray = [n] struct {
        key     string
        value   string
}{
  {"Amdahl, Gene", "USA"}, {"Berners-Lee, Tim", "UK"},
  {"Boole, George", "UK"}, {"Godel, Kurt", "Austria"},
  {"Karp, Richard", "USA"}, {"Knuth, Donald", "USA"},
  {"Leibniz, Gottfried", "Germany"}, {"Makimoto, Tsugio", "Japan"},
  {"Moore, Gordon", "USA"}, {"Torvalds, Linus", "Finland"},
  {"Turing, Alan", "UK"}, {"Von Neumann, John", "Hungary"},
  {"Wu, Wenjun", "China"}, {"Yang, Xiong", "China"},
  {"Yao, Andrew", "China"},
}
func lookup(studentName string) string {
  var low, high, mid int
  low, high = 0, n-1
  for low <= high {
        mid = ( low + high ) / 2
        if studentName == studentsArray[mid].key {
                return studentsArray[mid].value
        }
        if studentName > studentsArray[mid].key {
                low = mid + 1
        } else {  high = mid - 1  }
  }
  return "not found"
}
func main() {
  s, t := "Knuth, Donald", "Babayan, Boris"
  fmt.Println(s, "is from", lookup(s))
  fmt.Println(t, "is", lookup(t))
}
```

The screen outputs are:

```
> go run ./binary.search.go
Knuth, Donald is from USA
Babayan, Boris is not found
>
```

24. Hash search in O(1) time

Binary search ($\mathrm{O}(\log n)$) is much faster than linear search ($\mathrm{O}(n)$). However, some application scenarios need an even faster algorithm with constant complexity, i.e., $\mathrm{O}(1)$. For instance, when one registers for an Internet service, the system may need to instantly check against a trillion-record dictionary, to see if a particular user name, e.g., "johnSmith", is already chosen by another user. When one compiles a document, the document-writing software system constantly checks against a million-record English dictionary, to see if a word just entered is misspelled.

A method called **hashing** can help achieve this goal. The hash search algorithm is based on three basic observations.

- First, although the number of keys and records in the dictionary may be quite large, the number of keys actually stored is much smaller.
- The keys actually stored can be organized as a **hash table**, such that a **hash index** can be computed in $\mathrm{O}(1)$ time from an input key by a **hash function**, to directly access an element of the hash table. That is, a lookup operation needs only $\mathrm{O}(1)$ time, when we are lucky.
- When we are not lucky, several keys may map to the same hash index, a situation called *collision*. The collided keys need to be further organized, e.g., as a linked list. If most of search time is spent on looking through a linked list, the *worst-case* time complexity for lookup becomes $\mathrm{O}(n)$. However, computer science has produced optimized hash search algorithms, such that the *average* time complexity for lookup becomes $\mathrm{O}(1)$.

Here we use the concept of **average time complexity**. A dictionary, once produced, will often be looked up many times. Suppose there are *m* lookup operations in total. Some lookup operations take $\mathrm{O}(n)$ time, and some take only $\mathrm{O}(1)$ time. Assume the i-th lookup operation takes $T(i)$, where $1 \le i \le m$. Then the average time complexity for a lookup operation is $(\sum_{i=1}^{m} T(i))/m$. Suppose 1 trillion lookup operations are performed. One million of them each take $\mathrm{O}(n)$ time, and the rest each take $\mathrm{O}(1)$ time. Then, for each lookup operation, the worst-case time complexity is $\mathrm{O}(n)$, but the average time complexity becomes only $\mathrm{O}(1)$.

As a concrete example of hashing, suppose a legitimate user name consists of 10 digits and letters, such as "johnSmith9". Then the number of possible user names is huge ($62^{10} \approx 8.4 \times 10^{17}$). However, the number of user name strings actually stored could be much smaller, say 1 million. The actually stored keys (user names) and records are organized as a hash table of 1 million elements.

For the studentsMap example, a key (e.g., "Knuth, Donald") is the name of a computer science student, family name first. Again, the possible number of keys is huge. Assume the number of keys and records actually stored is 15 and the hash table has 15 elements. Then there will be no collision. Assume a more realistic case where the number of keys and records actually stored is 15, and the hash table has only 6 elements. Then there will be collisions, and each group of collided records is organized as a linked list.

Similar to linear search, the hash search algorithm is given as input a dictionary of key-value pairs shown in Table 4.5, and a Key to look up. The output is the pairing Value, when the Key is found in the dictionary, or "not found" if otherwise. To implement the hash search algorithm in a Go program hash.search.go, we need to pay attention to the following details.

- Implement a linked list as a number of records connected by pointers.
- Implement a hash table as an array of such records.
- The value of an array index is computed by a hash function.
- The main function first fills up the hash table with record items.
- To lookup a Key such as "Knuth, Donald", first use its hash function output as index to access the array element of the hash table. If "Knuth, Donald" is found, return his country "USA". If not found there, continue traversing the linked list.

The key-value pairs of Table 4.5 are initialized in a map variable studentsMap.

```
var studentsMap = map[string]string{
  "Berners-Lee, Tim": "UK", "Wu, Wenjun": "China",
  "Godel, Kurt": "Austria", "Turing, Alan": "UK",
  "Knuth, Donald": "USA", "Leibniz, Gottfried": "Germany",
  "Von Neumann, John": "Hungary", "Amdahl, Gene": "USA",
  "Yao, Andrew": "China", "Moore, Gordon": "USA",
  "Yang, Xiong": "China", "Karp, Richard": "USA",
  "Boole, George": "UK", "Makimoto, Tsugio": "Japan",
  "Torvalds, Linus": "Finland",
}
```

- (1) Implement a linked list as a number of records connected by pointers.

The variable Record in hash.search.go is similar to studentsArray of the binary search example. They both use a struct data type to represent key-value pairs.

```
var studentsArray = [n] struct {
     key          string          // studentName
     value        string
}
```

However, variable Record is a three-field structure. In addition to the key-value pair, it has a *next* field, which is a pointer to the next Record in the linked list, where *Record denotes the memory address of a Record.

```
type Record struct {
  next            *Record
  studentName     string
  studentCountry  string
}
```

- (2) Implement a hash table as an array of such records.

Variable hashTable denotes an array of 6 elements, with array indices 0, 1, 2, 3, 4, 5. Each element hashTable[i] holds a pointer to a Record.

```
const HashTableSize = 6
var hashTable [HashTableSize] *Record
```

- (3) The value of an array index is computed by a hash function.

The element of an array A at index i is accessed by A[i]. In a hash table, the array element for studentName is accessed by first computing a hashFunction.

```
hashIndex := hashFunction(studentName)
entry := hashTable[hashIndex]
```

The hash function first finds the sum of ASCII values of characters in studentName, using code from the student name coding exercise. Then the modulus operator % is used to get the remainder of sum mod 6, where 6 is the size of the hash table.

```
func hashFunction(name string) int {
  sum := 0
  for i := 0; i < len(name); i++ { sum = sum + int(name[i]) }
  return sum % HashTableSize
}
```

For instance, given key="Berners-Lee, Tim", we have

```
 sum of "Berners-Lee, Tim" = 1418
 sum % hashTableSize = 1418 % 6 = 2
```

Thus, hashFunction("Berners-Lee, Tim") returns 2.

- (4) Before any looking up, the hash table needs to be filled with record items.

The action of filling out hashTable is done by the following code. Recall that the for loop ranges over studentsMap, one record (key-value pair) at a time. For each key-value pair, a new record is created and inserted to a hashTable element.

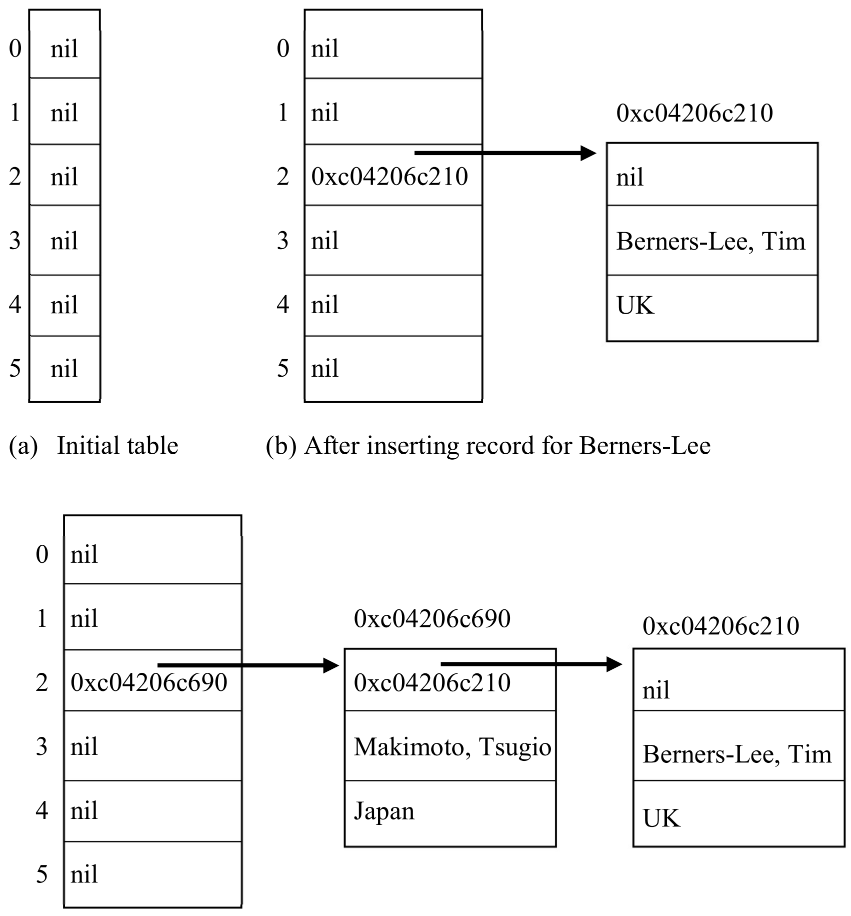
```
for key, value := range studentsMap {              // fill out hashTable
    hashIndex := hashFunction(key)
    newRecord := &Record{          // & denotes the address of Record
      next:              hashTable[hashIndex],
      studentName:       key,
      studentCountry:    value,
    }
    hashTable[hashIndex] = newRecord
  }
```

Initially, all elements of the array variable hashTable are initialized to the zero value of pointer, which is *nil*. That is, hashTable[i] contains value nil for all i.

Consider the case when the for loop goes to the key-value pair "Berners-Lee, Tim": "UK", where key= "Berners-Lee, Tim" and value= "UK". We know that the statement

```
hashIndex := hashFunction(key)
```

assigns 2 to hashIndex. The next few statements creates and inserts a new record for "Berners-Lee, Tim" to the element hashTable[hashIndex], that is, hashTable[2]. More specifically, we see the following execution steps, illustrated in Fig. 4.15.



(a) Initial table (b) After inserting record for Berners-Lee

(c) After inserting record for Makimoto

Fig. 4.15 The hash table with associated linked list for hashTable[2].

Figure 4.15a shows the initial configuration of hashTable, where all elements contains nil: the pointer points to nowhere. Figure 4.15b shows the configuration after the record for Berners-Lee is created and inserted. This new record has an address newRecord, and its three fields have the following values:

newRecord.next:	nil
newRecord.studentName:	"Berners-Lee, Tim"
newRecord.studentCountry:	"UK"

The system allocates memory space for this record, which happens to assign address 0xc04206c210 to newRecord.

Figure 4.15c shows the configuration after the record for Makimoto is created and inserted. Note that this record is inserted at the front of the linked list. This new record has an address newRecord, and its three fields have the following values:

newRecord.next:	0xc04206c210
newRecord.studentName:	"Makimoto, Tsugio"
newRecord.studentCountry:	"Japan"

The address of this record, newRecord, is 0xc04206c690. This is an address automatically generated by the Go programming language system. It may change when the same program executes again. That is why we use a more abstract arrowed line to denote a pointer. Figure 4.16 shows the filled-out hash table and linked lists.

- (5) A lookup operation first finds the hash table array element, and then traverses the linked list.

To look up the record for the input key "Knuth, Donald", we have

sum of "Knuth, Donald" = 1192; sum % hashTableSize = 1192 % 6 = 4

Thus, the hash table element is hashTable[4], which points to the record for Amdahl. Since the key "Amdahl, Gene" does not match the input key "Knuth, Donald", the program goes to the next record by following the *next* field. Then we have a match.

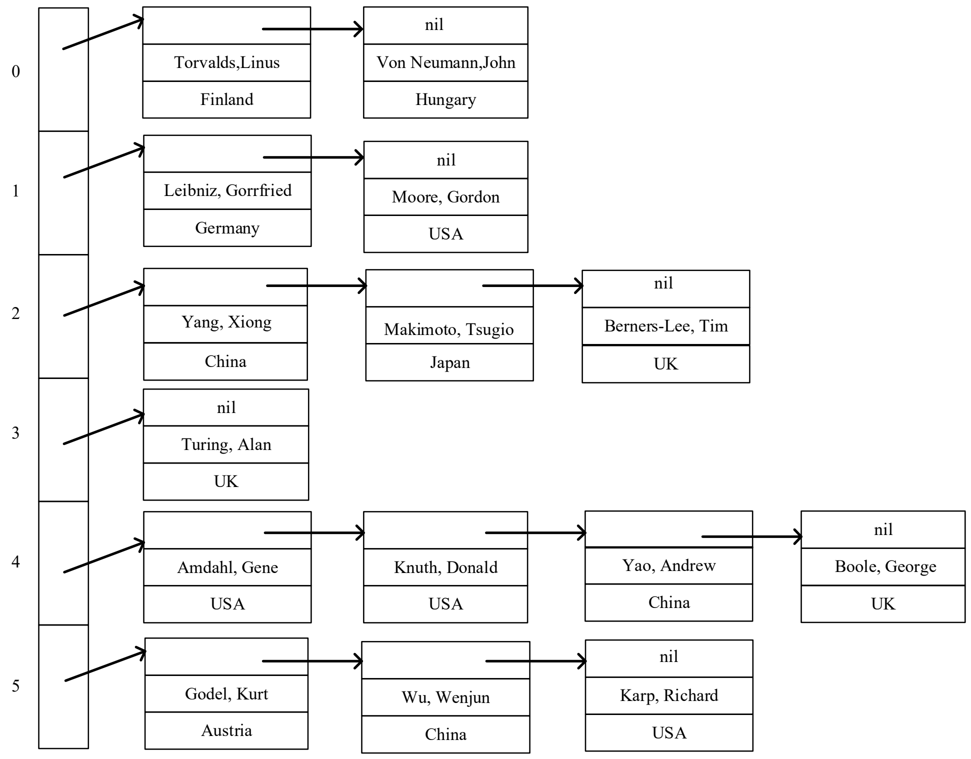


Fig. 4.16 The hash table with associated linked lists for studentsMap

The complete hash.search.go program follows.

```
package main
import "fmt"

type Record struct {
  next              *Record
  studentName       string
  studentCountry    string
}

const HashTableSize = 6
var hashTable [HashTableSize] *Record

func hashFunction(name string) int {
  sum := 0
  for i := 0; i < len(name); i++ { sum = sum + int(name[i]) }
  return sum % HashTableSize
}

func lookup(studentName string) string {
  hashIndex := hashFunction(studentName)
  entry := hashTable[hashIndex]
  current := entry
  for current != nil {
    if current.studentName == studentName {
      return current.studentCountry           // keys match
    }
    current = current.next                    // otherwise goto next
  }
  return "not found"
}

var studentsMap = map[string]string{
  "Berners-Lee, Tim": "UK", "Wu, Wenjun": "China",
  "Godel, Kurt": "Austria", "Turing, Alan": "UK",
  "Knuth, Donald": "USA", "Leibniz, Gottfried": "Germany",
  "Von Neumann, John": "Hungary", "Amdahl, Gene": "USA",
  "Yao, Andrew": "China", "Moore, Gordon": "USA",
  "Yang, Xiong": "China", "Karp, Richard": "USA",
  "Boole, George": "UK", "Makimoto, Tsugio": "Japan",
  "Torvalds, Linus": "Finland",
}
```

```
func main() {
  for key, value := range studentsMap {            // fill out hashTable
    hashIndex := hashFunction(key)
    newRecord := &Record{
      next:             hashTable[hashIndex],
      studentName:      key,
      studentCountry:   value,
    }
    hashTable[hashIndex] = newRecord
  }
  s, t := "Knuth, Donald", "Babayan, Boris"        // look up s and t
  fmt.Println(s, "is from", lookup(s))
  fmt.Println(t, "is", lookup(t))
}
```

The screen outputs are:

```
> go run ./hash.search.go
Knuth, Donald is from USA
Babayan, Boris is not found
>
```

4.4. P vs. NP

In the previous sections, we have learned a variety of effective methods to design smart algorithm for many problems. We always try to solve a problem as fast as possible. For example, in the sort algorithm, while the bubble sort algorithm and the insertion sort algorithm need $O(n^2)$ time, the merge sort algorithm needs only $O(n \log n)$ time. We call $O(n^2)$ or $O(n \log n)$ the time complexity of the corresponding algorithm. Thus, the merge sort algorithm is more efficient.

One may wonder if it is possible to design an even faster algorithm for the sorting problem. In this section, we will briefly introduce *complexity theory* which focuses on the complexity of a computational problem, instead of any particular algorithm. We want to answer the following question systematically:

Are some computational problems inherently difficult to solve effectively?

In section 4.4.1, we use the sorting problem as an example to illustrate the time complexity of a computational problem. We then introduce the most important complexity classes P and NP in section 4.4.2. In the final section, we show several examples of P and NP.

4.4.1. Time Complexity

Intuitively, the time complexity of a computational problem is the minimum time complexity of any algorithm which can solve this problem. It is an important measurement to understand how difficult a problem is.

Consider the sorting problem discussed before. Since we have merge sort algorithm to solve it in $O(n \log n)$ time, the time complexity of sorting problem is at most $O(n \log n)$. But is there any other algorithm which can solve the sorting problem in $o(n \log n)$ time? This question is hard to answer since the answer depends on the usage scenario. We need a more precise model to discuss the hardness of the sorting problem.

Let us assume that the elements to be sorted can only be compared in pairs. That is, we can treat every element as a black box, the only way to know the order of element a and b is to compare them with one comparison step. Of course, we assume any pair of elements can be compared and the results form a total order over all elements.

Under these assumptions, we will show that any algorithm needs at least $n \log n$ steps of comparison operations in the worst case. The idea comes from the information theory. The number of all possible orders is $n!$, which is actually the number of all possible permutations over $\{1,2,\dots,n\}$. For each comparison step, the result will give us 1 bit information, either $a < b$ or $a > b$. Thus, in the worst case, if some algorithm can use k comparison steps to distinguish all possible orders, it means that $2^k \geq n!$. Thus, the time complexity of any algorithm which can solve sorting problem is $\Omega(n \log n)$. We call it the lower bound of time complexity for sorting problem, and it illustrates how hard the problem is.

Thus, the time complexity for the sorting problem is $\Theta(n \log n)$. We say the merge sort algorithm is asymptotic optimal.

4.4.2. P and NP

The important work of computational complexity theory is to determine the time complexity and space complexity of various problems. But it is difficult to decide the exact time complexity for all problems, so the researchers create many complexity classes where each complexity class contains various problems whose complexity are similar in some ways. P and NP are two most famous complexity classes in the computational complexity theory.

Intuitively, the complexity class P contains all decision problems which have polynomial time algorithms. Here, decision problem means that the output of the problem is either YES or NO, and the polynomial time algorithm means the time complexity is a polynomial function over the size of input.

Complexity class P contains all decision problems that can be solved by a deterministic Turing machine using a polynomial amount of computation time.

In practical applications, people usually refer to polynomial time algorithms as effective algorithms. Therefore, an important task in the field of computational complexity is to determine whether certain computational problems have polynomial time algorithms. In other words, decide whether such problems belong to P or not. For example, the sorting problem is in P since it has a polynomial time algorithm. The bubble sort algorithm, insertion sort algorithm, merge sort algorithm and quicksort algorithm are all polynomial time algorithms.

However, people gradually discovered that many basic problems are in such a gray area: we have neither found polynomial time algorithms to solve these problems, nor have we been able to prove that such algorithms do not exist. These problems involve a wide range of areas, distributed in various fields: operational research, optimization, combinatorics, logic, artificial intelligence, big data processing, and so on. Even after more than half a century of development, theoretical computer scientists are still at a loss for this gray area, and most of their problems still stay in this gray area. However, researchers have made great progress in the characterization of these problems and found that a large class of these problems belongs to NP, another important complexity class we introduce next.

Complexity class NP contains all decision problems that can be solved by a *non-deterministic* Turing machine using a polynomial amount of computation time.

However, this definition is slightly difficult to use. We will not introduce non-deterministic Turing machine in this book, the interested readers please refer to any complexity theory textbook. Here, we introduce an equivalent definition of the class NP.

Complexity class NP (equivalent definition): contains all decision problems that whose "YES" answers can be verified in polynomial time by a *deterministic* Turing machine. That is, there exists a checking algorithm which can verify the correctness of "YES" answer with the help of a witness in polynomial time.

More precisely, a decision problem $A \in$ NP, if and only if there exists a polynomial time algorithm S which is a checking algorithm and two constants c, c' such that the following two conditions hold:

1) For any input instance x whose correct answer should be "YES", there exists a witness y such that $|y| \leq c'|x|^c$ and if the algorithm S takes (x, y) as the input, it will output "YES";
2) For any input instance x whose correct answer should be "NO", for any witness y who satisfies $|y| \leq c'|x|^c$, if the algorithm S takes (x, y) as the input, it will always output "NO".

Now, let us use a concrete example to help understand the definition of NP.

Example 35. The subset sum problem

Consider the following problem: given $2n$ integers, we want know whether it can be partitioned into two groups whose sums are the same. The integers are not required to be different from each other.

For example, if the input is $\{1,2,3,4,5,5\}$, the answer is YES since partition $\{1,2,3,4\}$ and $\{5,5\}$ have the same sum. If the input is $\{1,2,3,4,5,6\}$, the answer is NO since the sum of all integers is 21 which is an odd number.

This problem is quite important in cryptograph. However, till now, we do not know how to solve this problem in polynomial time over n. The naïve algorithm is to check all possible partitions. The time complexity of the naïve algorithm is $O(n2^n)$ which is exponential over n. On the other hand, we do not know how to prove that such problem cannot be solved in polynomial time. This is one example of the problems in the gray area we mentioned before. Here, we want to show that the subset sum problem is in NP.

To show that some problem belongs to NP, we need to construct the checking algorithm S and for any YES instance, construct the witness. For the subset sum problem, if x is a YES instance, we construct the witness $y = (y_1, y_2)$ where

(y_1, y_2) is a partition of input x and the sums of y_1 and y_2 are the same. The checking algorithm S is designed to check two things: 1) the witness $y = (y_1, y_2)$ is a partition of input instance x, i.e., every element in x appears exactly once in either y_1 or y_2; and 2) the sum of integers in y_1 is the same as the sum of integers in y_2. Obliviously, the size of witness is a polynomial over the size of input, and the checking algorithm also runs in polynomial time. It is also easy to show that for any NO instance, it is impossible to construct a witness which can pass the checking algorithm. Thus, we have shown that the subset sum problem is in NP.

☷

For any decision problems in P, it is also in NP, since we can compute the correct answer for any instance within polynomial time and we do not need witness at all. Thus, we have P ⊆ NP. In the computer science field, one of the most fundamental and far-reaching issues is whether the opposite direction holds or not, that is, if some decision problem can be verified efficiently, is it always be computed efficiently? The problem is called P versus NP problem:

P versus NP problem: is P=NP?

This question is one of the seven millennium prize problems. At present, among these seven problems, only the Poincaré conjecture has been solved by the mathematician Grigori Perelman in 2003. The remaining six problems have not been solved. Although it is still unconfirmed whether P is equal to NP, most scientists believe that the equality does not hold, that is, there exists some decision problem which can be efficiently verified but cannot be efficiently computed.

4.4.3. (***) Examples in the NP Class

As we mentioned before, in the complexity class NP, there are many fundamental problems which we do not know whether they belong to P or not. We discuss two more examples of them: the graph coloring problem and the Hamiltonian path problem.

Example 36. The graph coloring problem

The graph coloring problem can be specified as follows:

- **Input**: a graph $G = (V, E)$ where V is the set of vertices and E is the set of edges, an integer k;
- **Output**: decide whether there is a way to color each vertex with one of the k colors so that no adjacent vertices are of the same color.

Similar to the subset sum problem, the computer science community still does not know how to solve this problem in polynomial time over $n = |V|, m = |E|$ and k. The naïve algorithm is to check all possible coloring methods and the time complexity is $O(mn^k)$, which makes it an exponential time algorithm.

We now show the graph coloring problem belongs to NP. The witness of a YES instance is the valid way of coloring $f: V \to \{1,2, \dots, k\}$. The size of the witness is definitely polynomial over the size of input. The checking algorithm is straightforward. We only need to check for each edge, two endpoints do not have the same color. It is easy to check if the graph cannot be colored within k colors, it is impossible to find a valid way of coloring. Thus, Graph coloring problem belongs to NP.

☶

Example 37. The Hamiltonian path problem

The Hamiltonian path problem can be specified as follows:

- **Input**: a graph $G = (V, E)$ where V is the set of vertices and E is the set of edges;
- **Output**: decide whether there is a Hamiltonian path in this graph. A Hamiltonian path is a path which visits each vertex exactly one.

Again, we do not know how to solve this problem in polynomial time over $n = |V|$. The naïve algorithm is to check all possible paths so that the time complexity is $O(n \cdot n!)$ which is exponential over n.

We show that the Hamiltonian path problem belongs to NP. The witness of a YES instance is the valid Hamiltonian path $V_1, V_2, \dots, V_n$ which can also be seen as a permutation over V. The size of the witness is definitely polynomial over the size of input. The checking algorithm is also straightforward. We need to check two things: firstly, there is an edge between V_i and V_{i+1} for $i = 1, 2, \dots, n-1$; secondly, $V_1, \dots, V_n$ are different from each other. It is easy to check if the graph does not contain a Hamiltonian path, it is impossible to find some witness which can pass the checking algorithm. Thus, Hamiltonian path problem belongs to NP.

☶

In these two examples, it seems trivial to show that their decision problems belong to the complexity class NP. Actually, they not only belong to the class NP, but also are the most difficult problems in NP. They are **NP-complete** problems. If you can find a polynomial time algorithm for either the graph coloring problem or the Hamiltonian path problem, then every decision problem in NP has a polynomial time algorithm, which means P=NP. On the other hand, if you can prove that either the graph coloring problem or the Hamiltonian path problem does not have a polynomial time algorithm, you show P≠NP. So, if you are interested in P versus NP problem, you do not need to consider a class of problem. Instead, you can just think about one particular decision problem, for example, the graph coloring problem or the Hamiltonian path problem. Any ideas?

4.5. Exercises

1. Refer to Example 30. Show that Euclid's algorithm indeed computes the result gcd(x, y)=12 in three steps, given inputs x=36 and y=24.
2. Refer to Example 30. Show that Euclid's algorithm is indeed an algorithm, because it satisfies Knuth's five properties.
3. An algorithm must have at least one output, but it may not have any input. Please give an example of a meaningful algorithm without any input.
4. Which of the following statement is correct?

   (a) $0.1n^2$ is $O(n)$
   (b) $10000n$ is $O(n^2)$
   (c) $n \log n$ is $\Omega(n)$
   (d) $10n^2 - 10n + 1$ is $\Theta(n)$

5. Which of the following statement is correct?

   (a) $\frac{n^3}{\ln n} + n^2$ is $O(n^2)$
   (b) $n^2 + \frac{2n^2 \log n}{\log \mathrm{lo}}$ is $\Theta(n^2 \log n)$
   (c) 2^{n^2} is $\Omega(n)$
   (d) $(\ln n)^{\ln n}$ is $O(n^{100})$

6. Please sort the following asymptotic formulas from small to large: $\Theta(\log n)$, $\Theta(n)$, $\Theta\left(\frac{n}{\log n}\right)$, $\Theta(n \log n)$.
7. Please sort the following asymptotic formulas from small to large: $\Theta((\log n)^n)$, $\Theta(n^{100})$, $\Theta\left(n^{\log n}\right)$, $\Theta\left((\log n)!\right)$.
8. The Sunway TaihuLight supercomputer was the world's fastest supercomputer from June 2016 to June 2018. It can finish 93 million billion operations per second. If we use this supercomputer to compute Steiner tree problem with 1000 vertices, how long do we need? The best algorithm for Steiner tree problem runs in $n^{\log_2 n}$ time where n is the number of vertices.
9. Given an unsorted array [1, 3, 9, 7, 6, 7, 8, 5, 2, 4], please describe the executing processes of the following sorting algorithms: bubble sort, insertion sort, merge sort and quicksort algorithms.
10. The insertion sort algorithm and the merge sort algorithm have a time complexity of $O(n^2)$ and $O(n \log n)$, respectively. What is the main reason that merge sorting is more efficient than insertion sorting?
11. For the single-factor optimization problem, why is that the equal division algorithm is not as efficient as the golden section algorithm? The textbook stated that it is smart to divide the problem into two equal sized subproblems.
12. For the integer multiplication problem, what is the main reason that the naïve algorithm of divide and conquer is not efficient?

13. Refer to the integer multiplication problem in Section 4.2.4. If you divide two numbers into 3 segments (each segment is an n/3-digit number), is it possible to find a faster algorithm for integer multiplication?
14. Consider the sorting problem of n numbers. <u>In the worst case</u>, how many comparisons do we need in the quicksort and the bubble sort algorithms?

    (a) $O(n\log n)$, $O(n^2)$
    (b) $O(n^2)$, $O(n^2)$
    (c) $O(n)$, $O(n\log n)$
    (d) $O(n\log n)$, $O(n\log n)$

15. Given a sorted array with n numbers, what is the running time if we want to check whether element x and y are in this array?

    (a) $\Theta(1)$
    (b) $\Theta(\log n)$
    (c) $\Theta(n/\log n)$
    (d) $\Theta(n)$

16. Solve the following recursion: $T(n) = (\ )$
$$\begin{cases} T(1) = 1 \\ T(n) = 2T(n-1) \end{cases}$$
17. Solve the following recursion: $T(n) = (\ )$
$$\begin{cases} T(1) = 1 \\ T(n) = 3T(\frac{n}{2}) + n^2 \end{cases}$$
18. Solve the following recursion: $T(n) = (\ )$
$$\begin{cases} T(1) = 1 \\ T(n) = 2T\left(\lfloor\frac{n}{\sqrt{2}}\rfloor\right) + n^2 \end{cases}$$
19. 128 students take part in a table tennis match. Assume the ability of the students forms a total order. If we want to decide the champion, how many matches do we need? If we want to decide the champion and runner-up, how many matches do we need?

    (a) 127, 128
    (b) 127, 133
    (c) 127, 192
    (d) 127, 253

20. There are 16 bottles of liquid and one of them is poisonous. The poisonous one can make the mouse die immediately. Now we want to know which bottle is poisonous. Each time, we can mix the liquid from several bottles and let one

mouse drink it. For each mouse, it can only drink once. In the worst case, how many mice do we need to find the poisonous bottle?

21. Suppose you are in a skyscraper and have one egg in your hand. You want to know from which floor can the egg fall without breaking. If you test some floor but the egg is broken, you have no egg to test anymore. Thus, the only feasible solution is to test one more floor each time. Now, consider you have two identical eggs. How can you achieve the goal with as fewer number of tests as possible? You can assume that the egg must be broken if it falls from the top floor which is the n-th floor.
22. In the stable matching problem, is the stable matching unique? If it is not unique, please give an instance where there are at least two different stable matching.
23. In the stable matching problem, is the Gale-Shapley algorithm beneficial for boys or girls? Why?
24. Does the following decision problem belong to NP?

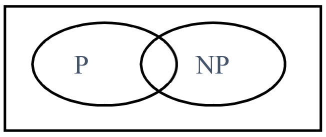
- Problem: Given 2n integers, decide whether we can partition them into two sets (each set contains n integers) where the sum of two sets is equal.

25. Does the following decision problem belong to NP?

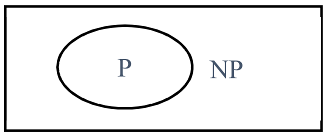
- Problem: Given two integers x and y, decide if x is a multiple of y.

26. What is the relation between P and NP? Assume the rectangle represents all decision problems which can be computed by Turing machine.

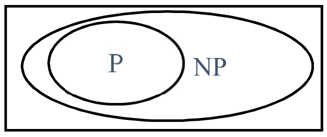
(a)



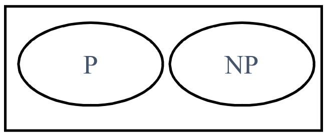
(b)



(c)



(d)



4.6. Bibliographic Notes

The chapter quotation is from Professor Brian Kernighan of Princeton University [1]. Professor Donald Knuth of Stanford University gave a five-point definition of algorithm [2]. Strassen's algorithm for matrix multiplication and improvements later can be found in [3-4]. The stable match problem is studied in [5-6]. The Clay Mathematics Institutes listed seven fundamental mathematic problems as the Millennium Prize problems, which are "important classic questions that have resisted solution for many years." [7]. One of them is the P vs. NP problem.

[1] Kernighan, B. W. (2017). Understanding the digital world: What you need to know about computers, the internet, privacy, and security. Princeton University Press.

[2] Knuth, D. E. (1997). The Art of Computer Programming. Addison-Wesley.

[3] Volker Strassen. (1969). Gaussian elimination is not optimal. Numerische Mathematik 13, 354–356.

[4] Karstadt, E., & Schwartz, O. (2017). Matrix multiplication, a little faster. In Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures (pp. 101-110).

[5] Gale, D., & Shapley, L. S. (1962). College admissions and the stability of marriage. The American Mathematical Monthly, 69(1), 9-15.

[6] Chen, J., Skowron, P., & Sorge, M. (2019). Matchings under preferences: Strength of stability and trade-offs. In Proceedings of the 2019 ACM Conference on Economics and Computation (pp. 41-59).

[7] https://www.claymath.org/millennium-problems.