离散数学及其应用 第3章 算法

3. Algorithms

3.1 Introduction to Algorithm

In many domains there are key general problems that ask for output with specific properties when given valid input.
To precisely state the problem, using the appropriate structures to specify the input and the desired output. Solve the general problem by specifying the steps of a procedure that takes a valid input and produces the desired output. This procedure is called an algorithm.

3.1.1 Basic Concepts

`Algorithms` : An algorithm is a finite set of precise instructions for performing a computation or for solving a problem.
`Pseudocode` : An intermediate step between an English language description of the steps and a coding of these steps using a programming language.
⚠️ Algorithms can be specified in different ways. Their steps can be described in English or in pseudocode. Pseudocodes are clearer to read than natural languages and easier to understand than programming languages. Moreover, pseudocodes are independent from any specific programming languages, so it is more transplantable.
`Input` : An algorithm has input values from a specified set.

`Output` : From the input values, the algorithm produces the output values from a specified set. The output values are the solution.
And the followings are properties we expect algorithms to have:

- `Definiteness` : The steps of an algorithm must be defined precisely.
- `Correctness` : An algorithm should produce the correct output values for each set of input values.
- `Finiteness` : An algorithm should produce the output after a finite number of steps for any input.
- `Effectiveness` : It must be possible to perform each step of the algorithm in a finite amount of time.
- `Generality` : The algorithm should work for all problems of the desired form.

3.1.2 Example Algorithm Problems

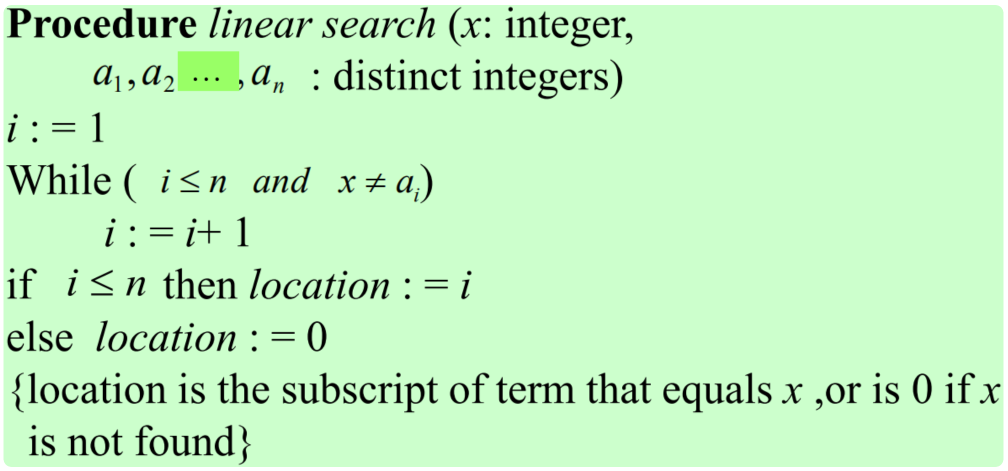
`Algorithmic Paradigms` : A general approach based on a particular concept that can be used to construct algorithms for solving a variety of problems.

- Greedy algorithm
- Brute-force algorithm
- Divide-and-conquer algorithms
- Dynamic programming
- Backtracking
- Probabilistic algorithms

📋 Searching Problems

Locate an element x in a list of finite distinct elements $a_1, a_2, ..., a_n$, or determine that it is not in the list.

- Linear Search Algorithm in Pseudocode



```
Procedure linear search ($x$: integer,
        $a_1, a_2 ... , a_n$ : distinct integers)
$i := 1$
While ( $i \le n$ and $x \ne a_i$)
        $i := i + 1$
if $i \le n$ then location $:= i$
else location $:= 0$
{location is the subscript of term that equals $x$ ,or is 0 if $x$
 is not found}
```

- Binary Search Algorithm in Pseudocode

```
**Procedure** *binary search* ($x$: integer,
        $a_1, a_2, \ldots, a_n$ : increasing integers)
$i := 1$ {$i$ is left endpoint of search interval }
$j := n$ { $j$ is right endpoint of search interval }
while $i < j$
begin
    $m := \lfloor (i + j)/2 \rfloor$
    if $x > a_m$ then $i := m + 1$
    else $j := m$
end
if $x = a_i$ then *location* $:= i$
else *location* $:= 0$
**{location is the subscript of term equal to $x$ ,or 0 if x is not found }**
```

The binary search algorithm iteratively restricts the relevant search interval until it closes in on the position of the element to be located. It is more efficient than linear search, but it only applies to searching increasing sequences.

📋 Sorting Problems

To sort the elements of a list is to put them in increasing order (numerical order, alphabetic, and so on). It is an important problem with many different algorithms for various situations.

- Bubble Sort Algorithm

```
**procedure** *bubblesort*($a_1, \ldots, a_n$ : real numbers with $n \geq 2$)
**for** $i := 1$ **to** $n - 1$
    **for** $j := 1$ **to** $n - i$
        **if** $a_j > a_{j+1}$ **then** interchange $a_j$ and $a_{j+1}$
{$a_1, \ldots, a_n$ is in increasing order}
```

- Insertion Sort Algorithm

```
procedure insertion sort($a_1, a_2, \ldots, a_n$: real numbers with $n \geq 2$)
for $j := 2$ to $n$
    $i := 1$
    while $a_j > a_i$
        $i := i + 1$
    $m := a_j$
    for $k := 0$ to $j - i - 1$
        $a_{j-k} := a_{j-k-1}$
    $a_i := m$
{$a_1, \ldots, a_n$ is in increasing order}
```

📋 Optimization Problems

Minimize or maximize some parameter over all possible inputs.

- Greedy Algorithms

Make the "best" choice at each step.

Making the "best choice" at each step does not necessarily produce an optimal solution to the overall problem, but in many instances, it does. It requires proof that the local best choices can lead to overall optimal solutions in the specific case before using the greedy algorithm.

3.2 The Growth of Functions

In computer science, we want to understand how quickly an algorithm can solve a problem as the size of the input grows, so that we can compare the efficiency of two different algorithms for solving the same problem and determine whether it is practical to use a particular algorithm as the input grows.

3.2.1 Big-O Notations

`Big-O Notations` Let f and g be functions. We say that $f(x) = O(g(x))$, read as " $f(x)$ is big-oh of $g(x)$ " if there exist constants C and k such that $|f(x)| \leq C|g(x)|$ whenever $x > k$.

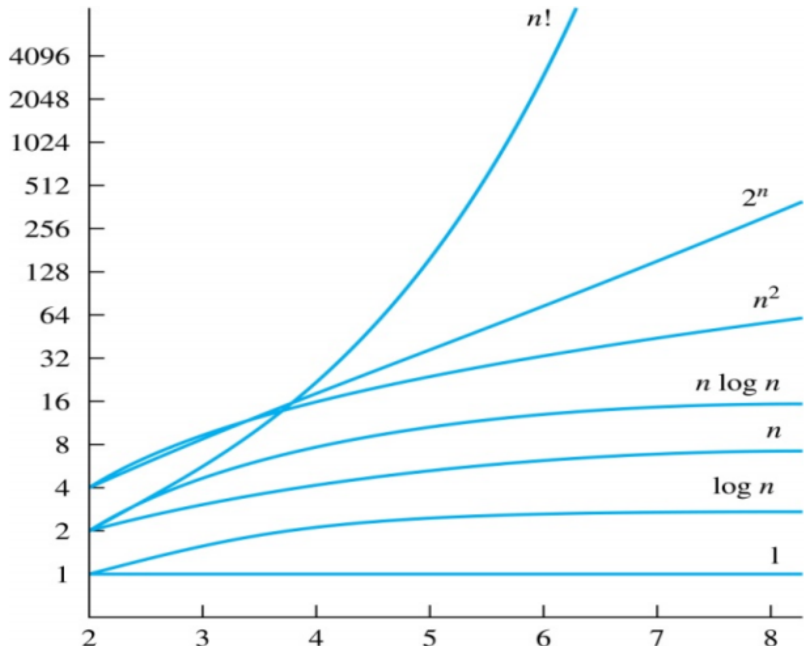
The constants C and k are called `witnesses` to the relationship $f(x)$ is $O(g(x))$. Only one pair of witnesses is needed to show $f(x) = O(g(x))$, and there are infinite pairs of witnesses if there is a pair.

If $f(x) = O(g(x))$ and $g(x) = O(f(x))$, we say that the two functions are of the `same order` .

For many applications, the goal is to select the function $g(x)$ in $O(g(x))$ as small as possible (up to multiplication by a constant, of course).

📋 If $f(x) = O(g(x))$ and $h(x)$ is larger than $g(x)$ when x is sufficiently large, then $f(x) = O(h(x))$.

3.2.2 Big-O Estimates for some Important Functions



Other functions are usually represented as big-O of these functions.

🌰 e.g.

Use big-O notation to estimate the sum of the first n positive integers.

Solution:

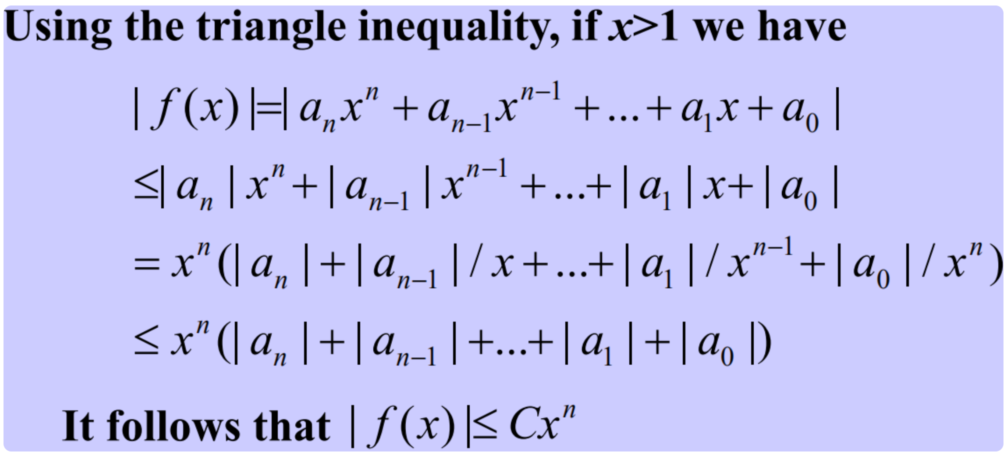
$$1 + 2 + ... + n \le n + n + ... + n = n^2$$

So $1 + 2 + ... + n$ is $O(n^2)$ taking $C = k = 1$.

📋 Big-O Estimates for Polynomials

Let $f(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$, where $a_0, a_1, ..., a_n$ are real numbers, then $f(x) = O(x_n)$. The leading term of a polynomial $a_n x^n$ dominates its growth.

Proof:



Using the triangle inequality, if x>1 we have

$$
\begin{aligned}
|f(x)| &= |a_n x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0| \\
&\le |a_n| x^n + |a_{n-1}| x^{n-1} + \ldots + |a_1| x + |a_0| \\
&= x^n (|a_n| + |a_{n-1}|/x + \ldots + |a_1|/x^{n-1} + |a_0|/x^n) \\
&\le x^n (|a_n| + |a_{n-1}| + \ldots + |a_1| + |a_0|)
\end{aligned}
$$

It follows that $|f(x)| \le Cx^n$

📋 Useful Big–O Estimates Involving Logarithms, Powers, and Exponents

- If $d > c > 1$, then $n^c = O(n^d)$, but n^d is not $O(n^c)$.
- If $b > 1$ and c and d are positive, then $(\log_b n)^c = O(n^d)$, but n^d is not $O((\log_b n)^c)$.
    - Every positive power of the logarithm of n to the base b , where $b > 1$, is big–O of every positive power of n , but the reverse relationship never holds.
- If $b > 1$ and d is positive, then n^d is $O(b^n)$, but b^n is not $O(n^d)$.
    - Every power of n is big–O of every exponential function of n with a base that is greater than one, but the reverse relationship never holds.
- If $c > b > 1$, then $b^n = O(c^n)$, but c^n is not $O(b^n)$.
    - if we have two exponential functions with different bases greater than one, one of these functions is big–O of the other if and only if its base is smaller or equal.

📋The Growth of Combinations of Functions

- If $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$, then $(f_1 + f_2)(x)$ is $O(\max(g_1(x), g_2(x)))$.
- If $f_1(x)$ and $f_2(x)$ are both $O(g(x))$, then $(f_1 + f_2)(x)$ is $O(g(x))$.

Proof:

- By the definition of big-O notation, there are constants C_1, C_2, k_1, k_2 such that $|f_1(x) \le C_1|g_1(x)$ when $x > k_1$ and $f_2(x) \le C_2|g_2(x)|$ when $x > k_2$.
- $$\begin{aligned}|(f_1+f_2)(x)| &= |f_1(x)+f_2(x)| \\ &\le |f_1(x)| + |f_2(x)| \quad \text{by the triangle inequality } |a+b| \le |a|+|b|\end{aligned}$$
- $$\begin{aligned}|f_1(x)| + |f_2(x)| &\le C_1|g_1(x)| + C_2|g_2(x)| \\ &\le C_1|g(x)| + C_2|g(x)| \quad \text{where } g(x) = \max(|g_1(x)|,|g_2(x)|) \\ &= (C_1+C_2)|g(x)| \\ &= C|g(x)| \quad \text{where } C = C_1 + C_2\end{aligned}$$
- Therefore $|(f_1+f_2)(x)| \le C|g(x)|$ whenever $x > k$, where $k = \max(k_1,k_2)$.

3.2.3 Other Notations

`Big-Omega Notation` : Let f and g be functions. We say that $f(x) = \Omega(g(x))$, read as " $f(x)$ is big-omega of $g(x)$ ", if there are constants C and k such that $|f(x)| \ge C|g(x)|$ whenever $x > k$.

$f(x) = \Omega(g(x)) \leftrightarrow g(x) = O(f(x))$. This follows from the definitions.

⚠️ Big-O gives an upper bound on the growth of a function, while Big-Omega gives a lower bound. Big-O tells us that a function grows no faster than another, while Big Omega tells us that a function grows at least as fast as another.

`Big-Theta Notation` : Let f and g be functions. We say that $f(x) = \Theta(g(x))$, read as " $f(x)$ is big-theta of $g(x)$ ", if and only if there exists constants C_1 , C_2 and k such that $C_1 g(x) < f(x) < C_2 g(x)$ whenever $x > k$. That is,

$f(x) = \Theta(g(x)) \leftrightarrow f(x) = \Omega(g(x)) \wedge f(x) = O(g(x))$.

⚠️ Sometimes writers are careless and write as if big-O notation has the same meaning as big-Theta.

📋 Big-Theta Estimates for Polynomials

Let $f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$, where $a_0, a_1, \ldots, a_n$ with $a_n \neq 0$. Then $f(x)$ is of order x^n or $f(x) = \Theta(x^n)$.

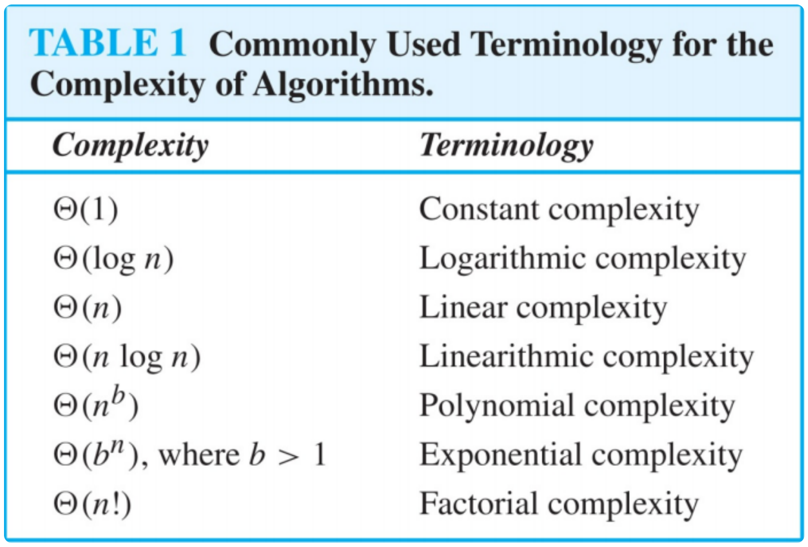


TABLE 1 Commonly Used Terminology for the Complexity of Algorithms.

Complexity	*Terminology*
$\Theta(1)$	Constant complexity
$\Theta(\log n)$	Logarithmic complexity
$\Theta(n)$	Linear complexity
$\Theta(n \log n)$	Linearithmic complexity
$\Theta(n^b)$	Polynomial complexity
$\Theta(b^n)$, where $b > 1$	Exponential complexity
$\Theta(n!)$	Factorial complexity

📋（不考，是课外积累的） Stirling's Approximation

$\ln n! \approx n \ln n - n \ (n \to +\infty)$, or more precisely $\lim\limits_{n\to+\infty} \dfrac{n!}{\sqrt{2\pi n}(\frac{n}{e})^n} = 1$.

Proof:

$$\begin{aligned}\sum_{k=1}^{n} \ln k &= \int_1^n \ln x \mathrm{d}x + \frac{1}{2}\ln n + \int_1^n \frac{g_1(x)}{x^2}\mathrm{d}x = n\ln n - n + \frac{1}{2}\ln n + 1 + \int_1^n \frac{g_1(x)}{x}\mathrm{d}x \\ &= n\ln n - n + \frac{1}{2}\ln n + 1 + \int_1^{+\infty} \frac{g_1(x)}{x}\mathrm{d}x + \frac{g_2(n)}{n} - \int_n^{+\infty} \frac{g_2(x)}{x^2}\mathrm{d}x \\ &= (n+\frac{1}{2})\ln n - n + \frac{1}{12n} + \frac{1}{2}\ln 2\pi + O(\frac{1}{n^2})\end{aligned}$$

3.3 How to Measure Complexity of Algorithms

An algorithm provides a satisfactory solution when the output is correct and efficient. The efficiency is implied by time complexity and space complexity. In this course, we focus on time complexity, and space complexity of algorithms is studied in later courses. We measure time complexity in terms of the number of operations an algorithm uses, such as comparisons and arithmetic operations (addition, multiplication, etc.). Big–O and big–Theta notations are applied.

We will focus on the `worst-case time complexity` of an algorithm. This provides an upper bound on the number of operations an algorithm uses to solve a problem with an input of a particular size.

⚠️ It is usually much more difficult to determine the `average case time complexity` of an algorithm, because we often have difficulty defining what is the average case.

🌰 e.g. Worst–Case Complexity of Linear Search

Determine the time complexity of the linear search algorithm.

Plain Text 复制代码

```
procedure linear search(x:integer, a1, a2, …,an: distinct integers)
i := 1
while (i ≤ n and x ≠ ai)
    i := i + 1
if i ≤ n then location := i
else location := 0
return location
{location is the subscript of the term that equals x, or is 0 if x is not found}
```

Solution:

Count the number of comparisons.

At each step two comparisons are made; $i \leq n$ and $x \neq ai$.

To end the loop, one comparison $i \leq n$ is made.

After the loop, one more $i \leq n$ comparison is made.

If $x = ai$, $2i + 1$ comparisons are used. If x is not on the list, $2n + 1$ comparisons are made and then an additional comparison is used to exit the loop. So, in the worst case, $2n + 2$ comparisons are made.

Hence, the complexity is $\Theta(n)$.