## Contents

1 Growth of functions 3
   1.1 Compare the growth rates of $f(x)$ and $g(x)$ .......................... 3
   1.2 Basic principles of growth rates ........................................... 3
   1.3 Examples ................................................................. 4
   1.4 Formal definitions ....................................................... 5

2 Analyzing algorithms’ complexity 5
   2.1 Definition of algorithm ................................................... 5
   2.2 Costs of instructions ..................................................... 6
It is particularly important to study the complexity, also known as *asymptotic performance*, in Computer Science. In the past decades, the Moore’s law has been observed in the computer hardware industry. Roughly speaking, the Moore’s law states the chip performance (computing ability, storage, *etc.*) doubles every 18 months.


With the development of the computers, people started to model the real-world problems with some mathematical models that can be solved by the computers.

- Internet routing: How can we more quickly find out a route from a computer to a server?
- Wireless internet: How can we support more people in a fixed-size area with limited number of access points?
- High-frequency trading: How can we more precisely predict the trend of the stock market to gain more profits?
- Marketing: How can we invest our limited budget to promote our new product better such that it is exposed to more people?

Those are only part of the real-life problems that I can imagine, and there are many more computation problems in our world. Obviously, some problems can be quickly solved by the computers due to its powerful computing abilities, but some problems need a lot of time even for powerful computers, and it became necessary to determine whether problems are *doable* in a practically acceptable amount of time.

The Moore’s law gave a good hint in setting up the boundary for ‘doable’ and ‘non-doable’ problems. Since the computing ability doubles every 18 years, as long as the ‘hardness’ of the problems grow more slowly than this speed, the time computers need to solve those problems will become smaller and smaller, and eventually in the future human beings will become able to solve the problem practically.
Therefore, it became particularly important in Computer Science to figure out whether the time and resources needed to solve a problem is ‘polynomially’ large or ‘exponentially’ large. If they grow polynomially with the scale of the problem (e.g., the number of the users in the problem), the advancement of the computer industry will definitely surpass the growth of the ‘hardness’, and therefore such problems are considered to be ‘solvable’. On the other hand, if they grow exponentially, human beings are not sure whether they can solve those problems even in the future since the computing ability also grows exponentially, and it is possible that we are not able to practically solve those problems even with a lot of time given to the development in the computer hardware industry.

1 Growth of functions

Often, the time or resources needed to solve a problem are represented with certain functions (e.g., time, memory consumption, and storage consumption required to finish executing the solution). Therefore, it is very important to know the growth rate of the functions. The growth rates of functions are often described with the following notations.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Reads as</th>
<th>(Informally) means</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(x)$ is $O(g(x))$</td>
<td>$f(x)$ is Big-Oh of $g(x)$</td>
<td>$f(x)$ is at most as fast as $g(x)$ (or slower)</td>
</tr>
<tr>
<td>$f(x)$ is $o(g(x))$</td>
<td>$f(x)$ is Little-Oh of $g(x)$</td>
<td>$f(x)$ is strictly slower than $g(x)$</td>
</tr>
<tr>
<td>$f(x)$ is $\Theta(g(x))$</td>
<td>$f(x)$ is Big-Theta of $g(x)$</td>
<td>$f(x)$ is exactly as fast as $g(x)$</td>
</tr>
<tr>
<td>$f(x)$ is $\Omega(g(x))$</td>
<td>$f(x)$ is Big-Omega of $g(x)$</td>
<td>$f(x)$ is at least as fast as $g(x)$ (or faster)</td>
</tr>
<tr>
<td>$f(x)$ is $\omega(g(x))$</td>
<td>$f(x)$ is Little-Omega of $g(x)$</td>
<td>$f(x)$ is strictly faster than $g(x)$</td>
</tr>
</tbody>
</table>

Because it is very hard to understand this concept from the definitions of those Big- and Little- notations, let’s see how to compare the growth rates first.

Note that the growth rates are meaningful for growing functions. Functions that are decreasing or negative are out of the scope when studying the growth rates.

1.1 Compare the growth rates of $f(x)$ and $g(x)$

It is quite simple to compare the growth rates.

$$\lim_{x \to +\infty} \frac{f(x)}{g(x)} = \begin{cases} +\infty & f(x) \text{ is strictly faster than } g(x) \\ 0 & f(x) \text{ is strictly slower than } g(x) \\ \text{a positive constant} & f(x) \text{ is exactly as fast as } g(x) \end{cases}$$

1.2 Basic principles of growth rates

However, sometimes we want to quickly compare the growth rates without calculating the above formula. Some principles can be used to make a quick decision.

Function classes The following table presents some function classes by the order of their growth rates (from the slowest to the fastest). Note that the list is not exhaustive.

The class difference is dominative when comparing the growth rates of two functions. Even the fastest function (if ever exists) in one class is slower than the slowest function (if ever exists) in the next class. For
Table 1: Function classes from the slowest to the fastest

<table>
<thead>
<tr>
<th>Class</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>10, 253, $3^n$</td>
</tr>
<tr>
<td>Log-logarithmic</td>
<td>$\log \log x$</td>
</tr>
<tr>
<td>Logarithmic</td>
<td>$\log x$</td>
</tr>
<tr>
<td>Poly-logarithmic</td>
<td>$3(\log x)^2 + \log x$</td>
</tr>
<tr>
<td>Polynomial</td>
<td>$x^2 + 3x + 1$</td>
</tr>
<tr>
<td>Exponential</td>
<td>$3^x, 10^{x/2}$</td>
</tr>
<tr>
<td>Factorial</td>
<td>$x!, (3x)!$</td>
</tr>
</tbody>
</table>

example, a polynomial function, no matter how fast it is, must be slower than any exponential function, no matter how slow the exponential function is.

The ‘fastest’ term in the function determines the growth rate. A function may contain many terms, e.g., $f(x) = x^4 + 4x^3 + 2^x + \log x$. Then, the term with the highest growth rate determines the growth rate of the entire function, therefore one only needs to compare the growth rates of the fastest terms.

Constants or constant factors are ignored. Because the constant terms or constant factors (i.e., coefficients) do not affect the limitation, they are ignored in comparing the growth rates. For example, $2^{10}x^3$ and $\frac{x^3}{1000}$ have exactly the same growth rates and they are equally fast because the coefficients, even though the difference is huge, are constants.

1.3 Examples

Now, we can easily compare the growth rates of two functions. Then, given a function, we are able to find out what other functions are faster, slower, or equally fast. For example,

- $f(x) = x^3 + 2x + \log x$
  - is strictly slower than: $x^4 + \log^3 x$, $2^x$, $x!$
  - is exactly as fast as: $21x^3 + 3x^2 + (\log x)^{10} + x \log \log x$
  - is strictly faster than: $3\log^3 x + \log x^2$, $2x^2 + 10x + 31$

Then, we can say

- $f(x)$ is
  - $O(x^4 + \log^3 x), O(2^x), O(x!), O(21x^3 + 3x^2 + (\log x)^{10} + x \log \log x)$
    * Because $f(x)$ is at most as fast as them.
  - $o(x^4 + \log^3 x), o(2^x), o(x!)$
    * Because $f(x)$ is strictly slower than them.
  - $\Theta(21x^3 + 3x^2 + (\log x)^{10} + x \log \log x)$
    * Because $f(x)$ is exactly as fast as them.
  - $\Omega(21x^3 + 3x^2 + (\log x)^{10} + x \log \log x), \Omega(3\log^3 x + \log x^2), \Omega(2x^2 + 10x + 31)$
Because \( f(x) \) is at least as fast as them.
- \( \omega(3 \log^3 x + \log x^2), \omega(2x^2 + 10x + 31) \)
- Because \( f(x) \) is strictly faster than them.

### 1.4 Formal definitions

With the understanding on how to find out the Big- and Little- notations given a function, we are ready to understand the formal definition of those notations. Note that the order of the \( \exists \) and \( \forall \) cannot be flipped.

**Big-Oh** \( f(x) \) is \( O(g(x)) \) if and only if

\[
\exists x_0 > 0, \exists c > 0, \forall x > x_0 > 0 : f(x) \leq cg(x)
\]

**Little-Oh** \( f(x) \) is \( o(g(x)) \) if and only if

\[
\forall c > 0, \exists x_0 > 0, \forall x > x_0 > 0 : f(x) < cg(x)
\]

**Big-Theta** \( f(x) \) is \( \Theta(g(x)) \) if and only if

\[
\exists x_0 > 0, \exists c_1 > 0, \exists c_2 > 0, \forall x > x_0 > 0 : c_1 g(x) \leq f(x) \leq c_2 g(x)
\]

**Big-Omega** \( f(x) \) is \( \Omega(g(x)) \) if and only if

\[
\exists x_0 > 0, \exists c > 0, \forall x > x_0 > 0 : f(x) \geq cg(x)
\]

**Little-Omega** \( f(x) \) is \( \omega(g(x)) \) if and only if

\[
\forall c > 0, \exists x_0 > 0, \forall x > x_0 > 0 : f(x) > cg(x)
\]

### 2 Analyzing algorithms’ complexity

#### 2.1 Definition of algorithm

An algorithm is a finite set of instructions for performing certain computation, and the purpose of the computation is to solve a given problem. For example,

**Problem** : Find the maximum value in a finite sequence of integers.

**Algorithm** :

1. Set the temporary maximum variable equal to the first integer in the sequence.
2. Compare the next integer in the sequence to the temporary variable. If it is larger than the temporary one, set the temporary maximum equal to the integer.
3. Repeat the previous step until the end.
4. Output the temporary variable as the maximum value in the sequence.
Then, the hardness of a given problem is often measured by the time complexity and storage complexity of the best-known algorithm to solve the problem. Nowadays in the big data era, the storage complexity is also very important, but in this course, we focus on the time complexity only for the learning purpose because the difference between analyzing time complexity and storage complexity is marginal.

2.2 Costs of instructions

Many different instructions are contained in algorithms. For example, given a sequence of integers $x_1, x_2, \ldots, x_n$, the following pseudo-code describes the aforementioned algorithm which finds the maximum value.

```
1: max ← x_1 # 1 assignment
2: for k ← 2 to n do # 1 addition to k and 1 comparison for k
3: if x_k > m then # 1 comparison
4: max ← x_k # 1 assignment
5: end if
6: end for
7: return max # 1 return
```

In each line, the number of involved instruction is presented. If we say the time cost of 1 assignment is $a$ ns, the cost of 1 addition is $b$ ns, the cost of 1 comparison is $c$ ns, and the cost of 1 return is $d$ ns (ns=nanoseconds), the total time cost in the above algorithm is

$$f(n) = a + (n - 1)(b + c) + (n - 1)(c + a) + d \quad (\text{ns})$$

where the first term $a$ corresponds to line 1, the second term $(n - 1)(b + c)$ corresponds to line 2, the third term $(n - 1)(c + a)$ corresponds to line 3 and line 4, and the last term $d$ corresponds to line 7.

If we analyze the growth rate of this function, we can see $f(n)$ is $\Theta(n)$, and $\Theta(n)$ is used to describe the time complexity of the algorithm because it tells that the time cost in finding the maximum value using the above algorithm grows linearly with the increase of $n$, the number of integers. Here, $n$ is also called as the size of the input.

Recall that the constants and constant coefficients are ignored in the growth rates of the functions. This applies to the time complexity of the algorithms as well. Let’s see the influence of those $a$, $b$, $c$, and $d$ in the complexity. Because they are constants irrelevant to the size of the input $n$, they do not contribute to the complexity of the algorithm. Therefore, for the sake of simplicity in the analysis, it is often assumed that the cost of every simple instruction is 1. At least in the scope of CS330, CS430, and CS535, there will be no complicated instructions whose cost cannot be assumed as 1.