Asymptotic Analysis: The Growth of Functions

Slides originally written by Prof. Yufei Tao and modified by Shiyuan Deng

Department of Computer Science and Engineering
Chinese University of Hong Kong

You have learned what it means by claiming that an algorithm has a worst-case running time $10 + 10 \log_2 n$, where n is the problem size.

In computer science, we rarely calculate the running time to such a detailed level. We typically ignore all the constants, but only worry about the dominating term. For example, instead of $10 + 10 \log_2 n$, we will keep only the $\log_2 n$ term.

In this tutorial, we will:

1. explain some reasons behind the "no-constant" principle;
2. review the notations $\mathbf{O}, \mathbf{\Omega}$, and $\mathbf{\Theta}$.

Why Not Constants?

Suppose that one algorithm has $5n$ atomic operations, while another algorithm $10n$. Which one is faster in practice?

The answer is: "it depends".

Not every atomic operation takes equally long in reality. For example, a comparison $a < b$ is typically faster than multiplication $a \cdot b$, which in turn is often faster than accessing a location in memory. Therefore, which algorithm is faster depends on the concrete operations they use.

Why Not Constants?

Suppose that Algorithm 1 runs in

$$n \cdot c_{mult} + 4 \cdot c_{mem}$$

time, where c_{mult} is the time of one multiplication, and c_{mem} the time of one memory access; Algorithm 2 runs in

$$9n \cdot c_{mult} + n \cdot c_{mem}$$

time. Again, which one is better depends on the specific values of c_{mult} and c_{mem}, which **vary from machine to machine**.

> However, in mathematics, we want to make **universal** conclusions that hold on **all** machines.

It is difficult (perhaps even impossible) to make any universal conclusion if you must take constants into account.

Why Not Constants?

Continuing from the previous slide, consider again two algorithms with costs $n \cdot c_{mult} + 4 \cdot c_{mem}$ and $9n \cdot c_{mult} + n \cdot c_{mem}$, respectively.

Here is a universal conclusion that we can make:

> Their costs differ by at most **some** constant factor.

To reach such a conclusion, none of the constants 4, 9, c_{mult}, and c_{mem} matters.

So, What *Does* Matter?

The growth of the running time with the problem size n.

We care about the efficiency of an algorithm when n is **large** (for small n, the efficiency is less of a concern, because even a slow algorithm would have acceptable performance).

So, What *Does* Matter?

Suppose that Algorithm 1 demands n atomic operations, while Algorithm 2 requires $10000 \cdot \log_2 n$.

For $n = 2^{30}$ (roughly 10^9), Algorithm 2 is faster by a factor of $\frac{n}{10000 \log_2 n} > 3579$. The factor continuously increases with n. When n tends to ∞, Algorithm 2 is infinitely faster.

Algorithm 2, therefore, is considered better than Algorithm 1 in computer science.

Art of Computer Science

Primary objective:

Minimize the growth of running time in solving a problem.

Next, we will review of the notations $\mathbf{O}, \boldsymbol{\Omega}$, and $\boldsymbol{\Theta}$.

Big-O

Let $f(n)$ and $g(n)$ be two functions of n.

We say that $f(n)$ **grows asymptotically no faster than** $g(n)$ if there is a constant $c_1 > 0$ such that

$$f(n) \quad \leq \quad c_1 \cdot g(n)$$

holds for all n at least a constant c_2.

We can denote this by $f(n) = O(g(n))$.

Example

Earlier, we say that an algorithm with running time $10000 \log_2 n$ is better than another one with running time n. Big-O captures this because:

$$\begin{aligned} 10000 \log_2 n &= O(n) \\ n &\neq O(10000 \log_2 n) \end{aligned}$$

An interesting fact:

$$\log_a n = O(\log_b n)$$

for any constants $a > 1$ and $b > 1$.

Because of the above, in computer science, we often omit constant logarithm bases in big-O. For example, instead of $O(\log_2 n)$, we will simply write $O(\log n)$.

- Essentially, this says that "you are welcome to put any constant base there; and it will be the same asymptotically".

Henceforth, we will describe the running time of an algorithm only in the asymptotical (i.e., big-O) form, which is also called the algorithm's **time complexity**.

For example, instead of saying that the running time of binary search is $f(n) = 10 + 10 \log_2 n$, we will say $f(n) = O(\log n)$, which captures the fastest-growing term in the running time. This is also binary search's time complexity.

Big-Ω

Let $f(n)$ and $g(n)$ be two functions of n.

If $g(n) = O(f(n))$, then we define:

$$f(n) = \Omega(g(n))$$

to indicate that $f(n)$ **grows asymptotically no slower than** $g(n)$.

The next slide gives an equivalent definition.

Big-Ω

Let $f(n)$ and $g(n)$ be two functions of n.

We say that $f(n)$ **grows asymptotically no slower than** $g(n)$ if there is a constant $c_1 > 0$ such that

$$f(n) \quad \geq \quad c_1 \cdot g(n)$$

holds for all n at least a constant c_2.

We can denote this by $f(n) = \Omega(g(n))$.

Big-Θ

Let $f(n)$ and $g(n)$ be two functions of n.

If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$, then we define:

$$f(n) \quad = \quad \Theta(g(n))$$

to indicate that $f(n)$ **grows asymptotically as fast as** $g(n)$.

Exercise 1

Verify all the following:

$$
\begin{aligned}
10000000 &= O(1) \\
100\sqrt{n} + 10n &= O(n) \\
1000n^{1.5} &= O(n^2) \\
(\log_2 n)^3 &= O(\sqrt{n}) \\
(\log_2 n)^{9999999999} &= O(n^{0.0000000001}) \\
n^{0.0000000001} &\neq O((\log_2 n)^{9999999999}) \\
n^{9999999999} &= O(2^n) \\
2^n &\neq O(n^{9999999999})
\end{aligned}
$$

Exercise 2

Verify all the following:

$$\begin{aligned}
\log_2 n &= \Omega(1) \\
0.001n &= \Omega(\sqrt{n}) \\
2n^2 &= \Omega(n^{1.5}) \\
n^{0.0000000001} &= \Omega((\log_2 n)^{9999999999}) \\
\frac{2^n}{1000000} &= \Omega(n^{9999999999})
\end{aligned}$$

Exercise 3

Verify the following:

$$\begin{aligned} 10000 + 30\log_2 n + 1.5\sqrt{n} &= \Theta(\sqrt{n}) \\ 10000 + 30\log_2 n + 1.5n^{0.5000001} &\neq \Theta(\sqrt{n}) \\ n^2 + 2n + 1 &= \Theta(n^2) \end{aligned}$$