Asymptotic Analysis: The Growth of Functions

Yufei Tao

Department of Computer Science and Engineering
Chinese University of Hong Kong

So far we have been analyzing the time of algorithms at a "fine-grained" level. For example, we characterized the worst-case time of binary search as at most $f(n) = 10 + 10 \log_2 n$, where n is the problem size (i.e., the number of integers in the input).

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

In this lecture, we will:

1. Shed light on the rationale behind this "one-term-only" principle.
2. Define a mathematically rigorous way to enforce the principle—the **asymptotic approach**.

Why Not Constants?

Let us start with a question. Suppose that one algorithm has $5n$ atomic operations, while another algorithm $10n$. Which one is faster in practice?

The answer is: "it depends!"

This is because 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?

In order to be perfectly precise, we should measure the time of an algorithm in the form of

$$n_1 \cdot c_1 + n_2 \cdot c_2 + n_3 \cdot c_3 + ...$$

where n_i ($i \geq 1$) is the number of times the algorithm performs the i-th type of atomic operations, and c_i is the duration of one such operation.

Besides making algorithm analysis more complicated, the above approach does not necessarily make the comparison of algorithms easier. The next slide gives an example.

Why Not Constants?

Suppose that Algorithm 1 runs in

$$1000n \cdot c_{mult} + 10n \cdot c_{mem}$$

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

$$10n \cdot c_{mult} + 100n \cdot c_{mem}$$

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

However, regardless of the machine, their costs always differ by at most a constant factor, as is a safe conclusion to make. By ignoring such constant factors, we consider the two algorithms equally fast.

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).

Suppose that Algorithm 1 demands n atomic operations, while Algorithm 2 requires $10000 \cdot \log_2 n$ (note that the constant 10000 was deliberately chosen to be big). 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. In other words, when n tends to ∞, Algorithm 2 is infinitely faster.

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

Art of Computer Science

Primary objective:

> Minimize the growth of running time in solving a problem.

Recall how we went from running time proportional to n down to $\log_2 n$ in the dictionary search problem—that is considered to be a breakthrough in algorithm design.

Next, we will learn a cool concept to formalize the above intuition, namely, permitting us to rigorously examine whether a function has a faster growth than another.

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 $10n$ is considered equally fast as one with time $5n$. Big-O captures this by having both of the following true:

$$10n = O(5n)$$
$$5n = O(10n)$$

Let us prove the first equality (the second left to you). This is easy: there are constants $c_1 = 2$ and $c_2 = 1$ such that

$$10n \le c_1 \cdot 5n$$

holds for all $n \ge c_2$.

Note that many constants will allow you to prove the same. Here are another two: $c_1 = 10$ and $c_2 = 100$.

Example

Earlier, we say that an algorithm with running time $10000 \log_2 n$ is considered better than another one with time n. Big-O captures this by having both of the following true:

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

The first equality is easy to prove: there are constants $c_1 = 1$ and $c_2 = 2^{20}$ such that

$$10000 \log_2 n \leq c_1 \cdot n$$

holds for all $n \geq c_2$.

Example

Earlier, we say that an algorithm with running time $10000 \log_2 n$ is considered better than another one with time n. Big-O captures this by having both of the following true:

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

Let us prove the second inequality by contradiction. Suppose that there are constants c_1 and c_2 such that

$$n \leq c_1 \cdot 10000 \log_2 n$$

holds for all $n \geq c_2$. The above can be rewritten as:

$$\frac{n}{\log_2 n} \leq c_1 \cdot 10000.$$

The left hand side tends to ∞ as n increases. Therefore, the inequality cannot hold for all $n \geq c_2$.

Example

Verify all the following:

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

An interesting fact:

$$\log_{b_1} n \quad = \quad O(\log_{b_2} n)$$

for any constants $b_1 > 1$ and $b_2 > 1$.

For example, let us verify $\log_2 n = O(\log_3 n)$.

Notice that

$$\log_3 n = \frac{\log_2 n}{\log_2 3} \Rightarrow \log_2 n = \log_2 3 \cdot \log_3 n.$$

Hence, we can set $c_1 = \log_2 3$ and $c_2 = 1$, which makes

$$\log_2 n \leq c_1 \log_3 n$$

hold for all $n \geq c_2$.

An interesting fact:

$$\log_{b_1} n = O(\log_{b_2} n)$$

for any constants $b_1 > 1$ and $b_2 > 1$.

Because of the above, in computer science, we omit all the 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**.

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 the 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

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

holds for all n at least a constant c_2.

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

Example

Verify all the following:

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

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:

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

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

Example

Verify the following:

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