Asymptotic Analysis

Introduction to Theoretical CS

Jason Zhang

October 2025

Code++ Officers

Outline

Introduction

Introduction to Theoretical CS

To be able to effectively study and implement high level concepts (e.g., AI), we must first be able to understand the *foundation* and core of computer science. For this reason, we must study theoretical computer science. This slideshow is equivalent to a lecture that would usually be given in college (for UVA, it would be a lecture from Data Structures and Algorithms 2 with some background info from DSA 1).

Time Complexity

Recall the definition of time complexity as given in AP Computer Science

Definition

We use $O(g(n))$ to describe how a function $f(n)$ grows when n grows. $g(n)$ does not have constant factors or lower order terms.

Example

Example

Determine O for $f(n) = 3n^2 - 9n + 10$.

Solution

Removing constant factors and lower order terms, we recieve $f(n)$ is $O(n^2)$.

Problem

There is a problem with this definition: it is far too informal for us to be able to analyze how complex algorithms may behave. We will look for better.

Formal Definitions

Big-O

We say that a function $f(n)$ is $O(g(n))$ if and only if there exists a constant $c > 0$ and $n_0 > 0$ such that for all $n > n_0$, $|f(n)| \leq cg(n)$.

The above is *formal* definition for big-O. The properties of the big-O developed from AP Computer Science **follows** from our definition. Notice that it isn't a part of our definition, we made no comment about constant factors or anything like that. Additionally, while there are absolute value bars around f, we will mostly work with positive, increasing only functions so this can be dropped for our case.

Equivalent Definition

Some people choose to use a limit definition. That is,

$$\lim_{n \to \infty} \sup \frac{|f(n)|}{g(n)} < \infty$$

We will not be using this definition, but you should know it.

Example

Let us consider the same example from before. We claim that $f(n) = 3n^2 - 9n + 10$ is $O(n^2)$.

Proof

Select $c = 100$ and $n_0 = 1$. We can see (trivially) that for all $n > n_0$, $3n^2 - 9n + 10 \leq 100n^2$.

Claim

We want to prove two properties:

- Constant factors don't matter.
- Lower order terms don't matter.

Let us first formalize these two statements:

- If $f(n)$ is $O(g(n))$, then for any positive constant $k > 0$, $f(n)$ is $O(kg(n))$.
- If f_1 is $O(g(n))$ and f_2 is $O(h(n))$, then $f_1 + f_2$ is $O(\max g(n), h(n))$.

Statement 1

Let us attack the first statement: If $f(n)$ is $O(g(n))$, then for any positive constant $k > 0$, $f(n)$ is $O(kg(n))$.

Proof

We know there is some c_0 such that past a certain threshold n_0, $f(n) \leq c_0 g(n)$. Now, we will prove $f(n)$ is $O(kg(n))$. Select $c = \frac{c_0}{k}$ and keep n_0 the same as before. Since $ckg(n) = c_0 g(n)$ and $f(n) \leq c_0 g(n)$, $f(n) \leq ckg(n)$ as desired.

Important Note

From our definition of big-O, it should be realized that O is a $\leq$ sign. NOT AN $=$ sign! We will develop that later. However, this means n is $O(n^2)$ (pick $c = 1$ and $n_0 = 1$). Hypothetically, you could write $O(100^{n^2})$ on virtually every single question on your big-O quiz and be correct theoretically.

Statement 2

The claim is: If f_1 is $O(g(n))$ and f_2 is $O(h(n))$, then $f_1 + f_2$ is $O(\max g(n), h(n))$.

An important thing to note is that WE ARE COMPUTER SCIENTISTS. NOT mathematicians. We do not need to be as formal as a mathematician (we could, but that would take more work).

Proof (via Hand-Waving)

I will use $\leq$ in place of O so the intuition becomes more clear. Assume without loss of generality that $g \leq h$. We know that $f_2 \leq h$ and $f_1 \leq g$. Thus $f_1 \leq h$. Now, assume the most powerful case for f_1 and f_2, that is $f_1, f_2 \approx h$. So $f_1 + f_2 \approx 2h$, but constant factors don't matter. So $f_1 + f_2$ is $O(\max g(n), h(n))$.

Two New Players

Definition

We say that $f(n)$ is $\Omega(g(n))$ if and only if $g(n)$ is $O(f(n))$.

We say that $f(n)$ is $\Theta(g(n))$ if and only if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

Note that we could go through the whole pick-constant definition with these two but again-we are lazy.

Developing the Analogy

If O is $\leq$, then Ω is $\geq$ and Θ is $=$. This is apparent from the definition.

Notice, your AP Computer Science teacher actually wanted Θ, NOT O. n is NOT $\Theta(n^2)$, it is simply $\Theta(n)$. However, technically writing $\Theta(2n)$ would still be fine, although weird.

Two More Players

There are two more players that we won't get into much. That is o (little o) and ω (little omega). These stand in for $<$ and $>$ respectively.

Best, Average, Worst Case

Usually, Ω, Θ, and O stand in for best, average, and worst case respectively. And it makes sense why, O is describing an upper bound and hence your worst case.

HOWEVER

They are **NOT** limited to these cases. I can just as well describe the worst case with Ω if I only know a lower bound about it.

Methods of Analysis

Introduction

There are many different methods to analyze time complexity such as using induction, substitution, etc. We will be looking at perhaps one of the most famous, the *Master Theorem*.

Theoretical Statement

Theorem

Suppose $T(n) = aT(\frac{n}{b}) + f(n)$ for $a \geq 1$, $b > 1$ and the division here is either floor or ceiling division (and obviously there will be a base case). Then one of three cases may apply:

1. If $f(n)$ is $O(n^c)$ and $\log_b(a) > c$ then $T(n)$ is $\Theta(n^{\log_b(a)})$.
2. If $f(n)$ is $\Theta(n^c)$ and $\log_b(a) = c$ then $T(n)$ is $\Theta(n^{\log_b(a)} \log n)$. Note that this case can be generalized a bit.
3. If $f(n)$ is $\Omega(n^c)$ and $\log_b(a) < c$ then $T(n)$ is $\Theta(f(n))$. Note that a special condition must be true for this but it almost always is so we ignore it.

Application

Let's say we want to find the time complexity of an algorithm described by the relation $T(n) = 4T(\frac{n}{2}) + n^2$. Calculate $\log_2(4) = 2$. Notice that $f(n) = n^2$ is $\Theta(n^2)$, so we are in case two. Thus,

$$T(n) \in \Theta(n^2 \log n).$$

Exercise

Find the time complexity of Merge Sort.

Solution

Let us define the recurrence relationship. Realize that Merge Sort splits the current segment into two groups, each of half size. When recombining/zipping, it takes $2 \times \frac{n}{2} = n$ operations. Thus the relation is

$$T(n) = 2T(\frac{n}{2}) + n.$$

Calculate $\log_2(2) = 1$. We are in case two again, so the answer is $\Theta(n \log n)$.

Preliminary

A full proof would be extremely laborious. So, we will be hand-waving again.

Without f

Suppose $f(n) = 0$. So $f(n) \in \Theta(1)$. Suppose $T(1)$ is some base case. Realize that

$$T(n) = aT(\frac{n}{b}) = a^2 T(\frac{n}{b^2}) = a^3 T(\frac{n}{b^3}) = \cdots$$

This ends when $\frac{n}{b^k}$ is 1 for which $k = \log_b(n)$.

Without f

Thus $T(n) = a^k T(1) = a^{\log_b(n)} T(1)$. Changing base to base-$a$ we get $a^{\frac{\log_a(n)}{\log_a(b)}} T(1) = \left(a^{\log_a(n)}\right)^{\frac{1}{\log_a(b)}} T(1) = n^{\log_b(a)}$ as desired.

With f

If f is nonzero, we will have to modify our intuition a little bit.

1. If f is smaller than the recursive part (meaning-O, $c < \log_b(a)$, then the recursive part dominates and absorbs f as per our definition of Θ. Thus, $\Theta(n^{\log_b(a)})$.
2. If f is equal to the recursive part (Θ, $c = \log_b(a)$), then they combine. A logarithmic factor is introduced, but the reason behind that is beyond the scope of this lecture. We recieve $\Theta(n^{\log_b(a)} \log n)$.
3. If f is bigger than the recursive parth (Ω, $c > \log_b(a)$), then f dominates and thus $\Theta(f(n))$.

Conclusion

Conclusion

While most are still going to stick to the AP Computer Science definition, it is important to know and have an understanding of the machinery behind the tools you use. Learning this is the gateway into higher level nontrivial concepts.