Time Complexity and Big-Oh

COMP221 - Suhas Arehalli

1 The RAM Model

- Our Goal: *Formalizing* what we know about algorithms
  - We can use heuristics and tricks to get at Big-Os — that's what we did in COMP128!
  - Here we want to build up the big-O & time complexity machinery we use from the basics!
- Our First Step: Building the RAM (Random Access Machine) Model
  - This is a *simplification* of a real computer.
  - the simplicity allows us to do analyses that are relatively language and hardware independent!
  - This model says that
    1. Simple operations (math, comparisons, local assignments, etc.) take **1 time step**.
    2. loops and method calls are *composite* operations. **We have to break them down into a sequence of simpler operations**.
    3. Memory accesses are always **1 time step**.
  - Remember that these are technically *wrong*, if we're talking about real computers
    1. Additions and Multiplications are both simple operations under RAM, but multiplications, in practice, take longer on real CPUs!
    2. Loops and Method calls you write in a high-level programming language are often optimized by compilers in unintuitive ways!
    3. In modern computers, memory access speeds are fairly complicated due to caching and paging! Think about temporal and spatial locality and cache hierarchies if you've taken COMP240!
  - Despite being wrong, this model is very useful in practice![1]
- Example:

  Consider a Linear Search algorithm (Alg. 1).

  If we run LinearSearch([3, -1, 2, 12, 6], 2), we can use the RAM model to count steps:

  1. $index \leftarrow 1$

[1] This is an oblique reference to a famous quote by statistician George E.P. Box: "All models are wrong, but some are useful." The quote itself first appeared Box (1979; *Robustness in the strategy of scientific model building*), but the idea behind it appeared in Box (1976; *Science and Statistics*). A big and important idea!

Algorithm 1 A simple linear search algorithm

```
function LinearSearch(Array $A$, Target $e$)
    for $index \leftarrow 1$ to $N$ do
        $x \leftarrow A[index]$
        if $e == x$ then
            return index
        end if
    end for
    return NULL
end function
```

2. $x \leftarrow 3$
3. $x == e$?. False!
4. $index \leftarrow 2$
5. $x \leftarrow -1$
6. $x == e$?. False!
7. $index \leftarrow 3$
8. $x \leftarrow 2$
9. $x == e$?. True!
10. return index (2!)

So for this instance of the problem, our algorithm solves it in 10 time steps.

- This is still strange. We want to think about *algorithms* in general, not just their behavior on particular *instances* of the problem they solve.
  - **The issue?** We can only count the number of steps on a particular instance, and it's unreasonable to think about every possible instance.
  - **Our Solution?** We will think in terms of *Best-Case*, *Worst-Case*, or *Average-Case* time complexities for problems of a particular size.
  - We have to consider problem size because larger problems (i.e., a larger array to search through) will take longer to solve, but some array/target pairs take longer to find than others, even if both arrays have length n!
- In general, we will case about the worst-case analysis.
  - Why? **We want to prepare for the worst!** Terrible algorithms may look good if we only look at their best-case behavior.
  - Example:

    Consider Alg. 2: In the best-case, this will run in constant time no matter the array! And, we can easily show that this algorithm will always return a correct value (If we return, we're in an if statement that tells us i is the correct return value!)! However, we **do not** want to consider this a good algorithm, right? [2] What happens when the element is not actually in the array?

[2] Advanced Note: If we want to be really precise, we can say that the algorithm, in the worst case, has an

Algorithm 2 A not-so-good search algorithm

```
function RandomSearch*(Array $A[1 \dots n]$, Target $e$)
    while True do
        Let $i \in \{1, \dots, n\}$ selected uniformly at random
        if A[i] = e then
            return i
        end if
    end while
end function
```

2 Big-O(h)

- With the RAM model, we can start to construct growth functions
  - A function $f(n)$ that tells us how many time steps an algorithm takes to execute for a problem of size n in the worst/best/average-case.
- But growth functions are too precise, and difficult to work with.
- **Our goal:** Be *lazy* (avoid dealing the with precision of growth functions), but avoid being *sloppy*
  - We want to make sure we preserve distinctions we care about, and avoid being bogged down by details we don't care about.
  - To do this, we'll construct a formalism (Big-Oh) that preserves the things we care about.
- What do we care about?
  1. Upper and lower bounds in terms of *nice* functions
  2. Behavior "in the limit" (as $n \to \infty$)
  3. Greater than linear scale (i.e., making our time steps k times as fast)
- This leads directly to Big-Oh notation

  Def 1 (Big-O)**.** *$f(n) \in O(g(n))$ if and only if $\exists c > 0, n_0$ such that $\forall n \geq n_0$,*

  $$f(n) \leq c \cdot g(n)$$

  Def 2 (Big-Ω)**.** *$f(n) \in \Omega(g(n))$ if and only if $\exists c > 0, n_0$ such that $\forall n \geq n_0$,*

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

unbounded runtime — it's not guaranteed to terminate! The probability that we do not terminate after k iterations (if the element is in the array) is $(\frac{n-1}{n})^k$, which will approach 0 as $n \to \infty$, but will never reach 0! Thus we are not guaranteed to terminate, but will terminate with probability 1. Similarly, we can analyze the *expected* number of steps it will take to find an element — the *average-case* analysis. If X is the number to steps to return a value, the expected number of steps, $E[X]$, can be written out recursively as $E[X] = \frac{1}{n}(1) + \frac{n-1}{n}(1 + E[X])$ which, if we solve for $E[X]$, gets us $E[X] = n$. Thus, on average, thisRandomSearch has the same time complexity as LinearSearch, just with higher highs (maybe we guess right on the first try!) and lower lows (maybe we never terminate...). However, if we want reliability, it seems like worst-case analysis is for us!

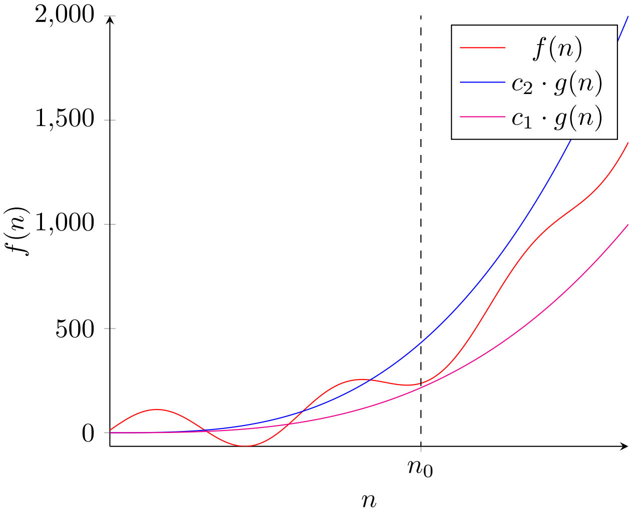

Figure 1: A visualization of our Big-Oh relationships. A messy growth function $f(n)$ is bounded from above by $c_2 \cdot g(n)$ for all $n \geq n_0$ and is thus $O(g(n))$. It is also bounded from below by $c_1 \cdot g(n)$ for $n \geq n_0$, and it thus $\Omega(g(n))$. As a result of both, it's $f(n) \in \Theta(g(n))$.

Def 3 (Big-Θ)**.** *$f(n) \in \Theta(g(n))$ if and only $f(n) \in O(g(n))$ and $f(g(n)) \in \Omega(n)$.*

- **Note:** The definition of Big-Θ here differs from one you might see in some textbooks, which looks like this

  Def 4 (Big-Θ, alternative)**.** *$f(n) \in \Theta(g(n))$ if and only if $\exists c_1 > 0, c_2 > 0, n_0$ such that $\forall n \geq n_0$,*

  $$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

  With a small trick, you can prove that both definitions are equivalent (remember when proving things are equivalent, you need to show that assuming either definition is true will let you show the other One direction is straightforward, the other requires the trick!).

- Think of Big-O saying that some multiple of g is an upper bound of f for sufficiently large n. Big-Ω is the same, but with a lower bound!

- I said that we wanted to only talk about nice functions. Since Big-Oh notation lets us talk in terms of $g(n)$s rather than $f(n)$s, we should try and pick nice functions as our $g(n)s$ when we prove Big-Ohs.

  - What are nice functions? From fast-growing to slow, we have:

    $$n! \gg 2^n \gg n^3 \gg n^2 \gg n \log n \gg n \gg \log n \gg 1$$

2.1 Example: InsertionSort

Consider the insertion sort algorithm in Alg. 3.

Algorithm 3 An insertion sort algorithm.

```
function InsertionSort(Array A)
    for $i \leftarrow 2$ to $N$ do
        $j \leftarrow i$
        $x \leftarrow A[j]$
        $y \leftarrow A[j-1]$
        while $j > 1$ and $x < y$ do
            $A[j] \leftarrow y$
            $A[j-1] \leftarrow x$
            $j \leftarrow j-1$
            $x \leftarrow A[j]$
            $y \leftarrow A[j-1]$
        end while
    end for
end function
```

Step through the algorithm to help you understand how this algorithm works. We'll prove it's correctness later.

Now to building a growth function: There are 5 times steps worth of code that execute within the while loop, and 3 time steps worth of code (two comparisons and a logical and!) to check the while loop's condition. So that is 8 steps per loop.[3]

How many times will the while loop loop? In general *we don't know!*. But we do know, in the *worst-case*, it will run $i-1$ times (consider the case where the ith element is smaller than all elements to it's left. Finally, before the while loop, but within the for loop, we have 3 time steps worth of operations.

That means that the ith iteration of the for loop will take $8(i-1)+3$ time steps. So the full algorithm will take...

$$\begin{aligned} f(n) = \sum_{i=2}^{n}(8(i-1)+3) &= \sum_{i=2}^{n}(8i-5) \\ &= 8\sum_{i=2}^{n} i - \sum_{i=2}^{n} 5 \\ &= 8(\frac{n(n+1)}{2} - 1) - 5(n-1) \\ &= 4n^2 + 4n - 8 - 5n + 5 \\ &= 4n^2 - n - 3 \end{aligned}$$

And so we have our growth function for insertion sort under our RAM model!

[3] If you disagree on my counts here, that's fine! How you count depends on what you consider *basic operations*, but what we can hopefully agree on is that each segment has a *constant* number of operations — what constant, in the end, doesn't matter, as you'll show eventually!

Now, your COMP128 intuitions should tell you that this algorithm should be $O(n^2)$. In fact, it should be $\Theta(n^2)$ (often what we wanted to talk about when we said Big-O in COMP128 was actually Big-Θ!). Lets prove that to be true.

Statement 1. $f(n) = 4n^2 - n - 3 \in \Theta(n^2)$

To prove that insertion sort is $O(n^2)$, we need to show $f(n) \in O(n^2)$, which, by definition, is equivalent to showing that for some c, n_0, for all $n > n_0$,

$$\begin{aligned} f(n) &\leq c \cdot g(n) \\ 4n^2 - n - 3 &\leq cn^2 \\ 0 &\leq (c-4)n^2 + n + 3 \end{aligned}$$

Now for a small trick: Since we only care about inequalities, we can swap this complex quadratic function for a simpler one! Since

$$(c-4)n^2 + n + 3 \geq (c-4)n^2 + n$$

We can show

$$\begin{aligned} (c-4)n^2 + n &\geq 0 \\ (c-4)n + 1 &\geq 0 \\ (c-4)n &\geq -1 \end{aligned}$$

assuming $n > 0$ (i.e., choose $n_0 > 0$, say, $n_0 = 1$), If we choose $c \geq 4$ (say, $c = 5$), we can convince ourselves that $(n-4)n^2 + n \geq 0$.

Now, to put the trick into place: consider the following inequalities:

$$(c-4)n^2 + n + 3 \geq (c-4)n^2 + n \geq 0$$

We've shown the first half (since adding 3 can only make something larger), and we saw earlier that choosing $c = 5$, $n_0 = 1$ allows us to show the second. Thus, cutting out the middleman, we show

$$(c-4)n^2 + n + 3 \geq 0$$

Which we showed above was equivalent to $f(n) \in O(n^2)$, by definition. Thus, insertion sort is $O(n^2)$

To show $f(n) \in \Omega(n^2)$, we can do something similar. Plugging in f and g into the definition of Big-Ω, we find that we need to show that for some $c > 0$ and n_0, for all $n \geq n_0$

$$\begin{aligned} f(n) &\geq c \cdot g(n) \\ 4n^2 - n - 3 &\geq cn^2 \\ 0 &\geq (c-4)n^2 + n + 3 \end{aligned}$$

Our intuition here should be that we should pick a $c < 4$ (say, 3) here, so the n^2 term is negative and we can get this polynomial below 0. We can also apply a variation on the trick used for Big-O: if $n \geq 1$, then $3n \geq 3$! As a result, we can set up the series of inequalities:

$$\begin{aligned}
0 &\geq (c-4)n^2 + n + 3n \geq (c-4)n^2 + n + 3 \\
0 &\geq (c-4)n^2 + 4n \geq (c-4)n^2 + n + 3
\end{aligned}$$

So we only need to show that...

$$\begin{aligned}
0 &\geq -n^2 + 4n \\
0 &\geq -n + 4 \\
n &\geq 4
\end{aligned}$$

This is true if we choose $n_0 \geq 4$, thus with $c = 3$, $n_0 = 4$, $f(n) \in \Omega(n^2)$!

Finally, since $f(n) \in O(n^2)$ and $f(n) \in \Omega(n^2)$, $f(n) \in \Theta(n^2)$ by (my) definition!

To write out a formal proof for this, you should do the scratch work I've just shown you first, and then take some time to rewrite: start with your choice of c and n_0 and demonstrate that the inequalities hold, knowing all of the values of the variables. It would look something like this:

Proof. We will first show that $f(n) \in O(n^2)$, and then that $f(n) \in \Omega(n^2)$.

To see that $f(n) \in O(n^2)$, consider $c = 5$ and $n_0 = 1$. Then observe that, for $n \geq n_0 = 1$

$$f(n) = 4n^2 - n - 3 \leq 4n^2 \leq 5n^2 = cn^2$$

And so, by definition, $f(n) \in O(n^2)$.

To see that $f(n) \in \Omega(n^2)$, consider $c = 3$ and $n_0 = 4$. First observe that for $n \geq 4$ that

$$f(n) = 4n^2 - n - 3 \geq 4n^2 - 4n$$

and then note that we can reduce $4n^2 - 4n \geq 3n^2$ for $n \geq 4$ as follows

$$\begin{aligned}
4n^2 - 4n &\geq 3n^2 \\
4n - 4 &\geq 3n \\
n &\geq 4
\end{aligned}$$

Which is true by assumption. Thus for $n \geq 4$ we have that

$$f(n) \geq 4n^2 - 4n \geq 3n^2 = cn^2$$

and thus $f(n) \in \Omega(n^2)$ by definition. Since $f(n) \in O(n^2)$ and $f(n) \in \Omega(n^2)$, then $f(n) \in \Theta(n^2)$ by definition. □

2.2 Closing Notes

The initial process is long, and often involves algebraic tricks for manipulating inequalities. Regardless, you will be expected to know how to prove big-O(h) time complexities from the c, n_0-definitions using the techniques we've seen here. We can, in fact, generalize some inequality tricks we've seen above:

1. If $k \geq 0$, $f(n) + k \geq f(n)$ for any function f.
2. If $n \geq 1$, $f(n) + kn \geq f(n) + k$ for any function f.

These should hopefully be intuitive, san the formal notation — adding adding numbers that are positive makes things larger!

And don't forget some basic inequality manipulation gotchas around multiplying both sides by things:

1. If $k \geq 0$, then $f(n) \geq g(n)$ iff $kf(n) \geq kg(n)$ (you can multiple both sides of an inequality by $k \geq 0$ and the two statements are equivalent).
2. If $k < 0$, then $f(n) \geq g(n)$ iff $kf(n) \leq kg(n)$ (if you multiply both sides of an inequality by $k < 0$, you must flip the inequality to preserve equivalence!).

2.3 Bonus: Big-Oh with Limits

In practice these Big-Oh problems can become easier if we pull out more powerful mathematical tools like the limit.

Def 5 (Big-O (Limit))**.** *$f(n) \in O(g(n))$ if and only if*

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

Def 6 (Big-Ω (Limit))**.** *$f(n) \in O(g(n))$ if and only if*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$$

Your intuitions should be as follows:

- If f grows faster than g, the ratio of f to g will grow without bound as $n \to \infty$. By this definition, $f(n) \in O(g(n))$ when this **doesn't** happen (i.e., when f grows roughly the same speed or slower than g!).
- Similarly, if g grows faster than f, the ratio of f to g will tend toward 0 (the denominator of this fraction gets larger and larger!). If $f(n) \in \Omega(g(n))$ under this definition, this cannot be true, and thus f grows as fast or faster than g.

2.3.1 Example

$$\begin{aligned}
f(n) &= 2n^4 - 3n^2 + 7 \\
g(n) &= n^5
\end{aligned}$$

Under the c, n_0-definition, we'd have to do some sneaky algebraic tricks or do some fancy reasoning about the behavior of quintic polynomials. But with limits, we simply evaluate

$$\begin{aligned}
\lim_{n\to\infty} \frac{f(n)}{g(n)} &= \lim_{n\to\infty} \frac{2n^4 - 3n^2 + 7}{n^5} \\
&= \lim_{n\to\infty} (\frac{2}{n} - \frac{3}{n^3} + \frac{7}{n^5}) \\
&= \lim_{n\to\infty} \frac{2}{n} - \lim_{n\to\infty} \frac{3}{n^3} + \lim_{n\to\infty} \frac{7}{n^5} \\
&= 0 - 0 + 0 \\
&= 0
\end{aligned}$$

Since $0 < \infty$, $f(n) \in O(n^5)$. But $0 \not> 0$, so $f(n) \notin \Omega(n^5)$, and thus $f(n) \notin \Theta(n^5)$.

Note: Limits are not a part of the formal prerequisites of this course, and so I'll never ask you to use the limit definition (though I **will** ask you to explicitly use the c, n_0-definition!). There may be some points later in the course where I'll simply ask you to prove a particular growth function has a certain big-$O/\Omega/\Theta$ time complexity, and in those cases I'll allow the limit definition.

Note 2: Proving that these definitions are equivalent to the others is unfortunately outside of the scope of the course. Aspiring mathematicians can take a shot at it after taking something like Real Analysis!