Growth of Functions
(CS 1800)
Introducing Big-O and Friends

• How do we decide whether a difference in running time matters — is $n^3$ significantly better than $n^2$?

• How do we compare running times that seem very different — $n!$ versus $2^n$?

• Today we’ll cover big-O, big-Ω, and big-Θ and the conventions by which computer scientists describe running times.
Asymptotic Growth

• We will generally only be concerned with asymptotic running times — what happens when input size $N$ gets large.

• We want to ignore multiplication by constant factors.

  • Whether individual operations take 2 or 3 processor cycles isn’t something we want to worry about.

• We’ll show three ways of talking about function growth — an upper bound ($O$), a lower bound ($\Omega$), and a tight bound ($\Theta$)
A Fast Growth Rate Results in a Slow Algorithm

- Suppose each x-axis is input size, and each y-axis is number of operations the algorithm must perform. Which graph do we want for our algorithm?

- “Fast growth rate” may sound good to an economist but isn’t desirable for a running time
Big-O: A Definition

- Let \( f \) be a function of the natural numbers.
- \( f = O(g(n)) \) if, for some \( c \) and \( n_0 \), \( f(n) \leq cg(n) \) for all \( n \geq n_0 \).
- Translation: at some point, \( g(n) \) will always grow faster than \( f(n) \), ignoring constants.

\[ f(n) = O(g(n)) \]

Notice \( g(n) \) here would pass \( f(n) \) eventually regardless of the slope of \( f(n) \).
Functions With the Same Growth Rate Are Big-O of Each Other

- If two functions are both essentially the same growth rate (e.g. both linear), they will be big-O of each other.

\[ g(n) = O(f(n)) \quad \text{mult by } c \quad f(n) = O(g(n)) \]
Big-O is an Upper Bound

- Here, $g(n) = O(f(n))$ — they are not the same growth rate, but $f(n)$ is always bigger after $n_0$ ($c=1, n_0 = 0$)
Big-O is the “≤” of Function Relations

• There are asymptotic operators we’ll cover that are analogous to each of ≤, ≥, =, <, and >

• Of these, Big-O is most similar to ≤.
  
  • It holds when two growth rates are essentially equal: N = O(2N). (Both linear)
  
  • It holds when the first growth rate is asymptotically less than the second: N = O(2^N). (Linear vs exponential)

• It is the most commonly used of the bounds because with algorithms, we usually want an upper bound on the worst case running time.

  • “The worst case running time of my algorithm is O(n^2)” — no worse than n^2 and maybe better. (If it's not better, no need to tell anyone...)
\[ f = O(h) \text{ and } g = O(h) \quad \text{implies } f + g = O(h) \]

- For example, \( f(n) = n^2 \) and \( g(n) = n \) are both \( O(n^2) \), so \( f(n) + g(n) = n^2 + n \) is \( O(n^2) \)

- “A bound that works for each term works for the sum”

- Proof: if \( n \) is larger than both \( f \) and \( g \)’s \( n_0 \), then \( f \leq c_1 h \) and \( g \leq c_2 h \), so \( f + g \leq (c_1 + c_2)h \) and \( (c_1 + c_2) \) is the constant required by the definition of big-O.

- This results in a golden rule for algorithm analysis:

  Executing one algorithm, then another results in a running time that is big-O of the max of the two running times.
A Polynomial of Degree \( d \) is \( O(n^d) \)

• Given a polynomial \( a_0n^d + a_1n^{d-1} + \ldots + a_d \), note that each term is \( O(n^d) \)

  • Each term \( a_jn^j \) is at most \( |a_j|n^d \) for \( n \geq 1 \), so \( |a_j| \) is the necessary constant in each case

  • In \( 2n^2 + 5n + 7 \), \( 5n \) is \( O(n^2) \) because \( 5n \leq 5n^2 \)

• So by the result on the previous slide, summing the terms results in a function that is still \( O(n^d) \).

So always drop lower-order terms when describing a polynomial running time: \( O(N^2) \), not \( O(N^2 + N + 1) \).
Increasing the Degree Increases the Growth Rate

- $n$ is $O(n)$, but also $O(n^2)$: both are upper bounds, but the first is tighter

- $n^2$ is not $O(n)$: no constant could make $n$ consistently bigger
Every Logarithm Grows Slower Than Every Polynomial

- Logarithmic growth is visibly slower than linear

- For every $b > 1$ and every $x > 0$, $\log_b n = O(n^x)$.

- All logarithms are within constant factors of each other: $\log_b n = (\log_c n) / (\log_c b) = \text{a constant times } \log_c n$, for all bases $b$ & $c$

- So we will often talk about $O(\log n)$ without specifying a base
  - Though occasionally we will speak of $\lg n$ (base 2) and $\ln n$ (base e)
Every Exponential Grows Faster Than Every Polynomial

- For every $r > 1$ and every $d > 0$, $n^d = O(r^n)$
- $\lim_{n \to \infty} n^d/r^n = 0$ for $r > 1$
- Unlike logs, exponentials with different bases have different big-O: $3^n$ is not $O(2^n)$, since $(3/2)^n$ isn’t a constant
Exponential Time is Much Worse than Polynomial Time

- Time to process $N$ inputs at 1 million instructions per second with the given running times (p. 34 of Tardos and Kleinberg, *Algorithm Design*)

<table>
<thead>
<tr>
<th>$N$</th>
<th>$n$</th>
<th>$n^3$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>$&lt;1s$</td>
<td>$&lt;1s$</td>
<td>$&lt;1s$</td>
</tr>
<tr>
<td>100</td>
<td>$&lt;1s$</td>
<td>$&lt;1s$</td>
<td>$10^{17}$ years</td>
</tr>
<tr>
<td>1000</td>
<td>$&lt;1s$</td>
<td>18min</td>
<td>$&gt;10^{25}$ years</td>
</tr>
<tr>
<td>10000</td>
<td>$&lt;1s$</td>
<td>12 days</td>
<td>$&gt;10^{25}$ years</td>
</tr>
</tbody>
</table>
An Algorithm is Considered Tractable if it is Polynomial Time

• Where polynomial time means, $O(n^d)$ for some $d$

• Note that this includes logarithmic time, because logarithmic algorithms are $O(n^d)$ too

• Likewise, constant time
A Warning About Equality

• It is conventional to say \( f(n) = n = O(n) \), but the \( = \) in red does not function the way you expect an equal sign to.

• Trying to use it like an equal sign can lead you to absurd conclusions like \( f(n) = n = O(n) = O(n^2) = n^2 \)

• A less misleading notation is to say \( f(n) \in O(n) \) — “\( f(n) \) belongs to this category of functions” — but this just isn’t as common

• Just remember \( = \) means “is” for big-O, e.g. \( f(n) \) “is” \( O(n) \), but a cat “is” an animal while not all animals are cats
Review of Big-O So Far: Practice Questions

- Identify whether \( f(N) = O(g(N)) \), \( g(N) = O(f(N)) \), both, or neither.

1. \( f(N) = 2N, \quad g(N) = 3N \)
2. \( f(N) = N, \quad g(N) = 2^N \)
3. \( f(N) = N^2, \quad g(N) = N^2 + N \)
4. \( f(N) = N^{0.0001}, \quad g(N) = \log N \)
5. \( f(N) = N^{100}, \quad g(N) = 1.1^N \)
6. \( f(N) = N \log N, \quad g(N) = N \)
7. \( f(N) = \sin N, \quad g(N) = \cos N \)
Review of Big-O So Far: Practice Questions

- Identify whether \( f(n) = O(g(n)) \), \( g(n) = O(f(n)) \), both, or neither.

1. \( f(N) = 2N \), \( g(N) = 3N \) \( \text{both!} \)
2. \( f(N) = N \), \( g(N) = 2^N \) \( f(N) = O(g(N)) \)
3. \( f(N) = N^2 \), \( g(N) = N^2 + N \) \( \text{both!} \)
4. \( f(N) = N^{0.0001} \), \( g(N) = \log N \) \( g(N) = O(f(N)) \)
5. \( f(N) = N^{100} \), \( g(N) = 1.1^N \) \( f(N) = O(g(N)) \)
6. \( f(N) = N \log N \), \( g(N) = N \) \( g(N) = O(f(N)) \)
7. \( f(N) = \sin N \), \( g(N) = \cos N \) \( \text{neither} \)
Lower Bounds: $\Omega$

- Sometimes we may want to say, “This algorithm must take \textbf{at least} this much time”
  - Either to claim an algorithm isn’t that great, or that a running time can’t be improved further.

- Just as big-O serves as an upper bound on the running time, big-$\Omega$ serves as a lower bound, the $\geq$ to big-O’s $\leq$

- The definition is identical to big-O with one inequality reversed

- $f(n) = \Omega(g(n))$ if, for some $c$ and $n_0$, $f(n) \geq cg(n)$ for all $n \geq n_0$. 
g is $\Omega(f)$ iff f is $O(g)$

- Checking the definitions, for $n \geq n_0$:
  - $g \geq 1^*f(n)$ so $g = \Omega(f)$
  - $f \leq 1^*g(n)$ so $f = O(g)$

The rule in the slide title makes sense if big-O is like $\leq$ and big-$\Omega$ is like $\geq$
Lower Bounds Help Us Know if We’re Optimal

• Consider an algorithm for finding the max of an array by iterating down it and keeping the biggest value. This is $O(N)$, but can we improve further?

• Thinking about how max works, it can’t be done in fewer than $N$ steps on an unsorted array — if we fail to look at a number, that might be the max.

• So we argue that this problem requires $\Omega(N)$ steps. Now we know an $O(N)$ time can’t be improved on.
Tight Bounds: $\Theta$

- If $f = O(g)$ and $f = \Omega(g)$, then $f = \Theta(g)$.
  - $f$ can be “sandwiched” between $c_1g(n)$ and $c_2g(n)$
- Unlike big-$O$, $\Theta$ implies that you are giving the actual (rough) growth rate instead of an upper bound.
  - $n^d = O(r^n)$ for all $d$ and $r$, but $n^d \neq \Theta(r^n)$ … polynomial times are very different from exponential
- This is the asymptotic bound analogous to “$=$” — growth rates must be effectively the same for one to be big-$\Theta$ of the other
Other Bounds: Little-o, Little-ω

• If we want to say one growth rate is strictly faster or slower than another, we use little-o and little-ω:
  
  • \( n = o(n^2) \)
  
  • \( n = \omega(\log n) \)
  
• These are analogous to < and > for growth rates.

• The technical definition of little-o is \( f(n) = o(g(n)) \) if

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.
\]

• Similarly, \( f(n) = \omega(g(n)) \) if the limit is infinite.

• Alternately, one can replace the \( \leq \) and \( \geq \) in the definitions of big-O and big-Ω with < and > to get definitions of \( o \) and \( \omega \).
Other Bounds Practice

- Of $o$, $\omega$, $O$, $\Theta$, and $\Omega$, which relations hold between $f(n)$ and $g(n)$ if …
  - $f(n) = n^2$, $g(n) = n^2 + n$
  - $f(n) = 2^n$, $g(n) = 3^n$
  - $f(n) = n^2$, $g(n) = n \log n$
Other Bounds Practice

- Of $o$, $\omega$, $O$, $\Theta$, and $\Omega$, which relations hold between $f(n)$ and $g(n)$ if ...

- $f(n) = n^2$, $g(n) = n^2 + n$ $f(n)$ is $O(g(n))$, $\Omega(g(n))$, $\Theta(g(n))$

- $f(n) = 2^n$, $g(n) = 3^n$ $f(n)$ is $o(g(n))$, $O(g(n))$

- $f(n) = n^2$, $g(n) = n \log n$ $f(n)$ is $\Omega(g(n))$, $\omega(g(n))$
Other Running Times: Factorial

• This is worse than exponential: \( n! = \omega(2^n) \)

• We could bound it by saying it’s no worse than \( n^n \) — but in fact, it’s \( o(n^n) \)

• “Sterling’s approximation” gives a more precise bound but we won’t cover that somewhat nasty equation

• But we could use Sterling’s approximation to show \( \log(n!) = \Theta(n \log n) \) (roughly because \( \log n^n = n \log n \))
Other Running Times: Polylogarithmic Running Times

- Suppose we’re trying to compare running times of $\Theta(\log^3 n)$ and $\Theta(n)$ — which one is faster?

- $\log^3 n$ is $(\log n)(\log n)(\log n)$

- Not only does every logarithm grow more slowly than a polynomial, but $\log^b n$ always grows slower than $n^a$ for $a > 0$. That is, $\log^b n = o(n^a)$ — in the limit, $\log^b n / n^a$ is 0.
Other Running Times: \( \lg^* n \)

- "Log-star": the number of times you would need to take the \( \log_2 \) (\( \lg \)) to get a number \( \leq 1 \)

- \( \lg^* 2 = 1 \) (\( \lg 2 = 1 \))

- \( \lg^* 4 = 2 \) (\( \lg \lg 4 = 1 \))

- \( \lg^* 16 = 3 \) (\( \lg \lg \lg 16 = 1 \))

- \( \lg^* 65536 = 4 \) (\( 2^{16} = 65536 \))

- \( \lg^* (2^{65536}) = 5 \)

- That is a pretty big number, so \( \lg^* n \) is usually 5 or less!
We describe algorithm speed by the growth in number of operations required as a function of N, the input size.

- We want that function to grow as slowly as possible!

- big-O: an upper bound on a growth rate, ignoring constants

- big-Ω: a lower bound on a growth rate, ignoring constants

- big-Θ: a tight bound on a growth rate, ignoring constants

- o, ω: “strictly less than,” “strictly greater than”

- Each polynomial degree Θ(n^k) is its own category of growth rate, and others are possible: Θ(n log n), Θ(n!) …

- Your algorithm’s choice of data structures can affect this degree of efficiency