Algorithm “efficiency” will pervade the course; what is it, how do we quantify it? “Big-O”!

Why big-O: measuring algorithm efficiency
What’s big-O: definition and related concepts
Reasoning with big-O: examples & applications
  - polynomials
  - exponentials
  - logarithms
  - sums

Polynomial Time
Why big-O: measuring algorithm efficiency
What is the $n^{\text{th}}$ prime number?

Let $p_n = n^{\text{th}}$ prime, $n \geq 1$, e.g.:

- $p_1 = 2$
- $p_2 = 3$
- $p_3 = 5$
- $p_4 = 7$
- $p_5 = 11$
- $p_6 = 13$

After much study, we know $p_n \sim n \log n$; even:

$$\frac{p_n}{n} = \log n + \log \log n - 1 + \frac{\log \log n - 2}{\log n} - \frac{(\log \log n)^2 - 6 \log \log n + 11}{2(\log n)^2} + o\left(\frac{1}{(\log n)^2}\right)$$

Great precision! But, often a simple, smooth, upper bound, is more convenient, e.g.: $p_n = O(n \log n)$

https://en.wikipedia.org/wiki/Prime_number_theorem
Our correct TSP algorithm was incredibly slow

_No matter what computer you have_

As a 2\textsuperscript{nd} example, for large problems, mergesort beats insertion sort – \( n \log n \) vs \( n^2 \) matters a lot

_No matter what computer you have_

Even tho m-sort is more complex & inner loop is slower

We want a general theory of “efficiency” that is

Simple
Objective
Relatively independent of changing technology
Measures \textit{algorithm}, not code
But still \textit{predictive} – “theoretically bad” algorithms should be bad in practice and vice versa (usually)
The *time complexity* of an algorithm associates a number $T(n)$, the *worst-case time* the algorithm takes, with each *problem size* $n$.

Mathematically,

\[ T: \mathbb{N}^+ \rightarrow \mathbb{R} \]

i.e., $T$ is a function mapping positive integers (problem sizes) to positive real numbers (number of steps).

“Reals” so, e.g., we can say $\sqrt{n}$ instead of $\left\lceil \sqrt{n} \right\rceil$

“Positive” so, e.g., $\log(n)$ and $2^n/n$ aren’t problematic
computational complexity

- Problem size
- Time

\( T(n) \)
computational complexity: general goals

Asymptotic growth rate: i.e., characterize growth rate of worst-case run time as a function of problem size, up to a constant factor, e.g. $T(n) = O(n^2)$

Why not try to be more precise?
- Average-case, e.g., is hard to define, analyze, maybe misleading
- Technological variations (computer, compiler, OS, ...) easily 10x or more
- Being more precise is much more work

A key question is "scale up": if I can afford this today, how much longer will it take when my business is 2x larger? (E.g. today: $cn^2$, next year: $c(2n)^2 = 4cn^2 : 4 \times$ longer.)

Big-O analysis is adequate to address this.

NOT saying $c=10$ vs $c=2$ is irrelevant; it’s secondary.
What’s big-O: definition and related concepts
O-notation, and relatives.

Given two functions $f$ and $g$: $\mathbb{N}^+ \rightarrow \mathbb{R}$

- $f(n)$ is $O(g(n))$ iff there is a constant $c > 0$ so that $f(n)$ is eventually always $\leq c \cdot g(n)$
- $f(n)$ is $\Omega(g(n))$ iff there is a constant $c > 0$ so that $f(n)$ is eventually always $\geq c \cdot g(n)$
- $f(n)$ is $\Theta(g(n))$ iff there are constants $c_1, c_2 > 0$ so that eventually always $c_1 g(n) \leq f(n) \leq c_2 g(n)$

"Eventually always P(n)" means "P(n) is true" except perhaps for finitely many "small" values of n. Formally: "$\exists n_0$ s.t. $\forall n > n_0$ P(n) is true."

For our applications, $f(n)$ is the (messy) actual time complexity of an algorithm, whereas $g(n)$ is a simple approximation to it.
Computational complexity

Problem size vs. Time

$T(n)$
Example: $T(n) = \Theta(n \log n)$ since for all problem sizes $n > n_0$, the worst case run time $T(n)$ is between $n \log_2 n$ and $2n \log_2 n$.
A program with initialization and two nested loops

initialize
for i in 1 .. n
    for j in 1 .. n
        do_something_simple(i, j)

might have runtime \( \approx \) like this:
If \( T(n) = n^2 + 30n + 5000 \), then \( T(n) = \Theta(n^2) \), since for all \( n \geq 135 \), we have \( n^2 \leq T(n) \leq 1.5n^2 \).
Summary of Big-O

Asymptotic Notation in Seven Words

suppress constant factors and lower-order terms

too system-dependent irrelevant for large inputs

Source: Tim Roughgarden’s Algorithms book
Reasoning with big-O: examples & applications

- polynomials
- exponentials
- logarithms
- sums
Show $10n^2 - 16n + 100$ is $O(n^2)$:

$10n^2 - 16n + 100 \leq 10n^2 + 100$

$= 10n^2 + 10^2$

$\leq 10n^2 + n^2 = 11n^2$ for all $n \geq 10$

$\therefore O(n^2)$ [and also $O(n^3), O(n^4), O(n^{2.5}), \ldots$]
Show $10n^2 - 16n + 100$ is $\Omega(n^2)$:

$10n^2 - 16n + 100 \geq 10n^2 - 16n$

$\geq 10n^2 - n^2 = 9n^2$ for all $n \geq 16$

$\therefore \Omega(n^2)$ [and also $\Omega(n)$, $\Omega(n^{1.5})$, …]

Therefore also $10n^2 - 16n + 100$ is $\Theta(n^2)$

[but not $\Theta(n^{1.999})$ or $\Theta(n^{2.001})$]
Polynomials:

\[ p(n) = a_0 + a_1 n + \ldots + a_d n^d \]
is \( \Theta(n^d) \) if \( a_d > 0 \)

Proof:

\[
p(n) = a_0 + a_1 n + \ldots + a_d n^d \\
\leq |a_0| + |a_1| n + \ldots + a_d n^d \\
\leq |a_0| n^d + |a_1| n^d + \ldots + a_d n^d \quad \text{(for } n \geq 1) \\
= c n^d, \text{ where } c = (|a_0| + |a_1| + \ldots + |a_{d-1}| + a_d) \\
\therefore p(n) = O(n^d)
\]

Exercise: show that \( p(n) = \Omega(n^d) \)

Hint: this direction is trickier; focus on the “worst case” where all coefficients except \( a_d \) are negative.
Example: For any \( a \), and any \( b > 0 \), \((n+a)^b\) is \( \Theta(n^b) \)

\[
(n+a)^b \leq (2n)^b \quad \text{for } n \geq |a|
\]
\[
= 2^b n^b
\]
\[
= c n^b \quad \text{for } c = 2^b
\]
so \((n+a)^b\) is \( O(n^b) \)

\[
(n+a)^b \geq (n/2)^b \quad \text{for } n \geq 2|a| \text{ (even if } a < 0) \]
\[
= 2^{-b} n^b
\]
\[
= c' n \quad \text{for } c' = 2^{-b}
\]
so \((n+a)^b\) is \( \Omega(n^b) \)
Example: $\sum_{1 \leq i \leq n} i = \Theta(n^2)$

Proof:

(a) An upper bound: each term is $\leq$ the max term
$$\sum_{1 \leq i \leq n} i \leq \sum_{1 \leq i \leq n} n = n^2 = O(n^2)$$

(b) A lower bound: each term is $\geq$ the min term
$$\sum_{1 \leq i \leq n} i \geq \sum_{1 \leq i \leq n} 1 = n = \Omega(n)$$

This is valid, but a weak bound.

Better: pick a large subset of large terms
$$\sum_{1 \leq i \leq n} i \geq \sum_{n/2 \leq i \leq n} n/2 \geq \left\lfloor n/2 \right\rfloor^2 = \Omega(n^2)$$

E.g.: for $i = 1..n$ {
    for $j=1$ to $i$ {
        ...
    }
}
Transitivity.
If \( f = O(g) \) and \( g = O(h) \) then \( f = O(h) \).
If \( f = \Omega(g) \) and \( g = \Omega(h) \) then \( f = \Omega(h) \).
If \( f = \Theta(g) \) and \( g = \Theta(h) \) then \( f = \Theta(h) \).

Additivity.
If \( f = O(h) \) and \( g = O(h) \) then \( f + g = O(h) \).
If \( f = \Omega(h) \) and \( g = \Omega(h) \) then \( f + g = \Omega(h) \).
If \( f = \Theta(h) \) and \( g = \Theta(h) \) then \( f + g = \Theta(h) \).

Proofs are left as exercises.
For all $r > 1$ (no matter how small) and all $d > 0$, (no matter how large)
$n^d = O(r^n)$

In short, every exponential grows faster than every polynomial!

(To prove this, use calculus tricks like L’Hospital’s rule.)
logarithms

\[ \log_a b = x \text{ means } a^x = b \]  \hspace{1cm} \text{definition}

**Examples:**

\[ 10^6 = 1,000,000, \text{ so } \log_{10} 1000000 = 6 \]
\[ 10^5 = 100,000, \text{ so } \log_{10} 100000 = 5 \]

i.e., \( \log_{10}(n) \approx \text{number of digits in } n \); also
\[ \log_2(n) \approx \text{number of bits in } n \]

**Key properties:**

\[ \log(x \cdot y) = \log(x) + \log(y) \]
\[ \log(x/y) = \log(x) - \log(y) \]
\[ \log(x^y) = y \cdot \log(x) \]

**Change-of-base formula:**

\[ \log_a(x) = \frac{\log_b(x)}{\log_b(a)} \]
\[ \log_2(x) = \frac{\log_{10}(x)}{\log_{10}(2)} = \frac{\log_{10}(x)}{0.30103} \]
Example: For any $a, b > 1$ \( \log_a n \) is $\Theta(\log_b n)$

\[
\log_a b = x \text{ means } a^x = b \\
a^{\log_a b} = b \\
(a^{\log_a b})^{\log_b n} = b^{\log_b n} = n \\
(\log_a b)(\log_b n) = \log_a n \\
c \log_b n = \log_a n \text{ for the constant } c = \log_a b \\
\text{So:} \\
\log_b n = \Theta(\log_a n) = \Theta(\log n)
\]

Corollary: base of a log factor is usually irrelevant, asymptotically. E.g. “$O(n \log n)$” [but \( n^{\log_2 8} \neq O(n^{\log_8 8}) \)]
Logarithms:
For all $x > 0$, *(no matter how small)* \( \log n = O(n^x) \)

*log grows slower than every polynomial*
big-theta, etc. are not always “nice”

\[ f(n) = \begin{cases} 
  n^2, & n \text{ even} \\
  n, & n \text{ odd} 
\end{cases} \]

\( f(n) \neq \Theta(n^a) \) for any \( a \).

Fortunately, such nasty cases are rare

\( n \log n \neq \Theta(n^a) \) for any \( a \), either, but at least it’s simpler.
Polynomial Time
the complexity class P: polynomial time

P: The set of problems solvable by algorithms with running time $O(n^d)$ for some constant $d$  
(d is a constant independent of the input size $n$)

Nice scaling property: there is a constant $c$ s.t. doubling $n$, increases time only by a factor of $c$.  
(E.g., $c \sim 2^d$)

Contrast with exponential: For any constant $c$, there is a $d$ such that $n \rightarrow n+d$ increases time by a factor of more than $c$.  
(E.g., $c = 100$ and $d = 7$ for $2^n$ vs $2^{n+7}$)
polynomial vs exponential growth

\(2^{2n}\)

\(2^{n/10}\)

\(1000n^2\)
why it matters

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds $10^{25}$ years, we simply record the algorithm as taking a very long time.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n$</th>
<th>$n \log_2 n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$1.5^n$</th>
<th>$2^n$</th>
<th>$n!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>4 sec</td>
</tr>
<tr>
<td>30</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>18 min</td>
<td>$10^{25}$ years</td>
</tr>
<tr>
<td>50</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td></td>
<td>11 min</td>
<td>36 years</td>
</tr>
<tr>
<td>100</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td></td>
<td>1 sec</td>
<td>12,892 years</td>
<td>very long</td>
</tr>
<tr>
<td>1,000</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td></td>
<td></td>
<td>18 min</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>10,000</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td></td>
<td></td>
<td>2 min</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>100,000</td>
<td>&lt; 1 sec</td>
<td></td>
<td></td>
<td></td>
<td>3 hours</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>1,000,000</td>
<td>1 sec</td>
<td></td>
<td></td>
<td></td>
<td>31,710 years</td>
<td>very long</td>
<td>very long</td>
</tr>
</tbody>
</table>

not only get very big, but do so abruptly, which likely yields erratic performance on small instances
Next year’s computer will be 2x faster. If I can solve problem of size $n_0$ today, how large a problem can I solve in the same time next year?

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Size Increase</th>
<th>E.g. $T=10^{12}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(n)$</td>
<td>$n_0 \rightarrow 2n_0$</td>
<td>$10^{12} \rightarrow 2 \times 10^{12}$</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>$n_0 \rightarrow \sqrt{2} n_0$</td>
<td>$10^6 \rightarrow 1.4 \times 10^6$</td>
</tr>
<tr>
<td>$O(n^3)$</td>
<td>$n_0 \rightarrow 3\sqrt{2} n_0$</td>
<td>$10^4 \rightarrow 1.25 \times 10^4$</td>
</tr>
<tr>
<td>$2^n/10$</td>
<td>$n_0 \rightarrow n_0+10$</td>
<td>$400 \rightarrow 410$</td>
</tr>
<tr>
<td>$2^n$</td>
<td>$n_0 \rightarrow n_0+1$</td>
<td>$40 \rightarrow 41$</td>
</tr>
</tbody>
</table>
why “polynomial”?

Point is not that $n^{2000}$ is a nice time bound, or that the differences among $n$ and $2n$ and $n^2$ are negligible.

Rather, simple theoretical tools may not easily capture such differences, whereas exponentials are qualitatively different from polynomials, so more amenable to theoretical analysis.

“My problem is in P” is a starting point for a more detailed analysis

“My problem is not in P” may suggest that you need to shift to a more tractable variant, or otherwise readjust expectations
Summary
A typical initial goal for algorithm analysis is to find a reasonably tight, i.e., $\Theta$ if possible, asymptotic, i.e., $O$ or $\Theta$, bound on usually upper bound worst case running time as a function of problem size.

This is rarely the last word, but often helps separate good algorithms from blatantly poor ones – so you can concentrate on the good ones!

As one important example, poly time algorithms are almost always preferable to exponential time ones.