Measuring Complexity
Measuring Complexity

• What constitutes an efficient algorithm?
  • Runs quickly on large, ‘real’ instances of problems
  • Qualitatively better than brute force
  • Scales well to large instances
Measuring Complexity: Brute Force

- Efficient: Qualitatively better than brute force

- Brute force: often exponentially large because
  - Might examine all subsets of a set: $2^n$
  - Might examine all orderings of a list: $n!$

- But $2^n$ is still not efficient even though it’s qualitatively better than $n!$
Measuring Complexity : Scalability

- Defining Scalability
  - Doubling problem size increases resource use by a constant factor

- Examples
  - $n^k : (2n)^k = 2^k \cdot n^k = c \cdot n^k$ for any fixed $k$
  - $\log n : \log 2n = \log 2 + \log n \leq c \log n \ (n \geq 2)$

- But not
  - $2^n : 2^{2n} = 2^n \cdot 2^n$
  - $n! : (2n)! \geq n^n \cdot n!$
Polynomial Time

Functions that exhibit scalability can be bounded above by some *polynomial* function.

This leads us to define an efficient algorithm as one having running time bounded above by a polynomial function.

But how do we measure running time?
**Worst Case Runtime**

- Worst-case running time: the maximum number of steps needed to solve a problem instance of size $n$
  - Overestimates the typical run-time
  - Can frequently be determined by algorithm analysis
  - Frequently captures the runtime in practice
  - Often there’s no “worst” case we can identify
  - Don’t fall into the “the worst case is when…” trap!
Average Case Runtime

- Average Case Runtime: The *mean* number of operations needed across *all instances* of size \( n \)

- A much more nuanced, realistic measure

- Very difficult to determine in practice

  - Are all instances equally likely?

  - If not, can we specify the probability distribution?
# Growth of Functions

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>$n = 10$</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>$n = 30$</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>$n = 50$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>11 min</td>
<td>36 years</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 100$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>12,892 years</td>
<td>$10^{17}$ years</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 1,000$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>18 min</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 10,000$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>2 min</td>
<td>12 days</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 100,000$</td>
<td>&lt; 1 sec</td>
<td>2 sec</td>
<td>3 hours</td>
<td>32 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 1,000,000$</td>
<td>1 sec</td>
<td>20 sec</td>
<td>12 days</td>
<td>31,710 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
</tbody>
</table>
Asymptotic Growth

What matters: How functions behave “as n gets large”
Asymptotic Upper Bounds

**Def’n:** $g(n)$ is *eventually bounded above by* $f(n)$ if for all sufficiently large $n$, $g(n) \leq f(n)$

That is, for some $N$, $g(n) \leq f(n)$ for all $n \geq N$

**Def’n:** $O(f(n)) = \{ g(n) : \text{for some constant } c > 0, \ g(n) \text{ is eventually bounded above by } c f(n) \}$

**Examples**

- $100 \ n^2 \in O(n^3) : 100 \ n^2 \leq c \ n^3$ for $c = 100$ and $n \geq 1$

- $3n \in O(n \log_2 n) : 3n \leq c \ n \log_2 n$ for $c = 1$ and $n \geq 8$
More Challenging Examples

• $\log_2(\log_2(n)) \in O(\log_2(n))$

• Fact: For $1 < a < b$, $0 < \log(a) < \log(b)$

• For $n > 2$, $1 < \log_2(n) < n$, so $0 < \log_2(\log_2(n)) < \log_2(n)$

• $3n^4 - n^3 + 10n^2 - 4n + 5 \in O(n^5)$

• $3n^4 - n^3 + 10n^2 - 4n + 5 \leq 3n^4 + 10n^2 + 5 \leq 18n^4 \leq 18n^5$

• In fact, this shows that $3n^4 - n^3 + 10n^2 - 4n + 5 \in O(n^4)$
Asymptotic Lower Bounds

Def’n: $g(n)$ is \textit{eventually bounded below by} $f(n)$ if for all sufficiently large $n$, $f(n) \leq g(n)$

Def’n: $\Omega(f(n)) = \{ g(n) : \text{for some constant } c > 0, \text{ } g(n) \text{ is eventually bounded below by } c \cdot f(n) \}$

Examples

- $100 \cdot n^3 \in \Omega(n^2) : c \cdot n^2 \leq 100 \cdot n^3 \text{ for } c = 1 \text{ and } n \geq 1$

- $n \log_2n \in \Omega(n) : c \cdot n \leq n \log_2n \text{ for } c = 1 \text{ and } n \geq 2$
Why Lower Bounds?

Show that an algorithm performs \textit{at least} so many steps

- Searching an unordered list of \( n \) items takes \( \Omega(n) \) steps in some cases
- Quicksort (and selection/insertion/bubble sorts) take \( \Omega(n^2) \) steps in some cases
- Mergesort takes \( \Omega(n \log(n)) \) steps in all cases
Asymptotically Tight Bounds

**Def’n:** $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$

That is: $\Theta(f(n)) = \{ g(n) : g(n) \in O(f(n)) \text{ and } g(n) \in \Omega(f(n)) \}$

If $g(n) \in \Theta(f(n))$, then $f$ is an **asymptotically tight bound** on $g$

**Example**

- $n^3 - 4n^2 + 2 \in \Theta(n^3)$
  - $n^3 - 4n^2 + 2 \in O(n^3) : n^3 - 4n^2 + 2 \leq cn^3$ for $n \geq 1$ & $c = 1$
  - $n^3 - 4n^2 + 2 \in \Omega(n^3) : c'n^3 \leq n^3 - 4n^2 + 2$ for $n \geq 8$ & $c' = 1/2$
A Tool

**Def’n:** A function $f: \mathbb{N} \to \mathbb{R}$ is **eventually positive** if $f(n) > 0$ for all sufficiently large $n \in \mathbb{N}$

**Thm:** Let $f(n)$ and $g(n)$ be eventually positive functions

If $\lim_{x \to \infty} \frac{f(x)}{g(x)} = 0$, then $f(x) \in O(g(x))$ and $g(x) \in \Omega(f(x))$

If $\lim_{x \to \infty} \frac{f(x)}{g(x)} = \infty$, then $f(x) \in \Omega(g(x))$ and $g(x) \in O(f(x))$

If $\lim_{x \to \infty} \frac{f(x)}{g(x)} = c > 0$, then $f(x) \in \Theta(g(x))$ and $g(x) \in \Theta(f(x))$
Using the Tool

Example: \(3n^4 - n^3 + 10n^2 - 4n + 5 \in O(n^5)\)

- Let \(f(n) = 3n^4 - n^3 + 10n^2 - 4n + 5\) and \(g(n) = n^5\)

- \(f(n)/g(n) = (3n^4 - n^3 + 10n^2 - 4n + 5)/n^5\)

- \((3n^4 - n^3 + 10n^2 - 4n + 5)/n^5 = 3/n - 1/n^2 + 10/n^3 - 4/n^4 + 5/n^5\)

- \(3/n - 1/n^2 + 10/n^3 - 4/n^4 + 5/n^5 \to 0\) as \(n \to \infty\)

Example: \(5n^3 + n^2 - 10 \in \Omega(n \log_2 n)\)

- \((5n^3 + n^2 - 10)/n \log_2 n = 5n^2/\log_2 n - n/\log_2 n - 10/n \log_2 n\)

- \((5n - 1) n/\log_2 n - 10/n \log_2 n \to \infty\) as \(n \to \infty\)
Using the Tool

Example: \(5n^3 + n^2 - 10 \in \Theta(n^3)\)

- \(\frac{(5n^3 + n^2 - 10)}{n^3} = 5 + \frac{1}{n} - \frac{10}{n^3}\)
- \(5 + \frac{1}{n} - \frac{10}{n^3} \rightarrow 5\) as \(n \rightarrow \infty\)

Note: \(a_k n^k + a_{k-1} n^{k-1} + \ldots + a_1 n + a_0 \in \Theta(n^k)\) if \(a_k > 0\)
A (Much) Better Tool

Thm: L’Hôpital’s Rule

If

1. \( \lim_{n \to \infty} f(n) = \lim_{n \to \infty} g(n) = \infty \)

2. \( \lim_{n \to \infty} f'(n)/g'(n) \) exists

then \( \lim_{n \to \infty} f(n)/g(n) = \lim_{n \to \infty} f'(n)/g'(n) \)

Example: \( \log n \in O(n) \): Here \( f(n) = \log n \) and \( g(n) = n \)

1. \( \lim_{n \to \infty} f(n)/g(n) = \lim_{n \to \infty} (\log n)/n = \lim_{n \to \infty} (\ln n/\ln 2)/n \)

2. \( \lim_{n \to \infty} f'(n)/g'(n) = \lim_{n \to \infty} (1/(n \ln 2))/1 = 0 \)
Speaking of logs…

**Def’n:** For any \( b > 1 \), \( \log_b n \) is the number \( x \) such that \( b^x = n \).

**Facts**

- For every \( b > 1 \) and \( \varepsilon > 0 \), \( \log_b n \in O(n^\varepsilon) \) (logs grow slowly!)
  - Because as \( n \to \infty \), \( (\log_b n)/n^\varepsilon \to 0 \) (by L’Hôpital’s Rule)
- \( \log_a n = \log_b n / \log_b a \)
  - Example: \( \log_2 n = \log_{10} n / \log_2 10 \approx 0.3 \log_{10} n \)
Fact

• For every $r > 1$ and $d > 0$, $n^d \in O(r^n)$ (exponentials grow fast)

  • Because as $n \to \infty$, $n^d/r^n \to 0$ (by L’Hôpital’s Rule)

• Restated: Every exponential function grows faster than every power of $n$ (in fact, faster than every polynomial)

Review basic facts about logs and exponentials!
**Thm:** Let $f, g,$ and $h$ be eventually positive functions

Transitivity of $O,$ $\Omega,$ and $\Theta$

If $f \in O(g)$ and $g \in O(h),$ then $f \in O(h)$ (same for $\Omega,$ $\Theta$)

Sum Rule

If $f \in O(h)$ and $g \in O(h),$ then $f + g \in O(h)$ (same for $\Omega,$ $\Theta$)

Careful: This assumes that $f$ and $g$ are eventually positive!
In the Wild

Sample Usage: Quicksorting an $n$-element array

• Thm: The number of operations performed is $O(n^2)$

• Thm: The average number of operations performed is $O(n \log(n))$

• Thm: Quicksort will always execute $\Omega(n \log(n))$ steps

Sample Usage: Basic Bubblesort runs in time $\Theta(n^2)$
\( \Theta(f) \) and Operation Count

Which operations should count when analyzing run time?

- All of them?
  - Can be difficult and/or tedious

- Better: A \textit{representative sample}
  - If the algorithm performs \( f(n) \) operations, it suffices to count \( g(n) \) of them, where \( g(n) \in \Theta(f(n)) \)