CSE 421: Introduction to Algorithms

Complexity

Yin-Tat Lee
Defining Efficiency

“Runs fast on typical real problem instances”

Pros:
• Sensible,
• Bottom-line oriented

Cons:
• Moving target (diff computers, programming languages)
• Highly subjective (how fast is “fast”? What is “typical”? )
Measuring Efficiency

Time \approx \# \text{ of instructions executed in a simple programming language}

- only simple operations (+, *, -, =, if, call, …)
- each operation takes one time step
- each memory access takes one time step
- no fancy stuff (add these two matrices, copy this long string, …) built in
Time Complexity

Problem: An algorithm can have different running time on different inputs

Solution: The complexity of an algorithm associates a number \( T(N) \), the “time” the algorithm takes on problem size \( N \).

Mathematically,

\( T \) is a function that maps positive integers giving problem size to positive integers giving number of steps.
Time Complexity (N)

Worst Case Complexity: \textbf{max} \# steps algorithm takes on any input of size \(N\)

Average Case Complexity: \textbf{avg} \# steps algorithm takes on inputs of size \(N\)

Best Case Complexity: \textbf{min} \# steps algorithm takes on any input of size \(N\)
Time Complexity on Worst Case Inputs

\[ T(N) \]

\[ 2N \log_2 N \]

\[ N \log_2 N \]

Problem size \( N \)

Time
O-Notation

Given two positive functions $f$ and $g$

- $f(N)$ is $O(g(N))$ iff there is a constant $c > 0$ s.t., $f(N)$ is eventually always $\leq c \cdot g(N)$

- $f(N)$ is $\Omega(g(N))$ iff there is a constant $\varepsilon > 0$ s.t., $f(N)$ is $\geq \varepsilon \cdot g(N)$ for infinitely

- $f(N)$ is $\Theta(g(N))$ iff there are constants $c_1, c_2 > 0$ so that eventually always $c_1 g(N) \leq f(N) \leq c_2 g(N)$
Asymptotic Bounds for common fns

- **Polynomials:**
  \[ a_0 + a_1 n + \cdots + a_d n^d \text{ is } O(n^d) \]

- **Logarithms:**
  \[ \log_a n = O(\log_b n) \text{ for all constants } a, b > 0 \]

- **Logarithms:** log grows slower than every polynomial
  For all \( x > 0 \), \[ \log n = O(n^k) \]

- \( n \log n = O(n^{1.01}) \)
An algorithm runs in polynomial time if $T(n) = n^{O(1)}$. Equivalently, $T(n) = n^d$ for some constant d.

<table>
<thead>
<tr>
<th>Name</th>
<th>Complexity class</th>
<th>Running time ($T(n)$)</th>
<th>Examples of running times</th>
<th>Example algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant time</td>
<td>$O(1)$</td>
<td></td>
<td>10</td>
<td>Determining if an integer (represented in binary) is even or odd</td>
</tr>
<tr>
<td>inverse Ackermann time</td>
<td>$O(a(n))$</td>
<td></td>
<td></td>
<td>Amortized time per operation using a disjoint set</td>
</tr>
<tr>
<td>iterated logarithmic time</td>
<td>$O(\log^* n)$</td>
<td></td>
<td></td>
<td>Distributed coloring of cycles</td>
</tr>
<tr>
<td>log-logarithmic</td>
<td>$O(\log \log n)$</td>
<td></td>
<td>log $n$, log$(n^2)$</td>
<td>Amortized time per operation using a bounded priority queue[^2]</td>
</tr>
<tr>
<td>logarithmic time</td>
<td>DLOGTIME</td>
<td>$O(\log n)$</td>
<td></td>
<td>Binary search</td>
</tr>
<tr>
<td>polylogarithmic time</td>
<td>poly$(\log n)$</td>
<td>$(\log n)^2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fractional power</td>
<td>$O(n^c)$ where $0 &lt; c &lt; 1$</td>
<td>$n^{1/2}$, $n^{2/3}$</td>
<td></td>
<td>Searching in a kd-tree</td>
</tr>
<tr>
<td>linear time</td>
<td>$O(n)$</td>
<td>$n$</td>
<td></td>
<td>Finding the smallest or largest item in an unsorted array</td>
</tr>
<tr>
<td>&quot;n log star n&quot; time</td>
<td>$O(n \log^* n)$</td>
<td></td>
<td></td>
<td>Seidel’s polygon triangulation algorithm.</td>
</tr>
<tr>
<td>quasilinear time</td>
<td>$O(n \log n)$</td>
<td>$n \log n$, $\log n!$</td>
<td></td>
<td>Fastest possible comparison sort; Fast Fourier transform.</td>
</tr>
<tr>
<td>quadratic time</td>
<td>$O(n^2)$</td>
<td>$n^2$</td>
<td></td>
<td>Bubble sort; Insertion sort; Direct convolution</td>
</tr>
<tr>
<td>cubic time</td>
<td>$O(n^3)$</td>
<td>$n^3$</td>
<td></td>
<td>Naive multiplication of two $n \times n$ matrices. Calculating partial correlation.</td>
</tr>
<tr>
<td>polynomial time</td>
<td>P</td>
<td>$2^{O(\log n)} = \text{poly}(n)$</td>
<td>$n$, $n \log n$, $n^{10}$</td>
<td>Karmarkar’s algorithm for linear programming; AKS primality test</td>
</tr>
<tr>
<td>quasi-polynomial time</td>
<td>QP</td>
<td>$2^{\text{polylog}(n)}$</td>
<td>$n^2 \log^2 n$, $n^{\log n}$</td>
<td>Best-known $O(\log^2 n)$-approximation algorithm for the directed Steiner tree problem.</td>
</tr>
<tr>
<td>sub-exponential time (first definition)</td>
<td>SUBEXP</td>
<td>$O(2^{n^\varepsilon})$ for all $\varepsilon &gt; 0$</td>
<td>$O(2^{\log n \log^\alpha n})$</td>
<td>Assuming complexity theoretic conjectures, BPP is contained in SUBEXP[^3]</td>
</tr>
<tr>
<td>sub-exponential time (second definition)</td>
<td></td>
<td>$2^{o(n)}$</td>
<td>$2^{n^{1/3}}$</td>
<td>Best-known algorithm for integer factorization</td>
</tr>
<tr>
<td>exponential time (with linear exponent)</td>
<td>E</td>
<td>$2^{O(n)}$</td>
<td>$1.1^n$, $10^n$</td>
<td>Solving the traveling salesman problem using dynamic programming</td>
</tr>
<tr>
<td>exponential time</td>
<td>EXPTIME</td>
<td>$2^{\text{poly}(n)}$</td>
<td>$2^n$, $2^{n^2}$</td>
<td>Solving matrix chain multiplication via brute-force search</td>
</tr>
<tr>
<td>factorial time</td>
<td>$O(n!)$</td>
<td>$n!$</td>
<td></td>
<td>Solving the traveling salesman problem via brute-force search</td>
</tr>
<tr>
<td>double exponential time</td>
<td>2-EXPTIME</td>
<td>$2^{\text{poly}(n)}$</td>
<td>$2^{2^n}$</td>
<td>Deciding the truth of a given statement in Presburger arithmetic</td>
</tr>
</tbody>
</table>
Why it matters?

Suppose we can do 1 million operations per second.

<table>
<thead>
<tr>
<th>n</th>
<th>n</th>
<th>n log₂ n</th>
<th>n²</th>
<th>n³</th>
<th>1.5ⁿ</th>
<th>2ⁿ</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>10²⁵ 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>10¹⁷ 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>

not only get very big, but do so abruptly, which likely yields erratic performance on small instances

Outdated: Nvidia announced a “computer” this Tue that do 2 quadrillion (2 × 10¹⁵) operations/sec. It brings down the 31,710 years to 500 sec. However, 2¹⁰⁰ operations still takes millions of years.
Why “Polynomial”?

Point is not that $n^{2000}$ is a practical 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
Graphs
Undirected Graphs $G=(V,E)$
Graphs don’t Live in Flat Land

Geometrical drawing is mentally convenient, but mathematically irrelevant:

4 drawings of a single graph:
Terminology

- **Degree of a vertex**: \# edges that touch that vertex

\[ \text{deg}(6) = 3 \]

- **Connected**: Graph is connected if there is a path between every two vertices

- **Connected component**: Maximal set of connected vertices
Terminology (cont’d)

• **Path**: A sequence of distinct vertices s.t. each vertex is connected to the next vertex with an edge

• **Cycle**: Path of length > 2 that has the same start and end

• **Tree**: A connected graph with no cycles
Degree Sum

**Claim**: In any undirected graph, the number of edges is equal to $(1/2) \sum_{\text{vertex } v} \deg(v)$

**Pf**: $\sum_{\text{vertex } v} \deg(v)$ counts every edge of the graph exactly twice; once from each end of the edge.

$|E| = 8$

$$\sum_{\text{vertex } v} \deg(v) = 2 + 2 + 1 + 1 + 3 + 2 + 3 + 2 = 16$$
**Odd Degree Vertices**

**Claim**: In any undirected graph, the number of odd degree vertices is even

**Pf**: In previous claim we showed sum of all vertex degrees is even. So there must be even number of odd degree vertices, because sum of odd number of odd numbers is odd.

4 odd degree vertices: 3, 4, 5, 6
Let $G = (V, E)$ be a graph with $n = |V|$ vertices and $m = |E|$ edges.

**Claim:** $0 \leq m \leq \binom{n}{2} = \frac{n(n-1)}{2} = O(n^2)$

**Pf:** Since every edge connects two distinct vertices (i.e., $G$ has no loops) and no two edges connect the same pair of vertices (i.e., $G$ has no multi-edges) it has at most $\binom{n}{2}$ edges.