CSE 421: Introduction to Algorithms

Course Overview

Shayan Oveis Gharan
HW1 is due Thursday April 08 at 11:59PM
Please submit to Gradescope

Late Submission: Fill out an extension request in edstem.

How to submit?
• Double check your submission before the deadline!!
• Please typeset your solution if possible

Guidelines:
• Always justify your answer
• You can collaborate, but you must write solutions on your own
• Your proofs should be clear, well-organized, and concise. Spell out main idea.
• Sanity Check: Spell out when you use assumptions of the problem
Induction: Intro 2

Prove that every instance of stable matchings with n companies and n applicants where some participant declare others as unacceptable has at most $n! = n(n - 1) \ldots 21$ many perfect matchings.
Five Representative Problems

1. Interval Scheduling
2. Weighted Interval Scheduling
3. Bipartite Matching
4. Independent Set Problem
5. Competitive Facility Location
Interval Scheduling

**Input:** Given a set of jobs with start/finish times

**Goal:** Find the **maximum cardinality** subset of jobs that can be run on a single machine.
Interval Scheduling

**Input:** Given a set of jobs with start/finish times

**Goal:** Find the **maximum weight** subset of jobs that can be run on a single machine.
Bipartite Matching

Input: Given a bipartite graph

Goal: Find the maximum cardinality matching
Independent Set

Input: A graph

Goal: Find the maximum independent set

Subset of nodes that no two joined by an edge
Competitive Facility Location

**Input**: Graph with weight on each node

**Game**: Two competing players alternate in selecting nodes. Not allowed to select a node if any of its neighbors have been selected.

**Goal**: Does player 2 have a strategy which guarantees a total value of $V$ no matter what player 1 does?

Second player can guarantee 20, but not 25.
Five Representative Problems

Variation of a theme: Independent set Problem

1. Interval Scheduling
   $n \log n$ greedy algorithm

2. Weighted Interval Scheduling
   $n \log n$ dynamic programming algorithm

3. Bipartite Matching
   $n^k$ maximum flow based algorithm

4. Independent Set Problem: NP-complete

5. Competitive Facility Location: PSPACE-complete
Main Objective: Design Efficient Algorithms that finds optimum solutions in the Worst Case
Measuring Efficiency

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

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; write it/charge for it as above
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

On which inputs of size $N$?
Time Complexity (N)

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

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

Best Case Complexity: \( \text{min} \) # steps algorithm takes on any input of size \( N \)
Why Worst-case Inputs?

• Analysis is typically easier

• Useful in real-time applications  
  e.g., space shuttle, nuclear reactors)

• Worst-case instances kick in when an algorithm is run as a module many times  
  e.g., geometry or linear algebra library

• Useful when running competitions  
  e.g., airline prices

• Unlike average-case no debate about the right definition
Time Complexity on Worst Case Inputs

\[ T(N) = 2N \log_2 N \]

\[ T(N) = N \log_2 N \]
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 \( x > 0 \), \( \log n = O(n^k) \)

- \( n \log n = O(n^{1.01}) \)
Efficient = Polynomial Time

An algorithm runs in polynomial time if $T(n)=O(n^d)$ for some constant $d$ independent of the input size $n$.

Why Polynomial time?

If problem size grows by at most a constant factor then so does the running time

- E.g. $T(2N) \leq c(2N)^k \leq 2^k(cN^k)$
- Polynomial-time is exactly the set of running times that have this property

Typical running times are small degree polynomials, mostly less than $N^3$, at worst $N^6$, not $N^{100}$
Why it matters?

- #atoms in universe $< 2^{240}$
- Life of the universe $< 2^{54}$ seconds
- A CPU does $< 2^{30}$ operations a second

If every atom is a CPU, a $2^n$ time ALG cannot solve $n=350$ if we start at Big-Bang.

<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>11 min</td>
<td>36 years</td>
<td>very long</td>
</tr>
<tr>
<td>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>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>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>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>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.