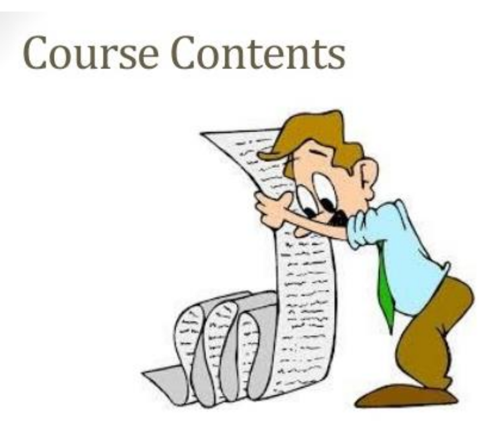
CSE 421

Course Overview / Complexity



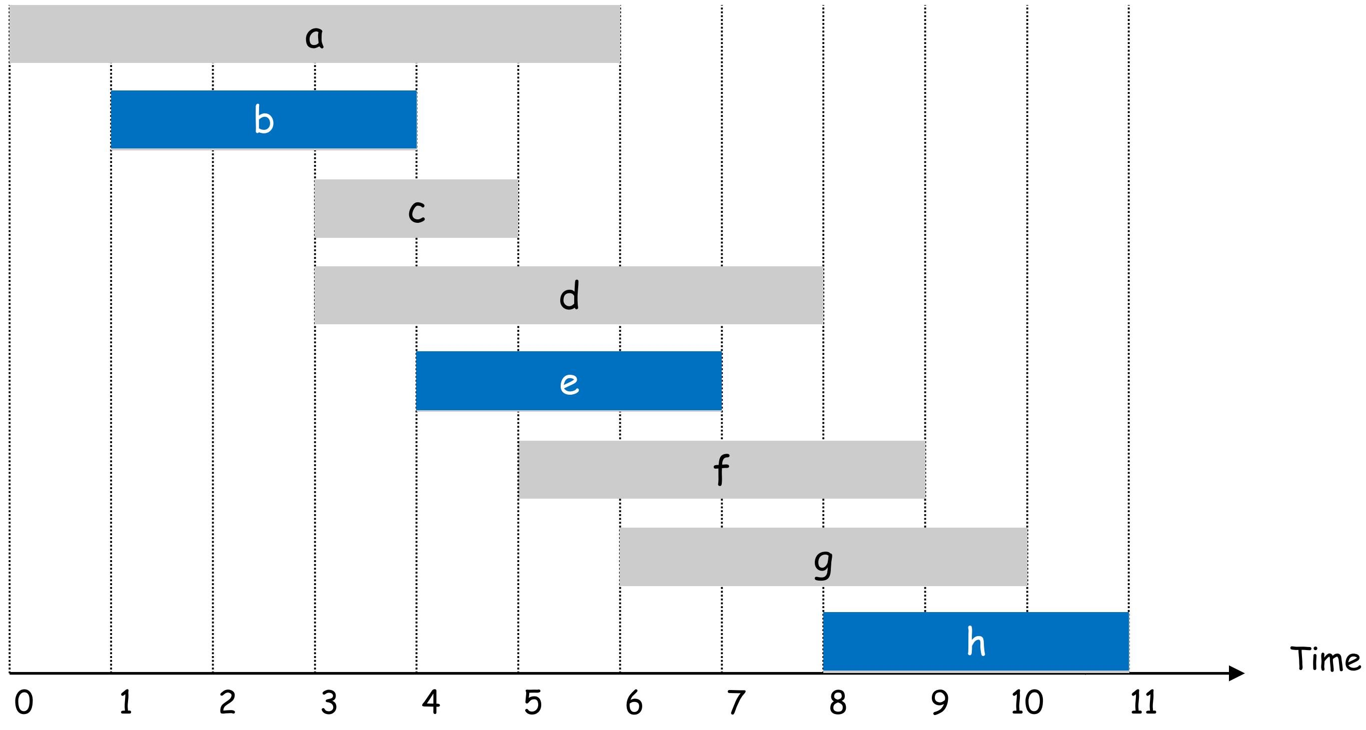
Five Representative Problems

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

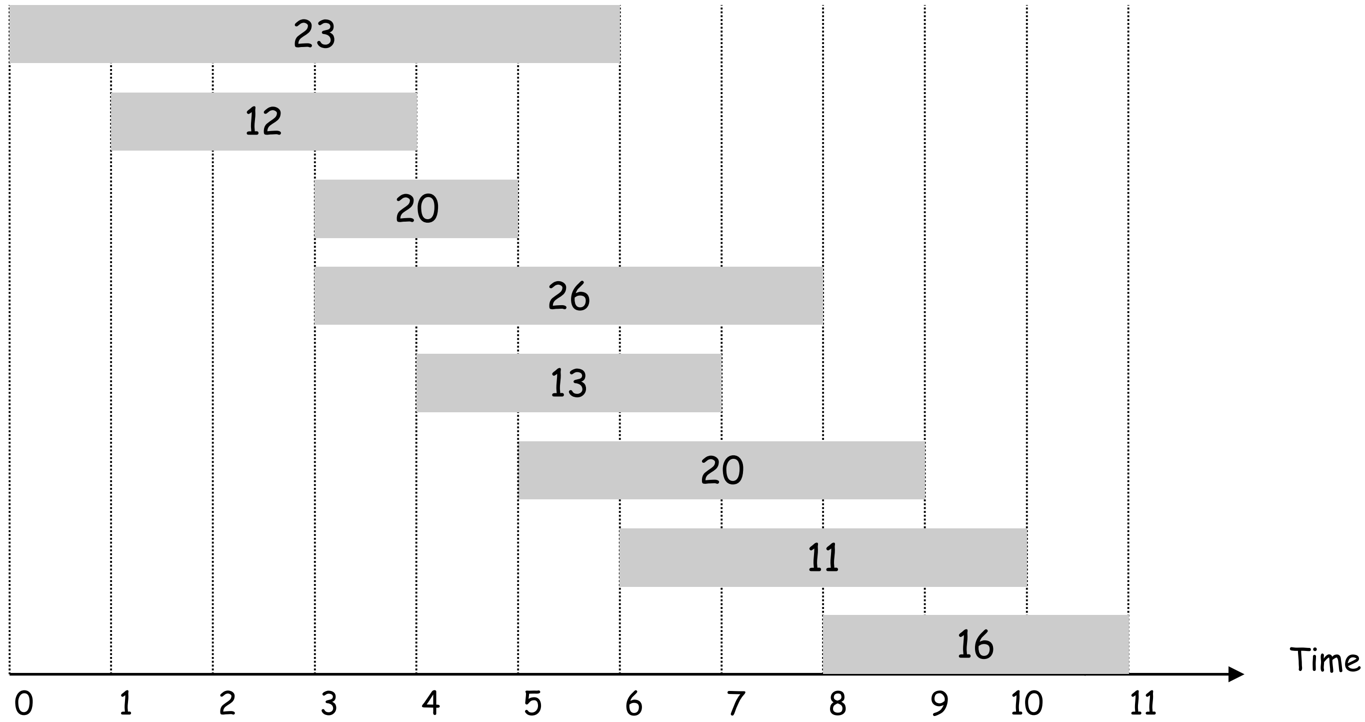
Goal: Find the maximum cardinality subset of jobs that can be run on a single machine.



Interval Scheduling

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

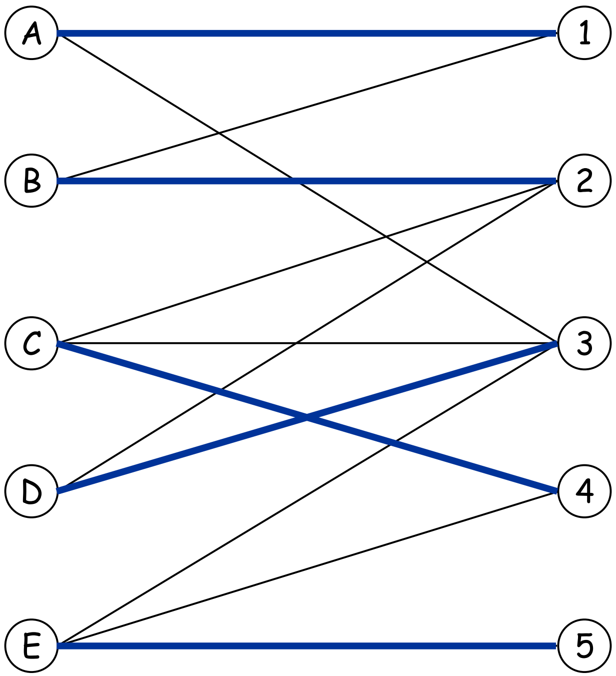
Goal: Find the maximum weight subset of jobs that can be run on a single machine.



Bipartite Matching

Input: Given a bipartite grah

Goal: Find the maximum cardinality matching

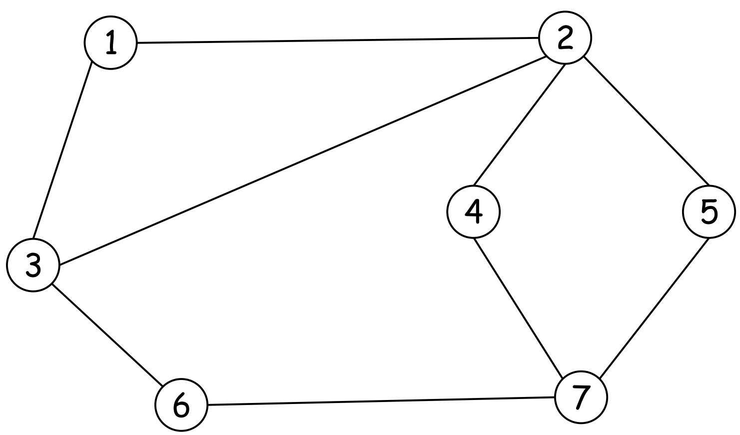


Independent Set

Input: A graph

Goal: Find the maximum independent set

Subset of nodes that no two joined by an edge

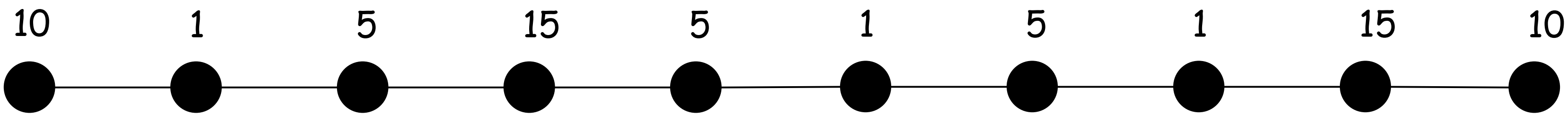


Competitive Facility Location

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

Defining Efficient Algorithms

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 ≈ # 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**.

On which inputs of 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: **max** # steps algorithm takes on any input of size **N**

This Couse

Average Case Complexity: **avg** # steps algorithm takes on inputs of size **N**

Best Case Complexity: **min** # steps algorithm takes on any input of size **N**

Why Worst-case Inputs?

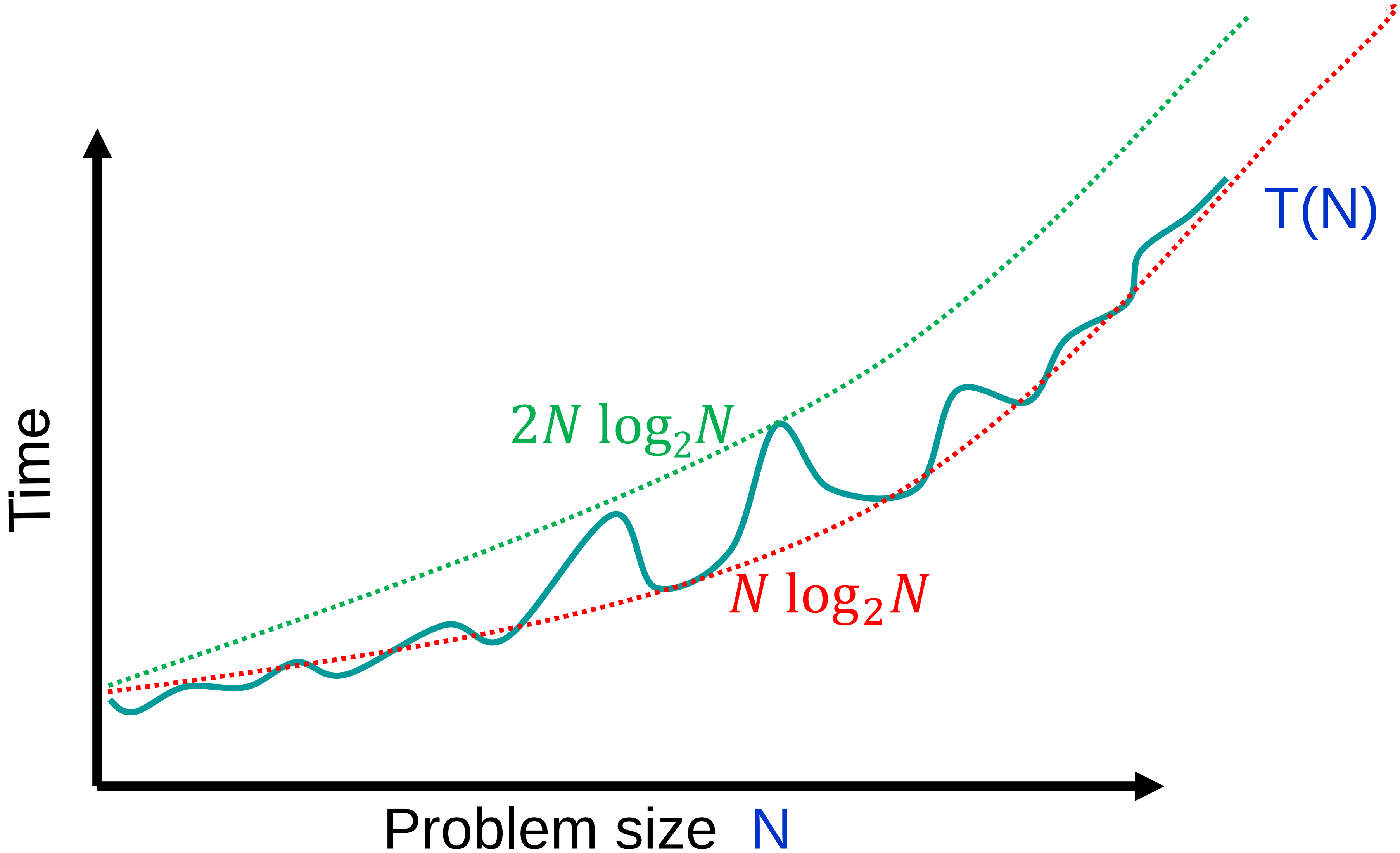
- Analysis is typically easier
- Useful in real-time applications

  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



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 **$\leq c\, g(N)$**

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

  $a_0 + a_1 n + \cdots + a_d n^d$ is $O\left(n^d\right)$

- Logarithms:

  $\log_a n = O(\log_b n)$ for all constants $a, b > 0$

- Logarithms: log grows slower than every polynomial

  For all $x > 0$, $\log n = O(n^k)$

- $n \log n = O\left(n^{1.01}\right)$

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. $\mathbf{T(2N)} \le c(2N)^k \le 2^k(cN^k)$
- 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.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

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

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