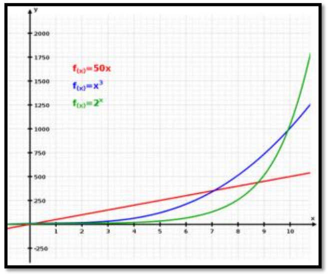
CSE 421: Algorithms

Winter 2014

Lecture 3: Asymptotic analysis

Reading: Chapter 2 of Kleinberg-Tardos



defining efficiency

"Runs fast on typical real-world problem instances"

- Pro: sensible, straight to the point
- Cons:
  - moving target (different computers, architectures, compilers, Moore's law)
  - highly subjective (how fast is "fast"? what is "typical"?)

defining efficiency

"Runs fast on a specific suite of benchmarks"

- Pro: sensible, straight to the point
- Cons:
  - previous problems
  - which benchmarks?
  - algorithms can be tuned to do well on the benchmarks
  - have to find a benchmark before we can compare algorithms? (design would take forever)

defining efficiency

Instead, we:

- Give up on detailed timing, focus on *scaling.*
- Give up on "typical." Focus on worst-case behavior.

defining efficiency: the RAM model

- RAM = Random Access Machine
- Time ≈ # of instructions executed in an ideal assembly language
  - each simple operation (+,*,-,=, if, call) takes one time step
  - each memory access takes one time step

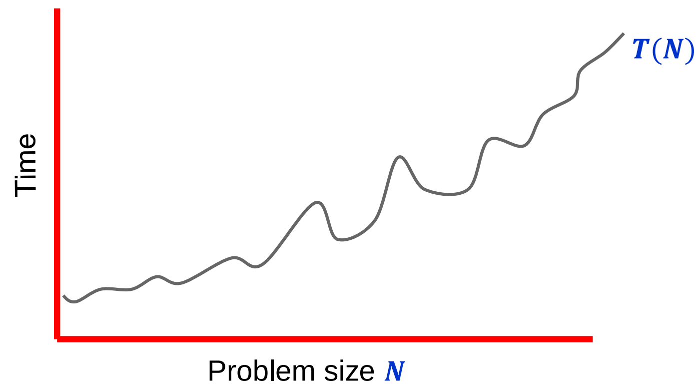
complexity analysis

- Problem size N
  - **Worst-case complexity**: max # steps algorithm takes on any input of size N
  - **Average-case complexity**: average # steps algorithm takes on inputs of size N

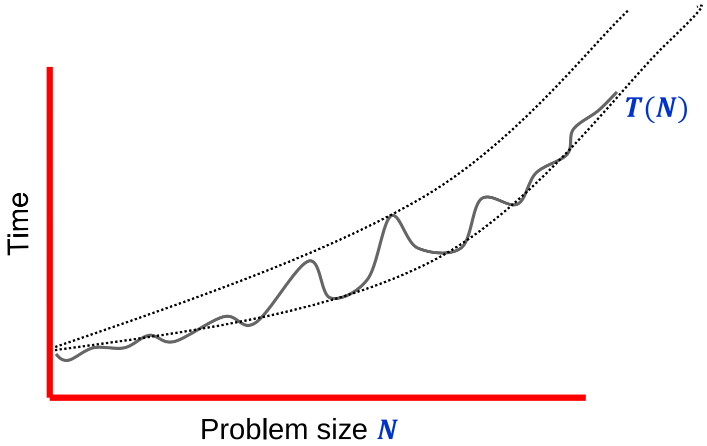
complexity

- The complexity of an algorithm associates a number $T(N)$, the worst/average-case/best time the algorithm takes, with each problem size N.
- Mathematically,
  - $T: \mathbb{N} \to \mathbb{R}_{\geq 0}$ is a function that maps positive integers giving problem size to positive real numbers giving number of steps

complexity



complexity



asymptotic growth rates

Given two functions $f, g : \mathbb{N} \rightarrow \mathbb{R}_+$

- $f(n)$ is $O(g(n))$ iff there is a constant $c > 0$ so that $f(n)$ is eventually always $\leq c \cdot g(n)$

  Upper bounds $\leq$

- $f(n)$ is $\Omega(g(n))$ iff there is a constant $c > 0$ so that $f(n)$ is eventually always $\geq c \cdot g(n)$

  Lower bounds $\geq$

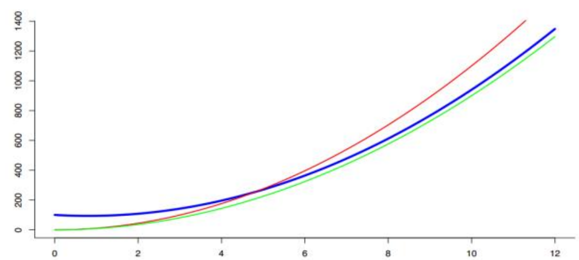
- $f(n)$ is $\Theta(g(n))$ iff it is both $O(g(n))$ and $\Omega(g(n))$ $\approx$

little-o

Given two functions $f, g : \mathbb{N} \rightarrow \mathbb{R}_+$

example

Show that $10n^2 - 16n + 100$ is $\Omega(n^2)$



asymptotic bounds for polynomials

$p(n) = a_0 + a_1 n + \cdots + a_d n^d$ **is** $\Theta(n^d)$ **if** $a_d > 0$

asymptotics of...

$$\sum_{i=1}^{n} i$$

properties

transitivity

additivity

logarithmic vs. polynomial vs. exponential

$$\log_{\mathrm{b}}(n) = o(n^a) = o(c^n)$$

for all constants a, b, and c

scaling

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.

	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

polynomial time = "efficient"

P = class of problems solvable by algorithms running in **polynomial time**, i.e. $O(n^d)$ for some constant d

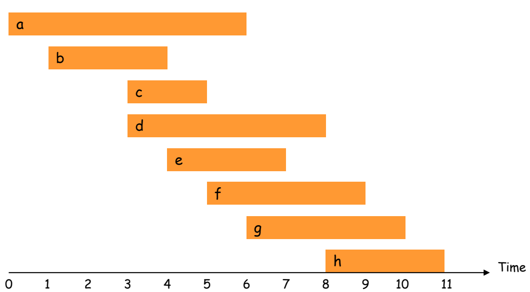
scaling: When input size doubles, running time increases by a constant factor.

vs exponential

interval scheduling

- Input. Set of jobs with start times and finish times.
- Goal. Find maximum cardinality subset of mutually compatible jobs.

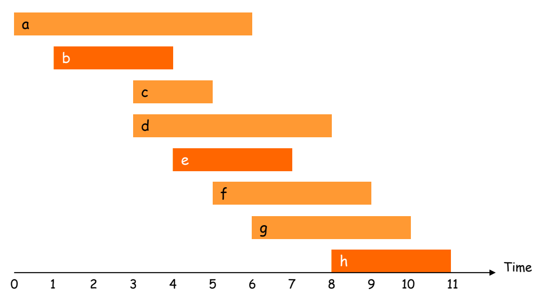
jobs don't overlap



interval scheduling

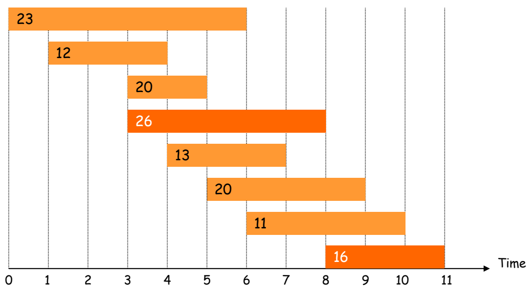
- Input. Set of jobs with start times and finish times.
- Goal. Find maximum cardinality subset of mutually compatible jobs.

jobs don't overlap



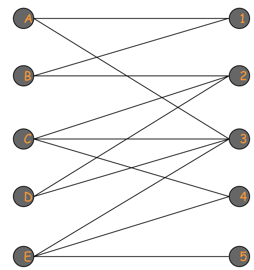
weighted interval scheduling

- Input. Set of jobs with start times, finish times, and weights.
- Goal. Find maximum weight subset of mutually compatible jobs.



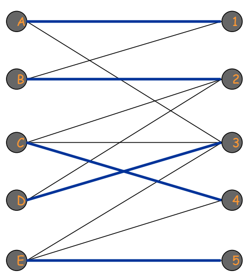
bipartite matching

- Input. Bipartite graph.
- Goal. Find maximum cardinality matching.



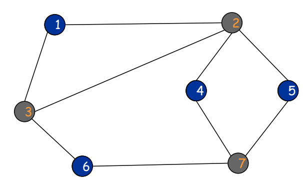
bipartite matching

- Input. Bipartite graph.
- Goal. Find maximum cardinality matching.



independent set

- Input. Graph.
- Goal. Find maximum cardinality independent set.



representative problems

- Variations on a theme: independent set.

- Interval scheduling: $O(n \log n)$ greedy algorithm.
- Weighted interval scheduling: $O(n \log n)$ dynamic programming algorithm.
- Bipartite matching: $O(n^k)$ max-flow based algorithm.
- Independent set: NP-complete.
- Competitive facility location: PSPACE-complete (see book)