Theory of Computer Algorithms (4005-800-01): Introduction

Prof. Richard Zanibbi

Course Administrative Information

Course Web Page

http://www.cs.rit.edu/~rlaz/algorithms20082

Course Syllabus and Tentative Schedule

Available from course web pages; handout

Resources

Additional textbooks/references/URLs listed on course web pages; some books to be placed on 2hr reserve in Watson library

Why and How do We Study Algorithms?

How we study algorithms: Space and Time

The Time-Space Tradeoff

Generally speaking, the more space used to store information, the less time needed to compute desired information, and vice versa.

- Simple example: table lookup for exponents, vs. function computing exponents iteratively
- Depending on the problem, at times we also want to consider other resources (e.g. power on mobile devices)

Emphasis for Algorithm Analysis: Worst-Case Time Requirements

Why emphasize time?

In general, unless the space requirements are truly excessive, we want the *fastest* algorithm

- Analyses related to time often generalize quite directly to space and other resource types
- Note: while fast algorithms are useful, speed is not the *only* criteria for selecting an algorithm

Worst-case? Why so pessimistic?

Worst-case analyses are useful in practice and provide *guarantees*

A Benchmark: Brute-Force Search

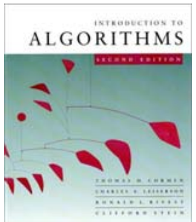
Operation

Brute-force search enumerates the set of possible solutions, and checks each one

- e.g. to sort a list of numbers, generate all permutations until a sorted list is found
- Note: while inelegant and inefficient, brute-force search is correct, making it a useful for understanding and comparison

Our Goal

Preserve correctness, while finding faster solutions; apply knowledge to produce more *informed* algorithms



Why study algorithms and performance?

- Algorithms help us to understand ***scalability***.
- Performance often draws the line between what is feasible and what is impossible.
- Algorithmic mathematics provides a ***language*** for talking about program behavior.
- Performance is the ***currency*** of computing.
- The lessons of program performance generalize to other computing resources.
- Speed is fun!

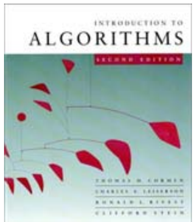
Measuring Algorithm Performance

T(n)

The time, in "primitive" computations needed by an algorithm for input size *n* (normally worst-case analysis)

- Primitive computations might be: assembly instructions, a line of C/Java, a basic operation in pseudo code (e.g. assignment, addition)
- Goal: machine-independent performance measure

R·I·T



Kinds of analyses

Worst-case: (usually)

- $T(n)$ = maximum time of algorithm on any input of size n.

Average-case: (sometimes)

- $T(n)$ = expected time of algorithm over all inputs of size n.
- Need assumption of statistical distribution of inputs.

Best-case: (bogus)

- Cheat with a slow algorithm that works fast on *some* input.

Efficient Algorithms

Polynomial Time Algorithm

If there exist constants c and d (c, d > 0) such that running time is bounded by cn^d for all input sizes (n)

Efficient Algorithm (Page 33 K&T)

An algorithm is *efficient* if it has a polynomial running time

- This measure (normally) reflects efficiency of algorithms and *tractability* of problems in practice
- Problems with polynomial solutions usually require low order polynomials (e.g. n, n^2, n^3)

R·I·T

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

Example: The Problem of Stable Matching

(Chapter 1.1, K&T)

Problem Definition

Task

Design a procedure to match individuals in two sets that is *self-enforcing* (i.e. stable)

Example

Matching job applicants to employers (e.g. for co-op positions)

Problem Formulation

Issues in the General Case

Asymmetric matching: companies need multiple employees, applicants need one job

Sizes: we may have a different sized sets

Simplification

Sets to be matched are of the same size, and each individual is in exactly one match (pair)

All individuals have a *preference list* ranking matches with individuals of the other set

Matches

Perfect Match

All individuals in both sets (e.g. men and women) are paired with a unique partner

Stable Match (Our Goal)

A matching that is perfect and stable, where no two individuals mutually prefer to leave their matching in order to join together

An instability: m and w' each prefer the other to their current partners.

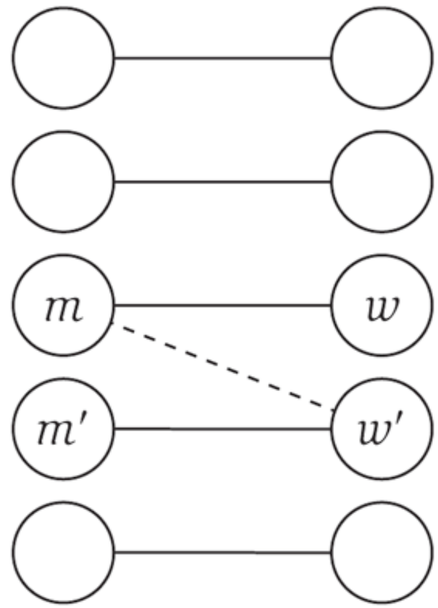


Figure 1.1 Perfect matching S with instability (m, w').

A Solution: Gale-Shapley Algorithm

```
Initially all m ∈ M and w ∈ W are free
While there is a man m who is free and hasn't proposed to
every woman
   Choose such a man m
   Let w be the highest-ranked woman in m's preference list
      to whom m has not yet proposed
   If w is free then
      (m, w) become engaged
   Else w is currently engaged to m'
      If w prefers m' to m then
         m remains free
      Else w prefers m to m'
         (m, w) become engaged
         m' becomes free
      Endif
   Endif
Endwhile
Return the set S of engaged pairs
```

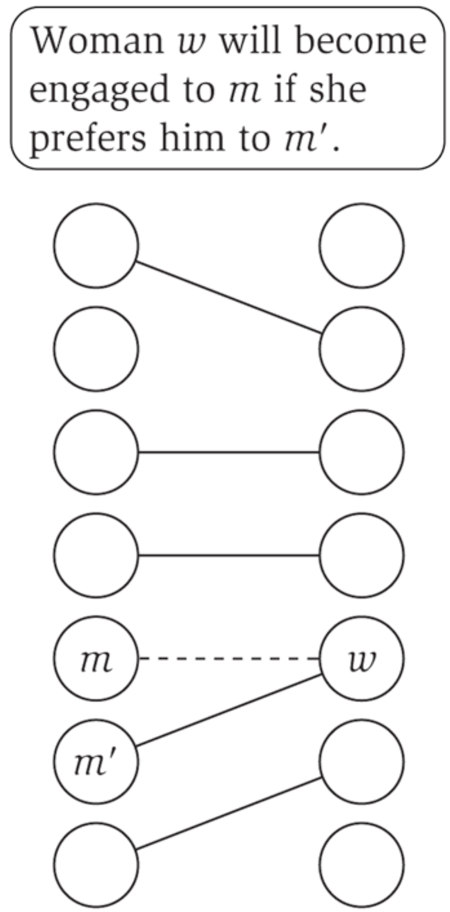


Figure 1.2 An intermediate state of the G-S algorithm when a free man m is proposing to a woman w.

Properties of the G-S Algorithm

Upper Bound on Worst-Case Run-time (*Performance*)

(1.3) G-S Terminates after at most n^2 iterations of the while loop

Match Properties (*Correctness*)

(1.1) Each woman remains engaged from their first proposal, and the sequence of partners to which she becomes engaged gets better and better (according to her preference list)

(1.4) If m is free at some point, then there is a woman to whom he has not proposed

(1.5) The set S returned at termination is a perfect matching

(1.6) The set S returned at termination is a stable matching

Additional Properties

(1.7) Every execution of the G-S algorithm yields the same matching S*, where S* = {(m, best(m)) : m ε M)}.

(1.8) In stable matching S*, each woman is paired with her worst valid partner

valid partner: partner in a stable matching

best valid partner: highest ranked partner in a stable matching

R·I·T

Overview: Five Representative Problems

(Chapter 1.2 K&T)

I. Interval Scheduling Problem

Task

Assign a resource (e.g. lecture hall) to requests with known time intervals, maximizing the number of satisfied requests that do not conflict

Solved By

Greedy algorithm: sort requests, then single pass produces solution

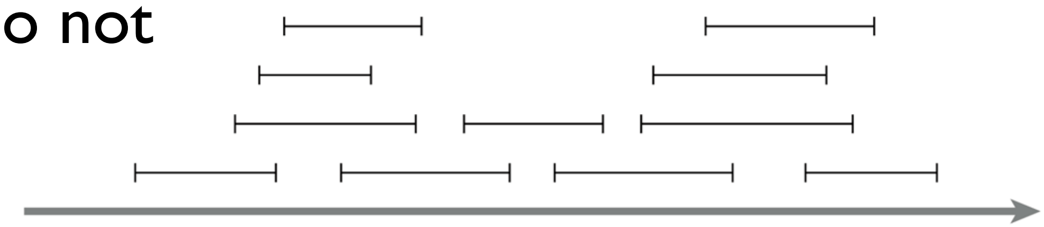


Figure 1.4 An instance of the Interval Scheduling Problem.

R·I·T

2. Weighted Interval Scheduling Problem

Modification

Requests have an associated weight $v_i > 0$. New goal is to maximize weight of satisfied requests that do not conflict

Special Case

If for all i $v_i = 1$, instance of regular Interval Scheduling problem

Solved By (*not* greedy alg!)

Dynamic Programming: build optimal value over all possible solutions using an efficient table-based strategy

R·I·T

3. Bipartite Matching Problem

Task

Find a the largest set of edges producing disjoint pairs of nodes in a bipartite graph

- e.g. each x paired with a unique y

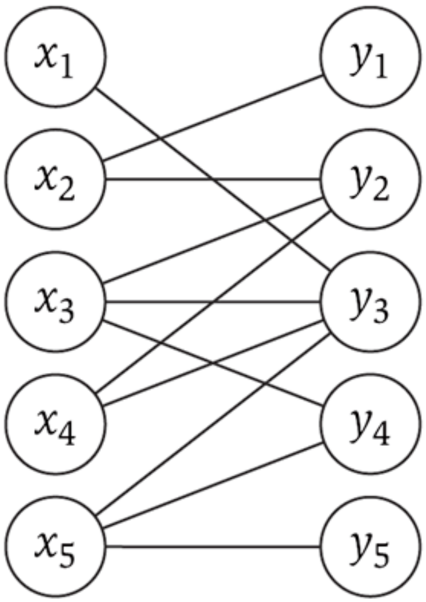


Figure 1.5 A bipartite graph.

Solution *(not greedy or dynamic!)*

Augmentation: Inductively construct larger and larger matchings with backtracking *(used in network flow problems)*

4. Independent Set Problem

Task

Identify max no. of nodes not joined by an edge in a graph

- Edges represent 'conflicts'
- Interval scheduling, bipartite matching instances of I.Set problem

Solution (general case)

No efficient algorithm believed to exist (can use brute force). Problem is NP-complete

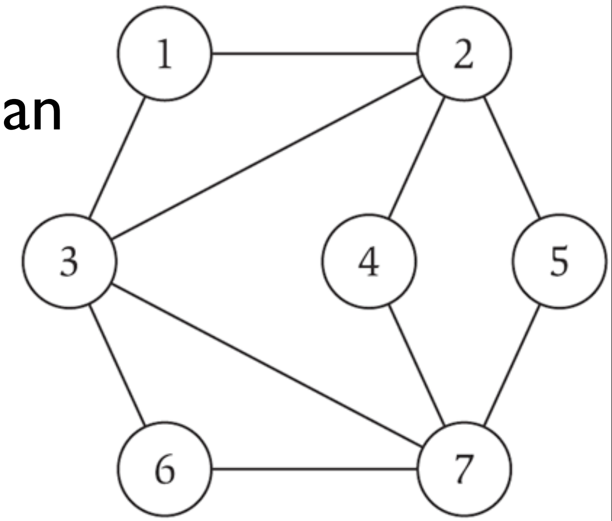


Figure 1.6 A graph whose largest independent set has size 4.

R·I·T

5. Competitive Facility Location

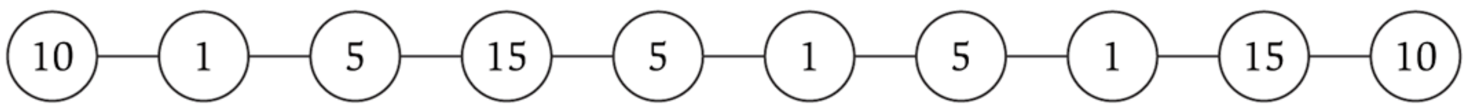


Figure 1.7 An instance of the Competitive Facility Location Problem.

Task

Two companies take turns selecting locations (nodes) forming an independent set, trying to maximize the value of selected nodes. Is there a strategy for player 2 guaranteeing a node set with at least value B?

(Comp. Facil. Location, Cont'd)

Solution

No short 'proof' for a solution; requires detailed case analysis (i.e. game traces); problem is PSPACE-complete (believed harder than NP-complete problems)

Many game playing and planning problems belong to PSPACE

Asymptotic Order of Growth

(Section 2.2, K&T)

Asymptotic Complexity

Definition

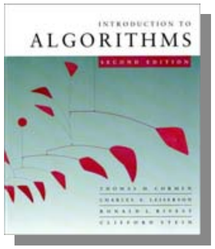
Characterizes worst-case running time of an algorithm in the *limit*, i.e. as input size goes to infinity

- Rate of growth defined as proportional to some function *f(n)* (i.e. within a constant multiple of *f(n)*)
- *f(n)* normally simple (e.g. n^2), not a detailed characterization such as: $1.62n^2 + 3.5n + 8$
- Consider upper (big 'O'), lower (Ω), and tight (Θ) bounds

(One) Purpose

R·I·T

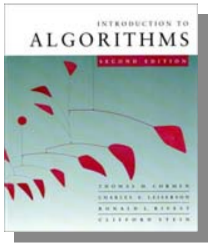
Identify sets of algorithms with similar behavior



Asymptotic notation

O-notation (upper bounds):

We write $f(n) = O(g(n))$ if there exist constants $c > 0$, $n_0 > 0$ such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$.

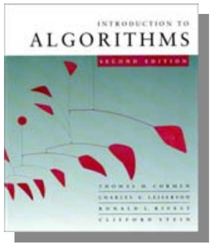


Asymptotic notation

O-notation (upper bounds):

> We write $f(n) = O(g(n))$ if there exist constants $c > 0$, $n_0 > 0$ such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$.

EXAMPLE: $2n^2 = O(n^3)$ $\quad$ ($c = 1$, $n_0 = 2$)



Asymptotic notation

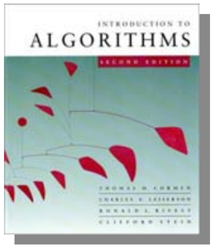
O-notation (upper bounds):

> We write $f(n) = O(g(n))$ if there exist constants $c > 0$, $n_0 > 0$ such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$.

EXAMPLE: $2n^2 = O(n^3)$ $\quad (c = 1, n_0 = 2)$

functions, not values

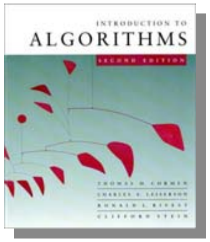
funny, "one-way" equality



Set definition of O-notation

$O(g(n)) = \{ f(n)$: there exist constants $c > 0$, $n_0 > 0$ such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0 \}$

EXAMPLE: $2n^2 \in O(n^3)$



Macro substitution

Convention: A set in a formula represents an anonymous function in the set.

EXAMPLE:

$$f(n) = n^3 + O(n^2)$$

means

$$f(n) = n^3 + h(n)$$

for some $h(n) \in O(n^2)$.

Macro substitution

Convention: A set in a formula represents an anonymous function in the set.

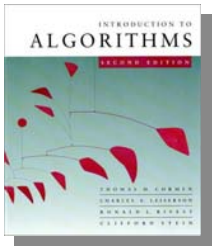
EXAMPLE: $n^2 + O(n) = O(n^2)$

means

for any $f(n) \in O(n)$:

$$n^2 + f(n) = h(n)$$

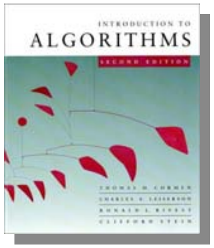
for some $h(n) \in O(n^2)$.



Ω-notation (lower bounds)

O-notation is an *upper-bound* notation. It makes no sense to say $f(n)$ is at least $O(n^2)$.

$\Omega(g(n)) = \{ f(n)$: there exist constants $c > 0$, $n_0 > 0$ such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$ $\}$

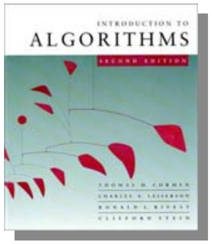


Ω-notation (lower bounds)

O-notation is an *upper-bound* notation. It makes no sense to say $f(n)$ is at least $O(n^2)$.

$$\Omega(g(n)) = \{ f(n) : \text{there exist constants } c > 0,\ n_0 > 0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0 \}$$

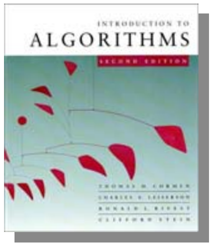
EXAMPLE: $\sqrt{n} = \Omega(\lg n)$ $\quad (c = 1,\ n_0 = 16)$



Θ-notation (tight bounds)

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$$

f(n) = Θ(g(n)) means that both f(n) = O(g(n)) and f(n) = Ω(g(n))

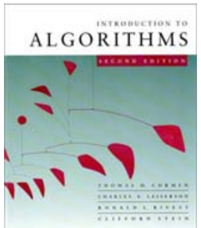


Θ-notation (tight bounds)

$$\Theta(g(n)) = O(g(n)) \ \cap \ \Omega(g(n))$$

EXAMPLE: $\frac{1}{2}n^2 - 2n = \Theta(n^2)$

f(n) = Θ(g(n)) means that both f(n) = O(g(n)) and f(n) = Ω(g(n))



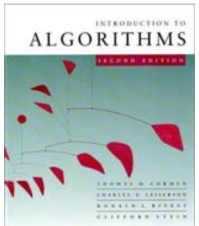
Θ-notation

Math:

$\Theta(g(n)) = \{ f(n) :$ there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0 \}$

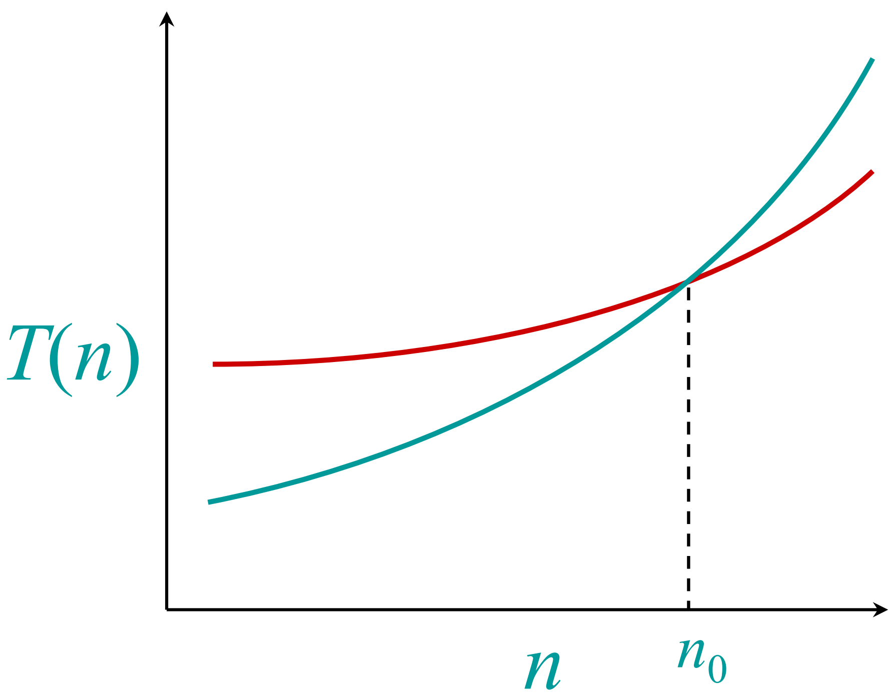
Engineering:

- Drop low-order terms; ignore leading constants.
- Example: $3n^3 + 90n^2 - 5n + 6046 = \Theta(n^3)$



Asymptotic performance

When n gets large enough, a $\Theta(n^2)$ algorithm *always* beats a $\Theta(n^3)$ algorithm.



- We shouldn't ignore asymptotically slower algorithms, however.
- Real-world design situations often call for a careful balancing of engineering objectives.
- Asymptotic analysis is a useful tool to help to structure our thinking.