CS38
Introduction to Algorithms

Lecture 1

April 1, 2014

Outline

- administrative stuff
- motivation and overview of the course

  stable matchings example

- graphs, representing graphs
- graph traversals (BFS, DFS)
- connectivity, topological sort, strong connectivity

Administrative Stuff

- Text: Introduction to Algorithms (3^{rd} Edition) by Cormen, Leiserson, Rivest, Stein
  - “CLRS”
  - recommended but not required
- lectures self-contained
- slides posted online

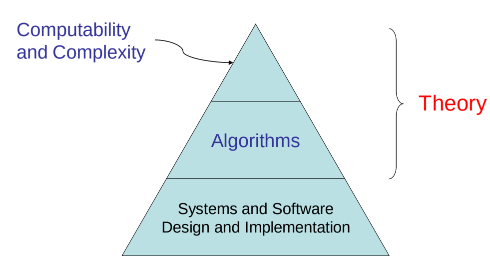
Administrative Stuff

- weekly homework
  - collaboration in groups of 2-3 encouraged
  - separate write-ups (clarity counts)
- midterm and final
  - indistinguishable from homework except cumulative, no collaboration allowed

Administrative Stuff

- no programming in this course
- things I assume you are familiar with:
  - programming and basic data structures: arrays, lists, stacks, queues
  - asymptotic notation “big-oh”
  - sets, graphs
  - proofs, especially induction proofs
  - exposure to NP-completeness

Motivation/Overview



Motivation/Overview

- at the heart of programs lie algorithms
- in this course algorithms means:
  - abstracting problems from across application domains
  - worst case analysis } main figure of merit
  - asymptotic analysis ("big-oh") } main figure of merit
  - rigorous proofs paradigm (vs. "heuristics")

Motivation/Overview

- algorithms as a key technology
- think about:
  - mapping/navigation
  - Google search
  - Shazam
  - word processing (spelling correction, layout…)
  - content delivery and streaming video
  - games (graphics, rendering…)
  - big data (querying, learning…)

Motivation/Overview

- In a perfect world
  - for each problem we would have an algorithm
  - the algorithm would be the fastest possible

What would CS look like in this world?

Motivation/Overview

- Our world (fortunately) is not so perfect:
  - for many problems we know embarrassingly little about what the fastest algorithm is
    - multiplying two integers or two matrices
    - factoring an integer into primes
    - determining shortest tour of given n cities
  - for many problems we suspect fast algorithms are impossible (NP-complete problems)
  - for some problems we have unexpected and clever algorithms (we will see many of these)

Motivation/Overview

- Two main themes:
  - algorithm design paradigms
  - algorithms for fundamental problems

    (data structures as needed)
- NP-completeness and introduction to approximation algorithms

definitions and conventions

What is a problem?

- Some examples:
  - given n integers, produce a sorted list
  - given a graph and nodes s and t, find a shortest path from s to t
  - given an integer, find its prime factors
- problem associates each input to an output
- a problem is a *function*:

$$f:\Sigma^* \rightarrow \Sigma^*$$

What is an algorithm?

- a problem is a *function*:

$$f:\Sigma^* \rightarrow \Sigma^*$$

- formally: an algorithm is a Turing Machine that computes function f
- more informal: a precisely specified sequence of basic instructions computing f
- level of detail is a judgment call

high-level description ⇔ detailed pseudo-code

Underlying model

- Officially, Random Access Machine (RAM)
  - essentially, low level programming language like assembly code
- Will not come up in this course
- We all can distinguish between, e.g.
  - x ← i-th element of array A (single step)
  - x ← minimum element of array A (not a single step)

Worst-case analysis

- Figure of merit: resource usage
  - running time (primary for this course)
  - storage space
  - others…
- Always measure resource usage via:
  - function of the input size
  - value of the fn. is the maximum quantity of resource used over all inputs of given size
  - called "worst-case analysis"

Asymptotic notation

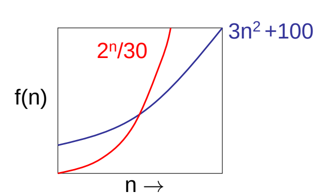
- Measure time/space complexity using asymptotic notation ("big-oh notation")
  - disregard lower-order terms in running time
  - disregard coefficient on highest order term
- example:

$$f(n) = 6n^3 + 2n^2 + 100n + 102781$$

  - "f(n) is order n^3"
  - write $f(n) = O(n^3)$

Asymtotic notation

- captures behavior for "large n"



Asymptotic notation

Definition: given functions $f,g:\mathbf{N} \rightarrow \mathbf{R}^+$, we say $f(n) = O(g(n))$ if there exist positive integers c, n_0 such that for all $n \geq n_0$

$$f(n) \leq cg(n).$$

- meaning: f(n) is (asymptotically) less than or equal to g(n)
- if $g > 0$ can assume $n_0 = 0$, by setting $c' = \max_{0 \leq n \leq n_0}\{c, f(n)/g(n)\}$

Asymptotic notation facts

each bound asymptotically less than next

- "logarithmic": O(log n)
  - $\log_b n = (\log_2 n)/(\log_2 b)$
  - so $\log_b n = O(\log_2 n)$ for any constant b; therefore suppress base when write it
- "polynomial": $O(n^c) = n^{O(1)}$
  - also: $c^{O(\log n)} = O(n^{c'}) = n^{O(1)}$
- "exponential": $O(2^{n^\delta})$ for $\delta > 0$

Why worst case, asymptotic?

- Why worst-case?
  - well-suited to rigorous analysis, simple
  - stringent requirement better
- Why asymptotic?
  - not productive to focus on fine distinctions
  - care about behavior on large inputs
  - general-purpose alg. should be scalable
  - exposes genuine barriers/motivates new ideas

Stable matchings example

Stable matchings

- Motivation:
  - n medical students and n hospitals
  - each student has ranking of hospitals
  - each hospital has ranking of students
  - Goal: match each student to a hospital
  - Goal: make the matching stable

Definition: (student x, hospital y) pair unstable if x prefers y to its match and y prefers x to its match

Stable matchings

- Captures many settings, e.g.,
  - employee/employer
  - students/dorms
  - men/women
- Usually described via men/women:
  - ranked list of n women for each of n men
  - ranked list of n men for each of n women
  - f: produce a stable matching (no unstable pairs)

Stable matchings

- Does a stable matching always exist?
- Is there an efficient algorithm to find one?

Gale-Shapley Stable Matching Algorithm
Input: ranking lists for each man, women
1. S ← empty matching
2. WHILE some man m is unmatched and hasn't proposed to every woman
3. w ← first woman on m's list to whom m has not yet proposed
4. IF w is unmatched THEN add pair (m,w) to matching S
5. ELSE IF w prefers m to her current partner m' THEN
   replace pair (m',w) with pair (m,w) in matching S
6. ELSE w rejects m

Stable matchings

- We have
  - a well-defined problem
  - a proposed algorithm
- Now we need to
  - prove correctness
  - bound running time, possibly requiring filling in implementation details

Stable matchings

Gale-Shapley Stable Matching Algorithm
Input: ranking lists for each man, women
1. S ← empty matching
2. WHILE some man m is unmatched and hasn't proposed to every woman
3. w ← first woman on m's list to whom m has not yet proposed
4. IF w is unmatched THEN add pair (m,w) to matching S
5. ELSE IF w prefers m to her current partner m' THEN
   replace pair (m',w) with pair (m,w) in matching S
6. ELSE w rejects m

Lemma: algorithm terminates with all men, women matched, and with no unstable pair

Proof: terminates?

Stable matchings

Gale-Shapley Stable Matching Algorithm
Input: ranking lists for each man, women
1. S ← empty matching
2. WHILE some man m is unmatched and hasn't proposed to every woman
3. w ← first woman on m's list to whom m has not yet proposed
4. IF w is unmatched THEN add pair (m,w) to matching S
5. ELSE IF w prefers m to her current partner m' THEN
   replace pair (m',w) with pair (m,w) in matching S
6. ELSE w rejects m

Lemma: algorithm terminates with all men, women matched, and with no unstable pair

Proof: all matched?

Stable matchings

Gale-Shapley Stable Matching Algorithm
Input: ranking lists for each man, women
1. S ← empty matching
2. WHILE some man m is unmatched and hasn't proposed to every woman
3. w ← first woman on m's list to whom m has not yet proposed
4. IF w is unmatched THEN add pair (m,w) to matching S
5. ELSE IF w prefers m to her current partner m' THEN
   replace pair (m',w) with pair (m,w) in matching S
6. ELSE w rejects m

Lemma: algorithm terminates with all men, women matched, and with no unstable pair

Proof: unstable pair (m, w)?

Stable matchings

Lemma: algorithm terminates with all men, women matched, and S containing no unstable pair

Proof:

terminates: only n^2 possible proposals, 1 per iteration

all matched: suppose not. Then some m unmatched and some w unmatched. So w never proposed to. But m proposed to everyone if ends unmatched.

Stable matchings

Lemma: algorithm terminates with all men, women matched, and S containing no unstable pair

Proof:

pair (m, w) not in S

case 1: m never proposed to w,

$\Rightarrow$ m prefers his current partner

case 2: m proposed to w

$\Rightarrow$ w rejected m (in line 6 or line 5)

in both cases (m, w) is not an unstable pair.

Stable matchings

Gale-Shapley Stable Matching Algorithm
Input: ranking lists for each man, women
1. S $\leftarrow$ empty matching
2. WHILE some man m is unmatched and hasn't proposed to every woman
3. w $\leftarrow$ first woman on m's list to whom m has not yet proposed
4. IF w is unmatched THEN add pair (m,w) to matching S
5. ELSE IF w prefers m to her current partner m' THEN replace pair (m',w) with pair (m,w) in matching S
6. ELSE w rejects m

Lemma: can implement with running time $O(n^2)$

Proof: create two arrays wife, husband

wife[m] = w if (m,w) in S, 0 if unmatched (same for husband)

Stable matchings

Gale-Shapley Stable Matching Algorithm
Input: ranking lists for each man, women
1. S $\leftarrow$ empty matching
2. WHILE some man m is unmatched and hasn't proposed to every woman
3. w $\leftarrow$ first woman on m's list to whom m has not yet proposed
4. IF w is unmatched THEN add pair (m,w) to matching S
5. ELSE IF w prefers m to her current partner m' THEN replace pair (m',w) with pair (m,w) in matching S
6. ELSE w rejects m

- implementing step 5? for each preference list pref can create inv-pref via: for i = 1 to n do inv-pref[pref[i]] = i
- w prefers m to m' iff inv-pref[m] < inv-pref[m']
- $O(n^2)$ preprocessing; O(1) time for each iteration of loop

Stable matchings

- We proved:

Theorem (Gale-Shapley '62): there is an $O(n^2)$ time algorithm that is given

n rankings of women by each of n men

n rankings of men by each of n women

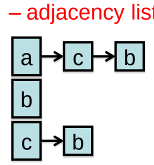
and outputs

a stable matching of men to women.

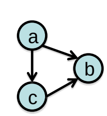
Basic graph algorithms

Graphs

- Graph G = (V, E)
  - directed or undirected
  - notation: n = |V|, m = |E| (note: $m \leq n^2$)
  - adjacency list or adjacency matrix



	a	b	c
a	0	1	1
b	0	0	0
c	0	1	0



Graphs

- Graphs model many things…
  - physical networks (e.g. roads)
  - communication networks (e.g. internet)
  - information networks (e.g. the web)
  - social networks (e.g. friends)
  - dependency networks (e.g. topics in this course)

… so many fundamental algorithms operate on graphs

Graphs

- Graph terminology:
  - an undirected graph is connected if there is a path between each pair of vertices
  - a tree is a connected, undirected graph with no cycles; a forest is a collection of disjoint trees
  - a directed graph is strongly connected if there is a path from x to y and from y to x, $\forall$ x,y$\in$V
  - a DAG is a <u>D</u>irected <u>A</u>cyclic <u>G</u>raph

Graph traversals

- Graph traversal algorithm: visit some or all of the nodes in a graph, labeling them with useful information
  - breadth-first: useful for undirected, yields connectivity and shortest-paths information
  - depth-first: useful for directed, yields numbering used for
    - topological sort
    - strongly-connected component decomposition

Breadth first search

```
BFS(undirected graph G,  starting vertex s)
for each vertex v, v.color = white, v.dist = ∞, v.pred = nil
s.color = grey, s.dist = 0, s.pred = nil
Q = ∅; ENQUEUE(Q, s)
WHILE Q is not empty u = DEQUEUE(Q)
      for each v adjacent to u
           IF v.color = white THEN
               v.color = grey, v.dist = u.dist + 1, v.pred = u
               ENQUEUE(Q, v)
           u.color = black
```

<u>Lemma</u>: BFS runs in time O(m + n), when G is represented by an adjacency list.

Proof?

Breadth first search

```
BFS(undirected graph G,  starting vertex s)
for each vertex v, v.color = white, v.dist = ∞, v.pred = nil
s.color = grey, s.dist = 0, s.pred = nil
Q = ∅; ENQUEUE(Q, s)
WHILE Q is not empty u = DEQUEUE(Q)
      for each v adjacent to u
           IF v.color = white THEN
               v.color = grey, v.dist = u.dist + 1, v.pred = u
               ENQUEUE(Q, v)
           u.color = black
```

<u>Lemma</u>: BFS runs in time O(m + n), when G is represented by an adjacency list.

Proof: each vertex enqueued at most 1 time; its adj. list scanned once; O(1) work for each neighbor

BFS example from CLRS