2/3/15

6.046 Lecture #1

Fernando Trujano

Design and Analysis of Algorithms

- Sign up for a recitation section on stellar!

Modules:

- Divide and Conquer
- Optimization: greedy, dynamic programming
- Network flow
- Intractability · approximation
- Advanced topics

Very similar problems can have very different complexity

Recall: P : class of problems solved in polynomial time $O(n^k)$ for some consta

NP : " " verifiable " "

NP-Complete : in NP and as hard as any problem in NP

Interval Scheduling

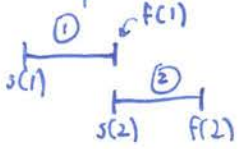
single resource

Resources and requests $1, \ldots, n$

$s(i)$ start time
$f(i)$ finish time $\quad s(i) < f(i)$

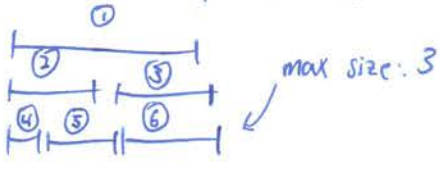
can satisfy with single resource simultaniously

Two requests i and j are compatible if they don't overlap. $\quad f(i) \leq s(j)$ or $f(j) \leq s(i)$



Goal: select compatible subset of requests that is of maximum size.

Ex



Claim: We can solve this problem using a greedy algorithm.

↑ maximize now, don't look ahead.

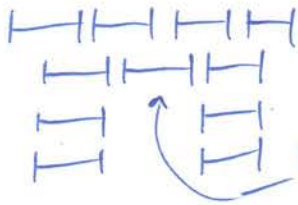
A greedy algorithm is a myopic algorithm that processes input one piece at a time with no apparent look ahead

Greedy Interval Scheduling

1. Use a simple rule to select a request i
2. Reject all requests incompatible with i
3. Repeat until all requests are processed

Heuristic:

- each request Find # incompatible requests. select the one with the min # incompatibles.



- select R(i) with earliest finish time. Scan throw F(i), pick min compatible.

↳ Always works → but need to prove with mathematical rigor.

Claim: Given a list of intervals L, greedy algorithm with earliest Finish time produces k^* intervals, where k^* is maximum.

Proof: By induction on k^*

Base Case: $k^* = 1$ Any interval works

Inductive step:

Suppose the claim holds For k^* and we are given a list of F intervals whose optimal schedule whose optimal schedule has k^*+1 intervals

$$S^*[1, 2, \ldots k^*+1] = \langle S(j_1)F(j_1)\rangle, \langle S(j_{k^*+1}), F(j_{k^*+1})\rangle$$

$$S[1, \ldots k] = \langle S(i_1), F(i_1)\rangle, \langle S(i_k), F(i_k)\rangle$$

k and k^* not comparable $\quad F(i_1) \leq F(j_1)$

$$S^{**} = \langle S(i_1)F(i_1)\rangle, \langle S(j_2, F(j_2)\rangle, \ldots \langle S(j_{k^*+1}), F(j_{k^*+1})\rangle$$

S^{**} is also optimal.

Define L' = set of intervals $S(i) \geq F(i_1)$

Since S^{**} is optimal For L, $S^{**}[2, \ldots k^*+1]$ is optimal For L

↳ subsets of optimal solution must be optimal

∴ optimal schedule for L' has k^* size

By inductive hypothesis, run the greedy algorithm on L' should produce a schedule of size k^*

By constructure, the greedy alg on L' gives $S[2..k]$ of size $k-1 = k^*$

so $S[1...k]$ of size k^*+1

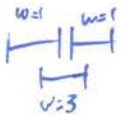
optimal because $k = k^*+1$

□

Weighted Interval Scheduling

each request has weight $w(i)$

Schedule subset of requests with maximum weights



↳ earliest time won't work now!

Dynamic Programing

$$R^{\geq x} = \{ \text{request } j \in R \mid s(j) \geq x \}$$

subproblems

$\forall\, x = f(i)$ $\quad R^{f(i)}$ requests later than $f(i)$

n: # of requests in original problem

subproblems $= n$

solve each subproblem once and memoize

subproblems $\cdot$ time/subproblem

(assume $O(1)$ for lookups)

DP Guessing

Try each request i as a possible first request

$$opt(R) = \max_{1 \leq i \leq n} \left(w_i + opt\left(R^{f(i)}\right) \right)$$

complexity $O(n^2)$ ✓

One small change can change the whole problem

non-identical machines

$T = \{T_1, \ldots T_m\}$

weight = 1 For all requests

BUT $Q(i) \subseteq T$ is a set of machines that i runs on.

↳ so some requests can only run on some machines.

This problem is NP-complete.

↳ can $k \leq n$ requests be scheduled.

6.046 Lecture #2

Fernando Trujan

Paradigm
Convex Hull
Median Finding

Divide and Conquer

Paradigm : Given a problem of size n. Divide it into "a" subproblems of size n/b $b>1$. ($a \geq 1$)
Solve each subproblem recursively.
Combine solutions of subproblems into overall solution. ("merge")

$T(n) = aT(n/b) + [\text{work for merge}]$

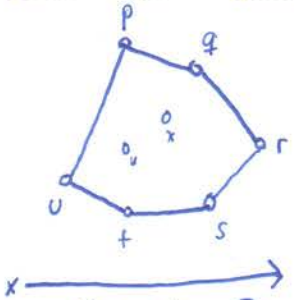
Convex Hull (2D)

Given n points in plane

$$S = \{ (x_i, y_i) \mid i = 1, 2, \ldots n \}$$

assume no two have the same x or y coord, no three in same line

Convex Hull - smallest convex polygon containing all points in S CH(S)



Sequence of points on the boundry in clockwise order as doubly linked list.

$p \leftrightarrow q \leftrightarrow r \leftrightarrow s \leftrightarrow t \leftrightarrow u \leftrightarrow p$

Brute Force?

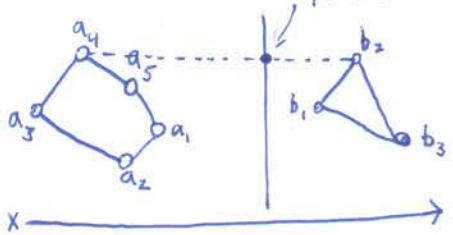
- Draw a line between any two points — $O(n^2)$ segments
- check if all points are in one of the halves — $O(n)$ test complexity
- if yes → segment is part of convex hull

$\Rightarrow O(n^3)$

Divide and Conquer

- Sort the points by x-coords
- For input set S
  Divide into left half A and rightside B by x-coords
  compute CH(A) and CH(B)
  Combine.

How to merge?



$y(i,j)$ = upper tangent → max

Pairwise generation of segm

Obvious merge algorithm: Look at all pairs of points $\Rightarrow \Theta(n^2)$

Two-Finger and string algorithm:

```
i=1
J=1
while ( y(i, j+1) > y(i,j)  or  y(i-1, j) > y(i, j) ):
    if y(i, j+1) > y(i,j):   # move right finger
        j = J+1 (mod q)
    else:
        i = i-1 (mod p)
return (a_i, b_j) as upper tangent
```

Complexity:

On every iteration one of the counters will update

$\Rightarrow \Theta(n)$ merge.

Overall Complexity: $T(n) = aT(\frac{n}{2}) + \Theta(n)$

$= \Theta(n \log n)$ ✓

cut and paste? - see notes

Median Finding

Median finding in better than $\Theta(n \log n)$ time.

Given a set of n numbers, define rank of (x) as #'s in the set that are $\leq x$

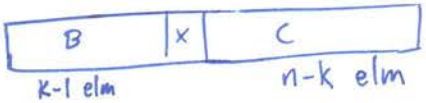
Find element of rank $\lfloor \frac{n+1}{2} \rfloor$ lower median $\lceil \frac{n+1}{2} \rceil$ upper median

select$(S, \overset{\text{rank}}{i})$

- ~~Divide~~ Pick $x \in S$
- compute $k = \text{rank}(x)$

$B = \{y \in S \mid y < x\}$

$C = \{y \in S \mid y > x\}$



$\Theta(n^2)$ if bad selection → inbalanced B and C

if $k = i$: return x
else if $k > i$: return select(B, i)
else if $k < i$: return select$(C, i-k)$

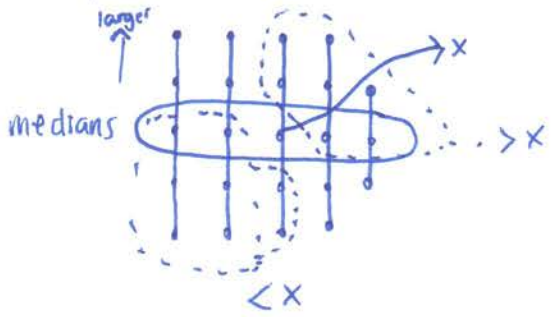
?

Pick x cleverly

Arrange S into columns of size 5 ($\lceil \frac{n}{5} \rceil$ cols)

Sort each col (big elms in top) linear time (since only 5)

Find "median of medians" of x



How many elements are guaranteed to be $> x$?

Half of the $\lceil \frac{n}{5} \rceil$ groups contribute at least

3 elements $> x$ except for 1 group with less than 5 elements and 1 group that contains x

At least $3(\lceil \frac{n}{10} \rceil - 2)$ elements are $> x$

" " $< x$

Recurrence:

$$T(n) = \begin{cases} O(1) \text{ for } n \le 140 \\ T(\lceil \frac{n}{5} \rceil) + T(\lceil \frac{7n}{10} + 6 \rceil) + \underbrace{\Theta(n)}_{\text{sorting of all columns}} \end{cases}$$

$T(\lceil \frac{n}{5} \rceil)$: finding median of medians

$T(\lceil \frac{7n}{10} + 6 \rceil)$: discard $\frac{3n}{10} - 6$ elements

2/6/15

6.046 Recitation #1

Fernando Trujano

Deepak Narayanan deepakn@mit.edu OH: Wed 7-9 pm

- Weighted Interval Scheduling
- Divide and conquer (Matrix multiplication)
- Master's Theorem

Weighted Interval Scheduling

n requests, each associated with start time, end time and a weight

Want to maximize the total weight of intervals picked

$O(n^2)$ Dynamic Programming Solution

$R^x = \{ \text{request } j \in R \mid s(j) \geq x \}$

If $x = f(i)$, then $R^{f(i)}$ represents the set of requests that follow i

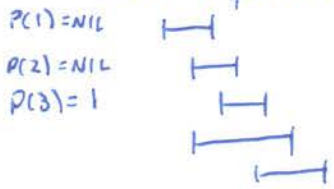
$R^{f(i)}, R^{f(f(i))}, R^{f(3)} \ldots R^{f(n)}$

$$opt(R) = \max_{1 \leq i \leq n} \{ w_i + opt(R^{f(i)}) \}$$

subproblems: n
time/problem: $O(n)$
Total: $O(n^2)$ ← not that great :-

$O(n \log n)$ Solution

Sort requests according to Finish time → $O(n \log n)$



$f(1) < f(2) < \ldots < f(n)$

$P(j)$ = largest index $i < j$ such that i and j are compatible

$M[i]$ → max weight for requests $\{1, 2, 3 \ldots i\}$

$$M[i] = \max \{ w_i + M[p(i)], M[i-1] \}$$

$w_i + M[p(i)]$ ← i belongs in optimal configuration

$M[i-1]$ ← does not

subproblems: n
time/problem: $O(1)$
total (for this step): $O(n)$

Overall Total:
$O(n) + O(n \log n)$
$= O(n \log n)$

Divide and Conquer

Matrix Multiplication

Square matrices A and B, compute $C = A \cdot B$

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

$\underbrace{\qquad\qquad}_{O(n)}$

Total runtime of naive $O(n^3)$ ← not good!

Strassen's Algorithm

$A, B, C \rightarrow 2^k \times 2^k$ ← $2^{k-1} \times 2^{k-1}$

Divide matrices: $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ $C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$ $B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$

$$C_{11} = A_{11}B_{11} + A_{12}B_{12}$$
$$C_{12} = A_{11}B_{12} + A_{22}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$
$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

} math

$\underbrace{\qquad\qquad}$ $\Theta(n^2)$ since n^2 terms in each of these matrices

$$T(n) = 8T(n/2) + \Theta(n^2)$$

$$T(n) = \Theta(n^{\log_2 8}) = \Theta(n^3)$$ ← wtf? still the same ∴

But wait....

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$M_2 = (A_{21} + A_{22})B_{11}$$
$$M_3 = A_{11}(B_{12} - B_{22})$$
$$M_4 = A_{22}(B_{21} - B_{11})$$
$$M_5 = (A_{11} + A_{12})B_{22}$$
$$M_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$
$$M_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

↑ now we only have 7 multiplications!

magic math →

$$C_{11} = M_1 + M_4 - M_5 + M_7$$
$$C_{12} = M_3 + M_5$$
$$C_{21} = M_2 + M_4$$
$$C_{22} = M_1 - M_2 + M_3 + M_6$$

more additions

$$T(n) = 7T(n/2) + \Theta(18n^2)$$

$$T(n) = \Theta(n^{\log_2 7}) = \Theta(n^{2.807})$$

YAY slightly better

Master's Theorem

$$T(n) = aT\left(\frac{a}{b}\right) + \Theta(f(n))$$

Case I: leaves dominate

$$f(n) = \Theta(n^{\log_b a})$$

$$T(n) = \Theta(n^{\log_b a})$$

To prove, look at recursion tree

Case II: About equal

$$f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$$

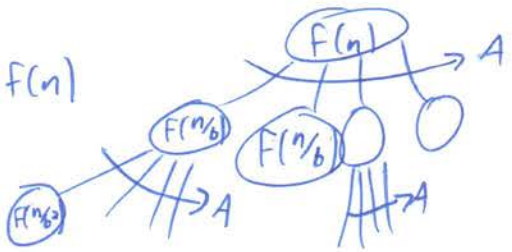
$$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$$?

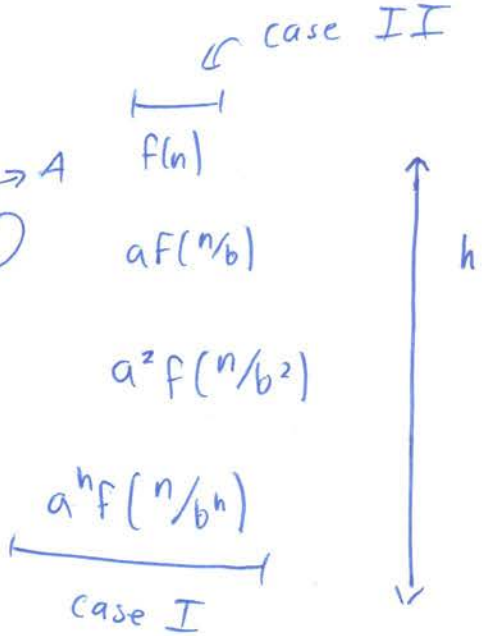
Case III: $f(n)$ dominates

$$f(n) = \Theta(n^{\log_b a + \epsilon})$$

$$T(n) = \Theta(f(n))$$

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$





2/12/15

6.046 Lecture

Fernando Trujano

- Fast Fourier Transform FFT
  - polynomial operations vs representations
  - divide and conquer algorithms
  - collapsing samples / roots of unity
  - FFT, IFFT and polynomial multiplication

Polynomial

$$A(x) = a_0 + a_1 x + a_2 x^2 + \dots + a_{n-1} x^{n-1}$$
$$= \sum_{k=0}^{n-1} a_k x^k$$
$$= \langle a_0, a_1, \dots, a_{n-1} \rangle$$ ← vector notation just coefficients

Operations on polynomials

① evaluation : $A(x)$ and $x_0 \to A(x_0)$

Horner's Rule:

$$A(x) = a_0 + x(a_1 + x(a_2 + \dots + x(a_{n-1})\dots)) \Rightarrow O(n) \text{ time}$$

② Addition : $A(x)$ and $B(x) \to C(x) = A(x) + B(x) \quad \forall x$

$$C_k = a_k + b_k$$

③ Multiplication : $A(x), B(x) \to C(x) = A(x)B(x)$

$$(a_0 + a_1 x + a_2 x^2 + \dots)\cdot(b_0 + b_1 x + b_2 x^2 + \dots) = a_0 b_0 + (a_1 b_0 + a_0 b_1)x + \dots$$

$$C_k = \sum_{j=0}^{k} a_j b_{k-j} \Rightarrow O(n^2) \text{ :(}$$

TODAY : $O(n \log n)$

convolution of vectors A and reverse(B)

(inner products of all possible shifts)

Representations

Ⓐ coeff. vector : $a_0 + a_1 x + a_2 x^2 + \dots$

Ⓑ roots : $r_0, r_1, \dots, r_{n-1} \quad P = c(x-r_0)(x-r_1)\dots(x-r_{n-1})$

Ⓒ Samples (x_k, y_k) for $k = 0, 1, \dots n-1$

$A(x_k) = y_k \quad x_k$'s distinct

Algorithms	(A) Coeffs	(B) roots	(C) samples
(1) Evaluation	$O(n)$	$O(n)$	$O(n^2)$
(2) Addition	$O(n)$	∞	$O(n)$
(3) Multiplication	$O(n^2)$	$O(n)$	$O(n)$

$O(n \lg n)$

(convert coeffs $\longleftrightarrow$ samples in $n \lg n$ using FFT

Matrix View → Vandermonde Matrix

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & & & & \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{pmatrix}$$

$V_{jk} = x_j^k$

coeffs → samples = $V \cdot A$ $O(n^2)$

samples → coeffs = $V \backslash A$

- Gaussian elimination $O(n^3)$
- $V^{-1} \cdot A \rightarrow O(n^2)$

Divide and Conquer algorithm

goal: $A(x)$ For $x \in X$

(1) divide into even and odd coefficients

$$A_{even}(x) = \sum_{k=0}^{n/2} a_{2k} x^k = \langle a_0, a_2, a_4, \ldots \rangle \qquad a_{2k} x^k$$

$$A_{odd}(x) = \sum a_{2k+1} x^k = \langle a_1, a_3, a_5 \ldots \rangle$$

(2) Conquer: recursively compute $A_{even}(Y)$ and $A_{odd}(Y)$ For $Y \in X^2 = \{ x^2 \mid x \in X \}$

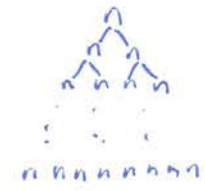
(3) Combine

$$A(x) = A_{even}(x^2) + x A_{odd}(x^2) \quad \text{for } x \in X$$

↩ Just algebra

$$T(n, |X|) = 2 \cdot T(n/2, |X|) + O(n + |X|)$$

$|X|$ never gets smaller



??? ↓

collapsing set X if

$|X^2| = |X|/2$ and X^2 is collapsing

or $|X| = 1$

Every # has exactly 2 $\sqrt{}$

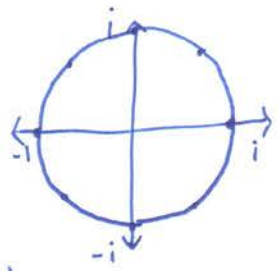
$|X|2$: $X = \{1\}$

$|X|=2$: $X = \{-1, 1\}$

$|x|=4$ $X = \{i, -i, -1, 1\}$

$|x|=8$ $X = \{\pm\frac{\sqrt{2}}{2}(1+i), \pm\frac{\sqrt{2}}{2}(1-i), i, -i\}$

n^{th} roots of unity



$(\cos\theta, \sin\theta)$

$= \cos\theta + i\sin\theta$

$= e^{i\theta}$

For $\theta = 0, T/n, 2T/n, \ldots, \frac{n-1}{n}T$ $\quad T = 2\pi$

$(e^{i\theta})^2 = e^{i(2\theta)} = e^{i(2\theta \bmod T)}$

$e^{i\,kT/n} = e^{i(2k)\,T/n}$ $\quad \boxed{e^{iT} = 1}$

$n = 2^k \Rightarrow n^{th}$ roots of unity collabsing

$X_k = e^{ik\,T/n}$

Fast Forier Transform (FFT)

= Divide and conquer algorithm Fod DFT (discrete)

Discrete Forier Transfor (DFT)

= V·A for $X_k = e^{ik\,T/n}$

$V_{jk} = x_j{}^k = e^{ijk(T/n)}$

Fast polynomial multiplication

$A^* = FFT(A)$ A coef -> sample

$B^* = FFT(B)$

$C_k^* = A_k^* B_k^* \; \forall k$

$C = IFFT(C^*)$

3/4

Claim $V^{-1} = \bar{V}/n$

$$P = V \cdot \bar{V} = nI = \begin{pmatrix} n & & 0 \\ & \ddots & \\ 0 & & n \end{pmatrix}$$

$$P_{jk} = (\text{row } j \text{ of } V) \cdot (\text{col } k \text{ of } \bar{V})$$

$$= \sum_{m=0}^{n-1} e^{i\tau jm/n} \cdot e^{-i\tau mk/n}$$

$$= \sum_{m=0}^{n-1} e^{i\tau m/n (j-k)}$$

if $j \neq k$: $\sum_{m=0}^{n-1} \left(e^{i\tau \frac{j-k}{n}}\right)^m$

geometric series $\sum_{k=0}^{n-1} z^k = \frac{z^n - 1}{z - 1}$

$$= \frac{\left(e^{i\tau \frac{j-k}{n}}\right)^n - 1}{e^{i\tau \frac{j-k}{n}} - 1}$$

$$= 0$$

Applications :

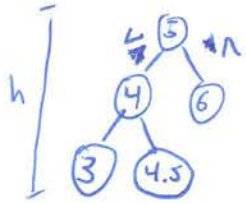
2/13/15

6.046 Recitation #2

Binary Search Trees
2-3 Trees
B-trees

Binary Search Tree (BST)

Search, Insert, Delete } $\Theta(\log n)$ time



Search, Insert, Delete } $O(h)$

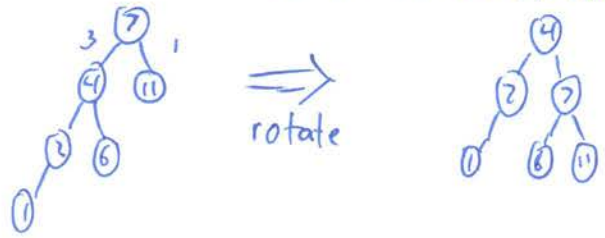
↳ so we want h to be small → balanced tree

Balanced BST

Invariant: For every node, difference b/w depth of left and right subtree must be at most 1. $\Rightarrow h = O(\log n)$

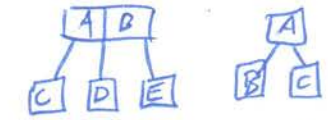
Rotations

Need to make sure invariant holds after insertion



2-3 Trees

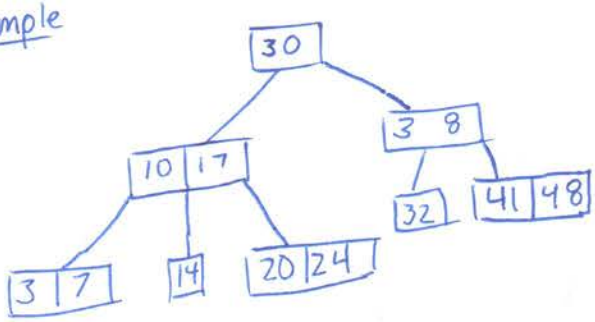
- Each node with non-leaf children must have two children or three children
- All leaves are at the same level
- All data is sorted order
- Every leaf node must contain one or two values



C<A E>B
A<D<B

B<A
C>A

Example



elements : n

height = O(lg n)

↗ more dense than BST

Operations

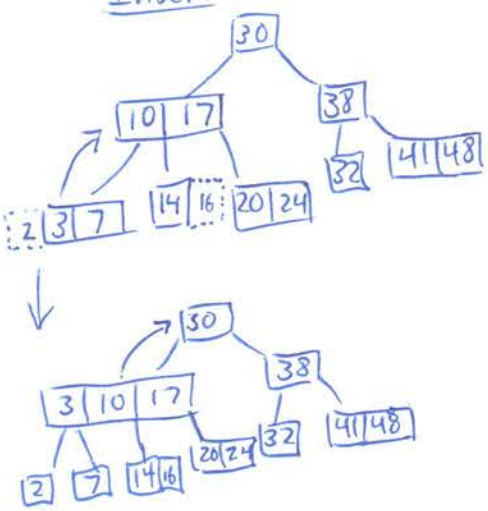
Search

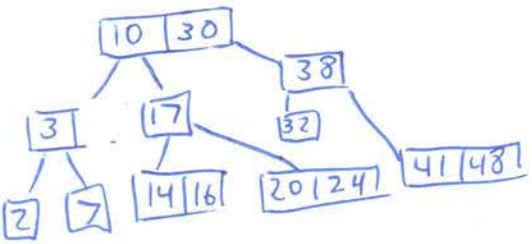
looking for k

like BST with an extra check if 3 node root.

Runtime : O(lgn)

Insert

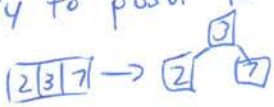




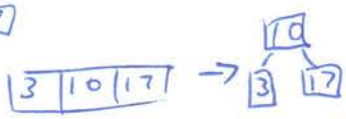
Insert 16 : Search For 16, insert at spot

Insert 2 : Inserting would create a node with three values → breaks invariant

Solution : Take median of 3 elements and try to push to parent.



and recurse ↓



Done :

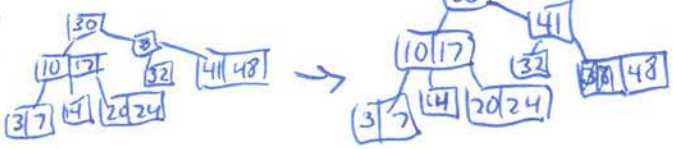
If root has three elements

make a new root

↳ will push an element at most the height of the tree

Runtime : O(logn)

Deletion



Delete 38

Successor of any element in an internal node will be a leaf node.

- Swap with successor
- delete if possible
  - if not => special case → see book

Runtime O(logn)

B-trees

Parameterized by constant B

- Each internal node must have $\geq B$ children and $\leq 2B$ children.
- All leaves must be at same depth
- Data is sorted

If a node has i children, it has $i-1$ values.

Insert, Search, delete $\Big\}$ $O(\log n)$

6.046 Notes

Fernando Trujano

Representing Polynomials

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

Coefficient Representation

- Vector of coefficients $\langle a_0, a_1, \ldots a_{n-1} \rangle$

Evaluating: $\Theta(n)$ using Horner's rule: $A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \ldots + x_0(a_{n-1})$

Adding: $\Theta(n)$ $= c$ where $c_j = a_j + b_j$: Add all coeffs w/ same degree

Multiplying: $\Theta(n^2)$

convolution $c_j = \sum_{k=0}^{j} a_k b_{j-k}$ $\quad c = a \otimes b$

Point-value representation

- Set of point value pairs $\{(x_0, y_0), (x_1, y_1) \ldots (x_{n-1}, y_{n-1})\}$ where $y_k = A(x_k)$

- Evaluating: $\Theta(n^2)$ using Horner's rule. $\Theta(n \log n)$ if points chosen cleverly. ★
  (n points)

Interpolation - inverse of evaluating

For any set of n points there is a unique polynomial of degree-bound n

- Adding: $\Theta(n)$
- Multiplying: $\Theta(n)$

★ Chose the complex roots of unity (in next section 30.2)

can take Discrete Forier Transform of coeff vector → point-value

inverse DFT in $\Theta(n \log n)$

So: originally coeffs $\xrightarrow[\Theta(n^2)]{\text{multiply}}$ solution $\Theta(n^2)$

now: coeffs $\xrightarrow[\Theta(n \log n)]{\text{evaluation}}$ point-value $\xrightarrow[\Theta(n)]{\text{multiply}}$ $\xrightarrow[\Theta(n \log n)]{\text{interpolation}}$ solution $\Theta(n$

2/19/15

6.046 Lecture

Fernando Trujano

van Emde Boas

- series of improved DSs
- Insert, Successor
- Delete
- Space

Goal: maintain n elements among $\{0, 1, \ldots, u-1\}$

subject to: Insert, Delete, Successor

Successor in $O(\lg\lg u)$ time

if $u = n^{O(1)}$ or $n^{\lg^{O(1)} n}$ then $\lg\lg u = O(\lg\lg n)$

- application: network routers

Intuition: How to get $O(\lg\lg u)$

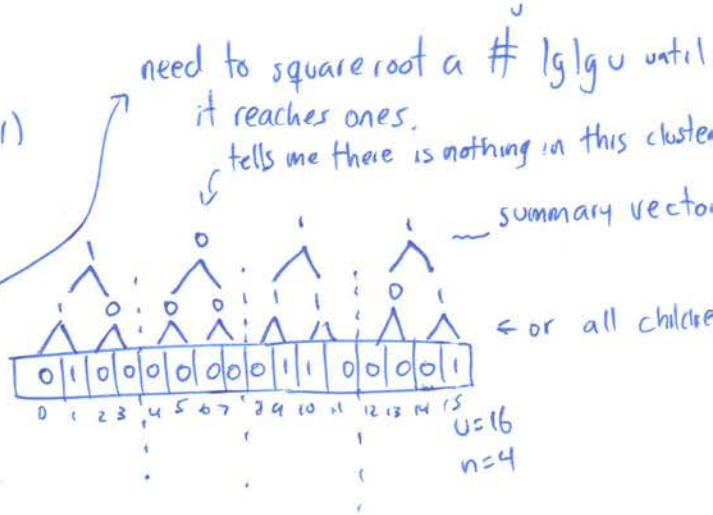
- binary search on levels of tree
- $T(k) = T(k/2) + O(1)$
  $= O(\lg k)$

$T(\lg u) = T\left(\frac{\log u}{2}\right) + O(1)$

$T'(u) = T(\sqrt{u}) + O(1)$

$= O(\lg\lg u)$

need to square root a # $\lg\lg u$ until it reaches ones.



① bit vector = array of size u
0 = absent 1 = present
Insert/Delete: $O(1)$
successor: $O(u)$

② Split universe into $\sqrt{u}$ clusters of size $\sqrt{u}$
Insert: $O(1)$
Successor(x): - look in x's cluster
- look for next one bit in summary vector and Find 1st one in that cluster

$O(\sqrt{u})$

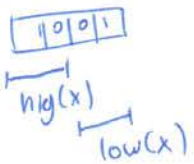
Using recursion we can turn this to $T(\sqrt{u})$!

if $x = i\sqrt{u} + j$

$0 \le j \le \sqrt{u}$

$x = 9$

$9 = 2 \cdot 4 + 1$



$high(x) = \lfloor x/\sqrt{u} \rfloor$

$low(x) = x \bmod \sqrt{u}$

$index(i, j) = i\sqrt{u} + j$

(3) recurse: V = size u vEB

- $V.cluster[i]$ = size $\sqrt{u}$ vEB
- $V.summary$ = size $\sqrt{u}$ vEB

```
Insert(V, x):
  - Insert into cluster: Insert(V.cluster[high(x)], low(x))
  - Update summary structure: Insert(V.summary, high(x))
```

```
Successor(V, x):
  i = high(x)
  j = successor(V.cluster[i], low(x))   ← try to find in x's cluster
    if j = ∞:   // wrong cluster
      i = successor(V.summary, i)   // look for next 1 in summary
      j = successor(V.cluster[i], -∞)
  return index(i, j)
```

Runtimes:

These aren't that great because we do more than 1 recursive call for each call.

Ex: Insert $T(u) = 2T(\sqrt{u}) + O(1)$ ← could be better

$T(\lg u) = 2T\left(\frac{\lg u}{2}\right) + O(1) = O(\lg u)$

↖ would be fixed if there was only 1 recursive ca

Successor: $\sim O(\lg u)$

④ Store min and max Augment data structure with min and max

```
Insert(V, x):
    if x < V.min:
        V.min = x    // Every structure knows minimum
    if x > V.max
        V.max = x
    :
```

```
Successor(V, x)
    i = high(x)
    :
    :
    j = V.cluster[i].min    // Now could only take 2 recursive calls
```

But we can do better...

```
Successor(V, x):
    i = high(x)
    if low(x) < V.cluster[i].max:
        j = successor(V.cluster[i], low(x))
    else:                                          // Only 1 recursive call
        i = successor(V.summary, high(x))
        j = V.cluster[i].min
    return index(i, j)

// Runtime O(lg lg u)
```

Insert not done yet:
First time we insert into cluster still takes two recursive calls...

(5) Don't store min recursively

```
Insert(V, x):
    if x < V.min: return V.min
    if V.min = None: V.min = V.max = x  return
    if x < V.min: swap(x <-> V.min)
    if x > V.max: V.max => x
    if V.cluster[high(x)].min = None:
        Insert(V.summary, high(x))      // if this gets called
    Insert(V.cluster[high(x)], low(x))  // this call takes O(1) because of
```

Runtime: $O(\lg\lg u)$

Now for delete:

```
Delete(V, x):
    if x = V.min:
        i = V.summary.min
        if i = None: V.min = V.max = None return
        x = V.min = index(i, V.cluster[i], min
    Delete(V.cluster[high(x)], low(x))
    if V.cluster[high(x)].min = None:
        Delete(V.summary, high(x))
    if x = V.max: if V.summary.max = None: V.max = V.min
                  else: i = V.summary.max
```

(6) Space is $O(u)$

- only store nonempty clusters

=> V.cluster = hashtable // instead of array

- $O(n \lg\lg u)$ space

2/20/15

6.046 Recitation

Fernando Trojan

Union-Find and Non-rigorous amortized analysis

Union-Find data structure

Disjoint-set data structure

Collection of pairwise independent sets $\{S_1, S_2, S_3, \ldots S_n\}$ ($\{1,2\}$ $\{3\}$)

each set S_i should have a single representative element $repr[S_i]$

- MAKE-SET(x) → $\{x\}$, $repr[\{x\}] = x$
- FIND-SET(x) → First find set S_x which contains x, and then returns $repr[S_x]$
- UNION(x,y) → Replace S_x and S_y with $S_x \cup S_y$ if $S_x \neq S_y$

Motivation: Keep tracle of connected components in dynamically changing graph
↳ $G = V, E$
Add node v to G
(v has associated edges)

Linked list solution

Represent each set as a doubly linked list

$\{1,2\}$ 1 ⇄ 2
$\{3\}$ 3
$\{4,5,6\}$ 4 ⇄ 5 ⇄ 6

rep(S_i) = head of list representing S_i

MAKE-SET(x) → Initialize x as a lone node O(1)

FIND-SET(x) → Hashtable that maps x to node containing x O(n)
-follow parent pointer to head and return repr. $\theta(n)$ ←BAD

UNION(x,y) → link head of one list and tail of another

repr($S_x \cup S_y$) : head of 1st list.

O(n) to find head and tail.
↳ BAD

Problem: Not efficient to walk through list to find repr and tail

Solution: Augment all x with $repr[S_x]$

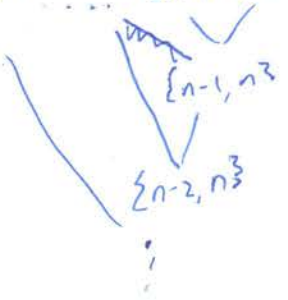
Find-set: O(1)

Union(x,y): Have to update all pointers in y O(n) ← still bad

Poorly performing case:

Initially n sets with one element n

{1} {2} {3} {4} ... {n-1} {n}



Total cost: $1+2+3+\ldots+n = \frac{n(n-1)}{2} = O(n^2)$

Solution: Append smaller to the end of larger
↳ then don't have to update too many pointers

Total runtime: $1+1+1\ldots. = O(n)$

Amortized Analysis

Recall table doubling from hashing:

Sometimes inserts can be very expensive, but usually really fast

With our improvement to the augmented linked list, n calls to make set
followed by n calls to union costs an amortized time of $O(n \log n)$

⇓

Union costs $O(\log n)$ on average

Consider one node x. Updating repr[S_x] costs 1.

When UNION(x, y) is called

Either $\text{len}(S_x) > \text{len}(S_y)$ → x's pointer not updated

or $\text{len}(S_y) < \text{len}(S_x)$ → x's pointer updated

$\text{len}(S_x)$ at least doubles post union operation.

⇒ repr[S_x] is updated at most $\log n$ times.

2/24/15

6.046 Lecture

Fernando Trujan

Amortization

- aggregate method
- accounting method
- charging method
- potential method
- table doubling
- binary counter
- 2-3 trees

Recall: Table doubling [6.006]

- n items in table of size m $\Rightarrow$ expected cost of $O(1+m/n)$
  - if n grows to m then double m $m' = 2m$ $O(m)$ work

Total cost: For n insertions $= \theta(2^0 + 2^1 + 2^2 + \ldots 2^{\lg n}) = \theta(n)$

$\Rightarrow$ "amortized cost per operation" $= O(1)$

Aggregate Method

total cost of k operations / k = amortized cost per operation

Amortized bounds

- assign cost for each (amortized) operation such that "preserve the sum"

$$\sum_{op} \text{amortized cost} \geq \sum_{op} \text{actual cost}$$

Ex: 2-3 trees achieve $O(1)$ (c) worst-case per create-empty $O(\lg n)$ (i) amortized per insert.

$\emptyset$ (d) amortized per delete ← since can't delete something that was not inserted

total cost $= O(c + i \lg n^* + d \lg n^*) = O(c + i \lg n^* + d(0))$

$\lg n^* \leq i$

n^*: max size over all time

Accounting Method

- allow an operation to store credit in bank account (≥ 0)
- allow an operation to pay for time using stored credit. balance ≥ 0 (unused time)

2-3 Trees

Claim: $O(\lg n)$ (NO STAR) per insert

$\emptyset$ per delete, amortized

- put 1 coin worth $\theta(\lg n)$ per insert (So each coin has a different value)
- delete consumes 1 coin amortized cost = actual cost + deposits - withdrawals
- invariant: 1 coin of $\lg i$ for $i = 1 \ldots n$

Table

Table doubling (insertions only)

- when insert, add on it worth $c = O(1)$
- when double, last $n/2$ items have coins

$\Rightarrow$ amortized cost $= \theta(n) - c\frac{n}{2} = 0$ if set c large.

Ex: empty table 

insert x 

double 

insert y 

double 

insert z 

Charging method:

- allow operations to charge costs retroactively to past (not Future)
- amortized cost = actual cost - total charge to past + total charge in Future

Table doubling (inserts only)

- charge doubling ($\theta(n)$) to inserts since last doubling ($n/2$) $\Rightarrow \theta(1)$ each
- only charge insert once

Table doubling (and halving) (inserts and deletes): $m = \theta(n)$

100% Full $\rightarrow$ double
25% Full $\rightarrow$ halve
$\Big\}$ new array always 50% Full

have to delete at least $m/4$ items to halve

- charge halving to $\geq m/4$ deletions since last resize
  doubling to $\geq m/2$ insertions "

Potential Method

- define potential function Φ mapping data-structure configuration → nonnegative integer

Potential high costs in future

- amortized cost = actual cost + $\Delta\Phi$
  ↳ Φ(after op) − Φ(before op)

Σ amortized cost = Σ actual costs + Φ(end) − Φ(beginning)

⇒ pay Φ(beginning) at beginning
- should be $O(1)$ or $\emptyset$

Binary Counter

increment

00110 10111 → 00110 11000

+

- increment costs (# trailing 1's + 1)
  Φ = # 1 bits

$\Delta\Phi$ should not change that much after each operations

- increment destroys + 1s, creates one 1

Amortized cost = 1 + t − t + 1 = 2

2-3 Trees (insertions only)

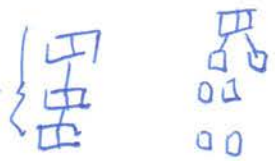
- how many splits per insert?
  ≤ lg n worst case
  $O(1)$ amortized

Φ = # nodes with 3 children

splits = k

$\Delta\Phi$ = −k + 1

amortized cost = k − k + 1 = 1

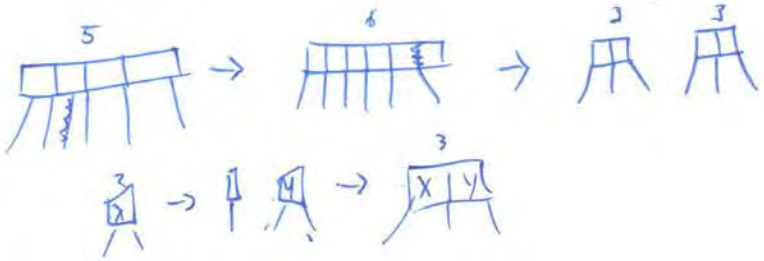


(2,3) Trees

Inserts and Deletes in (2,5) -trees

(2 ≤ #children ≤ 5)

Bad cases



Φ = #nodes w/ 2 children + #nodes with 5 children

insert w/ k splits $\Delta\Phi = -k+1$

delete w/ k merges $\Delta\Phi = -k+1$

$\Rightarrow O(1)$ splits/merges per insert/delete

2/26/15 6.046 Lecture Fernando Trujan

Randomized Algorithms

- Why randomized
- Matrix multiply verification
- Quick Sort

- Algorithm that generates a random number $r \in \{1, \ldots R\}$ and makes decisions based on r's value

On the same input on diff executions ↳ run for diff # steps
- could produce diff outputs

Probably Correct	Probably Fast	Probably correct + probably Fast
Monte Carlo	Las Vegas	Atlantic City

Matrix Product

$C = A \times B$

Simple Algorithms: $O(n^3)$ multiplications

Strassen - multiply two 2×2 matrices using 7 mults (instead of 8): $O(n^{\log_2 7}) = O(n^{2.81})$

Coppersmith-winograd: $O(n^{2.376})$

- Get a $O(n^2)$ algorithm

IF $A \times B = C$, then prob[output = YES] = 1
IF $A \times B \neq C$, then prob[output = YES] $\leq 1/2$ ← False positive
↳ run it k times $\frac{1}{2^k}$ => still $O(n)$

entries $\in \{0,1\}$ mod 2

Freval's Algorithm

Choose random binary vector $r[1 \ldots n]$
show that $Pr[r_i = 1] = 1/2$ independently for $i = 1, \ldots n$
IF $A(B_r) = C_r$, output YES
else output NO
IF $AB = C$, $A(B_r) = (AB)_r = C_r$ // No false negatives

Analyzing correctness: $AB \neq C$

Claim: if $AB \neq C \Rightarrow \text{Prob}[ABr \neq Cr] \geq 1/2$

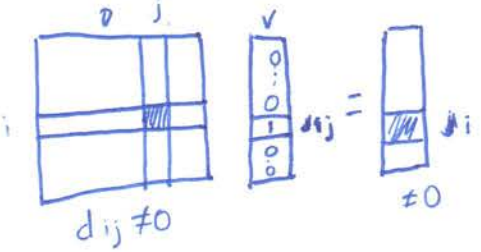
Let $D = AB - C$. Our hypothesis is that $D \neq 0$.

We need to show that there are many r such that $Dr \neq 0$

Specifically, $\text{Prob}[Dr \neq 0] \geq 1/2$ for randomly chosen r

$Dr = 0$ case // "bad r" => False positive

$D = AB - C \neq 0 \Rightarrow \exists\ i, j$ st $d_{ij} \neq 0$



$Dv \neq 0$ $(Dv)_i = d_{ij} \neq 0$

Take any r that can be chosen by our alg s.t $Dr = 0$

$r' = r + v$ ← vector addition $\quad Dr' = D(r+v) = 0 + Dv \neq 0$

r to r' is 1 to 1 ; if $r' = r + v$
$= r'' + v$
then $r = r''$

$Dr = 0$

Discover an r' s.t $Dr' \neq 0$ $\quad r \leftrightarrow r'$ 1 to 1 mapping

Number of r' for which $Dr' \neq 0 \geq$ number of r for which $Dr = 0$

$\Rightarrow \Pr[Dr \neq 0] \geq 1/2$
↑ randomly chosen

Quick Sort

Divide and Conquer in place $O(n)$ no auxillary space
all work in divide step

n-elmt array A

Divide: 1. Pick a pivot elm x in A. → CLRS p 171
Partition array into sub arrays

Conquer: Recursively sort sub arrays L & G

Combine:

Basic Quick Sort

pivot $x = A[1]$ or $A[n]$

- partition given x $O(n)$ time

Worst case analysis

sorted or reverse sorted

one side L or G has n-1 elements. and other 0

$T(n) = T(0) + T(n-1) + \theta(n)$

$T(n) = T(n-1) + \theta(n)$ ← divide step

$= \theta(n^2)$

Pivot selection: intelligently → worst case $O(n \log n)$

guarantee balanced L and G using median selection that runs in $\theta(n)$ time.

$T(n) = 2T(n/2) + \theta(n) + \theta(n)$ → divide/partition

↓ thanks to median based pivoting.

↳ recursive median selection

Randomized QuickSort

expected time is $O(n \log n)$ for all input array A

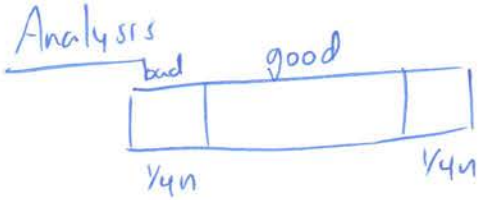
CLRS p 181-184

Paranoid Quicksort

↙ afraid of unbalanced partition

Repeat
  choose pivot to be random element of A
  Perform partition
Until resulting partition is s.t $|L| \leq \frac{3}{4}|A|$ and $|G| \leq \frac{3}{4}|A|$

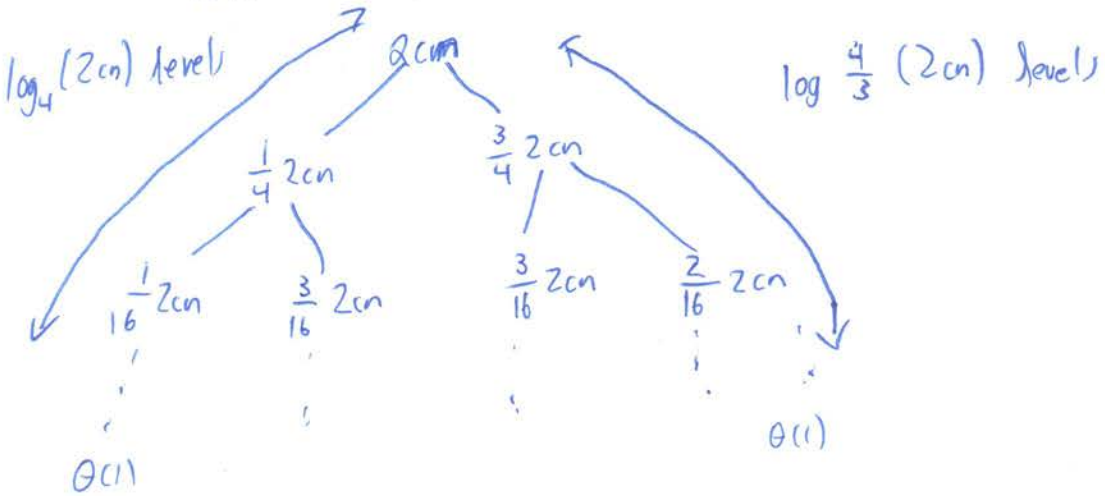
Analysis



A call is good with prob $\geq 1/2$

$$T(n) = T(n/4) + T(3n/4) + E(\#\text{ iterations}) \cdot c \cdot n$$

$$T(n) = T(n/4) + T(3n/4) + 2 \cdot cn$$



$2cn$ work at each level
max $\log_{4/3}(2cn)$ levels $\Rightarrow O(n \log n)$

2/27/15

6.046 Recitation

Fernando Trejano

Expected running time of: Randomized Select
Randomized quicksort

Randomized select

Given an array and index. Find the k^{th} largest (or smallest) element in the array.

```
RANDOMIZED_SELECT (A, p, r, i):
    if p==r: return A[p]
    q -> ~~pivot~~ -> Partition(A,p,r)
    k -> q - p + 1
    if i ≤ k : RANDOMIZED_SELECT (A, p, q, i)
    else: RANDOMIZED_SELECT (A, q+1, r, i-k)
```

// Different ways to select pivot

$T(n) \rightarrow$ Expected Runtime of select on input size n

$$Ex[T(n)] \leq \underbrace{\Theta(n)}_{\text{from PARTITION}} + Ex[\max(T(1), T(n-1))]$$

```
RANDOMIZED_PARTITION(A,p,r):
    i -> RANDOM(p,r)
    Exchange A[p] and A[i]
    return PARTITION(A,p,r)
```

$$T(n) \leq cn + \sum_{i=0}^{n-1} P[E_i](\max(T_i, T(n-1)))$$

$$\leq cn + \sum_{i=0}^{n-1} \frac{1}{n} \cdot \max(T_i, T(n-1))$$

$$\leq cn + \frac{2}{n} \sum_{i=0}^{n/2-1} \max(T(i), T(n-1))$$

By substitution

Guess $T(n) = O(n)$

By induction:

For all $k < n$ $T(n) = O(n) \Rightarrow T(n) \leq cn$ (constant)

$$T(n) \leq n + \frac{2}{n} \sum_{i=0}^{n/2-1} \max(T(i), T(n-i))$$

$$\leq n + \frac{2}{n} \sum_{i=0}^{n/2-1} \max(ci, c(n-i))$$

$$\leq n + \frac{2}{n} \sum_{i=n/2}^{n-1} ci$$

$$T(n) \leq n + \frac{2}{n} \cdot c \sum_{n/2}^{n-1} i \leq n + c\left(\frac{3n}{4}\right) = n\left(1 + \frac{3c}{4}\right)$$

If $c=4$

$$T(n) \leq n\left(1 + \frac{3c}{4}\right) = 4n$$

$$= cn$$

~~can~~ have runtime worse than $O(n)$ if we pick multiple bad pivots.

$T(n) = O(n) + T(n-1) = O(n^2)$

Randomized Quick Sort

```
RANDOMIZED-QUICK_SORT (A, p, r)
  if p < r:
    q ← PARTITION(A, p, r)
    RANDOMIZED_QS(A, p, q-1)        // Recurse on both sides
    RANDOMIZED-QS(A, q+1, r)
```

$$T(n) \leq n + Ex[T(i) + T(n-i)]$$

$$\leq n + \sum_{i=0}^{n-1} \underbrace{Pr(E_i)}_{1/n} (T(i) + T(n-i))$$

2/3

$$\leq n + \sum_{i=0}^{n-1} \frac{1}{n} \left(T(i) + T(n-i-1) \right)$$

Substitution - (like in randomized select)

Guess $T(n) = \Theta(n \log n)$

For all $k < n$ $T(k) = O(k \log k)$

$$T(n) \leq n + \frac{2}{n} \sum_{i=0}^{n-1} T(i) \quad \text{(i log i)} \longrightarrow \left[\sum_{i=0}^{n-1} T(i) + T(n-i-1) \right]$$

$$\leq n + \frac{2}{n} \sum_{i=0}^{n-1} i \log i$$

Use:

$$\sum_{i=0}^{n-1} i \log i \leq \frac{1}{2} n^2 \log n - \frac{1}{8} n^2$$

$$T(n) \leq n + \frac{2}{n} \sum_{i=0}^{n-1} i \log i \leq n + \frac{2}{n} \left(\frac{1}{2} n^2 \log n - \frac{1}{8} n^2 \right)$$

$$\leq cn \log n + n \left(1 - \frac{c}{4} \right)$$

$$\leq cn \log n \quad \square$$

Expectation vs Amortized

Expectation → Averaging over random decisions

Amortized → Averaging over operations

3/3/15

6.046 Lecture

Fernando Trujano

Skip Lists

- Data structure (randomized)
- With high probability (whp) notion
- whp analysis of SEARCH

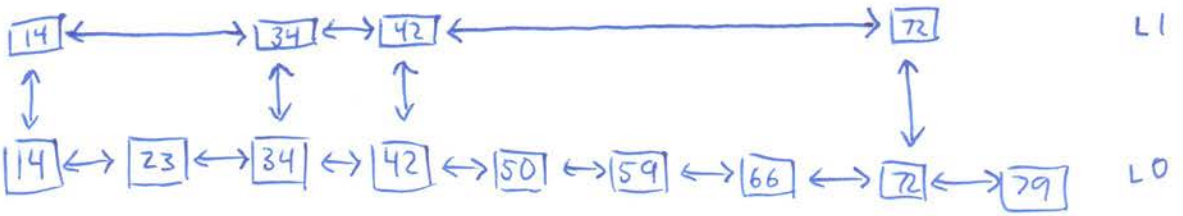
Skip List

- easy to implement
- Maintain a dynamic set of n elements in $O(\log n)$ time per operation in expectation and whp

One sorted linked list

Search: $O(n)$

Two sorted linked lists



↳ NYC Subway Express line.

Search(x): (2-linked list)

- Walked right in top linked list(L_1) until going right would go too far
- walk down to bottom list (L_0) until element is found (or not)

Analysis: Search Cost $\approx |L_1| + \frac{|L_0|}{|L_1|}$ ← only traverse a portion of L_0

minimize this: when terms are equal

$|L_1|^2 = |L_0| = n$

$|L_1| = \sqrt{n}$

Search cost $\approx O(\sqrt{n})$

2 sorted lists → $2\sqrt{n}$ ← dispersed optimally

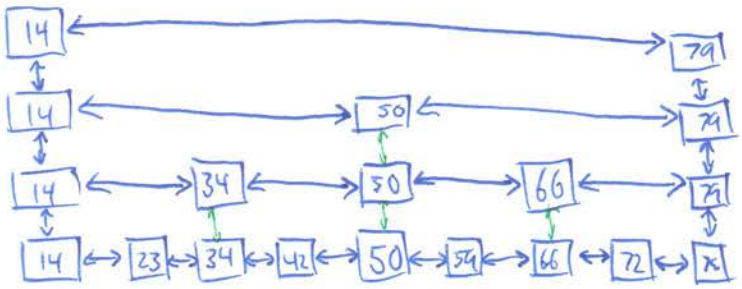
3 sorted lists → $3\sqrt[3]{n}$

k sorted lists → $k\sqrt[k]{n}$

log n sorted lists → $\lg \sqrt[\lg n]{n} = 2 \log n$

* This only works with a static set!

Invariant: If the station i is in L_x. It must also be in $L_{x-1} \ldots L_0$



Insert (x)

To insert an element x into skip list:
- Search(x) to see where x fits into bottom list
- Always insert into bottom list
- Insert into some of the lists above (which ones) (Promote)

→ Flip fair coin
If heads: (maybe newly created) → promote x to next level up and repeat
else: stop

Delete(x): Search(x) and delete at all levels

Warmup Lemma: # levels in n-element skip list is $\underbrace{O(\lg n)}_{c \lg n}$ whp ← prob $1 - \frac{1}{n^{\alpha}}$ ← related to α

Proof: Failure probability (not $\leq c \log n$ levels)
$= Pr(> c \lg n \text{ levels})$
$= Pr(\text{some element gets promoted} > c \lg n \text{ times})$
$\leq n \cdot Pr(\text{element } x \text{ gets promoted} > c \lg n \text{ times})$ ← union bound

$= n\left(\frac{1}{2}\right)^{c \lg n} = \frac{n}{n^c} = \frac{1}{n^{c-1}} = \frac{1}{n^\alpha} \quad \alpha = c-1$

Search: Thm: Any search in an n-element skip list costs $O(\lg n)$ whp

Analyze Search backwards

bsearch starts[ends] of node in bottom list
At each node visited
  if node was not promoted higher (tails ere)
    then we go to came from left
  if node was promoted higher
    then we go to came from up

- stop [start] when we reach top level (or ~)

Proof

Backwards search makes "up" moves and "left" moves each with $Pr = 1/2$

moves going up $<$ # levels
$\leq c \lg n$ whp (By Warmup Lemma)

Total # moves = # moves till you get $c \lg n$ up moves

Claim: Total # moves = # coinflips until $c \lg n$ heads $= O(\lg n)$ whp

□

Chernoff: Let Y be a random variable representing the total # of tails in a series of m independent coin flips where each flip has pr p of heads. Then for all $r > 0$ we have

$$Pr[Y \geq E[Y] + r] \leq e^{-\frac{2r^2}{m}}$$

Lemma : For any c, there is a constant d such that with the # heads in flipping $d \lg n$ fair coins is at least $c \lg n$

$d = 8c$

$E[Y] = m/2$

$r = d \lg n - c \lg n$

Randomization: Universal and Perfect hashing

Dictionary Problem: We want an ADT that supports the following operations.

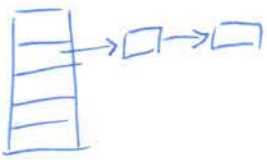
INSERT(item):
DELETE(item):
SEARCH(key).
↳ items have distinct keys

Hashing

Do these operations in $O(1)$ time and $O(n)$ space

U = # all possible items
n = # of current keys in table
m = # slots in table

Collision Resolution: Hashing with chaining



$\Theta(1+\alpha)$ time per operation
↳ load factor n/m

Simple Uniform Hashing

$$\Pr_{k_1 \neq k_2}(h(k_1) = h(k_2)) = 1/m$$

(collision!)

↳ assuming input keys are random :: ↗ Like basic quicksort
solution → universal hashing

Universal Hashing

Choose random hash function h from H ↳ Universal Hash Family

- now we assume h is random and keys can be whatever ↳ Like Randomized Quicksort

$$\Pr_{h \in H}\{h(k) = h(k')\} \leq \frac{1}{m} \text{ for all } k \neq k'$$

Theorem: For n arbitrary distinct keys and for random $h \in H$ (✓ universal)

$$E[\#\text{ keys colliding in a slot}] \leq 1 + \alpha \quad (\alpha = n/m)$$

Proof

Let $I_{i,j}$ = Indicator Random variable $= \begin{cases} 1 & \text{if } h(k_1) = h(k_2) \\ 0 & \text{otherwise} \end{cases}$

$$E[\#\text{ keys colliding with } k_i] = E\left[\sum_{j=1}^{n} I_{i,j}\right]$$

$$= \sum_{j=1}^{n} E[I_{i,j}] \quad \text{By linearity of expectation}$$

$$= \sum_{j \neq i} \underbrace{E[I_{i,j}]}_{Pr(I_{ij}=1) = P_r(h(k_i)=h(k_j)) = 1/m} + E[I_{i,i}]$$

$$= n/m + 1 = 1 + \alpha \quad \square$$

Hash Families

Dot product hash family

- assume m prime, $U = m^r$ for int r

  view keys in base m: $k = \langle k_0, k_1, \ldots k_{r-1} \rangle$

- for key $a = \langle a_0, a_1 \ldots a_{r-1} \rangle$

  define $h_a(k) = a \cdot k \bmod m$

  $= \sum_{i=0}^{r-1} a_i \cdot k_i \bmod m$

$H = \{h_a \mid a \in \{0,1 \ldots U-1\}\}$

- storing h_a in H requires storing 1 key $\Rightarrow$ computing $h_a(k)$ takes $O(1)$

Theorem: dot-product hash family is universal

Proof. Choose any two keys $k \neq k'$, differ in some digit $k_d \neq k'_d$

Let not $d = \{0, 1, \ldots, r-1\}$

$$\Pr_a\{h_a(k) = h_a(k')\} = \Pr_a\left\{\sum_{i=0}^{r-1} a_i k_i = \sum_{i=0}^{r-1} a_i \cdot k'_i \pmod m\right\}$$

$$= \Pr_a\left\{\sum_{i \neq d} a_i k_i + a_d \cdot k_d = \sum_{i \neq d} a_i \cdot k'_i + a_d k'_d \pmod m\right\} = \Pr_a\left\{\sum_{i \neq d} a(k_i - k'_i) + a_d(k_d - k'_d) = 0 \pmod m\right\}$$

$$= \Pr_a\left\{a_d = -(k_d - k'_d)^{-1} \sum_{i \neq d} a_i (k_i - k'_i) \pmod m\right\}$$

$$= \mathop{E}_{a_{not\,d}}\left[\Pr_{a_d}\{a_d = f(k, k', a_{not\,d})\}\right] = \sum_x \Pr\{a_{not\,d} = x\} \Pr_{a_d}\{a_d = f(k, k', x)\}$$

$$= \mathop{E}_{a_{not\,d}}[1/m] = 1/m \quad \square$$

21

Another universal Hash Family CLRS

- prime $p \geq u$

$$h_{ab}(k) = [(a \cdot k + b) \bmod p] \bmod m$$

- $\mathcal{H} = \{ h_{ab} \mid a, b \in \{0, 1, \ldots, u-1\} \}$

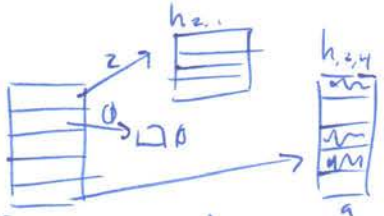
Static Dictionary Problem

given n keys to store in table, support Search(k)

Perfect Hashing - no solutions

- polynomial build time whp
- O(1) time search → worst case
- O(n) space → worst case

Idea: 2-level hashing



$\{0, 1 \ldots u-1\} \to \{0, 1 \ldots m-1\}$

① Pick h_1 from a universal hash family

For $m = \Theta(n)$ (nearby prime): hash all items with chaining with h_1

② For each slot $j \in \{0, 1 \ldots n-1\}$

- let l_j = # items in slot j = $|\{i \mid h(k_i) = j\}|$
- pick $h_{2,j} : \{0, 1 \ldots u-1\} \to \{0, 1, \ldots m_j\}$

From universal hash family

For $l_j^2 \leq m_j \leq O(l_j^2)$ (nearby prime)

- replace chain in slot j with hashing-with-chaining using $h_{2,j}$

Space $= O\left(n + \sum_{j=0}^{m-1} l_j^2\right)$

(1.5) IF $\sum_{j=0}^{m-1} l_j^2 > cn$ redo step ① ↝ to guarantee space = O(n)

↑ some constant

Search time = $O(1)$ For first table h_1

$+ O(\text{max chain size in second table})$

(2.5) while $h_{2,j}(k_i) = h_{2,j}(k_{i'})$ (ANY collision) For any $i \neq i', j$

repick $h_{2,j}$ and rehash those ℓ_j items

↑ guarantees search $O(1)$

=> no collisions in second level

3/6/15

6.046 Recitation

Hashing and Dynamic Programming

Universal Hashing

Hash Family $\mathcal{H} = \{h_1, h_2, \ldots\}$

Pick h at random from $\mathcal{H}$

Guarantee that

$$\Pr(h(k) = h(k')) \leq 1/m \quad \forall\, k \neq k'$$

No assumption about input keys

$$\mathcal{H} = \{h_a : h_a(k) = a \cdot k \bmod m\}$$ (m prime)

(a: random $0 \leq a_i \leq m-1$)

Proof that $\mathcal{H}$ is universal: $\Pr(h_a(k) = h_a(k'))$ where $k \neq k'$

$k = \{k_0, k_1, \ldots, k_{r-1}\}$

$k' = \{k'_0, k'_1, \ldots, k'_{r-1}\}$

There must exist some index d s.t. $k_d \neq k'_d$

$$\Pr(h_a(k) = h_a(k')) = \Pr\left(\sum_{i=0}^{r-1} a_i k_i = \sum_{i=0}^{r-1} a_i k'_i \bmod m\right)$$

$$\Pr\left(\sum_{i \neq d} a_i k_i + a_d k_d \equiv \sum_{i \neq d} a_i k'_i + a_d k'_d \pmod m\right)$$

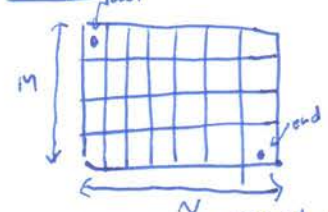
$$= \Pr\left\{a_d(k_d - k'_d) \equiv \left(\sum_{i \neq d} a_i (k'_i - k_i)\right) \bmod m\right)$$

⋮

Dynamic Programming

1. Define subproblems
2. Determine relationship b/w subproblems (Guessing)
3. Determine how to compute final answer given solutions to subproblems

Example



Goal: Get robot to end

How many ways

can only move → ↓

Subproblem

$S[i,j] \rightarrow$ # ways of getting from $(1,1)$ to (i,j)

$$S[i,j] = S[i-1, j] + S[i, j-1]$$

Base Case

$S[1,1] = 1$

$S[M,N] =$ solution

subproblems: $M \times N$

time/subproblem: 1

Total: $O(M \times N)$

Example

n rectangular blocks → length l, height h, breadth b:

want to maximize height of a tower subject to constraints

constraint:

IF B_i is stacked on top of B_j, then $l_j \geq l_i$ and $b_j \geq b_i$

1) Sort blocks in non-increasing order of length and then breadth

$H[i]$ → Height of tallest tower with B_i on top

$$H[i] = h_i + \max_{j<i,\ l_j \geq l_i \text{ and } b_j \geq b_i} H[j]$$

↳ guess over all blocks that come before B_i

Example

Counting Boolean Parenthesis

True, False, And, OR, XOR

True AND False XOR TRUE

⇓

(True AND False) XOR TRUE → True

True AND (False XOR True) → True

Solution: 2

Goal: Find # ways to parenthesize string to evaluate to true.

$T[i,j]$ → # ways parenthesizing substring $s[i,j]$ s.t the expression evaluates to true,

$F[i,j]$ " " " False.

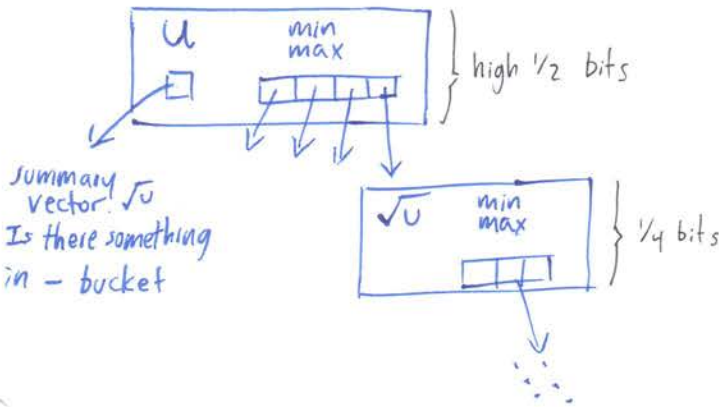
$$T[i,j] = \sum_{k=1}^{j-1} \begin{cases} T[i,k] \cdot T[k+1,j] & \text{if } s[k] = \text{AND} \\ T[i,k] \cdot F[k+1,j] + F[i,k] \cdot T[k+1,j] & \text{if } s[k] = \text{XOR} \\ T[i,k]\,T[k+1,j] + T[i,k]\,F[k+1,j] + F[i,k]\,T[k+1,j] & \text{if } s[k] = \text{OR} \end{cases}$$

Solution $T[1,n]$

6.046 Quiz 1 Review

Topics:
- Divide and Conquer
- vEB trees
- FFT
- B-Trees
- Amortization and Union Find
- Augmentation
- Randomized Algo
- Hashing
- Skip lists

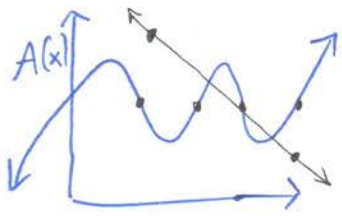
vEB Trees



Operations in $O(\lg \lg u)$ ← universe, not n!

FFT

Polynomials: $A(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1}$ (degree n-1)

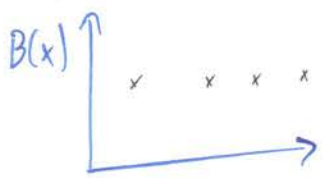


degree n
need (n+1) points to represent.

$B(x) =$ (degree n-1)

→ line degree 1
=> need 2 points to represent

$A(k) = ??$
can evaluate in $O(n)$ times with Horner's Rule.



$A(x)$ = degree n
$B(x)$ = degree n
$C(x) = O(2n)$ => 2n+1 points

$$A(x) = A_{even}(x^2) + xA_{odd}(x^2)$$

where

A_{even} = even terms of $A(x)$

A_{odd} = " " $A(x)/x$

Derivation:

$$\sum_{k=0}^{\lceil \frac{n}{2}-1 \rceil} a_{2k} x^k$$

$$\sum_{k=0}^{\lfloor \frac{n}{2}-1 \rfloor} a_{2k+1} x^k$$

FFT picks the $(2n+1)$th roots of unity as points to evaluate (DFT)

$$\left\{ e^{\frac{2\pi i k}{2n+1}} \right\} \text{ For } 0 \le k < 2n+1$$

↓ squared items (x^2)

half the elements

$(2n+1)/2$ roots of unity

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

$$= O(n \log n)$$

$$a_k = \sum_{j=0}^{n-1} a_j e^{\frac{2\pi i}{n} jk}$$

IFFT → inverse FFT to go back!

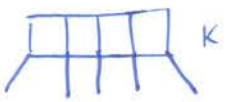
Example Problem: SP 2012 Q1 P4

B-Trees

- Each node can have multiple keys. k
- ~~Each~~ non-leaf node has k+1 children - ?

- non root node $B-1 \le \#keys \le 2B-1$
- root node $1 \le \#keys < 2B-1$ } split when reaching upper bound
- non leaf node $B \le \#children < 2B$

- achives $O(\lg n)$
  - Insert, delete, search
- node maintains key in sorted orded.



Amortization

Union Find

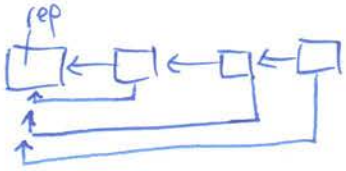
Make-Set(x) O(1)

Find-rep(x) O(1)

Union(x,y) O(n)

↓

$O(\lg n)$ amortized



Randomization

- Quicksort
- SkipLists
- Hashing
- Matrix Multiplication

Randomized quicksort

- pick a random pivot
- compare to pivot, partition
- Recurse

Skip Lists

- $O(\lg n)$ sorted layer
- every element appears at bottom layer

Insert(x)

- Search for item
- insert at bottom layer
- Flip a coin { Heads => promote to next level and repeat
  { Tails -> return

Want to ensure whp that there aren't more than $c \log n$ layers

Sample Problems

Worst case running time search: $O(n)$
-> every element promoted or none

Hashing

Universal Hashing : Choose function $h \in \mathcal{H}$ (set of hash functions)

$\mathcal{H}$ - universal hash family

$$\Pr_{\substack{h \in \mathcal{H} \\ k \neq k'}} \{ h(k) = h(k') \} \leq \frac{1}{m}$$

Example:

Dot-Product Hash Function:

Mult Mod Prime.
- Pick prime p. pick $a, b < p$

$$h_{a,b,p}(k) = [ak + b \bmod p] \bmod m$$

3/4

Example Prob: Consider Family hash Functions $H = \{f, g, h\}$

map $\{0, 1, 2\} \to \{0, 1\}$

$f(0) = 1 \quad f(1) = 0 \quad f(2) = 0$

$g(0) = 0 \quad g(1) = 1 \quad g(2) = 0$

$h(0) = 0 \quad h(1) = 0 \quad h(2) = 1$

Is this a universal Hash Family?

Yes

$1/m = \frac{1}{2}$

$Pr\{\text{collision}\} = 1/3$

Perfect Hashing

- Know all elements in advance
- O(1) lookups WORST CASE
- O(n) space
- $O(n^* \log^2 n)$ whp create table

Sample Question:

When achieving perfect hashing, there are no collisions at 1st and 2nd levels.

FALSE

Can have collisions at First level but NOT at second.

Monte Carlo Algorithm - Always polynomial - whp expected correct

Las Vegas Algorithm - whp Expected polynomial - Always correct

Dynamic Programming

1) Define subproblem
2) Relate subproblem solutions w/ recurrence + base case
3) Recurse and memoize or bottom up DP table
4) Solve original problem
Analyze runtime

DAG

Coin Changing

Input: n denominations of coins. $1 = V_1 < V_2 < \dots < V_n$
Goal: make change for unit of money C use as few coins.
$M(j)$: min # coins to make change for j

$M(j) = \min_i \{M(j-V_i)\} + 1$

Runtime: $O(nC)$

Longest Increasing Subsequence

Input: $A_1 \dots A_n$
Goal: Find longest strictly increasing sequence (not necessarily contiguous)

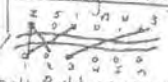
$L(j)$ = LIS ending at position j

$L(j) = \max_{i<j,\ A[i]<A[j]} \{L(i)\} + 1$

Solution = $\max_j \{L(j)\}$

Runtime: $O(n^2)$

Building Bridges



Goal: Build as many (non-crossing) bridges as possible
(like longest increasing subsequence)
$X(i)$: index of corresponding city on northern bank.
Compute $X(i)$'s in $O(n \log n)$ time.
Find longest subsequence of $x(1) \dots x(n)$ $O(n^2)$

Edit Distance

Input: Strings $A[1..n]$ and $B[1..m]$
Costs: C_i, C_d, C_R (insertion, deletion, replacement)
Goal: Min cost of transforming $A \to B$

$T(i,j)$ = min cost to transform $A[1..i]$ into $B[1..j]$

$$T(i,j) = \min \begin{cases} C_d + T(i-1,j) & \text{delete } A[i] \\ T(i,j-1) + C_i & \text{transform } A \to B[1..j-1] \text{ then insert } B[j] \\ \begin{cases} T(i-1,j-1) & \text{if } A[i] = B[j] \\ T(i-1,j-1) + C_R & \text{if } A[i] \neq B[j] \end{cases} & \text{transform } A[1..i-1] \to B[1..j-1] \text{ then transform } A[i] \to B[j] \end{cases}$$

$O(mn)$ # subproblems

Solution: $T(n,m)$

Balanced Partition ("fairly dividing up pets")

Input: n integers
Goal: Partition integers into S_1 and S_2
Minimize $|\text{Sum}(S_1) - \text{Sum}(S_2)|$

$P(i,j) = \begin{cases} 1 & \text{if some subset of } \{A_1 \dots A_i\} \text{ has sum } j \\ 0 & \text{otherwise} \end{cases}$

$P(i,j) = 1$ if $P(i-1,j) = 1$ or $P(i-1, j-A_i) = 1$
$= \max\{P(i-1,j), P(i-1, j-A_i)\}$

$S = (\sum A_i)/2$
Find min val of $S-i$ such that $P(n,i) = 1$
$\Rightarrow \min_{i \le S} \{S-i : P(n-i) = 1\}$

Integer knapsack Problem

Input: n types of items, each with size s_i (int) and value v_i, knapsack capacity C
Goal: Pack items in knapsack maximize value. (repeated items ok)

$M(j)$ = max val pack in knapsack c=j

For $j = 1$ to C # bottom up approach
compute $(M(j))$

$M(j) = \max\{M(j-1), \max_{\text{items } i} \{M(j-s_i) + v_i\}\}$

(jth slot empty or some item in j slot)

Runtime: C subproblems
n per subproblem $O(nC)$

Solution: Store a back pointer to $M(-)$ that was max.
$M(C)$ will be result. Following back pointers will give items used.

"nmp" - solutions can be confirmed in poly time
"NP-hard" at least as hard as everything in NP.
"NP complete" - NP and NP-hard

Box Stacking

Input: n boxes each with all orientations of original set
$w_i \le d_i$
h_i = height
width w_i
depth d_i

Goal: Construct stack of max height can only stack box i on j if $w_i < w_j$ and $d_i < d_j$

$\text{Sum}(S_1) = i$
$\text{Sum}(S_2) = \sum - S_i$
Solution: $\min_{i \le S} \{S-i : P(n-1) = 1\}$ since $|\text{sum}(S_1) - \text{sum}(S_2)| = \sum - 2i$

$H(j)$ = tallest stack of boxes with box j on top.

$H(j) = \max_{i<j,\ w_i > w_j,\ d_i > d_j} \{H(i)\} + h_j$

Solution: $\max_j \{H(j)\}$ = which box will end on top.

Runtime = $\Theta(n^2)$

3/10/15

6.046 Lecture

Fernando Trujano

Augmentation

- easy tree augmentation
- order statistic trees
- level-linked 2-3 trees
- range trees

-ea

Idea: Take "off the shelf" data structure and modify to store extra info.

Easy tree augmentation:

- goal: store f(subtree rooted @ x) at each node x in x.F
- suppose x.F can be computed in O(1) time from x, children, and children.F
- if modify set S of nodes then costs O(# ancestors of S) to update F fields.
  ⇒ O(lgn) updates in AVL tree → 2-3 trees

* isolated to subtree

Order statistic trees

- ADT / interface:
  - insert(x), delete(x), successor(x)
  - rank(x): index of x in sorted order
  - select(i): return key of rank (i)

O(lgn)/operation

Augmentation

- use easy tree augmentation with F(subtree) = #nodes in subtree

x.F = 1 + sum(c.F for c in x.children)
(F = size)

Implementation

- constant work/step
- O(lgn) steps

rank(x):
- rank = x.left.size
- walk from x to root of tree
  - if we go left x → x': rank += x'.left.size + 1

select(i):
- x = root
- rank = x.left.size + 1
  - if i = rank: return x
  - if i < rank: x = x.left
  - if i > rank: x = x.right, i -= rank

repeat

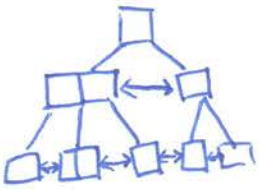
1/3

Pro-tip: Only augment things that can be easily maintained. NOT depth, rank

Level-linked 2-3 trees

Idea: store level-left and level-right pointers for each node

- easy to maintain in O(1) overhead

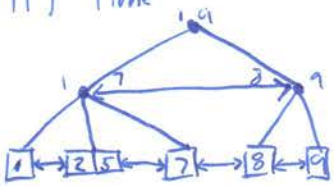


Finger Search Property

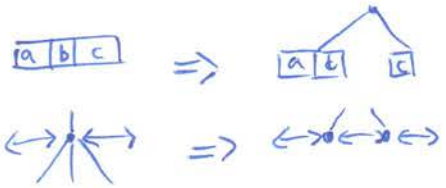
search(x from y):

$O(\log|\text{rank}(x) - \text{rank}(y)|)$ time

Store the data in the leaves!



Search - augment to store min and max of subtrees

Insert 

Search(x from y)

v = leaf containing y
if v.min ≤ x ≤ v.max
return regular search in v's subtree
else if x < v.min : v = v.levels-left
else if x > v.max : v = v.level.right
*: v = v.parent // go upp

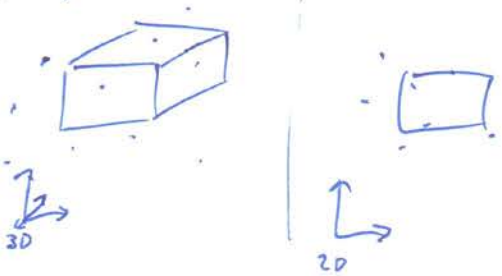
↓??

Orthogonal range search

goal: preprocess n points in d-l to support query given box

Find - # points in box
- k points in box → $\Theta(k)$ output

$O(\lg^d n + |\text{output}|)$



1D:

- sorted array
- BST $O(\lg n + k \lg n)$
- level linked $O(\lg n + k)$

1D range trees

range-query ([a,b])
- search(a)
- search(b)
- trim common prefix
- → $O(\lg n)$ nodes and rooted subtrees

2D range search

- 1D range tree on all points by x

for each node v in x-tree:
store 1D range tree by y on all points in v's rooted sub-tree

$O(\lg^2 n)$ time

Space:
each key lives in $\lg n$ subtrees (all ancestors)
$O(n \log^{d-1} n)$

3/3

18.1, 18.2

6.046 Notes

Fernando Trujano

B-trees

B-trees

minimum degree t - Each node must have at least $t-1$ keys. → all node at least t children
each node can have at most $2t-1$ keys → " at most $2t$ children

So if $t=2$ we get a 2-3-4 tree!

Height: $h \leq \log_t \frac{n+1}{2}$

All leaves are at the same level

3/16/15

6.046 Lecture

Dynamic Programming
- Longest Palindrome sequence
- Optimal Binary Search Trees
- Alternating coin game

DP Notions

1. Characterize structure of optimal solution
2. Recursively define the value of an optimal solution based on optimal solutions of subproblems
3. Compute value of optimal solution
4. Construct optimal solution from computed information

Longest Palindromic Sequence

Given a string: $A[1 \ldots n]$ $n \geq 1$

Find longest palindrome that is a subsequence

Ex: character → carac turboventilator → rotator

Strategy $L(i,j)$: length of longest palindromic subsequence $x[i, \ldots j]$ $i \leq j$

```
def L(i,j):                          → look at L[i,j]   //memoizing
    if i == j: return 1
    if x[i] == x[j]:                 //don't recurse if L(i,j) already
        if i+1 == j: return 2        //computed
        else return 2 + L(i+1, j-1)  // Look inside
    else: return max( L(i+1, j), L(i, j-1) )
```

subproblems $\Theta(n^2)$ × time to solve each subproblem given smaller ones are solved $\Theta(1)$ $= \Theta(n^2)$

Optimal BST

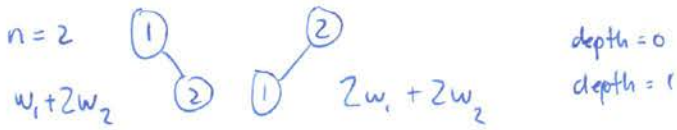
Keys $k_1, k_2, \ldots k_n$ $k_1 < k_2$ w.l.o.g. $k_i = i$

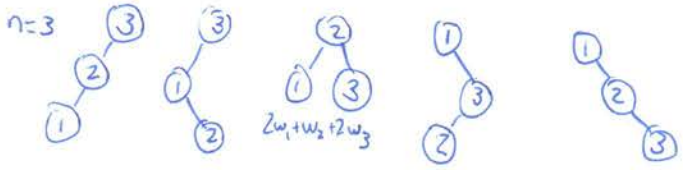
Weights $w_1, w_2 \ldots w_n$ (search probability)

Find BST T that minimizes $\sum_{i=1}^{n} w_i (\text{depth}_T(k_i) + 1)$ ← minimize expected search cost

Enumeration

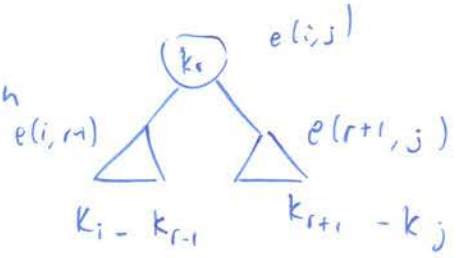
exponentially many trees





Greedy Solution

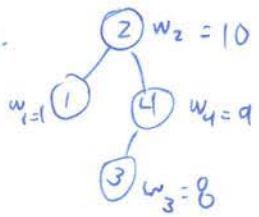
Pick K_r in some greedy fashion
highest weight



$e(i,j)$ = cost of optimal BST on $k_i, k_{i+1} \ldots k_j$

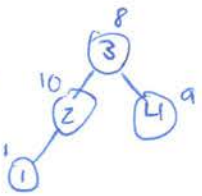
Counterexample

Greedy algorithm would produce.



cost = 54

But,



is better: cost = 49
So, Fully greedy alg wont work

DP Solution

Have to guess which node will be root

$$e(i,j) = \begin{cases} w_i & \text{if } i = j \\ \min\limits_{1 \le i \le j} \left(e(i, r-1) + e(r+1, j) + w_r + w(i,j) \right) \end{cases}$$

, sum of all weights from i to j

Alternating coin game

Row of n coins of values $V_1 \ldots V_n$ n even

select either the first or last coin from

remove permanently, recieve the value

$V_1 \quad V_2 \quad V_3 \quad \ldots \quad V_{n-1} \quad V_n$ n even

Basic States

First player: compare $V_1 + V_3 + \ldots V_{n-1}$ to $V_2 + V_4 + \ldots V_n$

if greater go first
else go second $\Rightarrow$ never lose

Goal: maximize value ~~first~~ assuming you move first.

$v(i,j)$ max value we can definetely win if it is our turn and only coins $V_i \ldots V_j$ remain

$v(i,i)$ one coin left $\Rightarrow$ pick i

$v(i,i+1)$ pick max of two

$$V(i,j) = \max\left\{ \underbrace{\text{range is } i+1, j}_{\text{pick } V_i} + V_i \;,\; \underbrace{\text{range is } i, j-1}_{\text{pick } V_j} + V_j \right\}$$

(opponent moves)

Need to assume opponent will make the best move for them

Solution $v(i+1, j)$ subproblem with opponent picking

$\Rightarrow$ we are guaranteed the min $\{ v(i+1, j-1), v(i+2, j) \}$

opponent picks V_j ; opponent picks $V_i + 1$

$$V(i,j) = \max\left\{ \min\left\{ \begin{matrix} v(i+1, j-1) \\ v(i+2, j) \end{matrix} \right\} + V_j \;,\; \min\left\{ \begin{matrix} v(i, j-2) \\ v(i+1, j-1) \end{matrix} \right\} + V_j \right\}$$

subproblems $\theta(n^2)$

time / subproblems $\theta(1)$

3/17/15

6.046 All-Pairs Shortest Paths

Fernando Trujano

- dynammic programming
- matrix multiplication
- Floyd-Warshall algorithm
- Johnson's algorithms
- different constraints

Recall: single-source shortest paths

- given directed graph $G=(V,E)$, vertex $s \in V$, edge weights $w: E \to \mathbb{R}$
- Find $\delta(s,v)$ = shortest-path weight $s \to v$ $\forall v \in V$

situation	algorithm	time
unweighted ($w=1$)	BFS	$O(V+E)$
nonneg edge weights	Dijkstra	$O(V \lg V + E)$
general	Bellman-Ford	$O(VE)$
acyclic (DAGs)	Topological Sort + Bellman-Ford	$O(V+E)$

↳ Relax edges in topo sort

All-pairs shortest paths

given $G=(V,E,w)$ Find $\delta(u,v)$ For all $u, v \in V$

situation

	- Run SSSP algorithm V times	
general	$\lvert V \rvert \times$ Bellman-Ford	$O(V^2E)$
general	Johnson's	$O(V^2 \lg V + VE)$

Dynamic Programming

1. Subproblems
2. guessing
3. Recurrence
4. Acyclic (topo order)
5. Solve original problem

$d_{uv}^{(m)}$ = weight of shortest path $u \to v$ using $\leq m$ edges

last edge (x,v)

$d_{uv}^{(m)} = \min_{\text{for } m=0 \text{ : } \lvert V \rvert - 1 \text{ for } u,v \text{ in } V} \left(d_{ux}^{(m-1)} + w(x,v) \text{ for } x \in V \right)$ $\quad d_{uv}^{(0)} = \begin{cases} \infty \\ 0 \text{ if } u=v \end{cases}$

$d_{uv}^{(n-1)} = d_{uv}^{(n)}$

if no negative weight cycles → should not repeat vertice

↳ no neg if $d_{vv}^{(n-1)}$ is negative

```
for m in range(1, n):        # n = |V|
    for u in V:
        for v in V:
            for x in V:                      # w(x,v)
                if duv > dux + dxv: duv = dux + dxv   // Relaxation step
```

$\Rightarrow O(V^4)$

Matrix Multiplication

$A, B \to C = AB$ $\quad O(n^3)$ $\alpha(n^{2.8})$ $O(n^{2.37})$

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

define $\odot = +$, $\oplus = \min$ (semiring)

$D^{(m)} = D^{(m-1)} \odot W = W^{(m)}$ (DP)

$$W^{(1)} = \begin{pmatrix} 0 & & \infty \\ & 0 & \\ \infty & & \ddots \end{pmatrix}$$

$D^{(m)} = (d_{ij}^{(m)})_{ij}$ ← matrix $\quad W = (w(i,j))_{ij} \quad V = \{1, 2, 3 \ldots n\}$

Repeated Squaring: $W^{(1)} W^{(1)} \to W^{(2)}$
$W^{(2)} W^{(2)} \to W^{(4)}$
$W^{(4)} W^{(4)} \to W^{(8)}$
$\Rightarrow O(V^3 \lg V)$ ↳ better than $\alpha(V^4)$

Transitive Closure: $t_{ij} = \begin{cases} 1 & \text{if } \exists \text{ path } i \to j \\ 0 & \text{otherwise} \end{cases}$

$\odot = OR \quad \oplus = AND$

ring $O(n^{2.3728})$

Floyd-Warshall algorithm

DP II

① Subproblems $c_{uv}^{(k)}$ = weight of shortest path $u \to v$ whose intermediate vertices $\in \{1, 2, \ldots k\}$

② guessing: is $k \in$ path?

③ $c_{uv}^{(k)} = \min \{ c_{uv}^{(k-1)}, c_{uk}^{(k-1)} + c_{kv}^{(k-1)} \}$ $\Rightarrow$ $O(V^3)$ ← yay
(k not in path)

$c_{uv}^{(0)} = w(u,v)$

if $E = \Theta(V^2)$

```
c = (w(u,v))
for k = 1, 2, ... n
    for u in V
        for v in V
            if c_uv > c_uk + c_kv:
                c_uv = c_uk + c_kv
```

Johnson's algorithm

① Find function $h: V \to \mathbb{R}$ such that $w_h(u,v) = w(u,v) + h(u) - h(v) \geq 0 \quad \forall u,v \in E$

② Run dijkstra on $(V, E, w_h) \to \delta_h(u,v)$

③ claim: $\delta(u,v) = \delta_h(u,v) - h(u) + h(v)$

Proof of claim:

consider path p from u to v

$u = v_0, v_1, v_2 \ldots v_k = v$

$$w_h(p) = \sum_{i=1}^{k} w_h(v_{i-1}, v_i)$$

$$= \sum_{i=1}^{k} \left[w(v_{i-1}, v_i) + h(v_i - 1) - h(v_i) \right] \quad \text{telescope}$$

$$= w(p) + h(v_0) - h(v_k)$$

$$= w(p) + h(u) - h(v)$$

$\Rightarrow$ shortest paths preserved $\square$

① Finding H $\quad w(u,v) + h(u) - h(v) \geq 0$

$$h(v) - h(u) \leq w(u,v) \quad \forall u,v$$

System of difference constraints

~~if neg weight cycle~~

Theorem: IF (V, E, w) has neg weight cycle $\Rightarrow$ no solution to difference constraints

Proof: Consider negative weight cycle $v_0 \to v_k \to v_0$

$$h(v_1) - h(v_0) \leq w(v_0, v_1)$$
$$h(v_2) - h(v_1) \leq w(v_1, v_2)$$
$$\vdots \qquad \vdots$$
$$h(v_k) - h(v_{k-1}) \leq w(v_{k-1}, v_k)$$
$$h(v_0) - h(v_k) \leq w(v_k, v_0)$$

+ (everything cancels)

$$0 \leq w(c) < 0$$

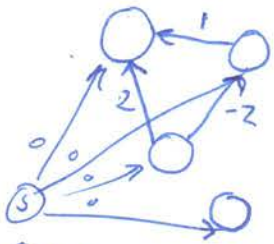
$\square$

Theorem: if (V, E, w) has no neg-weight cycle then $\exists$ solution

Proof:

(constructive proof

add s to V
add (s, v) to E $\forall v \in V$
$w(s, v) = 0$ $\forall v \in V$
$h(v) = \delta(s, v)$ finite $\forall v \in V$



Add s connected to all

$w(u,v) + h(u) - h(v) \geq 0$
$w(u,v) + \delta(s,u) + \delta(s,v) \geq 0$

$\delta(s,v) \leq \delta(s,u) + w(u,v)$ ← Triangle Inequality

□

$O(V^2 \lg V + VE)$

3/20/15

6.046 Recitation

Fernando Trus

Greedy Algorithms

- MST (recap)
- Process scheduling
- Event overlap
- Fractional make change

Greedy Algorithms

Myopic

Best decision now is the best global decision as well. ← how to prove this?

Minimum Spanning Tree

Graph $G \to V, E$

Tree that contains all V vertices, and minimizes edge weight sums

Given a vertex subset $S \subset V$

The edge (v,u) with min weight s.t. $u \in S$ and $v \in V - S$ must belong to the MST of G.

Proof (by contradiction)

Let the MST of graph G by T.

Assume, for purposes of contradiction, let t not contain edge e.

Let's say T contains the edge e' instead

Consider

$e \to$ Min weight edge between vertex $u \in S$ and $v \in V - S$

- see notes

3/31/14

6.046 Lecture

Fernando Troja

Network Flow

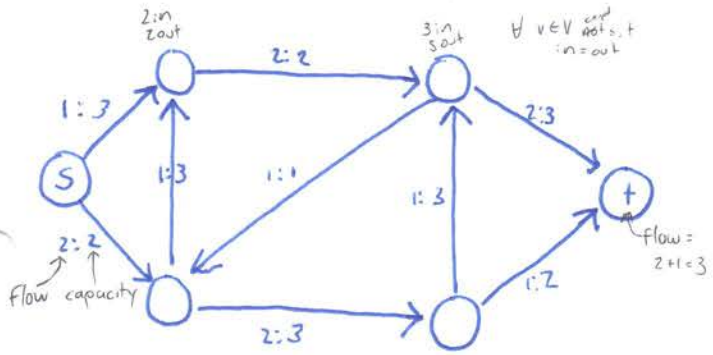
- Flow Networks
- Max Flow problem
- Cuts
- Residual networks
- Augmenting Paths
- Max Flow min cut theorem ↙ Ford Fulkeyon algorithm

Flow Network

$G(V, E)$ directed graph

- Two distinguished vertices source s, sink t
- each edge $(u,v) \in E$, non-negative capacity $c(u,v)$
  if $(u,v) \notin E$. $c(u,v) = 0$

↳ bandwith / amount of traffic that can go through edge.



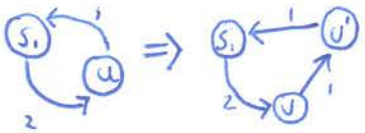
Goal: Increase flow at t by changing flow on other edges. Cannot exceed node capacity

Maximum Flow problem

Given a flow network, G find a flow with maximum value on G

Assumptions

no self-loop edges allowed



Defs

Flow - A flow on G is a function $F : V \times V \to R$ satisfying:

Capacity constraint: For all $u, v \in V$ $F(u,v) \leq c(u,v)$

Flow conservation: For all $u \in V - \{s,t\}$ $\sum_{v \in V} F(u,v) = 0$

Skew-symmetry: For all $u, v \in V$, $F(u,v) = -F(v,u)$ ↙ implicit summation

The value of Flow f, denoted $|f|$ $|f| = \sum_{v \in V} F(s,v) = F(s,V)$

simple problems

$f(x,x) = 0$

$f(x,y) = -f(y,x)$ → don't double count

$f(x \cup y, z) = f(x,z) + f(y,z)$: if $x \cap y = \emptyset$

Theorem $|f| = f(V,t)$

Proof $|f| = f(s,V) = f(V,V) - f(V-s,V)$

$f(V,V) = 0$

$= f(V, V-s)$

$= f(V,t) + f(V, V-s-t)$

$-f(V-s-t, V)$

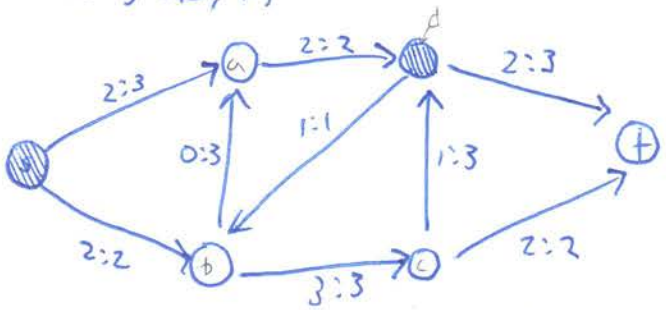
$f(V, V-s-t) = 0$ by Flow conservation

$= f(V,t)$ □

Cuts

A cut (S,T) of a flow network $G=(V,E)$ is a partition of V s.t $s \in S$ and $t \in T$

If f is a flow on G, then the flow across the cut is $f(S,T)$



: S

: T

max flow = d

$f(S,T) = (2+2) + (-2+1-1+2)$

$s,a \quad s,b \quad d,a \quad d,b \quad d,c \quad d,t$

Capacity of cut $c(S,T) = (3+2) + (1+3) = 9$

$s,a \quad s,b \quad d,b \quad d,t$

Value of any flow is bounded by the capacity of any cut

Another characterization of flow value.

Lemma: For any flow f and any cut (S, T) we have $|f| = f(S, T)$

(doesn't matter how you partition)

Proof $f(S, T) = f(S, V) - f(S, S)$

$= f(S, V)$

$= f(s, V) + f(S - s, V)$

$= f(s, V) = |f|$ ($S - s$ does not contain t)

(can figure out maximum flow through network by making arbitrary cuts!)

Residual Network $G(V, E)$

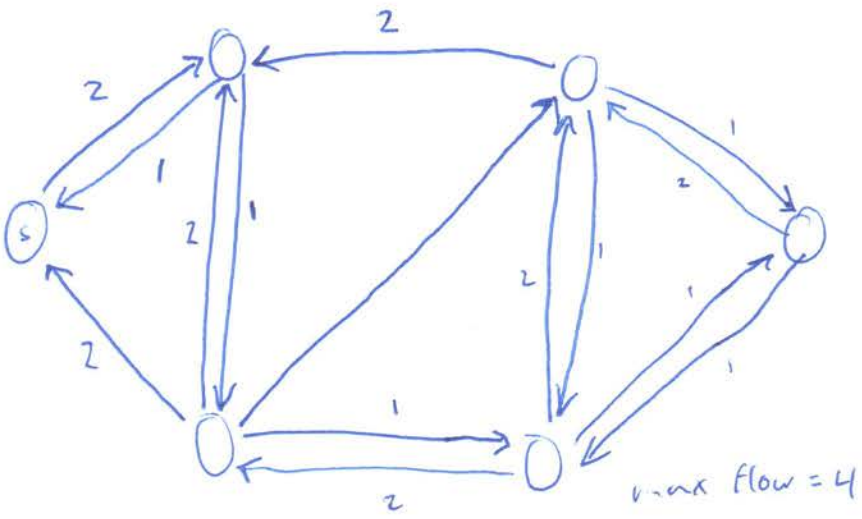
$G_f(V, E_f)$: strictly positive residual capacity

$c_f(u, v) = c(u, v) - f(u, v) > 0$

edges in E_f admit more flow

IF $(v, u) \notin E$, $c(v, u) = 0$ but $f(v, u) = -f(u, v)$

G_f



3/13

6.046 Notes 26.1

Fernando Trujano

Flow Networks

Capacity constraint: For all $u, v \in V$ $\quad 0 \leq f(u,v) \leq c(u,v)$

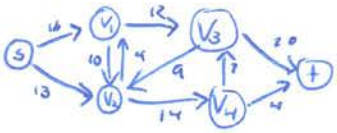
Flow from one vertex to another, non negative and not exceed capacity

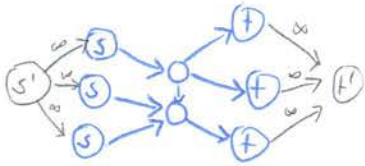
Flow conservation: For all $u \in V - \{s, t\}$ (source, sink)

$$\sum_{v \in V} f(u,v) = \sum_{v \in V} f(v,u)$$

Flow into vertex must come out of vertex

Flow Problem:



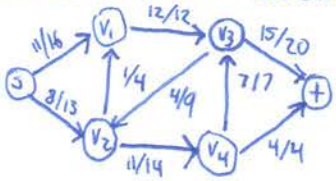


Multiple sources and sinks:

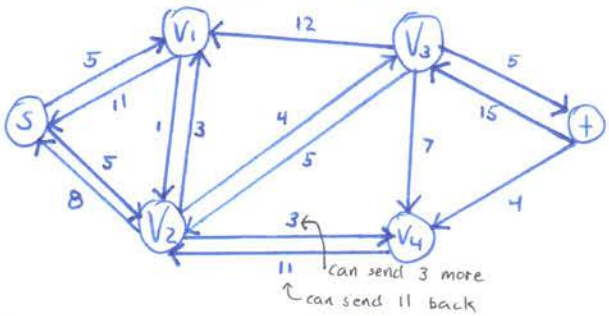
- Create a supersource s' that connects to all sources and $c(s', s) = \infty$
- Create a supersink t', " " $c(t', t) = \infty$

Ford Fulkerson Method

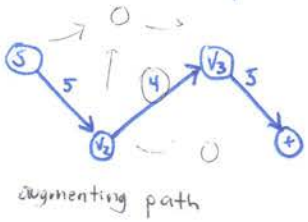
Residual Network - Roadmap for adding Flow to original network



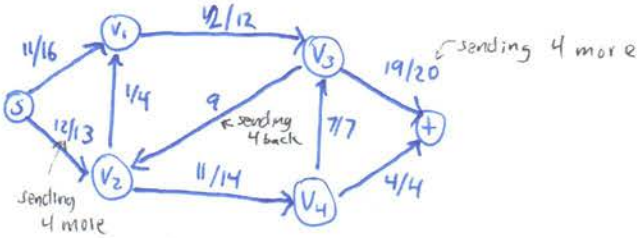
Residual Network ⟹



Augmenting path - path from s to t in the residual network



choosing this ⟹




```
FORD-FULKERSON(G, s, t)
    f = 0
    while there exists an augmenting path p in residual network G_f
        augment flow f along p
    return f
```

6.046 Lecture

Fernando Trujano

Linear Programming

- Examples: politics, flow, shortest paths
- General / Standard form
- Simplex algorithm - Iterating over slack forms

Politics

How to campaign to win an election?

estimate votes obtained / dollar spent advertising in sport of a particular issue

Policy	Demographic		
	Urban	Suburban	Rural
Build roads	-2	5	3
Gun control	8	2	-5
Farm subsidies	0	0	10
Gasoline tax	10	0	2
population	100,000	200,000	50,000
majority	50,000	100,000	25,000

Want a majority in EACH demographic by spending the minimum amount of money.

Algebraic Setup

Let x_1, x_2, x_3, x_4 denote money spent / issue

Minimize $x_1 + x_2 + x_3 + x_4$ (n variables)

(1) subject to $-2x_1 + 8x_2 + 0x_3 + 10x_4 \geq 50,000$ ← urban column

(2) $5x_1 + 2x_2 + 0x_3 + 0x_4 \geq 100000$

(3) $3x_1 - 5x_2 + 10x_3 + 2x_4 \geq 25,000$

(m constraints) $x_1, x_2, x_3, x_4 \geq 0$

optimal solution

$x_1 = 205000/111$

$x_2 = 425000/111$

$x_3 = 0$

$x_4 = 625000/111$

$x_1 + x_2 + x_3 + x_4 = 3100000/111$

x_i : real numbery

Standard Form for Linear programming

- minimize or maximize linear objective function subject to linear inequalities (or equations)

variables: $\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$ objective function: $\vec{c} \cdot \vec{x} = c_1 x_1 + c_n x_n$

inequalities: $A\vec{x} \leq \vec{b}$

max $\vec{c} \cdot \vec{x}$, st ↑ and $\vec{x} \geq 0$

Certificate of Optimality

Is there a short certificate that shows LP soln is optimal?

consider: (equations from page 1)

$$\frac{25}{222} x(1) + \frac{46}{222} x(2) + \frac{14}{222} (3)$$

$$\Rightarrow x_1 + x_2 + \frac{140}{222} x_3 + x_4 \geq 3100000/111$$

$$x_1 + x_2 + x_3 + x_4 \geq x_1 + x_2 + \frac{140}{222} x_3 + x_4 \geq 3100000/111$$

LP Duality

Theorem:

primal form		dual Form
max $\vec{c} \cdot \vec{x}$	=	min $\vec{b} \cdot \vec{y}$
s.t $A\vec{x} \leq \vec{b}$		s.t $A^T \vec{y} \geq \vec{c}$
$\vec{x} \geq 0$		$\vec{y} \geq 0$

Converting to standard form

1) Minimize $-2x_1 + 3x_2$ Negate to $2x_1 - 3x_2$ and maximize
2) Suppose x_j does NOT have a non-negativity constraint $x_j \geq 0$
   x_j replace with $x_j' - x_j''$ $x_j' \geq 0$ $x_j'' \geq 0$
3) equality constraint $x_1 + x_2 = 7$ $x_1 + x_2 \leq 7$ $-x_1 - x_2 \leq -7$
4) $\geq$ constraint translates to $\leq$ by (-1) multiply

Maximum Flow

$$\max \sum_{v \in V} f(s,v) = |f|$$

skew-symmetry s.t $f(u,v) = -f(v,u) \quad \forall u,v \in V$

conservation $\sum_{v \in V} f(u,v) = 0 \quad \forall u \in V - \{s,t\}$

capacity $f(u,v) \le c(u,v) \quad \forall u,v \in V$

Linear constraints!

f_1, c_1, f_2, c_2 (two distinct disjoint opts)

f_1, f_2 single capacity (

$$2f_1(u,v) + f_2(u,v) \le c(u,v)$$

Shortest path

From vertex S

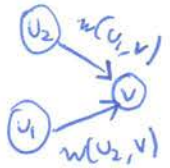
$d[v]$ = weight of a path from $s \to v$

~~min~~ max $\sum_V d[v]$

s.t $d[v] - d[u] \le w(u,v) \quad \forall (u,v) \in E$

$d[S] = 0$

triangle inequality



$d[v] - d[u_1] \le w(u_1,v)$

$d[v] - d[u_2] \le w(u_2,v)$

$d[v] = \text{MIN}($ $)$

Simplex Algorithm

worst case $\binom{m+n}{n}$ iterations

Flow:

Represent LP in slack form

Convert one slack form into an equivalent slack form whose objective value has not decreased and has likely increased.

Keep going till the opt solution becomes obvious

See notes

3/3

4/10/15

6.046 Recitation

Fernando T

Reductions

NP-completeness

Definitions:

P: Set of decision problems D that are polynomial time solvable

NP: Set of decision problems that are verifiable in polynomial time

NP-Hard: " " " " " reducable from any problem in NP

NP-Complete: " " " " NP and NP-hard

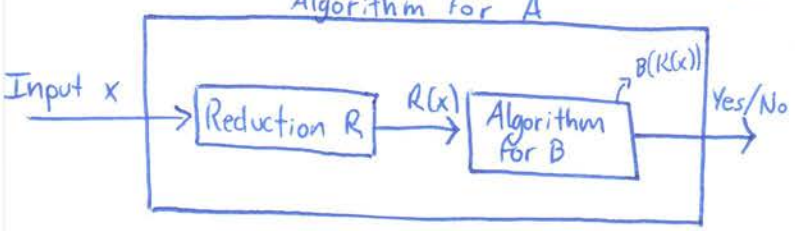
Reductions: Compare the hardness of different problems

Karp Reduction

Let A be a problem that takes $x \in X$ as input and spits out $\{0,1\}$ as output

Let B " " $y \in Y$ " "

A is poly-reducable to B if there exists a function $R: X \to Y$ s.t $A(x) = B(R$ (poly)



IF B can be solved in poly time $\Rightarrow$ A can be solved in poly time

If A is "hard" then B is "hard" as well

Reducing Hamiltonian Cycle to Hamiltonian Path

HAM-CYCLE is NP complete. Want to prove HAM-PATH is NP complete too

HAM-CYCLE - Given a directed graph $G = (V,E)$ is there a cycle that visits every vertex exactly once

HAM-PATH - " " is there a path "

Step 1: Show HAM_PATH is in NP. (can verify in polynomial time)

Input $x \to$ Graph G. Want to prove that $\exists$ V(x,y) that runs in poly time

$y \to$ Path P that we want to check.

Step 1 $\to$ Does y touch each vertex exactly once? $O(V^2)$

Step 2 $\to$ Is every edge in p in G? $O(VE)$

$\Rightarrow$ HAM PATH IS IN NP

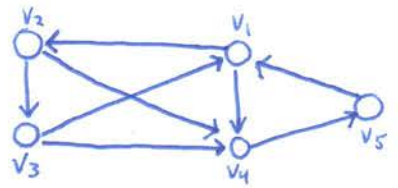
Step 2 - Show HAM_PATH is NP-HARD

Karp Reduction from HAM-CYCLE to HAM-PATH

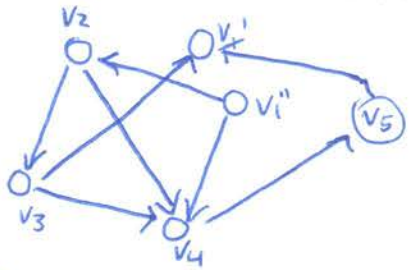
a) Given an instance of HAM_CYCLE G. Want to create input to HAM-PATH.

Pick some arbitrary vertex $v \in G$ and split it into v' and v''

All edges into v are now into v' and all edges out of v are out of v''



Pick v_1



$v_1'' \to v_2 \to v_3$
$\to v_4 \to v_5 \to v_1'$

→ If G has a ham cycle then G' has ham path. Because we can use edges in our original cycle but now our path starts at v_1'' and ends at v_1'

← If G' has a ham path then G must have a ham cycle.

The hamiltonian path p in G' must start at v_1'' and end at v_1'

$\Rightarrow$ HAM PATH is NP HARD □

4/14/15

6.046 Lecture

Fernando Truja

Approximation Algorithms

Definitions

Vertex cover
Set cover
Partition
} NP-complete or NP-hard

Approximation Algorithms and Schemes

An algorithm for a problem of size n has an approximation ratio $\rho(n)$ if for any input, algo produces a solution with cost c s.t

$$\max\left(\frac{c}{c_{opt}}, \frac{c_{opt}}{c}\right) \leq \rho(n) \text{ - approx algo}$$

An approximation scheme takes as input $\epsilon > 0$ and for any fixed ϵ, the scheme is a $(1+\epsilon)$-approximation algorithm

poly in n, but not necesarily in ϵ

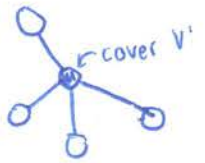
$O(n^{2/\epsilon})$ running time

Polynomial time approx scheme PTAS

Fully PTAS:
poly in n and $1/\epsilon$ $\Rightarrow$ $O(n/\epsilon^2)$

Vertex Cover

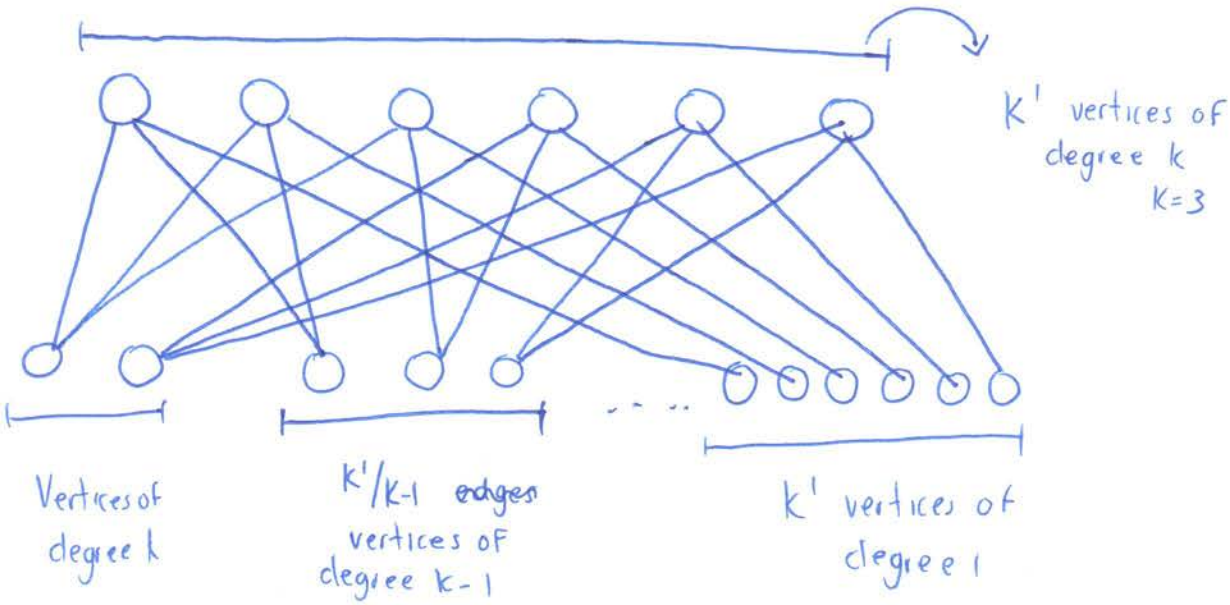
- Set of vertices that cover all edges



Undirected graph $G(V, E)$

Find subset $V' \subseteq V$ s.t if (u, v) is an edge of G then either $u \in V'$ or $v \in V'$ or both

- Find a V' so $|V'|$ is minimum



Algo could pick all bottom vertices BUT top: optimal: $= k! = n$

So $k!\left(\frac{1}{k}+\frac{1}{k-1}+\dots 1\right) \approx k! \log k$

↳ still dependent on n. ▽

Approx Vertex cover is a 2-approximation algorithm

Let A denote the edges that are picked

$C = 2|A|$ vertices picked.

Show that $C_{opt} \geq |A|$ Need to cover every edge including all edge in A.

$C \leq 2C_{opt}$ Need to pick a different vertex from each edge in A

Approx Vertex Cover

$C \leftarrow \phi \quad E' \leftarrow E$

while $E' = \phi$

Pick $(u, v) \in E$ arbitrarily

$C \leftarrow C \cup \{u\} \cup \{v\}$

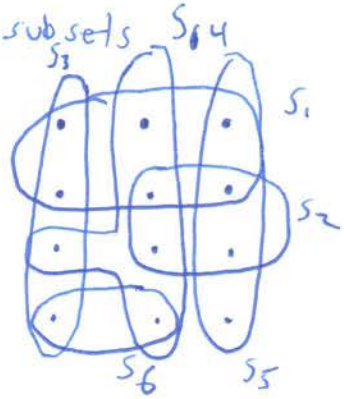
Delete from all edges incident on u or v

return C

et cover :

Given a set X and a family of (possibly overlapping) subsets $S_1, S_2, \ldots S_m \subseteq X$ s.t $\bigcup_{i=1}^{m} S_i = X$

Find $C \subseteq \{1, 2, \ldots m\}$ while minimizing $|C|$ s.t. $\bigcup_{i \in C} S_i = X$



opt : S_3, S_4, S_5
$|C'| = 3$

Approximate-Set-Cover is a $(\ln(n)+1)$ approx algo

Pick largest S_i, remove all elements in Section X and others S_j

Proof : Assume there is a cover C_{opt} $|C_{opt}| = t$ let X_k be set of elements in iteration k $(X_0 = X)$ $\forall k$, X_k can be covered by t sets

(Pidgeonhole principle

$\Rightarrow$ one of them covers at least $\frac{|X_k|}{t}$ elements

$\Rightarrow$ algo picks a set of (current) size $\geq \frac{|X_k|}{t}$

$\Rightarrow \forall k \quad |X_{k+1}| \leq \left(1 - \frac{1}{t}\right) |X_k|$

$|X_k| = 0$, done $\Rightarrow \forall k, \; |X_k| \leq \left(1 - \frac{1}{t}\right)^k n$ constant rate of shrinkage

cost $= k$ $e^{k/t} n < 1$

Footnote)

3/15/15

6.046 Quiz II Review

Fernando Trujan

- Dynamic Programming
- Greedy Algorithms
- Network Flow
- Linear Programming
- Complexity and Approx Algorithms
- All-pair shortests paths

DP Vertex Cover Problem

A vertex cover of an undirected graph is a subset of its vertices s.t for every edge (u,v) in the graph, either v or u or both are in vertex cover.

- Find minimal vertex cover for a tree

Step 1: $d[v]$ → Size of Minimum vertex cover for a tree rooted at v

Step 2: Guessing: Is vertex v in minimal vertex cover or not?

Step 3: $d[v] = \min \begin{cases} 1 + \sum_{u \in v.children} d[u] \\ I(v \text{ has left child}) \left(1 + \sum_{u \in v.left} d[u]\right) + I(v \text{ has right child}) \left(1 + \sum d[u]\right) \end{cases}$ Assume binary tree

Step 4: Solution = $d[tree.root]$

subproblems: $O(n)$ Time / Subproblem $= O(1)$ => Total time: $O(n)$

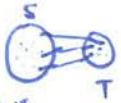
All-pair shortests paths

DP $d_{uv}^{(m)}$: Distance to go from $u \to v$ in at most m edges

$$d_{uv}^{(m)} = \min\left(d_{ux}^{(m-1)} + w(x,v)\right) \Rightarrow O(V^4) \quad \text{or} \quad d_{uv}^{(m)} = \min\left(d_{ux}^{m/2} + d_{xv}^{m/2}\right) \Rightarrow O(V^3 \log V)$$

Ex: Find diameter of weighted graph in $O(V^3)$
Sol: Run APSP, look through all pairs and pick max

Greedy Algorithms

MST: cut property: 

→ lightest edge that connect S and T will be in an MST

Know Prim's and Kruskal's

Ex: All edges in graph G have unique weights ⇒ MST is unique

Ex: simple Cycle → edge with the lowest weight is in some MST
→ edge with heighest weight is not in an MST

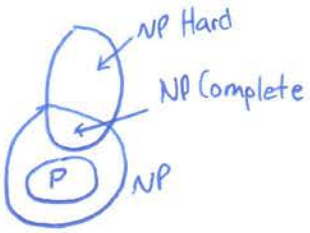
Flows ① $F(u,v) \leq c(u,v)$
② Conservation of Flow
③ Flow leaving S = Flow entering T

max flow = c(min cut)

Ford Fulkerson - slow $O(|E|F)$ BFS to find augmenting paths to $\min(VE, |F|)$

Edmon Karp - $O(VE^2)$ ← bound # augmenting paths

NP Hard / Complete Problems



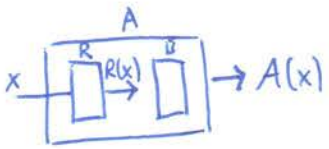
P = { problems solvable in poly-time }

NP = { decision problems with w/ efficient verification algorithm } (↗ poly-time)

= { decision problems solvable in non-deterministic poly-time }

NP-hard = Problem D is NP-Hard if $A \leq_p D$ $\forall A \in NP$

Reduction: Problem A poly-time reduces to B. $A \leq_p B$ $A \leq_p B$
if $\exists$ a poly-time alg R s.t $A(x) = B(R(x))$



NP-complete: Problem D is NP complete if $D \in NP$, and D is NP Hard

Important Stuff

- 3 SAT :
  - 3DM
  - Supermario, 4 port
- Hamiltonian Cycle / Path

Example

Ham Path $=_p$ Ham Cycle

$G=(V,E)$ ⇒ Create G'

1) new vertex v to G
2) Add edges to/from all vertices from/to V

Ham Cycle in G' ⇒ Ham Path in G

consider $u_1 \to u_{i-1} \to v \to u_{i+1} \rightsquigarrow u_n \to u_1$

rearrange $v \to u_{i+1} \rightsquigarrow u_n \to u_1 \rightsquigarrow u_{i-1} \to v$

Hamiltonian Path in G

TODO show other direction

Ham path in G ⇒ Ham Cycle in G'

$u_1 \rightsquigarrow u_n$

$v \to u_1 \rightsquigarrow u_n \to v$

Approximation Algorithms

- Algorithm for a problem of size n has an approximation ratio of $\rho(n)$ if it produces a solution s.t the cost C of the solution is at most $\rho(n)$ Factor away from optimal.

PTAS - Poly Time Approximation Scheme
Takes in ϵ and a problem instance and produces $(1+\epsilon)$ approx of optimal $O(n^{\epsilon})$

FPTAS - Fully PTAS - Same thing but runtime is poly in $n, \frac{1}{\epsilon}$ $O(\frac{n}{\epsilon})$

Vertex Cover - Repeat:
take any $(u,v) \in E$
add u,v to cover-set
remove all edges From u and v

4/28/15 6.046 Lecture Fernando Trujan

Distributed Algorithms

Distributed Algorithms - run on multiple machines / processes. Complicated and error prone

- Infinite state interactive state machines

Distributed Networks

$\Gamma(u)$ = set of neighbors of vertex u.

vertex = process

Synchronous Network Model

- Each process has out/in ports.
  - processes don't know who it's ports chanels connect to.
- Processes may be indistinguishable from each other

Synchronous rounds

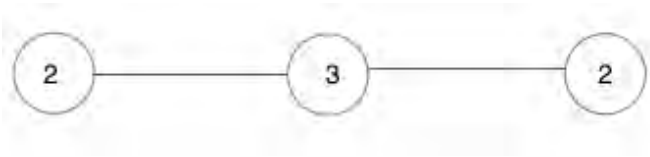
- in each round
  each process usses current state to send message to alls
  processes recieve message and change state based

Fernando Trujano February 12, 2015

6.046 Problem 1-1

Collaborators: *Erika Lu, Benjamin Gunby*

a) In the following graph, the optimal solution would be to pick both of the restaurants with value of 2 for a total value of 4. This greedy algorithm will however, pick the middle restaurant with value 3, and remove all the neighbors for an incorrect solution with a total.



b) We use a dynamic programming approach. Where for each subproblem we consider two scenarios. We either have the restaurant in the final solution or we don't. We'll call these $R(i)$ and $R'(i)$ respectively. The base cases are as follows:

$R(i) = p(i)$
$R'(i) = 0$

The recursive solution for each of the equations is:

$R(i) = p(i) + \sum_{j \in children(i)} R'(j)$

$R'(i) = \sum_{j \in children(i)} max(R'(j), R(j))$

For the first equation, we assume we are including restaurant i so we add it's profit plus the sum of the profit we could get if we don't include all of i's children. For the second equation, we assume we are not including restaurant i so we don't include it's profit and add the max profit we could get from each children (we could add each one or not).

We return the max of $R(n)$ and $R'(n)$, where n is the total number of restaurants.

Proof: Since each sub problem (by definition) produces the correct output, our final solution will be correct.

Analysis: Number of subproblems: $O(n)$
Time per subproblem: $O(1)$
Total Runtime: $O(n)$

c) We begin by making an array N of size n where n is the total number of vertices in the graph G. We go through each vertex and count the number of neighbors of that vertex (this is the degree). We place each vertex v in $A[degree(v)]$. We then check if there are any restaurants in $N[0]$. If there are, we "pick" these restaurants and remove them from the array. Otherwise, we look at all of the vertices in $N[1]$ and pick one at random. We "pick" this restaurant, remove it's neighbor (a) and update all of a's neighbors by shifting them one to the left in the array (effectively updating their neighbor counts). We repeat this process, each time "picking" a new restaurant. The algorithm terminates when there are no elements in $N[0]$ or $N[1]$. At this point, we return all of the picked restaurants.

Proof: Every time we pick a restaurant, we pick the one that will remove the least number of restaurants from consideration.We are essentially always removing leaves from the tree, since we know that the most number of nodes will reside at the deepest level of the tree.

Analysis: The initial construction of N takes $O(n)$ time since we only visit each restaurant once. Since the graph is a tree, we know that there are at most $n-1$ edges. Since we can only remove each edge once, we can update at most $n-1$ times. Therefore, the total running time of this algorithm is $O(n)$

d) For each node, we consider two cases and recursively calculate the maximum profit for each. In one case, we don't include this node in our final solution, we have to consider the $n-1$ remaining nodes. If we do include the node, we have to consider at most $n-2$ nodes, since we removed the node and its neighbor. The recurrence is as follows $T(n) = T(n-1) + T(n-2)$. This is the fibonacci recurrence and it solves to $T(n) = \varphi^n$

Fernando Trujano February 12, 2015

6.046 Problem 1-2

Collaborators: *Erika Lu, CLRS*

a) If a set of requests is on or on the boundary of the same square this means that their Eucledian distance $d \leq 1$. This is because the squares are $\frac{1}{2} \times \frac{1}{2}$ and the maximum distance two points can be apart in this square is equal to 1.

b) This algorithm will use a divide and conquer algorithm to correctly compute if any two points are within distance one of each other in $O(nlogn)$ time. In the trivial case where there are only two points, we simply check if their distances are greater than one and return them if they are. Otherwise, we find a vertical line that bisects the set of all points into R_L and R_R such that $R_L = |L|/2 = R_R$ (we can break parity ties by taking the ceil of $|L|/2$ for R_R) where the points in R_R are to the right of the line and the points in R_L are to the left.

For each one of these subsets, we make a recursive call to the algorithm making sure to pass in the correct (updated) L_X and L_Y for each of the subsets. The "combine" part of this algorithm can assume that both the left and right subsets don't have any points distance 1 from each other, if that was the case, the algorithm would have returned before reaching this step. With that assumption, we only need to check the points of both sets that are near our vertical bisecting line, since these are the only points that could possibly be closer than 1 in the entire combined set. In order for two points to be closer than 1 to each other they must be a distance of 2 away from the vertical bisecting line, knowing this, we only have to check points residing in this area. We can create an array consisting of only the points that lie in this area and attempt to find another point that is 1 distance or less apart. If we find a pair, we return it, else, we continue with the merge.

Proof: Since we only compare a finite amount of points on each merge (only the opens that lie one distance away from either side of the dividing line). Since both sides have already been checked, for each point close to the line, we only need to check 7 adjacent squares (using the logic from a). If the algorithm never returns a point each sub problem has been verified to have no closes points and the entire set composed of all subsets has also been checked. At this time we can correctly conclude that there are no pairs of points a distance of 1 apart.

Analysis: This algorithm gives the following recurrence: $T(n) = 2T(n/2) + O(n)$ since we spit the problem into two problems of half of the size and each comparison takes $O(n)$ time. This recurrence, is identical to merge sort, and it solves to $O(nlogn)$

c) The algorithm described above can be modified to detect 3 pairs that are a distance of 1 or less apart. We begin by changing our base (trivial) case to when there are only 3 points in the input. In this case, we compare each point against each other to detect if they are

1 or less distance apart. We continue the algorithm as before, but when merging instead of checking 7 adjacent squares we will have to check 15 (2 for each square plus yourself) of them. If we detect three points that are close together, we return them, else we continue with the algorithm. The running time changes only by a constant factor so asymptotically it remains $O(nlogn)$

Fernando Trujano February 19, 2015

6.046 Problem 2-1 and 2-2

Collaborators: *Erika Lu, Amol Bhave*

Problem 2-1)

a) Description: This naive algorithm checks every possibility of the source string to see if it matches the the pattern string. We iterate through the entire source string, starting at index 0, and checking if the substring from 0 to m of the source string matches the pattern string (taking into account that * can be either an 'a' or 'b'). If it matches, we add our current index to a list. We then increase the index i by 1 and check the substring from i to i+m. We repeat this process until the algorithm terminates when i is equal to the length of the source string - length of pattern string.

Correctness: Since we are checking every single possibility, the algorithm will always return the correct output.

Analysis: Each check of the algorithm requires $O(m)$ checks, since (in the case of a match), we will have to check every index of the pattern string. We are doing $O(n)$ checks (one for each index of the source string). Therefore, this algorithm will run in $O(nm)$ time.

b) This algorithm takes inspiration from cross-correlation used in signal processing to determine how similar two signals are to each other. We begin by mapping each string to a polynomial. Our chosen mapping, which will be explained later is as follows:

$$A = 1, B = -1, * = 0$$

We chose to represent the coefficient of the term with degree i as the index of the string. For example, the string "aba" would map to $1 - x + x^2$ or $< 1, -1, 1 >$ as a vector. To get the cross-correlation, we map each string S and $reverse(P)$ to their corresponding polynomials using the method described above and get S' and P' respectively. Notice that we reverse the pattern string, before the multiplication. This is because of the inherent nature of convolution where the second polynomial is essentially reversed. By reversing the polynomial before the multiplication, we guarantee that we will get the cross-correlation that we are looking for (after the multiplication). This approach also compares parts of each substring with each other, but we only want the entire pattern string compared to an equally long substring of the source string. Since we want to make sure we compare the entire pattern string, we need to strip away the first and last $m - 1$ terms of our product. Each coefficient (for a term of degree i) in this polynomial represents how similar the two strings are after the pattern has been shifted i times). For example, the polynomial represented by the vector $< 2, 0, 2 >$ with a calculated max of 2 suggests that the pattern string matched at index 0 and 2.

It is easy to calculate what the maximum coefficient will be. This happens in the case where we have an exact match between the two compared strings and we get a coefficient equal to $m - num(wildcards)$. This is because for every character exactly matched we get a +1 for the coefficient since $1^2 = 1$ and $(-1)^2 = 1$. So, in the event of an exact match the coefficient will be m. If the pattern string has a wild card (*) we need to account for this by updating our max to m minus the number of wild cards in the pattern string.

Once we have calculated the maximum, we can simply take the "peaks" of the cross-correlation. This is where the strings are most similar, in fact, this is when the strings are an exact match. So we just need to scan through our cross-convolution for coefficient values equal to the max and take note of the term degree they where in. After scanning through, we simply return a list of all of the matches. We chose the following mapping in order to maximize the value of our cross-correlation when two strings are similar and minimize the value when they are not. For example, the cross-correlation of the two strings 'a' and 'b' would be -1. 'a' and 'a' (or 'b' and 'b') would have a value of 1 because of our chosen mapping. '*' is chosen to map to 0 so that it neither increases or decreases the cross-correlation and essentially remains a 'neutral' state.

Example: $S =$ "$ababbab$", $P = ab*$. The first step in the algorithm is to convert both of the strings into the polynomial using the mapping describe above. S maps to $1 - x + x^2 - x^3 - x^4 + x^5 - x^6$ in vector form, this is $< 1, -1, 1, -1, -1, 1, -1 >$. Similarly P maps to $< 1, -1, 0 >$. For reasons described above, we need to reverse one of the polynomials, let's say $P' = < 0, -1, 1 >$. Multiplying P' and S yields our cross-correlation $C = -x + 2x^2 - 2x^3 + 2x^4 - 2x^6 + 2x^7 - x^8$. In vector form we have $C =< 0, -1, 2, -2, 2, -2, 2, , -1 >$. The comparison starts comparing the last character of the pattern string with the first character of the source string, leaving the beginning of the pattern string out. Since we are only focused on exact matches with the pattern string, we need to discard the first and last $m-1$ elements. This leaves us with $C' =< 2, -2, 2, -2 >$. The maximum we are looking for is $3 - 1 = 2$ since $|m| = 3$ and there is one wild card. All that is left to do is to scan C' and see when we see a coefficient greater than or equal to this max. This clearly happens at 0 and 2 so the algorithm will return $[0, 2]$.

Runtime: The runtime is equivalent to however long it takes to multiply the polynomials. This is $O(n^2)$ using a simple multiplication approach or $O(nlogn)$ using FFT as described in lecture. See c) for a more descriptive answer.

c) Constructing each of the polynomials takes $O(n)$ and $O(m)$ respectively. Reversing one of the polynomials (m is shorter so we'll reverse that), will take $O(m)$. Since $m << n$ this is $O(n)$. Multiplying these polynomials using the FFT algorithm takes $O(nlogn)$ time. We then strip the resulting polynomial in $O(m)$ time and scan through the polynomial in $O(n - 2m)$ time. Asymptotically, this algorithm will run in $O(nlogn)$.

d) We can use the algorithm described in b) with a few modifications in order to use 4

symbols (A,C,G,T). We need to come up with another mapping for these four values that results in a positive number when two matches are multiplied together and a negative (or invalid (to be explained later)) when two mismatches are multiplied. The mapping is as follows: For the source string, we use:

$$A = 1, C = -1, G = i, T = -i$$

and for the pattern string we use

$$A = 1, C = -1, G = -i, T = i, * = 0$$

Notice that the mapping for G and T is interchanged for each of the strings. This is to achieve the desired addition when two matches are multiplied together since $i \cdot (-i) = 1$ and $i \cdot i = -1$. We keep the wild card as 0 for similar reasons as described in b). The algorithm follows as described in b) with the only difference being at the end when we check the final polynomial C'. We know that any coefficient that is not a real number cannot be a match since it was at some point multiplied by a value that was not it's match and thus resulted in a non real answer. Since only the mapping of the initial polynomials is different, the algorithm has the same running time and can be proved like b).

Example: $D = ACGACCAT$ and $P = AC * A$. The first step is to match each string to its corresponding polynomial using the specific mappings described above. Doing this we get $D = 1 - x + ix^2 + x^3 + -x^4 + -x^5 + x^6 - ix^7$ and $P = 1 - x - x^3$In vector form this is $D =< 1, -1, i, 1, i, i, 1, -i >$ and $P =< 1, -1, 0, 1 >$. Multiplying D and the reverse of P yields $C = -ix^{10} + (1+i)x^9 - 2x^8 - ix^7 + 3x^6 - (2-i)x^5 - (2+i)x^4 + 3x^3 - (1-i)x^2 - x + 1$. For this example the max is $4 - 1 = 3$ because $m = 4$ and there is one wildcard. C in vector form is $< 1, -1, (1-i), 3, (-2-i), (-2-i), 3, -i, -2, (1+i), -i >$. Trimming C from the first and last $m - 1$ coefficients gives us $C' =< 3, (-2-i), (-2-i), 3, -i >$ from here, we can scan through C' and see that we have matches at 0 and 3 so the algorithm will return $[0, 3]$.

Problem 2-2)

a) Description: When $h1$ and $h2$ are the same height, the combine algorithm is as follows: In essence, we join the root nodes of both trees and do an insertion of the key at the new root node. This will take care of any splitting that may be necessary if the new root has too many keys.

Correctness: Both of the initial trees were valid B trees to begin with and since all of the keys in T_1 are strictly smaller than all of the keys in T_2 we are allowed to join the two root nodes and maintain all of the B tree properties with one exception. This occurs when joining both of the root nodes creates a node with more than $2t - 1$ keys. In this case, the regular inserting algorithm will fix the issue, by correctly splitting the node, increasing the height of the entire tree by one. The correctness of this operation follows from the correctness of B tree insertion described in the textbook.

Runtime: This algorithm takes constant time since it is completely independent of the height of the tree. This is because we are essentially "inserting" the new key at the top level and splitting the root node at most once if necessary. The runtime is $O(1)$

b)

Description: This approach is very similar to A except we insert the left b tree and the key at a different level. We start by adding k to the left child (as another key of this node) of the root of T_2. Since k is less than any key in T_2 we should add this key to the leftmost of the node. We can then combine that node with the root of T_1 and proceed with splitting if necessary.

Correctness: Since both T_1 and T_2 are valid B-tree and we are inserting the shorter tree on the second level of the taller tree, we will end up with all of the leaves at the same level. Additionally, since we are adding a key k to the affected node, we assure that we have at least one more children than we have keys (since T_2 was already a valid B tree)

Runtime: Inserting k and T_1 both take constant time and we are at most going to do a constant amount of splits (1 or two) so the total runtime of this algorithm remains $O(1)$

c)

Insert: Notice that the height of a subtree only changes when a node is moved up a level after a split. This is the basis for our slight modification to the insert algorithm. Anytime we insert a node, it will be inserted at the leaf, so we should set the height of it's subtree to 0. Then, if at any point the root goes up a level in the tree (after a split) we create a new node with its height equal to 1+ height of children. Since the height of a tree only changes on this occasion, as long we update the augmentation any time this happens, we will have an accurate height for all of the subtrees after any number of inserts.

Delete: Since deletions on B trees only happen at the root (never at the leaves) the height of every sub tree will remain constant regardless of the number of deletions. This is because when the height of the tree will only change at the root of the tree, which does not affect the height of any of the sub trees. It is important to notice that during a deletion we don't swap the entire node, but only the key of that node, so we don't have to worry about updating it's height (since the node itself did not move).

d)

Description: This algorithm uses a very similar approach as b), but makes use of the tree augmentation defined in c) to correctly combine T_1 and T_2 of arbitrary height ($h1$ and $h2$ respectively). The algorithm is as follows: Find $h1$ and $h2$ by looking at the height augmentation of the respective roots of each tree. We can easily find the larger tree by comparing both heights, we'll refer to this as T_L and the smaller tree as T_S.The next step

is very similar to b), but instead of placing the smaller tree at the second level, we need to insert T_S on the $|h1 - h2|$th level of T_L. We do this by walking down T_L $|h1 - h2|$ times. If $T_L = T_2$ we walk to the left every time otherwise, we walk to the right. This is because all of the keys in T_1 are strictly smaller than all of the keys in T_2. The algorithm follows as described in b).

Correctness: The correctness of this algorithm follows from the correctness of b) with the only difference being that we are inserting the smaller tree at a different level on the larger tree. Since we insert the smaller tree at the $|h1 - h2|$th level of the larger tree, we guarantee that all of the leaves remain at the same level. This is maintained even if the merging requires more that one split, since the height of the entire tree will change by at most one.

Runtime: Looking and comparing the height augmentations takes constant time. Walking down the tree to to the appropriate level takes $O(|h1 - h2|)$ since we need to take $|h1 - h2|$ steps to get to that level. The insertion will take $O(|h1 - h2|)$ time since in the worst case we will have to split at every level up to the root node. Overall the runtime is $O(|h1 - h2|)$

Fernando Trujano February 26, 2015

6.046 Problem 3-1

Collaborators: *Erika Lu*

a)

Since we are splitting the data structure into $u^{1/3}$ groups of $u^{2/3}$ numbers each, the original recurrence from CLRS changes to:

$$T(2^m) = T(2^{2m/3}) + O(1)$$

where m is lgu. We will keep the original indexing (changes explained later), so we do a constant amount of work per level. Like the original recurrence, we use changing variables: Let $S(m) = T(2^m)$, now we have:

$$S(m) = S(2m/3) + O(1)$$

By case 2 of the master method, the recurrence solves to $S(m) = O(lgm)$. Substituting back we see that $T(2^m) = O(lglgu)$. Notice that the asymptotic runtime remains the same as when we divided the structure into $u^{1/2}$ groups of $u^{1/2}$ numbers each.

Operations on the structure remain relatively unchanged, we the only difference being in the *HIGH*, *LOW*, and *INDEX* functions since these are the only that are dependent on the size of the groups. The functions are as follows:

$$high(x) = x/u^{2/3}$$

$$low(x) = xmodu^{2/3}$$

$$index(x, y) = x \cdot u^{2/3} + y$$

b) To implement the desired changes, we need to make sure to modify the pseudocode to exclude the maximum element from lower-level vEB structures. In essence, we can accomplish this by mirroring (in each function) any operations that have to do with the max, the rest remains the same (as explained in CLRS 20.3) with some minor edge cases that will require special attention. The functions are as follows:

INSERT: Insert remains the same but we mirror $x < V.min$ with a new case for $x > V.max$ This means that after line 4 (in the CLRS pseudo code), if x ¿ V.max we want to exchange x with V.max (mirroring what we do if x ¡ V.min). If V.min is equal to V.max (this happens when there is only one node in the tree) we set y equal to the current V.min and then update V.min a V.max with the min and max of x and y respectively. We need to return after this, effectively skipping lines 3-11 on CLRS Pseudo Code

SUCCESSOR: Also remains the same, but we mirror when min is not found in the predecessor. The original implementation of **successor** assumes that the max is included in the lower levels. Since this is no longer the case, we need to do a check to see if the max is the successor before returning NIL in line 13.

DELETE: Again we use the same implementation, mirror cases from the min to the max and account of some edge cases. In line 13, we add the following check: if x ¿ V.max: exchange x with V.max Aditionally, if x not equal to NIL after line 11. we need to exchange x with v.max. We also need to consider the case when the element we are trying to delete is the current maximum. In this case, we need to replace the max with the predecessor of the current max before deletion.

Since for each function, we "mirror" anything we do to the minimum to the maximum our runtime will at most double. Additionally, one of the additional checks for edge cases will be large enough to affect the overall runtime. Asymptotically, the runtime for all of the functions will remain the same.

Fernando Trujano March 4, 2015

6.046 Problem 4-1, 4-2

Collaborators: *Erika Lu*

Problem 4-1

a) The proposed data structure consists of a regular FIFO queue call it Q, and a doubly linked list with pointers to both the head and the tail, call this M. Q keeps track of the *ENQUUE* and *DEQUEUE* operations and M is used to keep track of the minimum,

b) The operations are as follows:

ENQUEUE(x) First, push x unto Q as any FIFO queue. Next we insert x onto the end of M, and check the element adjacent to it, call this j. If $x < j$ we pop off j from M by setting $j.next = j.next.next$. We repeat this process, checking the next adjacent element until we reach the end of the list (no adjacent element) or $x > j$. Note that x will never equal j since all elements are distinct.

DEQUEUE(x) First, pop the first element from Q and return it. To update M we need to look at the head of the list and check if it is equal to x. If so, we remove the element from M. If the head of the element is not equal to x, no updating of M is required.

FIND-MIN(x) Return the end of M by following the pointer. No updating of M or Q is required since this operation does not mutate the data structure.

c) ENQUEUE and DEQUEUE both rely on the original FIFO queue implementation of Q with minor changes to correctly modify M. Therefore, this proof will be focused on why FIND-MIN(x) will always return the correct answer.

Invariant 1: When an element x is QUEUED all elements greater than x will never be the minimum in the queue.

Proof: When we add an element x, we add it to the end of M and then "bubble" it up until any elements that were bigger than x are popped from M. It is safe to remove these elements, because they will never be the minimum of the queue. This is because Q is a FIFO queue and there are only two ways for the min to be updated:

Case 1: A smaller element is QUEUED. If an inserted element x is smaller than all elements in Q it will bubble up to the head of M through the ENQUQUE operation. The minimum is updated.

Case 2: The current minimum is DEQUEUED. Since Q is a FIFO queue, the only way for the

min m to be dequeued is for all of the other elements inserted before m to be dequeued first. Because of this, we can safely pop any elements bigger than m from M when ENQUEUEining m since once m is inserted the other elements will never be the min of the queue.

Invariant 2: The minimum of the queue is always located at the head of M.

Proof: This follows from the implementation of insert and is maintained through DEQUEUE() operations. Assume DEQUEUE returns x. There are two cases:

Case 1: x is at the head of M. This means that we are dequeueining the current minimum so M will need to be updated. We do this by removing the head (min) of M effectively setting the new head (new minimum) to the next element in the doubly linked list. This element will be the second smallest element in the queue because of INSERT's implementation (it bubbled all the way to the second spot, but was larger than the absolute min). The new minimum of the queue will still be located at the head of M.

Case 2: x is not at the head of M. This means that x was not the minimum of the queue. In this case, DEQUEUE simply removes the element from Q. The only way for x to not be at the head of M is that a smaller element was Inserted after x. Because of the way insert is implemented, this operation would have already removed x from M. Therefore, we can conclude that x is not in M and M is correct as is, with the minimum still at the head.

In both cases, the minimum remains at the head of M, maintaining the invariant.

The correctness of all operations follow from the above mentioned invariants.

d) Overview: The worst case for *ENQUEUE* occurs when we add m elements in ascending order and then insert x where x is smaller than all elements. This last operation will take $O(m)$ time since x will have to be "bubbled" up all the way to the head m times. This operation however, must be preceded by the addition of m elements in $O(1)$ time so the total amortized time remains $O(1)$. In addition, the next operation (after the worse case) will take constant time since all of the previous elements where removed. The worst case for *DEQUEUE* is constant time since we only pop an element from Q, compare it to the head of M and remove it if necessary. This takes constant time. Similarly, *FIND-MIN* will take constant time regardless of the input since we simply look at the head of M.

Analysis: We use the charge method to complete the analysis. When we enqueue an element we charge 1 unit to the present (for the insertion) and reserve 1 unit for the future for a total of 2 unit cost of insertion. When we dequeue an element, any popping that is necessary by the insertion can be charged to the past of the element being removed. This will only happen once, since an element can only be removed once. Find-Min takes constant time regardless of the previous inputs. Overall, the running time of a combination of m of these operations will be $O(m)$. With this analysis we conclude that *ENQUEUE*, *DEQUEUE* and

FIND-MIN all take O(1) amortized time.

Problem 4-2

a) Imagine an array of size m divided into 4 equal segments. Assuming a random pivot there is a probability of 1/2 that this pivot will lie on either of the middle segments. If we pick a pivot in these segments, the size of a sub-array can be at most $3m/4$ (this would occur when the pivot point is at the edge of either of these segments, creating sub-arrays of size $3m/4$ and $m/4$. The probability that x_i is a pivot is $1/m$ since each element has an equal chance of being chosen as the pivot. The probability of either of these events happening is at least 1/2 since the first event has a provability of 1/2

b) Using the claim in the prompt we know that if we flip a coin $3(\alpha+c)lgn$ times, then with a probability of $1-\frac{1}{n^\alpha}$ we will get at least $clgn$ heads. For this problem, we'll say that heads means that we picked a pivot in the middle two segments which creates a sub array of size no greater than $3m/4$. As proven in a, the probability of this is at least 1/2 so choosing heads is appropriate. Now,each heads essentially takes the subarray containing x_i and divides into two subarrays, one of which will have size of at most $3m/4$. We are interested in the number of times we would need to get heads in order for quick sort to reach it's last level of recursion, that is, when the subarray is of size 1 and only contains x_i. This would take $\log_{4/3} n$ "levels" or "heads" to accomplish. The total number that x_i will be compared with the pivot is $dlgn$ for some constant d to be derived next. Again, from the claim we know that we can get at least $clgn$ heads if we flip a coin $3(\alpha+c)lgn$ times, then with a probability of $1-\frac{1}{n^\alpha}$. Since we need at least $\log_{4/3} n$ heads we can set $\log_{4/3} n = clgn$. We now have $c = \log_{4/3} n/lgn$. Using change of base, we find that $\log_{4/3} n = \frac{logn}{log(\frac{3}{4}}$. Substituting this back into c we get:

$$c = \frac{1}{log(4/3)}$$

.

Since we are interested in a probability of at least $1-\frac{1}{n^2}$ we let $\alpha = 2$. Plugging this back into the equation in the claim we get a total number of $3(2+\frac{1}{log(4/3)})lgn = dlgn$. We conclude that

$$d = 3(2 + \frac{1}{log(4/3)})$$

c) If we take into account all $x_i \in A$, the probability that the number of comparisons will be at most $d'nlgn$ is upperbounded by the sum of the individual probabilities by the Union bound. This is equal to the opposite, 1 minus the probability that at least one event is not at most $d'nlogn$ comparisons. We have:

$$P(not) = 1 - (1 - 1/n^2) = 1/n^2$$

Adding all of these n individual event gives us (by the Union Bound) an upperbound of $n(1/n^2) = 1/n$. We are interested in the probability that all of the events are at most

$d'nlogn$ comparisons, we can get this using the above calculations and get $P = 1 - (1/n)$. Since we used this same answer in b) to calculate the probability we can conclude that

$$d = d' = 3(2 + \frac{1}{log(4/3)})$$

d) We can generalize our results from b and c to obtain a bound on the number of comparisons made by QUICKSORT that holds with probability $1 - \frac{1}{n^\beta}$. Notice that from b) to c) the lower bound of our probabilites changed from $1 - \frac{1}{n^2}$ to $1 - \frac{1}{n}$. Continuing this pattern if we want our probability to be $1 - \frac{1}{n^\beta}$ we need to begin with the individual probabilities being:

$$1 - \frac{1}{n^{\beta+1}}$$

. Plugging these into our original solution for be, we get our bounds equal to:

$$3((\beta + 1) + \frac{1}{log(4/3)})$$

Fernando Trujano March 20, 2015

6.046 Problem 5-1, 5-2

Collaborators: *Erika Lu*

Problem 5-1

a) The following is modeled after a simple Touring state machine. We start by comparing x to k. If $x < k$ we will be trying to move to the right. Otherwise, we will be moving left. Without loss of generality, this description will assume that we are moving to the right. The first state tries to go up if it can, otherwise it goes to the right until going to the right would be too far. At this point we switch over to the second state.

In this state we begin to look for k and follow the standard search algorithm except we start from our current node. We do this by trying to go to the right, if this is to far we go down and repeat. When we are at level 0, we go to the right until we find k. If we see a key greater than k at the bottom level, we can return that k does not exist.

The code is as follows:

```
FINGER-SEARCH(x, k):
    if (x.value == k) return x
    if (x.value < k) direction = 1  //Move right
    else direction = -1 //Move left

    state1 = true; //Start in state1
    state2 = false;
    currentNode = x;

    while (state1)
        if (currentNode.upParent)
            currentNode = currentNode.upParent //Go up if possible
        elif (currentNode.next.value > k)
            state1 = false
            state2 = true
            break
        elif( currentNode.value == k) return currentNode
        elif (currentNode.value == -INFINITY or currentNode.value == INFINITY)
            return None
        else currentNode = currentNode.next(direction)
    while (state2) //Go down
        if (currentNode.value == k) return currentNode
```

```
elif (currentNode.bottomLevel) currentNode = currentNode.next(direction)
elif ( currentNode.next(direction) < k)
    currentNode = currentNode.next(direction)
elif (currentNode.next(direction) > k) return None
elif (currentNode.value == -INFINITY or currentNode.value == INFINITY)
    return None
else currentNode = currentNode.downChildren
```

This algorithm is essentially a search backwards followed by a search forward so it's correctness follows from the correctness of the original search procedure for a skip list. The runtime can be improved however if m is close to k. This is because we only towards k and never have to touch any nodes to the left/right of m. This reduces the runtime to $O(lgm)$ where m is the difference in ranks of m and k. This runtime is guaranteed to run in $O(lgm)$ steps with high probability since (as previously mentioned) it is simple a backwards search followed by a forwards search. The original skip list runtimes and probabilities remain the same.

b) To support efficient rank search on skip list we augment each node with its number of left and right children as defined below: Left children: All nodes below the parent node and to the left of the node all the way until the next node to the left of the parent on the same level.

Right children: All nodes below the parent node and to the right of the node all the way until the next node to the right of the parent on the same level.

Slight modifications are needed to maintain the augmented data. The changes are as follows:

SEARCH(): Search remains the same as no data is mutated. The usual high-probability order-of-magnitude time bounds remain the same since no changes are made.

INSERT(x): Insert remains mainly the same with some additional book keeping on each step to maintain the augmented data. When we reach the bottom level we start counting the number of spots traveled, once we find a spot for x, this count will be the number of left children for x. We keep going to the right (and count) until we find y where y is the last node that does not have an "up pointer". This count will be the number of right children for x. We now follow the next points from x in both directions (after each of x's random promotions) and update it's neighbors children by subtracting x.leftChildren and x.rightChildren from each neighbor respectively. The algorithm remains relatively the same, with the extra step of counting the right children. This is however, still O(lgm) whp because it is essentially the same amount of work (asymptotically) as INSERT(y) where we have to walk $x.leftChildren_x.rightChildren$spots to the right before getting to Y. The high probability runtime follows from the original Insert procedure in Skip lists.

DELETE(x): The modification to delete is similar to that of Insert, but when we delete a node, we need to make sure we add all of it's children to their respective neighbors. This is done the same way we subtracted from the neighbors when inserting an element. Like Insert(x) the high probability runtime remains unchanged since the amount of extra work hides under a constant factor in the asymptotic notation of the runtime.

c) Rank search is very similar to range search, but we want to search for an element that is r elements away from x in faster than $O(r)$ in fact, we want this algorithm to run in $O(ln(r+1))$ with high probability.

```
RANK-SEARCH(x, r)
    state1 = true //Go to the right and up
    state2 = false //Start going down
    currentNode = x;
    remainingRank = r
    while (state1)
        numRightChildren = currentNode.next(1).children.length
        if (currentNode.upParent)
            currentNode = currentNode.upParent // move up if possible
        elif (currentNode.next(1).children().length > remainingRank )
            //Going to the right would be too far!
            state1 = false;
            state2 = true;
            break;
        elif ( numRightChildren < remainingRank)
            //update remaining rank and move the the right
            remainingRank = remainingRank - numRightChildren
            currentNode = currentNode.next(1)
    while (state2)
        if (remainingRank == 0) return currentNode
        else
            currentNode = currentNode.next(1) //Move right
            remainingRank = remianingRank - 1
```

Problem 5-2

a) The following greedy algorithm uses a min heap of pointers to the original set of n prizes. The elements in the min heap represent the elements with the highest prize value. We start by adding the first m elements of the set to a min heap, call this M. We then scan through the remaining $n-m$ elements of the list. We compare each of these elements i to the min of M. If $i > min(M)$ we pop the min from M and insert i into M. We do this because a larger element has been found, and M is supposed to contain the m prizes with largest

value. Otherwise, we just move to the next element and repeat. In the end, we will have a heap of size m (since we started with m elements and every time we add a new one we also remove the current min), these are the prizes with maximum value, and we should chose these prices.

To maintain the original order, we can maintain a heap of pointers to the original set and use those values for the heap operations. At the end, we can follow the points and "mark" each element in the original set. We can then go through the set (in order) and add all elements that are marked to a new "final" set. We then return this final set. These operations take O(n) and won't affect the asymptotic running time analyzed below.

Creating the min heap of size m will take $O(m)$ time. After that, we need to do $n - m$ min operations and O$(n - m)$ inserts for a total of $O((n - m)lgm)$ time. In total we get $O(m + (n - m)lgm) = O(nlgm)$. For space, the original set takes $O(n)$ while the heap takes $O(m)$ for a total of $O(m + n)$

It's worth noting that a simple DP algorithm could also be used, but it's running time would be $O(mn)$.

Another greedy solution involves a variation of the median-finding algorithm and runs in $O(n)$. We begin by modifying the media finding algorithm dicussed in lecture to find the element with $rank = m - 1$ call this x. Instead of $m/2$. We then scan through the original set and add any prize with a higher value than x. After the scan completes we will have selected the m prizes with the highest value from the set. This is guaranteed to maximize the total prize value. This algorithm runs in $O(n)$ as the median finding algorithm runs in $O(n)$ and the scan takes $O(n)$.

b) Since a simply greedy algorithm is not guaranteed to produce the most optimal result, the following algorithm uses Dynamic programming and will run in $O(mn)$ time. The sub problem $S_{j,q,A}$ means the max amount of value you can get if there are a total of j prices and you are allowed to pick at most q. Additionally, the last prize taken was of type A. Similarly $S_{j,q,B}$ means the max amount of value you can get if there are a total of j prices and you are allowed to pick at most q. Additionally, the last prize taken was of type B. The DP recurrence is as follows:

if $p_j.type == A$:

$$S_{j,q,A} = max(p_j.val + S_{j-1,q-1,A}, S_{j-1,q,A})$$

$$S_{j,q,B} = S_{j-1,q,B}$$

elif $p_j.type == B$:

$$S_{j,q,A} = S_{j-1,q,A}$$

$$S_{j,q,B} = mac(p_j.val + S_{j-1,q-1,A}, p_j.val + S_{j-1,q-1,B}, S_{j-1,q,B})p_j$$

For equation 1, we can either take p_j or not take it. For equation 2 we can't take p_j since we currently picked a prize of type B. Similarly for Equation 3, we cannot take the prize. For the last equation we can either take the B for the first time, take it (not for the first time), or don't take it.

The base cases are: $S_{j,j,(A|B)} = sum(pj)$ - add all prizes since we can pick all of them.

Our final solution will be $max(S_{n,m,A}, S_{n,m,B})$. The overall runtime is $O(nm)$ since there are $O(n)$ subproblems and each takes $O(m)$.

c) We use the same $S_{j,q}$ subproblems as in b) (ignoring type) but now we have to consider more sub sequences when making a selection. The recurrence is as follows. The base cases are: $S_{j,j} = sum(pj)$ - add all prizes since we can pick all of them. The recurrence becomes:

$$S_{j,q} = max(S - j - 1, q - 1, S - j - 2, q - 2, ..., S - 1, 1)$$

with a very important exception that we can only consider a case if the previous considered case is less than the current value under consideration. Our final solution will be $S_{n,m}$. The overall runtime of this dynamic programming solution is $O(mn^2)$ since each sub problem takes $O(n)$ to solve and there are $O(nm)$ subproblems.

d) This approach uses DP to find an optimal solution in $O(mn^2)$. We start by the node of the tree since (by the constraints of the problem) we have to take. We then consider the following cases:

Take m elements from the left, 0 elements from the right Take m-1 elements from the left, 1 elements from the right ... ... take 0 elements from the left, m elements from the right

We start at the bottom and memorize, working our way up the tree until we have our solution. The base cases are trivial, where the max value you can get at a leaf is equal to the value of the leaf. Define $S(m, i)$ as the max prize amount when we take m elements from the left and m-i elements from the right. The recurrence is as follows:

$$S(m, i) = max(S(m - i, i), S(m - i - 1, i - 1), ..., S(1, 1)$$

The final solution will be the the max of all S(m,i) for all i from 0 to m.

The overall runtime of this dynamic programming solution is $O(mn^2)$ since each sub problem takes $O(n)$ to solve and there are $O(nm)$ subproblems.

Fernando Trujano April 2, 2015

6.046 Problem 6-1, 6-2

Collaborators: *Erika Lu, Patrick Liscio*

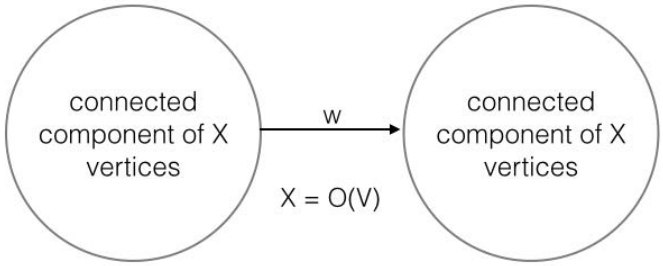
Problem 6-1

a) Case 1: $r = w_{i,j}$ Replacing an edge with another one of equal weight will not make any noticeable changes to D or Π. Therefore, no action is necessary. In this case the algorithm runs in $O(1)$ time.

Case 2: $r < w_{i,j}$ In this case we have to look into the distance matrix D and compare D(v to i) + r + D(j to v) to the D(u to v). If an update is required (when the distance is less) we update the distance matrix accordingly. Furthermore, we can update the predecessor matrix with the predecessor matrix of j to v. The running time of this algorithm is $O(V^2)$ since we look at all V^2 pairs.

Case 3: $r > w_{i,j}$ In this case, we will have to run Floyd Warshall again to update D and Π. This will take $O(V^3)$ time.

b) In the following example, changing the center edge from weight w to weight r where $r \neq w_{i,j}$ would force the algorithm to look at all pairs of points taking $\Omega(V^2)$ time. In the example below, the triple would be (i, j, r) where i is the vertex with an outgoing edge to the other set of connected components and j is the vertex this edge is adjacent to. r can be any non negative integer as long as $r \neq w_{i,j}$. G and D follow from the picture below:



c) The modification to Floyd-Warshall algorithm is as follows:

Subproblem $c(u, v, k, p)$ = weight of shortest path u to v whose intermediate vertices $\in 1, 2, ..., k$ in p number of hops.

Recursion:

$$c(u, v, k, 0) = 0$$

$$c(u, v, 0, 1) = w(u, v)$$

$$c(u, v, k, p) = min(c(u, v, k-1, p), c(u, k, k-1, 0)+c(k, v, k-1, 1), ...c(u, k, k-1, p)+c(k, v, k-1, p))$$

The rest of the algorithm follows from the Floyd Warshall algorithm described in lecture.

The final solution will be the min of all $c(u, v, k, h)$ Since the recursion now considers the all of the number of hops, the time to solve each subproblem increases by h. Since the number of sub problems also increases this algorithm will run in $O(V^4h)$

We can reduce the number of subproblmes to V^2h by implementing an algorithm very similar to the Dynamic program 1 described in Lecture 11 (APSP). Subproblems: $d(u, v, m) =$ weight of shortest path from u to v using ¡= m edges Recurrence:

$$d(u, v, m) = min(d(u, x, m-1) + w(x, v) for x in V)$$

$$d(u, v, 0) 0 if u = v; \infty else$$

Topological order: for m = 0,1,2,....h: for u and v in V: Final Solution: $d(u, v, h)$.

Number of subproblems = V^2h. Time per subproblem $O(V)$. Overall running time: $O(V^3h)$.

d) The following modification to matrix multiplication solves the bounded-hop APSP problem in $O(V^3 logh)$. In regular matrix multiplication we multiply elements and then add them together. We keep most of the process the same but instead of multiplying we add the entries and instead of adding we take the min of the entries. With this modified version of "multiplying" matrices we can make use of successive squaring to solve the bounded-hop ASPS problem. Using successive squaring, we can calculate logh powers of 2 for the weight matrix and multiply the corresponding matrices to get to h. For example if h is 14 we would multiply $W^2 * W^4 * W^8$. The resulting matrix will be the distance matrix D giving the shortest distances for at most h hop paths because of the way we are "multiplying" matrices.

The time required to "multiply" the matrices remains the same $O(V^3)$ since the changes do not change the asymptotic running time of the original algorithm. Since we use successive squaring of the weight matrix, we will only need to square the matrix at most $logh$ times. Therefore, this algorithm runs in $O(V^3 logh)$.

e) Case 1: $r = w_{i,j}$ Replacing an edge with another one of equal weight will not make any noticeable changes to D or Π. Therefore, no action is necessary. In this case the algorithm runs in $O(1)$ time.

Case 2: $r < w_{i,j}$ In this case we will have to consider and keep track of the least weight path of distance $< H$ and the least weight path of distance $< H - 1$ all the way to 1 for a total of H matrices. The rest is analogous to case 2 a). This operation takes $O(h)$ per matrix, since

there are $O(h)$ of these matrices and we need to check $O(V^2)$ pairs of points, this algorithm will run in $O(h^2V^2)$.

Case 3: $r > w_{i,j}$ Case three is analogous to part a) where we will have to rerun the algorithm to update D and Π. We can rerun the matrix multiplication algorithm explained in d) to update these in $O(V^3 log h)$ time.

Problem 6-2

a) Theorem: The undirected graph $G = (V, E)$ with a weight function providing nonnegative real valued weights, where all weights are different, has a unique MST.

Proof: (by contradiction). Consider G', an MST of G. Now, for the purposes of contradiction, assume that there is another MST for G, call it G'_2. Since an MST of G with no non-negative edges will have $v - 1$ edges G' and G'_2 must have the same number of edges.Since all the edges are different there is only one set of $v - 1$ edges in an spanning tree that will minimize the weight.

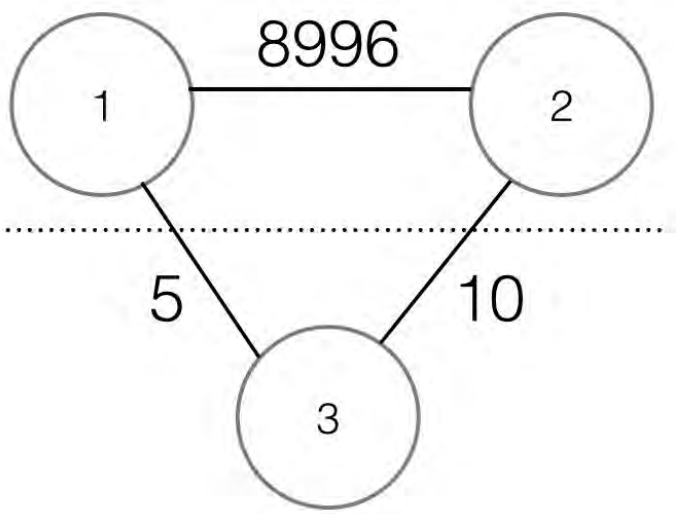
This can also be proved by the correctness of the algorithm described in b). Since b is a completely deterministic algorithm that makes no arbitrary decisions like picking the first element in a tie, it will always pick the unique MST of G.

b) The algorithm is correct. Theorem: For each node n the edge with the least weight coming out of n must be in the MST of G. Proof: (by contradiction). Assume, for the purposes of contradiction, that G' is an MST of G and $n \in V$ does not contain its original adjacent edge with the least weight. Now, we could replace any edge from n with its edge with least weight and still keep G' connected. With this change we have created an tree with a lower edge value sum than the original, contradicting the original claim that G' is an MST of G. We can generalize this theorem to connected components instead of nodes

For each iteration of the algorithm, we add at least a number of edges equal to half of the number of connected components. This happens on the worst case where we only connect each connected component to another. Since the number of connected components halves on each iteration we will need to do this $O(logV)$ times to get an MST. On each iteration, we need to check $O(E)$ edges. Therefore, the overall running time is $O(Elog(V))$.

c) The algorithm is incorrect. Counterexample: In the graph below, the MST has a total weight of $10 + 5 = 15$. With the algorithm described above, assume that the graph is partitioned arbitrarily into two sections as denoted by the dashed line. The algorithm will find the MST of both sections (for the top section this is simply vertices and and two directly connected to each other and just vertex 3 for the bottom part) and join the two with the least weight edge between them (in this case the edge between vertex 1 and vertex 3 worth 5).The total edge weight for the "MST" returned by this algorithm is over 9000. By the

initial analysis the tree returned by this algorithm is not the MST of the graph.



d)The algorithm is correct. Begin by performing DFS on the graph. Each time you visit a node n you add it two a list, call this L and a hash set, call this H. If n is in H it means we have seen this node before. We can backtrack using L until we find n again. All of the nodes we pass in this operation are part of a cycle. We take this cycle and remove the edge with the heaviest weight by looking at each edge in the cycle. We repeat this process of finding cycles and removing the most costly edge until no cycles are found. This happens when we transverse all V edges without finding one that we have already visited.

Finding the cycle takes $O(V)$ times since when visiting each node, it can either be an already visited node, or a new node. Since there are only V nodes, this can only happen $O(V)$ time. Similarly, looping though the edges to find the one with the heaviest weight will take $O(V)$ time since we can have at most V nodes in a cycle to loop though. We will do this no more than $O(E-(v-1))$ times since we can only remove enough edges to be left with v-1 for the solution to be a tree. Therefore, the overall running time of this algorithm is $O(VE-V^2+V)$

This algorithm works because we know that the largest edge weight in cycle cannot be in the final MST, by as many high weighted edges as we can, we guarantee to end up with an MST since the only edges that we can remove are those that make up a cycle.

Fernando Trujano April 9, 2015

6.046 Problem 7-1, 7-2, 7-3

Collaborators: *Erika Lu*

Problem 7-1

a) 1) By the max-flow min-cut theorem (from CLRS), we know that flow is maximum if the residual network contains no more augmenting paths. By increasing the capacity of an edge by 1, we are allowing for (potentially) one more augmenting path in the residual graph. We know that there may be at most 1 augmenting path (after the update) because of the following: 1) The original residual network had no augmenting paths (by max-flow min-cut theorem). 2) If an augmenting path is found, it must go through the updated edge (since there where no augmenting paths before). After the path is "used" it will be updated according to the Ford Fulkerson algorithm (effectively "removing" that augmenting path). The correctness of this operation follows form the correctness of Ford Fulkerson. Since the capacity can only increased by one, no more augmenting paths will exist. Since there can only be one augmenting path, the max flow can only increase by one.

2) Increasing the capacity of an edge by k is essentially the same as increasing the capacity of an edge by 1 k times. Therefore, this proof follows from the proof in 1). If we increase an edge by one k times, where each time we can increase the max flow by at most 1, then in total, we can increase the max flow by at most k. In other words, after updating the edge on the residual network, there will be at most k more augmenting paths.

3) In the worst case, decreasing the capacity of an edge by 1 will decrease the max flow by 1. This is because, after the update, there will be at most 1 less augmenting path in the residual network. This is a path that made use of (u, v) before, but using it now would exceed the capacity of the edge. Assuming the edge was already at it's full capacity, reducing it's capacity by 1 would force one of those augmenting paths to not exist. The max flow cannot decrease by more than 1 since the capacity of the edge only changed by one, and can therefore only affect 1 augmenting path. Similarly, the max flow cannot go up since there is no way to create new augmenting paths by decreasing the capacity of an edge. Therefore, the max flow can decrease by at most 1.

4) Decreasing the capacity of an edge by k is essentially the same as decreasing the capacity of an edge by 1 k times. Therefore, this proof follows from the proof in 3). If we decrease an edge by one k times, where each time we can decrease the max flow by at most 1, then in total, we can decrease the max flow by at most k.

b) In order to update the max flow without rerunning all of Ford Fulkerson, we will need to use the current flow and residual network and correctly update the residual edge affected by

the change. From here, we can simply run another iteration of Ford Fulkerson by checking to see if an augmenting path exists in the residual graph. If a path exists we augment the flow along that path and continue looking for paths.

Correctness: If we increase the capacity of one of the current edges by k , then the max flow can only increase a max of k times. Once we update the graph, this is essentially an "unfinished" Ford Fulkerson algorithm with at most k iterations to go. Therefore, the correctness follows from the correctness of Ford Fulkerson as described in detain in CLRS.

Runtime: For this algorithm, on the worst case, we will increase the flow by k (as proven in the first part of this problem). This means we will have to find at most k augmenting paths on the residual graph. Assuming we run BFS to do this, this will take $O(E' + V)$ time per iteration where E' is the number of edges in the residual graph. Notice that the number of verticies remains the same. Since each edge in the original graph can have a reversed edge $|E'| \leq |E|$. Therefore the total time it takes to find a single augmenting path is $O(2E + V)$. Since we might have to find up to k path, the total running time of this algorithm is $O(k(2E + V))$. Notice that the initial step of updating the residual edge accordingly takes constant time.

c) In order to update the max flow given that a single edge (u, v) has reduced its capacity we will need to do the following: We know that the max flow can only decrease (or stay the same) as proven in part a). First, we want to check if the reduction of the edge will even affect the max flow. If the current flow through that edge is still less or equal than the new (reduced) capacity, then the max flow will remain unchanged. If the capacity of the updated edge is less than the current flow, we would be violating max-capacity property and need to correct is as follows:

First, reduce the flow on (u, v) by k. Since the capacity of that edge also reduced by k and we started with a valid flow network, we will now have a valid flow network. At this point, with the capacity updated, and the flow through that edge updated, we will have to add back as much flow as we can. We can do this by running the same algorithm as described in b).

Correctness: First, if $f(u, v) <= r$ then no additional computation is needed since we still have a valid flow network, which we were told was the max flow of the graph. By our results from a) reducing the capacity of an edge by k can only decrease the total flow by k. With this in mind, we can (for a moment) assume that this will occur and subtract k form the flow to that edge. So, $f(u, v) = f(u, v) - k$. At this point, we can add up to $k - 1$ flow back into an edge (implying that the capacity change only reduced the max flow by 1). This is analogous to the algorithm described in b), where the flow can only go up. We find augmenting paths and update the flow accordingly. The correctness of this procedure follows from the correctness of b).

Runtime Updating the edge capacity and checking if the flow is affected takes constant time. Reducing the edge capacity also takes constant $O(1)$ time. In the worst case, we will remove k flow and add $k-1$ flow back. Notice that we cannot add more than $k-1$ flow because we did that initial check at the beginning (and the total flow can only go down (from the original) since we reduced the capacity of an edge). This means that we might have to run $k-1$ iterations of the algorithm described in b),yielding the same asymptotic running time of $O(k(2E+V))$.

Problem 7-2 We can model this problem as a max flow problem by setting the capacity of all edges to 1. $c(u,v) = 1$ for all u, v. Since an edge represents a road, a capacity of 1 represents that that road can only be taken once. Additionally, we connect all of the companies sources to a super sink S with an edge of capacity of 1, this ensures that we only find (at most) one path for each company.

With the graph built, we can run the Ford Fulkerson method to find the max flow of the graph. If the max flow is less than k $|f| < k$ there is no way to route k trucks so we return impossible. Otherwise we can returned the k paths as explained below: At this point we know that we can send k trucks through the graph given the constraints. Therefore, we can go through each of the source vertices and arbitrarily follow edges until we get to the sink. We return the path taken as the path for that source, remove the edges from the graph and continue to the next source.

On each iteration of Ford Fulkerson, when we find an augmenting path in the residual network, we will have increased the flux by exactly one. This is because all of the edge weights have a capacity of at most 1.

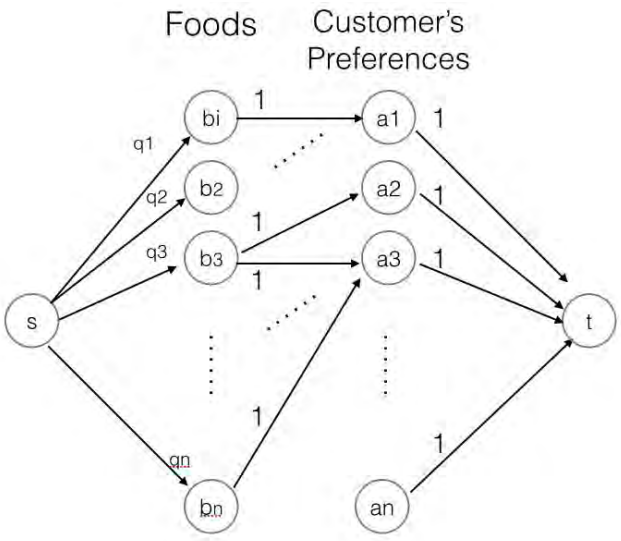
Correctness For this problem, we modified the given graph with capacities and a supersink and then model is as a max flow problem solved by Ford Fulkerson. The "flow" in this problem are the trucks, which take edges from their company source to the common destination/ the sink. An edge in the graph represents a road and by setting the capacity of this edge to 1, we represent a road that can only be taken once. Furthermore, we connect all company sources to a supersource by an edge with a capacity of 1, ensuring that each company can only send at most 1 truck to the sink. The correctness of the max-flow problem follows from the correctness of Ford Fulkerson as explained in CLRS. After running Ford-Fulkerson, we will know whether we can "send" k trucks through the graph or not. If the max flow is less than k, then there is no way to send all k trucks through the graph. Otherwise since it edge can only be used once, we can pick any path for each company, remove the used edges, and still have available edges for the rest of the company.

Runtime Building the graph will take $O(k+E)$ time since we have to set the capacity of all edges to 1 ($O(E)$)and link all of the k companies to the super source ($O(k)$). Once the graph is built we need to run the Ford Fulkerson method. This will take $O(E|f|)$ time. The max possible flow on any graph constructed this way is k because each of the k sources can

only be used once (since they are connected by an edge of capacity 1 to the super source). Therefore, running Ford Fulkerson will take $O(Ek)$. Note that finding all of the paths will take $O(E)$ since we will go through each edge exactly once (because we remove an edge after using it). Overall, the algorithm will run in $O(k + E) + O(Ek) = O(Ek)$

Problem 7-3 This problem can be modeled as a max flow problem is we construct a flow graph G as follows:

We begin with a source s that connects to all of the vertices on the first column. This column represents the different available foods $b_1, b_2, ...b_n$. Also, q_i $c(s, b_i) = q_i$ this means that an edge from s to bi will have a capacity equal to the quantity of that food. The third column of the graph represents customers and an edge from bi to ai implies that customer a_i can eat food b_i. All of the elements on the third column connect to the drain node t. We assign each one of these edges a capacity of 1 to make sure that each customer only gets 1 food item assigned to him/her. Below is a graphical representation of this description:



After building the graph, we can use the Ford Fulkerson method to find the max flow. From here, we can scan through the third column and look at the flow of incoming and outgoing edges to a_i. If customer a_i has an incoming flow of 1, this means that he/she got a food item. To find this item, we have to see where the flow is coming from and follow the edge to appropriate vertex on the second column, If customer a_i has an incoming flow of 0 he/she did not and should get a voucher. We continue to do this for all customers. At the end we will know what food each customer got, or if they got a voucher. Since this is a max flow problem, we tried to maximize the number of food that was distributed to customers which minimized the number of vouchers to be given out.

Runtime: This algorithms is essentially two parts. The first part consists on building the graph and the second consists on running the Ford Fulkerson method on that graph. Since the overall running time will be driven by the latter, I will analyze that first.

Ford Fulkerson runs in $O(E|f|)$. In the worst case we will have every customer getting something so the max flow $|f| = O(n)$. In the worst case, we will have a very dense graph where every customer sets every food item in their preferences, this forces the number of edges $E = O(nm)$. Combining these two, we get that the Ford Fulkerson method runs in $O(mn^2)$.

Building the graph is upperbounded by mn, which occurs when every customer sets every food item in their preferences. Combining both parts of the algorithm we get a runtime of $O(mn^2) + O(mn) = O(mn^2)$

Correctness: Our model of the problem as a max flow problem is correct because of the way the graph was constructed. In this problem we can think of the flow as the food, which travels through a graph of customers till the destination (sink). We connect all foods to the source, since initially all foods are available with a limited quantity equal to the capacity of the edge that connects to it from s. This ensures that we don't try to give out more food than we have. These foods connect to customers that stated that they can eat that food with an edge of capacity one. This ensures that each customer only receives one of each item. To ensure that each customer gets at most 1 item, we set the edges from the customer vertices to the drain to have a capacity of 1. The correctness of the max flow problem follows from the correctness of Ford Fulkerson. Once we are done with Ford Fulkerson, for each customer vertex, there are two cases: Either the customer received an item which means there is an incoming edge with flow of 1 into that vertex (and subsequently an edge with flow one leaving the vertex by the flow conservation property) or the customer did not receive any food and should get a voucher. Again, since this is a max flow problem, we tried to maximize the number of food that was distributed to customers which minimized the number of vouchers to be given out.

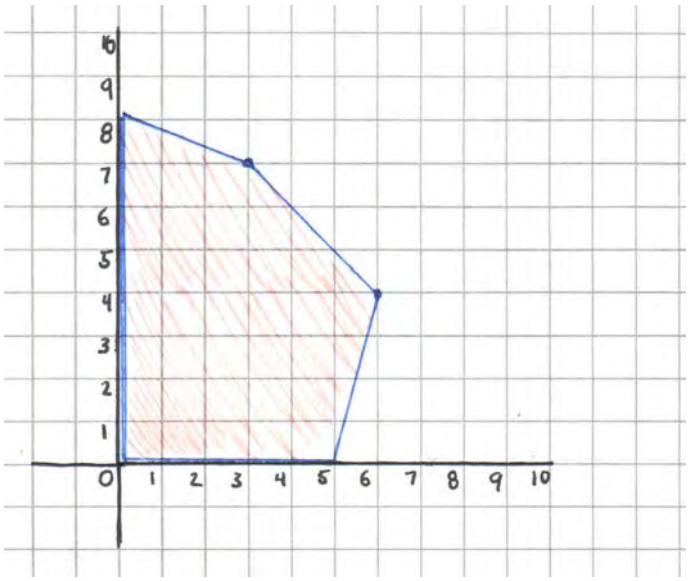
Fernando Trujano April 22, 2015

6.046 Problem 8-1, 8-2

Collaborators: *Erika Lu*

Problem 8-1

a) In the following diagram of the feasible region, the x axis corresponds to x_1 while the y axis corresponds to x_2



b)

Standard form:

$$x_1 + x_2 \leq 10$$
$$4x_1 - x_2 \leq 20$$
$$x_1 + 3x_2 \leq 24$$
$$x_1, x_2 \geq 0$$

Slack Form:

$$10 - x_1 - x_2 = x_3$$
$$20 - 4x_1 + x_2 = x_4$$
$$24 - x_1 - 3x_2 = x_5$$
$$Z = 4x_1 + x_2$$

c) First Pivot: $x_1 = 5$ and is then swapped with x_4

$$x_3 = 5 - \frac{5x_2}{4} + \frac{x_4}{4}$$
$$Z = 20 + 2x_2 - x_4$$
$$x_1 = \frac{20 + x_2 - x_4}{4}$$
$$x_5 = 19 - 13x_2/4 + x_4/4$$

2nd Pivot: $x_2 = 4$ and is then swapped with x_3

$$x_2 = 1/5(20 + x_4 - 4x_3)$$
$$Z = 28 - 3x_4/5 - 4x_3/5$$
$$x_1 = 6 - x_4/5 - x_3/5$$
$$x_5 = 6 - 2x_4/5 + 13x_3/5$$

Since the coefficients are all negative at this point in the modified LP, we can conclude that the max solution will be 29. We can then plug this in for the original objective function and solve for x_1 and x_2 given that x_3 and x_4 are zero. Therefore $x_1 = 6$ and $x_2 = 4$.

In our diagram from a), on the first iteration, we got our objective function to intersect at point (5,0). On the second iteration we got it at (6,4). At this point, we found a maximum and we can calculate our solution.

d) The dual LP of the standard form LP from part b) can be prooved by obtaining a certificate of optimality. By the LP Duality theorem, we show that this solution is optimal.

$$\overrightarrow{b} \cdot \overrightarrow{y} = \begin{bmatrix} 10y_1 & 20y_2 & 24y_3 \end{bmatrix}$$

$$A^T \overrightarrow{y} \geq \overrightarrow{c}$$

$$\begin{bmatrix} 1 & 4 & 1 \\ 1 & -1 & 3 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} \quad \geq \quad \begin{bmatrix} 4 \\ 1 \end{bmatrix}$$

Problem 8-2

a) Step 1: Show TRIPLE-SAT is NP

Given a solution to Φ, I can plug in the variables into the original equation and verify that they simplify to True. This can be done in polynomial time, for each of the 3 (different) proposed solutions.

Step 2: Reduce from known NP-Complete and show both ways

If we have a black box that can solve any SAT problem, we can modify our original TRIPLE-SAT problem in poly time, and use SAT to solve the problem. We modify the original problem p as follows. We take our original problem p and OR it with the result of XORing two new variables $AandB$. The modified problem looks like this:

$$p||(A \oplus B)$$

By constructing our problem this way, we guarantee that three solutions exists, (when A is true and B is false, A and B are false, A is false and B is true). We can then use SAT on this modified problem to generate three solutions to the original problem. Therefore,

To prove from TRIPLE-SAT to SAT, we just need to pick the solution where A and B are both false from TRIPLE-SAT and use that in SAT. We can do this in poly time. Since TRIPLE-SAT will return true if SAT returns true and viceversa,and TRIPLE-SAT is in NP, we can conclude that TRIPLE-SAT is NP Complete.

b) For this problem, we will reduce DONUT from 3SAT in order to prove that DONUT is NP Hard. To begin the reduction we will need to devise a way to map a 3SAT problem into it's corresponding map for a DONUT problem. We do this as follows. For each clause, we create individual nodes and link them to each other. We also link each one of these nodes to a new node (call these variables) corresponding to their complement. Finally, we connect each complement to a new node corresponding to the original value (or the complement of the complement). By connecting node A to its complement, we assure that they both can't be true (since you can't select two neighbors). Saying there is a donut shop at node A equates to saying A is true on the 3SAT problem.

We can build this graph from a 3SAT problem in polynomial time, we can then calculate k: $k = num(clauses) + num(variables)$.

Since DONUT can be reduced from 3SAT, and 3SAT is NP complete. A polynomial time

algorithm for DONUT would automatically imply that 3SAT is solvable via a polynomial time algorithm (with a reduction). This would then show that $P = NP$.

c) We can turn this problem into a decision problem by asking weather it is possible to produce a schedule that yields a profit of k. With this decision problem, we could solve the original problem by simply binary searching for the max profit. Note that the min profit is 0 and the max profit is the sum of all p_i: $k = \sum_{i=0}^{n} p_i$.

Step 1: Show that SCHEDULE is in NP Given a schedule, we could simply loop through the events, check that they are valid (not conflicting with each other) and then add up all the profits. From there we could determine if the the profits are greater than or equal to k.

Step 2: Reduce from known NP-Complete and show both ways

For this problem, we will reduce partition to a specific case of scheduling to show that any problem in NP can be solved if scheduling can be solved. To understand the logic behind the reduction, I will first explain some properties of PARTITION. In PARTITION, we want to take a set of integers as an input, and output two sets with equal sums. We can easily prove that each of these sums must be equal to $S/2$ where S is the sum of all elements in the original set. If we are "picking" different numbers to add to a set, we need to stop adding numbers once our sum reaches $S/2$.

With this in mind, we begin our reduction by setting the profit of a task equal to the corresponding integer value on the input to partition. So an input of (1,2,4) to partition would imply that there is a task with 1 profit, another with 2 and another with 4. Now we want k to be equal to $S/2$, since we are interested in knowing if we can get at least k profit with the scheduled tasks. We also say that deadline = $S/2$ since we want to stop picking tasks once we reach this point (similarly to how PARTITION stops picking integers once the first subset reaches S/2. Lastly, we set the duration of each task equal to the profit.

With this, we are able to reduce partition to a specific case of scheduling to show that scheduling is NP Hard.

Since SCHEDULE is in NP and is NP hard, SCHEDULING is NP Complete.

Fernando Trujano April 30, 2015

6.046 Problem 9-1, 9-2

Collaborators: *Erika Lu, Ben Gunby,Patrick Liscio,18.434 Notes*

Problem 9-1

a) To prove that $Alg1$ does not guarantee any constant approximation ratio, we first need to show that picking the most dense object may not be the best decision. This could be the case when picking the most dense object, prevents you from picking two less dense objects with a larger total value.

More generally, we want the density of a higher valued object to be less than the density of a lower valued object and have the sizes of both objects big enough so we can only pick one of them. The algorithm will pick the denser lower valued object instead of the optimal less dense one. Consider two objects a_1 and a_2. To meet the scenario described above, we want the following to hold:

$$\frac{v_1}{v_2} = \frac{1}{k}$$
$$\frac{1}{k} > \frac{s_1}{s_2}$$

To satisfy these, we can set the values to the following

$$v_1 = \frac{B}{k+1} = s_1$$
$$v_2 = \frac{Bk}{k+1}$$
$$s_2 = B$$

b) The first part of this algorithm is trivial. If we can fit all of the items, we take them all. This is the optimal solution. Otherwise, we sort the items by density and find the smallest index i that makes the total size of the first i items exceed the max capacity B. At this point, we consider taking either the first $i-1$ items, or item i.

We can establish an upperbound on the optimal solution, call this $V*$,. By taking the first i items would be taking i of the most dense items, and would also exceed our capacity, this is clearly an upper bound. In math terms:

$$\Sigma_{j=1}^{i} v_j \geq V*$$

Since the algorithm picks the max value of taking either the first $i-1$ items, or item i, and together these two (added) form an upperbound to the optimal solution, each of these must be at worst half of the optimal value. Therefore, Alg_2 always yields a 2-approximation to the optimal solution.

c) The following Dynamic Programming solution solves the knapsack problem. Define the subproblem to be $S_{i,v}$ which equals the min subset $a_1....a_i$ whose total value is exactly v. The base cases are:

$$S_{0,v} = \text{inf}$$

$$S_{i,0} = []$$

For the first case, no solution exists, so we return infinity. For the second case, the solution is empty since no objects have 0 or negative value. The recurrence is as follows:

$$S_{i,v} = minSize(S_{i-1,v-v_i} \cap [a_i], [S_{i-1,v}])$$

The left side (inside the min), represents choosing item i and the left side represent not choosing item i. We will need to calculate all $S_{i,v}$ and then chose the optimal form there. For the runtime analysis, not that we can obtain an upperbound on the optimal value by looking at V, the largest value of the all items in the set. An upperbound is then nV where n is the size of the set of the items in consideration. With this in mind, the total runtime will be $O(n^2V)$ since we will have to go through at most n different values for each $S_{i,v}$ that we calculate.

d) Notice that Alg_3 is not fully polynomial because because the values of the items may not be polynomially bounded in n. Alg_4 achieves FPTAS by scaling the values of each item to be polynimally bounded by n. It then runs Alg_3 and returns the solution S'.

To show that Alg_4 is FPTAS, we will begin by showing $V(S') \geq (1-\epsilon)OPT$ for the returned solution S' where OPT is the optimal solution. First, we must note that any object i will have $Kv'(i) \leq v(i)$ because we scaled down the value of each item by $K = \frac{\epsilon v_1}{n}$. This means that the max value for an optimal solution $S*$ can decrease by at most K for each item (since we scaled down the value of each by K). Since we have n items, the total value can decrease by at most nK. We then run Alg_3 with the scaled values and are guaranteed to get an optimal set for the scaled values and we know that this must be at least as good as picking $S*$ with the original values. At this point, we know that the value of the returned set S' is at least as good as $KV'(S*)$. As shown above this is the same as $V(S*) - nK = OPT - \epsilon V = (1-\epsilon) * OPT$. The running time of the algorithm follows from the runnign time of Alg_3 but V is now polynomial in n. More specifically, this algorithm runs in $O(n^2V) = O(n^2 floor(\frac{n}{\epsilon}))$ Since Alg_4 is also within a factor of $(1-\epsilon)$ from the OPT Alg_4 is FPTAS.

Problem 9-2

a) A minimal cycle cover A of T will include exactly one edge u, v from each cycle on T. This is because a cycle cover includes at least one edge from each cycle and a minimal would include at most one. Knowing that we only have one edge from each cycle in A, reversing all edges in A is equivalent to reversing one edge in every cycle of T, effectively breaking every cycle.

To finish the proof, we need to consider the case where reversing an edge would introduce a new cycle. This case however, cannot happen. The proof is by contradiction. Assume that there is an edge that when reversed, created a new cycle in the tournament graph T. Then, that edge must have been the only edge in A on some directed cycle T, this means there was another cycle not accounted in the cycle cover which is a contradiction.

b) This problem makes use of some important observations about Triangles and cycle covers. First, two triangles cannot share two edges, they can only share one. This can be proved by contradiction: If two distinct triangles share two edges, they share three vertices and are thus not distinct triangles. Next, we note that if any edge is shared among k or more triangles, it must be in the cycle cover. The last observation is that given an edge in a cycle we can construct a triangle containing that edge.

With these observations in mind, we can begin. The first step is to reverse all edges shared among more than k triangles. These are edges that would be in the set (per observation 2). For each edge that we find, we decrement k by one and continue until we can't find another edge or k = 0. On the latter case, we just check if the remaining graph is acyclic and then we are done. Otherwise, we will end up with a graph where every edge is in at most k triangles. It is important to note that at this point, we have not removed anything from the graph, we've only reversed some edges.

The next step can be thought of as a reversal of part a). In part a) we showed that if we have a k cover and reverse those k edges we end up with an acyclic graph. Now, we will be considering an acyclic graph and try to find which edges to reverse to get to it. Recall that each edge has is shared with at most k triangles. We now consider how many vertieces there are in these k trianges. For each triangle, we may have 2 vertices + k from the other k triangles. Since we have k of these we end up with $k(k+2) = k^2 + 2k$.

c) The first step in this algorithm is to perform the reduction/kernel described in part b). This step will take $O(n^3)$ since we will need to count the number of triangles and all other operations are $O(n^3)$ At this point, we can perform any brute force algorithm (check every combination) to solve the problem, and return the appropriate decision. the runtime of this would be $O(k^{f(k)})$ which is still a FTP algorithm since $k^{f(k)} = 2^{f(k)log(k)}$

6.046 Cheat Sheet

Fernando Trujano

Dynamic Programming

1) Define subproblems
2) Relate subproblem solutions w/ recurrence + base case
3) Recurse and memoize or bottom up DP table
4) Solve original problem
- Analize runtime

Longest Increasing Subsequence

Input: $A_1 \dots A_n$

Goal: Find longest strictly increasing sequence (not necessarily contiguous)

$L(j)$ = LIS ending at position j

$L(j) = \max_{i<j,\ A[i]<A[j]} \{ L(i) \} + 1$

Solution $= \max_j \{ L(j) \}$

Runtime: $O(n^2)$

Coin Change

Input: n denominations of coins $1 = v_1 < v_2 < \dots < v_n$

Goal: make change for amt of money C use as few coins.

$M(j)$: min # coins to make change for j

$M(j) = \min \{ M(j - v_i) \} + 1$

Runtime: $O(nC)$

Building Bridges

Goal: Build as many (non-crossing) bridges as possible (like longest increasing subsequence)

$X(i)$: index of corresponding city on northern bank.

Compute $X(i)$'s in $O(n\log n)$ time. Find longest subsequence of $x(1) \dots x(n)$ $O(n^2)$

Edit Distance

Input: Strings $A[1..n]$ $B[1..m]$

Costs: C_i, C_d, C_R (insertion, deletion, replacement)

Goal: Min cost of transforming $A \to B$

$T(i,j)$ = min cost to transform $A[1..i]$ into $B[1..j]$

$$T(i,j) = \min \begin{cases} C_d + T(i-1,j) & \text{(delete)} \\ T(i,j-1) + C_i & \text{(insert)} \\ T(i-1,j-1) & \text{if } A[i] = B[j] \\ T(i-1,j-1) + C_R & \text{if } A[i] \neq B[j] \end{cases}$$

$O(nm)$ # subproblems

Solution: $T(n,m)$

Integer Knapsack Problem

Input: n types of items each with size s_i (int) and value v_i, knapsack capacity C

Goal: Pack items in knapsack maximize value (repeated items ok)

$M(j)$ = max val pack in knapsack c=j

For j=1 to C # bottom up approach compute $M(j)$

$M(j) = \max \{ M(j-1),\ \max_{items} \{ M(j - s_i) + v_i \} \}$ (j-th slot empty or some item in j slot)

Fill knapsack of size $j - s_i$ optimally

Runtime: C subproblems, n per subproblem $O(nC)$

Solution: Store a back pointer to $M(\cdot)$ that was max. $M(C)$ will be result. Following back pointers will give items used.

"in NP" - solutions can be confirmed in poly time

"NP hard" at least as hard as everything in NP.

"NP complete" - NP hard and NP

Balanced Partition ("fairly dividing spoils")

Input: n integers $\{A_i\}$

Goal: Partition integers into S_1 and S_2. Minimize $|Sum(S_1) - Sum(S_2)|$

$$P(i,j) = \begin{cases} 1 & \text{if some subset of } \{A_1 \dots A_i\} \text{ has sum } j \\ 0 & \text{otherwise} \end{cases}$$

$i \le n$, $j \le nk$

$P(i,j) = 1$ if $P(i-1,j) = 1$ or $P(i-1, j-A_i) = 1$

$= \max\{ P(i-1,j), P(i-1, j-A_i) \}$

$S = (\sum A_i)/2$

Find min val of $S - i$ such that $P(n,i) = 1$ $\Rightarrow \min_{i \le S} \{ S - i : P(n,i) = 1 \}$

$sum(S_1) = i$, $sum(S_2) = 2S - i$

Solution: $2 \min_{i \le S} \{ S - i : P(n,i) = 1 \}$ since $|sum(S_1) - sum(S_2)| = 2S - 2i$

Box Stacking

Input: n boxes each with depth d_i, height h_i, width w_i (all orientations of original set), $w_i \le d_i$

Goal: Construct stack of max height. Can only stack box i on j if $w_i < w_j$ and $d_i < d_j$

$H(j)$ = tallest stack of boxes with box j on top.

$H(j) = \max_{i<j,\ w_i > w_j,\ d_i > d_j} \{ H(i) \} + h_j$

Solution: $\max_j \{ H(j) \}$ ~ which box will end on top.

Runtime $= \Theta(n^2)$

Weighted Interval Scheduling

Reasources and requests $1 \dots n$, start time $s(i)$, end time $f(i)$

Requests compatible if no overlap

Goal: select compatible subset of max weight. $w(i)$: weight of request

Subproblems: $R^x = \{ \text{request } j \in R \mid s(j) \ge x \}$

$opt(R) = \max_{1 \le i \le n} (w_i + opt(R^{f(i)}))$

Runtime: $O(n^2)$, can be $O(n\lg n)$ if presort first

Fast Forier Transform (FFT)

Representations

	coeffs	root	samples
eval	$O(n)$	$O(n)$	$O(n^2)$
addition	$O(n)$	∞	$O(n)$
mult	$O(n^2)$	$O(n)$	$O(n)$

Use FFT to convert between coeffs and samples in $O(n\log n)$

- Degree n poly needs n+1 samples to represent
- Can evaluate $A(x_0)$ in $O(n)$ w/ Horner's Rule

$A(x) = a_0 + x(a_1 + x(a_2 + \dots x(a_{n-1})\dots)))$

Divide and Conquer FFT: Divide into even/odd coeffs

$A(x) = A_{even}(x^2) + A_{odd}(x^2)$

Evaluate at $(2n+1)$th roots of unity $e^{\frac{2\pi i k}{2n+1}}$ for $0 \le k \le 2n+1$

Recurrence: $T(n) = 2T(n/2) + O(n) = O(n\log n)$ (since squaring the jth roots of unity yields the j/2th roots of unity)

Multiplication: Before ⇒ coeffs multiply → solution $O(n^2)$

Now: coeffs $\xrightarrow[\Theta(n\lg n)]{\text{evaluation}}$ point-value $\xrightarrow[\Theta(n)]{\text{multiply}}$ $\xrightarrow[\Theta(n\log n)]{\text{interpolation}}$ solution $\Theta(n\log n)$

Applications: Convolution - inner product of all relative shifts. Convolution of A and reverse(B)

Master Theorem

Solve recurrence in form $T(n) = a\,T(n/b) + f(n)$ (a = # sub prob, n/b = size/subprob, f(n) = divide + combine)

① Calculate $n^{\log_b a}$ and compare w/ $f(n)$

Case 1: $x > f(n) \Rightarrow T(n) = \Theta(n^{\log_b a})$

Case 2: $x = f(n) \Rightarrow T(n) = \Theta(n^{\log_b a} \log^{k+1} n) = \Theta(f(n)\log n)$

Case 3: $x < f(n) \Rightarrow T(n) = \Theta(f(n))$

Change of Variables:

$T(n) = 2T(\sqrt{n}) + \Theta(\log n)$

1) Let $n = 2^m$ and substitute

$T(2^m) = 2T(\sqrt{2^m}) + \Theta(\log 2^m) = 2T(2^{m/2}) + \Theta(m)$

2) Define $S(m) = T(2^m) = T(n)$

$S(m) = T(2^m) = 2T(2^{m/2}) + \Theta(m) = 2S(m/2) + \Theta(m)$

3) Solve with Master Theorem: $S(m) = \Theta(m\log m)$

4) Substitue back: $T(n) = \Theta(\log n \log\log n)$

Union Find

MAKE-SET(), FIND-SET(), UNION

Linked List Solution:

MAKE-SET(x): Initialize x as lone node $O(1)$

FIND-SET(x): Return repr of x $O(1)$

UNION(x, y): Join x and y. Could be $1+2+\dots n = \Theta(n^2)$ if we add larger to smaller. (need to update pointers everytime)

Solution: Merge smaller into larger. $O(n\log n)$ amortized → Each element can be charged at most lg n times since S at least doubles

Forest of Trees Solution: root = repr

MAKE-SET(x): Initialize x as lone node $O(1)$

FIND-SET(x): climb tree to root $\Theta(\text{height})$

UNION(x,y): climb to both roots, set pointer of smaller to larger $\Theta(\lg n)$

Path compression: when find-set completes, update rep pointers of all visited nodes to root of tree (rep). m operations ⇒ $O(m\lg n)$

⇒ Overall can get almost constant amortized time / operation

Amortized Analysis

amortized cost per operation

Aggregate Method: $\frac{\text{total cost k operations}}{k}$

Accounting Method: operations can store credit

- Can pay for time using stored credit.

Example: 2-3 trees insert ⇒ add coin worth $O(\lg n)$; on delete ⇒ remove coin worth lg n. 0 cost (can always pay since we can't delete something that wasn't inserted)

Charging Method: operations can charge to the past

amortized cost = actual cost - charge to past + charge by future

Example: Table doubling charges $O(n)$ to n/2 inserts since last doubling ↳ Inserts only charged once!

Potential Method: Define potential function Φ mapping structure configuration → non-negative integer.

- Potential high costs in the future 0011 → 0100

Example: Binary counter ⇒ we want ΔΦ to not change that much after each operation

Φ = # 1 bits. An increment costs (# trailing 1's + 1). Increment destroys t1s, creates one. Amortized cost $= 1 + t + 1 - t = 2$

Example: B-trees: Φ = # nodes w/ B children

Random Algorithms

generates random # and makes decisions based on value.

Monte Carlo - Always polynomial, expected correct with high probability (whp)

Las Vegas - Expected polynomial, always correct

Quicksort - ① Pick a pivot element x ② Partition ③ Recursively sort subarrays L, G. ④ Combine. $\Theta(n)$ for divide

- Can take $O(n^2)$ time if bad pivots are chosen.
- If pivots chosen at random ⇒ $O(n\lg n)$ whp expected

Skip list - Maintain dynamic set of n elements in $O(\lg n)$ expected /operation

2 sorted lists → $2\sqrt{n}$. lg n sorted list $\lg n \sqrt[\lg n]{n} = 2\lg n$

L_1, L_0 be too far.

Search(x): Walk right on top list L_n until going right would go too far. Walk down a level and repeat until element found (or not)

Insert(x): Search(x) and insert into bottom list L_0. Flip coin → Heads → promote x to next level and repeat. Tails → stop

levels in n element skip list is $O(\lg n)$ whp. Failure Prob: $Pr\{ > c \lg n \text{ levels} \} = Pr\{\text{some elmt got promoted} > c\lg n\} \le n \cdot Pr(\text{elmt x got promoted} > c\lg n) = n(\frac{1}{2})^{c\lg n} = \frac{n}{n^c} = \frac{1}{n^{c-1}} = \frac{1}{n^\alpha}$, $\alpha = c - 1$ (by union bound)

Invariant: If station i is in L_x it must also be in $L_{x-1} \dots L_0$

Search(x) - $O(\lg n)$ whp

Chernoff Bounds: $Pr[Y \ge E[Y] + r] \le e^{-2r^2/m}$

Expected = average over random choices

Frievald's algorithm $r[1 \dots n]$

① Choose random binary vector. If $A(Br) = Cr$ return YES else return NO

If $AB = C$: $A(Br) = (AB)r = Cr$ so no false negatives ⇒ Verify Matrix multiplication

If $AB \neq C \Rightarrow Prob[ABr \neq Cr] \ge 1/2$

Claim: with probability $1 - \frac{1}{n^\alpha}$, $3(\alpha + c)\lg n$ tosses produce at least $c\lg n$ heads

Chebyshev's: $Pr[|R - E(R)| \ge x] \le Var(R)/x^2$

Markov: $Pr(R \ge x) \le \frac{E(R)}{x}$ $(x > 0)$

$Var(R) = E[R^2] - E^2[R]$

Van Emde Boas Tree

Recursive tree data structure. Achieves time $O(\lg\lg u)$ insert, delete, successor, predecessor. explanation: memory

Split u into $\sqrt{u}$ clusters of size $\sqrt{u}$. Summary vector: tells if the cluster is empty or not. This achieves $O(\sqrt{u})$ time. Using recursion we turn this into $T(u) = T(\sqrt{u}) + O(1) = O(\lg\lg u)$

summary vector √u, 1/4 bits, min, max, next-lex leaves

Hashing

U = # possible keys, m = slots in table, n = # keys in table

Hashing with chaining (for collisions) $\Theta(1 + \alpha)$ time, α = load factor = n/m

Simple Uniform Hashing: $P(h(k_1) = h(k_2)) = 1/m$. Assumes input keys are random. $k_1 \neq k_2$ collision, hash function

Universal Hashing: choose random h from H. H = universal hash family. $Pr_h\{h(k) = h(k')\} \le 1/m$

$E[\#\text{keys colliding in a slot}] \le 1 + \alpha$

Mult mod prime H: $h_{a,b}(k) = [(ak + b) \bmod p] \bmod m$, pick prime p, $a, b < p$

Dot product: $k = \langle k_0, k_1, \dots k_{r-1} \rangle$, $a = \langle a_0, a_1, \dots a_{r-1} \rangle$, $h_a(k) = a \cdot k \bmod m$

Perfect Hashing: $O(1)$ search, $O(n)$ space. Takes $O(n\lg^2 n)$ to build table whp

Trees

Binary, 2-3, B

Binary: Search, Insert, Delete $O(h)$. Keep balanced to get $h = O(\log n)$ time.

2-3 Trees Properties/Invariant

- Nodes can have 1 or 2 keys k
- Nodes must have k+1 children
- All leaves at the same level
- Data kept in sorted order
- Height ≤ n lg n

A < D < B

Example: 30; 10 17, 38; 3 7, 14, 20 24, 32, 41 48

Operations: $O(\lg n)$ time

Search(x): Like BST ⇒ walk down tree

Insert(x): ① Search for x and insert. ② If overflow promote median → repeat

[2|3|7] ⇒ 3; 2, 7. If root has 3 elements, make new root. overflow! max 2 keys. ★ only way that height can change! and delete

Delete(x):

Fact: Successor of any element in internal node will be a leaf. If x not leaf: swap x w/ successor. Delete node, redistribute ↳ may need to and merge if invariants are broken

B-Trees

B = branching factor/min degree

Properties/Invariants

Each node: At least B-1 keys & At most 2B-1 keys

Children must = k+1

All leaves at the same depth

Helpful in reading blocks of data. Used in file system

Runtimes

Merge Sort (worst): $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n)$

Binary Search (worst): $T(n) = T(n/2) + \Theta(1) = \Theta(\lg n)$

Randomized QuickSort (Expected)
$T(n) = \frac{2}{n}\sum_{k=0}^{n-1} T(k) + \Theta(n) = \Theta(n \log n)$

Randomized QuickSort (worst): $T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$

Strassen's Algorithm (worst) $T(n) = 7T(n/2) + \Theta(n^2) = \Theta(n^{\lg 7})$

Selection (worst) $T(n) = T(\lceil n/5 \rceil) + T(\frac{7n}{10} + 6) + \Theta(n) = \Theta(n)$

Randomized Selection (worst): $T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$

Randomized Selection (Expected): $T(n) = \frac{2}{n}\sum_{k=\lfloor n/2 \rfloor}^{n} T(k) + \Theta(n) = \Theta(n)$

Median Finding: $O(n)$

Weighted Interval scheduling: $O(n \lg n)$

Polynomial Evaluation (Horner's): $O(n)$

Polynomial Multiplication (FFT): $O(n \lg n)$

BST Search / Insert / Delete: $O(h)$, $h = O(\lg n)$ balanced

Van Emde Boas: $O(\lg \lg U)$

Union Find: MAKE-SET: $O(1)$, FIND-REP: $O(\lg n)$, UNION: $O(\lg n)$ Amortized

Skip List: $O(\lg n)$ with high probability

SSSP: BFS $O(V+E)$
Dijkstra $O(V \lg V + E)$
Bellman-Ford $O(VE)$
Topo + 7 for DAG $O(V+E)$

APSP: Floyd Warshal $O(V^3)$
Johnson's $O(V^2 \lg V + VE)$

Flow: Ford-Fulkerson: $O(E \cdot F)$
Edmond Karp: $O(VE^2)$

Leader Election-Bidirectional ring of n processes: $O(\lg n)$

Luby(M|S) $O(|V| \log |V|)$

Hash Family $\mathcal{H}$ is universal if for each pair of distinct keys $x, y \in U$, the # of hash functions $h \in \mathcal{H}$ for which $h(x) = h(y)$ is exactly $|\mathcal{H}|/m$

It is possible to compute the smallest $\sqrt{n}$ elements of an n-element array in sorted order in $O(n)$ time: SELECT the $\sqrt{n}$ elements and partition around it, then sort those $\sqrt{n}$ elements in $O(n)$ time.

- $O(\lg \lg(n^n)) = O(\lg[n \lg(n)]) = O(\lg(n) + \lg \lg(n)) = O(\lg(n))$
- If the deterministic median finding algorithm had two recursive subproblems of sizes $\frac{1}{5}n$ and $\frac{31}{40}n$ (instead of $\frac{1}{3}n$ and $\frac{2}{3}n$) then algo still runs in $O(n)$.
  Total subproblem size $= \frac{n}{5} + \frac{31n}{40} = \frac{39n}{40} < n$ on each level
  Sum of geometric series $\Rightarrow$ linear
- Perfect hashing can use more than one hash function, one for first level and one for every bucket of the 2nd
- If Φ is a potential function for S, 2Φ is also a function for S and the amortized running time of each operation $w.r.t.\ 2\Phi$ is at most twice $w.r.t.\ \Phi$
- Skip lists can perform all operations in $O(\lg n)$ as long as prob of promoting p, $0 < p < 1$
- Amortized cost = actual cost + Φ(after) − Φ(before)

Dynamic Programming

Longest Palindromic Subsequence

Given a string, Find longest palindrome that is a subsequence. character → caract

Strategy: L(i,j) length of longest palindromic subsequence

```
def L(i,j):
    if i==j: return 1
    if x[i] == x[j]:
        if i+1==j: return 2
        else: return 2 + L(i+1, j-1)
    else: return max(L(i+1,j), L(i,j-1))
```

subproblems = $O(n^2)$ time/subproblem : $O(1)$

Coin Change - make change for N cents

Using coins of $S = \{s_1, s_2 \dots s_m\}$ denomination.

C[p]: # coins needed for p cents (minimize coins)

C[0] = 0 C[p] = min C[p−s_i] + 1

C[N] Final answer : $s_i \le p$

Runtime: O(mN) Pseudo-Polynomial

Linear Programming

Politics example: How to campaign to win election. Votes obtained per dollar:

Policy	Demographic 1	2	3	4
x_1	1	-2	5	3
x_2	2	8	2	-5
x_3	3	0	0	10
x_4	4	10	0	2
majority		50,000	100,000	25,000

Want to win majority in each demographic by spending minimum amount of money

Let x_1, x_2, x_3, x_4 denote dollars spent/issue.

Minimize: $x_1 + x_2 + x_3 + x_4$

Subject to: $-2x_1 + 8x_2 + 0x_3 + 10x_4 \ge 50{,}000$

$5x_1 + 2x_2 + 0x_3 + 0x_4 \ge 100{,}000$

$3x_1 + -5x_2 + 10x_3 + 2x_4 \ge 25{,}000$

$x_1, x_2, x_3, x_4 \ge 0$

Certificate of optimality - shows that LP solution is indeed optimal

LP Duality:

$$\max \vec{c}\cdot\vec{x} \text{ s.t } A\vec{x} \le \vec{b},\ \vec{x} \ge 0 \quad = \quad \min \vec{b}\cdot\vec{y} \text{ s.t } A^T\vec{y} \ge \vec{c},\ \vec{y} \ge 0$$

primal LP ↑ dual LP ↑

Converting to standard form:

1) Want to minimize $-2x_1 + 3x_2$ ⇒ Negate coefficients and maximize.

2) IF x_j does not have a non negativity constraint x_j replaced by $x_j' - x_j''$ $x_j', x_j'' \ge 0$

3) Equality constraint: $x_1 + x_2 = 7$ translates to $x_1 + x_2 \le 7$, $x_1 + x_2 \ge 7$

4) $\ge$ constraint translates to $\le$ by multiplication of -1

Worst-case running time for simplex is exponential, but it is usually efficient. IF constraint graph of linear system is acyclic, solution exists and can be found in $O(V+E)$ time using DAG shortest paths.

All-pairs shortest Paths

Recall: Single source shortest paths

situation	algorithm	time
unweighted ($w=1$) / nonneg edge w	BFS / Dijkstra	$O(V+E)$ / $O(E+V\lg V)$
general	Bellman-Ford	$O(VE)$
DAG	topo sort + ↑	$O(V+E)$

Floyd-Warshall Algorithm: $O(V^3)$ ⇒DP

1) Subproblem $C_{uv}^{(k)}$ = weight of shortest path $u \to v$ whose intermediate vertices $\in \{1,2,\dots k\}$

2) Guessing: Does shortest path use vertex k

3) $C_{uv}^{(k)} = \min\{C_{uv}^{(k-1)}, C_{uk}^{(k-1)} + C_{kv}^{(k-1)}\}$ ($C_{uv} = w(u,v)$)

4) Fork: for u,v ⇒ $C_{uv}^{(n)} < 0$ * Neg weight cycle

5) $\delta(u,v) = C_{uv}^{(n)}$

$O(V^3$ subproblems) $O(1)$ per subp

Johnson's Algorithm. Good for sparse G

① Find function $h: V \to R$ such that: $w_h(u,v) = w(u,v) + h(u) - h(v) \ge 0$ $\forall u,v \in V$ or determine that a neg weight cycle.

② Run Dijkstra's on (V, E, w_h) from every source vertex $s \in V$ ⇒ get $\delta_h(u,v)$ for all $u,v \in V$

③ Claim $\delta(u,v) = \delta_h(u,v) - h(u) + h(v)$

Apsp of $d_{uv}^{(m)}$ distance to go from u to v in at most m edges

$d_{uv}^{(m)} = \min\left(d_{ux}^{(m-1)} + w(x,v)\right) \Rightarrow O(V^4)$

$d_{uv}^{(m)} = \min\left(d_{ux}^{m/2} + d_{xv}^{m/2}\right) \Rightarrow O(V^3 \log V)$

Greedy Algorithms and Minimum Spanning Trees

Greedy Algorithm: repeatedly make locally best choice, ignoring effect on future

Properties:

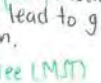
Optimal Substructure: optimal solution to problem incorporate optimal solution(s) to subproblem(s)

Greedy choice property: locally optimal choices lead to globally optimal solution.

Minimum Spanning Tree (MST)

Spanning tree $T \subseteq E$ of min weight.

Greedy choice property: cut-property - lightest edge that connects S to T will be in an MST.



Prim's Algorithm:

start with random vertex → visit

consider all edges out of visited that don't go to already visited vertex

pick smallest one + visit new node

$\Theta(V)$ · Extract-Min + $\Theta(E)$ · T decrease-key = $O(V \lg V + E)$ using Fib heaps?

Kruskal's Algorithm

Keep adding lowest-weight edges that don't create cycles

Pseudo-code using Union-Set()

```
T = ∅ // Build MST
For v∈V. Make-set(v)
sort E by weight
For e=(u,v) ∈ E  // in order
    if Find-set(u) ≠ Find-set(v).
        add e to T
        Union(u,v)
```

Correctness. when adding edge, we make use of Greedy choice property

In any MST. if we remove an edge → two remaining trees are MSTs and removed edge is least weight edge between them. Note that the converse is not true!

- If you increase all edge weights in G by a constant, G and G' will have the same MST (same edges)
- Add edge in MST ⇒ cycle
- If all edges in G have unique weights ⇒ MST will be unique
- Simple Cycle: edge w/ lowest weight in some MST ↔ edge w/ highest weight is not in an MST
- Paths along MST are NOT shortest paths
- if e in MST. There exist $x, y \in V$ where e is in a shortest path from $x \to y$

Flow Networks

Flow Network: Directed graph with source s and sink t. Each edge (u,v) has capacity $c(u,v)$

Flow ↔ Rate

Capacity Constraint: For all $u,v \in V$: $0 \le F(u,v) \le c(u,v)$

↳ Flow from one vertex to another nonneg and not > cap

Flow Conservation: For all $u \in V - \{s,t\}$ $\sum_{v \in V} F(u,v) = \sum_{v \in V} F(v,u)$

↳ Flow into vertex must come out

$|F| = \sum_{v \in V} F(s,v) = F(s,V)$ $F(X \cup Y, Z) = F(X,Z) + F(Y,Z)$ if $X \cap Y = \emptyset$

Properties: $F(X,X) = 0$ $F(X,Y) = -F(Y,X)$

$|F| = F(V,t)$

Cut: (S,T) is a partition of V such that $s \in S$ and $t \in T$

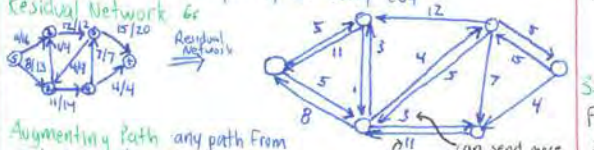
Flow across the cut = $F(S,T)$

$|F| = F(S,T)$ proof $F(S,T) = F(S,V) - F(S,S) = F(S,V) = F(S,V) + F(S-s,V) = F(s,V) = |F|$

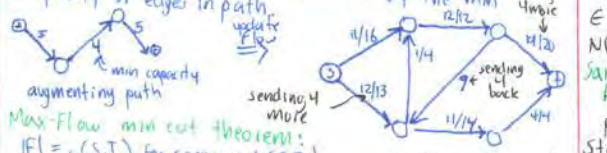
capacity of cut S,T = how much S can give to T = $c(S,T)$

$|F| = F(S,T) = c(S,T)$: value of any flow is bounded by the capacity of any cut

Residual Network G_F



Augmenting Path any path from s to t in G_F. Flow can be increased by the min capacity of edges in path



Max-Flow min cut theorem:

$|F| = c(S,T)$ for some cut (S,T)

F is a max Flow ⇒

F admits no augmenting paths

Ford Fulkerson Algorithm:

```
f[u,v] ← 0 for all u,v ∈ V
while an augmenting path p in G_f exists.
    augment f by c_f(p)
```

augmentations

Edmond's Karp: $O(E)$ BFS $O(VE)$ ⇒ $O(VE^2)$ time selects shortest augmenting path by running BFS

Flow across an edge might go up or down with each iteration

Flow Network Problem Tips

- Turn problem into Flow network

Preferences - represented with edges w/ capacity 1

super source's connected to each possibility

Vertex-capacity - limit vertex capacity to x by splitting vertex into two connected by edge of capacity x

Bipartite Vertex Cover: Given bipartite graph. Find S such that S covers G and $|S|$ is minimal.

Define Flow Network H: Create source vertex with edges of capacity 1 to every v in L. edge of

Create sink connecting all R to it with capacity 1

- Direct all edges in E from L to R with ∞ capacity
- Run max flow and return value

NP Hard / Complete Problems

P - set of problems solvable in poly time

NP - decision problems with poly-time verification / decision problems solvable in non-deterministic poly-time

NP-Hard: Problem D is NP-Hard if $A \le_P D$ $\forall A \in NP$ A reduces to D

NP-Complete: Problem D is NP-complete if $D \in NP$ and D is NP-hard

Reduction: Problem A poly time reduces to B aka $A \le_P B$ or $A \le_P B$ if ∃ a poly time alg R s.t $A(x) = B(R(x))$

Prove X is NP-Complete

① $X \in NP$ via nondeterministic algo or certificate + verifier

② Reduce from known NP complete problem Y to X

A) Poly time conversion From Y inputs to X inputs.

B) if $Y_{YES} \to X_{YES}$

C) if $X_{YES} \to Y_{YES}$

3SAT - Given boolean formula (ANDs of ORs) of the Form: $(x_i \vee x_3 \vee \bar{x}_6) \wedge (\bar{x}_2 \vee x_3 \vee x_7)$ (clause)

Is there a variable $\in \{T,F\}$ such that formula = T

$\in NP$ - verifiable in poly time

NP-hard: convert algo into circuit → formula

3CNF

Sample Reduction: Ham Cycle → Ham Path

Proof that Ham Path is NP-Complete

given NP-complete path that visits every vertex exactly once

Step 1) Show HAM-PATH is in NP

Want to prove that $\exists V(x,y)$ that runs in poly time. Graph → path

a) Does y touch each vertex once $O(V)$

b) Is every edge in p in G $O(V)$

⇒ HAM-PATH is in NP

Step 2) Show HAM-PATH is NP hard

Reduction. Given an instance of HAM Cycle G, want to create input to HAM PATH.

- Pick arbitrary vertex $v \in G$ and split into v' and v''
- All edges into v are now into v'
- All edges out of v are now out of v''

IF G has a ham cycle → G has HAM PATH use edges in original cycle but path starts at v'' and ends v'

IF G has ham path → G' must have HAM Ham-Path in G' must start at v'' and end at v'

⇒ HAM-PATH is NP-Complete

Ham Path in $G \Rightarrow$ Ham cycle in G': Ham path in G: $v \to u_1 \to u_2 \to \dots \to u_{n-1} \to v$

Consider $v, u_1, u_2, \dots u_{n-1}, v$ rearrange $v \to u_1 \to u_2 \to \dots \to u_{n-1} \to v$: Ham cycle in G'

1) New vertex v to G

2) Add edges to/from all vertices from/to $v \in$ Ham path in G

$G = (V,E) \Rightarrow$ create G'

Ham Path = path that visits every vertex exactly once

Ham Cycle = path that visits every vertex exactly once and ends at start vertex

Approximation Algorithms

Algorithm for a problem of size n has an approximation ratio of ρ(n) if it produces a solution s.t the cost C of the solution is at most ρ(n) factor away from optimal

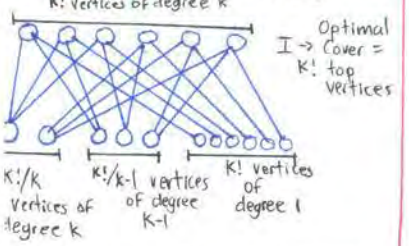
TAS - Polynomial time approx scheme
Takes in ε and problem instance $O(n^{2/\varepsilon})$
and produces $(1+\varepsilon)$ approx of optimal
FPTAS - Polynomial in n and $1/\varepsilon$ $O(n^2/\varepsilon)$

Vertex Cover: Repeat: 2-approx Algo for vertex cover
take any $(u,v) \in E$ Add (u,v) to set
remove all edges from u and v

Approx Vertex Cover Natural

$C \leftarrow \emptyset$ $E' \leftarrow E$
while $E' \neq \emptyset$:
pick V with max degree
$C = C \cup \{v\}$
remove V and all incident edges from E'
return C

A bad example: ex k=3
K! vertices of degree k



But algorithm may end up picking all the bottom vertices:
$SOL = K!\left(\frac{1}{K} + \frac{1}{K-1} + \dots 1\right) \approx K! \log k$
$\Rightarrow \log k$ worse than optimal

To prove fully PTAS show that:
$OPT \leq (1+\varepsilon)\cdot R$ or returned value
$OPT(1-\varepsilon) \leq R$

(n)-approximation algorithm: cost C produced by alg is within a factor of ρ(n) of the cost C^* of optimal solution
$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n)$

4 Reducing Clique to Independent Set

Clique: Given graph $G = (V, E)$ and integer k, is there a set of vertices $C \subseteq V$ with $|C| = k$ that form a complete graph?

Independent Set: Given graph $G = (V, E)$ and integer k, is there a set of vertices $I \subseteq V$ with $|I| = k$ such that for any $u, v \in I$, $(u, v) \notin E$?

Given that Clique is NP-Complete, we prove that Independent Set is NP-complete.

1. **Show Independent Set ∈ NP**

   To prove this, we need to prove there exists a verifier $\mathcal{V}(x, y)$. Let $x = (G, k)$ be a "yes" input. Let y be I that satisfies the condition.

   It takes $O(|I|)$ to check that $|I| = k$. It takes $O(|I|^2)$ to check that for every $u, v \in V'$, $(u, v) \notin E$. This checks in polynomial time that the certificate y proves that x is a valid input. Therefore, Independent Set is in NP.

2. **Show Independent Set ∈ NP-hard**

   We prove this by giving a Karp-reduction of Clique to Independent Set.

   (a) Given an input $x = (G, k)$ to Clique, create input G' which has the same vertices, but has edge (u, v) if and only if $(u, v) \notin E$. This takes $O(|E|)$ time, so this reduction takes polynomial time.

   (b) If I is a set of vertices that form a k-Independent Set for G', then $C = I$ is a k-Clique for G because for $u, v \in I$, Independent Set says that $(u, v) \notin E'$, but this implies that $(u, v) \in E$ for the Clique problem due to the method of construction. This shows that there are edges between every pair of nodes in C. In addition $|C| = k$, and so C is a k-clique.

   (c) If C is a set of vertices that form a k-Clique in G, then $I = C$ is a k-Independent set for G'. This is because $u, v \in C$ implies that $(u, v) \in E$ for Clique, and this implies that $(u, v) \notin E'$ for Independent Set. Since $|I| = |C| = k$, this shows that there are k elements in the construction G' that are not adjacent to each other.

3. This proves Clique reduces to Independent Set in polynomial time, which means that Independent Set is at least as hard as Clique, so k-Independent Set is NP-hard.

Problem 4. Red-Blue Network Flow [30 points] (2 parts)

You are given a flow network $G = (V, E)$ with distinguished source vertex s and distinguished sink vertex t. As usual, the edges in E are directed edges. The edges in E are now, however, of two types: blue and red. The blue edges are of the usual type: each blue edge e has a capacity $c(e)$ that is the *maximum possible* amount of flow that can pass through that edge. The red edges are built out of strange alien technology: each red edge e must have at least a certain amount $d(e)$ of flow passing through that edge. That is, $d(e)$ is the *minimum required* amount of flow that must pass through that edge.

Other that the fact that some edges are now red, every thing is as usual (e.g. flow must be conserved at all intermediate vertices, etc.). Each edge is either red or blue. We call such a network problem a "red-blue network flow problem". As usual, the goal is to find a flow of maximum possible value.

(a) Show that it is possible to compute a maximum red-blue network flow, or determine that no legal flow exists, in polynomial time.

(b) You now discover how to "turn off" red edges. If a red edge e is on, at least $d(e)$ units of flow must pass over it; if it is off, no flow can pass over it. You can turn some red edges on and others off. Show that it is now NP-complete to determine the maximum flow possible in a red-blue network.

Solution: (a) The problem can be formulated as a linear programming problem. Let $f(u, v)$ be the flow on edge $(u, v) \in E$, B and C be the sets of blue and red edges respectively. The LP is

$$\max \sum_v f(s, v) \ \ s.t.$$

$$f(u, v) \leq c(u, v) \ \ for \ any \ \ (u, v) \in B \ , f(u, v) \geq d(u, v) \ \ for \ any \ \ (u, v) \in C$$

$$f(u, v) \geq 0 \ , \sum_v f(u, v) - \sum_w f(w, u) \ \ for \ any \ \ u \in \{V - s, t\} \qquad (1)$$

The LP can be solved in weakly polynomial time by ellipsoid or interior point algorithm.

(b). Since the NP-complete is for decision problems, the question can be interpreted as determine if there is a legal k flow in the flow network. To prove the problem is NP-hard, we can reduce the SUBSET-SUM problem to the red-blue flow network problem. The SUBSET-SUM is the decision problem that determines if there is a subset in $S = \{x_1, x_2, ..., x_n\}$ whose sum equal to k.
We first create a red-blow flow network with source s, sink t and another n vertices. The edges in the flow network are as follows,
(s, u_i) for $i = 1, 2, ..., n$ are red edges that connects s to n vertices that represent n variables in the subset. $c(s, u_i) = x_i$
(u_i, t) for $i = 1, 2, ..., n$ are blue edges that connects u to sink t. $d(u_i, t) = x_i$.
The way we create the graph guarantees that the flow from s to t through u_i can either be x_i if the red edge (s, u_i) is on or 0 if the red edge is off. The flow in the red-blue network is therefore equal to the sum of the x_i whose corresponding red edge is on. Determine if there exists a flow k is the same as determine if there is a subset sum k. Creating the red-blue network takes polynomial running time. Since SUBSET-SUM is NP-complete, the red-blue flow network problem is NP-hard. Given the assignment of flow on each edge and the on/off assignment of red edges, it takes polynomial time to check if flow k exists, so the problem is in NP. Combining with the NP-hardness proof, the red-blue network-flow problem is NP-complete.

DP Vertex Cover Problem. A vertex cover of an undirected graph is a subset of its vertices such that for every edge (u,v) in the graph either v or u or both are in vertex cover.
Find minimal vertex cover for tree. binary.
1) d[v] → size of min vertex cover for tree rooted at v
2) Guessing: is vertex v in min vertex cover or not
3) $d[v] = \min\begin{cases} 1 + \sum_{\text{vev.children}} d[v] \\ I(v \text{ has left child})\left(1 + \sum_{\text{vev.left}} d[v]\right) + I(v \text{ has left child})\left(1 + \sum d[v]\right) \end{cases}$
4) Solution: d[tree.root] # subproblems O(n) Time/sp: O(1) ⇒ O(n)

• If A is NP-Hard and can be reduced to B. Then B is also NP Hard.
• Approximation Factor not necessarily carried over in reductions

Distributed Algorithms

- Algs that run on networked/multiple processors and share memory.
- Distributed Networks: based on undirected G
  associate process w/ Vertex
  Edge = two directed communication channel

Synchronous - rounds

- Communicate using messages
- Processes need not be distinguishable

Each Round:
- Processes determine messages to send to all ports based on state
- Messages sent to other end
- Processes compute new state based on message

Leader Election: Goal: one process elects itself leader

Breaking Symetry: No deterministic algorithm with indistinguishable processes work
- This is because all nodes start the same, send the same message, and make same change.

Solved by UID (Unique ID) For each process or
Randomness: Each process randomly picks UID from large set.
large set: $\{1, \dots r\}$ $r = n^2/2\varepsilon$ n processes $0 < \varepsilon < 1$
↳ w $Pr[1-\varepsilon]$ all IDs different. Proof: union bound

Algorithm:
- Choose random ID
- Exchange ID's, if max unique → elect, else repeat

Expected Time: $\leq 1/1-\varepsilon$

w $P \geq 1-\varepsilon$ algo finishes in one round

Maximal Independent Set (MIS)

"Every node either in S or has neighbor in S"
no two neighbors both in set
can't add any more nodes

Each process should output "IN" or "OUT"

Luby's MIS Algorithm

For all "active" nodes.
Round 1: Chose random val r from $1 \dots n^5$ send to all neighbors
- Recieve vals From all neighbors
  If r > all recieved join MIS. output "IN"

Round 2: If you joined, announce to all neighbors
If you recieve announcement, don't join => output "OUT"
If you decide (either way) become inactive

Repeat these rounds until all nodes decided

Correctness: Independence: when a node joins all neighbors become inactive
Max: only becomes inactive if it joined or can't join
Termination: show that graph gets smaller on each phase.

Breath First Spanning Trees

processes have no knowledge about G
- Distinguished vertex v_0 = root
- Processes know their UID and neighbor's UID
- Processes must produce BFS Tree rooted at v_0
- Vertex at distance d from v_0 appears at depth d.
- Processes should output parent.

"marked": boolean →
parent: UID or undefined →
send: boolean →

Simple BFS Algorithm:
Processes mark themselves as they became part of tree. Initially i_0 marked.

Round 1:
- If $i = i_0$: Process i sends "search" message to neighbors

If process i recieves message. Marks itself, outputs parent (i_0), plans to send on the next round.

Round $r > 1$
- If plan to send, send "search"
- If not marked and recieve, mark, select one sending neighbor to set as parent, plan to send.

Correctness:
All branches created synchronously growing one hop at a time.

Two corrections = ↑ complexity
Message: $O(n^n|E|)$
Time: $O(n^n \cdot n \cdot d)$
- messages might "pile" up in a channel
- $O(n!)$ paths => $O(n^n)$

Invariants:
At the end of r rounds, processes at distance $\leq r$ from v_0 are marked.
Process have parent defined if marked
Time Complexity: # rounds till done.
= Max distance bw nodes $\leq$ diam
Message Complexity: $O(|E|)$
- Can add child pointers by responding "parent" or "non parent" to "search"

Termination: upperbound on diam or
Convergecast: send "done" msg from leaves.
Process sends done when: knows gets responses to search => children gets "done" from children
When i_0 recieves done from all children: BFS tree is complete!

Counting # of nodes:
Set up a spanning tree and sum up the nodes at the end during convergecast.

Synchronous: Shortest Path's Trees

- Like BFS with weights.
- UID's.
- Processes only know neighbors and incident edge weights (weight from v_0 to j)

Output: parent(j), distance(d)

Bellman-Ford Shortest Paths Algorithm:
States: dist: shortest known distance from v_0 initially 0 For i_0, ∞ for others.
parent: UID or undefined
UID

Algorithm: each round:
- send distance (dist) message to all neighbors
- Recieve messages
- Relaxation step → $dist = \min_j(dj; +w(i,j))$
- If dist decreased set parent = j

Time Complexity: n-1 rounds
Message Complexity: $O(n \cdot |E|)$
Terminate w/ convergecast

Asynchronous Distributed Algorithms

- no rounds. "steps" executed one by one in any order. FIFO queue for messages.

Breath First Spanning Trees.

Synchronous BFS Algo won't work
Strategy: Each process keeps track of hop distance, changes parent on new shorter path and propagates improved distances
State: parent, dist, send(v) → FIFO of N
Transitions: recieve(m)$_{i,v}$
- if m+1 < dist: dist = m+1; parent = v
  for every w: add m+1 to send(w)

send(m)$_{v,u}$: Precondition: m = head(send(v))
remove head(send(v))

Message complexity: $O(n|E|)$
Time: $O(diam \cdot n \cdot d)$ ← Ignore local processing time
Correctness: Invariant: ← traverse each link
For any node v, and any r $0 \leq r \leq diam$, there is an at most r hop path from v_0 to v
=> by time r·n·d node v's dist $\geq r$

Shortest Path Tree

Bellman Ford: correct for small paths, asynchronous
Transitions: recieve(m)$_{v,u}$ parent = v
- if m+w(u,v) < dist: dist = m+w(u,v)
  for every w: add dist to send(w)

send(m)
remove head of send(v)

Cryptography Hashing

OW - one way
- Infeasable, given $y \in_R \{0,1\}^d$ to find any x s.t $h(x) = y$

CR - collision resistance (strong)
- Infeasable to Find x, x' s.t $x \neq x'$ and $h(x) = h(x')$

TCR - Target CR
- Infeasable, given x to Find $x' = x$ s.t $h(x) = h(x')$

PRF - Pseudorandomness
- Behavior indistinguishable from Random Oracle

NM - Non-Malleability
- Infeasable, given h(x) to produce h(x') where x and x' are "related"

Random Oracle - ideal, but not achievable in practice.
- Gives something random every time, except as required for consistency with previous answers. (deterministic)

Facts: CR => h is TCR
OW ⇏ CR, TCR (neither holds)
Collisions can be found in $O(2^{d/2})$
Inversion can be found in $O(2^d)$

Fixed Parameter Algorithms

- Aim for exact solution but isolate exponential term to a parameter
  => small parameter = fast solution

k-Vertex Cover: $O(2^k \cdot v)$ time

Kernelization - simplifying self reduction: Poly time alg converting input into small equivalent input
Ex: remove vertecies you know will be there and add them at the end