

## Final Exam Review

CMSC 451, Summer 2009

### Final Exam Topics

**Greedy Algorithms** (Interval Scheduling, Shortest Path, Minimum Spanning Tree, Union-Find, Clustering, and Huffman Coding)  
**Divide and Conquer** (Merge Sort, Inversion Counting, Closest Pair of Points, Multiplication, and Matrix Multiplication)  
**Dynamic Programming** (Weighted Interval Scheduling, Knapsack Problem, and Sequence Alignment)  
**Network Flow** (Max Flow/ Min Cut, Capacity Scaling, Bipartite Matching, Extensions, and Airline Scheduling)

**Intractability** (Poly-time reductions, NP, NP-Complete)  
**Approximation Algorithms** (Load Balancing, k-Center Problem, Pricing Method, Integer and Linear Programming)  
**Randomized Algorithms** (Global Minimum Cut and Closest Pair of Points)  
**Registrar's Problem**

2

### LPs

- Given a flow network  $G(V,E)$  with the set of lower bounds  $l(v,w)$  and upper bounds  $u(v,w)$  for each edge, and demands  $d(v)$  for each vertex, give an LP (with no objective function) to create a valid circulation.

### Google/Microsoft Interview Questions

- Given a number, describe an algorithm to find the next larger number that is prime.

### Google/Microsoft Interview Questions

- You are given a list of numbers. When you reach the end of the list you will come back to the beginning of the list (a circular list). Write the most efficient algorithm to find the minimum # in this list. Find any given # in the list. The numbers in the list are always increasing but you don't know where the circular list begins, ie: 38, 40, 55, 89, 6, 13, 20, 23, 36.

### Divide and Conquer

- Provide a divide-and-conquer algorithm for determining the largest and second largest values in a given unordered set of numbers. Provide a recurrence equation expressing the time complexity of the algorithm, and derive its solution.

## NP-Complete Reductions

- Dominating set: Given a graph  $G=(V,E)$  and integer  $0 < k \leq |V|$ , is there a subset  $D$  of  $V$  such that  $|D| \leq k$  and every vertex in  $V \setminus D$  is joined to at least one member of  $D$  by an edge in  $E$ ?
- Set cover: Given a universe  $U$ , subsets  $S_1, S_2, \dots, S_m$ , and integer  $k$ , are there  $\leq k$  subsets  $S_i$  such that their union is  $U$ ?
- Reduce dominating set to set cover

## NP-Complete Reductions

- Assume that 3-Color is NP-Complete. Prove that 4-Color is NP-Complete

## Google/Microsoft Interview Questions

- Assume you have an array that contains a number of strings (perhaps `char * a[100]`). Each string is a word from the dictionary. Your task, described in high-level terms, is to devise a way to determine and display all of the anagrams within the array (two words are anagrams if they contain the same characters; for example, `tales` and `slate` are anagrams.)

## Network Flow

- Let  $M$  be an  $n \times n$  matrix with each entry equal to either 0 or 1. Let  $m_{ij}$  denote the entry in row  $i$  and column  $j$ . A diagonal entry is one of the form  $m_{ii}$  for some  $i$ . Swapping rows  $i$  and  $j$  of the matrix  $M$  denotes the following action: we swap the values  $m_{ik}$  and  $m_{jk}$  for  $k=1,2,\dots,n$ . Swapping two columns is defined analogously.
- $M$  is rearrangeable if it is possible to swap some of the pairs of rows and some of the pairs of columns (in any sequence) so that after all the swapping, all the diagonal entries of  $M$  are 1.
- Is  $M$  rearrangeable? Poly-time algorithm: yes/no

## Selected Algorithms

## Interval Scheduling: Analysis

- Analysis:  $O(n \log n)$  time
  - Sorting  $O(n \log n)$
  - Check compatibility in  $O(1)$  by remembering last finish time

```
Sort jobs by finish times so that  $f_1 \leq f_2 \leq \dots \leq f_n$ .
set of jobs selected
A ← ∅
for j = 1 to n {
  if (job j compatible with A)
    A ← A ∪ {j}
}
return A
```

### Dijkstra's Algorithm: Time Analysis

Input: edge-weighted graph  $G = (V, E, \ell)$ , source  $s$ , sink  $t$   
 Let  $S$  be the set of explored nodes  
 For each  $u$  in  $S$ , store a distance  $d(u)$   
 Initialize  $S = \{s\}$  and  $d(s) = 0$   
 While  $S \neq V$   
     Select a node  $v$  not in  $S$  with at least one edge from  $S$  such that  
      $\pi(v) = \left( \min_{e = (u,v): u \in S} d(u) \right) + \ell_e$  is minimized  
     Add  $v$  to  $S$  and set  $d(v) = \pi(v)$  shortest path to some  $u$  in explored part, followed by a single edge  $(u, v)$   
 EndWhile

PQ Operation	Dijkstra	Array	Binary heap	d-way Heap	Fib heap †
Insert	$n$	$n$	$\log n$	$d \log_d n$	1
ExtractMin	$n$	$n$	$\log n$	$d \log_d n$	$\log n$
ChangeKey	$m$	1	$\log n$	$\log_d n$	1
IsEmpty	$n$	1	1	1	1
<b>Total</b>		$n^2$	$m \log n$	$m \log_{m/d} n$	$m + n \log n$

† Individual ops are amortized bounds

### Kruskal's Algorithm: Time Analysis

```

Kruskal(G, c) {
  Sort edge weights so that  $c_1 \leq c_2 \leq \dots \leq c_m$ .
  T ← ∅
  foreach (u ∈ V)
    make a set containing singleton u
  for i = 1 to m
    (u, v) =  $e_i$ 
    if (u and v are in different sets) {
      T ← T ∪ { $e_i$ }
      merge the sets containing u and v
    }
  return T
}
    
```

Use the **union-find** data structure.

- Sort:  $O(m \log n)$  time (since  $m = O(n^2)$ ,  $\log m = O(\log n^2) = O(2 \log n)$ )
- For all nodes:  $O(n)$  total time for MakeUnionFind
- For each edge:  $O(\log n)$  time for Find,  $O(1)$  time for Union
- Total:  $O(m \log n)$  time

### Optimal Prefix Codes: Huffman Encoding

```

Huffman(S) {
  if |S|=2 {
    return tree with root and 2 leaves
  } else {
    let y and z be lowest-frequency letters in S
    S' = S
    remove y and z from S'
    insert new letter @ in S' with  $f_{@} = f_y + f_z$ 
    T' = Huffman(S')
    T = add two children y and z to leaf @ from T'
    return T
  }
}
    
```

Build a tree for:  
 Alphabet: {a,e,k,l,u}  
 Frequencies:  $f_a=0.32, f_e=0.25, f_k=0.20, f_l=0.18, f_u=0.05$

### Counting Inversions: Implementation

Pre-condition. [Merge-and-Count] A and B are sorted.  
 Post-condition. [Sort-and-Count] L is sorted.

```

Sort-and-Count(L) {
  if list L has one element
    return 0 and the list L

  Divide the list into two halves A and B
  (r_A, A) ← Sort-and-Count(A)
  (r_B, B) ← Sort-and-Count(B)
  (r, L) ← Merge-and-Count(A, B)

  return r = r_A + r_B + r and the sorted list L
}
    
```

### Closest Pair Algorithm

```

Closest-Pair( $P_1, \dots, P_n$ ) {
  Compute separation line L such that half the points
  are on one side and half on the other side.  $O(n \log n)$ 
   $\delta_1$  = Closest-Pair(left half)
   $\delta_2$  = Closest-Pair(right half)
   $\delta = \min(\delta_1, \delta_2)$   $2T(n/2)$ 

  Delete all points further than  $\delta$  from separation
  line L  $O(n)$ 

  Sort remaining points by y-coordinate.  $O(n \log n)$ 

  Scan points in y-order and compare distance between
  each point and next 11 neighbors. If any of these
  distances is less than  $\delta$ , update  $\delta$ .  $O(n)$ 

  return  $\delta$ .
}
    
```

### Fast Matrix Multiplication

**Key idea.** multiply 2-by-2 blocks with only **7 multiplications**.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$\begin{matrix} P_1 = A_{11} \times (B_{12} - B_{22}) \\ P_2 = (A_{11} + A_{12}) \times B_{22} \\ P_3 = (A_{21} + A_{22}) \times B_{11} \\ P_4 = A_{22} \times (B_{21} - B_{11}) \\ P_5 = (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ P_6 = (A_{12} - A_{22}) \times (B_{21} + B_{22}) \\ P_7 = (A_{11} - A_{21}) \times (B_{11} + B_{12}) \end{matrix}$$

$$\begin{matrix} C_{11} = P_5 + P_4 - P_2 + P_6 \\ C_{12} = P_1 + P_2 \\ C_{21} = P_3 + P_4 \\ C_{22} = P_3 + P_1 - P_5 - P_7 \end{matrix}$$

- 7 multiplications.
- 18 = 8 + 10 additions and subtractions.

### Weighted Interval Scheduling: Memoization

**Memoization.** Store results of each sub-problem in a cache; lookup as needed.

```

Input: n, s1, ..., sn, f1, ..., fn, v1, ..., vn
Sort jobs by finish times so that f1 ≤ f2 ≤ ... ≤ fn.
Compute p(1), p(2), ..., p(n)

for j = 1 to n
    M[j] = empty
M[0] = 0

M-Compute-Opt(j) {
    if (M[j] is empty)
        M[j] = max(vj + M-Compute-Opt(p(j)), M-Compute-Opt(j-1))
    return M[j]
}
    
```

global array

### Knapsack Problem: Bottom-Up

**Knapsack.** Fill up an n-by-W array.

```

Input: n, W, w1, ..., wn, v1, ..., vn
for w = 0 to W
    M[0, w] = 0

for i = 1 to n
    for w = 1 to W
        if (wi > w)
            M[i, w] = M[i-1, w]
        else
            M[i, w] = max {M[i-1, w], vi + M[i-1, w-wi]}

return M[n, W]
    
```

### Sequence Alignment: Algorithm

```

Sequence-Alignment(m, n, x1x2...xm, y1y2...yn, δ, α) {
    for i = 0 to m
        M[0, i] = iδ
    for j = 0 to n
        M[j, 0] = jδ

    for i = 1 to m
        for j = 1 to n
            M[i, j] = min(α[xi, yj] + M[i-1, j-1],
                          δ + M[i-1, j],
                          δ + M[i, j-1])
    return M[m, n]
}
    
```

### Augmenting Path Algorithm

```

Augment(f, c, P) {
    b ← bottleneck(P)
    foreach e ∈ P {
        if (e ∈ E) f(e) ← f(e) + b
        else f(eo) ← f(eo) - b
    }
    return f
}
    
```

← the minimum cost of any edge along the path  
 forward edge  
 reverse edge  
 decrease the flow in the original graph along the edge by b if our path in the residual graph uses a reverse edge

```

Ford-Fulkerson(G, s, t, c) {
    foreach e ∈ E f(e) ← 0
    Gr ← residual graph

    while (there exists augmenting path P) {
        f ← Augment(f, c, P)
        update Gr
    }
    return f
}
    
```

### Capacity Scaling

```

Scaling-Max-Flow(G, s, t, c) {
    foreach e ∈ E f(e) ← 0
    Δ ← smallest power of 2 greater than or equal to C
    Gr ← residual graph

    while (Δ ≥ 1) {
        Gr(Δ) ← Δ-residual graph
        while (there exists augmenting path P in Gr(Δ)) {
            f ← augment(f, c, P)
            update Gr(Δ)
        }
        Δ ← Δ / 2
    }
    return f
}
    
```

### Load Balancing: List Scheduling

**List-scheduling algorithm.**

- Consider n jobs in some fixed order.
- Assign job j to machine whose load is smallest so far.

```

List-Scheduling(m, n, t1, t2, ..., tn) {
    for i = 1 to m {
        Li ← 0 ← load on machine i
        J(i) ← ∅ ← jobs assigned to machine i
    }

    for j = 1 to n {
        i = argmink Lk ← machine i has smallest load
        J(i) ← J(i) ∪ {j} ← assign job j to machine i
        Li ← Li + tj ← update load of machine i
    }
    return J(1), ..., J(m)
}
    
```

**Implementation.** O(n log m) using a priority queue.

Load Balancing: LPT Rule

Longest processing time (LPT). Sort n jobs in descending order of processing time, and then run list scheduling algorithm.

```
LPT-List-Scheduling(m, n, t1, t2, ..., tn) {
  Sort jobs so that t1 ≥ t2 ≥ ... ≥ tn
  for i = 1 to m {
    Li ← 0      ← load on machine i
    J(i) ← ∅    ← jobs assigned to machine i
  }
  for j = 1 to n {
    i = argmink Lk      ← machine i has smallest load
    J(i) ← J(i) ∪ {j}   ← assign job j to machine i
    Li ← Li + tj    ← update load of machine i
  }
  return J(1), ..., J(m)
}
```

k-Center Selection: Greedy Algorithm

Greedy algorithm [Gonzalez 1985]. Repeatedly choose the next center to be the site farthest from any existing center.

```
Greedy-Center-Selection(k, n, s1, s2, ..., sn) {
  C = ∅
  foreach si: dist(si, C) = infinity
  repeat k times {
    Select a site si with maximum dist(si, C)
    Add si to C
  }
  foreach si: dist(si, C) = distance to closest site in C
  return C
}
```

Observation. Upon termination all centers in C are pairwise at least  $r(C)$  apart.  
Pf. By construction of algorithm.

Pricing Method

Pricing method. Set prices and find vertex cover simultaneously.

```
Weighted-Vertex-Cover-Approx(G, w) {
  foreach e in E
    pe = 0
  while (∃ edge i-j such that neither i nor j are tight)
    select such an edge e
    increase pe as much as possible until i or j tight
  S ← set of all tight nodes
  return S
}
```

Weighted Vertex Cover: LP Relaxation

Weighted vertex cover. Linear programming formulation.

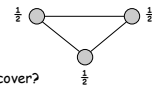
$$(LP) \min \sum_{i \in V} w_i x_i$$

$$\text{s.t. } x_i + x_j \geq 1 \quad (i,j) \in E$$

$$x_i \geq 0 \quad i \in V$$

Observation. Optimal value of (LP) is  $\leq$  optimal value of (IP).  
Pf. LP has fewer constraints.

Note. LP is not equivalent to vertex cover.



Q. How can solving LP help us find a small vertex cover?  
A. Solve LP and round fractional values.

Example Knapsack

Knapsack PTAS. Round up all values:  $\tilde{v}_i = \left\lceil \frac{v_i}{b} \right\rceil b$ ,  $\tilde{w}_i = \left\lceil \frac{w_i}{b} \right\rceil b$

```
Knapsack-Approx(ε) {
  b ← (ε/(2n)) vmax
  S ← solve Knapsack with values  $\tilde{v}_i$ 
  return S
}
```

$$\epsilon = 1/10$$

$$b = (\epsilon/(2 \cdot 5)) \cdot 27343199$$

$$= (1/100) \cdot 27343199$$

Item	Value	Weight
1	934,221	1
2	5,956,342	2
3	17,810,013	5
4	21,217,800	6
5	27,343,199	7

W = 11

original instance

Item	Value	Weight
1	4	1
2	22	2
3	66	5
4	78	6
5	100	7

W = 11

rounded instance

Contraction Algorithm

Contraction algorithm. [Karger 1995]

- Pick an edge  $e = (u, v)$  uniformly at random.
- Contract edge  $e$ .
  - replace  $u$  and  $v$  by single new super-node  $w$
  - preserve edges, updating endpoints of  $u$  and  $v$  to  $w$
  - keep parallel edges, but delete self-loops
- Repeat until graph has just two nodes  $v_1$  and  $v_2$ .
- Return the cut (all nodes that were contracted to form  $v_1$ ).



## Recall: Closest Pair Algorithm

```

Closest-Pair( $p_1, \dots, p_n$ ) {
  Compute separation line L such that half the points
  are on one side and half on the other side.  $O(n \log n)$ 
   $\delta_1 = \text{Closest-Pair}(\text{left half})$ 
   $\delta_2 = \text{Closest-Pair}(\text{right half})$ 
   $\delta = \min(\delta_1, \delta_2)$   $2T(n/2)$ 
  Delete all points further than  $\delta$  from separation
  line L  $O(n)$ 
  Sort remaining points by y-coordinate.  $O(n \log n)$ 
  Scan points in y-order and compare distance between
  each point and next 11 neighbors. If any of these
  distances is less than  $\delta$ , update  $\delta$ .  $O(n)$ 
  return  $\delta$ .
}

```

31