Final Review

Materials

- The final exam is **comprehensive**
- Focus on the materials after midterm

Time and Location

CS4321	R01	Thursday	3pm	325

Topics For Midterm

Topics	Reading
Introduction	1.1-1.3
Induction and Loop invariants	1.4-1.7, GT 1.3
Elementary Algorithmics	Chapter 2
Asymptotic Notation	Chapter 3
Algorithm Analysis - Analyzing control structures - Worst-case and Average-case - Amortized analysis	4.1-4.6
Solving Recurrences	4.7
Heap and Heap Sort	5.1-5.7
Binomial Heaps	5.8
Binary search tree, Splay Trees	GT 3.1, 3.4
Disjoint Set	5.9
Midterm	

Topics after midterm

Topics	Reading
Greedy Algorithms - Coin Change - Minimum Spanning Tree - Dijkstra's algorithm - Knapsack - Scheduling	6
Divide-and-Conquer - Mergesort and quicksort - Median - Strassen's algorithm - Closest pair	7

Topics After Midterm cont.

Topics	Reading
Dynamic Programming - Binomial coeffcient, coin change - 0-1 Knapsack - Floyd's algorithm - Matrix chain	Chapter 8
Exploring Graphs - Graph Search - Topological sorting - Brand and Bound	Chapter 9
Network Flow	GT 8.1-8.3
Decision Trees and Low Bound Arguments	12.1-12.3
P and NP Problems	12.4-12.6, 13.2

Study Guide

- Study the reviews for Midterm
- Study the questions on quizzes and homework assignments
- Understand how the covered algorithms work

Greedy algorithms

- Know the paradigm
  - Template of a greedy algorithm
  - Be able to design and analyze a greedy algorithm
- Understand the following algorithms
  - Coin change
  - MST (Prim's algorithm and Kruscal's algorithm)
  - Dijkstra's algorithm (single source shortest path)
  - Knapsack
  - Scheduling for shortest total response time

General characteristics of greedy algorithms

```
makeChange(int n)
{
   C = a set of available coins;
   S=∅;  // chosen coins
   s=0;
   while (C != ∅ && s != n) {
     x = a coin in C with the largest value
         such that s+x<=n;
     if no such x exists
           return "no solution found";
     else {
        C = C \ {x};
        S = S∪{x};
        s = s+x;
     }
   }
}
```

```
greedy(SET C)
{
   // C is set of candidates
   S= ∅;  // S is a partial solution

   while (C != ∅) {
     x = select(C);
     if  (feasible(S∪{x}))
        S = S ∪ {x};
     if (solution(S))
       return S;
   }
   return "no solutions";
}
```

objective() function is implicit

Kruskal's algorithm -- efficiency

$\Theta(a\log a)$

called at most a times each

called n-1 times

$\Theta(2a\ \alpha(2a, n))$

```
Kruskal(Graph G)  // G=<N,A>
{
   sort A by increasing weight;
    n = #nodes in N;
    T = ϕ ;
    make n initial sets, each contains a node in N;

   do { // for all sorted edges
      e = <u,v>; // shortest edge not yet considered
      uComponent = find(u);
      vComponent = find(v);

      if (uComponent != vComponent) {
         merge(uComponent, vComponent);
         T = T ∪ {e};
      }
   }  while (!(T contains n-1 edges))
   return T;
}
```

Prim's algorithm

```
Prim(int L[][])
{
   T = ∅;
   for (i=2; i<=n; i++) {          Θ(n)
      nearest[i] = 1;
      mindist[i] = L[i,1];
   }

   for (i=2; i<n; i++) {            Θ(n²)
      min = ∞;
      for (j=2; j<=n; j++) {
         if  (mindist[j] >=0 && mindist < min) {
            min = mindist[j];
            k = j;
         }
      }
```

```
         T = T ∪ <nearest[k], k>;
         mindist[k] = -1;

         for (j=2; j<=n; j++) {
            if (L[j,k] < mindist[j]) {
               mindist[j] =L[j,k];
               nearest[j] = k;
            }
         }
      }
   }
}
```

Dijkstra's algorithm

C: candidate set
S : partial solution set
L[i][j]: weight of edge <i,j>
D[i]: length of the special path for the source to node i.
P[i]: the previous node of i along its shortest path.

```
Dijkstra(Weight L[][])
{
   /* initialization */
   C = {i | 2 <= i <= n};   // S = {1}
   for (i=2; i<=n; i++) {
      D[i] = L[1,i];
      P[i] = 1;
   }

   for (i=1; i<=n-2; i++) { // repeat n-2 times
      v = some element of C minimizing D[v];
      C = C – {v};  // S = S ∪ {v}
      for (each w ) {
         if ( D[v]+L[v,w]  < D[w]) {
            D[w] = D[v]+L[v,w];
            P[w] = v;
         }
      }
   }
}
```

Analysis of Dijkstra's algorithm: using heap

a: #edges
n: #nodes

```
Dijkstra(Weight L[][])
{
   /* initialization */
   C = {i | 2 <= i <= n};   // S = {1}
   for (i=2; i<=n; i++) {                    Θ(n)
      D[i] = L[1,i];
      P[i] = 1;
   }
   buildHeap(D); // inverted heap              Θ(n)

   for (i=1; i<=n-2; i++) { // repeat n-2 times
      v = findMin(D[]);                       Θ(1)
      deleteMin(v);       //C = C – {v}; S = S ∪ {v}     O(log n)
      for (each w ) {                         At most a times if use adjacent list
         if ( D[v]+L[v,w]  < D[w]) {
            D[w] = D[v]+L[v,w];
            percolate(v);                     O(log n)
            P[w] = v;
         }
      }
   }
}
```

Total: O((a+n) log n)
=O(a log n)

The knapsack problem

- Given
  - n objects numbered from 1 to n. Object i has a positive weight w_i and a positive value v_i
  - a knapsack that can carry a weight not exceeding W
- Problem
  - Fill the knapsack in a way that maximize the value of the included objects, while respecting the capacity constraints
  - In this version, we assume that the objects can be broken into small pieces

A greedy algorithm

```
Knapsack(w[], v[], W)
{
   for (i=1; i<=n; i++)
     x[i] =0;
   weight = 0;

    while (weight < W) {
        i = select the best remaining object;
        if (weight + w[i] < W)
           x[i] = 1;
        else
          x[i] = (W-weight)/w[i];
    }
    return x;
}
```

The key is which object to select

Scheduling

- Minimizing time in the system
  - Know the problem
  - Know the proof
  - Know the algorithm
- Scheduling with deadlines (not required)

Minimizing time in the system

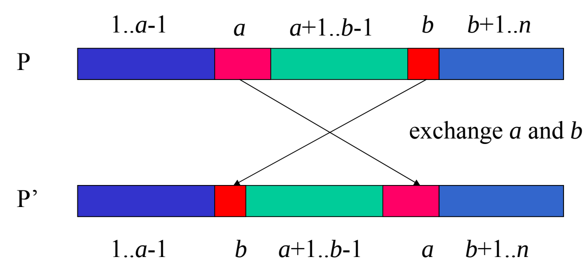
- Assume we submit *n* jobs into a system at the same time.
  - The service time of job *i*, t_i, know in advance
- Problem
  - Design an algorithm that minimizing the average response time
  - This is equivalent to
    - Minimizing the total response time
    - $T = \sum_{i=1}^{n}$ (response time of job j)

A greedy algorithm

- Algorithm
  1. Sort the jobs by their service times
  2. Repeat
     1. Serve the job with minimal service time among the remaining jobs
- Analysis
  - Step 1: O(n log n)
  - Step 2: Θ(n)
  - Total: O(n log n)

Optimality of the greedy scheduling algorithm

- Theorem 6.6.1. The greedy algorithm is optimal



Compares schedules P and P', job a at P' leaves at the same time as job b in P. Jobs b and a+1 to b-1 in P' leaves earlier then the corresponding jobs in P.

Optimality of the greedy scheduling algorithm

$$
\begin{aligned}
T(P) &= s_1 + (s_1 + s_2) + \dots (s_1 + s_2 + \dots s_n) \\
&= n s_1 + (n-1) s_2 + \dots + 1 s_n \\
&= \sum_{k=1}^{n} (n-k+1) s_k
\end{aligned}
$$

$$T(P) = (n-a+1)s_a + (n-b+1)s_b + \sum_{k=1, k\neq a,b}^{n} (n-k+1)s_k$$

$$T(P') = (n-a+1)s_b + (n-b+1)s_a + \sum_{k=1, k\neq a,b}^{n} (n-k+1)s_k$$

$$T(P) - T(P') = (n-a+1)(s_a - s_b) + (n-b+1)(s_b - s_a) = (b-a)(s_a - s_b) > 0$$

Divide and Conquer

- Given a problem, know how to design a D&C algorithm
- Know how to analyze a D&C algorithm
  - You need to remember the simple version of the Master Theorem.
- Know the following algorithms
  - Merge sort
  - Quick sort
  - Find median
  - Closest pair

Running-time analysis

- Assume that the *l* sub-instances have roughly the same size n/*b* for some constant *b*
- Let g(n) be the time required by DC on instances of size n, excluding the times need for the recursive calls. We have
  - $t(n) = l \cdot t(n/b) + g(n)$
- If $g(n) \in \Theta(n^k)$ for an integer k, we have

$$t(n) \in \begin{cases} \Theta(n^k) & \text{if} \quad l < b^k \\ \Theta(n^k \log n) & \text{if} \quad l = b^k \\ \Theta(n^{\log_b l}) & \text{if} \quad l > b^k \end{cases}$$

Merge sort

```
mergeSort(int T[n])
{
   if (n is sufficiently small)
     insertionSort(T);
   else {
     int U[n/2], V[(n+1)/2];
     copy T[1..n/2] to U[1..n/2];
     copy T[n/2+1,..,n] to V[1.., (n+1)/2];
     mergeSort(U[n/2]);
     mergeSort(V[(n+1)/2);
     merge(U,V,T);
   }
}
```

Merge two sorted arrays

```
Merge(U[m], V[n], T[m+n])
// merge sorted arrays U and V into T
{
   u = 0;   // cursor for U
   v = 0;   // cursor for V
   U[m] = V[n] = +∞; // sentinels
   for (t=0; t<m+n; t++) {  // t is cursor for T
     if (U[u] < V[v]) {
         T[t] = U[u];
         u++;
     } else {
        T[t] = V[v];
        v++;
     }
   }
}
```

What to do if we do not use the two sentinels?

Quick Sort

- Choose an element from the array to be sorted as a pivot
- Partition the array on either side of the pivot such that those no smaller than the pivot are to its right and those no greater are to its left
- Recursive calls on both sides

The algorithm

```
quickSort(T, i, j)
{
   l = pivot(T, i, j); // l is the position of the pivot at end
   if (i<l-1)
      quickSort(T, i, l-1);
   if  (l+1<j)
      quickSort(T, l+1, j);
}
```

Pivot I

```
int pivot(T, i, j)
{
  // choose T[i] as the pivot
  p = T[i];
  l = i;   //  left cursor
  r = j+1;   // right cursor
  do {
     l++;
  } while (T[l]<=p and l < r)

  do {
     r--;
  } while (T[r] > p);
```

```
 while (l<r) {
    swap(T[l], T[r]);
    do {
       l++;
    } while (T[l] <= p);
     do {
       r--;
     } while (T[r] >  p)
  }
  swap(T[i], T[r]);
  return r;
}
```

T[i] is at the bound after the algorithm

Analysis

- Worst case: the array is sorted, $\Omega(n^2)$
- Best case
  - $T(n) = 2T(n/2) + \Theta(n)$, $T(n) \in \Theta(n \log n)$
- Average case (not required)

Selection using pseudomedian

```
selection(T[1..n], s)
{
   l = 1; r = n;
   while (true) {
      x = pseudomedian(T[l..r]);
      p = pivot(T[l..r], x);
      if (s<p) r = p-1;
      else if (s>p) l = p+1;
      else return p;
   }
}
```

```
pseudomedian(T[1..n])
{
   if (n <= 5)
      return adhocmedian(T);
   z = ⌊n/5⌋;
   for (i=1; i<=z; i++)
      Z[i] = adhocmedian(T[5i-4..5i]);
   return selection(Z, ⌈z/2⌉);
}
```

We assume the elements are distinct.
You need to know the time complexity of this algorithm

Closest Pair

- Problem
  - Given n points on a two-dimension space, find the closest pair
- A simple algorithm
  - Calculate the distance for all possible pairs, find a smallest one
    - Total $\binom{n}{2}$ pairs
    - Cost: $\Theta(n^2)$
- A better algorithm
  - Divide-and-conquer

Algorithm

```
double closestPair(Points p)
{
   n = p.size();
   mergeSort(p);  // by x-coordinate
   return recursiveClosestPair(p, 1, n)
}
```

```
double recursiveClosestPair(p, i, j)
{
   if ( j-i < 3) {
     return adhocClosest(p, i, j);
     sort p[i..j] by y-coordinate;
   }

   k = (i+j)/2;
   deltaL = recursiveClosestPair(p, i, k);
   deltaR = recursiveClosestPair(p, k+1, j);
   delta = min(deltaL, deltaR);
   return findClosestInStrip(p, i, j, delta);
}
```

Note: p[i..j] are sorted by x-coordinate before getting in recursiveCloestPair();
sorted by y-coordinate after it returns;

findClosestInStrip

m = j-i+1

```
findClosestInStrip(p, i, j, delta)
{
   k = (i+j)/2;       l = p[k].x;
   // p[i..k] sorted by y-coordinate
   // p[k+1..j] sorted by y-coordinate
   merge(p, i, k, j);  // p[i..j] sorted by y-coordinate

   t = 0;
   for (k=i; k<=j; k++) {
     if (p[k].x > l – delta
      && p[k].x < l+delta)  // in the strip
       v[++t] = p[k];
   }
   for (k=1; k<t; k++) {
        for (s=k+1; s<=min(t, k+7); s++)
        delta = min(delta, dist(v[k], v[s]));
   }
   return delta
}
```

Cost?

$\Theta(m)$

$\Theta(m)$

O(m)

Total: $\Theta(m)$

Dynamic Programming

- Given a problem, know how to design a dynamic programming algorithm
- Know how to analyze a DP algorithm
- Know the following algorithms
  - Calculating Binomial Coefficient
  - Coin Change
  - 0-1 Knapsack
  - Floyd's algorithm
  - Matrix chain

Example: Binomial Coefficient

- We want to calculate $\binom{n}{k}$ which can be defined as follows.

$$\binom{n}{k} = \begin{cases} 1 & if \quad k = 0, n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & if \quad 0 < k < n \\ 0 & otherwise \end{cases}$$

A solution using dynamic programming

- We instead calculate bottom-up by filling the following table

n \ k	0	1	2	…….	k-1	k
0	1					
1	1	1				
2	1	2	1			
…						
n-1					C(n-1,k-1)	C(n-1,k)
n						C(n.k)

Cost: time Θ(nk) and space Θ(k)

Make change

- The principle of optimality
  - In an optimal sequence of decisions or choices, each subsequences must be also optimal
- Making a change
  - *c[i,j]* is an optimal solution to for making a change of *j* using the first *i* denominations
    - If at lease one coin of denomination *i* is used, then $c[i, j]=c[i, j-d_i]+1$
    - Otherwise, $c[i, j]=c[i-1, j]$
    - We obtain $c[i,j]=min(c[i-1, j], c[i, j-d_i]+1)$

Example

- Pay 8 units with coins worth 1, 4, and 6 units.
- Construct the following table top-down, left to right.

Amount	0	1	2	3	4	5	6	7	8
d1=1	0	1	2	3	4	5	6	7	8
d2=4	0	1	2	3	1 ←	2	3	4	2 ←
d3=6	0	1	2	3	1	2	1	2	2 ↑

c[3,8] = min(c[2,8], c[3,8-6]+1) = min(2, 3) = 2
c[2,8] = min(c[1,8], c[2,4]+1) = min(8, 2)=2
c[2,4] = min(c[1,4], c[2,0]+1) = min(4, 1) =1

The optimal solution 2. The greedy algorithm returns 3.

0-1 Knapsack

- n objects 1, 2, ..., n. Object i has weight w_i and value v_i
- The knapsack can carry a weight not exceeding W.
- Cannot split an object
- Maximize the total value
  - Maximize $\sum_{i=1}^{n} x_i v_i$ subject to $\sum_{i=1}^{n} x_i w_i \le W$,

where v_i , $w_i > 0$ and $x_i \in \{0,1\}$ for $1 <= i <= n$

By dynamic programming

- Set up a table C[0..n, 0..W] with one row for each available object and one column for each weight from 0 to W. Specifically, C[0, j] = 0 for all j.
- C[i,j] is the maximum value if the weight limit is j and only objects 1 to i are available
  - $C[i,j] = \max(C[i-1,j], C[i-1, j-w_i]+v_i)$;
- C[n,W] will be the solution

Algorithm

```
Knapsack0-1(v, w, n, W)
{
   for (w = 0; w<=W; w++) {
     c[0,w] = 0;
   }
   for (i=1;  i <=n; i++) {
       c[i,0] = 0
      for (w=1;  w<=W; w++) {
         if (w[i] <  w) {
            if  (c[i-1,w-w[i]]+v[i] > c[i-1,w])
                c[i,w] = c[i-1,w-w[i]] + v[i];
            else  c[i,w] = c[i-1,w]
         } else  c[i,w] = c[i-1,w]
      } // for w
   } for i
}
```

The run time performance of this algorithm is $\Theta(nW)$

Finding the objects

```
i=n;
k=W;
while (i>0 && k>0) {
  if (C[i,k] <> C[i-1,k]) {
       mark the i-th object as in knapsack;
       k = k-w[i];
       i = i-1;
  } else
      i = i-1;
}
```

Cost: O(n+W)

Floyd's algorithm

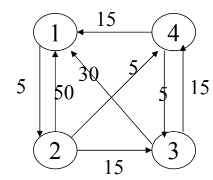
```
Floyd(int L[1..n][1..n])
{
   D = L;
   P = 0;
   for (k=1; k<=n; k++)
      for (i=1; i<=n; i++)
         for (j=1; j<=n; j++)
            if (D[i][k]+D[k][j] < D[i][j]) {
               D[i][j] = D[i][k]+D[k][j];
               P[i][j] = k;
            }
}
```

Track the shortest path

Cost: $\Theta(n^3)$

Why can we use just one array D?

Example



$$D_0 = L = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix}$$

$$D_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \underline{35} & 0 & 15 \\ 15 & \underline{20} & 5 & 0 \end{pmatrix} \quad D_2 = \begin{pmatrix} 0 & 5 & \underline{20} & \underline{10} \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \quad D_3 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ \underline{45} & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_4 = \begin{pmatrix} 0 & 5 & \underline{15} & 10 \\ \underline{20} & 0 & \underline{10} & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \quad P = \begin{pmatrix} 0 & 0 & 4 & 2 \\ 4 & 0 & 4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Find the shortest path

```
// Print shortest path between
  nodes i and j
printShortestPath(int i, int j)
{
    print i;
    printIntermediateNodes(i, j);
    print j;
}
```

```
// Print intermediate nodes in the shortest
   path between nodes i and j
printIntermediateNodes(int i, int j)
{
   int k = p[i][j];
   if (k == 0)
      return;
   else {
      printIntermediateNodes(i, k);
      print k;
      printIntermediateNodes(k, j);
   }
}
```

Find the shortest path between nodes 1 and 3 for the example.

Chained Matrix Multiplication

- Problem
  - Find an optimal *parenthesization* of chained matrix multiplication
- Chained matrix multiplication
  - $M = M_1M_2\dots M_n$
- Parenthesization
  - A product of matrices is fully parenthesized if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses

A recursive equation

- Let *m[i,j]* be the minimum number of scalar multiplications needed to compute $M_{i..j}$
  - The cost for is $M_{1..n}$ is *m[1,n]*
- Assume that the optimal parenthesization splits the product $M_i M_{i+1} \ldots M_j$ between M_k and M_{k+1}
  - Based on the principle of the optimality

    $m[i,j] = m[i,k]+m[k+1,j]+d_{i-1}\, d_k\, d_j$
  - We obtain the following recurrence

$$m[i,j] = \begin{cases} 0, & i = j \\ \min\limits_{i \le k < j}(m[i,k] + m[k+1,j] + d_{i-1}\, d_k\, d_j), & i < j \end{cases}$$

An implementation using dynamic programming

```
matixChain()
{
   for (i=1; i<=n; i++) // sequence of length 1
      m[i][i]=0;

   for (s=2; s<=n; s++) // s is the length of the sequence
      for (i=1; i<=n-s+1; i++) {
         j = i+s-1;
         m[i][j] = ∞;
         for (k=i; k<=j-1; k++)
            tmp = m[i,k]+m[k+1,j]+d[i-1]d[k]d[j]
            if (tmp < m[i][j]) {
               m[i][j] = tmp;
               p[i][j] = k; // split point
            }
      }
}
```

Cost:

$$n + \sum_{s=2}^{n}\sum_{i=1}^{n-s+1}(s-1)$$
$$= n + \sum_{s=2}^{n}(n-s+1)(s-1)$$
$$= n + \sum_{l=1}^{n-1}(n-l)l$$
$$= n + n\sum_{l=1}^{n-1} l - \sum_{l=1}^{n-1} l^2$$
$$= n + \frac{n^2(n-1)}{2} - \frac{n(n-1)(2n-1)}{6}$$
$$= (n^3 + 5n)/6$$
$$\in \Theta(n^3)$$

Example

M1	13×5
M2	5×89
M3	89×3
M4	3×34

i \ j	1	2	3	4
1	0	5785	1530	2856
2		0	1335	1845
3			0	9078
4				0

m[1][3]=min(m[1][1]+m[2][3]+13*5*3, m[1][2]+m[3][3]+13*89*3)=min(1530,9256)=1530
m[2][4]=min(m[2][2]+m[3][4]+5*89*34, m[2][3]+m[4][4]+5*3*34)=min(24208,1845)=1845

min[1][4]=min(m[1][1]+m[2][4]+13*5*34, k=1
m[1][2]+m[2][4]+13*89*34, k=2
m[1][3]+m[4][4]+13*3*34) k=3
=min(4055, 54201, 2856) = 2856

Graph Traversal

- Know the depth search, breadth search, and topological sort algorithms
- Know how to use graph search to solve problems
- Topics
  - Tree traversal
  - Preconditioning
    - find ancestor in a tree
  - Graph search
    - Breadth first
    - Depth first
      - Articulation points (not required)
      - Topological sort

Graph search: some concepts

- To keep track of progress, graph search colors each node *white, gray, or black*
  - All nodes start with *white*
  - A node is *discovered* at the first time it is encountered during the search, at which time it becomes *non-white*
  - Different search distinguishes itself by a different way to *blacken* or *gray* nodes

Breadth-first search

- Given a graph $G=<N, E>$, and a source node, s, start breadth-first search from s.
- Expands the frontier between discovered and undiscovered nodes uniformly across the breadth of the frontier
  - Discovers all nodes at distance k from s before discovering any nodes at distance $k+1$.
- Coloring: if $(u,v) \in E$ and vertex u is black, then node v is either black or gray
  - Black node: discovered and the node itself is finished
  - Gray node: discovered but not finished

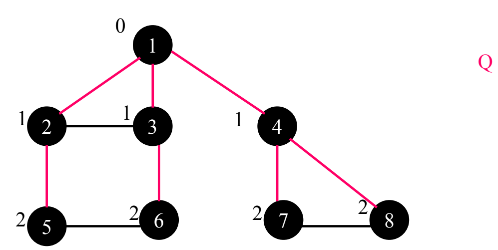
Breadth-first search algorithm

```
BFS(G,s)
{
   for each node u ∈ N – {s} {
      color[u] = WHITE;
      d[u] = ∞;
      π[u] = null;
   }

   color[s] = GRAY;
   d[s] =0
   enqueue(Q, s);
```

```
while (!empty(Q)) {
   u = dequeue(Q);
   for each v adjacent to u {
      if (color[v] == WHITE) {
         color[v] = GRAY;
         d[v] = d[u] + 1;
         π[v] = u;
         enqueue(Q, v);
      }
   }
   color[u] = BLACK;
}
```

d[]: tracks shortest distance, assuming each edge's weight is 1
π[]: tracks the parent-child relationship in the breadth-first tree

Breadth-first Example



Depth-first search

- Search deeper in the graph whenever possible
  - Edges are explored out of the most recently discovered node *v* that still has undiscovered edges leaving it
  - When all of *v*'s edges have been explored, the search "backtracks" to explore edges leaving the vertex from which *v* was discovered
  - This process finishes until all nodes reachable from the original source are discovered
  - Select one undiscovered node as the new source and continue the process

Depth-first search coloring and time stamps

- Coloring
  - Each nodes is initially *white*
  - A node is *grayed* if it is discovered during the search and *blackened* if it is finished, that is, when its adjacency list has been examined completely
- Timestamps
  - Each node *v* has two timestamps
    - *d[v]* records when v is discovered (grayed)
    - *f[v]* records when v is finished (blackened)

Depth-first search algorithm

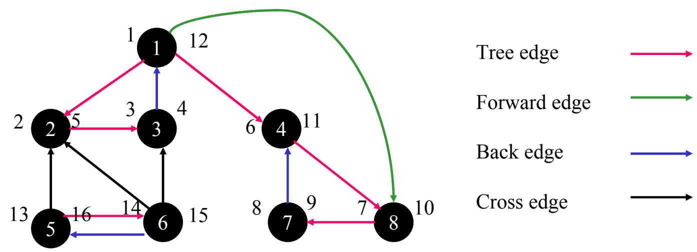
```
DFS(G)
{
   for each node u ∈ N {
      color[u] = WHITE;
      π[u] = null;
   }

   time = 0;

   for each node u ∈ N {
      if (color[u] == WHITE)
         DFS-Visit(u);
   }
}
```

```
DFS-Visit(u)
{
   color[u] = GRAY;
   d[u] = ++time;

   for each v adjacent to u {
      if  (color[v] == WHITE) {
         π[v] = u;
         DFS-Visit(v)
      }
   }

   color[u] = BLACK;
   f[u] = ++time;
}
```

Classification of graph edges

- After depth-first search of a directed graph, we can classify the graph edges into four categories
  - Tree edge
    - An edge in the search tree
  - Back edge
    - An edge *(u,v)* not in search tree and *v* is an ancestor of *u*
    - Indicates a loop
  - Forward edge
    - An edge *(u,v)* not in search tree and *u* is an ancestor of *v*
  - Cross edge
    - An edge *(u,v)* not in search tree and *v* is neither an ancestor nor a descendant of *u*

Example: depth-first search directed graph



Topological Sort

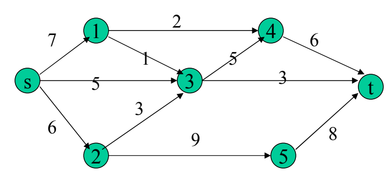
- Given an acyclic directed graph, topological sort finds a topological ordering of the nodes such that if there exists an edge *(u,v)*, then node *u* precedes node *v* in the ordering list.
- The finished time numbering gives us a reverse topological ordering
  - A node is finished after all the nodes it reaches have finished

Branch-and-Bound

- Not required

Maximum Flow Problem

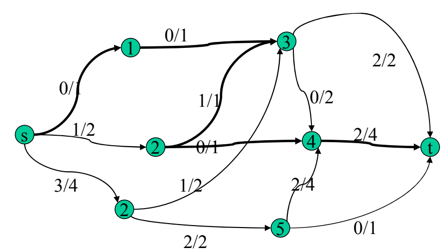
- Given a weighted directed graph
  - Each edge is a pipe whose weight denotes its capacity: the maximum amount it can transport
    - Use *c(e)* for the capacity of edge *e*
  - Given a source, *s*, and a sink, *t*, find the maximum amount (flow) can transfer from *s* to *t*



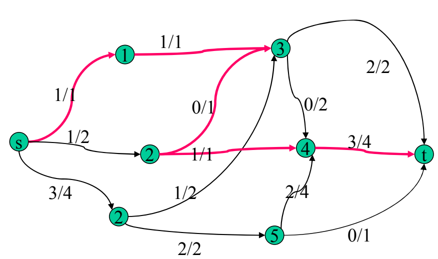
Concepts

- You need to know the follow concepts
  - Flow of a network
  - Capacity and flow of an edge
  - Cut
    - Capacity and flow of a cut
  - Residual capacity of an edge and a path
  - Augmenting Path
    - Residual capacity

Example of the Ford-Fulkerson Algorithm



Example of the Ford-Fulkerson Algorithm



The Ford-Fulkerson Algorithm

```
maxFlowFordFulkerson(N)
 // N =(G, c, s, t)
{
   for each edge e in N do {
     f(e) = 0;
     stop = false;
   }
   while (!stop) {
       traverse G starting at s to find an augmenting path for f;
       if an augmenting π path exists {
           Δ = minimum Δf(e) along π;
           for each edge e in π {
               if (e is an forward edge) f(e) += Δ; else f(e) -= Δ;
           }
       } else
           stop = true;
   }
}
```

The Edmonds-Karp Algorithm

- Try to find a "good" augmenting path each time
  - Choose an augmenting path with the smallest number of edges
    - Can be implemented using BFS traversal

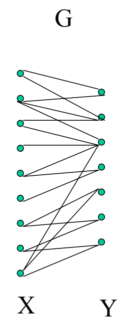
Maximum Bipartite Matching

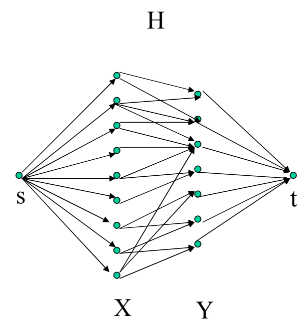
- Bipartite graph
  - a graph with vertices partitioned into two sets X and Y, such that every edge has one endpoint in X and the other in Y
- Matching in a bipartite graph
  - A set of edges that has no end points in common
- Maximum bipartite matching
  - The matching with the greatest number of edges

Reduction to the Maximum Flow Problem

- Let G be a partite graph whose vertices are partitioned into sets X and Y. Create a flow network H as follows
  - Add each vertex of G into H plus a source vertex s and a sink vertex t.
  - Add edges of G into H and make each edge orient from an endpoint in X to an endpoint in Y
  - Insert a directed edge from s to each vertex in X
  - Insert a directed edge from each vertex in Y to t
  - Assign each edge in H a capacity of 1

An example of reduction





All edges with capacity 1

Reduction to the Maximum Flow Problem

- Given the maximum flow f of H, define M as a set of edges such that e in M iff f(e) =1
  - M is a matching
  - M is a maximum matching
- Reverse transformation: given a matching M in H, define a flow f
  - For each edge e of H that is also in G, f(e) = 1 if e ∈M and f(e) = 0 otherwise.
  - For each edge of H incident to s or t and v be the other end point, f(e) = 1 if v is is an endpoint of some edge of M and f(e) = 0 otherwise

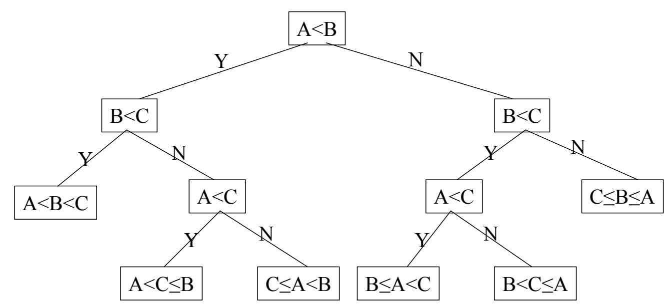
Complexity

- Arguing a lower bound
  - Information-theoretic arguments
- Given a comparison based problem
  - Know how to construct a decision tree
  - Know how to argue the lower bound using decision tree
  - Adversary arguments
- Proof equivalence of complexity or compare complexity
  - Linear reductions
  - NP-completeness

Information-theoretic arguments

- Particularly applies to those problem involving comparisons
- Uses a decision tree to represent the working process of an algorithm on all possible data of a given size
  - A decision tree is a binary tree where
    - Each internal node contains a test on the data
    - Each leaf contains an output, called *verdict*
    - A *trip* through the tree starts from the root and recursively goes to the left subtree or the right subtree depending on whether the answer to the root is "yes" or "no"
    - The trip ends when it reaches a leaf (verdict)

A decision tree for sorting

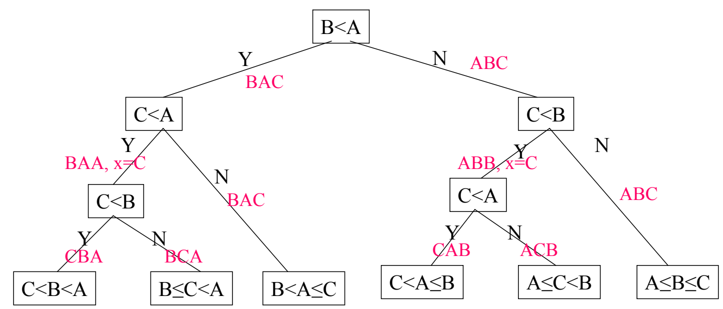


Note that the number of leaves = The number of possible outputs

Insertion Sort

```
void insertionSort(int A[], int n)
{ // array index starts from 1 here
  int i, j,  x;

   for (i=2; i<=n; i++) {
     x =A[i];
     j = i-1;
     while (j>0 && x<A[j]) {
       A[j+1] = A[j];
       j--;
     }
     A[j+1] = x;
   }
}
```

A three-item insertion sort decision tree



Observations

- The number possible outputs = The number of leaves (verdicts)
- The worst-case time is the height of the tree
- The average time is the average depth of leaves assuming equal distribution

Theorem

- Any binary tree with *k* leaves has an average height of at least *lg k*
- Any comparison-based algorithm takes a worst case time and average case time $\Omega(n \log n)$