Final Review

Materials

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

Time and Location

CS4321	R01	Tuesday	3pm	Fisher 139

Topics before midterm

Topics	Reading
Introduction	1
Induction and loop invariants	2.1
Asymptotic Notation	3.1-3.2
Algorithm Analysis - Analyzing control structures - Worst-case and Average-case - Amortized analysis	2.2 5.1-5.3 17.1-17.3
Solving Recurrences	4.1-4.3
Heap and Heap Sort	6
Binomial Heaps	19
In-class mid term	

Topics and Exams

Topics	Reading
Splay Trees	Notes, handouts
Disjoint Set	21.1-21.3
Greedy Algorithms - Greedy Strategy; Knapsack; Activity Selection - Minimum Spanning Tree - Dijkstra's algorithm	16.1-16.2 23 24.3
Divide-and-Conquer - Mergesort and Quicksort - Median	2.3 7.1-7.4 9

Topics and Exams

Topics	Reading
Dynamic Programming - 0-1 Knapsack - Matrix chain - Longest common subsequence - Floyd's algorithm	15.1-15.4 25.1-25.2
Exploring Graphs - Graph Search - Topological sorting	22.1-22.4
Network Flow and Matching	26.1-26.3
P and NP Problems	34-35
Final	

Study Guide

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

Splay Tree

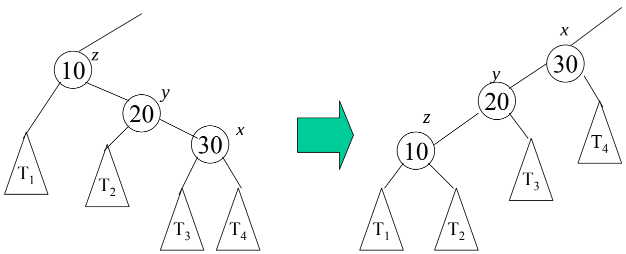
- A splay tree is a special binary search tree
- Know how to insert, search, and delete a node
- Know how to splay

Splay Trees

- Apply *splaying* after every access to keep the search tree balanced in an amortized sense
- Splaying
  - Splay *x* by moving *x* to the root through a sequence of restructurings
  - One specific operation depends on the relative positions of *x*, its parent *y*, and its grandparent *z*
    - Zig-Zig
    - Zig-Zag
    - Zig

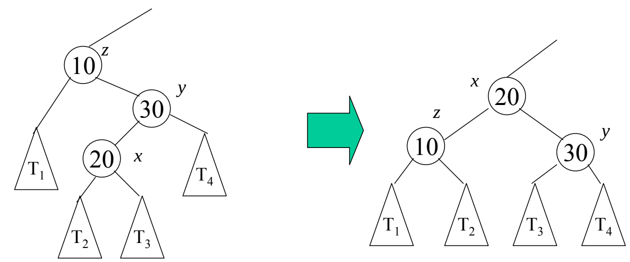
Splay *x*: zig-zig

The node *x* and its parent *y* are both left or right children



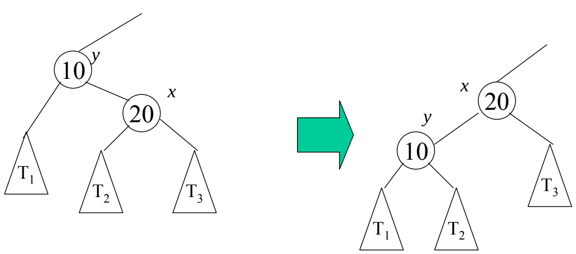
Splay *x*: zig-zag

One of *x* and *y* is a left child and the other is a right child.



Splay *x*: zig

The node x does not have a grandparent (or the grandparent is not of our concern)

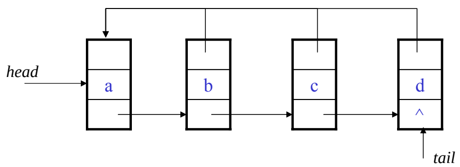


Disjoint set structures

- Definition
  - A collection of disjoint dynamic sets
  - Each set is identified by a representative which is some member of the set
- Three functions
  - *MakeSet(x)* – create a new set whose only member is x
  - *FindSet(x)* – find the set that contains *x;* return the representative
  - *Union(x, y)* – unites the corresponding sets that contains *x* and *y* respectively and choose a representative for the combined set
- Know the linked-list implementation
  - Know how to do weighted-union
- Know the disjoint-forest implementation
  - Know path compression and union by rank

Linked List Representation

- Represent each set as a link list
  - The first object is the representative
  - A pointer, *head*, pointing to the representative
  - A pointer, *tail*, pointing to the last object of the list
    - For easy union
  - Each object has two pointers
    - *next*: points to the next object in the list
    - *rep*: points to the representative

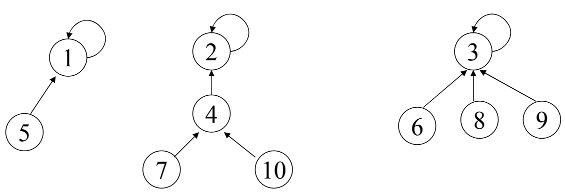


Weighted-Union for Linked-list

```
WeightedUnion(x, y)
{
   if (x.length > y.length) {
     Union(y, x); // append y to x
     x.length += y.length;
   } else {
     Union(x, y); // append x to y
     y.length += x.length
   }
}
```

Disjoint-set forest

- Represent each set as a rooted tree
  - Each member points only to its parent
  - The root is the representative
  - The root's parent is itself



Implementation: disjoint-set forest

```
FindSet(x)
{
  r = x;
  while (r.parent != r)
    r = r.parent;
  return r;
}
```

$\Theta(n)$ in worst case

```
MakeSet(x)
{
   x.parent = x;
}
```

$\Theta(1)$

```
Union(x, y)
{
   Link(FindSet(x), FindSet(y));
}
```

$\Theta(n)$ in worst case

```
Link(x, y)
{
   x.parent = y;
}
```

$\Theta(1)$

$\Theta(n^2)$ in worst case for m operations when $m \in \Theta(n)$

Improvements: two heuristics

- Union by rank
  - Rank: for each node, its rank is an upper bound on its height
  - Union: the root with smaller rank is made pointed to the root with larger rank
- Path compression
  - Make each node on the *find path* directly point to the root
  - *find path*: the path FindSet goes through

Union by rank

```
Link(x,y)
{
   if (x.rank > y.rank) {
     y.parent = x;
  } else {
      x.parent = y;
      if (x.rank == y.rank)
         y.rank++;
  }
}
```

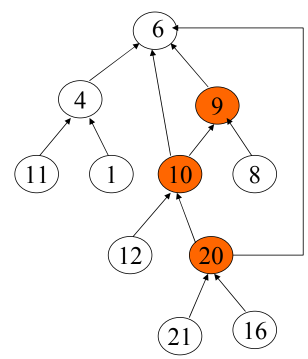
$\Theta(1)$

Path compression

- Squash the path when doing FindSet(), so the next FindSet() will be likely quicker (path compression).
  - first pass to find the root
  - second pass change the pointers along the path to the root and make them all point to the root

```
FindSet(x)
{
    r = x;
    while (r.parent <> r)
        r = r.parent;

    i = x;
    while (i <> r) {
        j = i.parent;
        i.parent = r;
        i = j;
    }
    return r;
}
```

An example



Greedy algorithms

- Know the paradigm
  - Be able to design and analyze a greedy algorithm
  - Be able to argue the optimality of a greedy algorithm using "cut&paste".
- Understand the following algorithms
  - Knapsack
  - Activity Selection
  - MST (Prim's algorithm and Kruscal's algorithm)
  - Dijkstra's algorithm (single source shortest path)

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
  - Fractional Knapsack Problem
    - the objects can be broken into small pieces
  - 0-1Knapsack Problem:
    - An object cannot be broken into pieces
    - Either choose it or not

Formal description: Fractional Knapsack

- Given W; w_i, v_i
- Find an array x_i, $1<=i<=n, 0<=x_i<=1$, to
  - Maximize $\sum_{i=1}^{n} x_i v_i$
  - And be subject to $\sum_{i=1}^{n} x_i w_i \le W$

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

fill the largest portion possible

Scheduling: an activity selection problem

- A set $S=\{a_1, a_2, \ldots, a_n\}$ activities wish to use a resource
  - The resource can be used by one activity at a time
  - Each activity has a start time s_i and a finish time f_i with $0 <= s_i < f_i$
  - If selected, activity take place at an half-open interval $[s_i, f_i)$.
  - Activities a_i, a_j are compatible if their intervals do not overlap: $s_i >= f_j \,||\, s_j >= f_i$
- The activity selection problem
  - Select a maximum-size subset of mutually compatible activities

The algorithm

```
GreedyActivitySelector(s, f) // s[] and f[] are start and finish times
                             // activities are sorted by finish time
{
    add a1 to A;

    lastSelected = 1;
    for (i=2; i<=n; i++) {
       if (s[i] >= f[lastSelected]) {
          add ai to A
          lastSelected = i;
       }
    }
    return A;
}
```

Summary: Greedy strategy

- Typical Steps
  - Cast the problem as one in which we make a choice and are left with one subproblem to solve
  - Proof of optimality if applicable:
    - Prove that there is always an optimal solution to the original problem, so that the greedy choice is always safe
      - Typically use "cut and paste"
    - Demonstrate that: an optimal solution to the subproblem combined with the greedy choice we have made is an optimal solution to the original problem

Kruskal's algorithm: cost

```
Kruskal(Graph G)  // G=<V, E>
{
  sort E by increasing weight;               <- O(Elog E)

  A = ϕ ;
  make n initial sets, each contains a node in V;   <- called makeSet V times

  for all sorted edges {
    e = <u,v>; // shortest edge not yet considered
    uComponent = find(u);                    <- called at most E times each
    vComponent = find(v);

    if (uComponent != vComponent) {
      Union(uComponent, vComponent);         <- called V-1 times
      A = A ∪ {e};
    }
  }
  return A;
}
```

Total:
$O(E\log E + E\alpha(V))$
$=O(E\log V)$

Prim's algorithm: an implementation

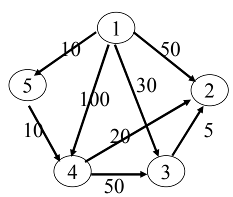
```
Prim(G, w, r)
{
   for each node u ∈ V {
      key[u] = ∞;
      π[u] = null;
   }
   key[r] = 0;
   Q.build(V); // Q is a priority queue use key[] as keys

    while (!Q.empty()) {
        u = Q.extractMin();
        for each v adjacent to u {
            if (v ∈Q && w(u,v) < key(v)) {
              Q.updateKey(v, w(u,v));
              π[v] = u;
            }
        }
    }
}
```

Dijkstra's algorithm

S : partial solution set
d[v]: length of the shortest special path for v.
π[v]: the previous node of v along its shortest (special) path.

```
Dijkstra(G, w, s)
{
   /* initialization */
   for each node v {
      d[v] = ∞;
      π[v] = null;
   }
   d[s] = 0;

   S = ∅;
   Q.build(V); // a priority Q use d[] as keys
   while (!Q.empty())
       u = Q.extractMin();
       S = S ∪ {u};
       for each v adjacent to u {
            if (v ∈Q && d[u] + w(u,v) < d(v)) {
              d[v] = d[u] + w(u,v);
              Q.decreaseKey(v, d(v));
              π[v] = u;
            }
       }
   }
}
```

Example



Step	u	S	d 1	2	3	4	5	π 1	2	3	4	5
Init	-	∅	**0**	∞	∞	∞	∞	-	-	-	-	-
1	1	{1}	0	50	30	100	**10**	-	1	1	1	1
2	5	{1, 5}	0	50	30	**20**	10	-	1	1	5	1
3	4	{1,5,4}	0	40	**30**	20	10	-	4	1	5	1
4	3	{1,5,4,3}	0	35	30	20	10	-	3	1	5	1
5	2	{1,5,4,3,2}	0	35	30	20	10	-	3	1	5	1

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
  - Selection and Find median

A general template

```
DC(x)
{
   if (x is sufficiently small or simple)
          adhoc(x);   // use a basic sub-algorithm

   decompose x into small instances x[0],..,x[l-1];  // divide

   for (i=0;  i<l; i++)                               //conquer
          s[i] = DC(x[i]);

   combine s[0],.., s[l-1] to obtain solution s for x;  // combine
   return s;
}
```

- Three conditions to be considered
  - When to use the basic sub-algorithm
  - Efficient decomposition and recombination
  - The sub-instances must be roughly the same size

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 for dividing and combining on instances of size n,
  - g(n) is the total time excluding the times need for the recursive calls.
  - We have *t(n) = l·t(n/b) + g(n)*
- If $g(n) \in \Theta(n^k)$ for an integer k, we have

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

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?

Merge sort

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

Cost

- Storage
- Execution time

$$t(n) = t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + \Theta(n)$$

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(A, p, r)
{
   if (p < r) {
      q = partition(A, p, r);
      quickSort(A, p, q-1);
      quickSort(A, q+1, r);
   }
}
```

Partition

```
int partition(A, p, r)
{
   x = A[r]; // use A[r] as the pivot
   i = p-1;
   for (j=p; j<=r-1; j++) {
      if (A[j] <= x) {
         i++;
         exchange(A[i], A[j]);
      }
   }
   exchange(A[i+1], A[r]);
   return i+1;
}
```

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

```
select(A[p..r], i) {
    if (p==r) return A[p];

    x = pseudomedian(A[p..r]);
    q = partition'(A[p..r], x);
    k = q-p+1;

    if (i==k)
        return A[q];
    if (i<k)
        return  select(A[p..q-1], i);
    if (i>k)
        return  select(A[q+1..r], i-k);
}
```

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

We assume the elements are distinct.

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
  - 0-1 Knapsack
  - Floyd's algorithm
  - Longest Common Sequence
  - Matrix chain

Example: Binomial Coefficient

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

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

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)

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 $\sum_{i=1}^{n} x_i v_i \quad subject \quad to \quad \sum_{i=1}^{n} x_i w_i \le W$,

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

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

Example

Weight limit	0	1	2	3	4	5	6	7	8	9	10	11
w_1 =1 v_1=1	0	1	1	1	1	1	1	1	1	1	1	1
w_2 =2 v_2=6	0	1	6	7	7	7	7	7	7	7	7	7
w_3 =5 v_3=18	0	1	6	7	7	18	19	24	25	25	25	25
w_4 =6 v_4=22	0	1	6	7	7	18	22	24	28	29	29	40
w_5 =7 v_5=28	0	1	6	7	7	18	22	28	29	34	25	40

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;
      i = i-1;
      k = k-w[i];
  } else
      i = i-1;
}
```

Cost: $O(n+W)$

The Matrix Chain Multiplication Problem

- Given a chain of $<M_1, M_2, \ldots, M_n>$ of matrices, where for $i = 1, 2, \ldots, n$, matrix M_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product in a way that $M_1 M_2 \ldots M_n$ minimizes the number of scalar multiplications

A recursive equation (optimal substructure)

- 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
    $m[i,j] = m[i,k]+m[k+1,j]+p_{i-1} p_k p_j$
  - We obtain the following recurrence

$$m[i,j] = \begin{cases} 0, & i = j \\ \min_{i \le k < j} (m[i,k] + m[k+1,j] + p_{i-1}\, p_k\, p_j), & i < j \end{cases}$$

An implementation using dynamic programming

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

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

Cost:

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

Example

M1	13×5
M2	5×89
M3	89×3
M4	3×34

i \ j	1	2	3	4
1	0	5785	1530_1	2856_3
2		0	1335	1845_3
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

Construct the optimal parenthesization

- In our algorithm, the matrix *s* tracks the split point
  - Can you use the matrix to construct the optimal parenthesization?

```
PrintOptimalParens(s, i, j)
{
    if (i==j)
        print ("M"_i)
    else {
        print("(");
        PrintOptimalParens(s, i, s[i][j]);
        PrintOptimalParens(s, s[i][j]+1, j);
        Print(")")
    }
}
```

Characterizing an LCS

- Let $X=\langle x_1,x_2,\dots,x_m\rangle$ and $Y=\langle y_1,y_2,\dots,y_n\rangle$ be sequences and let $Z=\langle z_1,z_2,\dots,z_k\rangle$ be any LCS of X and Y
  1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS X_{m-1} and Y_{n-1}
  2. If $x_m \neq y_n$, and $z_k \neq x_m$, then Z is an LCS X_{m-1} and Y
  3. If $x_m \neq y_n$, and $z_k \neq y_n$, then Z is an LCS X and Y_{n-1}

The optimal substructure of LCS

- Let $c[i,j]$ be the length of an LCS of the sequences X_i and Y_j, we obtain the following recurrence

$$c[i,j] = \begin{cases} 0 & \text{if} \quad i = 0 \,\|\, j = 0 \\ c[i-1, j-1] + 1 & \text{if} \quad i, j > 0 \,\&\&\, x_i = y_j \\ \max(c[i, j-1], c[i-1, j]) & \text{if} \quad i, j > 0 \,\&\&\, x_i \neq y_j \end{cases}$$

An implementation using dynamic programming

```
LCS(X, Y)
{
    for (i=1; i<=m; i++)  c[i][0]=0;
    for (j=1; j<=n; j++)  c[0][j] =0;

    for (i=1; i<m; i++)
      for (j=1; j<n; j++) {
          if (x[i] == y[j]) {
              c[i][j] = c[i-1][j-1] +1;
              b[i][j] = '↖'
          } else if (c[i-1][j] >= c[i][j-1]) {
              c[i][j] = c[i-1][j];
              b[i][j] = '↑'
          } else {
              c[i][j] = c[i][j-1];
              b[i][j] = '←'
          }
      }
}
```

Example

		0	1	2	3	4	5	6
		y	B	D	C	A	B	A
0	x	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Construct an LCS

- In our algorithm, the direction array b tracks the construction

```
PrintLCS(b, X, i, j)
{
    if (i==0 || j==0)
        return;
    switch (b[i][j]) {
        case '↖':
              PrintLCS(b, X, i-1, j-1)
              print x[i];
              break;
        case '↑':
              PrintLCS(b, X, i-1, j)
              break;
        case '←':
              PrintLCS(b, X, i, j-1)
              break;
    }
}
```

Floyd's algorithm

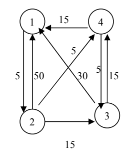
```
Floyd(int W[1..n][1..n])
{
   D = W;
   π = 0;
   for (k=1; k<=n; k++)
      for (i=1; i<=n; i++)
         for (j=1; j<=n; j++)
            if (D[i][k]+D[k][j] < D[i][j]) {
               D[i][j] = D[i][k]+D[k][j];
               π[i][j] = k;
            }
}
```

Track the shortest path

Cost: $\Theta(V^3)$

Why can we use just one array D?

Example



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

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

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

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.

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

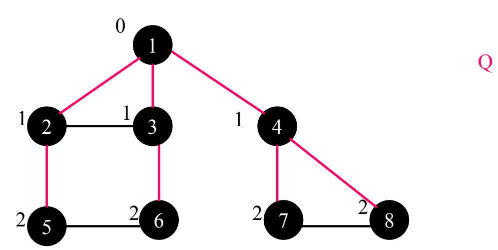
Breadth-first search algorithm

```
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

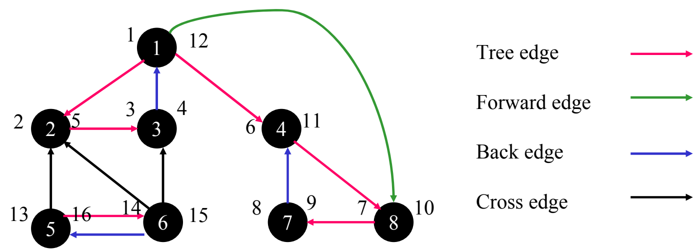
```
DFS(G)
{
   for each node u ∈ N {
     color[u] = WHITE;
     π[u] = null;
   }

   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

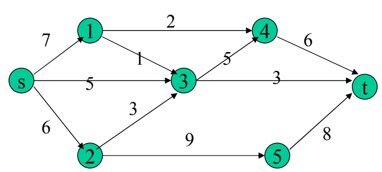


Topological Sort

- 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

Maximum Flow Problem

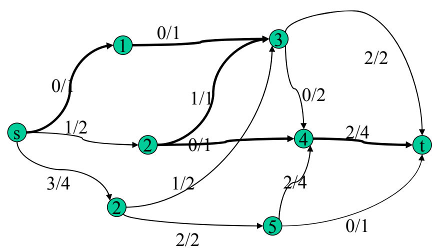
- Given a weighted directed graph
  - Each edge is a pipe whose weight denotes its capacity: the maximum amount it can transport
    - Use *c(e)* for the capacity of edge *e*
  - Given a source, *s*, and a sink, *t*, find the maximum amount (flow) can transfer from *s* to *t*



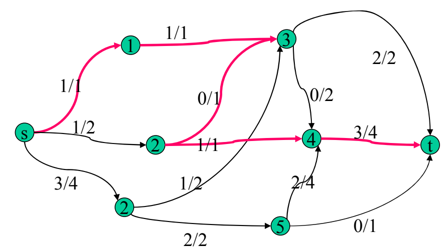
Concepts

- You need to know the follow concepts
  - Flow of a network
  - Capacity and flow of an edge
  - Cut
    - Capacity of a cut
    - Flow of 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 Δ_f(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

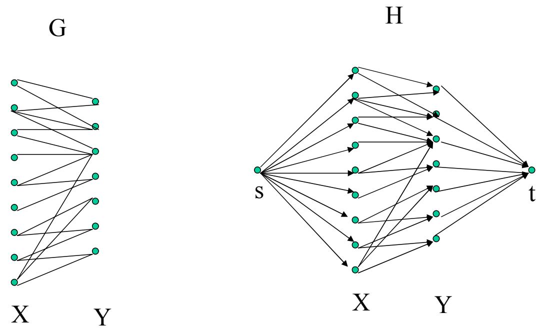
Maximum Bipartite Matching

- 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

Good Luck

Take it easy

Merry Christmas