• **Homework 4**
  • Due Saturday at 2am (so, basically Friday night)
• **Midterm: Tuesday, March 10, 2015**
  • Practice midterm online
  • Will only cover up to and including dynamic programming
    • I.e., this lecture will not be in the midterm
• **Office hours**
• Recall how to represent graphs
  • Adjacency list
    \[
    \text{ArrayList}\langle\text{Integer}\rangle \text{ adj}[] = \textbf{new} \ \text{ArrayList}\langle\text{Integer}\rangle[n];
    \]
    \[
    \textbf{for} \ (\text{int} \ i = 0; \ i < \text{adj}.\text{length}; \ ++i)
    \]
    \[
    \text{adj}[i] = \textbf{new} \ \text{ArrayList}\langle\text{Integer}\rangle();
    \]
  • Adjacency matrix
    \[
    \textbf{boolean} [][] \text{ adj} = \textbf{new} \ \text{boolean}[n][n];
    \]
  • Edge list
• Directed? Undirected?
• Graph $G = (V, E)$, vertices $V = \{ v_0, v_1, \ldots \}$, $E = \{ (v_0, v_2), (v_1, v_2), \ldots \}$
• Path
  There is a path of length $n$ between vertices $u$ and $v$ if there is a sequence of vertices $x_0, x_1, \ldots, x_n$ between them such that there is an edge between $x_i$ and $x_{i+1}$
• Connected
  In an undirected graph, two vertices are connected if there exists a path between them. An undirected graph is connected if every pair of vertices is connected.
• Strongly connected
  A directed graph is strongly connected if there exists a directed path in both directions between each pair of vertices.
• **Connected component**
  • Maximal connected subgraph

• **Cycle**
  • A path that begins and ends at the same vertex
**Exercise**: A forest is an undirected acyclic graph. If a forest with $n$ vertices has $C$ components, how many edges will it have?

**Exercise**: What is the maximum number of edges in a directed graph with $n$ vertices that has no loops (edges of the form $(v, v)$)?

**Exercise**: A walk is like a path, except the vertices need not be distinct. How would you compute, for each pair of vertices $(i, j)$, whether there is a walk of length $k$ from $i$ to $j$?
Depth first search
Visit every node in a graph by recursion
  With a stack
Visit the furthest nodes from the originating node
Perform backtracking
Depth First Search

Adjacency list:
0: 1, 2
1: 1, 3
2: 0, 3
3: 1, 2, 4, 5
4: 3
5: 3

Stack:
dfs(0)
dfs(1)
dfs(3)
static int INIT = 0, PROCESSING = 1, FINISHED = 2;

//DFS starting a given vertex vert
void dfs(ArrayList<Integer>[] adj, int vert, int[] state)
{
    if (state[vert] != INIT) return;
    state[vert] = PROCESSING;
    for (int i = 0; i < adj[vert].size(); ++i)
        dfs(adj, adj[vert].get(i), state);
    state[vert] = FINISHED;
}

void dfs(ArrayList<Integer>[] adj)
{
    int[] state = new int[adj.length];
    for (int i = 0; i < state.length; ++i) if (state[i] == INIT)
        dfs(adj, i, state);
}
• Visit nodes closest to the originating node first
• Implement with a queue
Breadth First Search

Adjacency list
0: 1, 2
1: 1, 3
2: 0, 3
3: 1, 2, 4, 5
4: 3
5: 3

Queue
1
2
/BFS starting at a given vertex vert
void bfs(ArrayList<Integer>[] adj, int vert, int[] state)
{
    Queue<Integer> queue = new ArrayDeque<Integer>();
    queue.add(vert);
    while (!queue.isEmpty())
    {
        int v = queue.poll();
        for (int i = 0; i < adj[v].size(); ++i)
        {
            int w = adj[v].get(i);
            if (state[w] == INIT)
            {
                queue.add(w);
                state[w] = PROCESSING;
            }
        }
        state[v] = FINISHED;
    }
}
void bfs(ArrayList<Integer>[] adj)
{
    int[] state = new int[adj.length];
    for (int i = 0; i < state.length; ++i) if (state[i] == INIT)
        bfs(adj[i], state);
}
• Both algorithms will only process each vertex once
  • O(V+E) running time for an adjacency list
  • Each determine a forest in a graph
    • An edge participating in the forest is a **tree edge**
    • An edge from a vertex to one of its ancestors is a **back edge**
    • An edge from a vertex to one of its descendants is a **forward edge**
    • An edge that is neither a back edge nor a forward edge is a **cross edge**
      o Only possible in directed graphs
  • BFS can be used as a shortest-path algorithm in an unweighted graph
**Exercise:** When performing a DFS (on undirected or directed), when a node v has just begun processing, what can be said of the other nodes that are processing?

**Exercise:** Give an algorithm to determine if a directed graph is acyclic.
Exercise: Show how to compute the number components in a graph. Give the runtime of your algorithm for an adjacency list and an adjacency matrix.

Exercise: In an undirected graph, a bridge is an edge whose removal increases the number of connected components. Give a simple algorithm for finding all bridges of a graph.
Exercise: You are on an $n \times n$ grid located at the lower left corner $(0, 0)$. Each turn you can take one step horizontally or vertically. You are also given a list of obstacles $L$ which you must avoid. Determine the length of the shortest path from $(0, 0)$ to $(n - 1, n - 1)$.

Exercise: A graph is called 2-colorable (or bipartite) if each vertex can be assigned the color red or blue such that adjacent vertices have different colors. Give an algorithm to determine if a graph is bipartite.
Suppose you are given the following grid to search:

```java
String mapStr[] = {
    "...........E",
    ".xxxxxxxxxx",
    "...........",
    "xxxxxxxxxx",
    "xxxxxxxxxx.",
    "S..........."
};
```

- **Two common grid traversal tricks:**
  - Pad the edges with invalid characters
    - Saves you from having to handle boundary conditions
  - Model the directions as diffs in directions.
    - E.g., model the four cardinal directions as \{ (-1, 0), (0, 1), (1, 0), (0, -1) \}
Exercise: When you get up in the morning and get dressed, you need to put on certain garments before others (e.g., underwear before pants). You can also only put on one garment at a time. How would you determine the order in which to get dressed?
• **Definition**: a topological sort of a DAG is a linear ordering of the vertices in the DAG so that vertex u comes before vertex v if if edge (u, v) exists in the DAG.
  • Every DAG has one or more topological orderings
• Implementation with DFS, BFS (Kahn’s algorithm)
• DFS approach to topological sort:
call DFS on each unvisited vertex,
as it finishes, add it to the front of
a linked list
  • Generating topological order in reverse
  • Recall: once DFS finishes a vertex, it has
    processed all vertices reachable from that
    vertex

```java
void dfs(ArrayList<Integer>[] adj, int vert, int[] state, 
        ArrayList<Integer> list)
{
    if (state[vert] != INIT) return;
    state[vert] = PROCESSING;
    for (int i = 0; i < adj[vert].size(); ++i)
        dfs(adj, adj[vert].get(i), state);
    state[vert] = FINISHED;
    list.add(vert);
}

ArrayList<Integer> toposort(ArrayList<Integer>[] adj)
{
    ArrayList<Integer> list = new ArrayList<Integer>();
    int[] state = new int[adj.length];
    for (int i = 0; i < state.length; ++i) if (state[i] == INIT)
        dfs(adj, i, state, list);
    Collections.reverse(list);
    return list;
}
```
Kahn’s algorithm:
1. Enqueue each vertex with zero incoming edges into priority queue Q
2. while Q is not empty:
   1. u = Q.dequeue
   2. L.append(u)
   3. remove v and all outgoing edges from graph
   4. for each neighbor u of v:
      1. if u now has 0 incoming edges: Q.enqueue(u)

Reference implementation
Exercise: what if the input graph to Kahn’s algorithm has cycles?
Competitive Programming 4.1, 4.2