1 Topics Review

1.1 Math Fundamentals

- **Recurrence Relations:**
  - Solve simple ones by writing out some terms and finding a pattern
  - Try to use Master's Theorem, or make a substitution first and then use it
  - Draw out a recursion tree, determine the number of levels and amount of work done at each layer.
- **Big-O Notation:** You should be very comfortable with the definitions for O, o, Ω, ω, and Θ.

1.2 Data Structures

- Arrays, Linked Lists, Stacks, and Queues.
- Heaps and Priority Queues
- Representations of a Graph – Adjacency Matrix and Adjacency List

1.3 Algorithms

You should be able to state how each of the following algorithms work and state its run-time.

- **Multiplication**. Karatsuba's Algorithm, Matrix Multiplication, Repeated Squaring
- **Mergesort**. Divide and Conquer, merging sorted sublists together.
- **Depth First Search**. Key idea is to use a stack to enumerate nodes. Assign **preorder** and **postorder** when pushing and popping nodes from the stack. Know what **tree edges**, **forward edges**, **back edges**, and **cross edges** are.
  - **Topological Sort**. Useful for ordering nodes in a Directed Acyclic Graph
  - **Strongly Connected Components**. Turn any graph into a DAG of SCCs.
- **Breadth First Search**. Key idea is to use a queue to store unprocessed nodes. Gives shortest paths in an unweighted graph.
- **Dijkstra's Algorithm** and **heaps**. Use DFS with a priority queue. Only works for positive edge weights.
- **Bellman Ford**. Recursive, takes advantage of the optimality of subpaths. Works on arbitrary graphs. Bonus: finds negative cycles!
- **Floyd-Warshall**. Finds all pairs shortest paths

- **Prim's Algorithm**. Build a tree from a single seed vertex. Uses the **Cut Property** to choose the smallest edge leaving the group.
- **Kruskal's Algorithm**. Build a tree by sorting the edges, joining disjoint groups of vertices. Uses the **Disjoint Set** data structure.

1.4 Algorithm Strategies

- **Greedy**. Pick the best option at each step. e.g., Prim's Algorithm or Kruskal's Algorithm
- **Divide and Conquer**. Split the problem into smaller subproblems (often in half), solve the subproblems, and combine the results. e.g., Mergesort, Integer Multiplication, Strassen's
- **Modify the Problem**. Build a graph from your problem or modify the given graph such that running an algorithm we talked about in class gives you the correct solution instantly.
- **Modify the Algorithm**. Make a slight modification to one an algorithm discussed in class to fit a particular problem.

2 Practice Problems

2.1 Asymptotic Notation

Exercise. *Give a counterexample to the following statement. Then propose an easy fix.*

If $f(n)$ and $g(n)$ are positive functions, then $f(n) + g(n) = \Theta(\min(f(n), g(n))$

Solution.
This is false. Let $f(n) = n$, $g(n) = 1$. Then, $\min(f(n), g(n)) = 1$ but $f(n) + g(n) = n + 1$ is not $\Theta(\min(f(n), g(n))) = \Theta(1)$.

However, it is true that $f(n) + g(n) = \Theta(\max(f(n), g(n)))$. To prove this, note that:

$$\max(f(n), g(n)) \leq f(n) + g(n) \leq 2\max(f(n), g(n)).$$

2.2 Number of Paths in a DAG

Exercise. *Given a directed acyclic graph $G = (V, E)$ and two nodes $u, v \in V$, calculate the number of distinct paths from u to v. (Two paths are considered distinct if they have at least 1 vertex not in common)*

Solution.
Since G is a DAG, we can perform a topological sort of its vertices in $O(n + m)$ time using DFS. Suppose our topological sort comes out to be $u, w_1, w_2, \ldots w_k, v$. Note that if v appears before u, then we should return 0 immediately since there would be no way to get from u to v.

Now, we define a recursive function $f(i)$ to be the number of distinct paths from u to w_i and let $u = w_0$. We are interested in the value $f(v)$.

Base Case: if $i = 0$, then $f(i) = 1$. There is only 1 way to get from u to u, which is to stay at u. **Recursive Case:** if $i \neq 0$, then

$$f(i) = \sum_{w_j:(w_j,w_i)\in E} f(j)$$

The number of ways to get to w_i is the sum of the number of ways to get to each of w_i's predecessors that have an edge connecting to w_i. Note that $0 \leq j < i$ and since the graph is topologically sorted, we guarantee that if there's an edge $w_j \to w_i$, then $j < i$.

Run-Time: First, the topological sort takes $O(n + m)$ time by DFS. Next, there can be up to n values of $f(i)$ to compute, and for each vertex w_j, the it takes $O(1) + O(\text{indegree}(v))$ time to compute $f(w_j)$ so the total run-time is $O(n+m)$. The sum of the indegrees of each vertex in a directed graph is the number of edges in the graph. Note that we can pre-compute G^R in order to be able to access indegrees rather than just outdegrees in $O(1)$ time.

2.3 Semiconnectedness

Exercise. *A directed graph $G = (V, E)$ is called semiconnected if for each pair of distinct vertices $u, v \in V$, there is either a path from u to v or a path from v to u. Find an algorithm to determine whether a directed graph is semiconnected.*

Solution.
Let G' be the DAG on the strongly connected components of G. Number the nodes of G' in some topological order. We claim that G is semiconnected iff there is always a path from the i-th node to the j-th node of G' if $i < j$.

Suppose that there is always a path from the i-th node to the j-th node of G' if $i < j$. Then, for any two vertices u and v in G, they either belong to the same SCC (in which case there is a path from u to v and a path from v to u), or they belong to two different SCC's, c_u and c_v. Without loss of generality, suppose c_u occurs earlier than c_v with respect to the topological order. Then, by hypothesis, there is a path from c_u to c_v, so there is a path from u to v as desired.

Now suppose that for some topological sort of G', there is no path from the i-th node (with respect to this sort) to the j-th node, where $i < j$. Then, there can be no path from the j-th node to the i-th node since $i < j$ and the nodes are topologically sorted. Hence, if u is a vertex lying in the i-th SCC and v a vertex lying in the j-th SCC, there are no paths from u to v or from v to u, so G is not semiconnected.

This observation allows us to devise an algorithm to determine if G is semiconnected: we first compute the strongly connected components of G and construct the graph G' in time $O(n+m)$, where $n = |V|$ and $m = |E|$. Then, using DFS from any vertex in G', we can topologically sort G' in $O(n + m)$ time. Now, we check if there is a path from the i-th SCC to the j-th SCC if $i < j$. Note that this is the case iff there is an edge in G' from the i-th SCC to the $i + 1$-th SCC. Hence, we can just scan through the topological sort in $O(n)$ time to see if the i-th SCC has an edge to the $i + 1$-th SCC. The total running time of the algorithm is $O(n + m)$.

2.4 Negative Cycles

Exercise. *Modify Bellman Ford to set dist[v] $= -\infty$ for all vertices v that can be reached from the source via a negative cycle.*

Solution.
Normally when we check to see whether a graph has a negative cycle, we check to see that $f(n, v) = f(n-1, v)$ for all $v \in V$. Since the graph has n vertices, any path that uses n edges must have touched some vertex twice because a path using n distinct vertices would be $n+1$ edges long. Therefore, the path given by $f(n, v)$ could potentially have taken a cycle. If $f(n, v) < f(n-1, v)$ for any $v \in V$, then we can say "Negative Cycle Detected".

It is important to see that if a negative cycle does exist in a graph, the condition that $f(n, v) = f(n-1, v)$ will not be broken for *all* vertices that can be reached from the source via a negative cycle. For some vertices that are reachable by a negative cycle, n vertices are not enough to have traveled down a negative cycle.

To solve this problem, we compute $f(2n, v)$ for all $v \in V$. In up to $2n$ vertices, we can get from the source to any vertex v as well as have enough edges to travel along any negative cycle in the graph. If there is a negative cycle from the source to v, then the path itself without the negative cycle has up to n vertices, while the negative cycle has up to n vertices as well. Any negative cycle of more than n vertices can be decomposed into 2 smaller cycles, at least 1 of which is a negative cycle.

The run-time of this solution is still $O(nm)$ because we are simply doing $2nm$ work.

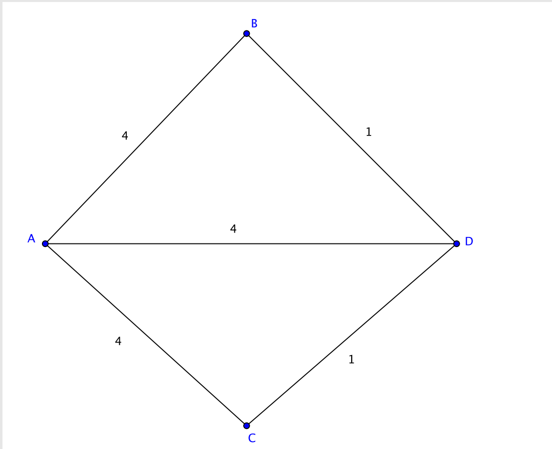
2.5 Minimum Spanning Tree True or False

Exercise. *True or False*

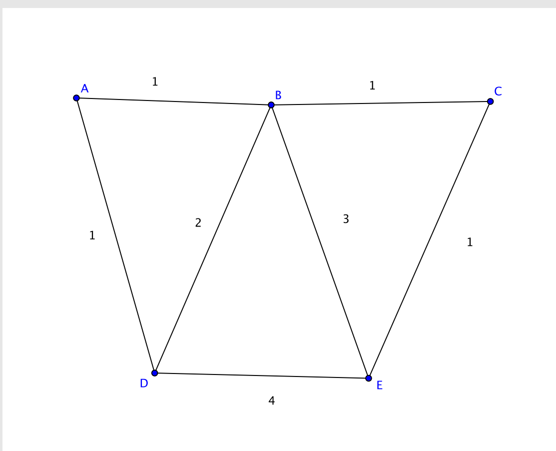
- *If G has a cycle with a unique heaviest edge e, then e cannot be part of any MST*
- *The shortest-path tree computed by Dijkstra's algorithm is necessarily an MST.*
- *Prim's algorithm works with negative weighted edges.*
- *If G has a cycle with a unique lightest edge e, then e must be part of every MST.*

Solution.

- True. Suppose e was part of some spanning tree. If we remove e, we get two sets S (containing e) and $V - S$ with no edges in the MST going from S to $V - S$. Now, some other edge e' in the cycle must go from S to $V - S$, and replacing e with e', we get another spanning tree. The total weight of this spanning tree is less than the weight of the former spanning tree since $w(e') < w(e)$, so the former spanning tree could not be minimal.
- False. For example, consider the following graph. The shortest path tree from A includes edges (A, B), (A, D), and (A, C), but a minimal spanning tree(for example, (D, B), (D, C), and (D, A)) has total weight 6.



- True. The proof that Prim's algorithm is correct (the cut property) does not require edge weights to be negative. Another way to see this is that if we add a constant to all edges in the graph, the minimum spanning tree remains the same (since all spanning trees have total weight increased by the same amount), so adding a large enough constant, all edge weights are positive.
- False. Consider the following graph. The cycle BDE has unique smallest edge (B, D), but is not in the spanning tree with edges (A, D), (A, B), (B, C), (C, E).



2.6 Minimum Spanning Tree with Few Edges

Exercise. *Given a weighted graph with n vertices and $m \leq n+10$ edges, show how to compute a minimum spanning tree in $O(n)$ time.*

Solution.
First check if G is connected with a DFS ($O(m+n) = O(n)$ time). If G is not connected, there exists no MST, so return impossible.

Now suppose G is connected. Initially let the set T consist of all m edges. Let $c = m - (n-1)$, i.e. let there be c too many edges in T. By the condition $m \leq n + 10$ we know that $c \leq 11$. Run the following procedure c times:

DFS from an arbitrary vertex, keeping a precedessor array for the vertices. There exists some cycle, and thus there will be some back edge (u, v). We know that u, v is a part of a cycle, so tracing back the precedessor array from u, we eventually reach v. Thus, we have detected a cycle in $O(n)$ time. Let $e = (u, v)$ be some edge in this cycle C with maximal weight (if there are ties choose e aribtrarily). Suppose there exists any tree T with edge e. We will show that there exists another tree T' without e with total weight less than or equal to T. Indeed, let $S \subset V$ be the set of vertices $s \in V$ such that in $T \setminus \{e\}$, s is connected to u (and hence not v). Then let the vertices of C be in the order $u, x_1, \ldots, x_k, v$. Then some edge $e' = (x_i, x_{i+1})$ connects S to $V \setminus S$ and by the maximality of e we have $w(e) \geq w(e')$. Thus we can add e', which creates a cycle

$$u \to v \to x_i \to x_{i+1} \to u$$

which can be broken by removing e. Since $w(e') \leq w(e)$ we have a new tree with total weight less than T.

Thus, there is an MST without e, so we can remove e and repeat the process on the new graph $G \setminus \{e\}$. After c repetitions, we will have a graph with $n-1$ edges, and hence a tree. This will be an MST because we know that after each edge we remove, there exists an MST among the remaining edges (by the argument above).

This runs in $O(11(m+n)) = O(m+n) = O(n)$ time.