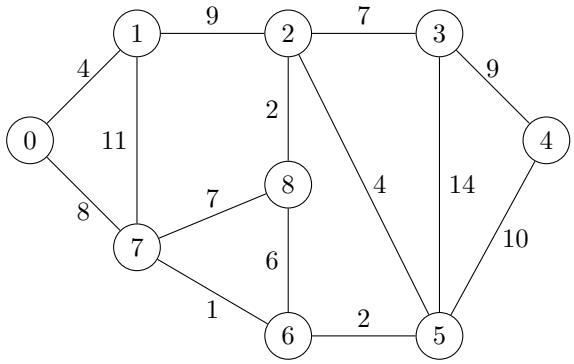
Recitation Problems - Com S 311

Week of Apr 29^{th} - May 4^{th}

1. Consider the following graph:

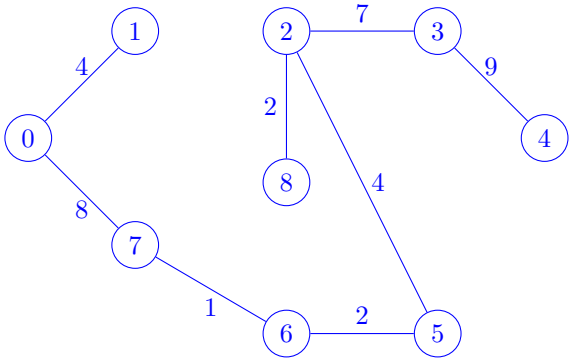


(a) Starting at vertex 0, apply Prim's algorithm and show the order in which edges are added to the Minimum Spanning Tree. Draw the partial tree after each edge is added.

Edges in Prim's algorithm added in the following order:

- (0, 1)
- (0, 7)
- (7, 6)
- (6, 5)
- (5, 2)
- (2, 8)
- (2, 3)
- (3, 4)

And the total cost of the minimum spanning tree is 37. The partial tree is as such:



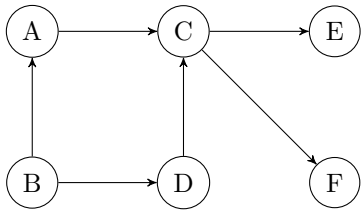
(b) What is the distance between vertex 0 and vertex 4 in your minimum spanning tree?

Distance between vertex 0 and vertex 4 in MST is 31.

(c) Is the distance computed in part (b) the shortest distance from vertex 0 to vertex 4? If not, what is the shortest distance between vertex 0 and vertex 4?

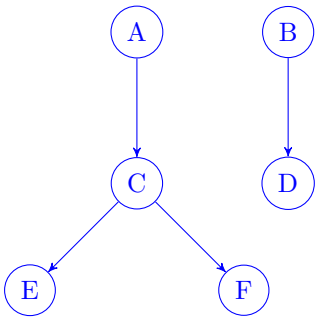
Shortest distance from vertex 0 to vertex 4 is not 31, but 21 via this sequence of edges: $(0, 7) \rightarrow (7, 6) \rightarrow (6, 5) \rightarrow (5, 4)$

2. Consider the following graph:

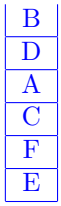


(a) Does this graph have a valid topological ordering? Why?
Yes, it does, because it is a directed, acyclic graph (DAG).

(b) Do a DFS starting at vertex A to produce a topological ordering. Whenever you have a choice of vertices to explore, always pick the one that is alphabetically first. Draw a picture of the stack produced during the DFS.
The topological order is: B, D, A, C, F, E. The DFS trees are as such:



The stack produced is as such:



(c) Give another topological ordering of the above graph, different from the one in part (b).
Another topological order: B, A, D, C, E, F

3. Answer the following True/False questions?

(a) Binary search in a sorted array is worst case $O(\log n)$. True

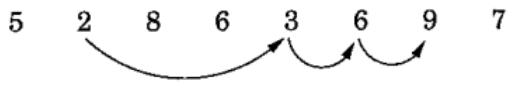
(b) The minimum spanning tree can be used to find the minimum cost path between any two vertices. False

(c) In a weighted graph, the minimum spanning tree is always unique. False

(d) A decision problem X is NP-Complete if $X \in$ NP and for all problems $y \in$ NP, y is polynomial time reducible to X. True

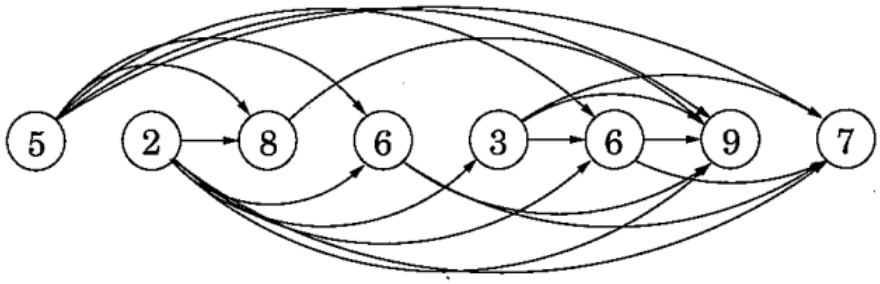
(e) Dynamic Programming algorithms take advantage of overlapping subproblems by solving each subproblem once and storing the solution in a dictionary (or table) for O(1) constant time lookup. True

4. In the longest increasing subsequence problem, the input is a sequence of numbers $a_1, ..., a_n$. A subsequence is any subset of these numbers taken in order, of the form $a_{i_1}, a_{i_2}, ..., a_{i_k}$, where $1 \leq i_1 < i_2 < ... < i_k \leq n$, and an increasing subsequence is one in which the numbers are getting strictly larger. The task is to find the increasing subsequence of greatest length. For instance, the longest increasing subsequence of $5, 2, 8, 6, 3, 6, 9, 7$ is $2, 3, 6, 9$.



Write an algorithm to find the increasing subsequence of greatest length given a sequence of numbers a with a length of n. Analyze your algorithm's runtime.

Let's create a graph of all permissible transitions: establish a node i for each element a_i, and add directed edges (i, j) whenever it is possible for a_i and a_j to be consecutive elements in an increasing subsequence, that is, whenever $i < j$ and $a_i < a_j$. Using the example above, the graph will look something like this:



Notice that the created graph $G = (V, E)$ is a directed acyclic graph (DAG), since all edges (i, j) have $i < j$, and there is a one-to-one correspondence between increasing subsequences and paths in this DAG. Therefore, this problem reduces to simply finding the longest path in the DAG.

```
for j = 1, 2, ..., n:
    L(j) = 1 + max{L(i) for each (i, j) in E}
return max{L(j) for j = 1, 2, ..., n}
```

Here, $L(j)$ is the length of the longest path - the longest increasing subsequence - ending at j (plus 1, since strictly speaking we need to count nodes on the path, not edges). We see that any path to node j must pass through one of its predecessors, and therefore $L(j)$ is 1 plus the maximum $L(\cdot)$ value of these predecessors. If there are no edges into j, we take the maximum over the empty set, zero. And, the final answer is the largest $L(j)$, since any ending position is allowed.

However, the above snippet of code only tells us the length of the longest increasing subsequence, if we want to recover the sequence itself, while computing $L(j)$, we should also note down $prev(j)$, the next-to-last node on the longest path to j. We need to modify our algorithm as such:

```
for j = 1, 2, ..., n:
    max_path = 0
    prev(j) = null
    for each (i, j) in E:
        if (L(i) > max_path):
            max_path = L(i)
            prev(j) = i
    L(j) = 1 + max_path

Let k be the node with maximum L-value
seq = ""
while k != null:
    seq = seq + k
    k = prev(k)
return seq
```

The runtime for this algorithm is dominated by the construction of the graph, which in the worst case takes $O(n^2)$ time.