Q1

Given a maximum heap H, what is the time complexity of finding the 3 largest keys in H?

By using extract-max 3 times we can get the 3 largest items in the heap.
Calling *heapify* on the root each time will take at most $lg(n)$ steps thus the complexity is $3lg(n) = \theta(lg(n))$ of getting the 3 largest items in the heap.

What is the time complexity of finding the ith largest key in H?

Following the formula used in the 1st question, we can use extract-max *i* amount of times to get the *ith* largest element, thus the complexity is $O(ilg(n))$

Explain if you can improve the time complexity of finding ith largest key in H.

Use a max PQ and insert the largest element in H into it, then repeat extract-max on the PQ and insert the children of the extracted element from the original heap H. After repeating *i-1* number of times, the next extract-max on PQ will be the *ith* largest element. Extracting and inserting to the PQ takes $lg(i)$ time as there will never be more than *i* number of keys. Thus the complexity is $(i-1)(2lg(i) + lg(i)) = \theta(ilg(i))$

Q2

Given a graph G with n vertices such that for every $v \in V$, $\boldsymbol{deg}(v) \geq n$ / 2 , then G is one connected component.

In other words, every vertex is connected to at least half of the vertices in the graph.

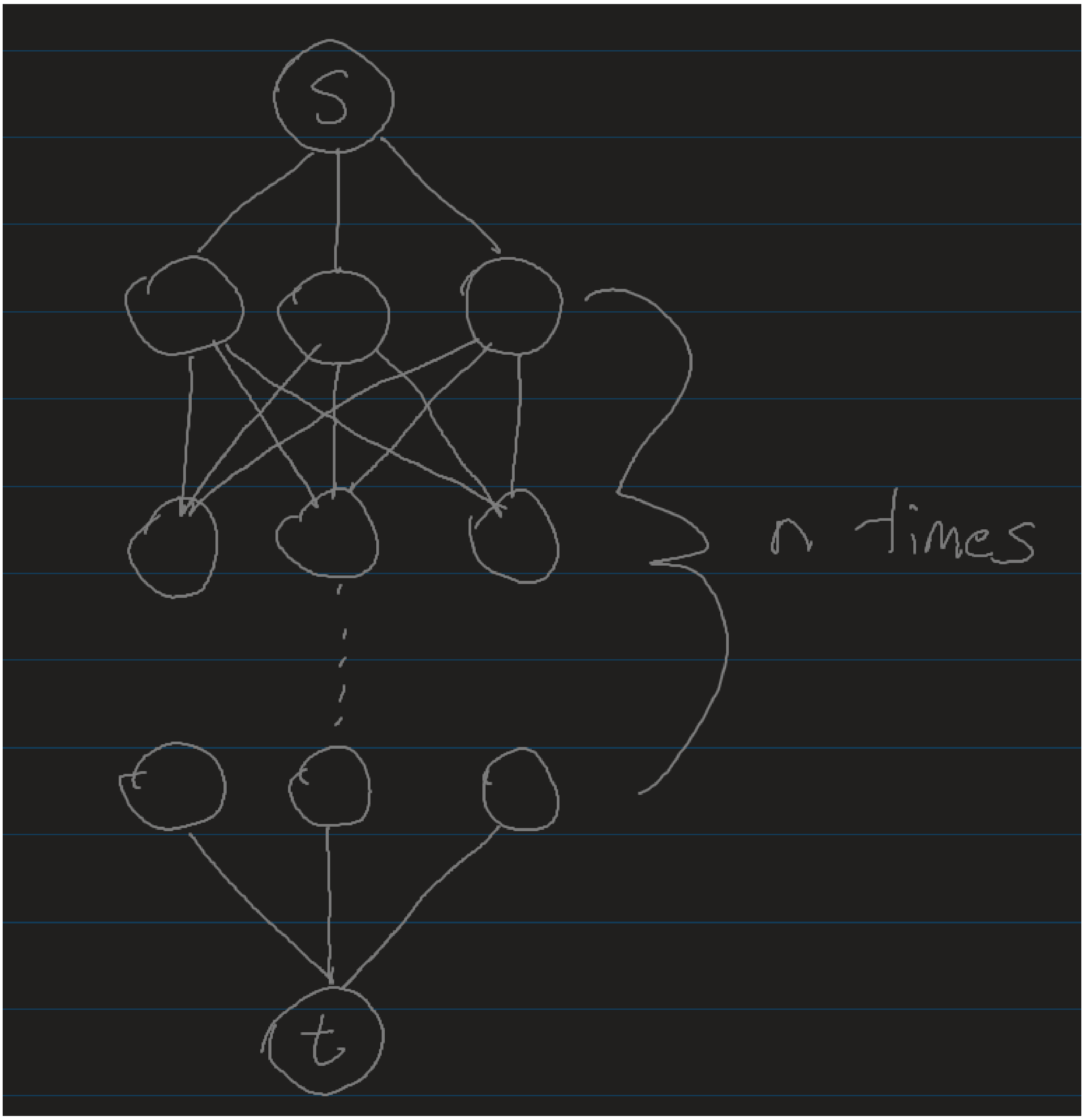
Suppose there exists a node $u \in V$ that is part of connected component *C* that is not connected to at least 1 node of G. The size of C must be at least $\frac{n}{2} + 1$ because every node must be connected to at least half the nodes in the graph + itself.

Now consider node $w \notin C$, we know $deg(w) \geq \frac{n}{2}$ by definition of G, however, because the size of C is $\frac{n}{2} + 1$, there only exists $n - \frac{n}{2} + 1 = \frac{n}{2} - 1$ nodes not in the connected component C. Therefore there must exist a node connected to w that is in C. Thus by contradiction the graph G is one connected component.

Q3

For every natural number n, there is an undirected graph of cn + k vertices such that for some pair of vertices s and t in the graph, there are 3^n shortest paths from s to t.

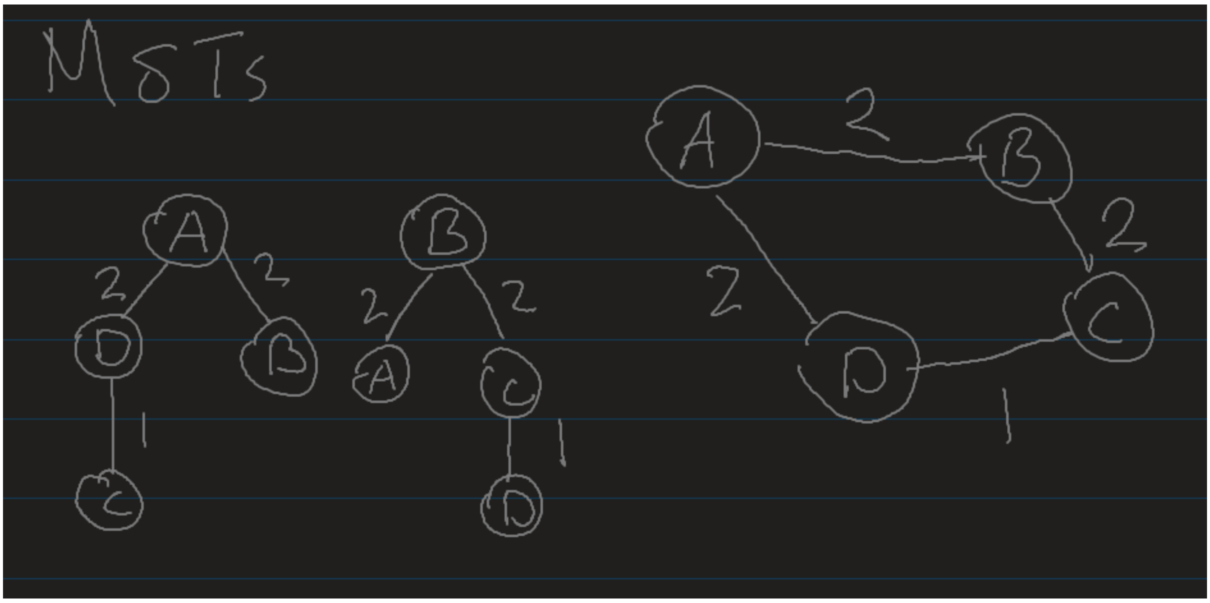
Let c=3, k=2, and set up the graph like the figure below with each edge weight being 1. It's easy to see that the graph below has exactly 3n + 2 nodes.



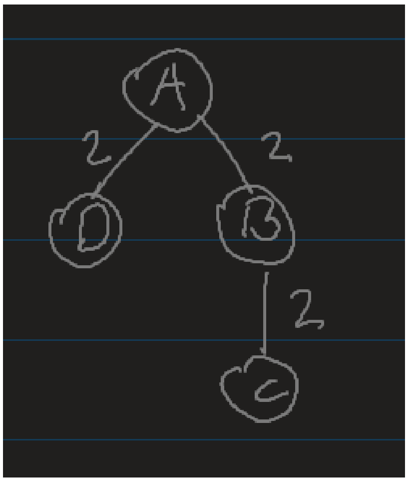
Notice that for each row of 3 nodes, the number of paths to the next node multiplies by 3 because each node in every row is connected to each node in the next row. Since there are n rows the total number of paths starting from s and ending at t is exactly 3^n. Because every edge is weighted the same and each edge always traverses to the next row, the weight of each path from s to t is the same.

Q4

Consider an undirected graph G = (V, E) with non-distinct, non-negative edge weights. If the edge weights are not distinct, it is possible to have more than one MST. Suppose we have a spanning tree $T \subset E$ with the guarantee that for every $e \in T$, e belongs to some minimum-cost spanning tree in G. Can we conclude that T itself must be a minimum-cost spanning tree in G?



Take G = the graph shown on the right side of the figure above. Notice on the left in the figure, we can make 2 distinct MSTs, and within those MSTs you can find every edge in the graph.



Take the spanning tree shown in the figure to the left. Every node in G is reached in this tree, so it is easy to see that it is a spanning tree. However, the weight of this spanning tree is larger than the weight of the previously shown MSTs, thus there exists a spanning tree with all edges that exist in MSTs, but is not itself an MST.

Q5

a) Give an algorithm to determine whether a directed graph has a cycle. What should your complexity be?

Using a DFS, if any backedges exist, then there is a cycle. The complexity should be the same as DFS, O(n + m) or O(|V| + |E|).

Pseudocode taking from class slides:

```
time = 0
start = any vertex
dfs(start)
dfs(v)
     v.state = visited
     v.start = time
     time += 1
     for each neighbour w of v
          if w.state == visited
               # we have a cycle
          else if w.state == not_visited
               add edge vw to tree T
               dfs(w)
          v.finish = time
          v.state = finished
          time += 1
```

Just add a boolean return value and make it true on finding a cycle, else just return false at the end of the for loop

b) Using DFS, construct an algorithm that either returns a valid ordering of the vertices to build the object or a cycle confirming no such ordering exists. Again, what should your complexity be?

```
DFS(G=(V,E),s)
    for v in V: # initialize the arrays
        state[v] = not_visited; d[v] = infinity
        f[v] = infinity; p[v] = NULL

    new stack S
    time = 0
    state[s] = visited; d[s] = time; p[s] = NULL
    S.push(NULL)

    for edge (s,v) in E:
        S.push(s,v)

    while not S.is_empty():
        (u,v) = S.pop()

        if (v == NULL): // Done with u
            time += 1
            f[u] = time
            state[u] = finished

        else if (state[v] == not_visited):
            state[v] = visited
            time += 1
            d[v] = time
            p[v] = u
            S.push((v,NULL)) # end of v's neighbours
            for edge (v,w) in E:
                S.push((v,w))
```

We can take the regular DFS algo (credit to slides), and add another stack (lets call it T). Everytime after a node is finished, we will push it onto this new stack T. In addition we will add a if statement within the while loop to check if the popped neighbour is visited, and if it is, that means we have a backedge and thus there exists a cycle and we can push it on stack T and break out of the loop as there is no need to further traverse after finding a cycle.

Popping the elements out of T will give either a cycle if it exists in the graph or a valid ordering to build the object, as we know that the finishing time of nodes is lower for the lower heights of DFS trees and vice versa.