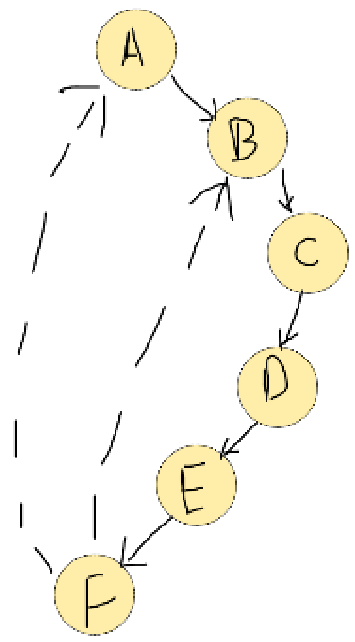
Philippe WS CSC-349

Lab 2

8.) A graph has a unique topological sorry if and only if it has a directed path between two vertices where the path contains every single vertex in the graph once.

10.a)



10.b) Since the algorithm is using DFS, it will observe the cycle from A -> … -> F -> A and the cycle B -> … -> F -> B but not the cycle from A -> B -> F -> A. In an undirected graph this would be the shortest cycle, with a length of three, but the algorithm only observes cycle with a single back edge, thus this algorithm is not sufficient to detect the shortest cycle in an undirected graph.

11.a) The problem we are facing is proving that the directed graph is strongly connected. We know that a graph is strongly connected if from any vertex A it is possible to arrive at vertex B, and vice versa, for any two vertices. Thus, we also can conclude that there exists a cycle between every single vertex on a strongly connected the graph. For the mayor's statement to be truthful, the entire city must be strongly connected, where the city can be represented as a graph with intersections as vertices and streets as edges.

To prove that the graph is strongly connected, we can perform DFS twice on the city graph. The methodology is to first choose any arbitrary vertex v and perform DFS, keeping track of the DFS tree it creates. If DFS does not create a singular tree, then we know for sure the graph is not strongly connected, since if no *path* exists between all nodes on the graph, then *no cycle* can exist between them either. If it does successfully create a singular tree, then we must construct the inverse of the graph, and perform DFS a second time from the same vertex v, tracking the DFS tree once again. Then, again if DFS does not create a singular tree, we do not have a *path* that exists between every node, and of course *no cycle* that exists between every node. However, if for both cases, a path between every node exists, then we have the existence of a cycle between every node, from which then we can conclude that the graph is strongly connected. Since the time complexity of DFS is O(n), then running it twice will still be contained within O(n).

11.b.) Given that the mayors first assumption is false, then for the second assumption to be true, the city graph would be composed of two or more strongly connected components and be sink, otherwise the mayor second assumption would fail for certain vertices. Thus, we can use DFS (O(n)) to discover each strongly connected component first. Once each one is found, we can create a meta-graph of the strongly connected components, where each node is a strongly connected component, and the edge between any two nodes is the edge from the original graph that makes the two connecting nodes disjoint or acyclic.

This meta-graph should form a DAG, since all cycles are contained within the strongly connected components, or nodes. In the same way we can use DFS to obtain source nodes with topological sorting, with the inverse of our meta-graph, we can obtain any sinks by running DFS on the node containing the town hall. If our town hall node returns as the first element or "source node" in our topological sort of the inverse graph, then we know both conditions to be true, and thus the mayors second assumption to be true.

3.)_

```
Class ListElement{
        int value;
        int listId;
}
//Assuming Each list in lists is an ascending sorted queue.
sortKListsAlgorithm(Array[k] lists, int k; int n){
        Array[k*n] sortedList;
        Heap[k] minHeap;
        //Construct min heap using bottom up approach
        //However, the minHeap should be constructed using the minimum elements from each
        array
        for(i = 0; i < k; k++){
                minHeap.add( lists[i].pop() );
        }
        minHeap.constructBottomUp();
        //This will result in a heap of size k with the smallest element of all lists at the root.



        for(i = 0; i < k*n; i++){
                //Grab the root to add to the sorted list
                heapRoot = minHeap.removeRoot();
                //Grab the next value to insert into the heap
                nextMinValue =  lists[heapRoot.listId].pop();
                //Reached end of queue, so returned null, thus insert a max value.
                if(nextMinValue == null)
                        nextMinValue = Integer.MAX_VALUE;
                minHeap.insert(nextMinValue);
                sortedList.[i] = heapRoot;
        }
        return sortedList;
}
```

5.) For this solution, we must use two heaps. One max heap and one min heap. These heaps will be built as the sequence is handed to the algorithm. The goal of the min heap is to store largest half of the elements of the sequence, and the goal of the max heap is to store the smallest half of the elements in the sequence.

For the first number in the sequence, we can assign it either of the two heaps, lets say the minheap. It will of course return as the median.

For the second number in the sequence, we must check if it is greater than the root of the min heap. If it is the case, swap the min and max heap first elements.

Now that each heap has been established, take in the next number in the sequence. If it is greater than the root of the min heap, then perform a heap insertion of said number into the min heap. Otherwise, heap insert the number into the max heap.

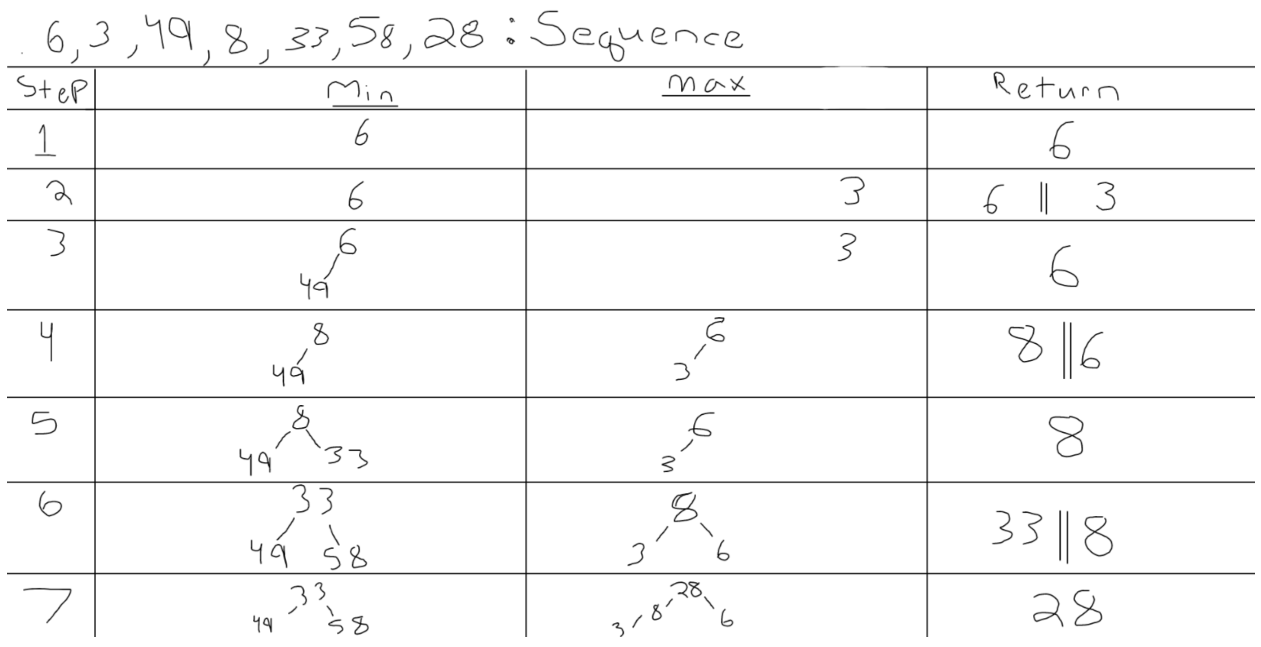
After each insertion, if either heap size is two greater than the other, then perform heap insertion of the root of the greater sized heap into the other one, then instantiate the heap property for both.

Also after each insertion, if the sum of the size of both heaps is even, return either root as the median, otherwise, return the root of the heap whose size is greatest.

Repeat the third step onward until the sequence has been exhausted.

Heap insertion, which is performed using drift up, has a time complexity of O($\log_2$(n)). For each new element, the worst-case number of drifts would be three, one for the new element, then two to reestablish the heap properties for both heaps (given one heaps size exceeds the other by 2), thus the worst case is O(3*$\log_2$(n)) or simply O($\log_2$(n)).

Here is a written-out example of the ordering for a random sequence. It exemplifies the worst case and the resulting heaps from those cases.



6, 3, 49, 8, 33, 58, 28 : Sequence

Step	Min	Max	Return
1	6		6
2	6	3	6 ‖ 3
3	6 / 49	3	6
4	8 / 49	6 / 3	8 ‖ 6
5	8 / 49 \ 33	6 / 3	8
6	33 / 49 \ 58	8 / 3 \ 6	33 ‖ 8
7	33 / 49 \ 58	28 / 8 / 3 \ 6	28