CS 3510 Honors Algorithms
Solutions : Midterm 1

Problem 1. [**Divide and Conquer**]
The three ways of dividing the problem correspond to three recursions. We could write down the recursion and solve for the running time in each of the three cases.

In the first case $T(n) = 3T(n/2) + \Theta(n^2\sqrt{n})$ is the recursion. We could solve this using Master's theorem. Case 3 of Master's theorem would apply in this case, giving a Theta bound of $T(n) = \Theta(n^2\sqrt{n})$.

In the second case $T(n) = 4T(n/2) + \Theta(n^2)$. Second case of Master's theorem applies and we get $T(n) = \Theta(n^2 \log n)$.

In the third case we have $T(n) = 5T(n/2) + \Theta(n \log n)$. The first case of Master's theorem applies here and we get, $T(n) = \Theta(n^{\log_2 5})$.

Comparing the three Theta bounds, we can see that the second case yields the least running time. So the best alternative is the one where we split the problem into 4 pieces.

Problem 2. [**Depth First Search**]
The DFS can be executed and the pre/post numbers would be as shown below. (Ties are resolved in alphabetical order)

Vertex	Pre/Post	Vertex	Pre/Post	Vertex	Pre/Post
A	1/10	B	11/18	C	13/14
D	2/9	E	3/6	F	12/17
G	7/8	H	4/5	I	15/16

There are 5 strongly connected components in the graph. After shrinking the SCC's, we get the DAG in the Figure 1.

Problem 3. [**True or False**]

(a) **False**. Counterexample. Consider the following triangle ABC where the directed edges are $\{(A, B), (A, C), (C, B), (B, A)\}$. If we start a DFS from A, resolving ties alphabetically, then we get a cross edge from C to B. But there is a cycle and hence graph is strongly connected.

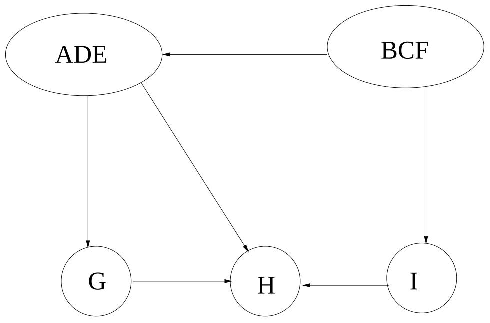

Figure 1: Strongly Connected Components

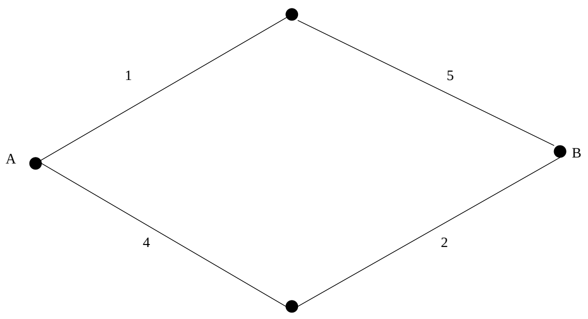

Figure 2: Edge weights are distinct. But two shortest paths.

(b) **False**. Counterexample. Consider a simple graph with two vertices A, B and one directed edge (A, B). If we start a DFS from A, we will get no cross edges but clearly the graph is not strongly connected.

(c) **False**. Counterexample. Consider an undirected graph with two vertices and an edge joining them. Irrespective of the weight, that edge is the heaviest edge in the graph, and all the edge weights are trivially distinct. But the edge is part of the MST. (which is the only spanning tree in the graph)

(d) **True**. See solution for problem 2(b) in Homework 4.

(e) **False**. Counterexample. See figure 2.

(f) **False**. If there is a cycle in the graph, there is no longest path as we can always cycle around and increase the path length. However, if there is no cycle, we can use Bellman ford to find the longest path.

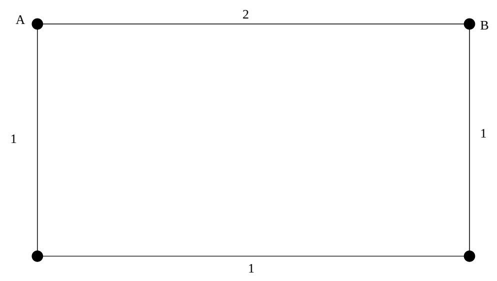

Figure 3: Shortest path from A to B is not in any MST

(g) **True**. Taking the graph with all the weights multiplied by -1 and finding a shortest path is equivalent to this, which can be done by Dijkstra's.

(h) **True**. Since the tree height can be atmost $\log n$, FIND can be executed in $O(\log n)$ time. It follows that UNION has the same running time complexity.

(i) **False**. It is the amortized cost which is $O(\log^* n)$. A single operation is upper bounded by $O(\log n)$ and not $O(\log^* n)$.

(j) **False**. The minimum spanning tree need not be the shortest path tree. As a counterexample, consider the graph in figure 3.

Problem 4. [**Strongly Connected Components**]

(a) Thinking of intersections as vertices, we get a directed graph. The mayor's claim is that the graph is strongly connected. ie., the whole graph is one strongly connected component. Finding the strongly connected components can be done using DFS in linear time (done in class). The only change required is to have a check if the first 'Explore' finds all the vertices in the graph.

(b) Here the mayor makes a weaker claim. She says that from every vertex reachable from the townhall, we should be able to trace back a path to the townhall. In other words, the strongly connected component including the townhall should not have any outgoing nodes. The townhall SCC should be a sink. As seen in class, we can isolate the SCC containing townhall in linear time. Now we can run through all the edges in that SCC and check if any of them is directed outside that SCC. If not, the weaker claim is true. This also can be done in linear time.

Problem 5. [**Searching in an array**]
The algorithm is as given below. We look at the middle element of the array first. If the element $a[n/2] = n/2$, we are done. Else if $a[n/2] > n/2$, since all the array entries are distinct integers, it means that none of the elements in the latter half of the array would satisfy $a[i] = i$. So we can recurse in the former half. IF $a[n/2] < n/2$, by a similar argument, we need to look at only the latter half of the array.

As the base case, we return TRUE or FALSE for a singleton array by just looking at the element and index. Thus the algorithm is correct.

```
Check(a,0,n-1);

Check(array, first, last) {
    if (first = last) {
         if (a[first] = first) return TRUE;
         else return FALSE;
    }
    mid = floor((first + last)/2);
    if (array[mid]=mid) return TRUE;
    else {
         if (array[mid]>mid) Check (array, first, mid);
         if (array[mid]<mid) Check (array, mid, last);
    }
}
```

For the running time analysis, notice that at every recursion the array size is halved. For an array of size n, we would require atmost $\log n$ steps, each of constant time. So the running time is $O(\log n)$.