CSE 373- Spring 2009

Homework 2 solutions:

1.

(a) Sort the list, and find the number of times each number appears. O(n log n) to sort and O(n) to find the mode after sorting.

(b) This one is tricky. But, the first step of the solution is:

Here, A is the given array of size n.

```
J = 1;
For i = 1 to n  {
  If (A[i]=A[i+1]) then {
            B[j] = A[i];
            j++;
  }
}
```

In the above, we are creating a new array B of size at most n/2. It can be shown that if there is an element M that appears in A at least (n/2+1) will also appear in B at least (n'/2+1) times where n' is the size of the new array B.

Now, we can repeat the above process of array B, and so on. The total worst-case running time of the above process is n + n/2 + n/4 … 1 = 2n, which is O(n).

2. Sort the numbers and calculate the sum of any (k-2) numbers. There are $k\,C_n^{k-1}$ = O(n^{k-1}) number of additions. Then, for each such set of (k-2) numbers, use binary search to find the k-1 th and the k-th element that could result in the total being equal to T. This step takes O(log n) time for each set. Thus, the total time is O($n^{k-1} \log n$).

3. Each list has the minimum at front. Find the minimum among the lists using binary tree/ heap in O(log k) time. Remove the element from the list. Repeat the process again (n times).

4.

(a) We use the property given in (b) to get a linear time algorithm to determine whether a graph is bipartite. The property says that an undirected graph is bipartite if it can be colored by two colors. The algorithm we present is a modified DFS that colors the graph using 2 colors.
Input: Graph G
Output: returns true if the graph is bipartite, false otherwise

```
for all v in V
visited(v)= false
color(v) = GREY
    while some s in V such that visited(s) = false
    visited(s) = true
    color(s) = WHITE
    S = [s] (stack containing v)
        while S is not empty
        u = pop(S)
        for all edges (u,v) in E
            if visited(v) = false
                visited[v] = true
            push(S,v)                    //find all the neighbor nodes of u

            if color(v) = GREY   // v is not yet colored by any other of its neighbors
                if color(u) = BLACK
                    color(v) = WHITE   // give adjacent nodes opposite color
                if color(u) = WHITE
                    color(v) = BLACK

            else if color(v) = WHITE   //v is already colored by some other neighbor
        node
                if color(u) != BLACK  //conflicts with u's color
                    return false

            else if color(v) = BLACK
                if color(u) != WHITE
                    return false
    return true
```

(b) An undirected graph is bipartite if and only if it contains no odd cycle.
Proof:
If part=> Let the graph is bipartite. The two vertex sets are V1 and V2.
Consider a path P whose start vertex is s, end vertex is t and it passes through vertices u1, u2, ..., un and the associated edges are (s, u1), (u1, u2), ..., (un, t). Now if P is a cycle, then s and t are the same vertices. Without loss of generality assume s is in V1. Each edge (ui, ui+1) goes from one vertex set to other. Therefore a path must have 2·i edges to come back into the same vertex set where $i \in N$. Since s and t are in same vertex set, so the length of the cycle formed must be 2·i which is even.

Only if part=> Suppose the graph has a cycle of odd length. Let the cycle be C and it passes through vertices u1, u2, ..., un where u1 = un. The associated edges are (u1, u2), ..., (un−1, un). We start coloring edges of using two colors WHITE and BLACK. Without any loss of generality u1 is colored WHITE while un−1 is colored BLACK since n is odd and therefore n − 1 is even. Choosing color of un as WHITE conflicts with the color of un−1 while choosing color as BLACK conflicts with the color of u1. Therefore it is not possible to color an odd cycle with 2 colors which implies that the graph is not bipartite(using the property mentioned in (b))

(c) At most 3 colors are needed to color an undirected graph with exactly one odd cycle. It follows from the only if proof of part (b).

5.

(a)You can form a graph G where the vertex set V corresponds to all possible situations of the containers. A vertex v(a b c) represents the state where the 10 pints, 7 pints and 4 pints container has a pints , b pints and c pints of water respectively. There will be a directed edge from vertex u to v, if the state of v is possible from the state of u by the given pouring constraints. i.e. there will be an edge from (0 7 4) to (7 0 4) or to (4 7 0). You need to find whether there is a path from (0 7 4) to (_ 2 _) or (_ _ 2).

(b) Use DFS or BFS.

(c) Yes, there is a path. (0 7 4) – (4 7 0) – (10 1 0) – (6 1 4) – (6 5 0) – (2 5 4) – (2 7 2)

6. We can compute topological sorting in O(m+n) time using adjacency list.

```
L ← Empty list that will contain the sorted elements
 S ← Set of all nodes with no incoming edges
      while S is non-empty do
            remove a node n from S
            insert n into L
            for each node m with an edge e from n to m do
                  remove edge e from the graph
                  if m has no other incoming edges
                        then insert m into S
return L
```

An alternative algorithm for topological sorting is based on depth-first search. Loop through the vertices of the graph, in any order, initiating a depth-first search for any vertex that has not already been visited by a previous search. The desired topological sorting is the reverse postorder of these searches. That is, we can construct the ordering as a list of vertices, by adding each vertex to the start of the list at the time when the depth-first search is processing that vertex and has returned from processing all children of that vertex. Since each edge and vertex is visited once, the algorithm runs in linear time

7.

(a) The tree contains the edges AE, EB, EF, FG, GC, GD, GH. Cost of the MST is 19.

(b) 2 possible MSTs. The edge EB can be replaced by FB.

(c) AE, EF, EB/FB, FG, GH, GC, GD

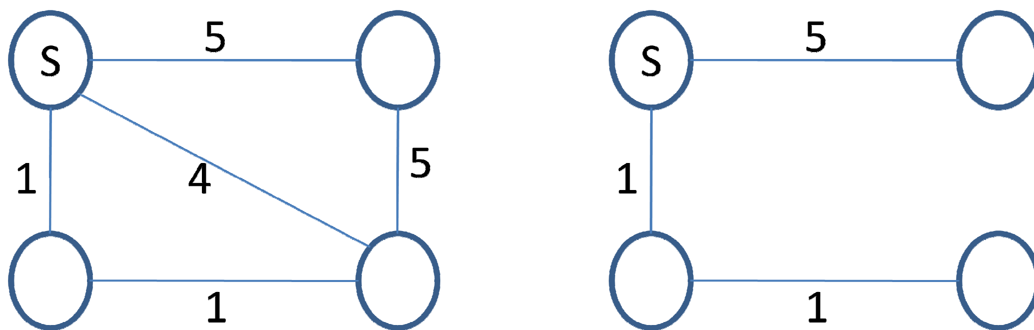
8. Connected acyclic graph (tree) with n vertices have n-1 edges. If it has more than that, then it contains at least one cycle. And therefore, at least one edge can be removed (from that cycle) and the graph will still remain connected. Just explore the edges using unless you get the nth edge. When you get the nth edge, you can stop and say there is a cycle.

9.

(b) True. The unique heaviest edge e is part of a cycle. If e is in the MST, then there is another lower weight edge e1 of that cycle that is not in the MST. We can replace e by e1 and get a lower weight MST => a contradiction.

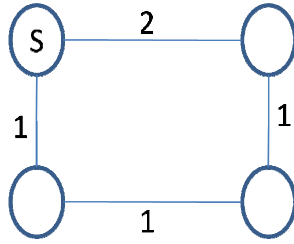
(d)True. Suppose you have an MST without the lowest weight edge e. Add e to the MST, it will create a cycle. Then remove any other edge e1 from that cycle. E1 must have higher weight than e as e is unique. So, you will get a lower weight MST => a contradiction.

(f)False.

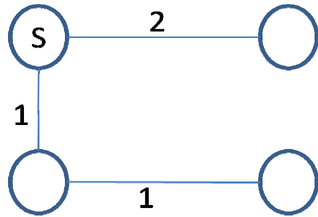


The edge with weight 4 is the unique lightest edge in the top-right cycle. But it is also part of other cycle where there are lighter edges. This edge is not a part of the MST.

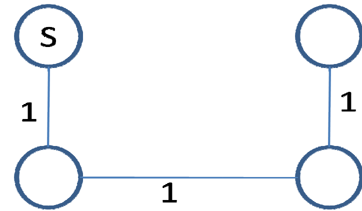
(g)False. The shortest path tree computed by Dijakstra's algorithm is not always minimum spanning tree. The following is an example:



Original topology



Dijakstra's shortest path tree



Minimum spanning tree

10. T is the MST of G and H is a connected subgraph of G.

Let e∈ T intersect H and e is not in an MST T1 of H.

As e∈ T, it is the minimum cut edge. If e is not in T1, then there is another edge e1 that connects this cut. We can swap e1 with e, to get an MST of H with e.