NATIONAL UNIVERSITY OF SINGAPORE

CS2040 – DATA STRUCTURES AND ALGORITHMS

(Semester 4: AY2018/19)

Time Allowed: 2 Hours

INSTRUCTIONS TO STUDENTS

1. Do **NOT** open the question paper until you are told to do so.
2. This assessment paper consists of **Twelve (12)** printed pages and **Nine (9)** questions with possible subsections.

   Important tips: Pace yourself! Do **not** spend too much time on one (hard) question.
3. Answer all questions. Write your answer for the structured questions directly in the space given after each question.
4. You are allowed to use PENCIL to write your answers in this question paper.
5. When this Assessment starts, **please immediately write your Student Number** below (using pen). No extra time given at the end of the exam to do so!
6. This is a **Open Book Assessment**.

STUDENT NUMBER:

A								

(Write your Student Number above legibly with a pen.)

Questions	Possible	Marks
Q1-5	15	
Q6	15	
Q7	20	
Q8	20	
Q9	30	
Total	100	

Section A – Analysis (15 Marks)

Prove (the statement is correct) or disprove (the statement is wrong) the following statements below. If you want to prove it, provide the proof or at least a convincing argument. If you want to disprove it, provide at least one counter example. 3 marks per each statement below (1 mark for circling true or false, 2 marks for explanation):

1. With the partition method of quicksort as written below, the time complexity of the partition method is O(N) in the worst case where $N = j - i + 1$. **[true/false]**

```
public static int partition(int[] a, int i, int j) {
  int p = a[i]; // p is the pivot, the i-th item
  int m = i;    // Initially S1 and S2 are empty
  for (int k=i+1; k<=j; k++) {
    if (a[k] < p) {
      m++;
      for (int c=k; c > m; c--) {
        int temp = a[c];
        a[c] = a[c-1];
        a[c-1] = temp;
      }
    }
  }
  int temp = a[i];
  a[i] = a[m];
  a[m] = temp;
  return m;
}
```

False.

In the worst case, the 1st half of the unexplored region is in the right partition (putting values into right region is O(N) time), then the 2nd half is in the left partition. This means for ½N of the values you have to shift all the way from the unexplored region past the right partition into the left partition, thus each shift is O(½N) = O(N) time. This is done for ½N of the values, thus total shift done = O(N^2). Total time complexity is O(N)+O(N^2) = O(N^2).

2. When an item at index i of a min-heap is updated with a new value, calling both ShiftUp(i) and ShiftDown(i) will re-heapify the min-heap. However only one of the operations will cause swaps to happen or none will cause any swap to happen (item remains where it is). It is impossible for both to cause swaps to take place. **[true/false]**

True.
When a value is updated it can become bigger or smaller than the original value. If it becomes bigger, then calling ShiftUp nothing will happen as its parent will be smaller than it, if it is still smaller than either of its two child then it will remain where it is (no shifting up or down), if it becomes bigger than both its children it will shift down. The same argument applies when it comes smaller than its original value.

3. The smallest bipartite graph in terms of number of edges is a tree. **[true/false]**

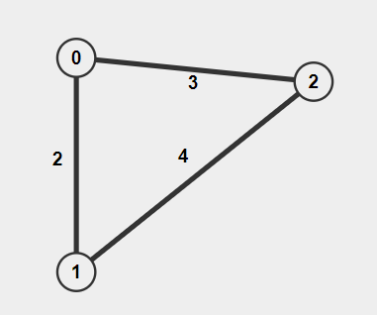
False.
A totally disconnected graph is still a bipartite graph as you can simply put any group of vertex into the 1st set and the rest into the 2nd set and you won't have edges between vertices in the same set.

4. For the counting component algorithm, since each vertex is visited once and all its neighbor are processed, the time complexity can be written as the sum of all neighbors processed for all vertices, which is O(E) in general instead of O($V + E$) or O(max(V, E)). **[true/false]**

False.
If the graph is a totally connected graph, where E=0, time complexity is not O(1). This is because, calling DFS/BFS on each vertex will be O(1) time as the source vertex will have no neighbors, but we still have to go through all vertices to count the number of components. Thus total time is O(V). In general V and E are independent of each other so we should write O(V+E) instead.

5. For any connected weighted undirected graph, Prim's algorithm will always maintain a sub-tree of the MST as the algorithm progresses until the final MST is obtained at the end of the algorithm. Kruskal's algorithm on the other hand will always produce a forest of subtrees of the MST (disconnected subtrees of the MST) at least once during the running of the algorithm, but will merge these subtrees into the final MST at the end. You may assume for Prim's for edges with same weight, they are ordered by neighbor vertex number, and for Kruskal's the edge list is sorted first by edge weight then by 1st vertex number then 2nd vertex number. **[true/false]**

   False.
   In the given connected weight graph below, both Prim's and Kruskal's will maintain a subtree of the MST of the graph during the running of their algorithm and at the end produce the same MST.



   Edit: Also can be True.
   If we consider one vertex as a subtree, in the beginning Kruskal's must have N subtrees.

Section B – Applications (85 Marks)

Write in pseudo-code (code-like or English description is ok as long as it is clear and specific). Any algorithm/data structure/data structure operation not taught in CS2040 must be described, there must be no black boxes. Partial marks will be awarded for correct answers not meeting the time complexity required.

6. **Another linked list manipulation question … [15 marks]**

   Given a BasicLinkedList A where each node contains a string, write the algorithm for the method

   public BasicLinkedList NewList(BasicLinkedList A, int p)

   which will create a new BasicLinkedList that only contains strings of length >= p from A in the same relative ordering. This may result in an empty list returned if there are no strings in A which are as long or longer than p. You cannot destroy A in the process of creating the new linkedlist. You cannot use any of the methods in the BasicLinkedList class and the Java LinkedList API. You have access to the head of A. This means you have to make use of references to ListNode and manipulate next pointers to achieve your goal. You can create new ListNode if required, and have access to operations in ListNode.

public BasicLinkedList NewList(BasicLinkedList A, int p)

```
let curA = A.head
create new BasicLinkedList B
let curB = B.head  (which is null at the start)

while (curA != null)
    if (curA.item.length() >= p)
       create new listNode t storing curA.item
       if (curB == null)
         curB = B.head =  t
       else
         curB.next = t
         curB = curB.next
    curA = curA.next
```

7. **Can we make it simpler … [20 marks]**

a) John works in a diamond processing factory. His friend Tom has made a machine that will scan each diamond that passes by the machine on a conveyor belt. The scan will first tag the diamond with a unique id number and also determine the weight of the diamond. This pair of info will then be inserted into a min-heap with the weight as key. After every 1 hr, the smallest diamond is obtained by looking at the root of the min-heap and thrown away. The min-heap is then reset and the whole process is repeated. Even though using a min-heap result in a O(lgN) per insertion where N is the # of diamonds scanned per hr and peeking at the top of the heap is O(1) time, John cannot help but think this solution is too complicated for what it is trying to achieve and can be made simpler and more efficient. Please design an even simpler algorithm without using min-heap that will do the same thing but is even more efficient (i.e better time complexity)! **[10 marks]**

Simply use a variable to keep track of the weight of smallest diamond as they are scanned one by one and another variable to keep track of the id of the smallest diamond. Update both whenever encounter a smaller diamond. After one hr remove the diamond with the current smallest weight. This only take O(N) time.

b) Jerry works in the same diamond factory, and he deals with cutting the diamonds after the unwanted ones have been thrown away. The diamonds will come to him in batches of size N where 100 <= N <= 1,000,000. Now if the diamonds are too small or too large (based on weight), they require special processing which is not handled by Jerry. Thus he needs to efficiently select only those which are within a certain weight range $[X, Y]$ where X <= weight <= Y. Tom has yet again suggested building another machine that will again scan the diamonds one by one and insert them by their weight into an AVL tree. The algorithm is then as follows
1.) Build the AVL tree in O($NlgN$) time.
2.) Search for diamond p with weight just >= X → O(lgN)
3.) From p just keep calling successor until diamond with weight <= Y → O(N)
4.) Return all diamonds found in 2.) as the diamonds to be processed → O(N)
Total time complexity is O($NlgN$)

Jerry just like John thinks this is too complicated for what he is trying to achieve, and has asked you to once again design an even simpler algorithm without using AVL, that will do the same thing but is even more efficient! **[10 marks]**

Keep two variables X and Y where X is the left boundary of the weight range and Y is the right boundary of the weight range. Whenever a diamond is scanned check if its weight is between X and Y inclusive if it is add it to the back an array/arraylist in O(1) time. At the end return that array/arraylist as the set of diamonds to be cut. Time complexity is O(N).

```
diamond[] <- init an array of to hold diamonds // where diamond[i] holds the weight of the ith diamond -> O(N)

call radix-sort on diamond // O(N)

Integer start = null;
Integer end = null;

for  each element in diamond, i:
  if start == null && diamond[i].weight >= X && diamond[i].weight <= Y:
    start = d.id
  else:
    if diamond[i] <= Y:
      end = i;

if (start == null): return empty array

if (start != null && end == null): return a subset of diamond[] from diamond[start] to diamond[diamond.length-1]

return a subset of diamond[] from diamond[start] to diamond[end]

// lmao wah lao i overcomplicated tingz again
```

```
UFDS t <- init a UFDS where each vertex v in V is initially a disjoint-set

for each <u,v> in E:
  if !t.isSameSet(u,v):
    t.union(u,v);

if t.numDisjointSets != V:
  return false

return true
```

8. **Can it be done without …. [20 marks]**

Given a graph G which is a subgraph of another connected graph G', we want to test if G is also a spanning tree of G'. Give an algorithm to do this without using BFS/DFS or any of their variants. You are only given the number of vertices V in G' and the edge-list E representing G.

Use UFDS and hashmap for this purpose.
1.) Create a UFDS u with number of disjoint set = size of E
2.) Create an empty hashmap h where key value are the same.
3.) Set vcount = 0, ccount = size of E
4.) For each edge (a,b) in E
    if u.sameSet(a,b)
        return false // there is a cycle so cannot be a tree
    else
        if a is not in h
            hash (a,a) into h and increment vcount by 1
        if b is not in h
            hash (b,b) into h and increment vcount by 1
        u.unionSet(a,b)
        Decrement ccount by 1

5.) if ccount == 1 and vcount == V
        return true
    else
        return false

9. **Star Track [30 marks]**

a) Captain Pica has led his team of intrepid galactic explorers to planet XIV. On this planet is located an underground network of junctions and connecting tunnels (Pica has a map of this underground network). There are N junctions, including the starting junction where Pica and his team is and at each junction is planted a photon bomb. If all N of them explode then the planet will be destroyed. These junctions are connected to each other via tunnels that can be travelled in both directions and there are $N-1$ to $\frac{1}{2}N^2$ such tunnels. There is always a way to reach any junction from any other junction via tunnels, and all the tunnel are different in length.

In order to stop the photon bombs from going off, Pica and his team (there are also N members in the team including Pica) start off as a group at the starting junction. At any newly discovered junction including the starting junction, 1 member will stay behind to disarm the bomb (it takes a long time to disarm a bomb) and the rest can split up into smaller groups if necessary to travel 1 to k of the k unexplored tunnels connected to the junction.

Since there are N members there must be a way from the starting junction for all members to each reach a junction and disarm the photon bomb there. Model the underground network of junctions and tunnels as a graph, and give an algorithm to minimize the total distances of tunnels travelled (count the distance travelled by a group along a tunnel as simply the length of the tunnel) and each junction is reached by 1 member of Pica's team. **[18 marks]**

i.) What do the vertices and edges in your graph represent? Explain. **[2 marks]**

Vertices are junctions and edges are tunnels that connect 2 junctions.

ii.) Are the edges directed or undirected? Explain. **[2 marks]**

Undirected as you can traverse a tunnel in both directions

iii.) Is this a connected or disconnected graph? Explain. **[2 marks]**

Connected graph as there is way to get from any junction to any other junction

iv.) What are the weights in your graph? Explain. **[2 marks]**

Length of the tunnels since you want to minimize distance travelled along tunnels

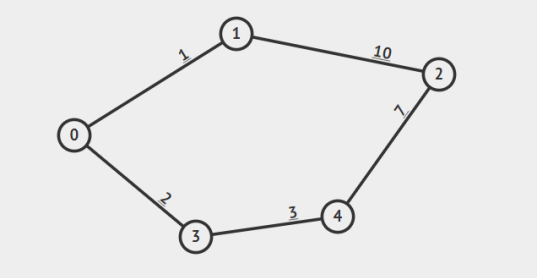
v.) Using the graph as described above give the most efficient algorithm you can think of to minimize total distances of tunnel travelled and also allow each junction to be reached by 1 member of Pica's team. **[10 marks]**

Model underground network as an undirected weighted connected graph where vertices are the junctions and edges are the tunnels, and weight of an edge is the length of the tunnel.

Minimizing total distance of tunnels travelled so that each member of the team can reach a junction is basically finding the cost of the MST of the graph as constructed above, since each tunnel that is used is travelled only once (MST edge) and groups travelling a tunnel is taken as one unit (total distance is simply MST edge weight).

Simply keep a variable $cost$ to track the MST cost, then run Kruskal's (or Prim's) and for each edge taken to be in the MST, add it to $cost$ and at the end return $cost$.

NOTE: SP spanning tree does not minimize total distance travelled in general. E.g in the graph below



SP spanning tree will include the edges 0-1, 1-2, 0-3, 3-4 with total edge weight = 16
MST will include the edges 0-1,0-3,3-4,4-3 with total edge weight = 13.

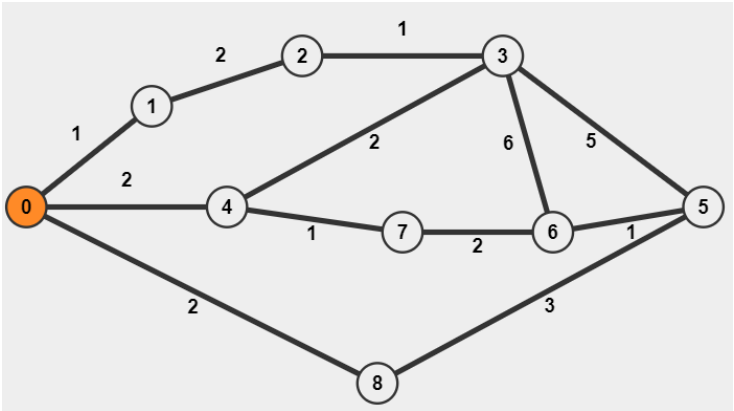
b) After saving planet XIV, Pica and his team have returned to their space ship. They now need to travel to M ($1 <= M <= 100$) planets to disarm all the photon bombs there, and the M planets must be visited in a specific sequence $\{A_1, A_2, \ldots, A_m\}$ that is given. The navigator of Pica's space ship uses a star map to travel the galaxy. This map is basically a graph that models discovered planets as vertices and there will be a bi-directed edge linking 2 planets if the straight line path between them have been explored previously (edge weight is distance between the planets). The graph is not a complete graph since not all straight line path between all pairs of planets have been explored, but it will be a connected graph meaning there is at least one path between any pair of discovered planets. There are between 1,000 and 100,000 discovered planets, and between 10,000 and 1,000,000 edges connecting planets.

To move from some planet a to some other planet b, the navigator will always use the **shortest distance** to get from a to b from the star map. However, from the starting point (which is planet XIV), once the space ship has reached some planet c from planet XIV by travelling P distance, the space ship can warp from c to XIV immediately and from XIV the space ship can warp back and forth between XIV and any planet that has shortest distance <= P from planet XIV (imagine P is the "radius" of warp from planet XIV), thus effectively making shortest distance between XIV and such planets **ZERO**. However once the spaceship warps, it uses up all its fuel and if it does not land back at planet XIV or any of the M planets to refill, the space ship will be marooned in space. Whenever the

spaceship travels a greater distance than the current P from planet XIV, P will be updated to this new distance. At the start, P = 0.

Given the above scenario, give an algorithm so that the navigator will minimize the travel distance to all M planets in the sequence required, starting from planet XIV.

E.g If there are 4 planets to be visited and the visitation sequence is {2,4,3,5} (numbering is their vertex number in the star map), and the star map is as follows (vertex 0 is planet XIV).



Bi-directed edge represented as undirected for ease of viewing

First the space ship will start at vertex 0 (planet XIV), and visit 2,4,3,5 as follows

- **Vertex 2** – Since P is currently 0, we have to travel to 2 via the path 0-1-2 with shortest distance 3. Now P = 3.
- **Vertex 4** – From vertex 2, the space ship can warp back to vertex 0 and then warp to vertex 4 for a total of 0 distance since shortest distance from vertex 0 to vertex 4 is only 2 which is < P = 3. Now P is still 3
- **Vertex 3** – From vertex 4 we can travel the edge 4-3 to reach vertex 3 with cost 2, but this is not optimal, instead warp back to vertex 0, then warp to vertex 2 and travel the edge 2-3 for a cost of 1. Now P = 3+1 = 4.
- **Vertex 5** – From vertex 3 we can travel the edge 3-5 to get to vertex 5 with cost 5 but again this is not optimal. We can possibly warp back to vertex 0 then warp to vertex 7 and use the edges 7-6-5, since shortest distance from vertex 0 to vertex 7 is only 3 and currently P = 4. However, since 7 is not in the visitation sequence, the space ship has run out of fuel and cannot move. The optimal is to warp back to vertex 0 from vertex 3 then warp to vertex 4 and travel the path 4-7-6-5 for a total cost of 4. Now P = 4+4 = 8.

We have now visited 2,4,3,5 in sequence and used a minimized total distance of 8.

Given the star map (a connected undirected weighted graph G(V,E)), and the visitation sequence $\{A_1,\ldots A_M\}$ of M planets, start from vertex 0 (planet XIV) and describe the most efficient algorithm you can think of that will visit all planets in the sequence given and travel the shortest distance possible. **[12 marks]**

This is SSSP with a twist.
Let S be the sequence of planets to be visited and G be the graph representing the star map.
Let V be number of discovered planets and E be number of edges connecting planets.

1.) Run SSSP using modified Dijkstra from vertex 0 → O((V+E)logV)
2.) Travel to the 1st planet s_1 in S. P = cost of SP(0, s_1). Add 0 and s_1 to set of planets S' visited.
3.) For each planet s_x still not visited in S → O(M)
    Run SSSP using modified Dijkstra from s_x → O((V+E)logV)
    If SP(s_x,0) <= P
        add s_x to S' without incrementing P → O(1)
        (this is equivalent of warping from previous planet visited back to 0 then warp from 0 to s_x)
    else
        For each planet in S' find a planet s'_x that minimizes SP(s_x, s'_x) → O(M)
        Add s_x to S' and set P = P+ SP(s_x, s'_x) → O(1)
        (this is equivalent of warping from previous planet visited back to 0 then warp from 0 to s'_x and travel along SP from s'_x to s_x which is also SP from s_x to s'_x)

Return P at the end. This is the minimized total distance travelled.
Total time taken is dominated by 3.) which is O(M(V+E)logV+M^2) = O(M(V+E)logV) since M^2 is bounded by 10,000.

~~~ END OF PAPER ~~~