Divij Mehra 999705225
Edward Moore 999947919
Ramaneek Gill 1000005754
Seung Hoon Lee 996033956

CSC263 Assignment #2

#1
A)

Cost = number of digits changed

The last digit changes every time INCREMENT is performed.
The second last digit changes every 100th time INCREMENT is performed.
The third last digit changes every 1000th time INCREMENT is performed.
The ith last digit changes every $\frac{1}{10^{m-1}}$ th time INCREMENT is performed.

The total cost is

$$\frac{m}{10^{m-1}} + \frac{m}{10^{m-2}} + \frac{m}{10^{m-3}} + \cdots + \frac{m}{10^1} + \frac{m}{10^0} = m\sum_{i=0}^{m}\frac{1}{10^i} < m\sum_{i=0}^{\infty}\frac{1}{10^i} = \frac{10m}{9}$$

$$\sum_{i=0}^{\infty}\frac{1}{10^i} = \textit{geometric series where } a = 1, r = \frac{1}{10}$$

$$= \lim_{n\to\infty} a\frac{1-r^n}{1-r} = \lim_{n\to\infty}(1)\frac{1-\frac{1}{10}^n}{1-\frac{1}{10}} = \frac{1-0}{0.9} = \frac{10}{9}$$

We can charge $10/9 for each INCREMENT operation

For each digit, we need to save enough for the time when the next digit has to be incremented. For example, the following table shows each increment saves 1/9 and uses up the accumulated credit when there is one more digit to increment.

	Beg Credit	CHARGE	Cost	Difference	End Credit
0	0	1 1/9	1	1/9	1/9
1	1/9	1 1/9	1	1/9	2/9
2	2/9	1 1/9	1	1/9	1/3
3	1/3	1 1/9	1	1/9	4/9
4	4/9	1 1/9	1	1/9	5/9

5	5/9	1 1/9	1	1/9	2/3
6	2/3	1 1/9	1	1/9	7/9
7	7/9	1 1/9	1	1/9	8/9
8	8/9	1 1/9	1	1/9	1
9	1	1 1/9	2	- 8/9	1/9
10	1/9	1 1/9	1	1/9	2/9

Idea: Charge each INCREMENT operation 10/9.

Credit Invariant:

"At any step during the sequence, each INCREMENT operation changes at least one digit if there is no carry over, and at most k digits if there is at least one carry over. By charging 10/9 for each INCREMENT operation, the data structure stores enough credits to cover the cost of incrementing digits even when there is a need to increment two or more digits (like the last column in the table)."

Proof by Induction:

Initially, counter is 0 and has not credit. The credit invariant is trivially true.
Assume the credit invariant is true and INCREMENT is performed.
The cost of changing one digit is covered by 9/9 from the charge of 10/9.

For the cost of changing digits caused by carry over…
10/9 is 1.111111111…
For 0.1 from 1.11111… we can accumulate an extra credit every tenth INCREMENT operation, which we can use when we need to increment every 10th digit.
For 0.01 from 1.11111… we can accumulate an extra credit every 100th INCREMENT operation, which we can use when we need to increment every 100th digit.
For 10^{-i}, we can accumulate an extra credit every 10^{i}th INCREMENT operation, which we can use when we need to increment 10^i th digit.

No other digit changes besides the least significant digit and digits that are updated due to carry over. As a result, the credit invariant is maintained.

This shows total charge for sequence is upper bound on total cost. For INCREMENT operation, total charge is $\frac{10}{9}m$, so amortized cost per operation is $\frac{\frac{10}{9}m}{m} = \frac{10}{9}$

B)

To support RESET, we need to accumulate enough credits to convert at most k digits back to 0 at any step so that total charge is greater than total cost at all times. For example, we should have $1 stored for anytime when there is only one digit and $2 stored for anytime when there are two digits and so on.

The charge for INCREMENT is the same as part A (, which is 10/9).

The actual cost of RESET is at most k, since there are at most k digits to change to 0.
After m operations, the number of digits, k, is at most $\lceil \log_{10} m \rceil$.
Therefore, the charge for RESET is $\lceil \log_{10} m \rceil$.

Idea: Chrage INCREMENT $\frac{10}{9}$ and charge RESET $\lceil \log_{10} m \rceil$

Credit Invariant:

Credit invariant consist of two parts, one for INCREMENT and the other for RESET.
Credit invariant for INCREMENT is the same as part A.
"At any step during the sequence, a RESET operation can change at most k digits. By charging $\lceil \log_{10} m \rceil$ for each RESET operation, the data structure stores enough credits to cover the cost of resetting every digit back to '0'."

Proof by Induction

Proof for INCREMENT is the same as part A.
Initially, the every digit is 0 and the data structure does not have any credit. Therefore, the invariant is trivially true.
Assume that the invariant is true after a certain number of INCREMENT and RESET operations. The next operation could be either INCREMENT or RESET.
If the next operation is INCREMENT, the credit invariant is maintained as shown in part A.
If the next operation is RESET, the charge of $\lceil \log_{10} m \rceil$ is enough to cover the cost for changing all k digits, since $\lceil \log_{10} m \rceil \geq k$. The maximum number of digits(worst case sequence complexity) to change is when all the operations up until the current operation were all INCREMENT, and there are $\lceil \log_{10} m \rceil$ digits at most.
If the last operation changed 9 to 0, and had a carry over, there would be one more digit than $\lceil \log_{10} m \rceil$. However, RESET would still take care of all the digits, since the last digit is 0. The cost of $\lceil \log_{10} m \rceil$ will be enough for RESET. Therefore, the credit invariant is maintained.
If the last operation did not generate carry over, RESET would be able to change all digits to 0 with the charge of $\lceil \log_{10} m \rceil$. Therefore, the credit invariant is maintained.
Hence, the number of credits in the data structure will never become negative, so the total charge for the sequence is an upper bound on the total cost of the sequence. The amortized cost per operation is less than $\frac{10}{9} + 1 = \frac{19}{9}$.

$$\frac{\lceil \log_{10} m \rceil}{m} \leq 1 \; for\ any\ m$$

2.

a)

room 1:

1, 4, 5, 8

room 2:

2, 3, 6, 7

b)

```
num_rooms = 0
allowed_in_this_room = FALSE

For each vertex v
    if(num_rooms == 0)  //if the graph isn't empty and there are 0 rooms availabe create one
        num_rooms = 1
    i = 1

    while (i <= num_rooms)
        for each vertex u in G.ADJ[v]     //G.ADJ is the adjacency list for the vertex
```

```
                if u is in room[i]    //then v is not     allowed in this room
                    allowed_in_this_room = FALSE
                    break                         //lets check if the next room is allowed
                else
                    allowed_in_this_room = TRUE

            if(allowed_in_this_room == TRUE)
                insert v into room[i]

            i = i+1

        if(allowed_in_this_room == FALSE) //every room that exists isn't allowed to host v,
sowe need a new room
            num_rooms = num_rooms + 1
            insert v into room[num_rooms]
```

c)

```
num_rooms = 0
allowed_in_this_room = FALSE

For i <- 1...n
    if(num_rooms == 0)  //if the graph isn't empty and there are 0 rooms availabe
create one
        num_rooms = 1
    k = 1
    while (k <= num_rooms)

        for j <- 1...n            //want to check if all of the nodes connected to G[i] are not in a specific room

                                  //if a room is allowed we will insert the vertex G[i] into the room

            if G[i,j] == 1   //then vertex j is connected to vertex i

                if G[i,j] is in room[k]    //then this vertex is not allowed in this room

                    allowed_in_this_room = FALSE

                    break

                else

                    allowed_in_this_room = TRUE

        if(allowed_in_this_room == TRUE)

            insert G[i] into room[k]

            break

        k = k+1

    if(allowed_in_this_room == FALSE) //every room that exists isn't allowed to host v, sowe need a new room

        num_rooms = num_rooms + 1
```

```
                insert G[i] into room[num_rooms]

        i = i+1
```

d)

There are 3 loops in my algorithm that are nested in each other, they will loop over each of the vertices, check each existing room (and create a new one if needed and check if the selected vertex is allowed in the specific room by checking if none of it's connected vertices are in there. In the worst case every vertex will be connected to each other, therefore there will be n rooms, n vertices, and n-1 connections which results in a runtime of O(n^3)

C's input size is n^2 whereas B's input size is n. Therefore I would choose b's algorithm over c since running time for b would be slightly less than c based on input size alone since they share the same worst case running time.

e)

room 1:

1, 2, 5, 8

room 2:

3, 6, 7

room 3:

4

f)

The maximum number of k rooms we need to fit n nodes into seperate rooms is less than or equal to n. Now that we have a maximum we can use the properties of a graph and a vertex's connections to try and reduce this number as small as possible.

A problem we can run into when using the algorithms above is limiting a rooms 'flexibility' to early by inputting students into a room where they don't have many collaborators. If we were to input students who rarely collaborated into an arbritrary room R first then it could prevent another student who has many collaborators from joining the room even if it is only limited by one student being in that room.

To find the minimum amount of rooms to seperate collaborators we want to fully put a "maximum restriction" on students that can't be allowed in a specific room first. This is done by first inserting students into rooms where students have the greatest amount of collaborators.

For example we will use the graph given to us in a).

We will then reorder the graph in descending amount of collobrator order.

1: [3, 6]

2: [4]

3: [1, 4, 5]

4: [2, 3, 6, 7]

5: [3, 6, 7]

6: [1, 4, 5]

7: [4, 5, 8]

8: [7]

becomes

4: [2, 3, 6, 7]

3: [1, 4, 5]

5: [3, 6, 7]

6: [1, 4, 5]

7: [4, 5, 8]

1: [3, 6]

2: [4]

8: [7]

Now we will run our algorithm decalred in the answer to question b).

Its output will be:

```
room 1

4, 5, 1 8

room 2

3, 6, 7 2
```

Notice how there is only 2 rooms that are used, unlike 3 in e).

To illustrate this new 'algorithm' the first two insertions are the most important.

In our first insertion we insert 4 intom room1, this puts a "maximum restriction" on room1 that prevent students 2, 3, 6, 7 from being inserted.

In our second insertion we insert student 3 into room2, this prevents students 1,4,5 from being inserted into room2.

The key to minimizing the amount of rooms used to seperate collaborators is to create as many restrictions as quickly as possible with the least amount of rooms that need to be created. This quickly creates tight bounds for each room for where a student will be placed during its insertion. When we have loose bounds on a room a student can be inserted in to many different rooms, this is problematic since it doesn't create a proper "guideline" for where non-inserted students belong.

Therefore by selectively choosing which students to insert first based on the descending order of amount of collaborators we can be assured of a minimum amount of rooms k will be used by the algorithm AssignRooms. Note that k is bounded: 0 <= k <= n

3.) a)

Let G be a connected, undirected graph, Gπ a DFS-tree for G, and v a vertex of G. Lets say another vertex u exists, and since Gπ is a tree, by definition, there is a path from u to v for all u,v∈V. Since v is the root in this case, and it only has one child, if we were to take the path from u to v, we would find that we cannot go further than v as π[v] does not exist. Because of this characteristic, if v was to be removed it could not be considered an articulation point, as the vertices on the path between u and v (excluding v) would still be reachable. Similarly, if u was the child of v and v was removed, u would still be reachable, thus v is not an articulation point. So, by contradiction, if v only had one child, removing v would not disconnect G.
Lets say v has a child u and π[v] exists (so v is not the root), and a back edge exists from u to π[v]. If v was to be removed, G would not be disconnected, as the back edge ensures that u and π[v] are still connected. This concept would thus apply for any descendant of u that has a back edge connecting it with any ancestor of v, meaning every vertex that was connected when v was part of the tree is again connected when v is removed. Therefore, by contradiction, v can only be an articulation point if no back edge from a descendant of u to any proper ancestor of v exists. This proves that when Gπ does not have a back edge and v is removed, G becomes disconnected, as there is now no path that exists between π[v] and u, and in turn when Gπ is made from G, there will be two connected components.

b) Let G be a connected, undirected graph and e = {u,v} be an edge in G. If G was to contain a simple cycle, and e was a part of this simple cycle, G - {e} would be connected, as there would be at least one other path from u to v in G. This is because removing e would break one of the paths, but u and v are still reachable from the path that wasn't destroyed. Therefore, e could not be a bridge, which is a contradiction.

c) Let List L1 represent a list of vertices
Let List L2 represent a list of edges

```
DFS(G=(V,E)):
    for each v in V:
        d[v] = f[v] = oo
        π[v] = NIL
        m[v] = 0
    time = 0  # global "clock"
    for each v in V:
        if d[v] = oo:   #v is "undiscovered"
            DFS-VISIT(G,v)

DFS-VISIT(G=(V,E),v):
    #v has not been examined before
    d[v] = time = time + 1
```

```
        m[v] = min(d[u])
        for each (v,u) in E:
            π[u] = v
            if m[v] == 0:   #v has no child
                L2.append(E)  #this edge is a branch
                DFS-VISIT(G,u)
            else if π[v] = NIL:  #v is the root
                if m[v] == 1:  #v has one child
                    L2.append(E)
                else if m[v] >= 2:  #v has more than one child
                    if f[u] = oo:  #u is an ancestor of v
                        pass  #this is a cycle
                    L1.append(v)
                    L2.append(E)
        f[v] = time = time + 1
        return L1, L2
```

DFS takes time O(n) in addition to calls to DFS-VISIT. DFS-VISIT is called at most once for every vertex v in V because it checks to see if this vertex has any children or has a parent with respect to the two if statements. Therefore, this algorithms' worst case running time would be O(n^2).

#**Algorithm Outline**

-if v has a parent but no children:

    -update list for # of branches (because v would be unreachable if this edge was removed)

-if v is the root

    -if it has one child, don't update list for AP (), add this branch to the list

    -if it has two children

        -check if there is a cycle

            -if cycle, skip

            -if no cycle, add v to list of AP's, add e to list of branches

---

4a)

We know that running BFS on a connected, undirected graph gives us a tree containing every vertex in the graph and some of the edges. Assume we start the BFS at vertex s and every new vertex discovered is added to the tree along with the edge between the discovered vertex and its parent.

Let us also assume that there is a cycle u-v-w in the graph. Now u is discovered and added to the tree. The BFS then discovers v and adds v to the tree along with the edge between u and v and similarly for vertex w along with the edge from v to w. However, the w-u edge is not added to the tree since u has already been discovered and thus there is no cycle in the tree.

The tree created contains every vertex from G, has no cycles and every vertex in T can be reached from any other vertex in T. Therefore, the tree T created is a spanning tree.

4b)

Let us assume that T is not a minimum spanning tree (MST) of G. Since T is a spanning tree, it cannot contain a cycle. Let us assume that it contains edge e from the cycle but does not contain edge e*. Let us now replace edge e in the spanning tree with edge e*. The weight of the original spanning tree T is less than or equal to the weight of the new spanning tree as the weight of e is less than or equal to the weight of e*. This is because the weight of each edge is a sum of the distances of the two vertices it connects, from the vertex that BFS starts at. This shows that the new spanning tree is either not a minimum spanning tree or is also a minimum spanning tree with equal weight in comparison to T but containing different edges.
Therefore, T is a minimum spanning tree.