Q1. Determining the end of an array.

A. →

Algorithm:

1) First find value Start checking the elements at index two to the power 'i', that is 2, 4, 8, 16, 32, 64, …, 2^i and stop once you have found the first occurrence of 'Infinity'. Suppose you stop at an index 'k'
2) Now use divide and conquer technique to find out the index of last number that is not infinity.

```
if(array[array.length] != INF)
   return array.length;

k=0
element = array[0]
while(element != INF)
   k = k + 1
   element = array[2^k]                    …STEP1

start=[2^(k-1)]
end =[2^k]

while(start < end)
   mid=(start + end)/2
   if(array[mid]! = INF)
      start = mid + 1
   else
      end = mid - 1

return (end - 1);                          …STEP2
```

Complexity:

1) As we are moving in the range of power of 2, and there are total 'n' non-infinity elements, first step of algorithm would take log(n) + 1 elements, as in the worst case to get first infinity we might travel to '2n' index. Therefore, running time would be log(2n) = log2 + log(n) = log(n) + 1
2) In second step as we do a divide and conquer search between 2^{k-1} to 2^k elements and since in worst case value of 2^k = 2n, the difference between 2^k - 2^{k-1} = n.
   Thus, we will apply divide and conquer on at max 'n' elements. Thus, running time of step two would be log(n)
3) Thus, total complexity would be

$$f(n) = \log(n) + 1 + \log(n)$$
$$= 2*\log(n) + 1$$
$$f(n) = O(\log(n))$$ … [as log(n/2) < log(n)]

Q2(a) **Part 1** k-colorable graph with Δ.

A. →

General Proof –

When we say, a graph has a degree Δ, we mean that vertices in graph has maximum Δ edges coming out from it. Thus, we can say that there are Δ neighboring vertices for a vertex. Hence, including the starting vertex there are total Δ + 1 vertices. Now suppose all these vertices are connected to each other, then we will require maximum Δ distinct colors to color whole graph.

Proof by Induction –

BASE CASE

1) Δ = **0**

colors required = Δ +1 = 0 + 1 = **1**

This means the vertices in the graph are not connected to each other at all. Coloring all the vertex with same color, we stick to the constraint that no two neighboring vertices have same color.

2) Δ = **1**

colors required = Δ +1 = 1 + 1 = **2**

This means that every vertex in the graph is connected to exactly one other vertex. Coloring two connected vertices with two different color, we stick to the constraint that no two neighboring vertices have same color.

INDUCTION STEP

For value k such that, **0 < k <** Δ

1) We know that a graph G with degree 'k' can be colored in 'k + 1' distinct colors.
2) Now suppose we add one more vertex in the graph G, such that the new node is connected to all other vertices in the graph and let`s call new graph as G` (This is worst case, in the best case new vertex would be connected to only one other or no other vertex in G).

degree of new graph G`	=	1	+	degree of old graph G
	=	1	+	(k)
	=	k + 1		

This means that degree of G` would be degree of G plus one, which is 'k +1'.

Now, since new vertex is connected to all other vertices in graph G, will have to give a new color to the new vertex.

total colors required for G`	=	1	+	total colors required for G
	=	1	+	(k + 1)
	=	k + 2		

We require k + 2 distinct colors to color graph G`

3) Hence, from induction step 1 and 2 we can say that, we require k + 2 colors to color graph which has degree k +1.

Q2(a) **Part 2** Construct graph with maximum degree Δ and that is Δ + 1 colorable.

A. →

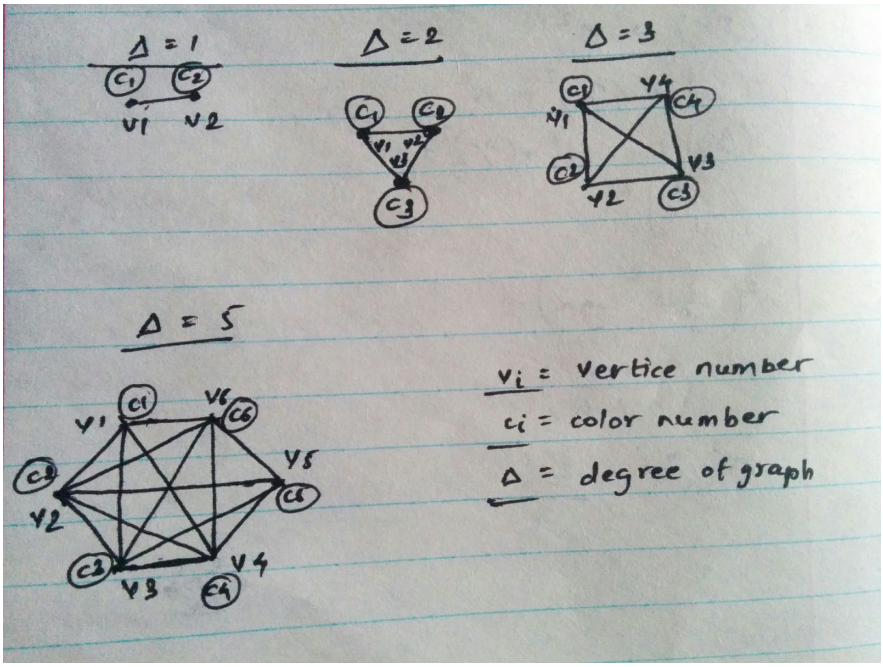
Basic Logic –

Construct a *connected graph* containing Δ + 1 vertices such that each vertex is connected to every other vertex in the graph.

We require Δ edges coming out from each vertex to connect it to every other index in graph, thereby making the degree of graph as 'Δ'.

And as we proved in part 1 of this question we will require Δ + 1 colors to color this graph.

Please refer following image for further understanding.



Q2(b) Graph with O(n) edges and n vertices can be colored in $O(\sqrt{n})$

A. →

Assume, that we divide 'n' vertices into two parts, vertices in part 1 will have degree $\sqrt{n}$ whereas, part 2 will have degree smaller than $\sqrt{n}$.

Let`s assume that there are $i*\sqrt{n}$, vertices whose degree is $\sqrt{n}$ and to maintain the maximum number of edges O(n) remaining vertices should have degree which is very small than $\sqrt{n}$.

Because if it is in terms of $\sqrt{n}$, then total number of vertices required for part two will be in terms of $n\sqrt{n}$ which contradicts given statement that number of edges are of order O(n). Suppose remaining $(n - \sqrt{n})$ vertices has 'k' degree [$k << \sqrt{n}$], then total number of edges in the graph would be

$$i * \sqrt{n} * \sqrt{n} + (n - i\sqrt{n}) * k$$
$$= i*n + n*k - i*k*\sqrt{n}$$
$$= n* i - i\sqrt{n}(1 - k)$$
$$= O(n)$$

Now, as defined in the first part of this question, vertices with degree $\sqrt{n}$ can be colored in total $\sqrt{n} + 1$ colors. And the remaining vertices will require at max k + 1 colors.

So, total number of distinct colors required at maximum would be

$$= \sqrt{n} + 1 + k + 1$$
$$= O(\sqrt{n}) \quad \text{… as } k << \sqrt{n}$$

(Note that we will not actually require k + 1 colors for the part 2, as we can make use of colors from part 1, however in any case number of colors required for part 1 would be dominant)

Q3 Number of shortest paths in social networks.

A. → Logic -

1. This problem is solved using 'breadth first search' technique.
2. Algorithm must save the depth and count obtained to reach to that vertex.
3. Then algorithm will breadth first search till we don't find the destination vertex.
4. Once we found the destination vertex, will save the depth as minHops of that node and then algorithm will explore remaining vertices only till depth minHops is not crossed.
5. If while exploring we found already visited vertex, then algorithm will check the depth of new vertex and old vertex and will increase the count only if newly found vertex has same depth otherwise new vertex will be discarded as it is.
6. Algorithm will end only once all vertices of depth minHops (depth of vertex when reached for the first time) have been explored or if destination is not found and all vertices have been explored.

Algorithm –

```
1. Vertex      {     name, depth, count        }
2. Algorithm(firstVertexName, destinationVertexName)   {
        count = 1,
        minHops = INF,
        currentDepth = 0,
        maxShortestPaths = 0,
        firstVertex       =        { firstVertexName,  currentDepth, count }
        listToExplore = {}        // can use Queue instead of list
        listToExplore.add(firstVertex)
        while(minHops >= currentDepth && listToExplore.size!=0)       {
                Vertex v= listToExplore.remove[0]
                currentDepth = v. depth
                if (currentDepth > minHops)      {
                        return maxShortestPaths;
                }//end if
                if (v.name = destinationVertexName)     {
                        minHops = v.depth
                }//end if
                for each neighbor nvi of v          {
                        if (nvi  present in listToExplore) {
                                if (nvi.depth > old_nvi.depth)     {
                                        continue; }
                                else      {
                                        old_nvi.count++}
                        }//end while
                        else      {
                                Vertex nv = { nvi, v.depth + 1, v.count }}
                }//end for
        }//end while
}//end algorithm
```

Complexity -

1. Worst case for the above algorithm would be in case when destination vertex is the last vertex or it is not present in the graph.
2. In the worst case the algorithm would end up exploring neighbors of all vertex in graph. Thus, in the end algorithm would travel through all vertices (n) using all the edges (m).
3. Hence, complexity of algorithm would be O(m + n)

Q4 A walk through entire graph. (Collaborator : Jhalak, Prasad)

A. →

Basic Logic –

1. This problem is solved using 'depth first search' technique.
2. Algorithm must save the depth obtained to reach that vertex.
3. Algorithm will start exploring a neighbor of given vertex that neighbor with height is added in visited vertex list and add edge from parent to that child is added in the edge traversal list.
4. If while exploring we found already visited vertex, then algorithm will check the depth of new vertex and old vertex, if depth of new duplicate vertex is greater than algorithm will add edges in both direction in the edge traversal list.
5. Algorithm will end only once all vertices have been explored.

Algorithm –

```
Function (Graph g, Vertex v)    {
        visitedVertexList.add(v)
        for each neighbor nvi of v   {
                if (nvi not in visitedVertexList)     {
                    depth[nvi]= depth[v]+1
                    traversalList.add(edge{'v', 'nvi'})
                    Function (g, nvi)
                    traversalList.add(edge{'nvi', 'v'})
                }// end if for not found
                else if (depth[nvi] > depth[v])       {
                    traversalList.add(edge{'v', 'nvi'})
                    traversalList.add(edge{'nvi', 'v'})
                }// end if for found
        }
}
```

Complexity -

1. Worst case for the above algorithm would be in case when destination vertex is the last vertex or it is not present in the graph.
2. In the worst case the algorithm would end up exploring neighbors of all vertex in graph. Thus, in the end algorithm would travel through all vertices (n) using all the edges (m).
3. Hence, complexity of algorithm would be $O(m + n)$