Khiladi 1 : Kishan Sinha
2019428
Khiladi 2 : Abhishek Chaturvedi
2019401

PROBLEM 1:

Code:

```
#include<bits/stdc++.h>
using namespace std;


map<string, vector<string>> graph;

stack<string> st;

map<string, int> visited;


void TopoSort(string s) {

    visited[s] = 1;

    for (string c : graph[s]) {

        if (visited[c] == 0) {

            TopoSort(c);
        }
    }

    st.push(s);
}


int main() {

    // Enter the number of organisms
```

```
    int n;
    cin >> n;

    // Initialising the graph

    for (int i = 1; i <= n; i++) {

        string o = "o";
        o += to_string(i);

        string e = "e";
        e += to_string(i);

        graph[o].push_back(e);
    }

    // Enter the number of organisms (m) which became extinct before the
other

    int m;
    cin >> m;

    // Enter the organism number which became extinct and before which
organism number in this order (m times)

    for (int i = 1; i <= m; i++) {

        int s1;
        cin >> s1;

        int s2;
        cin >> s2;

        string e = "e";
        e += to_string(s1);

        string o = "o";
        o += to_string(s2);

        graph[e].push_back(o);
    }
```

```
    // Enter the number of organisms (k) which overlapped

    int k;
    cin >> k;

    //Enter the organism numbers which overlapped (k times)
    for (int i = 1; i <= m; i++) {

        int s1;
        cin >> s1;

        int s2;
        cin >> s2;

        string e1 = "e";
        e1 += to_string(s1);

        string o2 = "o";
        o2 += to_string(s2);

        string e2 = "e";
        e2 += to_string(s2);

        string o1 = "o";
        o1 += to_string(s1);

        graph[o2].push_back(e1);

        graph[o1].push_back(e2);
    }

    for (int i = 1; i <= n; i++) {

        string o = "o";
        o += to_string(i);

        if (visited[o] == 0) {

            TopoSort(o);
        }

        string e = "e";
        e += to_string(i);
```

```
        if (visited[e] == 0) {

            TopoSort(e);
        }
    }

    while (!st.empty()) {

        cout << st.top() << " ";

        st.pop();
    }
}
```

According to the question we need to order the origin time and existence time for organisms according to some constraints. By analysing the constraints it can be seen that if two organisms overlap and it means that each organism came into existence before the other got out of existence, for example if s1 and s2 overlap then o1<e2 and o2<e1, this condition satisfies the overlap requirement. When an organism became extinct before the other originated, this means that the first's extinction time was less than the origin time of the other. For example, s1 became extinct before s2 originated then s1<o2.
Also, there was one additional basic condition for every organism that ex>ox for any oragism numbered x 1<=x<=n (no of organism). Since the extinction time for an organism is always greater than its origin time.
The above conditions mentioned also need to be followed in ordering and it can be easily achieved by topological sort on a directed acyclic graph.
The conditions make up the edges of the graph while the origin and extinction time (o1,o2,o3….,e1,e2…...) make up the vertices.
The condition that ox<ex can be achieved in the ordering by making an directed edge from vertex ox to ex for every condition of the type ox<ex and then applying the topological sort on the graph. The topological sort will ensure that ex comes after ox, because of its property that for an directed edge ox-ex ex will always come after ox in the sorted sequence.
Also, since we need to maintain the conditions in one direction the directed edges will work. For example - o2<o1 means o1's timestamp was more than o2's and it cannot happen the other way round.
The graph was made using the adjacency list concept using vectors.

Proof of Toposort:
In the function TopoSort(G) we are running a DFS and then store the string in a stack which maintains the order in top down manner.

Theorem: TopoSort(G) is run on a graph of strings called G. If G contains an edge (o, e), then the order of o is lesser than the order of e; in other words, o happened earlier than e, i.e., in the stack that we are returning, the order of the elements are in the same order where for every i , oi < ei

Proof: Consider the edge (o, e). When this edge is explored, **e** cannot be GRAY, because in that case **e** would be an ancestor of **o** and (o, e) would be a back edge meaning a cycle would exist but the graph is acyclic.
So **e** has to be either BLACK or WHITE. If **e** is WHITE, it becomes a child of **o**, and it occurs before **e** occurs. If **e** is BLACK, it's already finished so **o** will occur before **e** in the final answer. Thus for any edge (o, s) we have that **o occurs before e.**
Since DFS explores the children of nodes first, therefore we add the children in the stack first and then we add the current node. Therefore the children lie below the parent node so we print the stack in the top down manner (parent node will be printed before the children node) which is the required ordering of the topological sort. Therefore, for every edge ox->ex, ox appears before ex in the ordering.

Time Complexity: Time complexity will be the same as the TC of a DFS which is O(V + E). Number of vertices is 2n since for every organism we have two vertices (total n organisms are considered) and maximum number of edges can be n-1. Hence, asymptotic TC is **O(n)**

PROBLEM 2 :

Subproblem: **dist(i,j)** gives the minimum distance from the source to the vertex **i** using a maximum of **j** edges

Recurrence Relation: If **u** and **v** are two vertices connected by an edge of weight 'w', then:

dist(v,k) = min(dist(v,k-1) ,dist(u,k-1) + w))

Code:

```
#include<bits/stdc++.h>
using namespace std;

int main() {
```

```
    // cout<<"Enter the number of vertices"<<endl;

    int n;
    cin >> n;

    // cout<<"Enter the number of edges"<<endl;

    int e;
    cin >> e;




   // the graph

    vector<pair<pair<int, int>, int>> edges;
     // Taking the graph input

    for (int i = 1; i <= e; i++) {

        // cout<<"Enter the src, dest and weight of each edge"<<endl;

        int u, v, wt;

        cin >> u >> v >> wt;

        edges.push_back({{u, v}, wt});
    }

    // Store the distance for n vertices with i vertices 1<=i<=n-1
dist[i][j] = min distance of vertex i using <= j edges

    vector<vector<int>> dist(n + 1);

    for (int i = 1; i <= n; i++) {

        for (int j = 0; j <= n - 1; j++) {

            dist[i].push_back(INT_MAX);
        }
    }

    // cout<<"Enter the source"<<endl;
```

```
    int source;
    cin >> source;

    for (int j = 0; j <= n - 1; j++) {

        dist[source][j] = 0;
    }

    for (int k = 1; k <= n - 1; k++) {

        for (auto edge : edges) {

            int u = edge.first.first, v = edge.first.second, weight =
edge.second;

            if (dist[u][k - 1] == INT_MAX) {
                continue;
            }

            dist[v][k] = min(dist[v][k - 1], dist[u][k - 1] + weight);
        }
    }

    for (int i = 1; i <= n; i++) {

        if (i == source) {
            continue;
        }

        for (int k = 1; k <= n - 1; k++) {

            if (dist[i][k] == dist[i][n - 1]) {

                cout << "Best path from source to vertex " << i << " is
distance = " << dist[i][n - 1] << " and number of edges = " << k << endl;

                break;
            }
        }
    }
return 0;
}
```

Proof of Correctness: The algorithm we are using is the Bellman Ford algorithm to compute minimum distances and then we find the best path.

Claim: dist(v,n-1) is the shortest path from a source **s** to **v** for all vertices **v (i.e., the path with minimum weight)** .

We will prove this by induction.

To Prove: dist(v,k) is the minimum weight of a path from s to v that uses maximum k edges.

Base Case: If k = 0, then dist(v,k) = 0 if 's' and 'v' are the same (source same as destination) . Also dist(s,j) = 0 for any j <= n-1 (source same as destination).

Inductive Step: Assuming that for all vertices 'u', dist(u,k-1) is the minimum weight of a path from s to u that uses maximum k-1 edges.
Let X be the best path from source to v with <=k edges with u being the node just before v. If the shortest path from s to 'u' is called Y, then Y will have <= k-1 edges and this path must the shortest from s to u, because: if this path was any less than k-1 edges we could replace path X with Y + weight(u,v) which would contradict our assumption.
Now since this weight w(u,v) can be negative we take the best of cases and assign dist(v,k) as minimum of either the shortest path with k-1 edges i.e., dist(v,k-1) and the path Y + w(u,v) i.e., dist(u,k-1) + w(u,v) in every iteration.

Now since dist(v,n-1) depends on dist(v,k) where k < n-1 (which is already true according to our assumptions).
Therefore this algo works correctly.

Now that we have correctly calculated the minimum distance for every vertex using at most n-1 edges we know that the final minimum distance for every vertex from source is d[v][n-1] for every vertex v. Also, since we need to find the best path for every vertex that is the path with minimum distance from source which can be achieved by using minimum number of edges therefore, we can easily find that by comparing the values of d[v][i] with d[v][n-1] and running i from 1 to n-1. So, the first instance we get where d[v][n-1]=d[v][i] we get the answer for that vertex that is the minimum distance can be achieved with i edges. So our answer becomes i for that particular vertex and we can do this for every vertex to get the best path.

Time Complexity: There is a nested loop. The outer loop runs from 1 to n-1 --- corresponds to O(n).
The inner loop iterates over all the edges --- which O(m) complexity.
The overall complexity is O(mn) asymptotically.