Q1) PARTY TIME
Topic: Non Standard Knapsack type DP

Your favorite ADA TA is back with yet another challenge. His birthday is quite near, so he is planning to throw a birthday party for all the ADA students. For the party, he has stored some special juice (yes, only juice) in N containers of different capacities y_i in his room. Each container already contains some x_i juice. As your TA has a small room (just like you), he wants to minimize the number of containers used to store the special juice.

In order to minimize the number of containers, you need to pour the special juice from one container to another optimally to minimize the number of containers used (Only containers with non-zero amounts would be considered as used). Let's say if you pour z amount of special juice from container A to container B, then your effort score will increase by z amount.

Hence, the challenge for you is to minimize the number of containers as well as minimize the effort score e to transfer the special juice from N containers to only c container(s) (c is the minimum number of container required to store the special juice) to help your favorite TA organize his birthday party nicely.

Note: xi is always less than or equal to yi, meaning the amount of juice stored will always be less than or equal to the capacity of the container.

Input and constraints:

N $(1 \leq n \leq 100)$
Next line has $x_1, x_2 \ldots x_N$ that is N integers $(1 \leq x_i \leq 100)$
Next line has $y_1, y_2 \ldots y_N$ that is N integers $(1 \leq y_i \leq 100)$
Also, $(1 \leq x_i \leq y_i \leq 100)$

Output:
In single line, output c(minimum number of containers used) and e(minimum effort required to transfer special juice to the minimum number of containers)

Test case:
Input
4
3 3 4 3
4 7 6 5

Output
2 6
Answer
2 6

Solution:

Subproblem definition:

Let dp[i][j] represent the minimum effort required to transfer the special juice to j containers, considering only the first i containers.

The subproblem is quite tricky, read below to understand it further:

Maximizing the amount of juice stored: This is the primary goal of the problem - to maximize the amount of juice stored in the minimum number of containers. By maximizing this amount, we ensure that we're using the containers efficiently and minimizing the waste.

Minimizing the effort required: While maximizing the juice stored, we are indirectly minimizing the effort required to transfer the juice. This is because if we can store more juice in fewer containers, it means fewer transfers are needed. Each transfer incurs an "effort cost" equal to the amount of juice transferred. By maximizing the amount of juice stored in each step, we inherently minimize the number of transfers required, and thus, minimize the total effort.

So, even though we're technically maximizing the amount of juice stored in each container, the ultimate effect is to minimize the overall effort needed to transfer the juice to the minimum number of containers. Therefore, the subproblem is framed in terms of minimizing effort.

Recurrence relation:

dp[i][j]=max(dp[i−1][j−1],dp[i−1][j−y[i]]+x[i]), here i varies from 1 to {maximum capacity available in minimum possible containers to store the total juice} and j varies from 1 to {minimum number of containers required to store the special juice}.

Base cases:

- dp[0][0] = 0
- dp[i][0] = -infinity for i > 0
- dp[0][j] = -infinity for j > 0

Algorithm description:

Read input N and the arrays x and y. Sort the containers in non-increasing order of their capacities. Calculate the total amount of juice (suma). Find the minimum number of containers (k) required to store the total amount of juice. Initialize dp array with -∞. Initialize dp[0][0]=0. Loop over the containers from 1 to N: Loop over the sum of capacities from sumb to a[i].bb: Loop over the possible number of containers from k to 1: Update dp[j][kk] using the recurrence

relation. Find the maximum value of dp within the range suma to sumb. Output k and suma-maximum value of dp.

Time Complexity: O(N^2*C)

Code for reference:

```
#include <bits/stdc++.h>
using namespace std;
int n,dp[10005][105];
struct sss{
    int aa,bb;
}a[105];
bool cmp(sss x,sss y){
    return x.bb>y.bb;
}
int main(){
    scanf("%d",&n);
    int suma=0;
    for(int i=1;i<=n;i++){
        scanf("%d",&a[i].aa);
        suma+=a[i].aa;
    }
    for(int i=1;i<=n;i++){
        scanf("%d",&a[i].bb);
    }
    sort(a+1,a+n+1,cmp);
    int k=0,sumb=0;
    while(sumb<suma){
        sumb=sumb+a[++k].bb;
    }
    memset(dp,0x80,sizeof(dp));
    dp[0][0]=0;
    for(int i=1;i<=n;i++){
        for(int j=sumb;j>=a[i].bb;j--){
            for(int kk=k;kk>=1;kk--){
                dp[j][kk]=max(dp[j][kk],dp[j-a[i].bb][kk-1]+a[i].aa);
            }
        }
    }
    int nmax=-1;
    for(int i=suma;i<=sumb;i++){
        nmax=max(nmax,dp[i][k]);
```

```
    }
    printf("%d %d",k,suma-nmax);
}
```

Q2) THE TELEPORTATION MACHINE
topic: DP on bitwise OR

Tony Stark has created another breakthrough. He has created a teleportation machine which you can use to teleport anywhere. The machine has 2 modes (1 and 2). Mode 1 means that the machine is not ready for teleportation and is doing some computation, and mode 2 means it is ready. The machine automatically alternates between modes 1 and 2 without any human intervention. You can use the following steps to use the machine:-

- If the machine is in mode 1 it will output a no. **x** . You need to store this number with you (maybe append it to an array).

- If the machine is in mode 2, it will show a number (say **y**). Now you have to enter the details of where you want to go and along with it a special key. This key is calculated as the total count of subsequences in your array having bitwise OR = y .

At any time the array with you is nothing but the list of numbers generated by the machine in mode 1 till that time. In this problem, you basically need to give the special key every time the machine is in mode 2 and you have to do modulo **10^9 + 7.**

Input and Output:
More formally we can formulate the machine's behaviour as follows:-

It generates two types of queries **M=1** and **M=2** and a no. **x** (or **y** for query of second type) associated with each query.
The machine will ask **Q** queries in total.

When **P=1** you need to append **x** into your array

When **P=2**, you need to find the special key using the above algorithm (and using **y**) and print it **modulo 10^9 +7** . (modulo is % operator) .

Constraints: Q < =10^5 and 0<= x <=1023 and P=1 or P=2

Testcase:
6
2 0
2 1
1 2
1 2
2 2

2 1000

outputs:
1
0
3
0

Solution:

The contraints give an idea how to solve the problem , we can find a **O(Q.x)** as that will have **10^8** no. of ops and this can be handled by **c++ in 1 sec.**

We just have to find the count of all subsequences such that bitwise OR of all its numbers is equal to x. We can create a dp table for this.

Subproblem: **dp[i][x]=** count of subsequences of arr[0..i] having **bitwise OR** = x.

We can handle each query in O(1) later on once we have this dp table.

Recurrence: for the current index i, you can either consider the current element of the array or not consider it for the calculation of the bitwise OR.
Since 0<=x<=1023, you can do bitwise OR of the current element with all possible values of x on the previous index and add this in the corresponding count.

Following are the two recurrences:-

- **dp[i][j | arr[i]] = (dp[i][j | arr[j]] + dp[i-1][j])% m**, where | represents the bitwise OR and you consider the current element in the calculation. (0<=j<=1023) and m is 10^9+7

- **dp[i][j] = (dp[i][j] + dp[i-1][j])%m** , when you dont consider the current element in the calculation

Do modulo after calculating the sum at each point as (a+b)%m=a%m +b%m

Base Case: **dp[0][0]=1** as bitwise OR of empty sequence is 0 and ; **dp[0][j]=0** where j>0. Initialize the remaining dp table with 0.

Then store all the queries in an array in the format {P,x} ;if P=1 add x to the array and if P=2 print **dp[L-1][x]** where L is the length of the current array

CODE FOR REFERENCE

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

typedef long long int ll;
#define pb push_back
#define mod 1000000007

int main()
{

   ios_base::sync_with_stdio(false);
   int q,A,x;


    cin>>q;
    vector<int> arr;
    arr.pb(0);
    vector<pair<int,int>> questions;
    for(int i=0;i<q;i++)
    {
        cin>>A>>x;
        if (A==1) arr.pb(x);
        questions.pb({A,x});
    }

    int n=arr.size()-1;

    vector<vector<int>> dp(n+1,vector<int> (1024,0));
    // dp[i][j]= no. of subsequences of arr[0..i] having bitwise or = j

    dp[0][0]=1;

    for(int i=1;i<=n;i++)
    {
        for(int j=0;j<1024;j++)
        {
            dp[i][j]=(dp[i][j]+dp[i-1][j])%mod;
            dp[i][arr[i]|j]=(dp[i][arr[i]|j]+dp[i-1][j])%mod;
```

```
        }
    }

    int idx=0;
    for(int i=0;i<questions.size();i++)
    {
        if (questions[i].first==1) idx++;
        else cout<<dp[idx][questions[i].second]<<endl;
    }


}
```

Q3) Artist's dilemma
topic: DP on graphs
~~You are the sole survivor of the dungeon of strings. Now, since ...~~

Ok. Let us snap back to reality for once. Simon is your best friend. Since his birthday is approaching, you are preparing a gift for him. You have decided to give him a painting as he loves art, but you only know data structures and algorithms. So, you decided to take upon your favourite data structure(graph) and draw some mysterious animal using it.

The animal will have a face and many hands. Given this, the beauty of a graph is described as the product of the number of **vertices** used for making the face and the number of hands. Now, the condition for an edge to be a part of the face is as follows:

- The number marked on the vertices in the face must increase strictly as you move forward in one direction.
- The face can't have multiple connected components. This means that the vertices that are a part of the face will have to be connected.

For the hands, you decide that a hand will be a single edge such that one of the vertices in the edge is an **endpoint** for the face. Now, what is the maximum possible beauty of the art?
Note:
It is given that two distinct edges don't connect the same pair of points. Also, there is no edge from a vertex to itself.
Assume you can take an edge in both face and hand if it makes the art more beautiful.
Hint
Check the test case below and try making your own test cases as well.

Solution:

Once you understand what the question wants to ask of you, it is not as difficult anymore. You have to understand that the number of hands for any vertex is actually known to us. It will be just the size of the adjacency list of the respective vertices.

Now that you know this, we just need to find out the longest increasing sequence for every vertex when that vertex is the endpoint.
To do this, you can apply traversal over vertices in increasing order. Maintain a dp[n] array that stores the above-mentioned value for all the vertices.

```
#include <bits/stdc++.h>
using namespace std;
#define all(x) (x).begin(), (x).end()
typedef long long ll;
const int maxN = 1 << 17;
int dp[maxN];
vector<int> g[maxN];
int main() {
    int n, m;
    cin >> n >> m;
    while(m--) {
        int v, u;
        cin >> v >> u;
        g[v].push_back(u);
        g[u].push_back(v);
    }
    ll ans = -1;
    for (int v = 1; v <= n; v++) {
        dp[v] = 1;
        for (auto u : g[v]) {
            if (u < v) {
                dp[v] = max(dp[v], dp[u] + 1);
            }
        }
        ans = max(ans, dp[v] * (ll)g[v].size());
    }
    cout << ans << endl;
    return 0;
}
```

Q4) THE COOPER-HOFSTADTER REALIZATION
topic: Modified Dijkstra on graphs

Sheldon Cooper and Leonard Hofstadter are two brilliant Physicists. They are trying to solve string theory and Sheldon finally has a way to solve a part of the problem.
So the entire world has been condensed in the form of a directed graph with **N** vertices and **M** edges. In the graph, edge i connects two different vertices **Ui** and **Vi** with a length of **Wi**. **Wi** represents that it takes **Wi** seconds to move from **Ui** to **Vi** (direction is important).

Sheldon is currently standing on vertex **1** and Leonard can currently be standing on any vertex from **2** to **N (** call his position **k) .** Now. each node of the graph has some knowledge which the person standing there can acquire.

Also, it is known that Sheldon and Leonard have traveled to various nodes and accumulated a lot of extra knowledge in addition to their original aptitude, and their positions described above are their current positions(i.e. after getting all the knowledge from different nodes).

Now Sheldon believes that in order to solve the problem the two of them need to meet, and in the shortest time possible. Also, he believes (like always) that he is the more important and intelligent member of the two and hence it doesn't matter where Leonard stands currently (which is why his position is variable and Sheldon's is fixed).

So for all **2<=k<=N (all possible positions of Leonard),** Sheldon wants to know the minimum time in seconds required for them to meet so that he can ask Leonard to modify his journey accordingly(such that Leonard is currently standing from where he can reach Sheldon in minimum time and the problem is solved !!)

Input and Output:
More formally you need to print a line containing N−1 integers. The j-th integer represents the minimum time in seconds needed by the two of them to meet if initially Sheldon is standing on vertex 1 and Leonard on vertex j+1. Print −1 if it is impossible for them to meet according to the current configuration.

Constraints:
2<=N<=10^5 and 0<=M <=2.10^5 ; also 1<=Ui, Vi<=N , Ui != Vi , 1<= Wi <=10^9

Testcases:
5 7

1 2 2
2 4 1
4 1 4
2 5 3
5 4 1
5 2 4
2 1 1

Outputs:
1 -1 3 4

Solution:-

Suppose we are trying to find out the minimum time for Sheldon and Leonard to get from vertices **1** and **k (2<=k<=N)** respectively to the same vertex. If both end up on some vertex **v,** then the time required is **d(1,v)+d(k,v)**, with **d(x,y)** being the minimum time to go from vertex **x** to **y.**

Suppose **d′(x,y)** is the minimum time to go from vertex **x** to **y** in the reversed graph (i.e. all edges' directions are reversed). Then **d(1,v)+d(k,v)=d(1,v)+d′(v,k).**

The minimum time if both are initially on vertices **1 a**nd **k** respectively, is the minimum value of **d(1,v)+d′(v,k)** for all vertices **v.** This is the same as the minimum time to go from vertex **1** to **k** where in the middle of our path, we can reverse the graph at most once.

Therefore we can set up a graph like the following:

- Each vertex is a pair **(x,b),** where **x** is a vertex in the original graph and **b** is a boolean determining whether we have already reversed the graph or not.
- For each edge **i** in the original graph, there is an edge from **(Ui,0)** to **(Vi,0)** and an edge from **(Vi,1)** to **(Ui,1)**, both with weight **Wi.**
- For each vertex x in the original graph, there is a edge from **(x,0)** to **(x,1)** with weight **0.**

After this, we do the Dijkstra algorithm once on the new graph from vertex (1,0). Then, the optimal time if both start from vertices 1 and k in the original graph is equal to **d((1,0),(k,1))** in the new graph.

Time complexity: O(N+MlogM)

<u>CODE FOR REFERENCE</u>

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long int ll;
#define INF LONG_LONG_MAX
#define pb push_back


void dijkstra(vector<ll> &dis,vector<vector<pair<ll,ll>>> &adjlist,ll src,ll n)
{
    ll u,v,w,d;
    pair<ll,ll> curr_min;
    priority_queue<pair<ll,ll>,vector<pair<ll,ll>>,greater<pair<ll,ll>>> pq; //
min priority queue to get current min in (log n) time
    pq.push({0,src});

    while(!pq.empty())
    {
        curr_min=pq.top();
        pq.pop();
        u=curr_min.second; // (x,y) is distance , vertex pair in priority queue.
        d=curr_min.first;

        if (d!=dis[u]) continue;

        for (ll i=0;i<adjlist[u].size();i++)
        {
            v=adjlist[u][i].second;
            w=adjlist[u][i].first;

            if (dis[u]+w<dis[v])
            {
                dis[v]=dis[u]+w;
                pq.push({dis[v],v});
            }
        }
    }
}

int main()
```

```
{

    ios_base::sync_with_stdio(false);
    ll n,m,u,v,w;

    cin>>n>>m;
    vector<vector<pair<ll,ll>>> adjlist(2*n+1); // 1 to n is normal graph and n+1
to 2*n is reverse graph corresponding to vertices (1 to n)

    for(ll i=0;i<m;i++)
    {
        cin>>u>>v>>w;
        adjlist[u].pb({w,v}); // (u,0) -> (v,0)
        adjlist[v+n].pb({w,u+n}); // (v,1)-> (u,1)
    }
    for (ll i=1;i<=n;i++)
    {
        adjlist[i].pb({0,i+n}); // (u,0) -> (u,1) with weight 0
    }


    vector<ll> dis(2*n+1,INF);
    dis[1]=0;
    dijkstra(dis,adjlist,1,n);

    for (ll i=1;i<=2*n;i++)
    {
        if (dis[i]==INF) dis[i]=-1;
    }
    for (ll i=2;i<=n;i++)
    {
        cout<<dis[i+n]<<" ";  // final ans is dis((1,0) , (i,1))
    }

}
```

Q5) RIISE ONCE AGAIN
topic: Graph Traversals, Cycle detection

Just few days ago, we completed RIISE'24, the annual flagship event of IIIT Delhi, where many high-profile individuals from industry and academia came together. Let's imagine that you are the Events head for next year's RIISE'25. All the professors have given you a long list of tasks that need to be completed in N different rooms of our large campus connected by M one-way connections (yes, a directed graph, you are smart). You have called and assembled the entire team in Room 1. Before dispersing the team to different rooms, it is important to estimate the number of paths from Room 1 to all possible N rooms, as the distribution of work depends on the number of paths to each room from Room 1.

A path from any Room X to any Room Y is a sequence of connections where Room X should be at the start of the first connection of the path and Room Y is at the end of the last connection of the path. The next connection starts at the Room where the previous connection ends within the path. It is important to keep in mind that it is possible that there would not be any path from Room 1 to any Room X, or there can be exactly 1 path from Room 1 to any Room X, or there can be more than 1 path from Room 1 to Room X, or there can be an infinite number of paths from 1 to Room X.

If there is no path from Room 1 to any Room X, then return 0 for that room. If there is exactly 1 path from Room 1 to any Room X, then return 1 for that room. If there are more than 1 but a finite number of paths from Room 1 to any Room X, then return 2 for that room. If there are more than 1 but an infinite number of paths from Room 1 to any Room X, then return -1 for that room.

Note: Self-loops are possible, meaning connections from Room X to Room X are allowed. Multiple connections between rooms, let's say X and Y, are not possible; hence, at most 1 connection is allowed between any Room X and Room Y.

Input and Output:
The first line contains T test cases.
For every t test case:
The first line contains two integers N and M. The next M lines contain 2 integers representing connections between rooms.

Constraints:
t ($1 \le t \le 10^4$)
N and M ($1 \le N \le 4 * 10^5$, $0 \le M \le 4 * 10^5$)
The next line contains Xi and Yi integers where i varies from 1 to M. ($1 \le Xi, Yi \le N$)

Solution:

The first motivation for solving this problem is to write a lot of standard code like "find strongly connected components", do some DP over the condensed graph (the graph of strongly connected components), and so on.

In fact, this problem can be solved much more elegantly with less code if you have a little better understanding of how depth-first search works.

Consider a usual depth-first search on a digraph that is started from the vertex 1
. This will be a normal depth-first search, which will paint vertices using three colors: white (the vertex has not yet been found by the search), gray (the vertex is processing by DFS), and black (the vertex has already been processed by the DFS completely, that is, completely bypassed its subtree of the depth-first search tree).

Here's the pseudocode:

```
dfs(u):
    color[u] = GRAY
    for v in g[u]:
        if color[v] == WHITE:
            dfs(v)
    color[u] = BLACK
```

The following statements are true:

there is a cycle in the digraph reachable from s
 if and only if the root call dfs(s) visits in the line if color [v] == WHITE when color[v] == GRAY;
moreover, for each reachable cycle from s
 there is at least one vertex that will execute the previous item (then the vertex v
 belongs to the cycle);
if the root call dfs(s) visits in the line if color[v] == WHITE, when color [v] == BLACK, then there is more than one path (the opposite is not true).
It is clear that there are infinite paths from s
 to u
 if and only if there is a vertex v
 on some path from s
 to u
 such that v
 is in a cycle. Thus, we mark all such vertices v
 for which color[v] == GRAY at the moment of execution of the line if color [v] == WHITE. The fact is true: All vertices reachable from those noted in the previous phrase are those and only those vertices that an infinite number of paths lead to them.

A similar fact is also true for finding vertices to which at least two paths (but a finite number) lead. Let's mark all such vertices v
 for which color[v] == BLACK at the moment of execution of the line if color [v] == WHITE. The fact is true: Let's find all reachable vertices from those marked in the previous phrase. Let's discard those up to which we have already determined that the number of paths is infinite. The remaining reachable vertices are those and only those to which there are at least two paths (and their finite number).

So the solution looks like this:

let's make a depth-first search from the root, mark during it those vertices that were gray when trying to go to them (group A) and were black when trying to go to them (group B);
mark the vertices reachable from the group A (let's call them AA);
mark the vertices reachable from the group B (let's call them BB);
the answer for the vertex is:
0
 if it is not reachable from s
 (this determines the first DFS);
−1
, if it is from AA;
2
 if it is from BB (but not from AA);
1
, if it is not from AA and not from BB.

Time complexity: O(N+M)

Code for reference:

```
#include <bits/stdc++.h>

using namespace std;

#define forn(i, n) for (int i = 0; i < int(n); i++)

int n;
vector<vector<int>> g;
set<int> s[2];

void dfs(int u, vector<int>& color, bool use_s) {
```

```
    color[u] = 1;
    for (int v: g[u])
        if (color[v] == 0)
            dfs(v, color, use_s);
        else if (use_s)
            s[color[v] - 1].insert(v);
    color[u] = 2;
}

int main() {
    int t;
    cin >> t;
    forn(tt, t) {
        int m;
        cin >> n >> m;
        g = vector<vector<int>>(n);
        forn(i, 2)
            s[i] = set<int>();
        forn(i, m) {
            int x, y;
            cin >> x >> y;
            x--, y--;
            g[x].push_back(y);
        }
        vector<int> color = vector<int>(n);
        dfs(0, color, true);
        vector<vector<int>> c(2, vector<int>(n));
        forn(i, 2)
            for (auto u: s[i])
                dfs(u, c[i], false);
        forn(i, n) {
            int ans = 0;
            if (color[i] != 0) {
```

```
                ans = 1;
                if (c[0][i])
                    ans = -1;
                else if (c[1][i])
                    ans = 2;
            }
            cout << ans << " ";
        }
        cout << endl;
    }
}
```
