



# NOI STUFF

Clear everything by February. If I can use all the things well, I can get a gold. After February, try not to learn new concepts since you will want to use it during competition. But since you are not adept at it, it will be a waste of time.

## [Tutorial] Everything about unordered\_map - Codeforces

UPD: Tricks to make unordered\_map faster added. Hi! What is unordered\_map? It is a data structure like map but it is more than 4 times faster than map. you can use it in C++11 with including `#include <unordered_map>`. for example: `#include <unordered_map>` `int main(){ unordered_map mp; mp[5]=12; mp[4]=14; cout << mp[4]; }`

<https://codeforces.com/blog/entry/21853>

**CODEFORCES**  
Sponsored

## Introduction to Greedy Algorithms

Sometimes, if the algorithm is easy enough to implement, you don't even need to convince yourself it's correct; just code it and see if it passes. Competitive programmers refer to this as "Proof by AC," or "Proof by Accepted."

<https://usaco.guide/bronze/intro-greedy?lang=cpp>

**USACO  
Guide**

free, high quality resources

Questions:

- Can we use the internet during the competition? (Firefox is installed for some reason?) → NO
- Precompilers and all that, im sure you don't memorize it. What are some good practices and all.

**GET THE NEW IDE AND GET USED TO IT!!!**

### ▼ Dynamic programming

[https://www.youtube.com/watch?v=aPQY\\_2H3tE](https://www.youtube.com/watch?v=aPQY_2H3tE)

How to approach dynamic programming:

- 

### ▼ Advanced dynamic programming

Let  $f(C, D)$  be the number of ways to reach city  $C$  in  $D$  days. To reach  $f(C, D)$ , Kuno has to either reach city  $C$  in less than  $D$  days and wait for the remaining days, or he can reach city  $C$  in exactly  $D$  days. This allows us to form the recurrence function for  $f(C, D)$ .

Let  $S_C$  contain the list of flights with destination to city  $C$ ,  $(i, t)$ , where the flight departs from city  $i$ , and takes  $t$  time to fly.

$$f(C, D) = \begin{cases} 0, & \text{if } D < 0 \\ \min(500000001, f(C, D-1) + \sum_{(i,t) \in S_C} f(i, D-t)), & \text{otherwise} \end{cases}$$

This has  $O(nm)$  states, and takes an average of  $O(h)$  transition time. By using dynamic programming to memorize the answer for each state, we ensure that each state is only calculated once. This gives a total complexity of  $O(nmh)$ .

## Improved Algorithm

Recall that  $\text{Minadd}(x, y)$  denotes  $x = \min(x+y, 500000001)$ .

Read  $n, m, h$  and Initialise all entries  $a[i, p]$  with 0 except for  $a[0, 0]$  which is 1;

For  $i = 0, 1, \dots, n-2$  For each Flight leaving  $i$  Do Begin


Read next stop  $j$ , minimum stay  $k$ ;


If  $i < j$  Then Do Begin  $s=0$ ;

For  $p=0$  to  $m-1-k$  Do Begin  $\text{Minadd}(s, a[i, p]); \text{Minadd}(a[j, p+k], s)$  End End;

Output  $a[n-1, 0], a[n-1, 1], \dots, a[n-1, m-1]$ .

Journey
All Problems Newest Problems Unsolved Problems Recommended Problems



 <https://codebreaker.xyz/problem/journey3>

This question is pretty fucking insane.

The idea is forming a recurrence function and then using dp to optimise it.

The 'improved algorithm' is combinatory dp.

$s = \min(s + a[i][p], 5e8) \rightarrow$   $s$  slowly builds up btw, through the combination of  $a[i][p] \rightarrow$  ie  $s$  = sum of the ways in day  $p$ , city  $i$   
 $a[j][p+k] = \min(a[j][p+k] + s, 5e8) \rightarrow$  Using the previous  $s$  value, update the future  $a[j][p+k] \rightarrow$  ie adding the total number of ways you can get to city  $j$  in  $p+k$  days.

▼ Dijkstra/BFS tracking the nodes that is optimal

`add_counter()`  $\rightarrow$  recurring function, using parent to track the function

`memset(dis, 0x3F, sizeof(dis));` or `memset(-1)`

Another key thing to note about dijkstra/bfs shortest path is that it ONLY takes one of the possible path - ie if there is multiple paths, the calculations do not work. For a tree or a line, this is not an issue, as by definition it has no loop and thus will definitely have no multiple optimal paths.

**But for general graphs, this is the case - you can track the minimum distance, but it is significantly harder to keep track of all the possible shortest paths. Very troublesome.**

```
#include <bits/stdc++.h>
#include <vector>
#include <algorithm>
#include <set>
using namespace std;
#define int long long int
#define pq priority_queue
#define pii pair<int,int>
#define pb push_back

int n,m,q;

int dis[1005];
vector<int> adj[1005];
int counter[1005]; //all zero at first
int parent[1005];

void add_counter(int cur){
    counter[cur]++;
    if(parent[cur] == -1){
        return;
    }
    add_counter(parent[cur]);
}

void dijkstra(int start, int end){
    memset(dis, -1, sizeof(dis));
    memset(parent, -1, sizeof(parent));
    queue<int> bfs_q;

    dis[start] = 0;
    bfs_q.push(start);

    while(not bfs_q.empty()){
        int cur = bfs_q.front(); //choose the node
        for(int neigh : adj[cur]){
            if(dis[neigh] == -1){
                dis[neigh] = dis[cur] + 1;
                parent[neigh] = cur;
                bfs_q.push(neigh);
            }
            else if(dis[neigh] > dis[cur] + 1){
                dis[neigh] = dis[cur] + 1;
                parent[neigh] = cur;
                bfs_q.push(neigh);
            }
        }
        bfs_q.pop(); //pop the nodes
    }
    add_counter(end);
}

signed main(){
    cin >> n >> m;
    for(int i = 0; i < m; i++){
        int a,b;
        cin >> a >> b;
        adj[a].pb(b);
        adj[b].pb(a);
    }
    cin >> q;
    for(int i = 0; i < q; i++){
        int u,v;
        cin >> u >> v;
        dijkstra(u,v);
    }
    int max_ans = -1;
    int current_ans;
    for(int i = 0; i <= n; i++){
        if(max_ans < counter[i]){
            max_ans = counter[i];
            current_ans = i;
        }
    }
}
```

```

    cout << current_ans;
}

```

▼ SCC ie copy code → **Check and understand**

Kosaraju → For directed graphs

Again abusing ChatGPT: (check later)

Kosaraju's algorithm: This is a two-pass algorithm based on depth-first search (DFS). The first pass visits all vertices of the graph in a depth-first order and constructs a transpose graph. The second pass visits all vertices in the reverse order of the first pass and assigns them to the same component as the vertex that was visited first. The time complexity of Kosaraju's algorithm is also  $O(V + E)$ .

```

#include <iostream>
#include <vector>
#include <stack>

using namespace std;

const int N = 100;

vector<int> adj[N]; // original graph
vector<int> adjT[N]; // transpose graph
vector<vector<int>> scc; // vector of scc
stack<int> stk;
bool visited[N];

void dfs1(int u) {
    visited[u] = true;
    for (int v : adj[u]) {
        if (!visited[v]) {
            dfs1(v);
        }
    }
    stk.push(u);
}

void dfs2(int u) {
    visited[u] = true;
    scc.back().push_back(u);
    for (int v : adjT[u]) {
        if (!visited[v]) {
            dfs2(v);
        }
    }
}

void kosaraju() {
    // first DFS
    for (int i = 0; i < N; i++) {
        if (!visited[i]) {
            dfs1(i);
        }
    }

    // reset visited array
    for (int i = 0; i < N; i++) {
        visited[i] = false;
    }

    // second DFS
    while (!stk.empty()) {
        int u = stk.top();
        stk.pop();
        if (!visited[u]) {
            scc.push_back({});
            dfs2(u);
        }
    }
}

int main() {
    // Read the number of vertices and edges
    int n, m;
    cin >> n >> m;

```

```

// Read the edges of the graph
for (int i = 0; i < m; i++) {
    int u, v;
    cin >> u >> v;
    adj[u].push_back(v);
    adjT[v].push_back(u);
}

kosaraju();

// Print the strongly connected components
for (auto& component : scc) {
    for (int vertex : component) {
        cout << vertex << " ";
    }
    cout << endl;
}

return 0;
}

```

### ▼ Binary lifting with a lot of comments

```

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

const int nmax = 1e5;
int LN=17;
vector<int> adj[nmax];
int dp[nmax][17];
int level[nmax];
//the forloops, for some reason, MUST be ++i and --i. if it isn't, it does not work. bruh
//i think this is since for ++i and --i, the i variable is incremented BEFORE each iteration
//normally, there isnt an issue. but for this particular case, where the value of i and
//all that jazz is important, it is.
void dfs(int u, int p) {
    level[u] = level[p] + 1;
    dp[u][0] = p;
    for(int i=1; i<LN; ++i)
        dp[u][i] = dp[dp[u][i-1]][i-1];
}
/**
dp[u][i-1] represents the 2^(i-1)th ancestor of u.
dp[dp[u][i-1]][i-1] represents the 2^(i-1)th ancestor of the 2^(i-1)th ancestor of u.
dp[u][i-1] is the 2^(i-1)th ancestor of u and dp[dp[u][i-1]][i-1] is the 2^(i-1)th ancestor of the 2^(i-1)th ancestor of u, so
*/
for(int i=0; i<adj[u].size(); ++i) {
    int v = adj[u][i];
    if(v == p) continue;
    dfs(v, u);
}

int lca(int u, int v) {
    if(level[u] < level[v])
        swap(u, v);
    int diff = level[u] - level[v];
    for(int i=0; i<LN; ++i) {
        if((1<<i) & diff)
            u = dp[u][i];
    }
    /**
    For each power of 2, starting with the highest, the algorithm checks if
    the parent of the shallower node at that level (parent[shallowerNode][i]) is
    not equal to the parent of the deeper node at that level (parent[deeperNode][i]).
    If the two parents are not equal, it means that the shallower node can be moved up
    to that level by setting the shallower node to its parent at that level
    (shallowerNode = parent[shallowerNode][i]).

    The purpose of this step is to find a common ancestor for both nodes as fast as
    possible by jumping through the levels of the tree. The jumping is done by
    checking if the shallower node's 2^i-th ancestor is the same as the deeper node's
    2^i-th ancestor. If it is not, the shallower node is set to its 2^i-th ancestor, and
    this process continues until the shallower node's 2^i-th ancestor is the same as the
    deeper node's 2^i-th ancestor.

    IE some magical binary bullshitery
    */
    }
    if(u == v)
        return u;
}

```

```

for(int i=LN-1; i>=0; --i) {
    if(dp[u][i] != dp[v][i]) {
        u = dp[u][i];
        v = dp[v][i];
    }
}
return dp[u][0];
}

int main(){
    int n,m;
    cin >> n >> m; //n-1 vertices, m queries
    //note that the LN should literally just be log2(maxN)
    //not log2(n)
    for(int i =0;i<n-1;i++){
        int u,v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    dfs(1,0);
    for(int i =0;i<m;i++){
        int u,v;
        cin>> u >> v;
        cout << lca(u,v) << "\n";
    }
}

```

The jumping works with  $2^i$  in the binary lifting algorithm for finding the lowest common ancestor (LCA) because of the way the algorithm is designed. The algorithm uses a 2D array `dp` where `dp[u][i]` represents the  $2^{(i-1)}$ th ancestor of node `u`.

In the context of binary lifting for finding LCA, using  $2^i$  as the ancestor of a node `u` would mean that the node is  $i$  levels above the node `u`. It is similar to the concept of  $2^{(i-1)}$  ancestor that I explained earlier, the only difference is that instead of  $i-1$  level above, it would be  $i$  level above. The  $2^{(i-1)}$ th ancestor of a node `u` refers to the node that is  $2^{(i-1)}$  levels above `u` in the tree. For example, if the value of `i` is 2, then  $2^{(i-1)}$  is equal to 2, so the  $2^{(i-1)}$ th ancestor of `u` would be the node that is 2 levels above `u` in the tree.

This is because `dp[u][i]` is storing the  $2^i$ th ancestor of node `u`. Therefore, when we set `dp[u][i] = dp[dp[u][i-1]][i-1]`, we are essentially updating `dp[u][i]` to store the  $2^i$ th ancestor of node `u`, which is the  $2^{(i-1)}$ th ancestor of the  $2^{(i-1)}$ th ancestor of `u`.

1. `dp[u][i]` means the ancestor of `u` which is  $2^i$  above `u`
2. @January 28, 2023 5:49 PM@January 28, 2023 5:49 PM@January 28, 2023 5:49 PM  
so for example `dp[5][2]` is the ancestor of node 5 which is  $4=2^2$  above node 5 @January 28, 2023 5:49 PM
3. @January 28, 2023 5:49 PM@January 28, 2023 5:49 PM@January 28, 2023 5:49 PM  
to find that, you first jump to the ancestor of node which is  $2=2^1$  above node 5
4. @January 28, 2023 5:50 PM@January 28, 2023 5:50 PM@January 28, 2023 5:50 PM  
this will be `dp[5][1]` (`dp[u][i-1]` in the general form)
5. @January 28, 2023 5:50 PM@January 28, 2023 5:50 PM@January 28, 2023 5:50 PM  
then from `dp[5][1]` you go up another  $2^1$  steps @January 28, 2023 5:51 PM
6. @January 28, 2023 5:51 PM@January 28, 2023 5:51 PM@January 28, 2023 5:51 PM  
and now you are at `dp[5][2]`
7. @January 28, 2023 5:52 PM@January 28, 2023 5:52 PM@January 28, 2023 5:52 PM  
the purpose of the second piece of code is to lift the lower node so that both nodes (`u` and `v`) are at the same level
8. @January 28, 2023 5:53 PM@January 28, 2023 5:53 PM@January 28, 2023 5:53 PM  
(`1<<i`) & `diff` is to find the bits which are on for the binary representation of `diff`
9. @January 28, 2023 5:53 PM@January 28, 2023 5:53 PM@January 28, 2023 5:53 PM  
so for example `diff = 5`, meaning you want to go up 5 steps

10. @January 28, 2023 5:54 PM@January 28, 2023 5:54 PM@January 28, 2023 5:54 PM

you can break it down into go up  $1=2^0$  step, and then go up  $4=2^2$  step

11. @January 28, 2023 5:54 PM@January 28, 2023 5:54 PM@January 28, 2023 5:54 PM

so you are always jump a distance of  $2^j$ , which can be directly obtained from ur dp array

#### ▼ LCA Segment tree copying from g4g

### LCA using segment tree

The key here is noticing that the common parent is the lowest number in the range between values in a Euler tree traversal.

Then, code.

Our node is the node at the smallest level and the only node at that level amongst all the nodes that occur between consecutive occurrences (any) of u and v in the Euler tour of T.

1. Nodes visited in order of Euler tour of T
2. The level of each node visited in the Euler tour of T
3. Index of the **first** occurrence of a node in Euler tour of T (since any occurrence would be good, let's track the first one)

#### The first occurrences corresponding to every node in Euler tour of T:

Node	1	2	3	4	5	6	7	8	9
First Occurrence	0	1	11	2	4	12	14	5	7

#### An euler tour of the tree starting from node 1 will yield:

1	2	4	2	5	8	5	9	5	2	1	3	6	3	7	3	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

#### The corresponding levels for every node in Euler tour:

0	1	2	1	2	3	2	3	2	1	0	1	2	1	2	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

```
//(Note that LCA is useful for finding the shortest path between two nodes of a Binary Tree)
/* C++ Program to find LCA of u and v by reducing the problem to RMQ */
#include<bits/stdc++.h>
#define V 9 // number of nodes in input tree

int euler[2*V - 1]; // For Euler tour sequence
int level[2*V - 1]; // Level of nodes in tour sequence
int firstOccurrence[V+1]; // First occurrences of nodes in tour
int ind; // Variable to fill-in euler and level arrays

// A Binary Tree node
struct Node
{
    int key;
    struct Node *left, *right;
};

// Utility function creates a new binary tree node with given key
Node * newNode(int k)
{
    Node *temp = new Node;
    temp->key = k;
    temp->left = temp->right = NULL;
    return temp;
}

// log base 2 of x
int Log2(int x)
{
    int ans = 0 ;
    while (x>=1) ans++;
    return ans ;
}
```

```

}

/* A recursive function to get the minimum value in a given range
of array indexes. The following are parameters for this function.

st --> Pointer to segment tree
index --> Index of current node in the segment tree. Initially
        0 is passed as root is always at index 0
ss & se --> Starting and ending indexes of the segment represented
            by current node, i.e., st[index]
qs & qe --> Starting and ending indexes of query range */
int RMQUtil(int index, int ss, int se, int qs, int qe, int *st)
{
    // If segment of this node is a part of given range, then return
    // the min of the segment
    if (qs <= ss && qe >= se)
        return st[index];

    // If segment of this node is outside the given range
    else if (se < qs || ss > qe)
        return -1;

    // If a part of this segment overlaps with the given range
    int mid = (ss + se)/2;

    int q1 = RMQUtil(2*index+1, ss, mid, qs, qe, st);
    int q2 = RMQUtil(2*index+2, mid+1, se, qs, qe, st);

    if (q1==-1) return q2;

    else if (q2==-1) return q1;

    return (level[q1] < level[q2]) ? q1 : q2;
}

// Return minimum of elements in range from index qs (query start) to
// qe (query end). It mainly uses RMQUtil()
int RMQ(int *st, int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n-1 || qs > qe)
    {
        printf("Invalid Input");
        return -1;
    }

    return RMQUtil(0, 0, n-1, qs, qe, st);
}

// A recursive function that constructs Segment Tree for array[ss..se].
// si is index of current node in segment tree st
void constructSTUtil(int si, int ss, int se, int arr[], int *st)
{
    // If there is one element in array, store it in current node of
    // segment tree and return
    if (ss == se)st[si] = arr[ss];

    else
    {
        // If there are more than one elements, then recur for left and
        // right subtrees and store the minimum of two values in this node
        int mid = (ss + se)/2;
        constructSTUtil(si*2+1, ss, mid, arr, st);
        constructSTUtil(si*2+2, mid+1, se, arr, st);

        if (arr[st[2*si+1]] < arr[st[2*si+2]])
            st[si] = st[2*si+1];
        else
            st[si] = st[2*si+2];
    }
}

/* Function to construct segment tree from given array. This function
allocates memory for segment tree and calls constructSTUtil() to
fill the allocated memory */
int *constructST(int arr[], int n)
{
    // Allocate memory for segment tree

    // Height of segment tree
    int x = Log2(n)+1;

    // Maximum size of segment tree
    int max_size = 2*(1<<x) - 1; // 2*pow(2,x) -1

    int *st = new int[max_size];

```



```

// Fill the allocated memory st
constructSTUtil(0, 0, n-1, arr, st);

// Return the constructed segment tree
return st;
}

// Recursive version of the Euler tour of T
void eulerTour(Node *root, int l)
{
    /* if the passed node exists */
    if (root)
    {
        euler[ind] = root->key; // insert in euler array
        level[ind] = l;        // insert l in level array
        ind++;                // increment index

        /* if unvisited, mark first occurrence */
        if (firstOccurrence[root->key] == -1)
            firstOccurrence[root->key] = ind-1;

        /* tour left subtree if exists, and remark euler
        and level arrays for parent on return */
        if (root->left)
        {
            eulerTour(root->left, l+1);
            euler[ind]=root->key;
            level[ind] = l;
            ind++;
        }

        /* tour right subtree if exists, and remark euler
        and level arrays for parent on return */
        if (root->right)
        {
            eulerTour(root->right, l+1);
            euler[ind]=root->key;
            level[ind] = l;
            ind++;
        }
    }
}

// Returns LCA of nodes n1, n2 (assuming they are
// present in the tree)
int findLCA(Node *root, int u, int v)
{
    /* Mark all nodes unvisited. Note that the size of
    firstOccurrence is 1 as node values which vary from
    1 to 9 are used as indexes */
    memset(firstOccurrence, -1, sizeof(int)*(V+1));

    /* To start filling euler and level arrays from index 0 */
    ind = 0;

    /* Start Euler tour with root node on level 0 */
    eulerTour(root, 0);

    /* construct segment tree on level array */
    int *st = constructST(level, 2*V-1);

    /* If v before u in Euler tour. For RMQ to work, first
    parameter 'u' must be smaller than second 'v' */
    if (firstOccurrence[u]>firstOccurrence[v])
        std::swap(u, v);

    // Starting and ending indexes of query range
    int qs = firstOccurrence[u];
    int qe = firstOccurrence[v];

    // query for index of LCA in tour
    int index = RMQ(st, 2*V-1, qs, qe);

    /* return LCA node */
    return euler[index];
}

// Driver program to test above functions
int main()
{
    // Let us create the Binary Tree as shown in the diagram.
    Node * root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);

```

```

root->right->right = newNode(7);
root->left->right->left = newNode(8);
root->left->right->right = newNode(9);

int u = 4, v = 9;
printf("The LCA of node %d and node %d is node %d.\n",
      u, v, findLCA(root, u, v));
return 0;
}

```

## Questions and tactics

### ▼ Longest Flight Path (Non-Dijkstra Version)

#### Longest Flight Route

CSES - Longest Flight Route Uolevi has won a contest, and the prize is a free flight trip that can consist of one or more flights through cities. Of course, Uolevi wants to choose a trip that has as many cities as possible.

<https://cses.fi/problemset/task/1680>



#### 【题解】CSES 1680 Longest Flight Route

<https://yuihuang.com/cses-1680/>

<https://www.youtube.com/watch?v=9HMawd5jn9A>

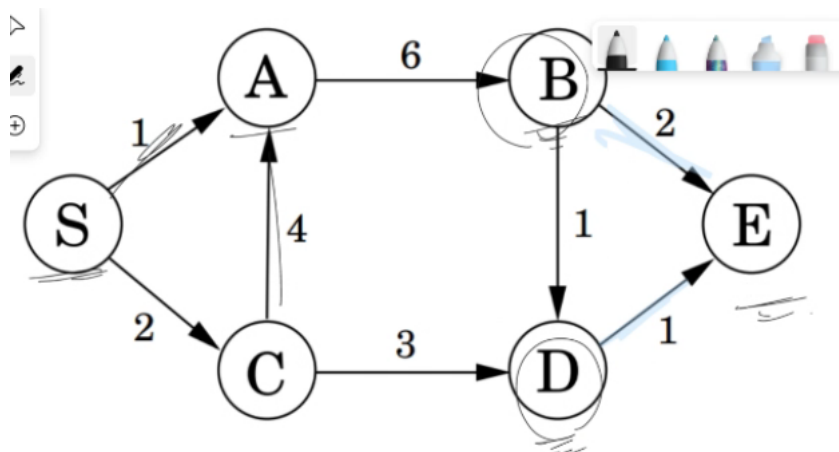
#### Ideone.com

Ideone is something more than a pastebin; it's an online compiler and debugging tool which allows to compile and run code online in more than 40 programming languages.

<https://ideone.com/QsHX3c>

Above is a Dijkstra algorithm for Longest Flight. Dijkstra algorithm (conventionally) only works when all weights are positive. HOWEVER, this is not the whole story. It works when the graph is monotonic - ie only non-increasing or non-decreasing. So like  $1 \rightarrow 2 \rightarrow 3 \rightarrow -1$ , would not work, but  $-1 \rightarrow -2 \rightarrow -3$  would (with some modifications) because it is monotonic in that it is non-increasing.

Of course, no cycles as well.



You only need to know the max distance at B and D to get max distance to E.

### The Usual Way (ie the way I submitted and understood)

A lot of dp questions can be solved with dag - but that does not mean it should be, since visualisation becomes very hard with hard dp questions.

The main part of topological sort/dag is the 1) state (ie nodes) and 2) transition (ie edges)

Storing reverse edge → so like E to B and E to D instead since you are going to go back from E to look for B and D

```
int pow(int base, int p) { int ret=1; while(p>0) { if (p%2==1) ret=ret*base; base=base*base; p=p/2; } return ret; }

int dp[maxn];
vector<pair<int,int>> edge[maxn];

int solve(int node) {
    if (dp[node]!=-1) return dp[node];
    for (auto v:edge[node]) {
        dp[node] = max(dp[v], v.second+solve(v.first));
    }
    return dp[node];
}

int main() {
    // freopen("input.txt", "r", stdin);
    std::ios::sync_with_stdio(false); cin.tie(0);
    memset(dp, -1, sizeof(dp));
    cout<<solve(E);
    return 0;
}
```

Topological sort done is done using BFS. I think it is also possible to do topological sort using DFS but it is more complicated.

### ▼ Divide and Conquer

The problem can be recursively defined by:

- $\text{power}(x, n) = \text{power}(x, n/2) * \text{power}(x, n/2)$ ; // if n is even
  - $x^n = x^{\frac{n}{2}} \times x^{\frac{n}{2}} = (x^2)^{\frac{n}{2}}$
- $\text{power}(x, n) = x * \text{power}(x, n/2) * \text{power}(x, n/2)$ ; // if n is odd

```
#include <bits/stdc++.h>
using namespace std;
void run(){
    long long base, power, mod;
    cin >> base >> power >> mod;
    long long ans = 1;

    while(power > 0){
        if(power % 2 == 1){
            ans = (ans*base)%mod; //if odd, ans * x
        }
        power = power >> 1;
        base = (base*base)%mod;
    }
    //power(x, n) = power(x, n/2) * power(x, n/2); // if n is even
    //I don't really get this part, why base = (base*base)?
    //OHH -> power(x, n/2) ^ 2 -> power/2, but x^2
    //if you write it out, for example, 5^4 = 5^2 * 5^2 = (5^2)^2
    //but you can't do mod for power, since that would just be wrong obviously.
    //so have to mod at the base
}
```

```

    }
    cout << ans << "\n";

int main(){
    int t;
    cin >> t;
    for(int i =0;i<t;i++){
        run();
    }
}

```

#### ▼ Coin Row/BearEatRabbit

```

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

int arr[200005];
long long int maxer[200005];
int n;
long long int second_maxer[200005];
//https://codebreaker.xyz/problem/beareatrabbit
//ie the coin row problem
int main(){
    cin >> n;
    for(int i =1;i<=n;i++){
        int a;
        cin >> a;
        arr[i] = max(a,0);
    }
    maxer[0] = 0;
    maxer[1] = arr[1];


    for(int i =2;i<=n;i++){
        maxer[i] = max(maxer[i-1], maxer[i-2] + arr[i]);
    }

    long long int ans = *max_element(maxer, maxer+n+1);
    cout << ans;
}

```

#### Bear Eat Rabbit

VERY IMPORTANT NOTE TO ALL DEC COURSE 2022 PARTICIPANTS (due to delays in email communication): Elementary is gone, so Advanced has a diagnostic test (if you fail, no Dec Course). Syllabus of diagnostic is anything in these prerequisite notes, please review them at this link.

 <https://codebreaker.xyz/problem/beareatrabbit>



You are given a row of  $N$  coins, each of which has value  $A_i$ . You cannot take two adjacent coins. Find the maximum value of coins you can take.

Let  $dp(i)$  denote the value of coins you can get by taking coins from the range  $[1, i]$  (1-indexed).

It is clear that the base case is  $dp(x) = 0$  for all  $x \leq 0$ .

Now we consider the transition.

If we take coin  $i$ , we cannot take coin  $i-1$ , so we should consider  $dp(i-2)$

If we do not take coin  $i$ , we can take coin  $i-1$ , so we should consider  $dp(i-1)$ .

Thus, the dp transition is just the maximum between  $dp(i-1)$  and  $dp(i-2) + A_i$ . The time complexity is  $O(N)$

I spent 2 hours on this. This made me so fucking stressed. **(Future me here. Wow, only two hours? Try 3 days!)**

This highlights the point that I need to 1) Grind more and 2) Understand the key concept of dp, which is dp state (ie  $dp(x)$  = 0, and what the array will look like) and dp relation  $\rightarrow$  ie consider  $dp(i-1)$ ?  $dp(i-2)$ ?  $+A[i]$  where?

Then form the relation of  $dp(i) = \max(dp(i-1), dp(i-2) + a[i])$

### Knapsack with repetition

Let's start with the version that allows repetition. As always, the main question in dynamic programming is, what are the subproblems? In this case we can shrink the original problem in two ways: we can either look at smaller knapsack capacities  $w \leq W$ , or we can look at fewer items (for instance, items  $1, 2, \dots, j$ , for  $j \leq n$ ). It usually takes a little experimentation to figure out exactly what works.

The first restriction calls for smaller capacities. Accordingly, define

$K(w)$  = maximum value achievable with a knapsack of capacity  $w$ .

Can we express this in terms of smaller subproblems? Well, if the optimal solution to  $K(w)$  includes item  $i$ , then removing this item from the knapsack leaves an optimal solution to  $K(w - w_i)$ . In other words,  $K(w)$  is simply  $K(w - w_i) + v_i$ , for some  $i$ . We don't know which  $i$ , so we need to try all possibilities.

$$K(w) = \max_{i: w_i \leq w} \{K(w - w_i) + v_i\},$$

where as usual our convention is that the maximum over an empty set is 0. We're done! The algorithm now writes itself, and it is characteristically simple and elegant.

<sup>1</sup>If this application seems frivolous, replace "weight" with "CPU time" and "only  $W$  pounds can be taken" with "only  $W$  units of CPU time are available." Or use "bandwidth" in place of "CPU time," etc. The knapsack problem generalizes a wide variety of resource-constrained selection tasks.

```
K(0) = 0
for w = 1 to W :
    K(w) = max{K(w - wi) + vi : wi ≤ w}
return K(W)
//relation is K(w) = max{K(w - wi) + vi, loop}
```

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

int n,v;

int arr[105];
int maxer[10005];


int main(){
    for(int i =0;i<10005;i++) maxer[i] = 30000;

    cin >> n >> v;
    for(int i =0;i<n;i++){
        cin >> arr[i];
        maxer[arr[i]] = 1;
    }
    maxer[0] = 0;
    for(int i =1;i<=v;i++){
        for(int j=0;j<n;j++){
            if(i>=arr[j]) maxer[i] = min(maxer[i], maxer[i- arr[j]] + 1);
        }
    }
    if(maxer[v] == 30000) cout << "-1";
    else cout << maxer[v];
}
```

Similar to 01 Knapsack, just twist  $arr[i]$  to  $+1$ , max to min, according to question

### moneychanger

Josephine the money changer was working at her kiosk one day when she realized how irritating it was when she had to give about 20 coins to her customer. Suddenly, she had an idea.

 <https://codebreaker.xyz/problem/moneychanger>



### ▼ TwoMeals/Max Sum of Two Non-Overlapping Subarrays

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

int n;
long long int arr[500005];
long long int front[500005];
long long int back[500005];

int main(){
    cin >> n;

    for(int i =0;i<n;i++) cin >> arr[i];

    //for(int i =1;i<=n;i++)printf("%d ", arr[i]);

    // //front iteration
    long long int sum1 = 0, sum2= 0, zero = 0;
    front[0] =0;back[0]=0;
    for(int i =0;i<n;i++){
        sum1 += arr[i];sum2 = max(sum2, sum1);
        sum1 = max(sum1, zero);

        front[i+1] = sum2;
    }

    // //back iteration
    sum1 = 0, sum2= 0;
    for(int i =n-1;i>=0;i--){
        sum1 += arr[i];
        sum2 = max(sum2, sum1);
        sum1 = max(sum1, zero);

        back[i] = sum2;
    }

    long long int ans = 0;
    for(int i =0 ;i<n;i++){
        //printf("%d %d %d\n", arr[i], front[i], back[i]);
        ans = max(ans, front[i] + back[i]);
    }
    cout << ans;
}
```

This applies the concept of 'Kadane's algorithm' which is used to find the maximum subarray in  $O(N)$  time. Not the most efficient, ie could be  $\log N$ , but good enough for most questions. Below is Kadane's algorithm

```
for(int i =0;i<n;i++){
    sum1 += arr[i];
    sum1 = max(sum1, zero);
    sum2 = max(sum2, sum1);
    front[i+1] = sum2;
}
```

How it works is that it tracks the prefix sum. However, if the prefix sum falls below 0, then  $\rightarrow 0$ . Then, there is another sum that is used to track the maximum, sum2. If sum1 prefix sum  $>$  sum2, sum2 = sum1, obviously. Subsequently, store into the array.

Repeat this array from the front and from the back. One interesting and confusing detail is that there needs to be a buffer of  $\text{front}[i] = 0$  for the code to be correct.

I think if you think the context of the question, when you select  $i=0$ , for instance, the element 2 below.

```

5
2 -1 5 0 -2

| 2 -1 5 0 -2 -> when considering i=0
2 | -1 5 0 -2 -> when considering i=1

```

Then the `front[0] = 0` most definitely since you are eliminating 2. But then if you test out the code with `printf("%d", back[i])` for `i=0`, it is 6? Shouldn't it be 5, since you are eliminating the 2?

**OHHH** You are **NOT** eliminating the two. You are just drawing a metaphorical line before the 2. So as shown above:

I resubmitted with no `back[0] = 0`, and the code still works. I think this explanation makes sense and is supported.

Alternative:

Handwritten notes on a whiteboard explaining a dynamic programming problem. The notes include a sequence of numbers 5, 6, -10, 4, -1, 6 and a table for  $dp[i][j]$  with values 0, 6, 8, -inf, 9, 4. The table is circled in red. There are also handwritten annotations like "max 6, -10", "dp[i][j]", and "positions".

#### ▼ 01 Knapsack 2.0 (Without Repetition)

$dp[i][j]$   $\rightarrow \sum w$  chosen items.

Consider available choice

- Take the  $i$ th item  $\rightarrow dp[i-1][j - w(i)] + \text{value}$
- Don't take the  $i$ th item  $\rightarrow dp[i-1][j]$

Handwritten notes on a whiteboard illustrating a dynamic programming solution for the Knapsack problem.

**Equation:**  $dp[i][j] = \max(\text{take } i^{\text{th}} \text{ item}, \text{not take } i^{\text{th}} \text{ item})$

**Take  $i^{\text{th}}$  item:**  $(dp[i-1][j-w[i]] + v[i])$

**Not take  $i^{\text{th}}$  item:**  $dp[i-1][j]$

**Base Case:**  $dp[0][j] = 0$

**DP Table:**

dp	0	1	2	3	4	5	6	7	8
0	0	0	-inf	30	-	-	-	-	-
1	0	30	-inf	30	50	-inf+50	80	-	-
2									

#### ▼ Longest Common Substring

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

string s1,s2;
int arr[1005][1005];
int result = 0;

int commoner()
{
    int m = s1.length();
    int n = s2.length();

    for (int i = 1; i <= m; i++)
    {
        for(int j=1;j<=n;j++){
            if(s1[i-1] == s2[j-1]){
                arr[i][j] = arr[i-1][j-1] + 1;
                result = max(result, arr[i][j]);
            }
            else{
                arr[i][j] = max(arr[i][j-1], arr[i-1][j]);
            }
        }
    }
    return result;
}

int main()
{
    cin >> s1 >> s2;
    cout << commoner();
    //cout << "\n";
    // int m = s1.length();
    // int n = s2.length();
    // for (int i = 1; i <= m; i++){
    //     for(int j=1;j<=n;j++){
    //         printf("%d ", arr[i][j]);
    //     }
    //     cout << "\n";
    // }
    return 0;
}
```



```

| a p p l e
h 0 0 0 0 0
a 1 0 0 0 0
p 0 1 1 0 0
p 0 1 1 0 0
y 0 0 0 0 0

```

Turns into:

```

| a p p l e
h 0 0 0 0 0
a 1 1 1 1 1
p 1 2 2 2 2
p 1 2 3 3 3
y 1 2 3 3 3

```

This is since the transition state is that  $dp(i,j) = dp(i-1,j-1)$  if  $s1[i-1] == s2[j-1]$ , ie both letters align

The reason for the array being filled up with non-zero numbers is so that the value can then be “transferred” → that’s why the original program with the

“else →  $arr[i][j] = 0$ ” does not work

Then it is necessary to see which one to take from, the upper or the beside, ie  $arr[i][j-1]$  vs  $arr[i-1][j]$ . This is to ensure that, for example you pick “apple” vs the first “p” in “happy” and want to know how many similar letters were there before. Then you would need the information going downwards, not only “sideways” as that would only cover cases like “happy” vs “l” in apple.

Another way to think of it is that horizontal compares h vs apple, then a vs apple.

But vertical compares happy vs a, happy vs p, etc.

**OH FUCK, I wrongly read a part of the question. THE SUBSEQUENCE NEED NOT BE CONTINUOUS! They just said “A subsequence is a sequence derived from an another original sequence by deleting some elements while leaving the order unchanged.” → that means that “apple” has subsequence “ape”**

**That’s why :**

appley


happyy

GETS 4! Not 3!

#### ▼ MagicWand (BFS with grid)

##### magicwand

Rar the Cat has found a greyscale picture H by W pixels wide. Each pixel is a value between 0 and 1023 (inclusive). Rar the Cat wants to edit the photo but is too lazy. He wants you to code the 'Magic Wand' feature.

 <https://codebreaker.xyz/problem/magicwand>



```

q.push(make_pair(x,y));
pair <int,int> loc;
arr[x][y] = 1;

while(!q.empty()){
    loc = q.front();

    q.pop();
    for(int i =0;i<4;i++){
        x_pos = loc.first + move_vert[i];
        y_pos = loc.second + move_hori[i];

        if(arr[x_pos][y_pos] == -1){
            q.push(make_pair(x_pos,y_pos));
            arr[x_pos][y_pos] = 1;
        }
    }
}

```

```

    }

}

```

VS

```

q.push(make_pair(x,y));
pair <int,int> loc;
while(!q.empty()){
    loc = q.front();
    arr[loc.first][loc.second] = 1;
    q.pop();
    for(int i =0;i<4;i++){
        x_pos = loc.first + move_vert[i];
        y_pos = loc.second + move_hori[i];

        if(arr[x_pos][y_pos] == -1)q.push(make_pair(x_pos,y_pos));
    }
}

```

The reason the first works and does not get TLE while the second does hit TLE is because:

“like now you still can have the same cell in ur queue multiple times. for example a cell is alrd in ur queue but not visited yet but you will still add it into ur queue if it can be reached from the current cell”

So by adding the `arr[x_pos][y_pos] = 1;` right after the `q.push()`, it eliminates this repeated queuing.

**This can be applied to other bfs algorithms as well. Maybe not the usual bfs questions since they don't usually have a repeated queue issue. BUT we should do it just in case.**

#### ▼ Segment Tree with Lazy Creation and Propagation (Range Sum Query)

```

#include <bits/stdc++.h>
using namespace std;
#define int long long int
//https://codebreaker.xyz/problem/segmenttree2
struct node{
    int s,e,m;
    int val, lazy;
    node *l = nullptr,*r = nullptr;

    node(int S, int E){//just creating
        s = S, e=E;
        m = (s+e)>>1;
        val = 0, lazy = 0;
    }

    void create(){
        if(s!=e){
            l = new node(s,m);
            r = new node(m+1,e);
        }
    }

    void propagation(){
        if(l == nullptr) create();
        //need to have l==nullptr for every function lol
        //or else the l->propagation() would result in the program propagating
        //from l, for which l->l == nullptr, thus RTE(11)

        if(lazy == 0) return;

        val += (lazy * (e - s + 1));

        if(s != e){
            l->lazy += lazy;
            r->lazy += lazy;
        }

        lazy =0;
    }
}

```

```

}

void update(int S, int E, int increment){
    if(l == nullptr) create(); //lazily create children

    if(S == s && E == e) lazy += increment;
    else{
        if(E <= m) l->update(S, E, increment);
        else if(S > m) r->update(S, E, increment); // [S,E] is in the right child
        else {
            l->update(S, m, increment);
            r->update(m+1, E, increment);
        }

        l->propagation(), r->propagation();

        val = l->val + r->val;
    }
}

int query(int S, int E){
    if(l == nullptr) create();

    propagation();
    if(S == s && E == e) return val;
    else if(E <= m) return l->query(S, E);
    else if(S > m) return r->query(S, E);
    else return (l->query(S, m) + r->query(m+1, E));
}
};

signed main(){
    int n,m;
    cin >> n >> m;
    node *root = new node(1,n);

    for(int i =0;i<m;i++){
        int type;
        cin >> type;
        if(type == 1){
            int a,b,c;
            //a--,b--;

            cin >> a >> b >> c;

            root->update(a,b,c);
        }
        else{
            int a,b;
            //a--,b--;
            cin >> a >> b;

            cout << root->query(a,b) << "\n"; //inclusive
        }
    }
}

```

Start with an anti-tree, and then slowly update. Can also be implemented using arrays. Size does not matter. Use `assert(total < maxn)` to check in debug. Honestly, just use the maximum size when using array, like `5e6+10`.

```

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

const int maxn=5e6+10;

int n,m;
ll tree[maxn],add[maxn];
int lc[maxn],rc[maxn];
int tot=1;

void fill(int c,int cl,int cr,ll val) {
    tree[c]=1ll*(cr-cl+1)*val;
    add[c]=val;
}

void push_down(int c,int cl,int cr) {
    if (lc[c]==-1) lc[c]=++tot;
    if (rc[c]==-1) rc[c]=++tot;
    int mid=cl+cr>>1;
    fill(lc[c],cl,mid,add[c]);
    fill(rc[c],mid+1,cr,add[c]);
}

```

```

        add[c]=0;
    }

    void update(int &c,int cl,int cr,int l,int r,ll val) {
        //the &c is basically the array but incomplete because lazy creation

        if (c==-1) c=++tot; //if the node has not been created yet
        if (l<=cl&&cr<=r) {
            fill(c,cl,cr,val);
            return;
        }
        int mid=cl+cr>>1;
        if (add[c]!=0) push_down(c,cl,cr);
        if (l<=mid) update(lc[c],cl,mid,l,r,val);
        if (r>mid) update(rc[c],mid+1,cr,l,r,val);
        if (lc[c]==-1) tree[c]=tree[rc[c]];
        else if (rc[c]==-1) tree[c]=tree[lc[c]];
        else tree[c]=tree[lc[c]]+tree[rc[c]];
        //    cout<<cl<<" "<<cr<<" "<<tree[c]<<endl;
        return;
    }

    ll query(int c,int cl,int cr,int l,int r) {
        if (c==-1) return 0;
        if (l<=cl&&cr<=r) return tree[c];
        int mid=cl+cr>>1;
        if (add[c]!=0) push_down(c,cl,cr);
        ll ret=0;
        if (l<=mid) ret+=query(lc[c],cl,mid,l,r);
        if (r>mid) ret+=query(rc[c],mid+1,cr,l,r);
        return ret;
    }

    int main() {
        //    std::ios::sync_with_stdio(false);
        memset(tree,0,sizeof(tree));
        memset(add,0,sizeof(add));
        memset(lc,-1,sizeof(lc));
        memset(rc,-1,sizeof(rc));
        scanf("%d",&n,&m);
        int rt=1;
        while (m--) {
            int op,l,r;
            scanf("%d%d%d",&op,&l,&r);
            if (op==1) {
                int k;
                scanf("%d",&k);
                update(rt,1,n,l,r,k);
            }
            else printf("%lld\n",query(rt,1,n,l,r));
        }
        assert(tot<maxn);
        return 0;
    }
}

```

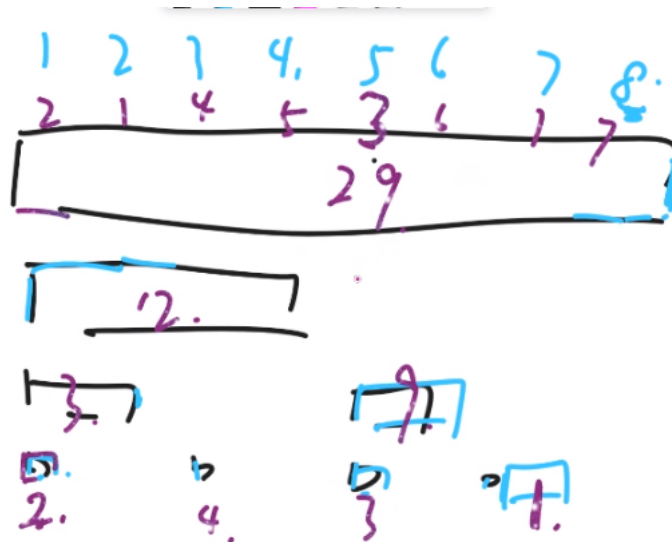
OR just use pointers and class. They automatically allocate space so its nice.

**Key:** Lazy propagation can be lazy = ... as well, depending on the type of question. This one has an =-1,0,1 instead

<https://codebreaker.xyz/problem/simpletask>

#### ▼ Fenwick Tree (Range query and point update)

A modification of segment tree



Fenwick can only solve sum query, not minimum, maximum and more complicated stuff

Similar to segment tree, only update nodes that have that position

Also  $\log(N)$  update and  $\log(N)$  query.

For range sum query, simply use prefix sum  $l_1$  - prefix sum  $l_2$

How to jump between nodes?

Need to know the size of the node. How large is the range covered by the node

For instance, for above example, element in position 4 covers 4 values.

So for range sum  $\rightarrow$  `return sum from 1 to 6 = tree[6] + tree[6-2 = 4].` You know that the size of 6 is 2 and size of 4 is 4.

How do you know?

The lowest bit that it is on. For example, for  $4 = 100 \rightarrow$  Lowest bit is  $100 \rightarrow 4$ , thus size is 4.

For  $6 \rightarrow 110 \rightarrow$  Lowest bit is  $10 \rightarrow 2$ , thus size 2.

How it works? IDK LOL

For arrays that are  $\neq$  the size of  $2^n$ , just put 0 at the end

How to find lowest bit:

```

3  #define ALL(x) x.begin(), x.end()
9  void inc(int &a, int b) {a=(a+b)%mod;}
0  void dec(int &a, int b) {a=(a-b)%mod;}
1  int lowbit(int x) {return x&(-x);}
2  ll pow(ll base, ll p) {ll ret=1; while(p>0) {if

```

`lowbit(int x)  $\rightarrow$  return  $x \& (-x)$ ;`

Why it works? lol

If possible, try to use fenwick tree, since the speed is higher and the space complexity is lower than segment tree.

ONLY 1 index, since it is only then does the bit indexing works

```

void update(int pos, int val) {
    while (pos < maxn) {
        tree[pos] += val;
        pos += lowbit(pos);
    }
}

```

Note that val is an increment value → For example, want to update 5 to 6? val = 1.

\*while pos > 0

```

int query(int pos) {
    int ret = 0;
    while (pos) {
        ret += tree[pos];
        pos -= lowbit(pos);
    }
    return ret;
}

```

The fenwick tree can also be modified to point query and range update.

Handwritten diagram illustrating the Fenwick tree structure for point query and range update. The diagram shows three rows of values:

- Row 1: 2, 3, (1, 4, 5), .
- Row 2: 2, 3, 3, 6, 7, .
- Row 3: 2, 1, 0, 3, 1, .

A horizontal line is drawn under the second row, and a curved line connects the first and second rows.

How to do this?

Store the array as the difference between adjacent positions. The difference is  $arr[i] - arr[i - 1]$ . First element is the same.

#### ▼ Counting Sort

ONLY use when the number of possible values are small - ie 26 like alphabet or 1-10, etc. Once the number becomes large, it becomes significantly more inefficient.

Counting sort is an efficient, stable sorting algorithm that operates by counting the number of occurrences of each unique element in the input array. It can be used to sort an array of elements that are in a small range, such as integers from 0 to 9 or characters in the alphabet.

Here's how it works:

1. Create a count array that will store the number of occurrences of each unique element in the input array. The count array should have the same number of elements as the range of possible values for the input array (e.g. 10 for integers from 0 to 9).
2. Initialize each element in the count array to 0.

3. Iterate through the input array and increment the count for each element in the count array. For example, if the input array contains the integers 3, 2, 5, 3, 7, the count array would be [0, 0, 2, 2, 0, 1, 0, 1, 0, 0] after this step.
4. Modify the count array so that each element at index  $i$  stores the sum of the elements at indices 0 through  $i$  in the count array. This step is called prefix sum. For example, the modified count array for the above example would be [0, 0, 2, 4, 4, 5, 5, 6, 6, 6].
5. Create a result array that is the same size as the input array.
6. Iterate through the input array in reverse order. For each element in the input array, place it in the result array at the index indicated by the count array, and decrement the count for that element in the count array. For example, if the input array is [3, 2, 5, 3, 7], the result array would be [2, 3, 3, 5, 7] after this step.
7. Return the result array.

#### ▼ Feast NOI 2019

```
#include <bits/stdc++.h>
#include <set>
#include <vector>
#include <queue>
using namespace std;
#define int long long int
#define pii pair<int,int>
//signbit to check for sign -> true if neg
int n,k;
vector <int>arr;

signed main(){
    ios::sync_with_stdio(0);
    cin.tie(0);
    cin >> n >> k;
    for(int i =0;i<n;i++){
        int a;
        cin >> a;
        arr.push_back(a);
    }

    while(arr[0] <= 0ll && !arr.empty())arr.erase(arr.begin());
    while(arr.back() <=0ll && !arr.empty())arr.pop_back();

    vector<int> res;

    res.push_back(arr[0]);

    for (int i = 1; i < arr.size(); i++) {
        if(arr[i]<=0ll and res[res.size()-1]<=0ll) res[res.size()-1]+=arr[i];
        else if(arr[i]>0ll and res[res.size()-1]>0ll) res[res.size()-1]+=arr[i];
        else res.push_back(arr[i]);
    }

    set<pii>setter; //index, value
    priority_queue <pii, vector<pii>, greater<pii>>pq; //abs, index

    for(int i=0;i< res.size();i++){
        setter.insert({i, res[i]});
        pq.push({abs(res[i]) , i});
    }

    while(setter.size() > 2 *k){
        int idx = pq.top().second;
        pq.pop();
        auto x = setter.lower_bound({idx, -1e18});

        if(x == setter.end() || x->first!=idx) continue;
        if(x == setter.begin()){
            setter.erase(setter.begin());
            setter.erase(setter.begin());
        }
        else if(x == (--setter.end())){
            setter.erase(--setter.end());
            setter.erase(--setter.end());
        }
        else{
            x--;
            int sum = 0;
            sum += x->second; x++;
        }
    }
}
```

```

        sum += x->second; x++;
        sum += x->second;

        for(int i =0;i<3;i++)setter.erase(x--);

        setter.insert({idx, sum});
        pq.push({abs(sum), idx});
    }
}
int ans = 0;

for(auto e: setter){
    if(e.second > 0) ans += e.second;
}
cout << ans;
}

```

The concept is that you use a minimum priority queue with  $\langle \text{abs val}, \text{index} \rangle$  and a set with  $\langle \text{index}, \text{val} \rangle$ . Then take the minimum absolute value using priority queue, obtain the index, merge the adjacent.

The key is that if the absolute value is at the beginning or the end, you just eliminate it as well as the adjacent value. This is possible since you deleted the all negative at the ends. Thus, the absolute value in question will always be a positive. Since the absolute value is a positive, an it is the smaller, when combined, you will get a negative value, which you should just discard.

This question is absolutely insane.

**Alternative solution: I think you can solve it using the dp-alien trick? whatever that is.**

#### ▼ CollectingMushroom

I remember Elton taking 30 minutes to run the program LOL!

```

#include <bits/stdc++.h>
#include <vector>
using namespace std;
#define pii pair<int,int>
/*
The first line of input will contain four integers: R, the number of rows, C, the number of
columns, D, the maximum distance between a sprinkler and a watered mushroom, and K, the
minimum number of sprinklers required for a mushroom to be harvestable
*/
int r,c,d,k;
vector <pii> coordinates_mushroom;

int main(){
    cin >> r >> c >> d >> k;
    int prefix_sprinkler[r + 10][c + 10];
    for(int i =0;i<r+10;i++){
        for(int f=0;f<c+10;f++){
            prefix_sprinkler[i][f] = 0;
        }
    }

    for(int i =1;i<=r;i++){
        for(int f = 1;f<=c;f++){
            char ct;
            cin >> ct;

            prefix_sprinkler[i][f] = prefix_sprinkler[i][f-1] + prefix_sprinkler[i-1][f] - prefix_sprinkler[i-1][f-1];

            if(ct == 'M'){
                coordinates_mushroom.push_back({i,f});
            }
            if(ct == 'S'){
                prefix_sprinkler[i][f]++;
            }
        }
    }

    // for(int i =1;i<=r;i++){
    //     for(int f = 1;f<=c;f++){
    //         printf("%d ",prefix_sprinkler[i][f] );
    //     }
    //     cout << "\n";
    // }
}

```



```

int ans = 0;
for(pii e: coordinates_mushroom){
    // use d
    //e.first -> x
    //e.second -> y

    int upper_edge_x = max(e.first - d, 1);
    int upper_edge_y = max(e.second - d, 1);

    int lower_edge_x = min(e.first + d, r);
    int lower_edge_y = min(e.second + d, c);

    int number_of_sprinklers = prefix_sprinkler[lower_edge_x][lower_edge_y];
    number_of_sprinklers-= prefix_sprinkler[upper_edge_x - 1][lower_edge_y];
    number_of_sprinklers-= prefix_sprinkler[lower_edge_x][upper_edge_y - 1];
    number_of_sprinklers+= prefix_sprinkler[upper_edge_x - 1][upper_edge_y - 1];

    // printf("NO: %d X: (%d %d) (%d %d) (%d %d)\n", number_of_sprinklers, e.first, e.second, upper_edge_x, upper_edge_y, lower_edge_x, lower_edge_y);
    // printf("%d %d %d %d\n", prefix_sprinkler[lower_edge_x][lower_edge_y], prefix_sprinkler[upper_edge_x - 1][lower_edge_y], prefix_sprinkler[lower_edge_x][upper_edge_y - 1], prefix_sprinkler[upper_edge_x - 1][upper_edge_y - 1]);
    if(number_of_sprinklers >= k) ans++;
}
cout << ans;
}

```

Issue 1: RTE(11) → At first, I used  $i=0; i \leq r+10$  for  $arr[r+10]$ . This causes RTE (11) since  $arr[r+10]$  does not exist! So it must be  $i < r+10$ !

2 dimensional prefix array →

```

prefix_sprinkler[i][f] =
prefix_sprinkler[i][f-1] +
prefix_sprinkler[i-1][f] -
prefix_sprinkler[i-1][f-1];

WOW! MUST INCLUDE -[i-1][f-1]

```

The strategy is to know how many sprinklers there are in an area  $(A_x, A_y)$  and  $(B_x, B_y)$ . Each mushroom has a 'zone' that the sprinklers must fall into, so you calculate the number of sprinklers using that 'zone'

#### ▼ Roadside Advertising 2018 NOI

Need to include all the nodes indicated

Idea 1) Use Union Disjoint, minimum tree?

Subtask 2:

Now, if the query mentions 5 vertices  $\{2, 4, 3, 1, 5\}$  with the leftmost vertex is vertex 2 (index 1 in array  $A$ ) and the rightmost vertex is vertex 4 (index 4 in array  $A$ ), then the answer is immediately  $pre[4] - pre[1-1] = pre[4] - pre[0] = 9 - 1 = 8$ . The time complexity of this solution is  $O(V + Q)$ , i.e. we can answer each query in  $O(1)$ . Minor details are omitted. This solution worth 23 points.

How do you know the rightmost and leftmost vertex??

Full solution

- Lowest common ancestor
- Sparse tree algorithm, minimum range tree query

### ▼ Dijkstra for shortest path

Add all the start nodes before starting the loop

```
#include <bits/stdc++.h>
#include <vector>
#include <algorithm>
#include <queue>
using namespace std;
// #define int long long int
#define pii pair<int, int>

int n, e;
vector<pii> arr[100005];
int dis[100005];
priority_queue<pii, vector<pii>, greater<pii>> q;

int main(){
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    memset(dis, 0x3F, sizeof(dis));
    cin >> n >> e;

    int a, b, c;
    for(int i = 0; i < e; i++){
        cin >> a >> b >> c;
        arr[a].push_back(make_pair(b, c));
        arr[b].push_back(make_pair(a, c));
    }
    dis[1] = 0;
    q.push(make_pair(dis[1], 1));
    int dist, u;
    while(!q.empty()){
        pii cur = q.top();
        q.pop();
        dist = cur.first;
        u = cur.second;
        if(dis[u] < dist) continue;

        for(pii e: arr[u]){
            if(dis[e.first] > dis[u] + e.second){
                dis[e.first] = dis[u] + e.second;
                q.push(make_pair(dis[e.first], e.first));
            }
        }
    }
    if(dis[n] == INT_MAX) cout << "-1";
    else cout << dis[n];
}
```

### ▼ Dijkstra for longest path (using AI) (Experimental! Dijkstra longest path does not really work)

```
#include <iostream>
#include <vector>
#include <queue>
#include <cstdio>
#include <cstring>
#include <limits>

using namespace std;

const int N = 100005;
const int inf = INT_MAX;

vector<pair<int, int>> edges[N];
int dist[N];
bool visited[N];

void dijkstra(int start) {
    priority_queue<pair<int, int>> q;
    for (int i = 1; i < N; i++) {
        dist[i] = -inf;
    }
    dist[start] = 0;
    q.push({0, start});

    while (!q.empty()) {
        dis = q.top().first;
        int u = q.top().second;
        q.pop();
```

```

        if(dist[u] != dis)continue; //or might have to < or >, just try it out

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

            if (dist[v] < dist[u] + w) {
                dist[v] = dist[u] + w;
                q.push({-dist[v], v});
            }
        }
    }
}

int main() {
    int n, m;
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= m; i++) {
        int u, v, w;
        scanf("%d%d%d", &u, &v, &w);
        edges[u].push_back({v, w});
        edges[v].push_back({u, w});
    }

    int start;
    scanf("%d", &start);
    dijkstra(start);

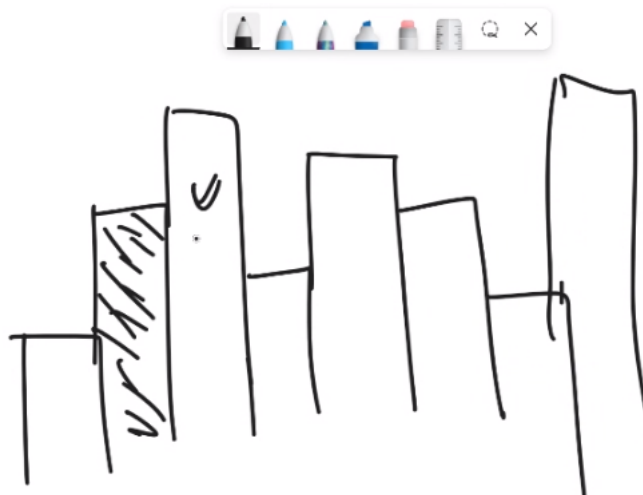
    for (int i = 1; i <= n; i++) {
        if (dist[i] == -inf) {
            printf("%d is unreachable\n", i);
        } else {
            printf("%d has distance %d\n", i, dist[i]);
        }
    }

    return 0;
}

```

▼ 2D segment tree?? (Included in NOI 2016 Ans for ski resort)

▼ Monotonic stack (starting from the right to left)



How to know what is the next largest bar? For all bars

The brute force way is  $O(N)$  for each bar  $\rightarrow O(N^2)$

Obviously, you want to optimize.

Monotonic stack decreases this to  $O(N)$ , ie  $O(1)$  per query (I think)

The goal is to get the FIRST higher bar

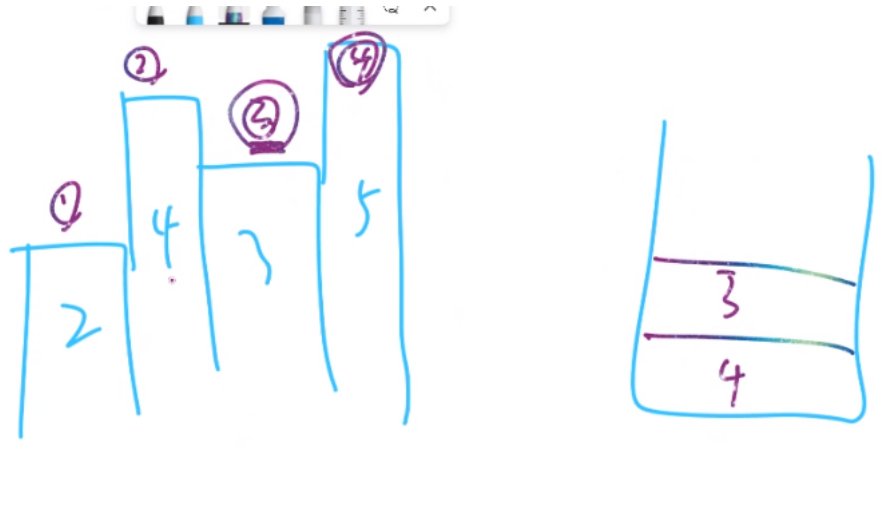
So, going from the left to right, in a stack, you can use monotonic stack to find this out.



First bar will be selected first, second bar is not, since first bar is to the left (right  $\rightarrow$  left query) and the bar is higher

Think of it as the first bar 'blocking' the second.

That is why, the thing that you keep in the stack, the height is monotonic - ie the top is lower value, bottom of stack is higher values.



But since brick 2 is better than brick 3, you take out brick 3. Compare brick 2 to 4, but 4 is better. So put in 2. Brick 1, you cannot tell which is better since it is left of brick 2 and smaller.

The answer is brick 1,2,3,4, but im fucking bricked UP.bruh.

You can store the size or the index, depending on what you want. But you need to store it in an identifiable way - ie can compare  $\rightarrow$  but for both it is fine since for index you can use `h[]` to compare

▼ Monotonic queue  $\rightarrow$  Dequeue (Left to right)

You are given an array of integers `nums`, there is a sliding window of size `k` which is moving from the very left of the array to the very right. You can only see the `k` numbers in the window. Each time the sliding window moves right by one position.

Return the max sliding window.

**Input:** `nums = [1,3,-1,-3,5,3,6,7]`, `k = 3`

**Output:** `[3,3,5,5,6,7]`

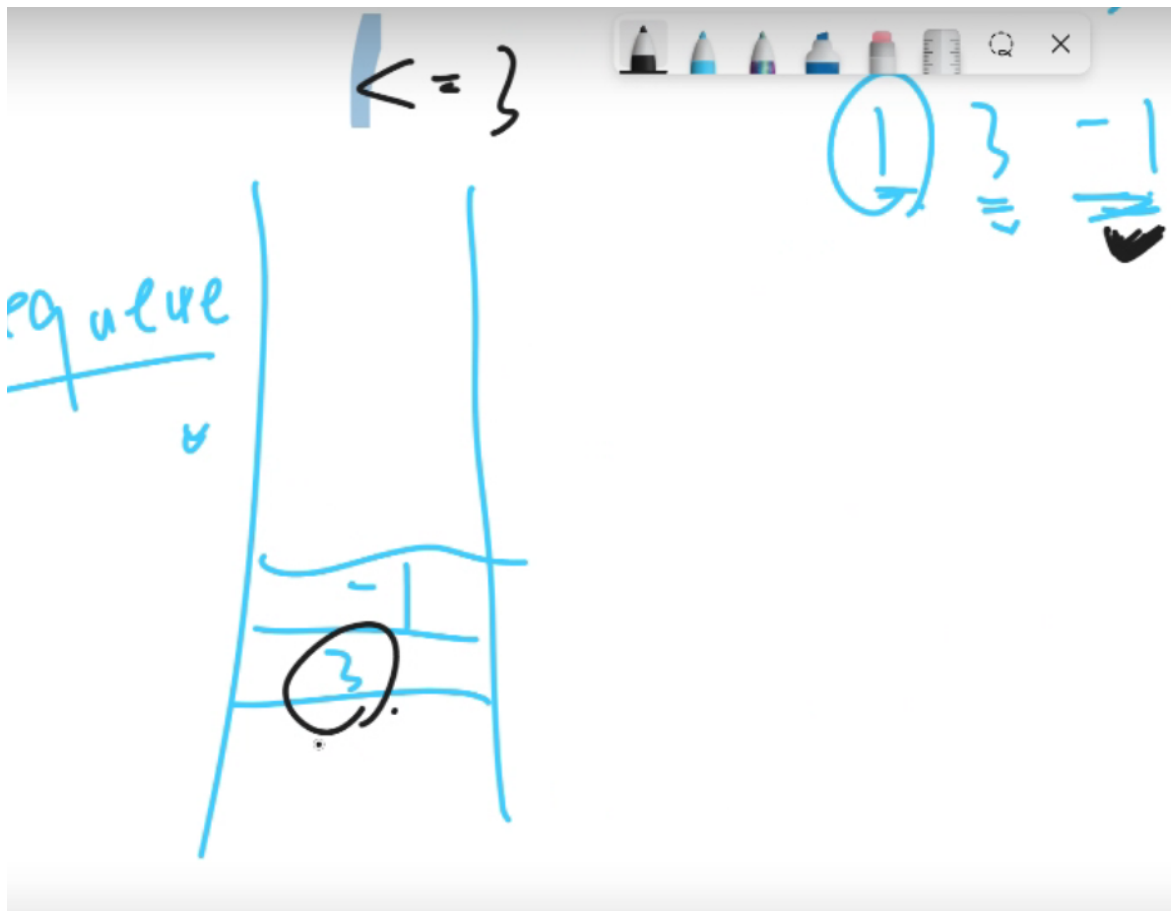
**Explanation:**

Window position	Max
<code>[1 3 -1]</code> <code>-3 5 3 6 7</code>	<b>3</b>
<code>1 [3 -1 -3]</code> <code>5 3 6 7</code>	<b>3</b>
<code>1 3 [-1 -3 5]</code> <code>3 6 7</code>	<b>5</b>
<code>1 3 -1 [-3 5 3]</code> <code>6 7</code>	<b>5</b>
<code>1 3 -1 -3 [5 3 6]</code> <code>7</code>	<b>6</b>
<code>1 3 -1 -3 5 [3 6 7]</code>	<b>7</b>

Brute force is  $O(NK)$

With monotonic dequeue (double queue), this becomes  $O(N)$

So processing  $\rightarrow$  Stack  $\rightarrow$  1  $\rightarrow$  3 to the right  $\rightarrow$  Always better than 1  $\rightarrow$  Better, replace 1  $\rightarrow$  -1? You might use, so put in  $\rightarrow$  -3? Keep as well.



At -1 element, bottom item is best, 3. → Use deque

Then 5 → for  $k=3$ , remove 3. Thus, need deque (double ended) to remove it → then at the end, there is only 5, since -1 and -3 is worse than 5.

3? → 5 is still the answer, but you still need to put in 3.

6? You can remove 3 and 5. Answer is 6.

7? Remove 7, answer is 7.

If needed, can store pair → index and value, etc

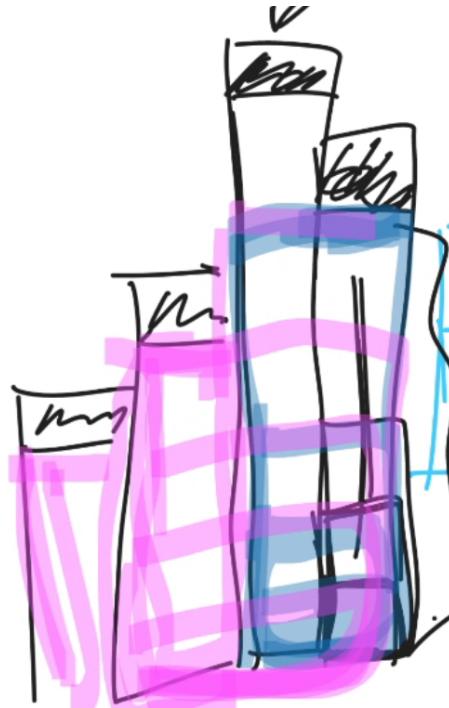
So something like this:

```
std::ios::sync_with_stdio(false);cin.tie(0);
for(int i = 0;i<n;i++) {
    while (!s.empty() and h[s.front()]<h[i]) s.pop_front();
    while (s.back()+k<i) s.pop_back();
}
return 0;
```

#### ▼ Fabric Monotonic

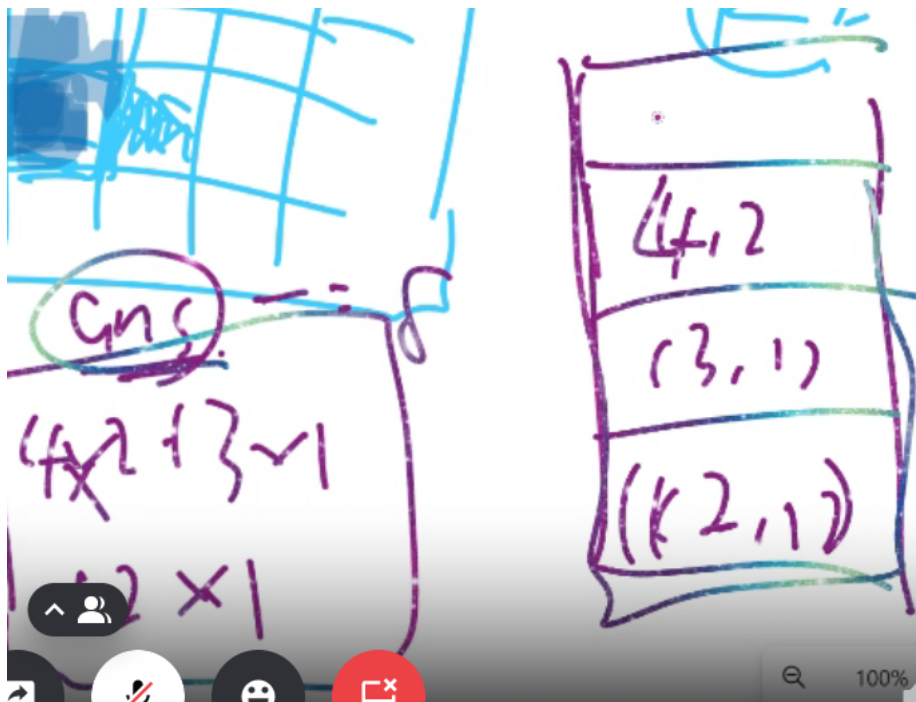
If you can represent some data as a histogram (bar stuff) and query some data from it, you can maybe use monotonic stack/queue.

So for instance, for NOI 2016 Fabric, you can 'represent' as a histogram.



For instance, first stack, in blue, bounded by 4th bar. Calculate.

Pink rectangle, bounded by 2nd bar. Calculate.



Monotonic stack would allow you to keep track of the answer very quickly. Remove? Just minus. Add? Just plus. All  $O(1)$ !

#### ▼ Pandaski DP sol first 4 subtasks

```
#include <bits/stdc++.h>
using namespace std;
#define int long long int
#define pii pair<int,int>

struct data_lol{
    int xi;
    int yi;
    int easiness;
    int score;
};

bool sorter(data_lol i, data_lol j){
    return i.yi < j.yi;
}

int n, h;
vector <pii> adj[10005];
data_lol arr_data[10005];
int dist[10005]; //maximise this
const int inf = LLONG_MAX;
bool visited[10005];

bool valid(int xi, int yi, int xj, int yj, int easiness){
    //yeah so swapped the yi and xi :(
    if(xi == xj && yi == yj) return false;

    int difficulty = max(abs(xj - xi) , abs(yi - yj));
    if(yi >= yj && difficulty <= easiness) return true;

    return false;
}

signed main(){

    cin >> n >> h;
    for(int i =0;i<n;i++){
        int xi, yi, score, easiness;
        cin >> xi >> yi >> score >> easiness;

        arr_data[i].xi = xi;
        arr_data[i].yi = yi;
        arr_data[i].score = score;
        arr_data[i].easiness = easiness;
    }
    sort(arr_data, arr_data+n, sorter);

    for(int i =0;i<n;i++){
        dist[i] = arr_data[i].score;
        for(int j = 0; j<n;j++){//can move from gate i to j
            if(valid(arr_data[i].xi, arr_data[i].yi ,arr_data[j].xi ,arr_data[j].yi , arr_data[i].easiness)){
                dist[i] = max(dist[i], dist[j] + arr_data[i].score);
                //printf("VALID %lld: %lld (%lld %lld)\n", j, dist[j], arr_data[j].xi, arr_data[j].yi);
            }
        }
        //printf("AIDS %lld: %lld (%lld %lld)\n", i, dist[i], arr_data[i].xi, arr_data[i].yi);
    }
    int maxer = 0;
    for(int i =0;i<n;i++){
        maxer = max(maxer, dist[i]);
    }
    cout << maxer;
}
```

#### ▼ unluckyfloors first 3 subtasks

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

int n;
/*
If Ti is 1, you are to convert Xi from the lucky numbering scheme to the conventional num-
```



bering scheme and print the result in a single line. However, if Xi is not a valid number in the lucky numbering scheme, print -1 as the answer instead.  
 If Ti is 2, you are to convert Xi from the conventional numbering scheme to the lucky numbering scheme and print the result on a single line.

```

ti = 1 -> can have -1
*/
bool check_unlucky(int num){
    string s = to_string(num);
    auto found_4 = s.find("4");
    auto found_13 = s.find("13");
    if(found_4 != string::npos || found_13 != string::npos) return false;
    else return true;
}

vector <pair<int,int>> lucky;
vector<int>conventional_array;

int main(){
    int counter_unlucky = 0;
    for(int i =1;i<=200000;i++){

        if(check_unlucky(i)){
            lucky.push_back({i, counter_unlucky});
        }
        else counter_unlucky++;

        conventional_array.push_back(i - counter_unlucky);
    }

    cin >> n;
    for(int i =0;i<n;i++){
        int t, num;
        cin >> t >> num;
        if(t == 1){
            pair <int,int> ele = {num, -1e9};
            auto iter1 = lower_bound(lucky.begin(), lucky.end(), ele);

            if(iter1 == lucky.end() || (*iter1).first != num)cout << "-1\n";
            else{
                cout << (*iter1).first - (*iter1).second << "\n";
            }
        }
        else{
            auto iter1 = lower_bound(conventional_array.begin(), conventional_array.end(), num);
            cout << (iter1 - conventional_array.begin() +1) << "\n";
        }
    }
}

```

#### ▼ unluckyfloors dp solution

## Subtask 4:

### Dynamic Programming

$dp(p, d)$ : no. of lucky numbers with  $p$  digits and  $d$  as first digit

#### Base Cases

- $dp(p, 4) = 0$ 
  - Any number with 4 isn't a lucky number
- $dp(1, 0) = 0$ 
  - 0 is not a lucky number
- $dp(1, d) = 1$ 
  - Any number with 1 digit is a lucky number (other than 4 and 0)

## Subtask 4:

### Dynamic Programming

$dp(p, d)$ : no. of lucky numbers with  $p$  digits and  $d$  as first digit

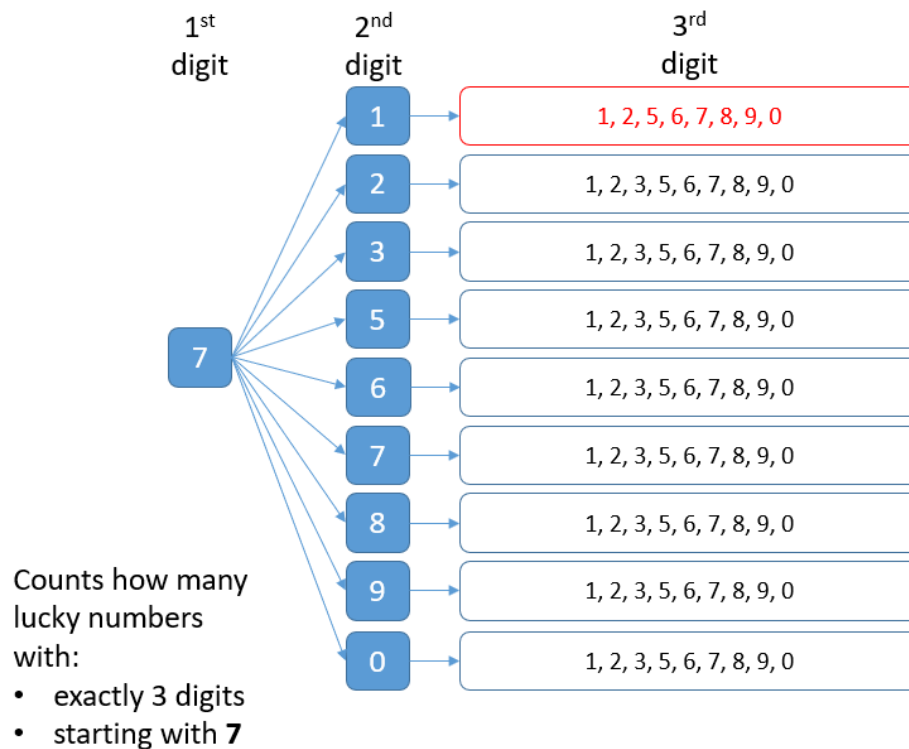
#### Transition

- $dp(p, d) =$ 
  - if  $d$  is 1
    - $\sum_i dp(p-1, i)$  where  $i \in \{0, 1, 2, 5, 6, 7, 8, 9\}$
  - if  $d$  is **not** 1
    - $\sum_i dp(p-1, i)$  where  $i \in \{0, 1, 2, 3, 5, 6, 7, 8, 9\}$

#### Computing Answer

- If  $X_i$  has  $K$  digits and of the form  $10^K - 1$ 
  - $\sum_{i=0}^9 dp(K, i)$

## Subtask 4



### ▼ DFS with matrix adjacency

```
// C++ implementation of the approach
#include <bits/stdc++.h>
using namespace std;

// adjacency matrix
vector<vector<int>> > adj;

// function to add edge to the graph
void addEdge(int x, int y)
{
    adj[x][y] = 1;
    adj[y][x] = 1;
}

// function to perform DFS on the graph
void dfs(int start, vector<bool>& visited)
{
    // Print the current node
    cout << start << " ";

    // Set current node as visited
    visited[start] = true;

    // For every node of the graph
    for (int i = 0; i < adj[start].size(); i++) {

        // If some node is adjacent to the current node
        // and it has not already been visited
        if (adj[start][i] == 1 && (!visited[i])) {
            dfs(i, visited);
        }
    }
}

int main()
{
    // number of vertices
```

```

int v = 5;

// number of edges
int e = 4;

// adjacency matrix
adj = vector<vector<int>>(v, vector<int>(v, 0));

addEdge(0, 1);
addEdge(0, 2);
addEdge(0, 3);
addEdge(0, 4);

// Visited vector to so that
// a vertex is not visited more than once
// Initializing the vector to false as no
// vertex is visited at the beginning
vector<bool> visited(v, false);

// Perform DFS
dfs(0, visited);
}

```

#### ▼ lboard ie fucking figuring out the 2d array

```

#include <bits/stdc++.h>
using namespace std;
#define int long long int
int n,m;
int values[1010][1010];
int prefix[1010][1010];
int temp[1010];
int min_prefix[1010][1010];

int prefix_sum(int lowx, int lowy, int highx, int highy){
    int ans = prefix[highx][highy];
    ans += prefix[lowx-1][lowy-1];
    ans -= prefix[lowx-1][highy];
    ans -= prefix[highx][lowy-1];
    //printf("%d %d %d %d\n", prefix[highx][highy], prefix[lowx-1][lowy-1], prefix[lowx-1][highy], prefix[highx][lowy-1]);
    return ans;
}

signed main(){
    cin >> n >> m;
    for(int i=1; i<=n; i++){
        for(int j=1; j<=m; j++){
            cin >> values[i][j];
        }
    }

    for(int i =1; i<=n; i++){
        for(int j=1; j<=m; j++){
            prefix[i][j] = prefix[i][j-1] + prefix[i-1][j] - prefix[i-1][j-1];
            prefix[i][j] += values[i][j];
        }
    }
    /**
     * 2 2
    8 1
    3 4
     * **/
    int maxer = LLONG_MIN;
    for(int i =1; i<=n; i++){
        for(int j=1; j<=m; j++){
            int ans = values[i][j];

            int horizontal_arm = 0, vertical_arm=0;

            for(int go_col = 1; go_col < j; go_col++){//fix x
                horizontal_arm = max(horizontal_arm, prefix_sum(i, go_col, i, j-1));
            }
            for(int go_col = m; go_col > j; go_col--){
                horizontal_arm = max(horizontal_arm, prefix_sum(i, j+1, i, go_col));
            }

            for(int go_row = 1; go_row < i; go_row++){// fix y
                vertical_arm = max(vertical_arm, prefix_sum( go_row, j, i-1, j));
            }
            for(int go_row = n; go_row > i; go_row--){
                vertical_arm = max(vertical_arm, prefix_sum(i+1, j, go_row, j));
            }

```

```

    }
    //printf("%lld %lld %lld %lld %lld %lld %lld\n", i, j, ans, vertical_arm, horizontal_arm, maxer, ans + vertical_arm + horizontal_arm);
    maxer = max(maxer, ans + vertical_arm + horizontal_arm);
}
}

cout << maxer;
}
/**
4 4
1 1 1 1
1 2 1 1
1 1 1 1
1 1 1 1
**/

```

So the issues:

- I did not think through the process actively initially, basically just blindly trusting instinct. This did not work at all. When working with complex things like 2d arrays, have to always be thinking about i and j variables and how it fits, into rows and columns and all
- I SWAPPED THE I AND J FOR A WHILE!

#### ▼ Subtask 1 bruteforce dragonfly

```

#include <bits/stdc++.h>
using namespace std;
#define pii pair<int,int>
int n,d; //nodes and num of dragonflies
vector<int> adj[100005];
int bugs[100005];
int species[100005];

int homes[100005];
int parents[100005];

void dfs(int current){
    //start from node 1

    for(int neigh : adj[current]){
        if(neigh != parents[current]){
            parents[neigh] = current;
            dfs(neigh);
        }
    }
}

int query(int x){
    set<int> species_eaten;

    while(x){
        if(bugs[x]){
            bugs[x]--;
            species_eaten.insert(species[x]);
        }
        x = parents[x];
    }

    return species_eaten.size();
}

int main(){
    cin >> n >> d;
    //parent of node 1 is 0
    for(int i =1;i<=n;i++) cin >> bugs[i];
    for(int i =1;i<=n;i++) cin >> species[i];
    for(int i=1;i<=d;i++) cin >> homes[i];
    for(int i = 0; i<n-1;i++){
        int u,v;
        cin >> u >> v;
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    dfs(1);

    for(int i =1;i<=d;i++){
        cout << query(homes[i]) << " ";
    }
}

```

```

    }
}
/**
 *
 * **/

```

### ▼ Treecutting subtask 1 2 3

```

#include <bits/stdc++.h>
using namespace std;
const int n_max = 3e5+10;
vector<int> adj[n_max];
bitset<n_max> ban;

int deepest(int n, int x) {
    queue<int> q;
    vector<int> ans(n, -1);
    ans[x] = 0;
    int biggest = x;
    q.push(x);
    while(q.size()) {
        int next = q.front();
        q.pop();
        for(int i: adj[next]) {
            if(ans[i] == -1 && !ban[i]) {
                ans[i] = ans[next] + 1;
                if(ans[i] > ans[biggest]) {
                    biggest = i;
                }
                q.push(i);
            }
        }
    }
    x = biggest;
    assert(q.empty()); //check actually empty
    ans = vector<int>(n, -1); //start declaring again
    ans[x] = 0;
    biggest = x;
    q.push(x);
    while(q.size()) {
        int next = q.front();
        q.pop();
        for(int i: adj[next]) {
            if(ans[i] == -1 && (!ban[i] || !ban[next])) {
                ans[i] = ans[next] + 1;
                if(ans[i] > ans[biggest]) {
                    biggest = i;
                }
                q.push(i);
            }
        }
    }
    //repeat bfs
    return ans[biggest];
}

int main() {
    int n; cin>>n;
    for(int i=1; i<n; i++) {
        int u,v; cin>>u>>v; u--; v--; //must have this!!!!
    }
    //tried writing my own version, but forgot about this. ded for like 15 mins
    adj[u].push_back(v);
    adj[v].push_back(u);
}

//this question is NOT a general graph - with N nodes and N-1 edges, and all nodes connected, it forms a tree 100%
//of the time. Thus, you can use ban[i] and ban[j] to simply partition the nodes
//as compared to general graphs where there might be another route even if one edge is taken away.
int ans = 0;
for(int i=0; i<n; i++) {
    for(int j: adj[i]) {
        ban[i] = true;
        ban[j] = true; //ie cannot access these edges

        ans = max(ans, deepest(n,i) + deepest(n,j)+1);
        ban[i] = false;
        ban[j] = false; //reverse
    }
}
printf("%d\n", ans);
}

```

#### ▼ LIS binary search

Remember how to do the usual LIS in  $N^2$ ?

Now, its  $N \log N$

So the usual tactic is dynamic programming brute force, where you observe that the most optimal subsequence has a subsequence of its own, and so on.

Thus, you can form  $d[i] = \max(d[j] + 1)$  for  $a[j] < a[i]$  since always increasing anyways. However, this is  $N^2$ . Not good enough.

Through another observation, you can improve to binary search

The observation is: for a valid subsequence, the variables are always sorted. Thus, binary search is possible.

Next, observe that, if you store  $d[i]$  as the element at which subsequence of length  $i$  terminates at, you can observe that if there is a more optimal  $a[j]$  to insert, it only changes at most one value,  $d[j]$ , as you would get  $d[j] = a[i]$ , which would be more optimal since the element it terminates at is smaller.

So:

- For each element in the input sequence:
  - a. Use binary search to find the index  $i$  of the smallest element in  $L$  that is greater than or equal to the current element.
  - b. If  $i$  is equal to the length of  $L$ , append the current element to  $L$ .
  - c. Otherwise, replace  $L[i]$  with the current element.
- The length of  $L$  is the length of the LIS.

In step 2.c of the algorithm, if the index  $i$  found by the binary search is not equal to the length of the list  $L$ , it means that there is already an element in that position in  $L$  that is greater than or equal to the current element from the input sequence. This element in  $L$  would not be part of a longer increasing subsequence than the one that includes the current element, so it is replaced with the current element.


By replacing the element in  $L[i]$  with the current element, the  $L$  list is always storing the smallest possible elements of the increasing subsequence. This way, the length of  $L$  is always the length of the LIS.

This takes advantage of the kinda-intuitive observation that the longest subsequence would have the smallest elements sorted.

#### ▼ Tower

##### Codeforces Round 271 Div. 2 Problem E - Pillars

Problem Statement: 474E - Pillars Solution: This problem has a very interesting use of segment tree in combination with DP. The DP idea is very standard: let  $f[i]$  be the maximum length of jumps using pillars from  $[1..i]$ , ending at  $i$ . Then  $f[i] = \max of f[j] + 1$ , for all  $j$  such that  $\lfloor (h_i - h_j) \rfloor \geq d$ .

 <https://abitofcs.blogspot.com/2014/12/codeforces-round-271-div-2-problem-e.html>

#### ▼ watching

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

int n,p,q;
int arr[2010];
bool valid(int w){
    //this is the tough part. using dp to check for validity.
    //so i remember that dp[i][j] is defined as the minimum number of large cameras to fill 1...i, such that number
    //of small camera used is <=j
    int dp[n][p+1];
    int pointer_s=0;
    int pointer_l=0;
    for(int i=0;i<n;i++){
```

```

while(arr[i] - arr[pointer_s] >= w) pointer_s++;
while(arr[i] - arr[pointer_l] >= 2*w) pointer_l++;
//pointer_s is the last location a small camera is placed on the array such that all i are covered
//similar to pointer_l

for(int j = 0; j <= p; j++){
    if(j){
        if(pointer_s == 0) dp[i][j] = 0; //pretty obvious why. if pointer_s = 0, this means that it is reachable
        //using only small cam at arr[0]. so no big cams needed.
        else if(pointer_l == 0) dp[i][j] = min(dp[pointer_s-1][j-1], 1);
        //this one is a bit tougher but still fine. question is why dp[pointer_s-1][j-1]?
        // i know it is to compare 0 vs 1, ie the scenario where no big cams are needed vs 1 big cam, but
        //why dp[p_s-1][j-1] -> j-1 means one less small camera. thus, take away one p_s.
        //-> this is possible without adding a single big camera using the property of [j]
        //picking dp[p_s-1][j-1] would be the equivalent to saying that you pick one more j
        else dp[i][j] = min(dp[pointer_s-1][j-1], dp[pointer_l - 1][j]+1);
        //this one is because:
        // either you pick a large camera -> dp[q][j-1] +1
        // or you don't -> dp[s][j-1] -> no big camera added
    }
    else{
        if(pointer_l == 0) dp[i][j] = 1; //ie zero small cameras, so need 1 big cam
        else dp[i][j] = dp[pointer_l - 1][j] + 1; //ie you NEED to add one l, since there is no small camera
        //and pointer_l != 0 -> that means that you can't just place one and be done with it
    }
}
}
//find minimum large cams
int ans = 1e9;
for(int i = 0; i <= p; i++){
    ans = min(ans, dp[n-1][i]);
}
if(ans > q) return false; //is valid
else return true;
}

int main(){
    cin >> n >> p >> q;
    for(int i = 0; i < n; i++) cin >> arr[i];

    if(p+q >= n) {cout << 1; return 0;}

    sort(arr, arr+n);
    int l = 1, r = 1e9;
    while(r - l > 1){
        int mid = (r+l) >> 1;
        if(valid(mid)){
            r = mid;
        }
        else{
            l = mid;
        }
    }
    cout << r;
}

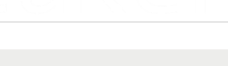
```

## ▼ Stamp

### stamps

Rar the Cat has diversified his operations of being selfish. I mean, selling fish. He has now decided to sell fish online, to other felines. As usual, being an online retailer, Rar has to mail his products to his consumers via post. This requires him to pay a certain amount of postage fees by placing stamps on

 <https://codebreaker.xyz/problem/stamps>



```

#include <bits/stdc++.h>
using namespace std;
#define int long long
#define pb push_back
int p, s;
vector<int> stamps;
pair<int, int> dp[110][10010]; //stores the number of stamps needed to get this value. min

signed main(){
    for(int j = 0; j < 10010; j++) dp[0][j] = {1e9, 0};
    cin >> p >> s;
    stamps.pb(0);
    for(int i = 1; i <= s; i++){
        stamps.pb(0);
    }
}

```



```

        cin >> stamps[i];
    }
    //sort(stamps.begin()+1, stamps.end());
    s = (int)(stamps.size()-1);
    for(int i =1;i<=s;i++){
        dp[i][0].first = 0;
        //empty set
        dp[i][0].second = 1;
    }

    for(int i =1;i<=s;i++){
        for(int cents = 1; cents <= p;cents++){
            if(cents >= stamps[i]){
                dp[i][cents].first = min(dp[i-1][cents].first, dp[i][cents-stamps[i]].first+1);
            }
            else dp[i][cents] = dp[i-1][cents];
            if(dp[i][cents].first == dp[i-1][cents].first) dp[i][cents].second = dp[i-1][cents].second;
            if(cents-stamps[i]>=0 and dp[i][cents].first == dp[i][cents - stamps[i]].first + 1) dp[i][cents].second += dp[i][cents - stamps[i]].second;
        }
    }

    int A=dp[s][p].first;/*
    for(int i=1;i<=s;i++){
        A = min(dp[i][p].first,A);
    }
    set<int>included;
    for(int i =1;i<=s;i++){
        if(dp[i][p].first == A) included.insert(dp[i][p].second);
    }
    */

    if(A==1e9) cout << "-1";
    else cout << A << "\n" << dp[s][p].second;
}

```

dp[i][j] state is defined as: minimum number of stamps of type 1,2,...i needed to add up to j cents.

FOR SOME REASON:

```

if(cents-stamps[i]>=0 and dp[i][cents].first == dp[i][cents - stamps[i]].first + 1) dp[i][cents].second += dp[i][cents - stamps[i]].second;

```

The (cents-stamps[i]≥0) here is very very important. I understand that without it, it is possible to access negative array, but from my experience, negative array defined outside main usually just gives 0. So lesson learnt: NEVER EVER DEAL with negative arrays.

```

^/ \
),k+1) {
    if (i>SZ(g[1])) continue;
    if (k-i>SZ(g[2])) continue;
    if (1ll*mn[1][x]*pre[1][i] + 1ll*mn[2][x]*pre[2][k-i] <= (1l)s) {
        hi = i;
        return true;
    }
}

```

#### Problem - D - Codeforces

Nura wants to buy  $k$  gadgets. She has only  $s$  burles for that. She can buy each gadget for dollars or for pounds. So each gadget is selling only for some type of currency. The type of currency and the cost in that currency are not changing.

 <https://codeforces.com/contest/609/problem/D>

**CODEFOR**  
Sponsored

[HCI NUSH 2022 Questions](#)

 [NOI Printed Notes](#)