Revisiting Questions:

**337: Minimum no of moves to make a palindrome (two pointer approach)**

You are given a string s consisting only of lowercase English letters.

In one **move**, you can select any two **adjacent** characters of s and swap them.

Return *the****minimum number of moves****needed to make* s *a palindrome*.

**Note** that the input will be generated such that s can always be converted to a palindrome.

**Example 1:**

**Input:** s = "aabb"

**Output:** 2

**Explanation:**

We can obtain two palindromes from s, "abba" and "baab".

- We can obtain "abba" from s in 2 moves: "a**ab**b" -> "ab**ab**" -> "abba".

- We can obtain "baab" from s in 2 moves: "a**ab**b" -> "**ab**ab" -> "baab".

Thus, the minimum number of moves needed to make s a palindrome is 2.

**Example 2:**

**Input:** s = "letelt"

**Output:** 2

**Explanation:**

One of the palindromes we can obtain from s in 2 moves is "lettel".

One of the ways we can obtain it is "lete**lt**" -> "let**et**l" -> "lettel".

Other palindromes such as "tleelt" can also be obtained in 2 moves.

It can be shown that it is not possible to obtain a palindrome in less than 2 moves.

**Constraints:**

* 1 <= s.length <= 2000
* s consists only of lowercase English letters.
* s can be converted to a palindrome using a finite number of moves.
* //not able to handle the test case 2 when it can swa eith previous ele
* /\*class Solution {
* public:
* bool ispalindrome(string s){
* int l=0,h=s.size()-1;
* while(l<=h){
* if(s[l]!=s[h]){
* return false;
* }
* l++,h--;
* }
* return true;
* }
* int solve(string s, int i){
* if(i>=s.size()-1){
* return 0;
* }
* swap(s[i],s[i+1]);
* if(ispalindrome(s)){
* return 0;
* }
* return 1+solve(s,i+1) ;
* swap(s[i],s[i+1]);
* return 1+solve(s,i+1) ;
* }
* int minMovesToMakePalindrome(string s) {
* return solve(s,0);
* }
* };\*/
* class Solution {
* public:
* int matching\_with\_lth\_ind(string s, int l, int k){
* while(k>l){
* if(s[k]==s[l]){
* return k;
* }
* k--;
* }
* return k;
* }
* int minMovesToMakePalindrome(string s) {
* int n = s.size();
* int l = 0, r = n - 1;
* int cnt = 0;
* while(l<r){
* if(s[l]==s[r]){
* l++,r--;
* }
* else{
* int k=r;
* k=matching\_with\_lth\_ind(s,l,k);
* if(k==l){
* swap(s[l],s[l+1]);
* cnt++;
* }
* else{
* while(k<r){
* swap(s[k],s[k+1]);
* cnt++;
* k++;
* }
* l++,r--;
* }
* }
* }
* return cnt;
* }
* };

**338: Unique Paths ||**

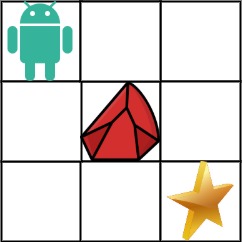
You are given an m x n integer array grid. There is a robot initially located at the **top-left corner** (i.e., grid[0][0]). The robot tries to move to the **bottom-right corner** (i.e., grid[m - 1][n - 1]). The robot can only move either down or right at any point in time.

An obstacle and space are marked as 1 or 0 respectively in grid. A path that the robot takes cannot include **any** square that is an obstacle.

Return *the number of possible unique paths that the robot can take to reach the bottom-right corner*.

The testcases are generated so that the answer will be less than or equal to 2 \* 109.

**Example 1:**



**Input:** obstacleGrid = [[0,0,0],[0,1,0],[0,0,0]]

**Output:** 2

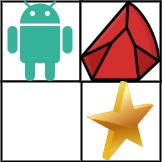
**Explanation:** There is one obstacle in the middle of the 3x3 grid above.

There are two ways to reach the bottom-right corner:

1. Right -> Right -> Down -> Down

2. Down -> Down -> Right -> Right

**Example 2:**



**Input:** obstacleGrid = [[0,1],[0,0]]

**Output:** 1

**Constraints:**

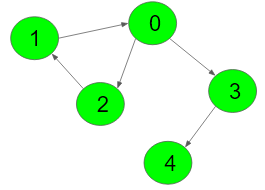
* m == obstacleGrid.length
* n == obstacleGrid[i].length
* 1 <= m, n <= 100
* obstacleGrid[i][j] is 0 or 1.
* //recursive solution
* /class Solution {
* public:
* int n,m;
* //dp[101][101];
* int solve(vector<vector<int>>& mat,int i ,int j){
* if(i>=n || j>=m || mat[i][j]==1){
* return 0;
* }
* if(i==n-1 && j==m-1){
* return 1;
* }
* return solve(mat,i+1,j)+solve(mat,i,j+1);
* }
* int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
* //memset(dp,-1,sizeof(dp));
* n=obstacleGrid.size();
* m=obstacleGrid[0].size();
* return solve(obstacleGrid,0,0);
* }
* };
* \*/
* //recursive solution + memoization
* //top down
* class Solution {
* public:
* int n,m;
* int dp[101][101];
* int solve(vector<vector<int>>& mat,int i ,int j){
* if(i>=n || j>=m || mat[i][j]==1){
* return 0;
* }
* if(i==n-1 && j==m-1){
* return 1;
* }
* if(dp[i][j]!=-1){
* return dp[i][j];
* }
* dp[i][j]=solve(mat,i+1,j)+solve(mat,i,j+1);
* return dp[i][j];
* }
* int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
* memset(dp,-1,sizeof(dp));
* n=obstacleGrid.size();
* m=obstacleGrid[0].size();
* return solve(obstacleGrid,0,0);
* }
* };\*/
* //bottom up;
* class Solution {
* public:
* int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
* int m = obstacleGrid.size(), n = obstacleGrid[0].size();
* vector<vector<int> > dp(m + 1, vector<int> (n + 1, 0));
* dp[0][1] = 1;
* for (int i = 1; i <= m; i++)
* for (int j = 1; j <= n; j++)
* if (!obstacleGrid[i - 1][j - 1])
* dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
* return dp[m][n];
* }
* };

**339: Mother Vertex**

Given a Directed Graph, find a Mother Vertex in the Graph (if present).   
A Mother Vertex is a vertex through which we can reach all the other vertices of the Graph.

**Example 1:**

**Input:**



**Output:** 0

**Explanation:** According to the given edges, all

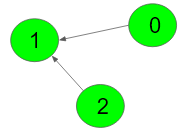
nodes can be reached from nodes from 0, 1 and 2.

But, since 0 is minimum among 0,1 and 2, so 0

is the output.

**Example 2:**

**Input:**



**Output:** -1

**Explanation:** According to the given edges,

no vertices are there from where we can

reach all vertices. So, output is -1.

**Your Task:**  
You don't need to read or print anything. Your task is to complete the function **findMotherVertex()**which takes V denoting the number of vertices and adjacency list as imput parameter and returns the verticex from through which we can traverse all other vertices of the graph. If there is more than one possible nodes then returns the node with minimum value.If not possible returns -1.  
  
**Expected Time Complexity:**O(V + E)  
**Expected Space Compelxity:**O(V)  
  
**Constraints:**  
1 ≤ V ≤ 500

class Solution

{

    void dfs(vector<int>adj[],vector<int>&visited,int node){

        visited[node]=1;

        for(auto &it:adj[node]){

            if(!visited[it]){

                dfs(adj,visited,it);

            }

        }

    }

    public:

    //Function to find a Mother Vertex in the Graph.

    int findMotherVertex(int V, vector<int>adj[])

    {

        // Code here

        for(int i=0;i<V;i++){

            vector<int>visited(V,0);

            int flag=1;

            dfs(adj,visited,i);

            for(int j=0;j<V;j++){

                if(visited[j]==0){

                    flag=0;

                    break;

                }

            }

            if(flag==1){

                return i;

            }

        }

        return -1;

    }

};

\*/

class Solution {

public:

    void dfs(vector<int> adj[], vector<int>& visited, int node) {

        visited[node] = 1;

        for (auto& it : adj[node]) {

            if (!visited[it]) {

                dfs(adj, visited, it);

            }

        }

    }

    int findMotherVertex(int V, vector<int> adj[]) {

        vector<int> vis(V, 0);

        int last\_fin\_vertex; // Remove the initialization

        // Finding last\_finished vertex

        for (int i = 0; i < V; i++) {

            if (vis[i] == 0) {

                dfs(adj, vis, i);

                last\_fin\_vertex = i;

            }

        }

        for (int i = 0; i < V; i++) {

            vis[i] = 0;

        }

        dfs(adj, vis, last\_fin\_vertex);

        for (int i = 0; i < V; i++) {

            if (vis[i] == 0) {

                return -1;

            }

        }

        return last\_fin\_vertex;

    }

};

**340: Min distance between two given nodes of binary tree**

Given a binary tree and two node values your task is to find the minimum distance between them.  
The given two nodes are guaranteed to be in the binary tree and nodes are numbered from 1 to N.  
Please Note that a and b are not always leaf node.

**Example 1:**

**Input:**

1

  / \

  2 3

a = 2, b = 3

**Output:** 2

**Explanation:** The tree formed is:

      1

     /   \

   2     3

We need the distance between 2 and 3.

Being at node 2, we need to take two

steps ahead in order to reach node 3.

The path followed will be:

2 -> 1 -> 3. Hence, the result is 2.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **findDist()**which takes the **root**node of the Tree and the two node values **a** and **b** as input parameters and returns the minimum distance between the nodes represented by the two given node values.

**Expected Time Complexity:**O(N).  
**Expected Auxiliary Space:**O(Height of the Tree).

**Constraints:**  
1 <= Number of nodes <= 104  
1 <= Data of a node <= 105

class Solution{

    public:

    Node \*find\_lca(Node\* root, int a, int b){

        if(root==NULL || root->data==a || root->data==b){

            return root;

        }

        Node \*left=find\_lca(root->left,a,b);

        Node \*right=find\_lca(root->right,a,b);

        if(left==NULL){

            return right;

        }

        else if(right==NULL){

            return left;

        }

        else{

            return root;

        }

    }

    int find\_level(Node\* root, int k, int level ){

        if(root==NULL){

            return -1;

        }

        if(root->data==k){

            return level;

        }

        int l=find\_level(root->left,k,level+1);

        if(l==-1){

            return find\_level(root->right,k,level+1);

        }

        return l;

    }

    int findDist(Node\* root, int a, int b) {

        //findind lca first

        Node \*lca=find\_lca(root,a,b);

        int d1=find\_level(lca,a,0);

        int d2=find\_level(lca,b,0);

        return d1+d2;

    }

};

**341: Same question but in bst:**

**Approach: similarly at first we find lca of the to given nodes and from lca we will add up the distance of both the nodes**

// CPP program to find distance between

// two nodes in BST

#include <bits/stdc++.h>

using namespace std;

struct Node {

    struct Node\* left, \*right;

    int key;

};

struct Node\* newNode(int key)

{

    struct Node\* ptr = new Node;

    ptr->key = key;

    ptr->left = ptr->right = NULL;

    return ptr;

}

// Standard BST insert function

struct Node\* insert(struct Node\* root, int key)

{

    if (!root)

        root = newNode(key);

    else if (root->key > key)

        root->left = insert(root->left, key);

    else if (root->key < key)

        root->right = insert(root->right, key);

    return root;

}

// This function returns distance of x from

// root. This function assumes that x exists

// in BST and BST is not NULL.

int distanceFromRoot(struct Node\* root, int x)

{

    if (root->key == x)

        return 0;

    else if (root->key > x)

        return 1 + distanceFromRoot(root->left, x);

    return 1 + distanceFromRoot(root->right, x);

}

// Returns minimum distance between a and b.

// This function assumes that a and b exist

// in BST.

int distanceBetween2(struct Node\* root, int a, int b)

{

    if (!root)

        return 0;

    // Both keys lie in left

    if (root->key > a && root->key > b)

        return distanceBetween2(root->left, a, b);

    // Both keys lie in right

    if (root->key < a && root->key < b) // same path

        return distanceBetween2(root->right, a, b);

    // Lie in opposite directions (Root is

    // LCA of two nodes)

    if (root->key >= a && root->key <= b)

        return distanceFromRoot(root, a) +

            distanceFromRoot(root, b);

}

// This function make sure that a is smaller

// than b before making a call to findDistWrapper()

int findDistWrapper(Node \*root, int a, int b)

{

if (a > b)

    swap(a, b);

return distanceBetween2(root, a, b);

}

// Driver code

int main()

{

    struct Node\* root = NULL;

    root = insert(root, 20);

    insert(root, 10);

    insert(root, 5);

    insert(root, 15);

    insert(root, 30);

    insert(root, 25);

    insert(root, 35);

    int a = 5, b = 55;

    cout << findDistWrapper(root, 5, 35);

    return 0;

}

**342: Roll the characters of the string**

Given a string **S** and an array roll where **roll[i]** represents rolling first **roll[i]** characters in the string, the task is to apply every roll[i] on the string and output the final string. Rolling means increasing the ASCII value of character, like rolling z would result in a, rolling b would result in c, etc.

**Example 1:**

**Input:** s = "bca"

roll[] = {1, 2, 3}

**Output:** eeb

**Explanation**: arr[0] = 1 means roll

first character of string -> cca

arr[1] = 2 means roll

first two characters of string -> dda

arr[2] = 3 means roll

first three characters of string -> eeb

So final ans is "eeb".

**Example 2:**

**Input:** s = "zcza"

roll[] = {1, 1, 3, 4}

**Output:** debb

**Your Task:**  
This is a function problem. You don't need to take any input, as it is already accomplished by the driver code. You just need to complete the function **findRollOut**() that takes String**S,**array**roll,**and integer**N**as parameters and returns the modified string.  
  
**Note-** The length of the array roll and length of the string are equal.

**Expected Time Complexity:** O(N).   
**Expected Auxiliary Space:** O(N).

class Solution{  \*\*TLE\*\*

    public:

    string findRollOut(string s, long long arr[], int n)

    {

        // Your code goes here

        for(int i=0;i<n;i++){

            for(int j=0;j<arr[i];j++){

                int x=s[j];

                if(s[j]=='z'){

                    s[j]='a';

                }

                else{

                    int x=s[j]+1;

                    char c=char(x);

                    s[j]=c;

                }

            }

        }

        return s;

    }

};\*/

class Solution{

public:

    string findRollOut(string s, long long arr[], int n)

        {

            long long int hash[n+2] = {0};

            for(int i=0; i<n; i++)

                hash[arr[i]]++;

            for(int i=n; i>=0; i--)

                hash[i] += hash[i+1];

            for(int i=1; i<=n; i++)

                s[i-1] = ((s[i-1]-'a')+hash[i])%26 + 'a';

            return s;

        }

};

**343: Shifting letters**

You are given a string s of lowercase English letters and an integer array shifts of the same length.

Call the shift() of a letter, the next letter in the alphabet, (wrapping around so that 'z' becomes 'a').

* For example, shift('a') = 'b', shift('t') = 'u', and shift('z') = 'a'.

Now for each shifts[i] = x, we want to shift the first i + 1 letters of s, x times.

Return *the final string after all such shifts to s are applied*.

**Example 1:**

**Input:** s = "abc", shifts = [3,5,9]

**Output:** "rpl"

**Explanation:** We start with "abc".

After shifting the first 1 letters of s by 3, we have "dbc".

After shifting the first 2 letters of s by 5, we have "igc".

After shifting the first 3 letters of s by 9, we have "rpl", the answer.

**Example 2:**

**Input:** s = "aaa", shifts = [1,2,3]

**Output:** "gfd"

**Constraints:**

* 1 <= s.length <= 105
* s consists of lowercase English letters.
* shifts.length == s.length
* 0 <= shifts[i] <= 109
* class Solution {
* public:
* string shiftingLetters(string s, vector<int>& shifts) {
* int n=shifts.size();
* vector<long long>pre(n,0);
* pre[0]=shifts[0];
* for(int i=1;i<n;i++){
* pre[i]=pre[i-1]+shifts[i];
* }
* int z=((s[0]-'a')+pre[n-1])%26;
* s[0]=char(z+'a');
* for(int i=1;i<n;i++){
* z=((s[i]-'a')+pre[n-1]-pre[i-1])%26;
* s[i]=char(z+'a');
* }
* return s;
* }
* };
* //2nd solution
* class Solution {
* public:
* string shiftingLetters2(string S, vector<int> shifts) {
* for (int i = shifts.size() - 2; i >= 0; i--){
* shifts[i] = (shifts[i] + shifts[i + 1]) % 26;
* }
* for (int i = 0; i < shifts.size(); i++){
* S[i] = (S[i] - 'a' + shifts[i]) % 26 + 'a';
* }
* return S;
* }
* };

**343: Minimum Absolute difference between elements with constraint**

You are given a **0-indexed** integer array nums and an integer x.

Find the **minimum absolute difference** between two elements in the array that are at least x indices apart.

In other words, find two indices i and j such that abs(i - j) >= x and abs(nums[i] - nums[j]) is minimized.

Return*an integer denoting the****minimum****absolute difference between two elements that are at least* x *indices apart*.

**Example 1:**

**Input:** nums = [4,3,2,4], x = 2

**Output:** 0

**Explanation:** We can select nums[0] = 4 and nums[3] = 4.

They are at least 2 indices apart, and their absolute difference is the minimum, 0.

It can be shown that 0 is the optimal answer.

**Example 2:**

**Input:** nums = [5,3,2,10,15], x = 1

**Output:** 1

**Explanation:** We can select nums[1] = 3 and nums[2] = 2.

They are at least 1 index apart, and their absolute difference is the minimum, 1.

It can be shown that 1 is the optimal answer.

**Example 3:**

**Input:** nums = [1,2,3,4], x = 3

**Output:** 3

**Explanation:** We can select nums[0] = 1 and nums[3] = 4.

They are at least 3 indices apart, and their absolute difference is the minimum, 3.

It can be shown that 3 is the optimal answer.

**Constraints:**

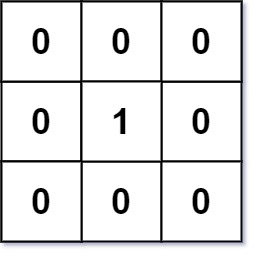
* 1 <= nums.length <= 105
* 1 <= nums[i] <= 109
* 0 <= x < nums.length
* //Tle
* /\*class Solution {
* public:
* int minAbsoluteDifference(vector<int>& nums, int x) {
* int n=nums.size();
* int mini=INT\_MAX;
* for(int i=0;i<n-x;i++){
* for(int j=i+x;j<n;j++){
* mini=min(mini,abs(nums[i]-nums[j]));
* }
* }
* return mini;
* }
* };\*/
* /\*class Solution {
* public:
* int minAbsoluteDifference(vector<int>& nums, int x) {
* int ans = INT\_MAX;
* set<int> s;
* for(int i = x; i < nums.size(); ++i){
* s.insert(nums[i-x]);
* auto it = s.upper\_bound(nums[i]);
* if(it != s.end()) ans = min(ans, abs(nums[i] - \*it));
* if(it != s.begin()) ans = min(ans, abs(nums[i] - \*prev(it)));
* }
* return ans;
* }
* };\*/
* int minAbsoluteDifference(vector<int>& nums, int x) {
* int n=nums.size(),ans=INT\_MAX;
* map<int,int>mp;
* for(int i=x;i<n;i++) mp[nums[i]]++;
* for(int i=0;i<n-x;i++)
* {
* auto it=mp.lower\_bound(nums[i]);
* ans=min(ans,abs(nums[i]-it->first));
* if (it != mp.begin()) {
* it--;
* ans = min(ans, abs(nums[i] - it->first));
* }
* mp[nums[i+x]]--;
* if(mp[nums[i+x]]==0) mp.erase(nums[i+x]);
* }
* return ans;
* }

**344: 01 matrix**

Given an m x n binary matrix mat, return *the distance of the nearest*0*for each cell*.

The distance between two adjacent cells is 1.

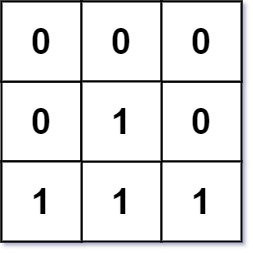
**Example 1:**



**Input:** mat = [[0,0,0],[0,1,0],[0,0,0]]

**Output:** [[0,0,0],[0,1,0],[0,0,0]]

**Example 2:**



**Input:** mat = [[0,0,0],[0,1,0],[1,1,1]]

**Output:** [[0,0,0],[0,1,0],[1,2,1]]

**Constraints:**

* m == mat.length
* n == mat[i].length
* 1 <= m, n <= 104
* 1 <= m \* n <= 104
* mat[i][j] is either 0 or 1.
* There is at least one 0 in mat.
* //memory limit exceeded(48/50)
* class Solution {
* public:
* int n,m;
* int bfs(vector<vector<int>>& mat,int i,int j){
* queue<pair<pair<int,int>,int>>q;
* int drow[]={-1,0,1,0};
* int dcol[]={0,1,0,-1};
* q.push({{i,j},0});
* int dis;
* int ans;
* while(!q.empty()){
* int r=q.front().first.first;
* int c=q.front().first.second;
* int dis=q.front().second;
* q.pop();
* if(mat[r][c]==0){
* ans=dis;
* break;
* }
* for(int i=0;i<4;i++){
* int nrow=r+drow[i];
* int ncol=c+dcol[i];
* if(nrow<n && nrow>=0 && ncol<m && ncol>=0){
* q.push({{nrow,ncol},dis+1});
* }
* }
* }
* return ans;
* }
* vector<vector<int>> updateMatrix(vector<vector<int>>& mat) {
* n=mat.size();
* m=mat[0].size();
* vector<vector<int>>ans;
* for(int i=0;i<n;i++){
* vector<int>temp;
* for(int j=0;j<m;j++){
* if(mat[i][j]==0){
* temp.push\_back(0);
* }
* else{
* int d=bfs(mat,i,j);
* temp.push\_back(d);
* }
* }
* ans.push\_back(temp);
* }
* return ans;
* }
* };

Sol2:

* class Solution {
* public:
* int n, m;
* vector<vector<int>> updateMatrix(vector<vector<int>>& mat) {
* n = mat.size();
* m = mat[0].size();
* vector<vector<int>> ans(n, vector<int>(m, INT\_MAX));
* queue<pair<int, int>> q;
* for (int i = 0; i < n; i++) {
* for (int j = 0; j < m; j++) {
* if (mat[i][j] == 0) {
* ans[i][j] = 0;
* q.push({i, j});
* }
* }
* }
* int drow[] = {-1, 0, 1, 0};
* int dcol[] = {0, 1, 0, -1};
* while (!q.empty()) {
* int r = q.front().first;
* int c = q.front().second;
* q.pop();
* for (int i = 0; i < 4; i++) {
* int nrow = r + drow[i];
* int ncol = c + dcol[i];
* if (nrow >= 0 && nrow < n && ncol >= 0 && ncol < m && ans[nrow][ncol] > ans[r][c] + 1) {
* ans[nrow][ncol] = ans[r][c] + 1;
* q.push({nrow, ncol});
* }
* }
* }
* return ans;
* }
* };

**345: Reorganize string (priority queue)**

Given a string s, rearrange the characters of s so that any two adjacent characters are not the same.

Return *any possible rearrangement of* s *or return* "" *if not possible*.

**Example 1:**

**Input:** s = "aab"

**Output:** "aba"

**Example 2:**

**Input:** s = "aaab"

**Output:** ""

**Constraints:**

* 1 <= s.length <= 500
* s consists of lowercase English letters.
* class Solution {  //Tle
* public:
* void solve(string &s, string &ds, vector<string> &ans, vector<int> &fre) {
* if (ds.size() == s.size()) {
* ans.push\_back(ds);
* return;
* }
* for (int i = 0; i < s.size(); i++) {
* if (!fre[i]) {
* fre[i] = 1; // Mark the index as visited
* ds.push\_back(s[i]);
* solve(s, ds, ans, fre);
* fre[i] = 0; // Backtrack: mark the index as not visited
* ds.pop\_back(); // Backtrack: remove the last character
* }
* }
* }
* string reorganizeString(string s) {
* int n = s.size();
* string ds;
* vector<string> ans;
* vector<int> fre(n, 0);
* solve(s, ds, ans, fre);
* for (int i = 0; i < ans.size(); i++) {
* string str = ans[i];
* int flag = true;
* for (int j = 0; j < str.size() - 1; j++) {
* if (str[j] == str[j + 1]) { // Compare characters, not their differences
* flag = false;
* break;
* }
* }
* if (flag == true) {
* return str;
* }
* }
* return "";
* }
* };

class Solution {

public:

    string reorganizeString(string s) {

        unordered\_map<char, int> mp;

        for (char c : s) {

            mp[c]++;

        }

        priority\_queue<pair<int, char>> pq;

        for (auto &it : mp) {

            pq.push({it.second, it.first}); // Push frequency and character

        }

        string str = "";

        while (pq.size() >= 2) {

            auto it1 = pq.top();

            int fre1 = it1.first;

            char c1 = it1.second;

            pq.pop();

            auto it2 = pq.top();

            int fre2 = it2.first;

            char c2 = it2.second;

            pq.pop();

            str += c1;

            str += c2;

            fre1--;

            fre2--;

            if (fre1 > 0) {

                pq.push({fre1, c1});

            }

            if (fre2 > 0) {

                pq.push({fre2, c2});

            }

        }

        if (!pq.empty()) {

            auto it = pq.top();

            int fre = it.first;

            char c = it.second;

            if (fre > 1) {

                return "";

            }

            str += c;

        }

        return str;

    }

};

**346: Frog Jump (water-stone)**

A frog is crossing a river. The river is divided into some number of units, and at each unit, there may or may not exist a stone. The frog can jump on a stone, but it must not jump into the water.

Given a list of stones' positions (in units) in sorted **ascending order**, determine if the frog can cross the river by landing on the last stone. Initially, the frog is on the first stone and assumes the first jump must be 1 unit.

If the frog's last jump was k units, its next jump must be either k - 1, k, or k + 1 units. The frog can only jump in the forward direction.

**Example 1:**

**Input:** stones = [0,1,3,5,6,8,12,17]

**Output:** true

**Explanation:** The frog can jump to the last stone by jumping 1 unit to the 2nd stone, then 2 units to the 3rd stone, then 2 units to the 4th stone, then 3 units to the 6th stone, 4 units to the 7th stone, and 5 units to the 8th stone.

**Example 2:**

**Input:** stones = [0,1,2,3,4,8,9,11]

**Output:** false

**Explanation:** There is no way to jump to the last stone as the gap between the 5th and 6th stone is too large.

**Constraints:**

* 2 <= stones.length <= 2000
* 0 <= stones[i] <= 231 - 1
* stones[0] == 0
* stones is sorted in a strictly increasing order.
* lass Solution {
* public:
* int n;
* unordered\_map<int, int> mp;
* int t[2001][2001];
* bool solve(vector<int>& stones, int curr\_stone\_index, int prevJump) {
* if(curr\_stone\_index == n-1)
* return true;
* if(t[curr\_stone\_index][prevJump] != -1)
* return t[curr\_stone\_index][prevJump];
* bool result=false;
* for(int nextJump = prevJump-1; nextJump <= prevJump+1; nextJump++) {
* if(nextJump > 0) {
* int next\_stone = stones[curr\_stone\_index] + nextJump;
* if(mp.find(next\_stone) != mp.end()) {
* result = result || solve(stones, mp[next\_stone], nextJump);
* }
* }
* }
* return t[curr\_stone\_index][prevJump] = result;
* }
* bool canCross(vector<int>& stones) {
* if(stones[1] != 1)
* return false;
* n = stones.size();
* for(int i = 0; i<n; i++) {
* mp[stones[i]] = i;
* }
* memset(t, -1, sizeof(t));
* return solve(stones, 0, 0);
* }
* };

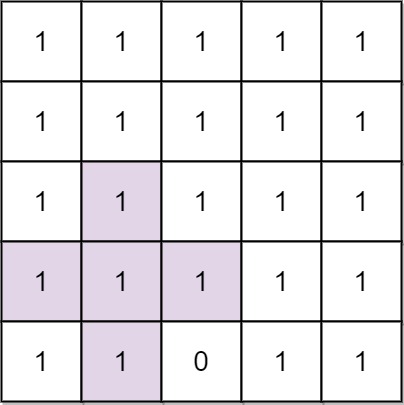
**347: Largest Plus sign**

You are given an integer n. You have an n x n binary grid grid with all values initially 1's except for some indices given in the array mines. The ith element of the array mines is defined as mines[i] = [xi, yi] where grid[xi][yi] == 0.

Return *the order of the largest****axis-aligned****plus sign of*1*'s contained in*grid. If there is none, return 0.

An **axis-aligned plus sign** of 1's of order k has some center grid[r][c] == 1 along with four arms of length k - 1 going up, down, left, and right, and made of 1's. Note that there could be 0's or 1's beyond the arms of the plus sign, only the relevant area of the plus sign is checked for 1's.

**Example 1:**



**Input:** n = 5, mines = [[4,2]]

**Output:** 2

**Explanation:** In the above grid, the largest plus sign can only be of order 2. One of them is shown.

**Example 2:**



**Input:** n = 1, mines = [[0,0]]

**Output:** 0

**Explanation:** There is no plus sign, so return 0.

**Constraints:**

* 1 <= n <= 500
* 1 <= mines.length <= 5000
* 0 <= xi, yi < n
* All the pairs (xi, yi) are **unique**.

class Solution {

public:

    int orderOfLargestPlusSign(int n, vector<vector<int>>& mines) {

    vector<vector<int>> mat(n, vector<int>(n, 1));

    for(auto it : mines)

    {

        int x = it[0] ;

        int y = it[1] ;

        mat[x][y] = 0;

    }

    vector<vector<int>> lef, rig, top, bot;

    lef = rig = top = bot = mat;

    for(int i=0;i<n;i++)

    {

        for(int j=0;j<n;j++)

        {

            int x = n - i - 1;

            int y = n - j - 1;

            if( (i>0)   && top[i][j] ) top[i][j] += top[i-1][j];

            if( (j>0)   && lef[i][j] ) lef[i][j] += lef[i][j-1];

            if( (x<n-1) && bot[x][y] ) bot[x][y] += bot[x+1][y];

            if( (y<n-1) && rig[x][y] ) rig[x][y] += rig[x][y+1];

        }

    }

    int ans = 0;

    for(int i=0;i<n;i++)

        for(int j=0;j<n;j++)

            ans = max(ans, min({lef[i][j], rig[i][j], top[i][j], bot[i][j]}));

    return ans;

}

};

**348: Minimum Remove to make Valid Parenthesis**

Given a string s of '(' , ')' and lowercase English characters.

Your task is to remove the minimum number of parentheses ( '(' or ')', in any positions ) so that the resulting *parentheses string* is valid and return **any** valid string.

Formally, a *parentheses string* is valid if and only if:

* It is the empty string, contains only lowercase characters, or
* It can be written as AB (A concatenated with B), where A and B are valid strings, or
* It can be written as (A), where A is a valid string.

**Example 1:**

**Input:** s = "lee(t(c)o)de)"

**Output:** "lee(t(c)o)de"

**Explanation:** "lee(t(co)de)" , "lee(t(c)ode)" would also be accepted.

**Example 2:**

**Input:** s = "a)b(c)d"

**Output:** "ab(c)d"

**Example 3:**

**Input:** s = "))(("

**Output:** ""

**Explanation:** An empty string is also valid.

**Constraints:**

* 1 <= s.length <= 105
* s[i] is either'(' , ')', or lowercase English letter.
* class Solution {
* public:
* string minRemoveToMakeValid(string s) {
* string str;
* int n = s.size();
* stack<int> st; // Store indices of opening brackets
* for (int i = 0; i < n; i++) {
* if (isalpha(s[i])) {
* str.push\_back(s[i]);
* } else if (s[i] == '{' || s[i] == '(' || s[i] == '[') {
* str.push\_back(s[i]);
* st.push(str.size() - 1); // Store the index in the updated string
* } else if ((s[i] == '}' || s[i] == ')' || s[i] == ']') && !st.empty()) {
* str.push\_back(s[i]);
* st.pop(); // Pop the index of the matching opening bracket
* }
* }
* // Mark characters to be removed
* while (!st.empty()) {
* int index = st.top();
* str[index] = '\*'; // Mark the character for removal
* st.pop();
* }
* // Construct the resulting string without marked characters
* string result;
* for (char c : str) {
* if (c != '\*') {
* result.push\_back(c);
* }
* }
* return result;
* }
* };

**349: Minimum no of swaps to make the string balanced**

You are given a **0-indexed** string s of **even** length n. The string consists of **exactly** n / 2 opening brackets '[' and n / 2 closing brackets ']'.

A string is called **balanced** if and only if:

* It is the empty string, or
* It can be written as AB, where both A and B are **balanced** strings, or
* It can be written as [C], where C is a **balanced** string.

You may swap the brackets at **any** two indices **any** number of times.

Return *the****minimum****number of swaps to make*s ***balanced***.

**Example 1:**

**Input:** s = "][]["

**Output:** 1

**Explanation:** You can make the string balanced by swapping index 0 with index 3.

The resulting string is "[[]]".

**Example 2:**

**Input:** s = "]]][[["

**Output:** 2

**Explanation:** You can do the following to make the string balanced:

- Swap index 0 with index 4. s = "[]][][".

- Swap index 1 with index 5. s = "[[][]]".

The resulting string is "[[][]]".

**Example 3:**

**Input:** s = "[]"

**Output:** 0

**Explanation:** The string is already balanced.

**Constraints:**

* n == s.length
* 2 <= n <= 106
* n is even.
* s[i] is either '[' or ']'.
* The number of opening brackets '[' equals n / 2, and the number of closing brackets ']' equals n / 2.

**Intuition and approach**

* First of all it is guarenteed that total number of '[' brackets is equal to total number of ']' brackets.
* So, for every bracket element we are sure that its counterpart exist.
* Also, to keep string balanced, at every i
  + *count of '[' >= count of ']'*
  + If at any moment *count of ']'* exceeds *count of '['*, this is the time when we need to do swaping.

As of now that we have noted few important observations, lets move to algorithm part.

1. To track if brackets are balanced at any index i, lets take a variable *balance*
   * balance = 0 , initially
2. When we see a opening bracket, we increament balance by one.
   * As we expect a closing bracket later who will balance it . '[' -> 1
3. Else When we see closing bracket, we decrement balance by one so that number
   * this will balance the opening bracket that occured previously. ']' -> -1
   * string: [ [ ] ]
   * balance: 1 2 1 0
   * (at last balance is zero which means all elements are matched)

1. Now, the main catch where we need to swap. When number of closing brackets exceed the already occured opening brackets. This is where we need to do swapping.
   * when *balance < 0*, or *balance == -1* , swap this element with first occuring '[' from last.
   * Doing this we change our current balance to 1 and now we can move further in search of closing brackets.
2. As soon as we swap the brackets, we need to again set balance to 1 as current bracket is opening -> '[' = 1

**Sol**

class Solution {

public:

    int minSwaps(string s) {

        int n=s.size();

        int bal=0,i=0,j=n-1,swaps=0;

        for(int i=0;i<n;i++){

            if(s[i]=='['){

                bal++;

            }

            else{

                bal--;

            }

            if(bal<0){

                while(i<j && s[j]!='['){

                    j--;

                }

                swaps++;

                swap(s[i],s[j]);

                bal=1;

            }

        }

        return swaps;

    }

};

**350: Remove Invalid Parenthesis**

Given a string s that contains parentheses and letters, remove the minimum number of invalid parentheses to make the input string valid.

Return *a list of****unique strings****that are valid with the minimum number of removals*. You may return the answer in **any order**.

**Example 1:**

**Input:** s = "()())()"

**Output:** ["(())()","()()()"]

**Example 2:**

**Input:** s = "(a)())()"

**Output:** ["(a())()","(a)()()"]

**Example 3:**

**Input:** s = ")("

**Output:** [""]

**Constraints:**

* 1 <= s.length <= 25
* s consists of lowercase English letters and parentheses '(' and ')'.
* There will be at most 20 parentheses in s.

class Solution {

public:

    vector<string>ans;

    unordered\_map<string,int>mp;//Just to prevent by taking duplicate string in ans

    void solve(string s,int min\_removals){

        if(mp[s]!=0){

            return ;

        }

        else{

            mp[s]++;

        }

        if(min\_removals==0){

            int min\_removals\_now=getminimum\_invalid(s);

            if(min\_removals\_now == 0){ //if we get this 0, it                                      means we got valid string

                ans.push\_back(s);

            }

            return;

        }

        for(int i=0;i<s.size();i++){

            string left=s.substr(0,i);//taken as it runs till (i-1) and we are skipping ith bracket

            string right=s.substr(i+1); //from i+1 to end or s.size

            solve(left+right,min\_removals-1);

        }

        return ;

    }

    int getminimum\_invalid(string &s){

        stack<char>st;

        int i=0,n=s.size();

        while(i<n){

            if(s[i]=='('){

                st.push('(');

            }

            else if(s[i]==')'){

                if(!st.empty() && st.top()=='('){

                    st.pop();

                }

                else{

                    st.push(')');

                }

            }

            i++;

        }

        return st.size();

    }

    vector<string> removeInvalidParentheses(string s) {

        int min\_removals = getminimum\_invalid( s);

        solve(s,min\_removals);

        return ans;

    }

};

**351: Minimum penalty for a shop**

You are given the customer visit log of a shop represented by a **0-indexed** string customers consisting only of characters 'N' and 'Y':

* if the ith character is 'Y', it means that customers come at the ith hour
* whereas 'N' indicates that no customers come at the ith hour.

If the shop closes at the jth hour (0 <= j <= n), the **penalty** is calculated as follows:

* For every hour when the shop is open and no customers come, the penalty increases by 1.
* For every hour when the shop is closed and customers come, the penalty increases by 1.

Return*the****earliest****hour at which the shop must be closed to incur a****minimum****penalty.*

**Note** that if a shop closes at the jth hour, it means the shop is closed at the hour j.

**Example 1:**

**Input:** customers = "YYNY"

**Output:** 2

**Explanation:**

- Closing the shop at the 0th hour incurs in 1+1+0+1 = 3 penalty.

- Closing the shop at the 1st hour incurs in 0+1+0+1 = 2 penalty.

- Closing the shop at the 2nd hour incurs in 0+0+0+1 = 1 penalty.

- Closing the shop at the 3rd hour incurs in 0+0+1+1 = 2 penalty.

- Closing the shop at the 4th hour incurs in 0+0+1+0 = 1 penalty.

Closing the shop at 2nd or 4th hour gives a minimum penalty. Since 2 is earlier, the optimal closing time is 2.

**Example 2:**

**Input:** customers = "NNNNN"

**Output:** 0

**Explanation:** It is best to close the shop at the 0th hour as no customers arrive.

**Example 3:**

**Input:** customers = "YYYY"

**Output:** 4

**Explanation:** It is best to close the shop at the 4th hour as customers arrive at each hour.

//intuition prefix sum

class Solution {

public:

    int bestClosingTime(string customers) {

        int max\_score = 0, score = 0, best\_hour = -1;

        for(int i = 0; i < customers.size(); ++i) {

            score += (customers[i] == 'Y') ? 1 : -1;

            if(score > max\_score) {

                max\_score = score;

                best\_hour = i;

            }

        }

        return best\_hour + 1;

    }

};

**352: Mimum no of taps to open to water a garden**

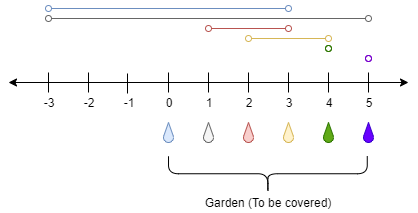
There is a one-dimensional garden on the x-axis. The garden starts at the point 0 and ends at the point n. (i.e The length of the garden is n).

There are n + 1 taps located at points [0, 1, ..., n] in the garden.

Given an integer n and an integer array ranges of length n + 1 where ranges[i] (0-indexed) means the i-th tap can water the area [i - ranges[i], i + ranges[i]] if it was open.

Return *the minimum number of taps* that should be open to water the whole garden, If the garden cannot be watered return **-1**.

**Example 1:**



**Input:** n = 5, ranges = [3,4,1,1,0,0]

**Output:** 1

**Explanation:** The tap at point 0 can cover the interval [-3,3]

The tap at point 1 can cover the interval [-3,5]

The tap at point 2 can cover the interval [1,3]

The tap at point 3 can cover the interval [2,4]

The tap at point 4 can cover the interval [4,4]

The tap at point 5 can cover the interval [5,5]

Opening Only the second tap will water the whole garden [0,5]

**Example 2:**

**Input:** n = 3, ranges = [0,0,0,0]

**Output:** -1

**Explanation:** Even if you activate all the four taps you cannot water the whole garden.

**Constraints:**

* 1 <= n <= 104
* ranges.length == n + 1
* 0 <= ranges[i] <= 100

//sol1:using recursion and memoization

class Solution {

public:

    map<pair<int,int>,int> mp;

    int N;

    int solve(int i, int maxEnd, vector<pair<int,int>>& range) {

        if(i >= range.size())

            return maxEnd >= N ? 0 : 1e9;

        if(range[i].first > maxEnd)

            return 1e9;

        if(mp.find({i,maxEnd}) != mp.end())

            return mp[{i,maxEnd}];

        int take = 1 + solve(i+1, max(maxEnd, range[i].second), range);

        int notake = solve(i+1, maxEnd , range);

        return mp[{i, maxEnd}] = min(take,notake);

    }

    int minTaps(int n, vector<int>& ranges) {

        N = n;

        vector<pair<int,int>> range;

        for(int i = 0; i < ranges.size(); i++) {

            int start = max(0, i - ranges[i]);

            int end   = min(n, i + ranges[i]);

            range.push\_back({start, end});

        }

        sort(range.begin(), range.end());

        int ans = solve(0, 0, range);

        return ans == 1e9 ? -1 : ans;

    }

};

* //solution2 using greedy
* class Solution {
* public:
* int minTaps(int n, vector<int>& ranges) {
* vector<int>vec(n+1,0);
* for(int i=0;i<n+1;i++){
* int start=max(0,i-ranges[i]);
* int end=min(n,i+ranges[i]);
* vec[start]=max(vec[start],end);
* }
* //implying similar logic to jump game2
* int taps=0,maxend=0,currend=0;
* for(int i=0;i<n+1;i++){
* if(i>maxend){
* return -1;
* }
* if(i>currend){
* taps++;
* currend=maxend;
* }
* maxend=max(maxend,vec[i]);
* }
* return taps;
* }
* };

**353: Extra characters in a string**

You are given a **0-indexed** string s and a dictionary of words dictionary. You have to break s into one or more **non-overlapping** substrings such that each substring is present in dictionary. There may be some **extra characters** in s which are not present in any of the substrings.

Return *the****minimum****number of extra characters left over if you break up*s*optimally.*

**Example 1:**

**Input:** s = "leetscode", dictionary = ["leet","code","leetcode"]

**Output:** 1

**Explanation:** We can break s in two substrings: "leet" from index 0 to 3 and "code" from index 5 to 8. There is only 1 unused character (at index 4), so we return 1.

**Example 2:**

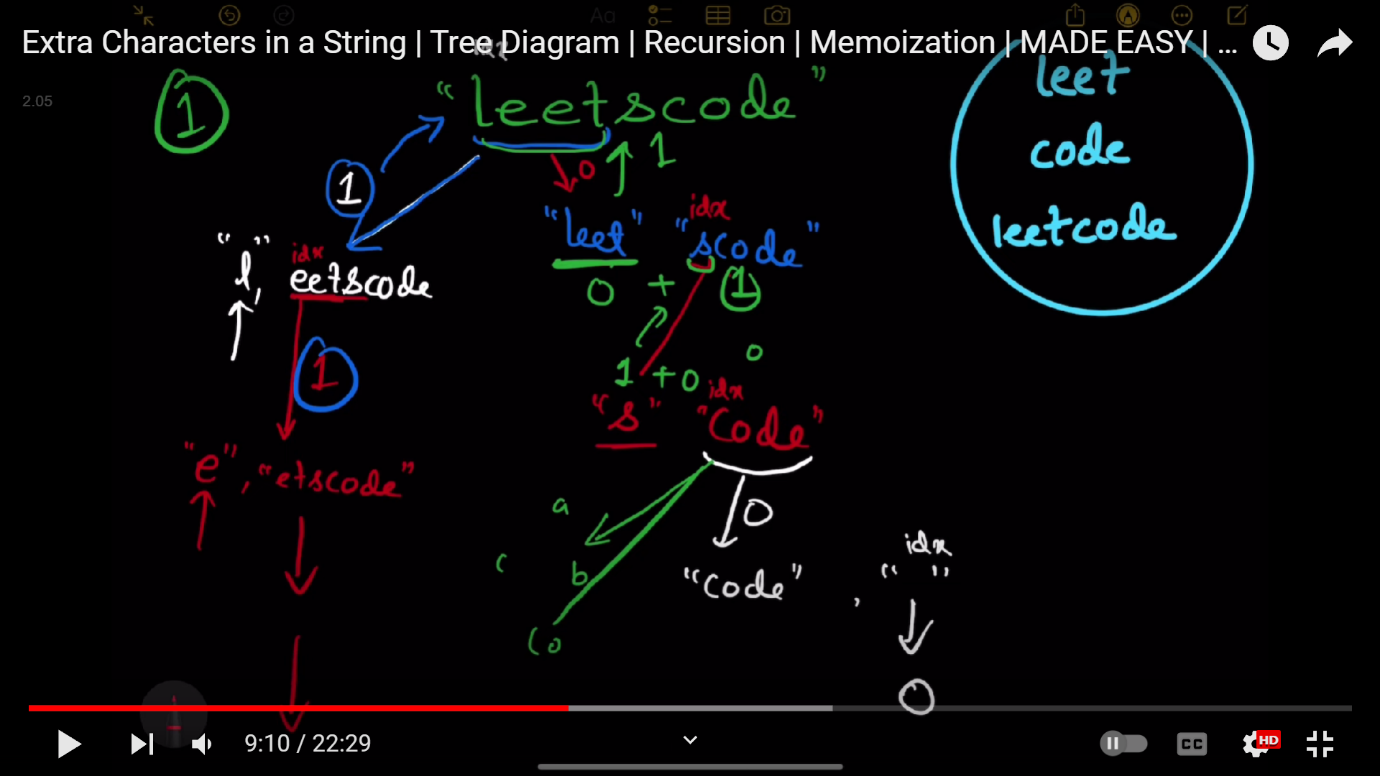
**Input:** s = "sayhelloworld", dictionary = ["hello","world"]

**Output:** 3

**Explanation:** We can break s in two substrings: "hello" from index 3 to 7 and "world" from index 8 to 12. The characters at indices 0, 1, 2 are not used in any substring and thus are considered as extra characters. Hence, we return 3.

**Constraints:**

* 1 <= s.length <= 50
* 1 <= dictionary.length <= 50
* 1 <= dictionary[i].length <= 50
* dictionary[i] and s consists of only lowercase English letters
* dictionary contains distinct words



class Solution {

public:

    unordered\_set<string>st;

    int dp[51];

    int solve(string &s,int idx){

        if(idx>=s.size()){

            return 0;

        }

        if(dp[idx]!=-1){

            return dp[idx];

        }

        string currstr="";

        int minextra=s.size();

        for(int i=idx;i<s.size();i++){

            currstr.push\_back(s[i]);

            int currextra=0;

            if(st.find(currstr)!=st.end()){

                currextra=0;

            }

            else{

                currextra=currstr.size();

            }

            int nextextra=solve(s,i+1);

            int totalextra=currextra+nextextra;

            minextra=min(minextra,totalextra);

        }

        return dp[idx]= minextra;

    }

    int minExtraChar(string s, vector<string>& dictionary) {

        int n=dictionary.size();

        memset(dp,-1,sizeof(dp));

        for(int i=0;i<n;i++){

            st.insert(dictionary[i]);

        }

        return solve(s,0);

    }

};

**354: Minimum replacements to sort the array**

You are given a **0-indexed** integer array nums. In one operation you can replace any element of the array with **any two** elements that **sum** to it.

* For example, consider nums = [5,6,7]. In one operation, we can replace nums[1] with 2 and 4 and convert nums to [5,2,4,7].

Return *the minimum number of operations to make an array that is sorted in****non-decreasing****order*.

**Example 1:**

**Input:** nums = [3,9,3]

**Output:** 2

**Explanation:** Here are the steps to sort the array in non-decreasing order:

- From [3,9,3], replace the 9 with 3 and 6 so the array becomes [3,3,6,3]

- From [3,3,6,3], replace the 6 with 3 and 3 so the array becomes [3,3,3,3,3]

There are 2 steps to sort the array in non-decreasing order. Therefore, we return 2.

**Example 2:**

**Input:** nums = [1,2,3,4,5]

**Output:** 0

**Explanation:** The array is already in non-decreasing order. Therefore, we return 0.

**Constraints:**

* 1 <= nums.length <= 105
* 1 <= nums[i] <= 109
* //greedy solution
* class Solution {
* public:
* long long minimumReplacement(vector<int>& nums) {
* int n=nums.size();
* if(n==1){
* return 0;
* }
* long long ans=0;
* int right=nums[n-1],cuts=0;
* for(int i=n-2;i>=0;i--){
* if(right>=nums[i]){
* right=nums[i];
* }
* else{
* cuts=((nums[i]-1)/right)+1;
* right=nums[i]/cuts;
* ans+=(cuts-1);
* }
* }
* return ans;
* }
* };

**357: Maximum sum of almost unique subarray**

You are given an integer array nums and two positive integers m and k.

Return *the****maximum sum****out of all****almost unique****subarrays of length*k*of* nums. If no such subarray exists, return 0.

A subarray of nums is **almost unique** if it contains at least m distinct elements.

A subarray is a contiguous **non-empty** sequence of elements within an array.

**Example 1:**

**Input:** nums = [2,6,7,3,1,7], m = 3, k = 4

**Output:** 18

**Explanation:** There are 3 almost unique subarrays of size k = 4. These subarrays are [2, 6, 7, 3], [6, 7, 3, 1], and [7, 3, 1, 7]. Among these subarrays, the one with the maximum sum is [2, 6, 7, 3] which has a sum of 18.

**Example 2:**

**Input:** nums = [5,9,9,2,4,5,4], m = 1, k = 3

**Output:** 23

**Explanation:** There are 5 almost unique subarrays of size k. These subarrays are [5, 9, 9], [9, 9, 2], [9, 2, 4], [2, 4, 5], and [4, 5, 4]. Among these subarrays, the one with the maximum sum is [5, 9, 9] which has a sum of 23.

**Example 3:**

**Input:** nums = [1,2,1,2,1,2,1], m = 3, k = 3

**Output:** 0

**Explanation:** There are no subarrays of size k = 3 that contain at least m = 3 distinct elements in the given array [1,2,1,2,1,2,1]. Therefore, no almost unique subarrays exist, and the maximum sum is 0.

**Constraints:**

* 1 <= nums.length <= 2 \* 104
* 1 <= m <= k <= nums.length
* 1 <= nums[i] <= 109

class Solution {

public:

    long long maxSum(vector<int>& nums, int m, int k) {

        int n=nums.size();

        unordered\_map<int,int>mp;

        int i=0,j=0;

        long long sum=0,ans=0;

        while(j<n){

            if(j-i<k){

                sum+=nums[j];

                mp[nums[j]]++;

                j++;

            }

            else {

                if(mp.size()>=m){

                    ans=max(ans,sum);

                }

                mp[nums[i]]--;

                if(mp[nums[i]]<=0){

                    mp.erase(nums[i]);

                }

                sum-=nums[i];

                i++;

            }

        }

        while(i<n){

            if(j-i<k){

                break;

            }

            else{

                if(mp.size()>=m){

                    ans=max(ans,sum);

                }

                mp[nums[i]]--;

                if(mp[nums[i]]<=0){

                    mp.erase(nums[i]);

                }

                sum-=nums[i];

                i++;

            }

        }

        return ans;

    }

};

**358: Print the longest increasing subsequence in the order of lowest ind**

**Sol: using bottom-up approach**

class Solution {

  public:

    vector<int> longestIncreasingSubsequence(int n, vector<int>& arr) {

        // Code here

        vector<int>dp(n,1);

        vector<int>hash(n,1);

        for(int i=0;i<n;i++){

            hash[i]=i;

            for(int j=0;j<i;j++){

                if(arr[i]>arr[j] && 1+dp[j]>dp[i]){

                    dp[i]=max(dp[i],1+dp[j]);

                    hash[i]=j;

                }

            }

        }

        int ans=-1,last\_ind=-1;

        for(int i=0;i<n;i++){

            if(dp[i]>ans){

                ans=dp[i];

                last\_ind=i;

            }

        }

        vector<int>temp;

        temp.push\_back(arr[last\_ind]);

        while(hash[last\_ind]!=last\_ind){

            last\_ind=hash[last\_ind];

            temp.push\_back(arr[last\_ind]);

        }

        reverse(temp.begin(),temp.end());

        return temp;

    }

};

**358: Hungry Rabbit Question (dp)**

There is m\*n table with a rabbit in one of the cell - rabbit is initiall at position [startR, startC] . rabbit is Hungry, want to eat something and crying. He is trying to come out of the table, rabbit is allowed to move to one of the four adjacent cells of a table, rabbit can move at most max no of moves(maxMoves).

Given integers m, n, maxMoves, startR, startC, return number of paths to move the rabbit out of the cell. The result might be very Large, return it using modulo 10^9+7

**Example 1**

Input m= 2, n = 2, maxMoves = 2, startR = 0, startC = 0

Output: 6

**Example 2**

Input m= 1, n = 3, maxMoves = 3, startR = 0, startC = 1

Output: 12

**Constraints:**

1 <= m, n <= 50

0 <= maxMove <= 50

0 <= startRow < m

0 <= startColumn < n

**Input**

The input will contain 5 values: m, n, maxMove, startRow, startColumn

**Output**

The output should contain one integer which is the answer mod 10^9 + 7.

**Example**

**input**

1 4 3 0 1

**output**

14

#include <iostream>

#include <vector>

using namespace std;

const int MOD = 1e9 + 7;

int findPaths(int m, int n, int maxMoves, int startR, int startC, vector<vector<vector<int>>>& memo) {

    if (startR < 0 || startR >= m || startC < 0 || startC >= n)

        return 1;  // If the rabbit is outside the grid, count this as a path.

    if (maxMoves <= 0)

        return 0;  // If no moves are left, return 0.

    if (memo[maxMoves][startR][startC] != -1)

        return memo[maxMoves][startR][startC];

    int dirs[4][2] = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};

    long long paths = 0;

    for (int d = 0; d < 4; ++d) {

        int nr = startR + dirs[d][0];

        int nc = startC + dirs[d][1];

        paths = (paths + findPaths(m, n, maxMoves - 1, nr, nc, memo)) % MOD;

    }

    memo[maxMoves][startR][startC] = paths;

    return paths;

}

int main() {

    int m, n, maxMoves, startR, startC;

    cin >> m >> n >> maxMoves >> startR >> startC;

    vector<vector<vector<int>>> memo(maxMoves + 1, vector<vector<int>>(m, vector<int>(n, -1)));

    int result = findPaths(m, n, maxMoves, startR, startC, memo);

    cout << result << endl;

    return 0;

}

**359: Split Linked List in Parts**

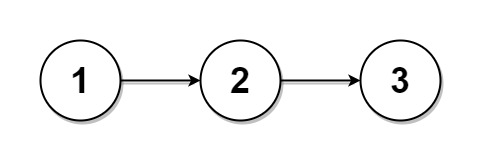
Given the head of a singly linked list and an integer k, split the linked list into k consecutive linked list parts.

The length of each part should be as equal as possible: no two parts should have a size differing by more than one. This may lead to some parts being null.

The parts should be in the order of occurrence in the input list, and parts occurring earlier should always have a size greater than or equal to parts occurring later.

Return *an array of the*k*parts*.

**Example 1:**



**Input:** head = [1,2,3], k = 5

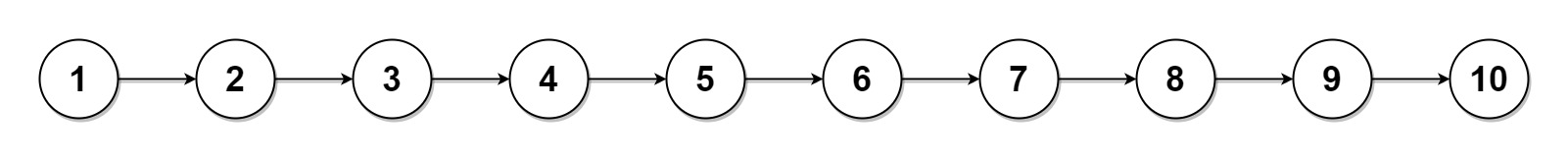
**Output:** [[1],[2],[3],[],[]]

**Explanation:**

The first element output[0] has output[0].val = 1, output[0].next = null.

The last element output[4] is null, but its string representation as a ListNode is [].

**Example 2:**



**Input:** head = [1,2,3,4,5,6,7,8,9,10], k = 3

**Output:** [[1,2,3,4],[5,6,7],[8,9,10]]

**Explanation:**

The input has been split into consecutive parts with size difference at most 1, and earlier parts are a larger size than the later parts.

**Constraints:**

* The number of nodes in the list is in the range [0, 1000].
* 0 <= Node.val <= 1000
* 1 <= k <= 50
* class Solution {
* public:
* vector<ListNode\*> splitListToParts(ListNode\* head, int k) {
* ListNode \*curr = head;
* int len = 0;
* // Calculate the length of the linked list
* while (curr != NULL) {
* curr = curr->next;
* len++;
* }
* // Calculate the number of elements in each part and the number of remaining elements
* int partSize = len / k;
* int extra = len % k;
* curr = head;
* vector<ListNode\*> ans;
* // Iterate through the linked list to split it into parts
* for (int i = 0; i < k; i++) {
* ans.push\_back(curr); // Store the current head of the part
* // Calculate the size of this part (add one extra element if necessary)
* int currentPartSize = partSize + (i < extra ? 1 : 0);
* // Move the current pointer to the end of the current part
* for (int j = 0; j < currentPartSize - 1; j++) {
* curr = curr->next;
* }
* // If we're not at the end of the list, set the next node as the head of the next part
* if (curr != NULL) {
* ListNode\* nextHead = curr->next;
* curr->next = NULL; // Disconnect the current part
* curr = nextHead; // Move to the next part's head
* }
* }
* return ans;
* }
* };\*/

**360 Combination sum |||**

Find all valid combinations of k numbers that sum up to n such that the following conditions are true:

* Only numbers 1 through 9 are used.
* Each number is used **at most once**.

Return *a list of all possible valid combinations*. The list must not contain the same combination twice, and the combinations may be returned in any order.

**Example 1:**

**Input:** k = 3, n = 7

**Output:** [[1,2,4]]

**Explanation:**

1 + 2 + 4 = 7

There are no other valid combinations.

**Example 2:**

**Input:** k = 3, n = 9

**Output:** [[1,2,6],[1,3,5],[2,3,4]]

**Explanation:**

1 + 2 + 6 = 9

1 + 3 + 5 = 9

2 + 3 + 4 = 9

There are no other valid combinations.

**Example 3:**

**Input:** k = 4, n = 1

**Output:** []

**Explanation:** There are no valid combinations.

Using 4 different numbers in the range [1,9], the smallest sum we can get is 1+2+3+4 = 10 and since 10 > 1, there are no valid combination.

**Constraints:**

* 2 <= k <= 9
* 1 <= n <= 60
* class Solution {
* public:
* vector<vector<int>>ans;
* void solve(int k,int target, vector<int>&ds,int idx){
* if(target==0 && k==0){
* ans.push\_back(ds);
* return;
* }
* if(idx>=10 || target<0 || k<0){
* return ;
* }
* // ds.push\_back(idx);
* // solve(k-1,target-idx,ds,idx+1);
* // ds.pop\_back();
* // solve(k,target,ds,idx+1) ;
* //another way of writing backtrack
* for(int i=idx;i<=9;i++){
* ds.push\_back(i);
* solve(k-1,target-i,ds,i+1);
* ds.pop\_back();
* }
* }
* vector<vector<int>> combinationSum3(int k, int n) {
* vector<int>nums={1,2,3,4,5,6,7,8,9};
* vector<int>ds;
* solve(k,n,ds,1);
* return ans;
* }
* };\*/
* **OR**
* class Solution {
* public:
* vector<vector<int>>ans;
* void solve(int k,int target, vector<int>&ds,int idx){
* if(target==0 && k==0){
* ans.push\_back(ds);
* return;
* }
* if(idx>=10 ||  k<0){
* return ;
* }
* if(target-idx>=0){
* ds.push\_back(idx);
* solve(k-1,target-idx,ds,idx+1);
* ds.pop\_back();
* }
* solve(k,target,ds,idx+1) ;
* }
* vector<vector<int>> combinationSum3(int k, int n) {
* vector<int>nums={1,2,3,4,5,6,7,8,9};
* vector<int>ds;
* solve(k,n,ds,1);
* return ans;
* }
* };

**361: Combination sum iv**

Given an array of **distinct** integers nums and a target integer target, return *the number of possible combinations that add up to* target.

The test cases are generated so that the answer can fit in a **32-bit** integer.

**Example 1:**

**Input:** nums = [1,2,3], target = 4

**Output:** 7

**Explanation:**

The possible combination ways are:

(1, 1, 1, 1)

(1, 1, 2)

(1, 2, 1)

(1, 3)

(2, 1, 1)

(2, 2)

(3, 1)

Note that different sequences are counted as different combinations.

**Example 2:**

**Input:** nums = [9], target = 3

**Output:** 0

**Constraints:**

* 1 <= nums.length <= 200
* 1 <= nums[i] <= 1000
* All the elements of nums are **unique**.
* 1 <= target <= 1000
* //solution using khandani backtracking take / notake
* class Solution {
* public:
* int dp[201][1001];
* int solve(vector<int>&nums,int target,int idx){
* if(target==0){
* return 1;
* }
* if(idx>=nums.size()||target<0){
* return 0;
* }
* if(dp[idx][target]!=-1){
* return dp[idx][target];
* }
* //take
* int take=solve(nums,target-nums[idx],0);
* //notake
* int notake=solve(nums,target,idx+1);
* return dp[idx][target]=notake+take;
* }
* int combinationSum4(vector<int>& nums, int target) {
* memset(dp,-1,sizeof(dp));
* return solve(nums,target,0);
* }
* };\*/
* //solution 2 using khandani backtracking take / notake
* //another way of writing backtracking using for loop
* class Solution {
* public:
* int dp[201][1001];
* int solve(vector<int>&nums,int target,int idx){
* if(target==0){
* return 1;
* }
* if(idx>=nums.size()||target<0){
* return 0;
* }
* if(dp[idx][target]!=-1){
* return dp[idx][target];
* }
* int result=0;
* for(int i=idx;i<nums.size();i++){
* int take=solve(nums,target-nums[i],0);
* result+=take;
* }
* //notake will be taken care of by for loop by incrementing i
* return dp[idx][target]=result;
* }
* int combinationSum4(vector<int>& nums, int target) {
* memset(dp,-1,sizeof(dp));
* return solve(nums,target,0);
* }
* };

**362: No of increasing subsequences**

Input : arr[] = {1, 2, 3, 4}

Output: 15

There are 15 increasing subsequences

{1}, {2}, {3}, {4}, {1,2}, {1,3}, {1,4},

{2,3}, {2,4}, {3,4}, {1,2,3}, {1,2,4},

{1,3,4}, {2,3,4}, {1,2,3,4}

/\* Dynamic Programming C/C++ program to count increasing

   subsequences \*/

#include<stdio.h>

#include<stdlib.h>

int countSub( int arr[], int n )

{

    int cSub[n];

    /\* Initialize cSub values for all indexes \*/

    for (int i = 0; i < n; i++ )

        cSub[i] = 1;

    for (int i = 1; i < n; i++ )

        for (int j = 0; j < i; j++ )

            if ( arr[i] > arr[j])

                cSub[i] += cSub[j];

    int result = 0;

    for (int i = 0; i < n; i++ )

          result += cSub[i];

    return result;

}

/\* Driver program to test above function \*/

int main()

{

    int arr[] =  {3, 2, 4, 5, 4};

    int n = sizeof(arr)/sizeof(arr[0]);

    printf("%d\n", countSub( arr, n ) );

    return 0;

}

**363: count no of longest increasing subsequence**

Given an integer array nums, return *the number of longest increasing subsequences.*

**Notice** that the sequence has to be **strictly** increasing.

**Example 1:**

**Input:** nums = [1,3,5,4,7]

**Output:** 2

**Explanation:** The two longest increasing subsequences are [1, 3, 4, 7] and [1, 3, 5, 7].

**Example 2:**

**Input:** nums = [2,2,2,2,2]

**Output:** 5

**Explanation:** The length of the longest increasing subsequence is 1, and there are 5 increasing subsequences of length 1, so output 5.

**Constraints:**

* 1 <= nums.length <= 2000
* -106 <= nums[i] <= 106

class Solution {

public:

    int findNumberOfLIS(vector<int>& nums) {

        int n=nums.size();

        vector<int>t(n,1); //lis ending at i

        vector<int>count(n,1); //count of lis ending at i;

        for(int i=0;i<n;i++){

            for(int j=0;j<i;j++){

                if(nums[j]<nums[i]){

                    if(t[j]+1==t[i]){  //we are incrementing count here because if we get length of lis already at i , it means we got another lis of same length at i

                        count[i]+=count[j];

                    }

                    else if(t[j]+1>t[i]){

                        t[i]=t[j]+1;

                        count[i]=count[j];

                    }

                }

            }

        }

        int cnt=0;

        int max\_value=\*max\_element(t.begin(),t.end());

        for(int i=0;i<n;i++){

            if(t[i]==max\_value){

                cnt+=count[i];

            }

        }

        return cnt;

    }

};

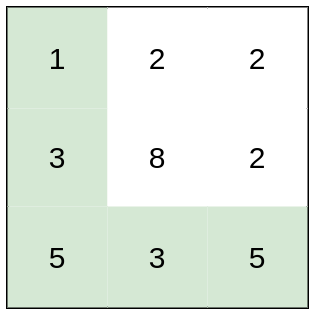
**364: Path With Minimum Effort**

You are a hiker preparing for an upcoming hike. You are given heights, a 2D array of size rows x columns, where heights[row][col] represents the height of cell (row, col). You are situated in the top-left cell, (0, 0), and you hope to travel to the bottom-right cell, (rows-1, columns-1) (i.e., **0-indexed**). You can move **up**, **down**, **left**, or **right**, and you wish to find a route that requires the minimum **effort**.

A route's **effort** is the **maximum absolute difference**in heights between two consecutive cells of the route.

Return *the minimum****effort****required to travel from the top-left cell to the bottom-right cell.*

**Example 1:**



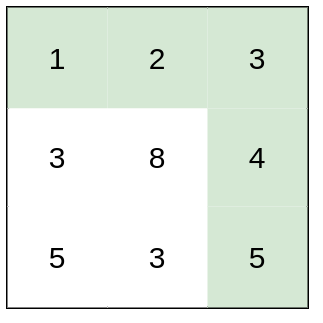
**Input:** heights = [[1,2,2],[3,8,2],[5,3,5]]

**Output:** 2

**Explanation:** The route of [1,3,5,3,5] has a maximum absolute difference of 2 in consecutive cells.

This is better than the route of [1,2,2,2,5], where the maximum absolute difference is 3.

**Example 2:**

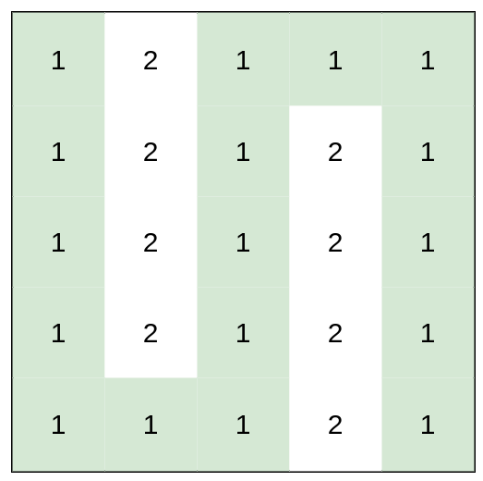


**Input:** heights = [[1,2,3],[3,8,4],[5,3,5]]

**Output:** 1

**Explanation:** The route of [1,2,3,4,5] has a maximum absolute difference of 1 in consecutive cells, which is better than route [1,3,5,3,5].

**Example 3:**



**Input:** heights = [[1,2,1,1,1],[1,2,1,2,1],[1,2,1,2,1],[1,2,1,2,1],[1,1,1,2,1]]

**Output:** 0

**Explanation:** This route does not require any effort.

**Constraints:**

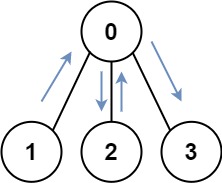
* rows == heights.length
* columns == heights[i].length
* 1 <= rows, columns <= 100
* 1 <= heights[i][j] <= 106
* //using dijkshra
* class Solution {
* public:
* int minimumEffortPath(vector<vector<int>>& heights) {
* int n=heights.size();
* int m=heights[0].size();
* priority\_queue< pair<int,pair<int,int>> , vector<pair<int,pair<int,int>>>, greater<pair<int,pair<int,int>>> >pq;
* vector<vector<int>>dis(n,vector<int>(m,1e9));
* dis[0][0]=0;
* pq.push({0,{0,0}});  // {dis,{row,col}}
* int drow[]={-1,0,1,0};
* int dcol[]={0,1,0,-1};
* while(!pq.empty()){
* auto it=pq.top();
* pq.pop();
* int diff=it.first;
* int r=it.second.first;
* int c=it.second.second;
* if(r==n-1 && c==m-1){
* return diff;
* }
* for(int i=0;i<4;i++){
* int nr=r+drow[i];
* int nc=c+dcol[i];
* // is safe
* if(nr>=0 && nr<n && nc>=0 && nc<m){
* int newEffort=max(abs(heights[nr][nc]-heights[r][c]),diff);
* if(newEffort<dis[nr][nc]){
* dis[nr][nc]=newEffort;
* pq.push({newEffort,{nr,nc}});
* }
* }
* }
* }
* return 0;
* }
* };

**365: Shortest Path Visiting All nodes**

You have an undirected, connected graph of n nodes labeled from 0 to n - 1. You are given an array graph where graph[i] is a list of all the nodes connected with node i by an edge.

Return *the length of the shortest path that visits every node*. You may start and stop at any node, you may revisit nodes multiple times, and you may reuse edges.

**Example 1:**

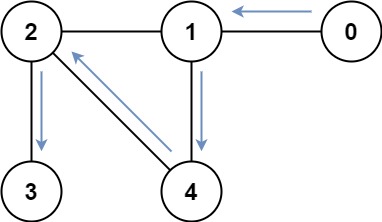


**Input:** graph = [[1,2,3],[0],[0],[0]]

**Output:** 4

**Explanation:** One possible path is [1,0,2,0,3]

**Example 2:**



**Input:** graph = [[1],[0,2,4],[1,3,4],[2],[1,2]]

**Output:** 4

**Explanation:** One possible path is [0,1,4,2,3]

**Constraints:**

* n == graph.length
* 1 <= n <= 12
* 0 <= graph[i].length < n
* graph[i] does not contain i.
* If graph[a] contains b, then graph[b] contains a.
* The input graph is always connected.

class Solution {

public:

    int shortestPathLength(vector<vector<int>>& graph) {

      int n=graph.size();

      if(n==0 || n==1){

          return 0;

      }

      queue<pair<int,int>>q; //node, maskvalue

      set<pair<int,int>>vis; //node, maskvalue

      for(int i=0;i<n;i++){

        int mask=1<<i;

        q.push({i,mask});

        vis.insert({i,mask});

      }

      int allVisStates =(1<<n)-1 ; //or pow(2,n)-1

      int path=0;

      while(!q.empty()){

        int size=q.size();

        path++;

        while(size--){

            auto it=q.front();

            q.pop();

            int node=it.first;

            int mask=it.second;

            for(auto &adj:graph[node]){

                int nextMask = mask | (1<<adj);

                if(nextMask==allVisStates){

                    return path;

                }

                if(vis.find({adj,nextMask})==vis.end()){

                    vis.insert({adj,nextMask});

                    q.push({adj,nextMask});

                }

            }

        }

      }

      return path;

    }

};

**366: Median of two sorted arrays**

[**https://www.codingninjas.com/studio/problems/median-of-two-sorted-arrays\_985294?leftPanelTab=0**](https://www.codingninjas.com/studio/problems/median-of-two-sorted-arrays_985294?leftPanelTab=0)

**atleast 3 sol , brute->better->optimal**

**367: Longest String Chain**

You are given an array of words where each word consists of lowercase English letters.

wordA is a **predecessor** of wordB if and only if we can insert **exactly one** letter anywhere in wordA **without changing the order of the other characters** to make it equal to wordB.

* For example, "abc" is a **predecessor** of "abac", while "cba" is not a **predecessor** of "bcad".

A **word chain**is a sequence of words [word1, word2, ..., wordk] with k >= 1, where word1 is a **predecessor** of word2, word2 is a **predecessor** of word3, and so on. A single word is trivially a **word chain** with k == 1.

Return *the****length****of the****longest possible word chain****with words chosen from the given list of*words.

**Example 1:**

**Input:** words = ["a","b","ba","bca","bda","bdca"]

**Output:** 4

**Explanation**: One of the longest word chains is ["a","ba","bda","bdca"].

**Example 2:**

**Input:** words = ["xbc","pcxbcf","xb","cxbc","pcxbc"]

**Output:** 5

**Explanation:** All the words can be put in a word chain ["xb", "xbc", "cxbc", "pcxbc", "pcxbcf"].

**Example 3:**

**Input:** words = ["abcd","dbqca"]

**Output:** 1

**Explanation:** The trivial word chain ["abcd"] is one of the longest word chains.

["abcd","dbqca"] is not a valid word chain because the ordering of the letters is changed.

**Constraints:**

* 1 <= words.length <= 1000
* 1 <= words[i].length <= 16
* words[i] only consists of lowercase English letters.

//top down dp

// time =o(N^2\*M) , m for checking checkpossible

class Solution {

private:

    bool checkPossible(string &s1, string &s2) {

        if (s1.size() != s2.size() + 1) {

            return false;

        }

        int first = 0, second = 0;

        while (first < s1.size()) {

            if (s1[first] == s2[second]) {

                first++, second++;

            } else {

                first++;

            }

        }

        return first == s1.size() && second == s2.size();

    }

    int longestStrChainUtil(vector<string>& words, vector<int>& memo, int idx) {

        if (memo[idx] != -1) {

            return memo[idx];

        }

        int maxChainLength = 1;

        for (int prev = 0; prev < idx; prev++) {

            if (checkPossible(words[idx], words[prev])) {

                int chainLength = 1 + longestStrChainUtil(words, memo, prev);

                maxChainLength = max(maxChainLength, chainLength);

            }

        }

        memo[idx] = maxChainLength;

        return maxChainLength;

    }

public:

    int longestStrChain(vector<string>& words) {

        int n = words.size();

        vector<int> dp(n, 1);

        int maxi = 1;

        sort(words.begin(), words.end(), comp);

        vector<int> memo(n, -1); // Initialize memoization table

        for (int i = 0; i < n; i++) {

            int chainLength = longestStrChainUtil(words, memo, i);

            maxi = max(maxi, chainLength);

        }

        return maxi;

    }

    static bool comp(string &s1, string &s2) {

        return s1.size() < s2.size();

    }

};\*/

//another way of writing top down code using concept of lis

class Solution {

public:

    int n;

    int t[1001][1001];

    bool predecessor(string& prev, string& curr) {

        int M = prev.length();

        int N = curr.length();

        if(M >= N || N-M != 1)

            return false;

        int i = 0, j = 0;

        while(i < M && j < N) {

            if(prev[i] == curr[j]) {

                i++;

            }

            j++;

        }

        return i==M;

    }

    int lis(vector<string>& words, int prev\_idx, int curr\_idx) {

       if(curr\_idx == n)

           return 0;

        if(prev\_idx != -1 && t[prev\_idx][curr\_idx] != -1)

            return t[prev\_idx][curr\_idx];

        int taken = 0;

        if(prev\_idx == -1 || predecessor(words[prev\_idx], words[curr\_idx]))

            taken = 1 + lis(words, curr\_idx, curr\_idx+1);

        int not\_taken = lis(words, prev\_idx, curr\_idx+1);

        if(prev\_idx != -1)

            t[prev\_idx][curr\_idx] =  max(taken, not\_taken);

        return max(taken, not\_taken);

    }

    static bool myFunction(string& s1, string& s2) {

        return s1.length() < s2.length();

    }

    int longestStrChain(vector<string>& words) {

        memset(t, -1, sizeof(t));

        n = words.size();

        sort(begin(words), end(words), myFunction); //You can select pairs in any order.

        return lis(words, -1, 0);

    }

};

//bottom up dp

class Solution {

    bool checkPossible(string &s1, string &s2){

        if(s1.size()!=s2.size()+1){

            return false;

        }

        int first=0,second=0;

        while(first<s1.size()){

            if(s1[first]==s2[second]){

                first++,second++;

            }

            else{

                first++;

             }

        }

        if(first==s1.size() && second==s2.size()){

            return true;

        }

        return false;

    }

    static bool comp(string &s1,string &s2)

    {

        return s1.size()<s2.size();

    }

    public:

    int longestStrChain(vector<string>& words) {

        sort(words.begin(),words.end(),comp);

        int n=words.size();

        vector<int>dp(n,1);

        int maxi=1;

        for(int i=0;i<n;i++){

            for(int prev=0;prev<i;prev++){

                if(checkPossible(words[i],words[prev]) && 1+dp[prev]>dp[i]){

                    dp[i]=1+dp[prev];

                }

            }

            maxi=max(maxi,dp[i]);

        }

        return maxi;

    }

};\*/

**368: Dota2 Senate (queue based question)**

In the world of Dota2, there are two parties: the Radiant and the Dire.

The Dota2 senate consists of senators coming from two parties. Now the Senate wants to decide on a change in the Dota2 game. The voting for this change is a round-based procedure. In each round, each senator can exercise **one** of the two rights:

* **Ban one senator's right:** A senator can make another senator lose all his rights in this and all the following rounds.
* **Announce the victory:** If this senator found the senators who still have rights to vote are all from the same party, he can announce the victory and decide on the change in the game.

Given a string senate representing each senator's party belonging. The character 'R' and 'D' represent the Radiant party and the Dire party. Then if there are n senators, the size of the given string will be n.

The round-based procedure starts from the first senator to the last senator in the given order. This procedure will last until the end of voting. All the senators who have lost their rights will be skipped during the procedure.

Suppose every senator is smart enough and will play the best strategy for his own party. Predict which party will finally announce the victory and change the Dota2 game. The output should be "Radiant" or "Dire".

**Example 1:**

**Input:** senate = "RD"

**Output:** "Radiant"

**Explanation:**

The first senator comes from Radiant and he can just ban the next senator's right in round 1.

And the second senator can't exercise any rights anymore since his right has been banned.

And in round 2, the first senator can just announce the victory since he is the only guy in the senate who can vote.

**Example 2:**

**Input:** senate = "RDD"

**Output:** "Dire"

**Explanation:**

The first senator comes from Radiant and he can just ban the next senator's right in round 1.

And the second senator can't exercise any rights anymore since his right has been banned.

And the third senator comes from Dire and he can ban the first senator's right in round 1.

And in round 2, the third senator can just announce the victory since he is the only guy in the senate who can vote.

**Constraints:**

* n == senate.length
* 1 <= n <= 104
* senate[i] is either 'R' or 'D'.

[**https://leetcode.com/problems/dota2-senate/description/?envType=study-plan-v2&envId=leetcode-75**](https://leetcode.com/problems/dota2-senate/description/?envType=study-plan-v2&envId=leetcode-75)

**369: Champagne Tower**

[**https://leetcode.com/problems/champagne-tower/?envType=daily-question&envId=2023-09-24**](https://leetcode.com/problems/champagne-tower/?envType=daily-question&envId=2023-09-24)

You are given a **2D** integer array coordinates and an integer k, where coordinates[i] = [xi, yi] are the coordinates of the ith point in a 2D plane.

We define the **distance** between two points (x1, y1) and (x2, y2) as (x1 XOR x2) + (y1 XOR y2) where XOR is the bitwise XOR operation.

Return *the number of pairs*(i, j)*such that*i < j*and the distance between points*i*and*j*is equal to*k.

**Example 1:**

**Input:** coordinates = [[1,2],[4,2],[1,3],[5,2]], k = 5

**Output:** 2

**Explanation:** We can choose the following pairs:

- (0,1): Because we have (1 XOR 4) + (2 XOR 2) = 5.

- (2,3): Because we have (1 XOR 5) + (3 XOR 2) = 5.

**Example 2:**

**Input:** coordinates = [[1,3],[1,3],[1,3],[1,3],[1,3]], k = 0

**Output:** 10

**Explanation:** Any two chosen pairs will have a distance of 0. There are 10 ways to choose two pairs.

**Constraints:**

* 2 <= coordinates.length <= 50000
* 0 <= xi, yi <= 106
* 0 <= k <= 100

**370: binary tree questions part of leetcode 75  
i>path sum iii**

**ii>longest zig zag path**

**371: remove duplicate letters**

[**https://leetcode.com/problems/remove-duplicate-letters/?envType=daily-question&envId=2023-09-26**](https://leetcode.com/problems/remove-duplicate-letters/?envType=daily-question&envId=2023-09-26)

**372: Reorder route to make all path lead to city zero**

[**https://leetcode.com/problems/reorder-routes-to-make-all-paths-lead-to-the-city-zero/description/?envType=study-plan-v2&envId=leetcode-75**](https://leetcode.com/problems/reorder-routes-to-make-all-paths-lead-to-the-city-zero/description/?envType=study-plan-v2&envId=leetcode-75)

**bfs solution is more intuitive than dfs**

**373: Count the no fo possible triangles**

**Two solutions**

* 1. **Using three loops**
  2. **Using sorting and two pointers**

[**https://www.geeksforgeeks.org/find-number-of-triangles-possible/**](https://www.geeksforgeeks.org/find-number-of-triangles-possible/)

**374: Lexicographically smallest k length subsequence**

**Two solutions:**

* + 1. **Using recursion and backtracking generating all subsequence of length k and sort them and return 0th string stored in vector <string>ans , time =2^k**
    2. **Using stack(monotonic)**

[**https://www.geeksforgeeks.org/lexicographically-smallest-k-length-subsequence-from-a-given-string/**](https://www.geeksforgeeks.org/lexicographically-smallest-k-length-subsequence-from-a-given-string/)

[**https://www.onlinegdb.com/edit/wI6l3oY\_h**](https://www.onlinegdb.com/edit/wI6l3oY_h) **//done both the approach**

**375: Beautiful subsequence: Given a string s of length n containing only letters 'a' and 'b'. Find the lexicographically smallest subsequence of s of length k having atleast x number of 'b'** .

[**https://www.onlinegdb.com/online\_c++\_compiler**](https://www.onlinegdb.com/online_c++_compiler)

**solved using deque , same concept of previous ques (374)**

**376: Maximum no of k divisible components**

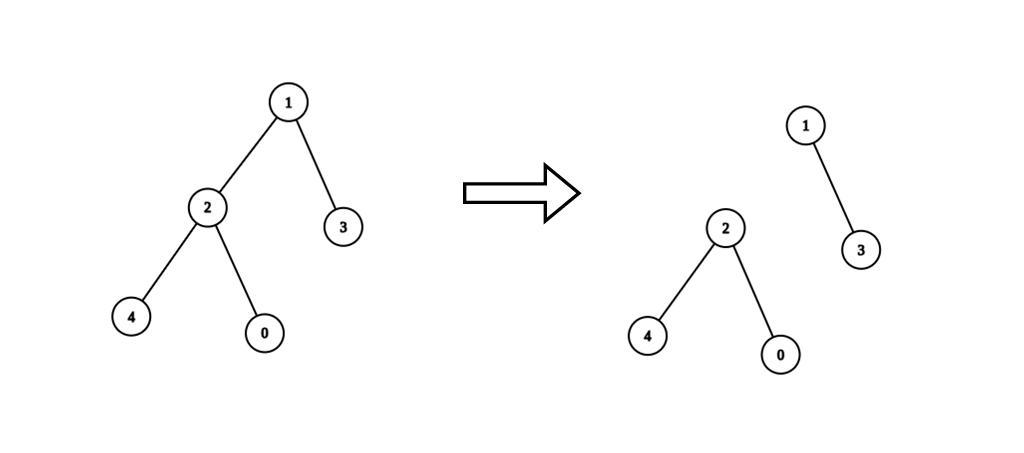
There is an undirected tree with n nodes labeled from 0 to n - 1. You are given the integer n and a 2D integer array edges of length n - 1, where edges[i] = [ai, bi] indicates that there is an edge between nodes ai and bi in the tree.

You are also given a **0-indexed** integer array values of length n, where values[i] is the **value** associated with the ith node, and an integer k.

A **valid split** of the tree is obtained by removing any set of edges, possibly empty, from the tree such that the resulting components all have values that are divisible by k, where the **value of a connected component** is the sum of the values of its nodes.

Return *the****maximum number of components****in any valid split*.

**Example 1:**



**Input:** n = 5, edges = [[0,2],[1,2],[1,3],[2,4]], values = [1,8,1,4,4], k = 6

**Output:** 2

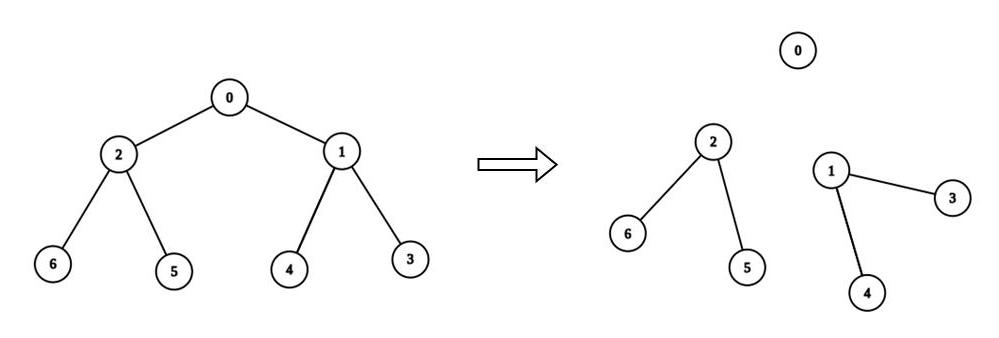
**Explanation:** We remove the edge connecting node 1 with 2. The resulting split is valid because:

- The value of the component containing nodes 1 and 3 is values[1] + values[3] = 12.

- The value of the component containing nodes 0, 2, and 4 is values[0] + values[2] + values[4] = 6.

It can be shown that no other valid split has more than 2 connected components.

**Example 2:**



**Input:** n = 7, edges = [[0,1],[0,2],[1,3],[1,4],[2,5],[2,6]], values = [3,0,6,1,5,2,1], k = 3

**Output:** 3

**Explanation:** We remove the edge connecting node 0 with 2, and the edge connecting node 0 with 1. The resulting split is valid because:

- The value of the component containing node 0 is values[0] = 3.

- The value of the component containing nodes 2, 5, and 6 is values[2] + values[5] + values[6] = 9.

- The value of the component containing nodes 1, 3, and 4 is values[1] + values[3] + values[4] = 6.

It can be shown that no other valid split has more than 3 connected components.

**Constraints:**

* 1 <= n <= 3 \* 104
* edges.length == n - 1
* edges[i].length == 2
* 0 <= ai, bi < n
* values.length == n
* 0 <= values[i] <= 109
* 1 <= k <= 109
* Sum of values is divisible by k.
* The input is generated such that edges represents a valid tree.

class Solution {

public:

     // dp array to which is storing the sum of descents including node value

    vector<int>dp;

    int dfs(vector<int>adj[],vector<int>&vis,int node,int &cnt,int k){

        vis[node]=1;

        for(auto &it:adj[node]){

            if(!vis[it]){

                dp[node]+=dfs(adj,vis,it,cnt,k);

            }

        }

        //this step is executed when no where to go in dfs call

        // going from bottom then coming up

        // if sum till current node is divible by k count in ans and return 0;

        if(dp[node]%k==0){

            cnt++;

            return 0;

        }

        return dp[node];

    }

    int maxKDivisibleComponents(int n, vector<vector<int>>& edges, vector<int>& values, int k) {

        dp.resize(n,0);

        for(int i=0;i<n;i++){

            dp[i]=values[i];

        }

        vector<int>adj[n];

        for(auto &it:edges){

            adj[it[0]].push\_back(it[1]);

            adj[it[1]].push\_back(it[0]);

        }

        vector<int>vis(n,0);

        int cnt=0;

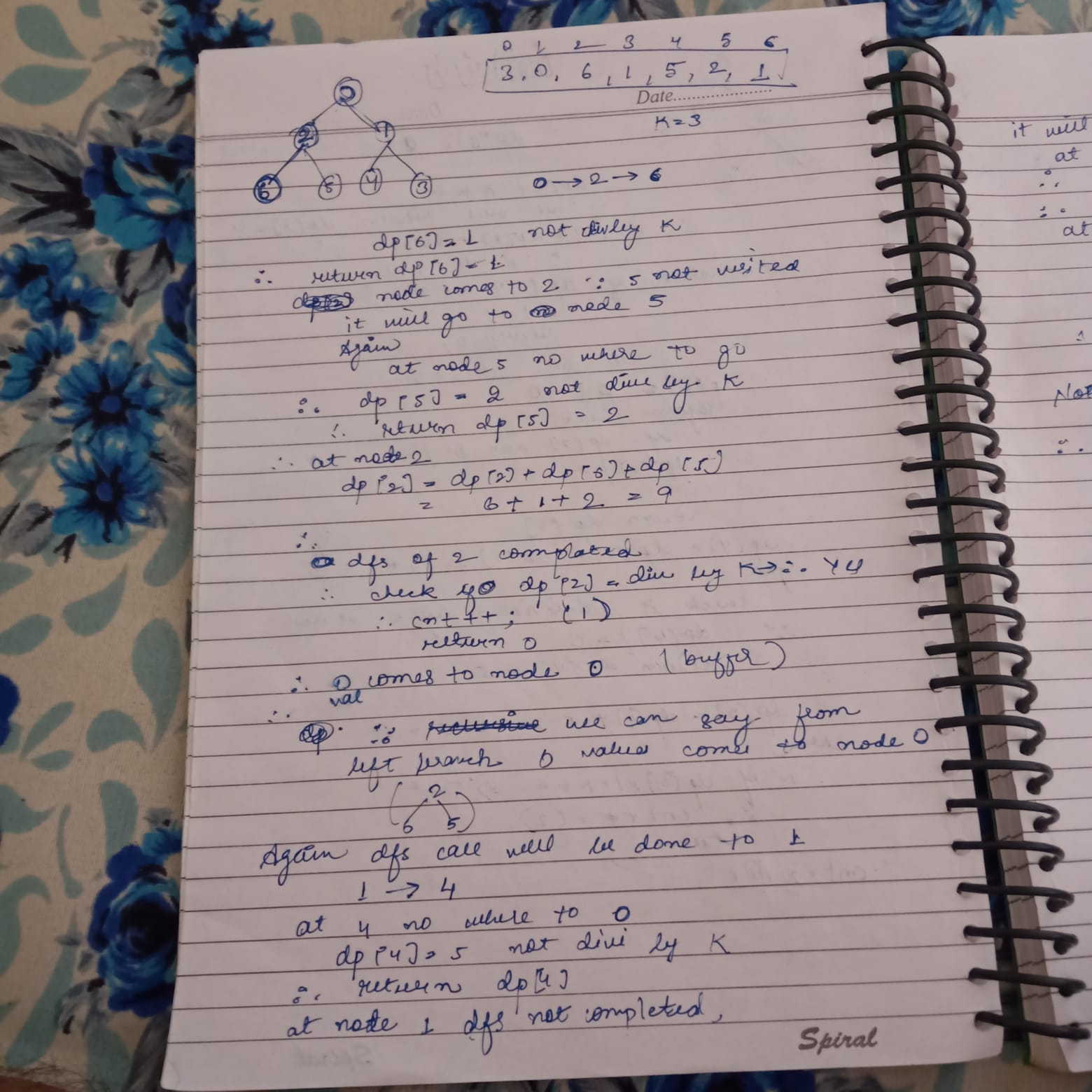
        dfs(adj,vis,0,cnt,k);

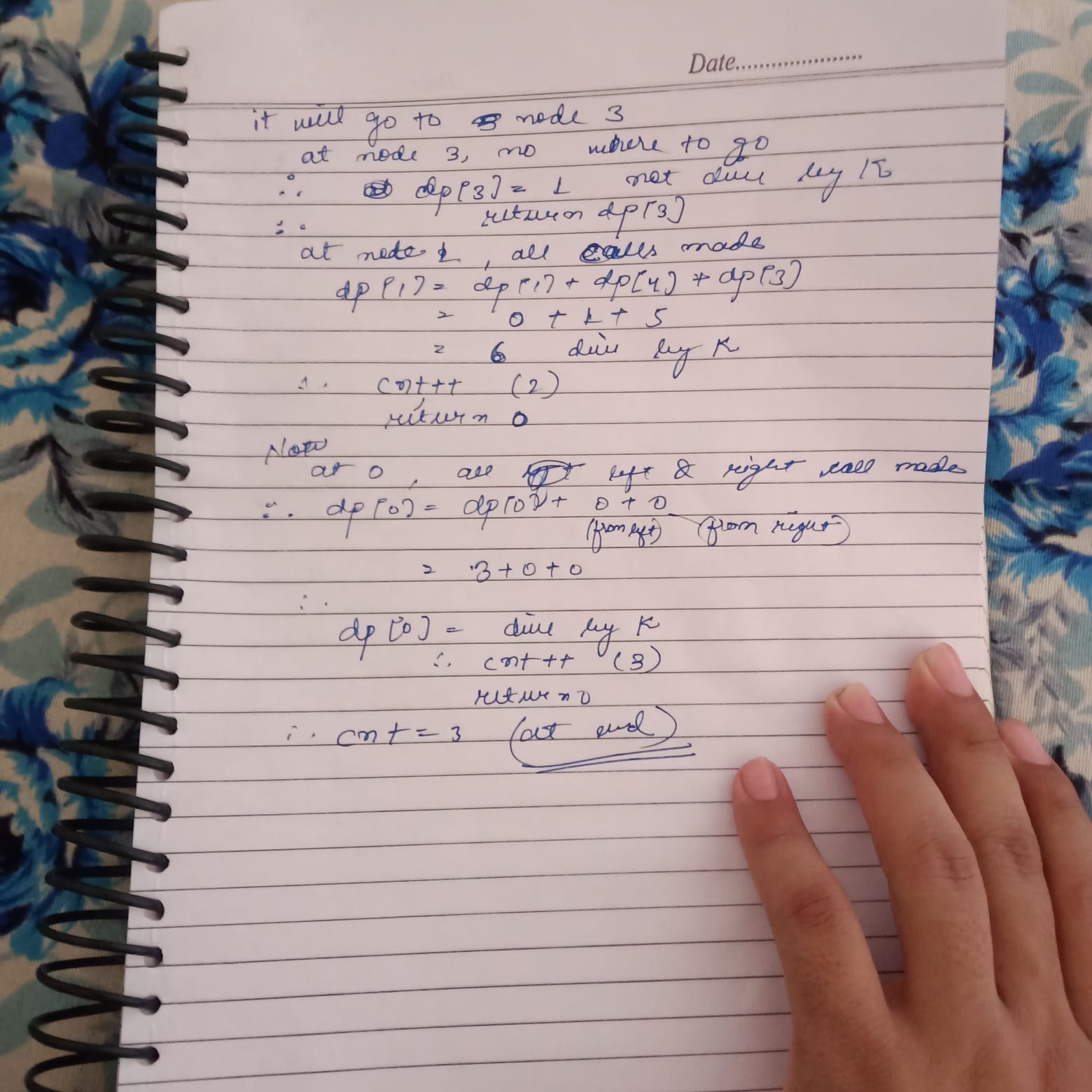
        return cnt;

    }

};

**Explanation:**





**377: Successful pairs of spells and potions (binary search)**

You are given two positive integer arrays spells and potions, of length n and m respectively, where spells[i] represents the strength of the ith spell and potions[j] represents the strength of the jth potion.

You are also given an integer success. A spell and potion pair is considered **successful** if the **product** of their strengths is **at least** success.

Return *an integer array*pairs*of length*n*where*pairs[i]*is the number of****potions****that will form a successful pair with the*ith*spell.*

**Example 1:**

**Input:** spells = [5,1,3], potions = [1,2,3,4,5], success = 7

**Output:** [4,0,3]

**Explanation:**

- 0th spell: 5 \* [1,2,3,4,5] = [5,**10**,**15**,**20**,**25**]. 4 pairs are successful.

- 1st spell: 1 \* [1,2,3,4,5] = [1,2,3,4,5]. 0 pairs are successful.

- 2nd spell: 3 \* [1,2,3,4,5] = [3,6,**9**,**12**,**15**]. 3 pairs are successful.

Thus, [4,0,3] is returned.

**Example 2:**

**Input:** spells = [3,1,2], potions = [8,5,8], success = 16

**Output:** [2,0,2]

**Explanation:**

- 0th spell: 3 \* [8,5,8] = [**24**,15,**24**]. 2 pairs are successful.

- 1st spell: 1 \* [8,5,8] = [8,5,8]. 0 pairs are successful.

- 2nd spell: 2 \* [8,5,8] = [**16**,10,**16**]. 2 pairs are successful.

Thus, [2,0,2] is returned.

**Constraints:**

* n == spells.length
* m == potions.length
* 1 <= n, m <= 105
* 1 <= spells[i], potions[i] <= 105
* 1 <= success <= 1010

class Solution {

public:

    int solve(int low,int high,int target,vector<int>&potions){

        int result=0;

        while(low<=high){

            int mid=(low+high)/2;

            if(potions[mid]>=target){

                result=mid;

                high=mid-1;

            }

            else{

                low=mid+1;

            }

        }

        return result;

    }

    vector<int> successfulPairs(vector<int>& spells, vector<int>& potions, long long success) {

        sort(potions.begin(),potions.end());

        int n=potions.size();

        int maxpotion=potions[n-1];

        vector<int>ans;

        for(int i=0;i<spells.size();i++){

            int spell=spells[i];

            long long minpotion = ceil((1.0\*success)/spell);

            if(minpotion>maxpotion){

                ans.push\_back(0);

                continue;

            }

            //basically we are finding the index from where potions value is >= minpotion

            int index =solve(0,n-1,minpotion,potions);

            //if from 2nd index we got this no of ele >=succes is n-2=3 if n=5

            ans.push\_back(n-index);

        }

        return ans;

    }

};

class Solution {

public:

    vector<int> successfulPairs(vector<int>& spells, vector<int>& potions, long long success) {

        sort(potions.begin(),potions.end());

        int n=potions.size();

        int maxpotion=potions[n-1];

        vector<int>ans;

        for(int i=0;i<spells.size();i++){

            int spell=spells[i];

            long long minpotion = ceil((1.0\*success)/spell);

            if(minpotion>maxpotion){

                ans.push\_back(0);

                continue;

            }

            //basically we are finding the index from where potions' value is >= minpotion

            int index = lower\_bound(potions.begin(),potions.end(),minpotion)-potions.begin();

            //if from 2nd index we got this no of ele >=succes is n-2=3 if n=5

            ans.push\_back(n-index);

        }

        return ans;

    }

};

**378: Find peak element (2 sol , 2nd with binary search)**

[**https://leetcode.com/problems/find-peak-element/description/?envType=study-plan-v2&envId=leetcode-75**](https://leetcode.com/problems/find-peak-element/description/?envType=study-plan-v2&envId=leetcode-75)

**379: Daily Temperatures**

Given an array of integers temperatures represents the daily temperatures, return *an array* answer *such that* answer[i] *is the number of days you have to wait after the* ith *day to get a warmer temperature*. If there is no future day for which this is possible, keep answer[i] == 0 instead.

**Example 1:**

**Input:** temperatures = [73,74,75,71,69,72,76,73]

**Output:** [1,1,4,2,1,1,0,0]

**Example 2:**

**Input:** temperatures = [30,40,50,60]

**Output:** [1,1,1,0]

**Example 3:**

**Input:** temperatures = [30,60,90]

**Output:** [1,1,0]

**Constraints:**

* 1 <= temperatures.length <= 105
* 30 <= temperatures[i] <= 100
* //coded by me and preferred by intuition wise
* class Solution {
* public:
* vector<int> dailyTemperatures(vector<int>& temp) {
* stack<int>st;
* int n=temp.size();
* vector<int>ans(n,0);
* st.push(n-1);
* ans[n-1]=0;
* for(int i=n-2;i>=0;i--){
* if(!st.empty() && temp[i]<temp[st.top()]){
* st.push(i);
* ans[i]=1;
* }
* else{
* while(!st.empty() && temp[i]>=temp[st.top()]){
* st.pop();
* }
* if(!st.empty()){
* int d=st.top();
* ans[i]=d-i;
* }
* st.push(i);
* }
* }
* return ans;
* }
* };

**380: Max dot product of Two subsequence**

Given two arrays nums1 and nums2.

Return the maximum dot product between **non-empty** subsequences of nums1 and nums2 with the same length.

A subsequence of a array is a new array which is formed from the original array by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (ie, [2,3,5] is a subsequence of [1,2,3,4,5] while [1,5,3] is not).

**Example 1:**

**Input:** nums1 = [2,1,-2,5], nums2 = [3,0,-6]

**Output:** 18

**Explanation:** Take subsequence [2,-2] from nums1 and subsequence [3,-6] from nums2.

Their dot product is (2\*3 + (-2)\*(-6)) = 18.

**Example 2:**

**Input:** nums1 = [3,-2], nums2 = [2,-6,7]

**Output:** 21

**Explanation:** Take subsequence [3] from nums1 and subsequence [7] from nums2.

Their dot product is (3\*7) = 21.

**Example 3:**

**Input:** nums1 = [-1,-1], nums2 = [1,1]

**Output:** -1

**Explanation:** Take subsequence [-1] from nums1 and subsequence [1] from nums2.

Their dot product is -1.

**Constraints:**

* 1 <= nums1.length, nums2.length <= 500
* -1000 <= nums1[i], nums2[i] <= 1000
* class Solution {
* public:
* int dp[501][501];
* int solve(vector<int>& nums1, vector<int>& nums2,int i,int j){
* if(i==nums1.size() || j==nums2.size()){
* return -1e9;
* }
* if(dp[i][j]!=-1){
* return dp[i][j];
* }
* //four conditions to take or not;
* int val=nums1[i]\*nums2[j]; // take i and j only
* int take\_i\_j = nums1[i]\*nums2[j] +solve(nums1,nums2,i+1,j+1);//we decided to take i and j and want to add more
* int take\_i=solve(nums1,nums2,i,j+1); //take i and explore for next j or skip curr j
* int take\_j=solve(nums1,nums2,i+1,j); // skip curr i and take j
* return dp[i][j]=max({val,take\_i\_j,take\_i,take\_j});
* }
* int maxDotProduct(vector<int>& nums1, vector<int>& nums2) {
* memset(dp,-1,sizeof(dp));
* return solve(nums1,nums2,0,0);
* }
* };

**381: Number of flowers in full bloom**

[**https://leetcode.com/problems/number-of-flowers-in-full-bloom/?envType=daily-question&envId=2023-10-11**](https://leetcode.com/problems/number-of-flowers-in-full-bloom/?envType=daily-question&envId=2023-10-11)

**382: Find the target in mountain Array**

**Prerequisite: Peak element in array;**

[**https://leetcode.com/problems/find-in-mountain-array/?envType=daily-question&envId=2023-10-12**](https://leetcode.com/problems/find-in-mountain-array/?envType=daily-question&envId=2023-10-12)

**383: Find indices with index and value difference || (min and max heap)**

[**https://leetcode.com/problems/find-indices-with-index-and-value-difference-ii/**](https://leetcode.com/problems/find-indices-with-index-and-value-difference-ii/)

**384: Binary Tree paths (no backtrack done)**

[**https://leetcode.com/problems/binary-tree-paths/**](https://leetcode.com/problems/binary-tree-paths/)