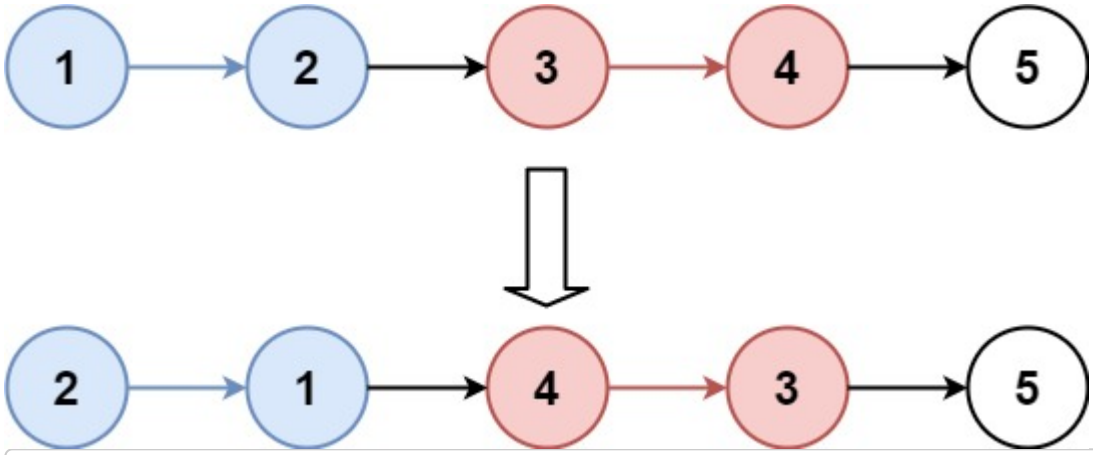
25. Reverse Nodes in k-Group

Given a linked list, reverse the nodes of a linked list *k* at a time and return its modified list.

k is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of *k* then left-out nodes, in the end, should remain as it is.

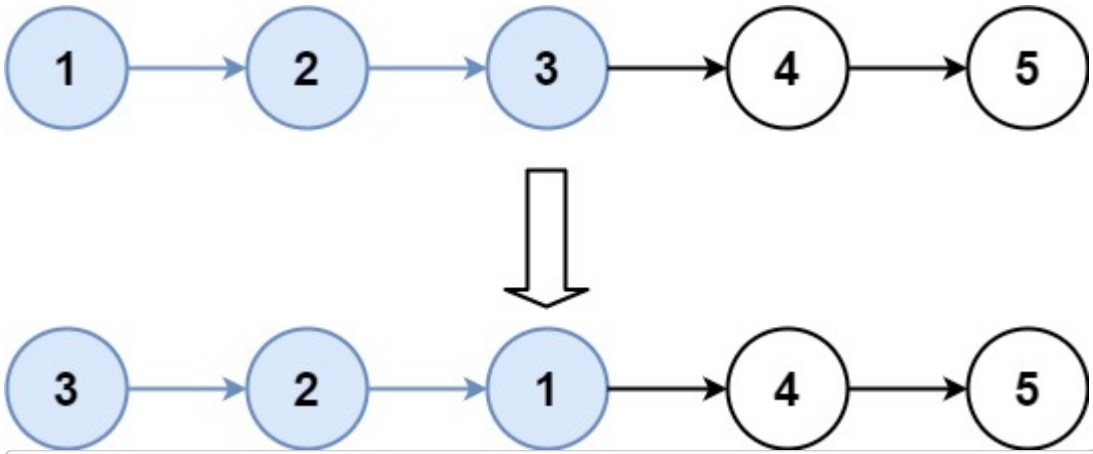
You may not alter the values in the list's nodes, only nodes themselves may be changed.

Example 1:




```
Input: head = [1,2,3,4,5], k = 2
Output: [2,1,4,3,5]
```

Example 2:




```
Input: head = [1,2,3,4,5], k = 3
Output: [3,2,1,4,5]
```

Example 3:

```
Input: head = [1,2,3,4,5], k = 1
Output: [1,2,3,4,5]
```

Example 4:

```
Input: head = [1], k = 1
Output: [1]
```

Constraints:

- The number of nodes in the list is in the range `sz` .
- `1 <= sz <= 5000`
- `0 <= Node.val <= 1000`
- `1 <= k <= sz`

Follow-up: Can you solve the problem in O(1) extra memory space?

---

remember this question recursive

```
ListNode* foo(ListNode* curr, int k)
    {
        if(curr==NULL)
            return NULL;

        ListNode* left= curr;
        int ct=1;
        while(curr->next && ct<k){
            curr=curr->next;
            ct++;
        }

        if(ct<k)
            return left;

        ListNode* right=curr;
        ListNode* res= foo(curr->next,k);
        //reverse the list between left and right
        ListNode* prev= NULL;
        ListNode* move= left;
        while(prev!=right){
            ListNode* temp= move->next;
            move->next=prev;
            prev=move;
            move=temp;
        }

        left->next= res;
        return right;
    }
    ListNode* reverseKGroup(ListNode* head, int k) {

        if(k==1)
            return head;

        return foo(head,k);
    }
```

Iterative

```
//iteratively
        ListNode* prev= NULL;
        ListNode* reshead=NULL;
        while(head)
        {
            ListNode* left= head;
            int ct=1;
            while(head->next && ct<k){
                head= head->next;
                ct++;
            }

            ListNode* right= head;
            head= head->next;

            if(ct<k){
                if(prev!=NULL)
                    prev->next= left;
                else
                    reshead= left;
            }

            else{
                //reverse the segment
                ListNode* lazy= NULL;
                ListNode* move= left;
                while(lazy!=right){
                    ListNode* temp= move->next;
                    move->next= lazy;
                    lazy= move;
                    move= temp;
                }

                // cout<<left->val<<" "<<right->val<<endl;
                if(prev!=NULL)
                    prev->next= right;
                prev= left;
                if(reshead==NULL)
                    reshead= right;

            }
        }
        return reshead;
```

43. Multiply Strings

Given two non-negative integers `num1` and `num2` represented as strings, return the product of `num1` and `num2`, also represented as a string.

Note: You must not use any built-in BigInteger library or convert the inputs to integer directly.

Example 1:

```
Input: num1 = "2", num2 = "3"
Output: "6"
```

Example 2:

```
Input: num1 = "123", num2 = "456"
Output: "56088"
```

Constraints:

- `1 <= num1.length, num2.length <= 200`
- `num1` and `num2` consist of digits only.
- Both `num1` and `num2` do not contain any leading zero, except the number `0` itself.

I recomment to follow the same format for this question to avoid WA by missing cases #1 using O(n1+n2) space

```
string multiply(string num1, string num2) {

        int n1= num1.length();
        int n2= num2.length();

        int ct=0;
        vector<int> res(n1+n2+10,0);
        vector<int> temp;
        for(int i=n2-1;i>=0;i--){
            temp.clear();
            for(int p=0;p<ct;p++)
                temp.push_back(0);

            int carry=0;
            for(int j=n1-1;j>=0;j--){
                int e1= num2[i]-'0';
                int e2= num1[j]-'0';

                int mul= e1*e2+carry;
                temp.push_back(mul%10);
                carry= mul/10;
            }
            if(carry>0)
                temp.push_back(carry);
            //add the temp to res
            carry=0;
            for(int k=0;k<temp.size();k++){
                int xx= res[k]+temp[k]+carry;
                res[k]=xx%10;
                carry= xx/10;
            }
            if(carry)
                res[temp.size()]+=carry;

            ct++; //to increment the places in next multiplication
        }

        //reconstruct ans

        while(res.size() && res.back()==0) //trailing zeroes
            res.pop_back();

        if(res.size()==0)
            return "0";

        reverse(res.begin(),res.end());
        string ans="";
        for(int i=0;i<res.size();i++)
```

```
            ans.push_back(res[i]+'0');
        return ans;
    }
```

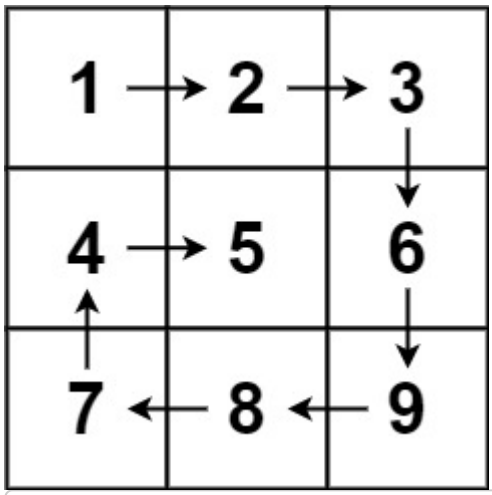
Approach: 2 using constant space for eval use indexing to keep track

```
string multiply(string num1, string num2) {

        int n1= num1.length();
        int n2= num2.length();

        vector<int> res(n1+n2+1,0);
        int itr1=0,itr2=0;

        for(int i=n2-1;i>=0;i--){
            int carry=0;
            itr2=0;
            for(int j=n1-1;j>=0;j--){
                int mul = (num1[j]-'0')*(num2[i]-'0')+res[itr1+itr2]+carry;
                carry= mul/10;
                // cout<<mul<<" "<<itr1<<" "<<itr2<<endl;
                res[itr1+itr2]= mul%10;
                itr2++;
            }
            if(carry)
                res[itr1+itr2]+=carry;
            itr1++;
        }

        // for(int i=0;i<res.size();i++)
        //     cout<<res[i]<<" ";
        while(res.size() && res.back()==0)
            res.pop_back();

        if(res.size()==0)
            return "0";

        string ans="";
        for(int i=res.size()-1;i>=0;i--)
            ans.push_back(res[i]+'0');

        return ans;
    }
```

54. Spiral Matrix

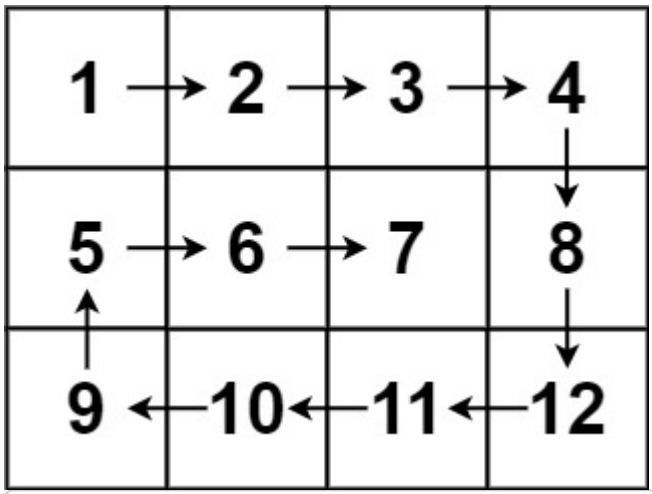
Given an `m x n matrix`, return *all elements of the* `matrix` *in spiral order*.

Example 1:




```
Input: matrix = [[1,2,3],[4,5,6],[7,8,9]]
Output: [1,2,3,6,9,8,7,4,5]
```

Example 2:




```
Input: matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]
Output: [1,2,3,4,8,12,11,10,9,5,6,7]
```

Constraints:

- `m == matrix.length`
- `n == matrix[i].length`
- `1 <= m, n <= 10`
- `-100 <= matrix[i][j] <= 100`

cramp it full

```
vector<int> res;
    int ct;
    void foo(int r1, int r2, int c1, int c2, vector<vector<int>> &matrix, int tot)
    {
        for(int i=c1;i<=c2 && ++ct<=tot;i++)
            res.push_back(matrix[r1][i]);

        for(int i=r1+1;i<=r2 && ++ct<=tot;i++)
            res.push_back(matrix[i][c2]);

        for(int i=c2-1;i>=c1 && ++ct<=tot;i--)
            res.push_back(matrix[r2][i]);

        for(int i=r2-1;i>r1 && ++ct<=tot;i--)
            res.push_back(matrix[i][c1]);

    }
    vector<int> spiralOrder(vector<vector<int>>& matrix) {

        int n= matrix.size();
        int m= matrix[0].size();

        res.clear();
        ct=0;
        int r1=0,r2=n-1,c1=0,c2=m-1;
        while(r1<=r2 && c1<=c2){
            foo(r1,r2,c1,c2,matrix,n*m);
            r1++;
            r2--;
            c1++;
            c2--;
        }
        return res;
    }
```

---

56. Merge Intervals

Given an array of `intervals` where `intervals[i] = [start`$_i$`, end`$_i$`]`, merge all overlapping intervals, and return *an array of the non-overlapping intervals that cover all the intervals in the input*.

Example 1:

```
Input: intervals = [[1,3],[2,6],[8,10],[15,18]]
Output: [[1,6],[8,10],[15,18]]
Explanation: Since intervals [1,3] and [2,6] overlaps, merge them into [1,6].
```

Example 2:

```
Input: intervals = [[1,4],[4,5]]
Output: [[1,5]]
Explanation: Intervals [1,4] and [4,5] are considered overlapping.
```

Constraints:

- `1 <= intervals.length <= 10^4`
- `intervals[i].length == 2`
- `0 <= start_i <= end_i <= 10^4`

```
vector<vector<int>> merge(vector<vector<int>>& intervals) {

        int n= intervals.size();

        vector<vector<int>> res;
        sort(intervals.begin(),intervals.end());
        int l= intervals[0][0];
        int r= intervals[0][1];
        for(int i=1;i<n;i++){

            if(intervals[i][0]> r){
                //intert the intervals
                res.push_back(vector<int>{l,r});
                l= intervals[i][0];
                r= intervals[i][1];
            }
            r= max(r,intervals[i][1]);
        }
        res.push_back(vector<int>{l,r});
        return res;
    }
```

75. Sort Colors

Given an array `nums` with `n` objects colored red, white, or blue, sort them **in-place (https://en.wikipedia.org/wiki/In-place_algorithm)** so that objects of the same color are adjacent, with the colors in the order red, white, and blue.

We will use the integers `0`, `1`, and `2` to represent the color red, white, and blue, respectively.

You must solve this problem without using the library's sort function.

Example 1:

```
Input: nums = [2,0,2,1,1,0]
Output: [0,0,1,1,2,2]
```

Example 2:

```
Input: nums = [2,0,1]
Output: [0,1,2]
```

Example 3:

```
Input: nums = [0]
Output: [0]
```

Example 4:

```
Input: nums = [1]
Output: [1]
```

Constraints:

- `n == nums.length`
- `1 <= n <= 300`
- `nums[i]` is `0`, `1`, or `2`.

Follow up: Could you come up with a one-pass algorithm using only constant extra space?

**Bhai rat le isko dobara smajhna mat dimaag hi kharab hoga non intutive hain **

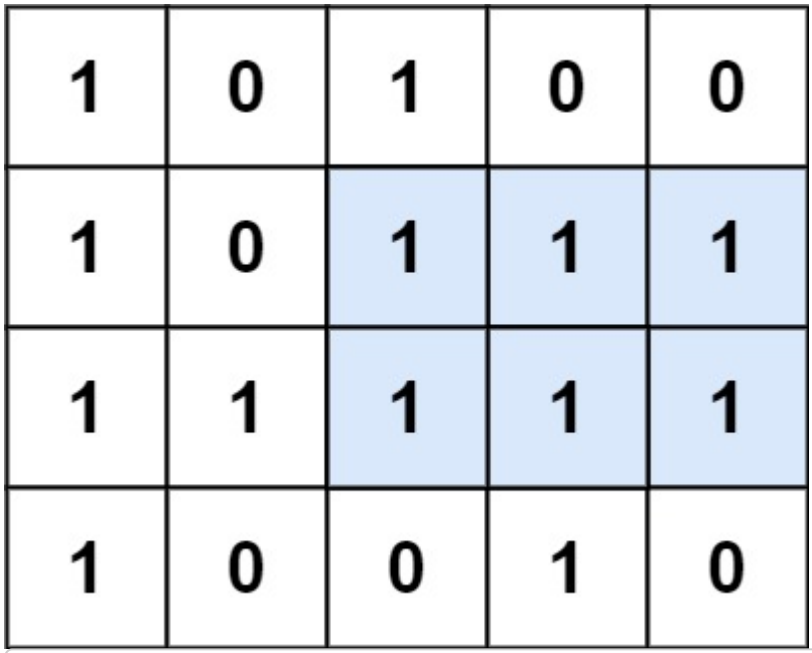
```
int n= nums.size();
        int ptr1=0,ptr2=0,ptr3=n-1;

        while(ptr2<=ptr3){
            if(nums[ptr2]==1)
                ptr2++;
            else if(nums[ptr2]==0)
                swap(nums[ptr2++],nums[ptr1++]);
            else
                swap(nums[ptr2],nums[ptr3--]);
        }
        return;
```

---

85. Maximal Rectangle

Given a `rows x cols` binary `matrix` filled with `0`'s and `1`'s, find the largest rectangle containing only `1`'s and return *its area*.

Example 1:




```
Input: matrix = [["1","0","1","0","0"],["1","0","1","1","1"],["1","1","1","1","1"],["1"
Output: 6
Explanation: The maximal rectangle is shown in the above picture.
```

Example 2:

```
Input: matrix = []
Output: 0
```

Example 3:

```
Input: matrix = [["0"]]
Output: 0
```

Example 4:

```
Input: matrix = [["1"]]
Output: 1
```

Example 5:

```
Input: matrix = [["0","0"]]
Output: 0
```

Constraints:

- `rows == matrix.length`
- `cols == matrix[i].length`
- `0 <= row, cols <= 200`
- `matrix[i][j]` is `'0'` or `'1'`.

---

question solved using stack approach of largerst area rectangle in a histogram

```
int foo(vector<int> &his, int n)
    {
        vector<int> dp1(n,-1),dp2(n,n);
        stack<array<int,2>> stk;
        //next smaller to left
        for(int i=n-1;i>=0;i--){
            while(!stk.empty() && stk.top()[0]>his[i]){
                dp1[stk.top()[1]]= i;
                stk.pop();
            }
            stk.push({his[i],i});
        }
        while(!stk.empty()) stk.pop();
        //next smaller to right
        for(int i=0;i<n;i++){
            while(!stk.empty() && stk.top()[0]>his[i]){
                dp2[stk.top()[1]]= i;
                stk.pop();
            }
            stk.push({his[i],i});
        }

        int res=0;
        //accumulate ans
        for(int i=0;i<n;i++){
            res= max(res,his[i]);
            res= max(res,his[i]*(dp2[i]-dp1[i]-1));
        }
        return res;
    }
    int maximalRectangle(vector<vector<char>>& matrix) {

        int n= matrix.size();
        if(n==0) return 0;
        int m= matrix[0].size();

        int ans=0;
        vector<int> his(m,0);

        //traverse up to down (sweep)
        for(int i=0;i<n;i++){
            for(int j=0;j<m;j++){
                if(matrix[i][j]=='0')
                    his[j]=0;
                else
                    his[j]+=1;
            }
```

```
            int res= foo(his,m);
            // cout<<res<<'\n';
            ans= max(ans,res);
        }
        //ans
        return ans;
    }
```

87. Scramble String

We can scramble a string s to get a string t using the following algorithm:

1. If the length of the string is 1, stop.
2. If the length of the string is > 1, do the following:
    - Split the string into two non-empty substrings at a random index, i.e., if the string is `s`, divide it to `x` and `y` where `s = x + y`.
    - **Randomly** decide to swap the two substrings or to keep them in the same order. i.e., after this step, `s` may become `s = x + y` or `s = y + x`.
    - Apply step 1 recursively on each of the two substrings `x` and `y`.

Given two strings `s1` and `s2` of **the same length**, return `true` if `s2` is a scrambled string of `s1`, otherwise, return `false`.

Example 1:

```
Input: s1 = "great", s2 = "rgeat"
Output: true

Explanation: One possible scenario applied on s1 is:
"great" --> "gr/eat" // divide at random index.
"gr/eat" --> "gr/eat" // random decision is not to swap the two substrings and keep the
"gr/eat" --> "g/r / e/at" // apply the same algorithm recursively on both substrings. d
"g/r / e/at" --> "r/g / e/at" // random decision was to swap the first substring and to
"r/g / e/at" --> "r/g / e/ a/t" // again apply the algorithm recursively, divide "at" t
"r/g / e/ a/t" --> "r/g / e/ a/t" // random decision is to keep both substrings in the
The algorithm stops now and the result string is "rgeat" which is s2.
As there is one possible scenario that led s1 to be scrambled to s2, we return true.
```

Example 2:

```
Input: s1 = "abcde", s2 = "caebd"
Output: false
```

Example 3:

```
Input: s1 = "a", s2 = "a"
Output: true
```

Constraints:

- `s1.length == s2.length`
- `1 <= s1.length <= 30`
- `s1` and `s2` consist of lower-case English letters.

IMPLEMENTATION BASED ON Matrix Chain Multiplication

```
class Solution {
public:
    unordered_map<string,bool> mp;
    bool is_scrambled(string a, string b)
    {
        if(a==b){
            //cout<<a<<" "<<b<<endl;
            return true;
        }

        string xx= a;
        xx.push_back('_');
        xx.append(b);

        //cout<<xx<<endl;
        if(mp.find(xx)!=mp.end())
            return mp[xx];

        vector<int> az(26,0),bz(26,0);
        for(char c: a)
            az[c-'a']++;
        for(char c: b)
            bz[c-'a']++;
        if(az!=bz)
            return false;

        int n= a.length();
        //break like in mcm format
        for(int i=1;i<=n-1;i++){

            //no swap condition
            if(is_scrambled(a.substr(0,i),b.substr(0,i)) && is_scrambled(a.substr(i),
b.substr(i)))
                return mp[xx]= true;
            if(is_scrambled(a.substr(0,i),b.substr(n-i)) && is_scrambled(a.substr(i),
b.substr(0,n-i)))
                return mp[xx]= true;
        }
        return mp[xx]= false;
    }

    bool isScramble(string s1, string s2) {

        int n= s1.length();

        //check if s1 and s2 are anagrams;
        vector<int> a(26,0),b(26,0);
        for(char c: s1)
```

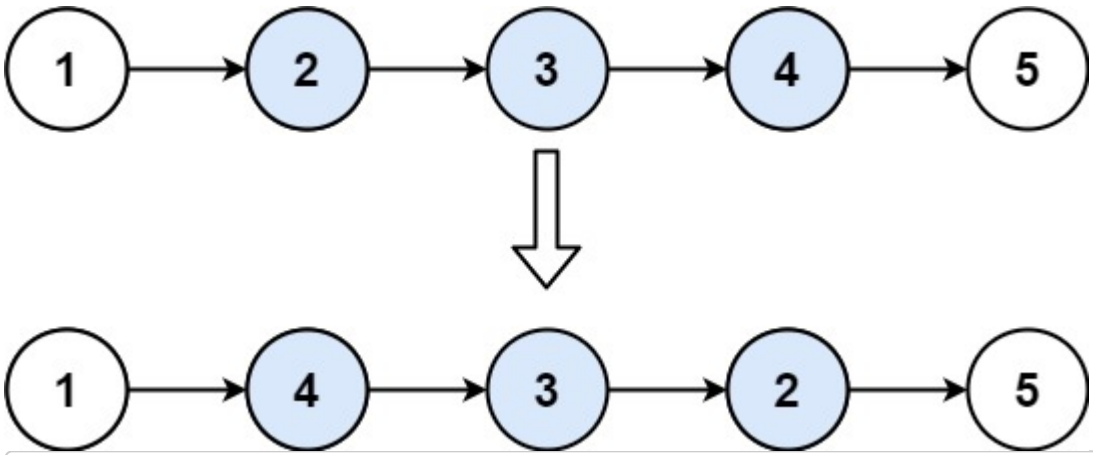
```
            a[c-'a']++;
        for(char c: s2)
            b[c-'a']++;
        if(a!=b)
            return false;

        mp.clear();
        return is_scrambled(s1,s2); //return true if s1 and s2 are scrambled strings
    }
};
```

---

92. Reverse Linked List II

Given the `head` of a singly linked list and two integers `left` and `right` where `left <= right`, reverse the nodes of the list from position `left` to position `right`, and return *the reversed list*.

Example 1:




```
Input: head = [1,2,3,4,5], left = 2, right = 4
Output: [1,4,3,2,5]
```

Example 2:

```
Input: head = [5], left = 1, right = 1
Output: [5]
```

Constraints:

- The number of nodes in the list is `n`.
- `1 <= n <= 500`

- `-500 <= Node.val <= 500`
- `1 <= left <= right <= n`

Follow up: Could you do it in one pass?

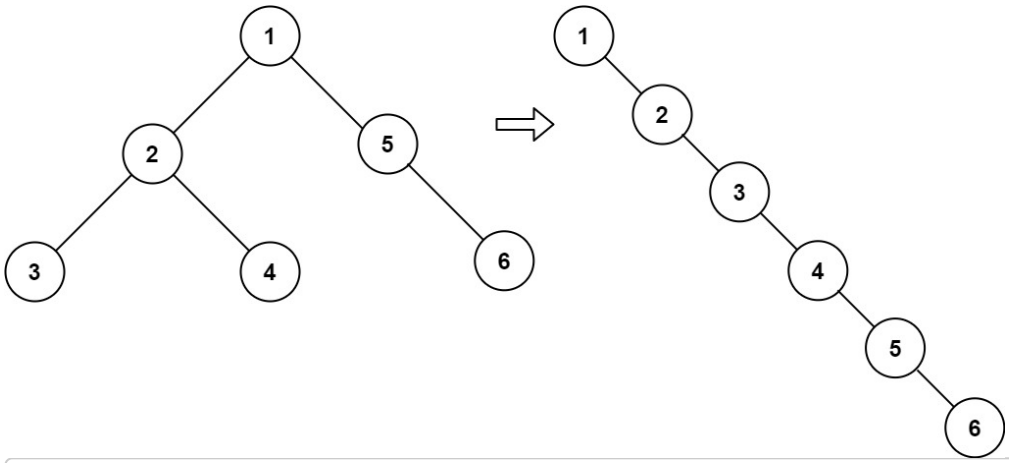
```
ListNode* reverseBetween(ListNode* head, int left, int right) {

        //trick in linked list question
        //when the head of linekd list might change create a dummy node

        if(head==NULL)
            return NULL;

        ListNode* dummy= new ListNode(-1);
        dummy->next=head;
        head= dummy;

        // for reversing a segment place four ptrs s,l,r,t
        ListNode* s=head;
        for(int i=1;i<=left-1;i++)
            s= s->next;

        ListNode* l= s->next;
        ListNode* r= l;
        ListNode* t=NULL;
        for(int i=1;i<=(right-left);i++)
            r= r->next;
        t= r->next;

        ListNode* prev= NULL;
        ListNode* move= l;
        while(prev!=r){
            ListNode* temp= move->next;
            move->next=prev;
            prev= move;
            move= temp;
        }

        s->next= r;
        l->next=t;

        return dummy->next;
    }
```

114. Flatten Binary Tree to Linked List

Given the `root` of a binary tree, flatten the tree into a "linked list":

- The "linked list" should use the same `TreeNode` class where the `right` child pointer points to the next node in the list and the `left` child pointer is always `null`.
- The "linked list" should be in the same order as a **pre-order traversal** (https://en.wikipedia.org/wiki/Tree_traversal#Pre-order,_NLR) of the binary tree.

Example 1:




```
Input: root = [1,2,5,3,4,null,6]
Output: [1,null,2,null,3,null,4,null,5,null,6]
```

Example 2:

```
Input: root = []
Output: []
```

Example 3:

```
Input: root = [0]
Output: [0]
```

Constraints:

- The number of nodes in the tree is in the range `[0, 2000]`.
- `-100 <= Node.val <= 100`

Follow up: Can you flatten the tree in-place (with `O(1)` extra space)?

iss question ko to ek dum cram kr lena recursive

```
void flatten(TreeNode* root) {

        if(root==NULL)
            return;

        flatten(root->left);
        flatten(root->right);

        if(root->left){
            TreeNode* move= root->left;
            while(move->right)
                move= move->right;

            move->right= root->right;
            root->right= root->left;
            root->left=NULL;
        }
    }
```

115. Distinct Subsequences

Given two strings `s` and `t`, return *the number of distinct subsequences of `s` which equals `t`*.

A string's **subsequence** is a new string formed from the original string by deleting some (can be none) of the characters without disturbing the remaining characters' relative positions. (i.e., `"ACE"` is a subsequence of `"ABCDE"` while `"AEC"` is not).

It is guaranteed the answer fits on a 32-bit signed integer.

Example 1:

```
Input: s = "rabbbit", t = "rabbit"
Output: 3
Explanation:
As shown below, there are 3 ways you can generate "rabbit" from S.
rabbbit
rabbbit
rabbbit
```

Example 2:

Input: s = "babgbag", t = "bag"
Output: 5
Explanation:
As shown below, there are 5 ways you can generate "bag" from S.
bab**g**bag
babgba**g**
babgb**ag**
ba**b**gb**ag**
babg**bag**

Constraints:

- 1 <= s.length, t.length <= 1000
- s and t consist of English letters.

---

nice question on LCS variation

```
int numDistinct(string s, string t) {

        int n= s.length();
        int m= t.length();

        vector<vector<long long>> dp(n+1,vector<long long>(m+1));
        for(int j=1;j<=m;j++)
            dp[0][j]=0;
        for(int i=0;i<=n;i++)
            dp[i][0]=1;

        for(int i=1;i<=n;i++){
            for(int j=1;j<=m;j++){
                if(s[i-1]!=t[j-1])
                    dp[i][j]= dp[i-1][j];
                else
                    dp[i][j]= dp[i-1][j]+dp[i-1][j-1];
            }
        }

        // for(int i=0;i<=n;i++){
        //     for(int j=0;j<=m;j++){
        //         cout<<dp[i][j]<<" ";
        //     }
        //     cout<<endl;
        // }

        return dp[n][m];
    }
```

116. Populating Next Right Pointers in Each Node

You are given a **perfect binary tree** where all leaves are on the same level, and every parent has two children. The binary tree has the following definition:

```
struct Node {
  int val;
  Node *left;
  Node *right;
  Node *next;
}
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to `NULL`.

Initially, all next pointers are set to `NULL`.

Follow up:

- You may only use constant extra space.
- Recursive approach is fine, you may assume implicit stack space does not count as extra space for this problem.

Example 1:

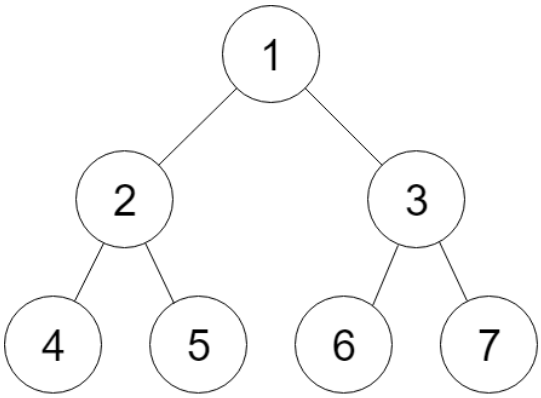


Figure A

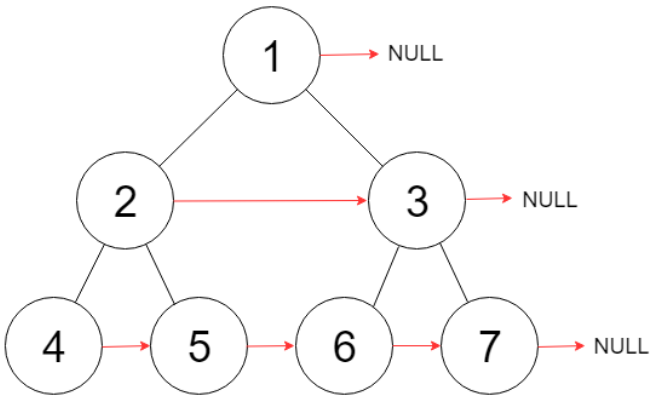


Figure B

```
Input: root = [1,2,3,4,5,6,7]
Output: [1,#,2,3,#,4,5,6,7,#]
Explanation: Given the above perfect binary tree (Figure A), your function should popul
```

Constraints:

- The number of nodes in the given tree is less than `4096`.
- `-1000 <= node.val <= 1000`

---

approach 1: using level order traversal approach 2: using level order with the help of next pointer

```
Node* connect(Node* root) {
       if(root==NULL)
           return root;

       Node* start_level= root;
       while(start_level->left){
           Node* move= start_level;
           while(move){
               move->left->next= move->right;
               move->right->next= (move->next)? move->next->left: NULL;
               move=move->next;
           }
           start_level= start_level->left;
       }
       return root;
   }
```

approach 3: using recursion

```
void foo(Node* L, Node* R)
    {
        if(L==NULL && R==NULL)
            return;

        L->next= R;
        foo(L->left,L->right);
        foo(R->left,R->right);
        foo(L->right,R->left);
    }

    Node* connect(Node* root) {

        if(root==NULL)
            return root;

        foo(root->left,root->right);
        return root;
    }
```

122. Best Time to Buy and Sell Stock II

You are given an array `prices` where `prices[i]` is the price of a given stock on the i^{th} day.

Find the maximum profit you can achieve. You may complete as many transactions as you like (i.e., buy one and sell one share of the stock multiple times).

Note: You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

Example 1:

```
Input: prices = [7,1,5,3,6,4]
Output: 7
Explanation: Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit = 5-1 = 4.
Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit = 6-3 = 3.
```

Example 2:

```
Input: prices = [1,2,3,4,5]
Output: 4
Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 = 4.
Note that you cannot buy on day 1, buy on day 2 and sell them later, as you are engagin
```

Example 3:

```
Input: prices = [7,6,4,3,1]
Output: 0
Explanation: In this case, no transaction is done, i.e., max profit = 0.
```

Constraints:

- `1 <= prices.length <= 3 * 10^4`
- `0 <= prices[i] <= 10^4`

GREEDY int maxProfit(vector& prices) {

```
    int n= prices.size();
    int ans=0;

    int l=0,r=0;

    while(r<n){
        while(r+1<n && prices[r+1]>prices[r])
            r++;
        ans+= prices[r]-prices[l];
        l=r;

        while(r+1<n && prices[r+1]<=prices[r]){
            r++;
            l++;
        }
        r++;
    }
    return ans;
}
```

---

123. Best Time to Buy and Sell Stock III

You are given an array `prices` where `prices[i]` is the price of a given stock on the `i`th day.

Find the maximum profit you can achieve. You may complete **at most two transactions**.

Note: You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

Example 1:

```
Input: prices = [3,3,5,0,0,3,1,4]
Output: 6

Explanation: Buy on day 4 (price = 0) and sell on day 6 (price = 3), profit = 3-0 = 3.
Then buy on day 7 (price = 1) and sell on day 8 (price = 4), profit = 4-1 = 3.
```

Example 2:

```
Input: prices = [1,2,3,4,5]
Output: 4
Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 = 4.
Note that you cannot buy on day 1, buy on day 2 and sell them later, as you are engagin
```

Example 3:

```
Input: prices = [7,6,4,3,1]
Output: 0
Explanation: In this case, no transaction is done, i.e. max profit = 0.
```

Example 4:

```
Input: prices = [1]
Output: 0
```

Constraints:

- `1 <= prices.length <= 10^5`
- `0 <= prices[i] <= 10^5`

---

int maxProfit(vector& prices) {

```
    int n= prices.size();
    int dp1[n],dp2[n];

    dp1[0]=0;
    int mnb= prices[0];
    for(int i=1;i<n;i++){
        dp1[i]= max(dp1[i-1],prices[i]-mnb);
        mnb= min(mnb,prices[i]);
    }

    dp2[n-1]=0;
    int mxs=prices[n-1];
    for(int i=n-2;i>=0;i--){
        dp2[i]= max(dp2[i+1],mxs-prices[i]);
        mxs= max(mxs,prices[i]);
    }

    int ans=dp1[n-1];
    for(int i=1;i<n;i++)
        ans= max(ans,dp1[i-1]+dp2[i]);

    return ans;
}
```

---

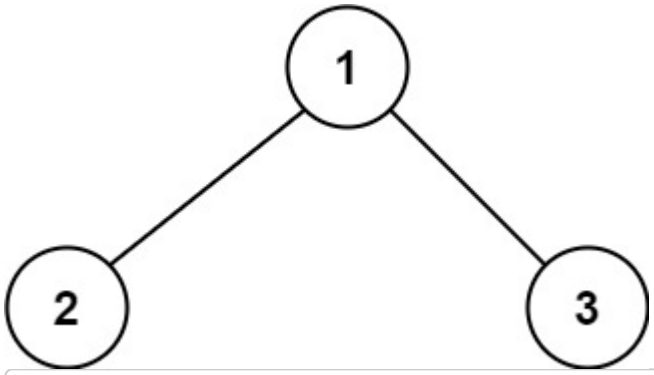
124. Binary Tree Maximum Path Sum

A **path** in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence **at most once**. Note that the path does not need to pass through the root.

The **path sum** of a path is the sum of the node's values in the path.

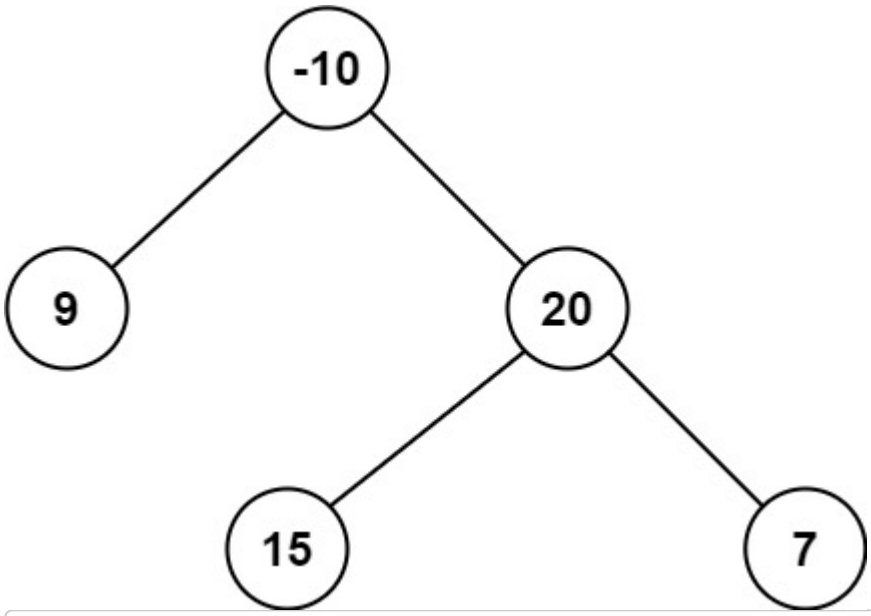
Given the `root` of a binary tree, return *the maximum **path sum** of any path*.

Example 1:




```
Input: root = [1,2,3]
Output: 6
Explanation: The optimal path is 2 -> 1 -> 3 with a path sum of 2 + 1 + 3 = 6.
```

Example 2:




```
Input: root = [-10,9,20,null,null,15,7]
Output: 42
Explanation: The optimal path is 15 -> 20 -> 7 with a path sum of 15 + 20 + 7 = 42.
```

Constraints:

- The number of nodes in the tree is in the range `[1, 3 * 10^4]`.
- `-1000 <= Node.val <= 1000`

This ques is DP on tree: There is another imp variation max path sum from leaf to another leaf question is available on gfg

TRICK : make decision on ans and what to return at every node

```
int ans;
    int foo(TreeNode* root)
    {
        if(root==NULL) return 0;

        int lans= foo(root->left);
        int rans= foo(root->right);

        ans= max(ans,root->val+lans+rans);
        ans= max(ans,root->val+max(0,max(lans,rans)));

        return max(root->val,root->val+max(lans,rans));
    }
    int maxPathSum(TreeNode* root) {

        ans=INT_MIN;
        foo(root);
        return ans;
    }
```

---

138. Copy List with Random Pointer

A linked list of length `n` is given such that each node contains an additional random pointer, which could point to any node in the list, or `null`.

Construct a **deep copy** (https://en.wikipedia.org/wiki/Object_copying#Deep_copy) of the list. The deep copy should consist of exactly `n` **brand new** nodes, where each new node has its value set to the value of its corresponding original node. Both the `next` and `random` pointer of the new nodes should point to new nodes in the copied list such that the pointers in the original list and copied list represent the same list state. **None of the pointers in the new list should point to nodes in the original list**.

For example, if there are two nodes `X` and `Y` in the original list, where `X.random --> Y`, then for the corresponding two nodes `x` and `y` in the copied list, `x.random --> y`.

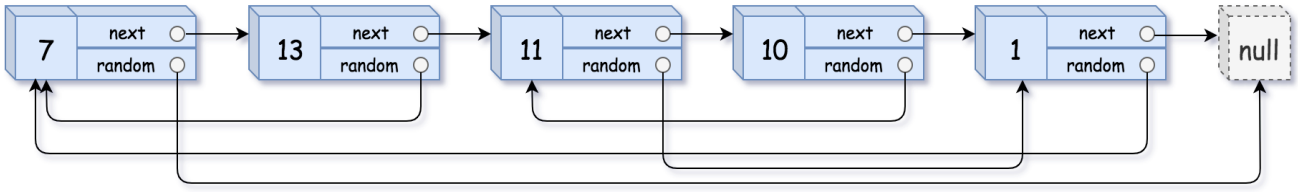
Return *the head of the copied linked list*.

The linked list is represented in the input/output as a list of `n` nodes. Each node is represented as a pair of `[val, random_index]` where:

- `val`: an integer representing `Node.val`
- `random_index`: the index of the node (range from `0` to `n-1`) that the `random` pointer points to, or `null` if it does not point to any node.

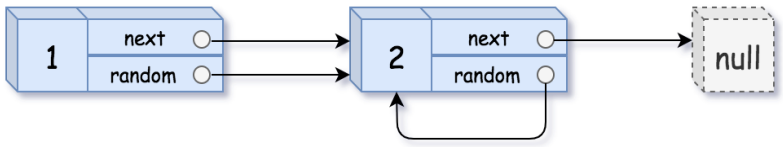
Your code will **only** be given the `head` of the original linked list.

Example 1:



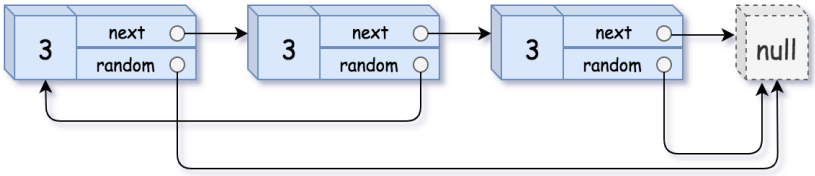

```
Input: head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
Output: [[7,null],[13,0],[11,4],[10,2],[1,0]]
```

Example 2:




```
Input: head = [[1,1],[2,1]]
Output: [[1,1],[2,1]]
```

Example 3:




```
Input: head = [[3,null],[3,0],[3,null]]
Output: [[3,null],[3,0],[3,null]]
```

Example 4:

```
Input: head = []
Output: []
Explanation: The given linked list is empty (null pointer), so return null.
```

Constraints:

- `0 <= n <= 1000`

- `-10000 <= Node.val <= 10000`
- `Node.random` is `null` or is pointing to some node in the linked list.

---

chances are high this question can be asked easy trick

approach 1 : with extra memory

```
    Node* copyRandomList(Node* head) {

        unordered_map<Node*, Node*> mp;
        mp[NULL]= NULL;

        Node* newhead=NULL;
        Node* move= head;
        Node* track=NULL;
        while(move){
            Node* nn= new Node(move->val);
            if(track==NULL){
                newhead= nn;
                track=nn;
            }
            else{
                track->next= nn;
                track= nn;
            }

            mp[move]= nn;
            move=move->next;
        }

        Node* temp= newhead;
        move= head;
        while(temp){
            temp->random= mp[head->random];
            temp= temp->next;
            head= head->next;
        }
        return newhead;
    }
```

approach 2: constant extra memory (this one is required must)

```
    Node* copyRandomList(Node* head) {

        if(head==NULL)
            return NULL;

        Node* move= head;
        while(move){
            Node* temp= new Node(move->val);
            temp->next= move->next;
            move->next=temp;
            move=move->next->next;
        }

        //set random pointers

        move= head;
        while(move){
            if(move->random==NULL)
                move->next->random=NULL;
            else
                move->next->random= move->random->next;
            move=move->next->next;
        }

//         extract the deep copy and rebuild orignal linked list
        Node* res=NULL;
        move=head;
        while(move){
            Node* temp= move->next;
            move->next= move->next->next;
            move=move->next;
            if(move)
                temp->next= move->next;
            if(res==NULL)
                res= temp;
        }
        return res;
    }
```

146. LRU Cache

Design a data structure that follows the constraints of a **Least Recently Used (LRU) cache (https://en.wikipedia.org/wiki/Cache_replacement_policies#LRU)**.

Implement the `LRUCache` class:

- `LRUCache(int capacity)` Initialize the LRU cache with **positive** size `capacity` .
- `int get(int key)` Return the value of the `key` if the key exists, otherwise return `-1` .
- `void put(int key, int value)` Update the value of the `key` if the `key` exists. Otherwise, add the `key-value` pair to the cache. If the number of keys exceeds the `capacity` from this operation, **evict** the least recently used key.

The functions `get` and `put` must each run in `O(1)` average time complexity.

Example 1:

```
Input
["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]
Output
[null, null, null, 1, null, -1, null, -1, 3, 4]

Explanation
LRUCache lRUCache = new LRUCache(2);
lRUCache.put(1, 1); // cache is {1=1}
lRUCache.put(2, 2); // cache is {1=1, 2=2}
lRUCache.get(1);    // return 1
lRUCache.put(3, 3); // LRU key was 2, evicts key 2, cache is {1=1, 3=3}
lRUCache.get(2);    // returns -1 (not found)
lRUCache.put(4, 4); // LRU key was 1, evicts key 1, cache is {4=4, 3=3}
lRUCache.get(1);    // return -1 (not found)
lRUCache.get(3);    // return 3
lRUCache.get(4);    // return 4
```

Constraints:

- `1 <= capacity <= 3000`
- `0 <= key <=` 10^4
- `0 <= value <=` 10^5
- At most 2 `*` 10^5 calls will be made to `get` and `put` .

** ek bar aur likh lena galti ho rhi must use dummy head and tail **

```
struct DDLNode{
    int key;
    int val;
    DDLNode* prev;
    DDLNode* next;
    DDLNode(int key, int val)
    {
        this->key= key;
        this->val= val;
        prev=NULL;
        next=NULL;
    }
};

void insertToHead(DDLNode* head, DDLNode* curr)
{
    curr->next= head->next;
    head->next->prev= curr;
    head->next= curr;
    curr->prev= head;
    return;
}

void deleteNode(DDLNode* curr)
{
    curr->prev->next= curr->next;
    curr->next->prev= curr->prev;
}

class LRUCache {
public:
    int sz;
    int cap;
    unordered_map<int,DDLNode*> mp;
    DDLNode* head;
    DDLNode* tail;
    LRUCache(int capacity) {
        sz=0;
        cap= capacity;
        mp.clear();

        //create dummy head and dummy tail
        head= new DDLNode(-1,-1);
        tail= new DDLNode(-1,-1);
        head->next=tail;
        tail->prev= head;

    }
```

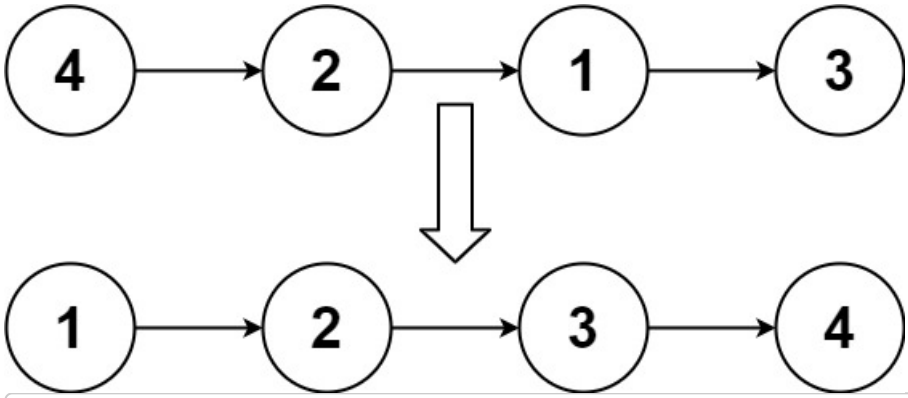
```
    int get(int key) {

        if(mp.find(key)==mp.end())
            return -1;

        deleteNode(mp[key]);
        insertToHead(head,mp[key]);
        return mp[key]->val;
    }

    void put(int key, int value) {
        if(mp.find(key) != mp.end()){
            mp[key]->val= value;
            deleteNode(mp[key]);
            insertToHead(head,mp[key]);
        }
        else{
            if(sz==cap){
                DDLNode* xx= tail->prev;
                deleteNode(tail->prev);
                mp.erase(xx->key);
                delete xx;
                xx=NULL;

                DDLNode* newnode= new DDLNode(key,value);
                insertToHead(head,newnode);
                mp[key]= newnode;
            }
            else{
                DDLNode* newnode= new DDLNode(key,value);
                insertToHead(head,newnode);
                mp[key]= newnode;
                sz++;
            }
        }
    }
};

/**
 * Your LRUCache object will be instantiated and called as such:
 * LRUCache* obj = new LRUCache(capacity);
 * int param_1 = obj->get(key);
 * obj->put(key,value);
 */
```

148. Sort List

Given the `head` of a linked list, return *the list after sorting it in **ascending order***.

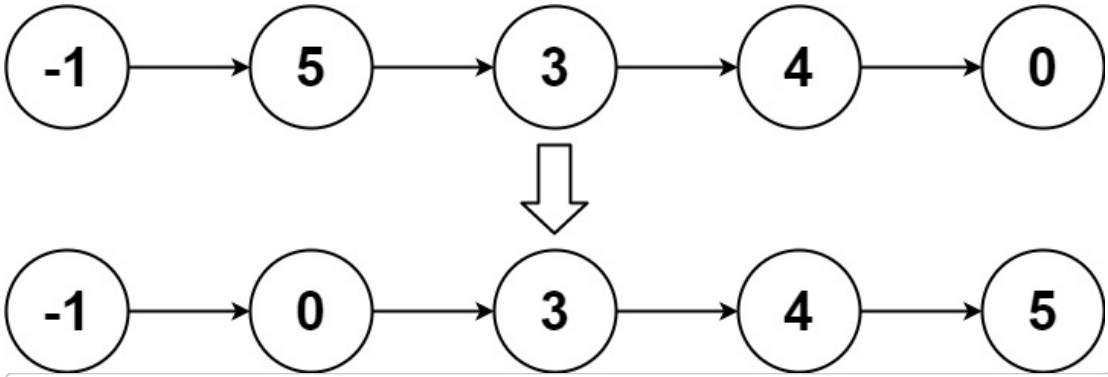
Follow up: Can you sort the linked list in `O(n logn)` time and `O(1)` memory (i.e. constant space)?

Example 1:




```
Input: head = [4,2,1,3]
Output: [1,2,3,4]
```

Example 2:




```
Input: head = [-1,5,3,4,0]
Output: [-1,0,3,4,5]
```

Example 3:

```
Input: head = []
Output: []
```

Constraints:

- The number of nodes in the list is in the range `[0, 5 * 10^4]`.

- -10^5 <= Node.val <= 10^5

O(nlogn) can be done using merge sort

```
ListNode* get_mid(ListNode* curr)
   {
       ListNode* slow= curr;
       ListNode* fast= curr->next;

       while(fast!=NULL && fast->next!=NULL){
           slow= slow->next;
           fast= fast->next->next;
       }

       //slow->next point to mid partition
       ListNode* temp= slow->next;
       slow->next=NULL;
       return temp;
   }

   ListNode* merge_util(ListNode* lres, ListNode* rres)
   {
       ListNode* temp=NULL;
       ListNode* reshead=NULL;
       while(lres!=NULL && rres!=NULL)
       {
           if(lres->val <= rres->val){
               ListNode* nn= new ListNode(lres->val);
               if(temp==NULL){
                   temp= nn;
                   reshead=nn;
               }
               else{
                   temp->next= nn;
                   temp=temp->next;
               }
               lres=lres->next;
           }
           else{
               ListNode* nn= new ListNode(rres->val);
               if(temp==NULL){
                   temp= nn;
                   reshead=nn;
               }
               else{
                   temp->next= nn;
                   temp=temp->next;
               }
               rres=rres->next;
           }
       }
```

```
        while(lres!=NULL){
            temp->next= new ListNode(lres->val);
            lres=lres->next;
            temp= temp->next;
        }

        while(rres!=NULL){
            temp->next= new ListNode(rres->val);
            rres= rres->next;
            temp=temp->next;
        }

        return reshead;
    }

    ListNode* merge_sort(ListNode* head)
    {
        if(head->next==NULL)
            return head;

        ListNode* mid= get_mid(head);
        ListNode* lres= merge_sort(head);
        ListNode* rres= merge_sort(mid);

        return merge_util(lres,rres);

    }
    ListNode* sortList(ListNode* head) {

        //using merge sort
        if(head==NULL)
            return NULL;

        ListNode* res= merge_sort(head);
        return res;
    }
```

152. Maximum Product Subarray

Given an integer array `nums`, find a contiguous non-empty subarray within the array that has the largest product, and return *the product*.

It is **guaranteed** that the answer will fit in a **32-bit** integer.

A **subarray** is a contiguous subsequence of the array.

Example 1:

```
Input: nums = [2,3,-2,4]
Output: 6
Explanation: [2,3] has the largest product 6.
```

Example 2:

```
Input: nums = [-2,0,-1]
Output: 0
Explanation: The result cannot be 2, because [-2,-1] is not a subarray.
```

Constraints:

- `1 <= nums.length <= 2 * 10^4`
- `-10 <= nums[i] <= 10`
- The product of any prefix or suffix of `nums` is **guaranteed** to fit in a **32-bit** integer.

Important problem revise it thoroughly version 1 --> TLE prefix the multiplication O(n^2) complexity

```
        int n= nums.size();
        long long dp[n+1];
        dp[0]=1;
        int idx0[n+1];
        idx0[0]=-1;
        //dp[i]==> represents the multication till i handling the 0 case
        //ct0[i] ==> represents the idx of last zero till i
        for(int i=1;i<=n;i++){
            if(nums[i-1]==0){
                dp[i]=1;
                idx0[i]=i;
            }
            else{
                dp[i]=dp[i-1]*nums[i-1];
                idx0[i]=idx0[i-1];
            }
        }

        long long ans= -1e15*1LL;
        for(int i=1;i<=n;i++){
            for(int j=i;j>=1;j--){
                if(idx0[i]<j){
                    long long temp= dp[i]/dp[j-1];
                    // cout<<temp<<endl;
                    ans= max(ans,temp);
                }
            }
            ans= max(ans,1LL*nums[i-1]);
        }
        return ans;
```

version2: DP solution : O(n) complexity **Think similar to kadane's to get the crux of this problem**

```
int n= nums.size();
       ll ans= -1e15*1LL;
       ll maxpos=1,maxneg=1;

       for(int i=0;i<n;i++){
           if(nums[i]==0){
               //reset
               maxpos=1,maxneg=1;
               ans= max(ans,0*1LL);
           }
           else if(nums[i]>0){
               ans= max(ans, maxpos*nums[i]);
               maxpos= maxpos*nums[i];
               maxneg= maxneg*nums[i];
           }
           else{
               ans= max(ans,maxneg*nums[i]);
               ll temp= maxpos;
               maxpos= max(1*1LL,maxneg*nums[i]);
               maxneg= min(1*1LL,temp*nums[i]);
           }
           //cout<<maxpos<<" "<<maxneg<<" "<<ans<<endl;
       }
       return ans;
```

---

188. Best Time to Buy and Sell Stock IV

You are given an integer array `prices` where `prices[i]` is the price of a given stock on the i^{th} day, and an integer `k`.

Find the maximum profit you can achieve. You may complete at most `k` transactions.

Note: You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

Example 1:

```
Input: k = 2, prices = [2,4,1]
Output: 2
Explanation: Buy on day 1 (price = 2) and sell on day 2 (price = 4), profit = 4-2 = 2.
```

Example 2:

```
Input: k = 2, prices = [3,2,6,5,0,3]
Output: 7
Explanation: Buy on day 2 (price = 2) and sell on day 3 (price = 6), profit = 6-2 = 4.
```

Constraints:

- `0 <= k <= 100`
- `0 <= prices.length <= 1000`
- `0 <= prices[i] <= 1000`

---

int maxProfit(int k, vector& prices) {

```
    int n= prices.size();
    if(n==0)
        return 0;

    int dp[k+1][n];

    for(int i=0;i<=k;i++){
        int mx= -1e7*1LL;
        for(int j=0;j<n;j++){
            if(i==0)
                dp[i][j]=0;
            else if(j==0)
                dp[i][j]=0;

            else{
                int cost= prices[j]+mx;
                dp[i][j]= max(cost,dp[i][j-1]);
            }
            if(i>0) mx= max(mx,dp[i-1][j]-prices[j]);
        }
    }
    // for(int i=0;i<=k;i++){
    //     for(int j=0;j<n;j++){
    //         cout<<dp[i][j]<<" ";
    //     }
    //     cout<<'\n';
    // }
    return dp[k][n-1];
}
```

---

218. The Skyline Problem

A city's **skyline** is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. Given the locations and heights of all the buildings, return *the **skyline** formed by these buildings collectively*.

The geometric information of each building is given in the array `buildings` where `buildings[i] = [left_i, right_i, height_i]`:

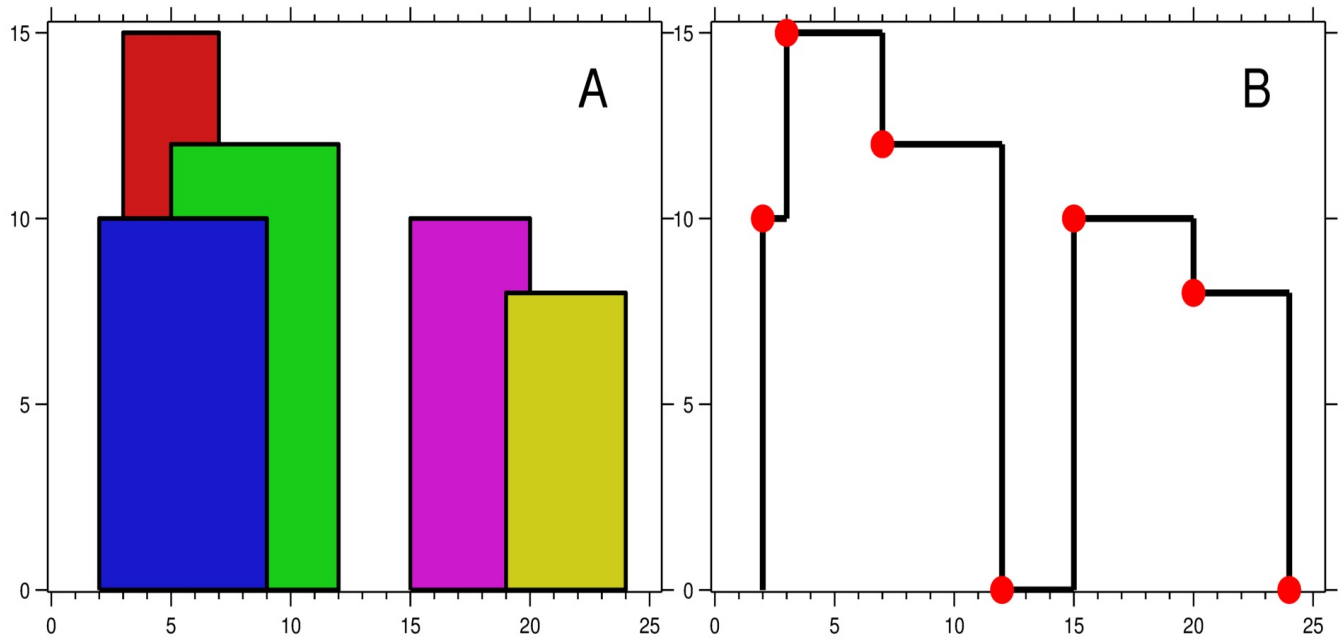
- `left_i` is the x coordinate of the left edge of the `i^th` building.
- `right_i` is the x coordinate of the right edge of the `i^th` building.
- `height_i` is the height of the `i^th` building.

You may assume all buildings are perfect rectangles grounded on an absolutely flat surface at height `0`.

The **skyline** should be represented as a list of "key points" **sorted by their x-coordinate** in the form `[[x_1,y_1],[x_2,y_2],...]`. Each key point is the left endpoint of some horizontal segment in the skyline except the last point in the list, which always has a y-coordinate `0` and is used to mark the skyline's termination where the rightmost building ends. Any ground between the leftmost and rightmost buildings should be part of the skyline's contour.

Note: There must be no consecutive horizontal lines of equal height in the output skyline. For instance, `[...,[2 3],[4 5],[7 5],[11 5],[12 7],...]` is not acceptable; the three lines of height 5 should be merged into one in the final output as such: `[...,[2 3],[4 5],[12 7],...]`

Example 1:



```
Input: buildings = [[2,9,10],[3,7,15],[5,12,12],[15,20,10],[19,24,8]]
Output: [[2,10],[3,15],[7,12],[12,0],[15,10],[20,8],[24,0]]
Explanation:
Figure A shows the buildings of the input.
Figure B shows the skyline formed by those buildings. The red points in figure B repres
```

Example 2:

```
Input: buildings = [[0,2,3],[2,5,3]]
Output: [[0,3],[5,0]]
```

Constraints:

- $1 <= buildings.length <= 10^4$
- $0 <= left_i < right_i <= 2^{31} - 1$
- $1 <= height_i <= 2^{31} - 1$
- buildings is sorted by $left_i$ in non-decreasing order.

---

bhai rat lena isko acche se smjhne layak zyada nhi hain agar smjh na aye tushar roy ka video dekh le

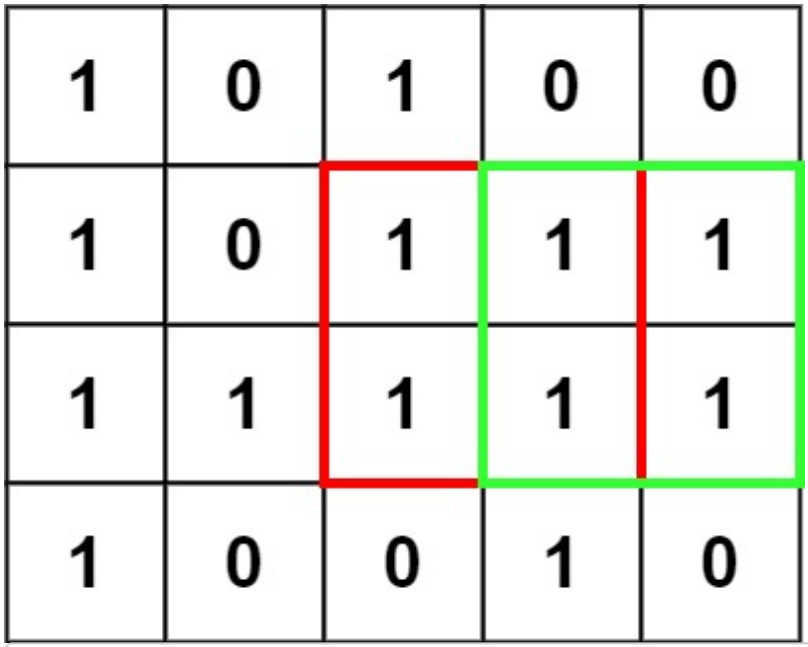
```
static bool comp(array<int,3> &a, array<int,3> &b)
    {
        if(a[0]==b[0]){
            //cases 1: s s
            if(a[2]==0 && b[2]==0)
                return a[1]>b[1];
            //case 2: e e
            else if(a[2]==1 && b[2]==1)
                return a[1]<b[1];
            //case 3: s e
            else
                return a[2]<b[2];
        }
        else
            return a[0]<b[0];
    }
    vector<vector<int>> getSkyline(vector<vector<int>>& buildings) {

        int n= buildings.size();
        vector<array<int,3>> vec;

        //0 start
        //1 end
        for(int i=0;i<n;i++){
            vec.push_back({buildings[i][0],buildings[i][2],0});
            vec.push_back({buildings[i][1],buildings[i][2],1});
        }
        //sorting step with special cases
        sort(vec.begin(),vec.end(),comp);

        multiset<int> st;
        st.insert(0);
        int mxval=0;
        vector<vector<int>> res;
        for(int i=0;i<vec.size();i++){

            if(vec[i][2]==0){
                st.insert(vec[i][1]);
                //check if maxval chaged
                if(*st.rbegin()!=mxval){
                    mxval= *st.rbegin();
                    res.push_back(vector<int>{vec[i][0],mxval});
                }
            }
            else{
                //remove the building
                st.erase(st.find(vec[i][1]));
                //check if mxval changed
```

```
                if(*st.rbegin()!=mxval){
                    mxval= *st.rbegin();
                    res.push_back(vector<int>{vec[i][0],mxval});
                }
            }
        }
        return res;
    }
```

221. Maximal Square

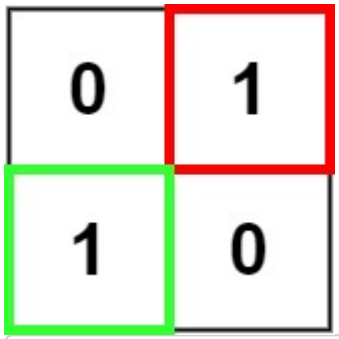
Given an `m x n` binary `matrix` filled with `0`'s and `1`'s, *find the largest square containing only* `1`'s *and return its area*.

Example 1:




```
Input: matrix = [["1","0","1","0","0"],["1","0","1","1","1"],["1","1","1","1","1"],["1"
Output: 4
```

Example 2:




```
Input: matrix = [["0","1"],["1","0"]]
Output: 1
```

Example 3:

```
Input: matrix = [["0"]]
Output: 0
```

Constraints:

- `m == matrix.length`
- `n == matrix[i].length`
- `1 <= m, n <= 300`
- `matrix[i][j]` is `'0'` or `'1'`.

---

comments are in the code for explaination

```
int maximalSquare(vector<vector<char>>& matrix) {

        int n= matrix.size();
        int m= matrix[0].size();

        int dp[n][m];

        //dp[i][j] --> represents the size of largest square with all 1's and ending
        //at i,j
        //matrix[i][j]==0 ===>   dp[i][j]=0
        //matrix[i][j]==1 ===>   dp[i][j]= 1+min(dp[i-1][j],dp[i][j-1],dp[i-1][j-1])
        int ans=0;

        for(int i=0;i<n;i++){
            for(int j=0;j<m;j++){
                if(i==0||j==0)
                    dp[i][j]= matrix[i][j]-'0';
                else{
                    if(matrix[i][j]=='0')
                        dp[i][j]=0;
                    else
                        dp[i][j]= 1+min({dp[i-1][j],dp[i-1][j-1],dp[i][j-1]});
                }
                ans= max(ans,dp[i][j]);
            }
        }
        return ans*ans;
    }
```

229. Majority Element II

Given an integer array of size `n`, find all elements that appear more than `⌊ n/3 ⌋` times.

Follow-up: Could you solve the problem in linear time and in O(1) space?

Example 1:

```
Input: nums = [3,2,3]
Output: [3]
```

Example 2:

```
Input: nums = [1]
Output: [1]
```

Example 3:

```
Input: nums = [1,2]
Output: [1,2]
```

Constraints:

- $1 <= nums.length <= 5 * 10^4$
- $-10^9 <= nums[i] <= 10^9$

THIS IS BASES ON BOYER MOORE VOTING ALGORITHM

```
vector<int> majorityElement(vector<int>& nums) {

       int n= nums.size();
       //observation is that atmost 2 majority can be present
       int ct1=0,ct2=0;
       int ele1=INT_MAX,ele2=INT_MAX;

       for(int i=0;i<n;i++){
           if(nums[i]==ele1)
               ct1++;
           else if(nums[i]==ele2)
               ct2++;
           else if(ct1==0){
               ele1= nums[i];
               ct1++;
           }
           else if(ct2==0){
               ele2= nums[i];
               ct2++;
           }
           else
               ct1--,ct2--;
       }

       vector<int> res;
       //2nd pass
       int t=0;
       int p=0;
       for(int x: nums){
           if(x==ele1)
               t++;
           if(x==ele2)
               p++;
       }
       if(t>n/3)
           res.push_back(ele1);
       if(p>n/3)
           res.push_back(ele2);

       return res;
   }
```

264. Ugly Number II

An **ugly number** is a positive integer whose prime factors are limited to `2`, `3`, and `5`.

Given an integer `n`, return *the* n^{th} ***ugly number***.

Example 1:

```
Input: n = 10
Output: 12
Explanation: [1, 2, 3, 4, 5, 6, 8, 9, 10, 12] is the sequence of the first 10 ugly numb
```

Example 2:

```
Input: n = 1
Output: 1
Explanation: 1 has no prime factors, therefore all of its prime factors are limited to
```

Constraints:

- `1 <= n <= 1690`

This is solved using 2 approaches ***using priority queue*** start with 1 as 1st ugly number start generating next greedy numbers but greedily pick smallest ugly number(use Priority queue) time complexity >NlogN <N^2

```
int nthUglyNumber(int D) {

        //greedily pick smallest element so far and then expand it to other multiples
        //check duplicates
        priority_queue<ll,vector<ll>,greater<ll>> pq;
        int a=2,b=3,c=5;
        pq.push(1);
        unordered_map<ll,bool> mp; //O(1) time lookup
        int ct=0;
        vector<int>res;
        while(ct<D){
            ll num= pq.top();
            pq.pop();
            if(mp[num]==true)
                continue;

            res.push_back(num);
            mp[num]=true;
            ct++;
            if(ct==D)
                return num;

            if(mp[num*a]==false);
                pq.push(num*a);
            if(mp[num*b]==false);
                pq.push(num*b);
            if(mp[num*c]==false);
                pq.push(num*c);
        }
        return 0;
    }
```

Approach 2 : DP use previously generated ugly numbers to genrate new ugly numbers This approach is not very intutive but can be understood

```
//greedy aapproach
        vector<int> vec(n);
        vec[0]=1;
        // 1st ugly number is 1
        //generate next ugly numbers in smallest pattern from previuos ugly numbers
        int i=0,j=0,k=0;
        for(int p=1;p<n;p++){

            vec[p]= min({vec[i]*2,vec[j]*3,vec[k]*5});
            if(vec[p]==vec[i]*2) i++;
            if(vec[p]==vec[j]*3) j++;
            if(vec[p]==vec[k]*5) k++;
        }
        return vec[n-1];
```

---

287. Find the Duplicate Number

Given an array of integers `nums` containing `n + 1` integers where each integer is in the range `[1, n]` inclusive.

There is only **one repeated number** in `nums`, return *this repeated number*.

You must solve the problem **without** modifying the array `nums` and uses only constant extra space.

Example 1:

```
Input: nums = [1,3,4,2,2]
Output: 2
```

Example 2:

```
Input: nums = [3,1,3,4,2]
Output: 3
```

Example 3:

```
Input: nums = [1,1]
Output: 1
```

Example 4:

```
Input: nums = [1,1,2]
Output: 1
```

Constraints:

- `1 <= n <= 10^5`
- `nums.length == n + 1`
- `1 <= nums[i] <= n`
- All the integers in `nums` appear only **once** except for **precisely one integer** which appears **two or more** times.

Follow up:

- How can we prove that at least one duplicate number must exist in `nums` ?
- Can you solve the problem in linear runtime complexity?

---

rar le isko acche se aage peeche hote hi galat ho ja rha hain sol

```
int findDuplicate(vector<int>& nums) {

        //using floyd cycle detection algo
        int l= nums[0];
        int h= nums[0];

        do{
            l=nums[l];
            h=nums[nums[h]];
        }while(l!=h);

        l= nums[0];
        while(l!=h){
            l=nums[l];
            h=nums[h];
        }
        return l;
    }
```

---

307. Range Sum Query - Mutable

Given an integer array `nums` , handle multiple queries of the following types:

1. **Update** the value of an element in `nums`.
2. Calculate the **sum** of the elements of `nums` between indices `left` and `right` **inclusive** where `left <= right`.

Implement the `NumArray` class:

- `NumArray(int[] nums)` Initializes the object with the integer array `nums`.
- `void update(int index, int val)` **Updates** the value of `nums[index]` to be `val`.
- `int sumRange(int left, int right)` Returns the **sum** of the elements of `nums` between indices `left` and `right` **inclusive** (i.e. `nums[left] + nums[left + 1] + ... + nums[right]`).

Example 1:

```
Input
["NumArray", "sumRange", "update", "sumRange"]
[[[1, 3, 5]], [0, 2], [1, 2], [0, 2]]
Output
[null, 9, null, 8]

Explanation
NumArray numArray = new NumArray([1, 3, 5]);
numArray.sumRange(0, 2); // return 1 + 3 + 5 = 9
numArray.update(1, 2);   // nums = [1, 2, 5]
numArray.sumRange(0, 2); // return 1 + 2 + 5 = 8
```

Constraints:

- `1 <= nums.length <= 3 * 10^4`
- `-100 <= nums[i] <= 100`
- `0 <= index < nums.length`
- `-100 <= val <= 100`
- `0 <= left <= right < nums.length`
- At most `3 * 10^4` calls will be made to `update` and `sumRange`.

Notes on segment tree: max size of segment tree= 4N

time complexity to build segment tree: O(N); space complexity to build segment tree: O(N) time complexity per query: O(logN)

Why is the complexity of this algorithm O(logn)? To show this complexity we look at each level of the tree. It turns out, that for each level we only visit not more than four vertices. And since the height of the tree is O(logn), we receive the desired running time.

We can show that this proposition (at most four vertices each level) is true by induction. At the first level, we only visit one vertex, the root vertex, so here we visit less than four vertices. Now let's look at an arbitrary level. By induction hypothesis, we visit at most four vertices. If we only visit at most two vertices, the next level has at most four vertices. That trivial, because each vertex can only cause at most two recursive calls. So let's assume that we visit three or four vertices in the current level. From those vertices, we will analyze the vertices in the middle more carefully. Since the sum query asks for the sum of a continuous subarray, we know that segments corresponding to the visited vertices in the middle will be completely covered by the segment of the sum query. Therefore these vertices will not make any recursive calls. So only the most left, and the most right vertex will have the potential to make recursive calls. And those will only create at most four recursive calls, so also the next level will satisfy the assertion. We can say that one branch approaches the left boundary of the query, and the second branch approaches the right one.

Therefore we visit at most 4logn vertices in total, and that is equal to a running time of O(logn).

In conclusion the query works by dividing the input segment into several sub-segments for which all the sums are already precomputed and stored in the tree. And if we stop partitioning whenever the query segment coincides with the vertex segment, then we only need O(logn) such segments, which gives the effectiveness of the Segment Tree.

---

309. Best Time to Buy and Sell Stock with Cooldown

You are given an array `prices` where `prices[i]` is the price of a given stock on the `i`th day.

Find the maximum profit you can achieve. You may complete as many transactions as you like (i.e., buy one and sell one share of the stock multiple times) with the following restrictions:

- After you sell your stock, you cannot buy stock on the next day (i.e., cooldown one day).

Note: You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

Example 1:

```
Input: prices = [1,2,3,0,2]
Output: 3
Explanation: transactions = [buy, sell, cooldown, buy, sell]
```

Example 2:

```
Input: prices = [1]
Output: 0
```

Constraints:

- `1 <= prices.length <= 5000`
- `0 <= prices[i] <= 1000`

---

int maxProfit(vector& prices) {

```
    int n= prices.size();
    int dp[n+1][2];
    dp[0][1]=0;
    dp[1][0]=-prices[0];
    dp[1][1]=0;

    for(int i=2;i<=n;i++){
        dp[i][0]= max(dp[i-1][0],dp[i-2][1]+(-prices[i-1]));
        dp[i][1]= max(dp[i-1][1],dp[i-1][0]+prices[i-1]);
    }

    return dp[n][1];
}
```

---

378. Kth Smallest Element in a Sorted Matrix

Given an `n x n matrix` where each of the rows and columns are sorted in ascending order, return *the* k^{th} *smallest element in the matrix*.

Note that it is the k^{th} smallest element **in the sorted order**, not the k^{th} **distinct** element.

Example 1:

```
Input: matrix = [[1,5,9],[10,11,13],[12,13,15]], k = 8
Output: 13

Explanation: The elements in the matrix are [1,5,9,10,11,12,13,13,15], and the 8th smal
```

Example 2:

```
Input: matrix = [[-5]], k = 1
Output: -5
```

Constraints:

- `n == matrix.length`
- `n == matrix[i].length`
- `1 <= n <= 300`
- `-10^9 <= matrix[i][j] <= 10^9`
- All the rows and columns of `matrix` are **guaranteed** to be sorted in **non-decreasing order**.
- `1 <= k <= n^2`

Approach 1 see the matrix as n sorted arrays and use conept used in merging k sorted list TC= O(klogn)

```
int kthSmallest(vector<vector<int>>& matrix, int k) {

        int n= matrix.size();
        priority_queue<pair<int,int>,vector<pair<int,int>>,greater<pair<int,int>>> p
q;

        //see the matrix as n sorted arrays
        for(int j=0;j<n;j++)
            pq.push({matrix[0][j],j});

        vector<int> ctr(n,0);
        int ans=-1;
        for(int i=1;i<=k;i++){
            int val= pq.top().first;
            int col= pq.top().second;
            int row= ctr[col];
            pq.pop();
            ans= val;
            row++;
            ctr[col]=row;
            if(row<n)
                pq.push({matrix[row][col],col});
        }
        return ans;
    }
```

*Approac-2: using Binary search * TC=O(n*logn logn*)

```
int check(vector<vector<int>> &mat, int val)
    {
        int n= mat.size();
        int ct=0;
        for(int i=0;i<n;i++){
            int idx= upper_bound(mat[i].begin(),mat[i].end(),val)-mat[i].begin();
            ct+= idx;
        }
        return ct;
    }
    int kthSmallest(vector<vector<int>>& matrix, int k)
    {
        int n= matrix.size();
        //using binary search on ans
        int l= matrix[0][0];
        int r= matrix[n-1][n-1];

        int ans;
        while(l<=r){
            int mid= l+(r-l)/2;
            //find number of elements in the matrix less than equal to mid
            int res= check(matrix,mid);

            if(res<k)
                l=mid+1;
            else{
                ans= mid;
                r=mid-1;
            }
        }
        return ans;
    }
```

---

435. Non-overlapping Intervals

Given an array of intervals `intervals` where `intervals[i] = [start`$_i$`, end`$_i$`]`, return *the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping*.

Example 1:

```
Input: intervals = [[1,2],[2,3],[3,4],[1,3]]
Output: 1
Explanation: [1,3] can be removed and the rest of the intervals are non-overlapping.
```

Example 2:

```
Input: intervals = [[1,2],[1,2],[1,2]]
Output: 2
Explanation: You need to remove two [1,2] to make the rest of the intervals non-overlap
```

Example 3:

```
Input: intervals = [[1,2],[2,3]]
Output: 0
Explanation: You don't need to remove any of the intervals since they're already non-ov
```

Constraints:

- `1 <= intervals.length <= 2 * 10^4`
- `intervals[i].length == 2`
- `-2 * 10^4 <= start_i < end_i <= 2 * 10^4`

Approach-1: Instead of finding the intervals to romove, we can find the maximum number of non overlapping intervals and then subtract it with total intervals ot get the answer ***same code as unweighted activity scheduling(greedy)***

```
static bool comp(vector<int> &a, vector<int> &b)
    {
        return a[1]<b[1];
    }

    int eraseOverlapIntervals(vector<vector<int>>& intervals) {

        //this is similar to maximum number of activities
        //or unweighted job scheduling
        int n= intervals.size();
        sort(intervals.begin(),intervals.end(),comp);
        int ans=1;
        int lst =intervals[0][1];

        for(int i=1;i<n;i++){
            if(intervals[i][0]>=lst){
                ans++;
                lst= intervals[i][1];
            }
        }
        return n-ans;
    }
```

DP solution Code same as LIS

```
//dp solution-1
        int n= intervals.size();
        sort(intervals.begin(),intervals.end());

        vector<int> dp(n,1);
        int ans=1;
        for(int i=1;i<n;i++){
            for(int j=i-1;j>=0;j--){
                if(intervals[i][0]>=intervals[j][1]){
                    dp[i]= max(dp[i],1+dp[j]);
                }
            }
            ans= max(ans,dp[i]);
        }
        return n-ans;
```

***Dp solution -2** Code same as weighted job scheduling

```
//dp solution-1
        int n= intervals.size();
        sort(intervals.begin(),intervals.end());

        vector<int> dp(n,1);
        int ans=1;
        for(int i=1;i<n;i++){
            int ct=1;
            for(int j=i-1;j>=0;j--){
                if(intervals[i][0]>=intervals[j][1]){
                    ct+=dp[j];
                    break;
                }
            }
            dp[i]= max(dp[i-1],ct);
        }
        return n-dp[n-1];
```

451. Sort Characters By Frequency

Given a string `s`, sort it in decreasing order based on the frequency of characters, and return *the sorted string*.

Example 1:

```
Input: s = "tree"
Output: "eert"
Explanation: 'e' appears twice while 'r' and 't' both appear once.
So 'e' must appear before both 'r' and 't'. Therefore "eetr" is also a valid answer.
```

Example 2:

```
Input: s = "cccaaa"
Output: "aaaccc"
Explanation: Both 'c' and 'a' appear three times, so "aaaccc" is also a valid answer.
Note that "cacaca" is incorrect, as the same characters must be together.
```

Example 3:

```
Input: s = "Aabb"
Output: "bbAa"
Explanation: "bbaA" is also a valid answer, but "Aabb" is incorrect.
Note that 'A' and 'a' are treated as two different characters.
```

Constraints:

- `1 <= s.length <= 5 * 10^5`
- `s` consists of English letters and digits.

---

solution using bucket sort other sorting solution is easy

```
string frequencySort(string s) {

        //solution using bucket sort
        int n= s.length();
        //total buckets= atmost n

        unordered_map<char,int> mp;
        for(int c: s)
            mp[c]++;

        vector<vector<char>> buckets(n+1);

        for(auto pr: mp)
            buckets[pr.second].push_back(pr.first);

        //eval the strnig
        string ans;
        for(int i=n;i>=1;i--){
            if(buckets[i].size())
                sort(buckets[i].begin(),buckets[i].end());

            for(char c: buckets[i]){
                for(int j=0;j<i;j++)
                    ans.push_back(c);
            }
        }
        return ans;
    }
```

---

556. Next Greater Element III

Given a positive integer `n`, find *the smallest integer which has exactly the same digits existing in the integer* `n` *and is greater in value than* `n`. If no such positive integer exists, return `-1`.

Note that the returned integer should fit in **32-bit integer**, if there is a valid answer but it does not fit in **32-bit integer**, return `-1`.

Example 1:

```
Input: n = 12
Output: 21
```

Example 2:

```
Input: n = 21
Output: -1
```

Constraints:

- `1 <= n <= 2^31 - 1`

```
//  12345018760  ==> 12345068710  ==> 12345060178
        long long val= (1*1LL<<31);
        val--;

        string num= to_string(n);
        int len= num.length();
        for(int i=len-1;i>=0;i--){
            if(num[i]=='9')
                continue;

            int xx=10;
            int idx=-1;
            for(int j=i+1;j<len;j++){
                if(num[j]-'0' > num[i]-'0' && num[j]-'0'<xx){
                    xx= num[j]-'0';
                    idx=j;
                }
            }

            if(idx!=-1){
                swap(num[i],num[idx]);
                sort(num.begin()+i+1,num.end());
                break;
            }
        }

        long long req= stoll(num);
        if(req>val)
            return -1;
        if(req<=n)
            return -1;

        return (int)req;
    }
```

---

581. Shortest Unsorted Continuous Subarray

Given an integer array `nums`, you need to find one **continuous subarray** that if you only sort this subarray in ascending order, then the whole array will be sorted in ascending order.

Return *the shortest such subarray and output its length*.

Example 1:

```
Input: nums = [2,6,4,8,10,9,15]
Output: 5
Explanation: You need to sort [6, 4, 8, 10, 9] in ascending order to make the whole arr
```

Example 2:

```
Input: nums = [1,2,3,4]
Output: 0
```

Example 3:

```
Input: nums = [1]
Output: 0
```

Constraints:

- `1 <= nums.length <= 10^4`
- `-10^5 <= nums[i] <= 10^5`

Follow up: Can you solve it in `O(n)` time complexity?

---

important question to learn concepts solution ranges from n^3 --> n^2 --> n important property of sorted array used ==> for an sorted array if seg is [L,R] min(L,R) is >= max(1,L-1) && max(L,R)<= min(R+1,N)

```
int findUnsortedSubarray(vector<int>& nums) {

        int n= nums.size();

        int l=0,r=n-1;

        while(l+1<n && nums[l]<=nums[l+1])
            l++;

        if(l==n-1)
            return 0;

        while(r-1>=0 && nums[r]>=nums[r-1])
            r--;

        // at this point l and r are at candidate bounds of unosorted array

        int mx=INT_MIN,mn=INT_MAX;
        for(int i=l;i<=r;i++){
            mx= max(mx,nums[i]);
            mn= min(mn,nums[i]);
        }

        //cout<<mx<<" "<<mn<<'\n';
        l--;r++;
        while(l>=0 && nums[l]>mn)
            l--;
        while(r<n && nums[r]<mx)
            r++;

        return r-l-1;
    }
```

This solution has O(n) tc and constant space

---

692. Top K Frequent Words

Given a non-empty list of words, return the *k* most frequent elements.

Your answer should be sorted by frequency from highest to lowest. If two words have the same frequency, then the word with the lower alphabetical order comes first.

Example 1:

```
Input: ["i", "love", "leetcode", "i", "love", "coding"], k = 2
Output: ["i", "love"]
Explanation: "i" and "love" are the two most frequent words.
    Note that "i" comes before "love" due to a lower alphabetical order.
```

Example 2:

```
Input: ["the", "day", "is", "sunny", "the", "the", "the", "sunny", "is", "is"], k = 4
Output: ["the", "is", "sunny", "day"]
Explanation: "the", "is", "sunny" and "day" are the four most frequent words,
    with the number of occurrence being 4, 3, 2 and 1 respectively.
```

Note:

1. You may assume *k* is always valid, 1 ≤ *k* ≤ number of unique elements.
2. Input words contain only lowercase letters.

Follow up:

1. Try to solve it in *O*(*n* log *k*) time and *O*(*n*) extra space.

---

comparator in a priority queue

```
//   |  a  b | <-- vector

    //   |b| <-- in PQ
    //   |a|
struct compare{
        bool operator() (pair<int,string> &a, pair<int,string> &b)
        {
            if(a.first==b.first)
                return a.second < b.second;
            else
                return a.first > b.first;
        }
    };

    vector<string> topKFrequent(vector<string>& words, int k) {

        int n= words.size();
        unordered_map<string,int> mp;
        for(string s: words)
            mp[s]++;

        priority_queue<pair<int,string>, vector<pair<int,string>>, compare> pq;

        for(auto pt: mp){
            if(pq.size()<k)
                pq.push({pt.second,pt.first});

            else if(pt.second >= pq.top().first){
                if(pq.top().first== pt.second){
                    if(pt.first < pq.top().second){
                        pq.pop();
                        pq.push({pt.second,pt.first});
                    }
                }
                else{
                    pq.pop();
                    pq.push({pt.second,pt.first});
                }
            }
        }
        vector<string> res;
        while(!pq.empty()){
            res.push_back(pq.top().second);
            pq.pop();
        }
        reverse(res.begin(),res.end());
        return res;
    }
```

714. Best Time to Buy and Sell Stock with Transaction Fee

You are given an array `prices` where `prices[i]` is the price of a given stock on the `i`th day, and an integer `fee` representing a transaction fee.

Find the maximum profit you can achieve. You may complete as many transactions as you like, but you need to pay the transaction fee for each transaction.

Note: You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

Example 1:

```
Input: prices = [1,3,2,8,4,9], fee = 2
Output: 8
Explanation: The maximum profit can be achieved by:
- Buying at prices[0] = 1
- Selling at prices[3] = 8
- Buying at prices[4] = 4
- Selling at prices[5] = 9
The total profit is ((8 - 1) - 2) + ((9 - 4) - 2) = 8.
```

Example 2:

```
Input: prices = [1,3,7,5,10,3], fee = 3
Output: 6
```

Constraints:

- `1 <= prices.length <= 5 * 10`^{4}
- `1 <= prices[i] < 5 * 10`^{4}
- `0 <= fee < 5 * 10`^{4}

DP

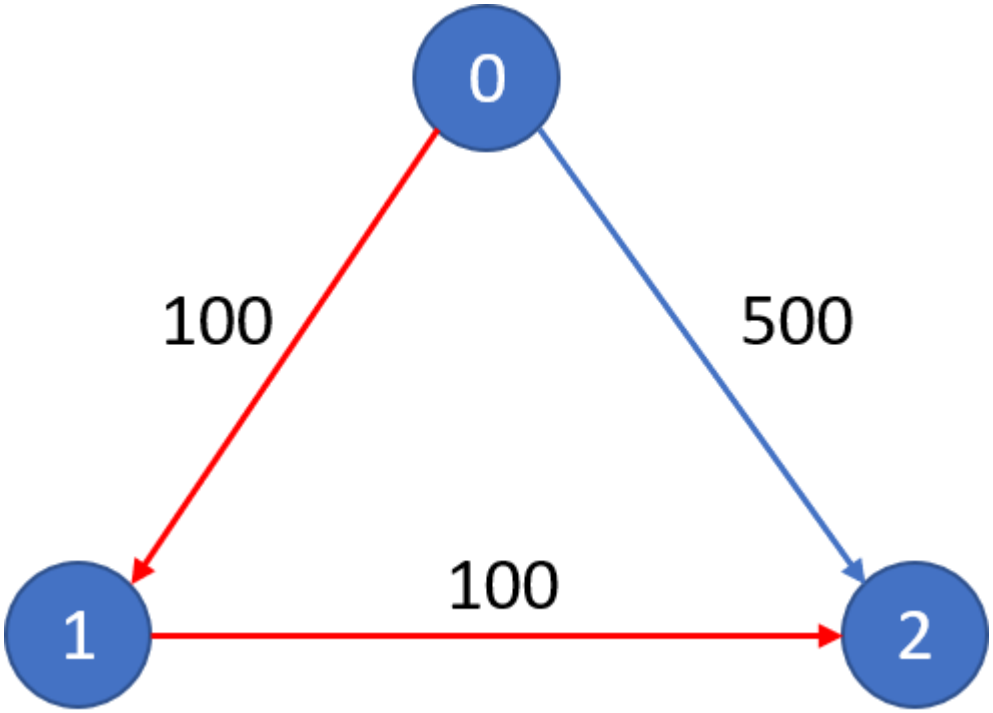
```
int n= prices.size();
       int dp[n][2];

       //dp[i][0]---> bought state i.e have one unsold stock in hand till i
       //dp[i][1]---> sold state i.e done with 0 or more complete transaction till i

       dp[0][0]=-prices[0] ,dp[0][1]=0;

       for(int i=1;i<n;i++){
           // buy stock
           int netcost= dp[i-1][1]+(-1*prices[i]);
           dp[i][0]= max(dp[i-1][0],netcost);
           //sell stock
           int nt= prices[i]+ dp[i-1][0]-fee;
           dp[i][1]= max(dp[i-1][1],nt);
       }

       return dp[n-1][1];
   }
```

787. Cheapest Flights Within K Stops

There are `n` cities connected by some number of flights. You are given an array `flights` where `flights[i] = [from_i, to_i, price_i]` indicates that there is a flight from city `from_i` to city `to_i` with cost `price_i`.

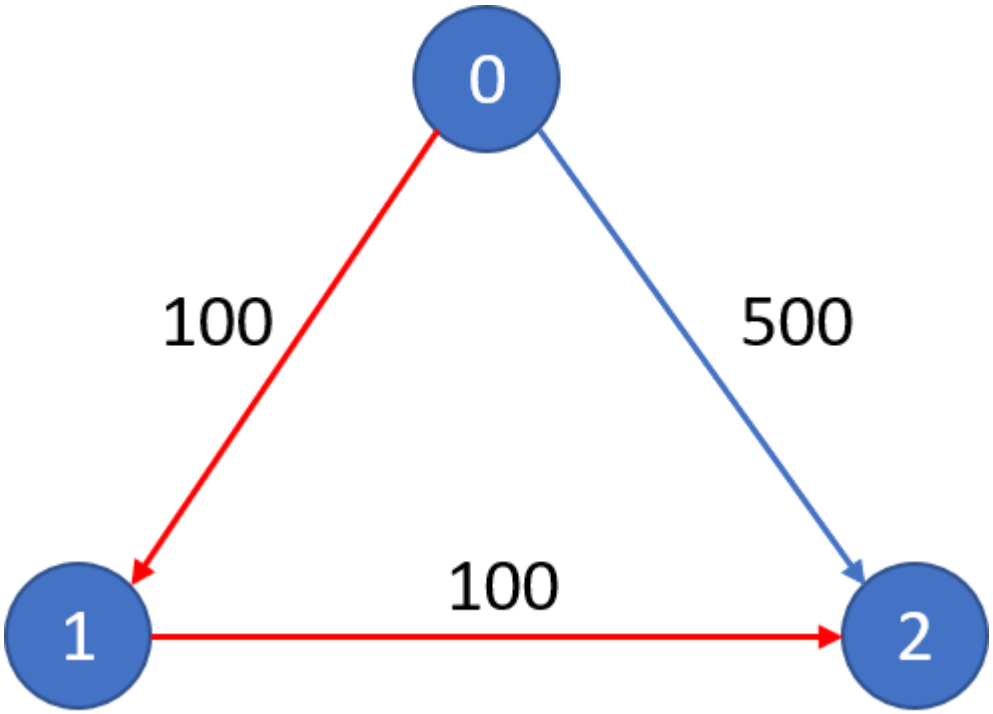
You are also given three integers `src`, `dst`, and `k`, return ***the cheapest price*** *from* `src` *to* `dst` *with at most* `k` *stops.* If there is no such route, return `-1`.

Example 1:



Input: n = 3, flights = [[0,1,100],[1,2,100],[0,2,500]], src = 0, dst = 2, k = 1
Output: 200
Explanation: The graph is shown.
The cheapest price from city 0 to city 2 with at most 1 stop costs 200, as marked red i

Example 2:



```
Input: n = 3, flights = [[0,1,100],[1,2,100],[0,2,500]], src = 0, dst = 2, k = 0
Output: 500
Explanation: The graph is shown.
The cheapest price from city 0 to city 2 with at most 0 stop costs 500, as marked blue
```

Constraints:

- `1 <= n <= 100`
- `0 <= flights.length <= (n * (n - 1) / 2)`
- `flights[i].length == 3`
- `0 <= from_i, to_i < n`
- `from_i != to_i`
- `1 <= price_i <= 10^4`
- There will not be any multiple flights between two cities.
- `0 <= src, dst, k < n`
- `src != dst`

Nice implementation on dijkstra

```
int findCheapestPrice(int n, vector<vector<int>>& flights, int src, int dst, int k) {

        //dijkstra
        int sz= flights.size();
        vector<array<int,2>> adj[n];
        for(int i=0;i<sz;i++)
            adj[flights[i][0]].push_back({flights[i][2],flights[i][1]});

        long long dis[n][k+2];
        for(int i=0;i<n;i++)
            for(int j=0;j<=k+1;j++)
                dis[i][j]=INT_MAX;

        priority_queue<array<long long,3>,vector<array<long long,3>>,greater<array<lo
ng long,3>>> pq;
        pq.push({0,0,src});
        dis[src][0]=0;
        while(!pq.empty()){
            auto u= pq.top();
            pq.pop();
            // cout<<u[2]<<" "<<u[1]<<" "<<u[0]<<endl;
            if(u[0] > dis[u[2]][u[1]])
                continue;

            if(u[1]>=k+1)
                continue;

            for(auto v: adj[u[2]]){
                //relax the edges
                if(dis[v[1]][u[1]+1] > v[0]+dis[u[2]][u[1]]){
                    dis[v[1]][u[1]+1] = v[0]+dis[u[2]][u[1]];
                    pq.push({dis[v[1]][u[1]+1],u[1]+1,v[1]});
                }
            }
        }

        long long ans=INT_MAX;
        for(int i=1;i<=k+1;i++)
            ans= min(ans,dis[dst][i]);
        return ans==INT_MAX?-1:ans;
    }
```

829. Consecutive Numbers Sum

Given an integer `n`, return *the number of ways you can write* `n` *as the sum of consecutive positive integers.*

Example 1:

```
Input: n = 5
Output: 2
Explanation: 5 = 2 + 3
```

Example 2:

```
Input: n = 9
Output: 3
Explanation: 9 = 4 + 5 = 2 + 3 + 4
```

Example 3:

```
Input: n = 15
Output: 4
Explanation: 15 = 8 + 7 = 4 + 5 + 6 = 1 + 2 + 3 + 4 + 5
```

Constraints:

- `1 <= n <= 10^9`

---

based on concept of divisors #tricky for sequence starting with a and n element sum will be using ap= n*(2a-n-1)/2 a/c n*(2a-n-1)= 2*N f1*f2= 2*N f1 and f2 are divsors of 2*N find different f1 and f2 pairs that will be the ans

```
int consecutiveNumbersSum(int n) {

        long long ans=0;
        long long N=n;
        N*=2;

        //f*(2*a+(f-1))= 2*n   a is a natural number
        //find factor f which satisfies the given condition
        for(int i=1;i<=sqrt(N);i++){
            if(N%i==0){
                long long f1= i;
                long long f2= N/i;
                f2-= (f1-1);
                // cout<<i<<" "<<f1<<" "<<f2<<'\n';
                if(f2>0 && f2%2==0)
                    ans++;


                if(i==N/i)
                    continue;
                //check the other way
                f1=N/i;
                f2= i;
                f2-= (f1-1);
                if(f2>0 && f2%2==0)
                    ans++;
            }
        }
        return ans;
    }
```

1024. Video Stitching

You are given a series of video clips from a sporting event that lasted `time` seconds. These video clips can be overlapping with each other and have varying lengths.

Each video clip is described by an array `clips` where `clips[i] = [start_i, end_i]` indicates that the ith clip started at `start_i` and ended at `end_i`.

We can cut these clips into segments freely.

- For example, a clip `[0, 7]` can be cut into segments `[0, 1] + [1, 3] + [3, 7]`.

Return *the minimum number of clips needed so that we can cut the clips into segments that cover the entire sporting event* `[0, time]`. If the task is impossible, return `-1`.

Example 1:

```
Input: clips = [[0,2],[4,6],[8,10],[1,9],[1,5],[5,9]], time = 10
Output: 3
Explanation:
We take the clips [0,2], [8,10], [1,9]; a total of 3 clips.
Then, we can reconstruct the sporting event as follows:
We cut [1,9] into segments [1,2] + [2,8] + [8,9].
Now we have segments [0,2] + [2,8] + [8,10] which cover the sporting event [0, 10].
```

Example 2:

```
Input: clips = [[0,1],[1,2]], time = 5
Output: -1
Explanation: We can't cover [0,5] with only [0,1] and [1,2].
```

Example 3:

```
Input: clips = [[0,1],[6,8],[0,2],[5,6],[0,4],[0,3],[6,7],[1,3],[4,7],[1,4],[2,5],[2,6]
Output: 3
Explanation: We can take clips [0,4], [4,7], and [6,9].
```

Example 4:

```
Input: clips = [[0,4],[2,8]], time = 5
Output: 2
Explanation: Notice you can have extra video after the event ends.
```

Constraints:

- `1 <= clips.length <= 100`
- `0 <= clips[i][0] <= clips[i][1] <= 100`
- `1 <= time <= 100`

THESE ARE IMPORTANT PROBLEMS ASKES IN INTERVIEWS The following problems has similar concepts:

1. Jump Game 2
2. Minimum number of taps to open to water the garden https://leetcode.com/problems/minimum-number-of-taps-to-open-to-water-a-garden/ (https://leetcode.com/problems/minimum-number-of-taps-to-open-to-water-a-garden/)
3. Video stiching

ALL the problems can be solves using greedy or dp

Greedy O(nlongn+ NT)

```
 int n= clips.size();
        int ans=0;
        int mxrange=0;
        int mnrange=0;
        sort(clips.begin(),clips.end());
        while(mxrange<T){
            int idx=0;
            while(idx<n && clips[idx][0]<=mnrange){
                mxrange= max(mxrange,clips[idx][1]);
                idx++;
            }

            if(mxrange==mnrange)
                return -1;
            ans++;
            mnrange=mxrange;
        }

        return ans;
```

DP solution (nlogn+NT)

```
sort(clips.begin(),clips.end());
        int n= clips.size();
        //at max we can use N clips
        vector<int> dp(101,101);
        dp[0]=0; // no clips required to have a total frame of len 0

        for(auto c: clips){
            int start= c[0];
            for(int i=1;i<=c[1];i++){
                dp[i]= min(dp[i],1+dp[c[0]]);
            }
        }
        return dp[T]>=10
```

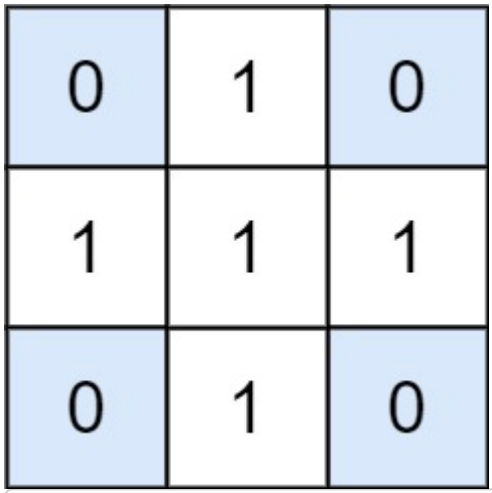
1074. Number of Submatrices That Sum to Target

Given a `matrix` and a `target`, return the number of non-empty submatrices that sum to `target`.

A submatrix `x1, y1, x2, y2` is the set of all cells `matrix[x][y]` with `x1 <= x <= x2` and `y1 <= y <= y2`.

Two submatrices `(x1, y1, x2, y2)` and `(x1', y1', x2', y2')` are different if they have some coordinate that is different: for example, if `x1 != x1'`.

Example 1:




```
Input: matrix = [[0,1,0],[1,1,1],[0,1,0]], target = 0
Output: 4
Explanation: The four 1x1 submatrices that only contain 0.
```

Example 2:

```
Input: matrix = [[1,-1],[-1,1]], target = 0
Output: 5
Explanation: The two 1x2 submatrices, plus the two 2x1 submatrices, plus the 2x2 submat
```

Example 3:

```
Input: matrix = [[904]], target = 0
Output: 0
```

Constraints:

- `1 <= matrix.length <= 100`
- `1 <= matrix[0].length <= 100`
- `-1000 <= matrix[i] <= 1000`
- `-10^8 <= target <= 10^8`

O(n*mn*m) solution using prefix sum in a matrix

```
//lets  try O(n*m*n*m) solution
       //using prefix sum

       int n= matrix.size();
       int m= matrix[0].size();

       ll dp[n+1][m+1];
       memset(dp,0,sizeof(dp));

       for(int i=1;i<=n;i++){
           for(int j=1;j<=m;j++){
               if(i==1 && j==1)
                   dp[i][j]= matrix[i-1][j-1];
               else if(i==1)
                   dp[i][j]= matrix[i-1][j-1]+dp[i][j-1];
               else if(j==1)
                   dp[i][j]= matrix[i-1][j-1]+dp[i-1][j];
               else{
                   dp[i][j]= dp[i-1][j]+dp[i][j-1]-dp[i-1][j-1]+matrix[i-1][j-1];
               }
           }
       }

       //now prefix is complete
       //check target sum
       int ans=0;
       for(int i=1;i<=n;i++){
           for(int j=1;j<=m;j++){
               for(int a=i;a<=n;a++){
                   for(int b=j;b<=m;b++){
                       ll sm=0;
                       sm= dp[a][b]-dp[i-1][b]-dp[a][j-1]+dp[i-1][j-1];
                       if(sm==target)
                           ans++;
                   }
               }
           }
       }
       return ans;
```

O(n*m*m) using hasing method

```
int numSubmatrixSumTarget(vector<vector<int>>& matrix, int target) {


    int n= matrix.size();
    int m= matrix[0].size();

    //using o(n*n*m) method using hasing
    for(int i=0;i<n;i++)
        for(int j=1;j<m;j++)
            matrix[i][j]+=matrix[i][j-1];

    int ans=0;
    unordered_map<int,int> mp;
    for(int i=0;i<n;i++){
        for(int j=i;j<n;j++){
            mp.clear();
            mp[0]=1;
            for(int k=0;k<m;k++){
                matrix[i][k]+= (i==j)?0:matrix[j][k];
                ans+= mp[matrix[i][k]-target];
                mp[matrix[i][k]]++;
            }
        }
    }
    return ans;
}
```

NOTE: Runtime of 1st solution is better than of second solution beacause practically 1st is taking less than than the upper bound

---

1105. Filling Bookcase Shelves

We have a sequence of `books` : the `i` -th book has thickness `books[i][0]` and height `books[i][1]` .

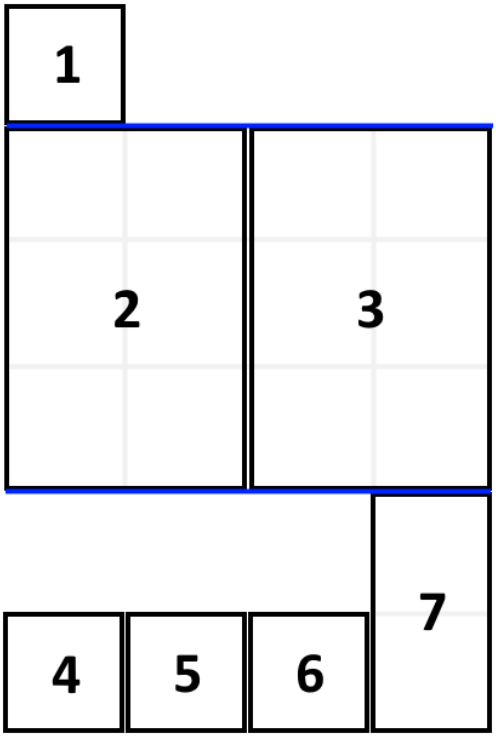
We want to place these books **in order** onto bookcase shelves that have total width `shelf_width` .

We choose some of the books to place on this shelf (such that the sum of their thickness is `<= shelf_width`), then build another level of shelf of the bookcase so that the total height of the bookcase has increased by the maximum height of the books we just put down. We repeat this process until there are no more books to place.

Note again that at each step of the above process, the order of the books we place is the same order as the given sequence of books. For example, if we have an ordered list of 5 books, we might place the first and second book onto the first shelf, the third book on the second shelf, and the fourth and fifth book on the last shelf.

Return the minimum possible height that the total bookshelf can be after placing shelves in this manner.

Example 1:




```
Input: books = [[1,1],[2,3],[2,3],[1,1],[1,1],[1,1],[1,2]], shelf_width = 4
Output: 6
Explanation:
The sum of the heights of the 3 shelves are 1 + 3 + 2 = 6.
Notice that book number 2 does not have to be on the first shelf.
```

Constraints:

- `1 <= books.length <= 1000`
- `1 <= books[i][0] <= shelf_width <= 1000`
- `1 <= books[i][1] <= 1000`

---

nice dp problem

```
int minHeightShelves(vector<vector<int>>& books, int shelf_width) {

        int n= books.size();
        int dp[n+1];
        dp[0]= 0;

        for(int i=1;i<=n;i++){
            int w= shelf_width;
            int mxh=0;
            int j=i;
            dp[i]=INT_MAX;
            while(j>0 && w-books[j-1][0]>=0){
                mxh= max(mxh,books[j-1][1]);
                w-=books[j-1][0];
                dp[i]= min(dp[i],mxh+dp[j-1]);
                j--;
            }
        }
        return dp[n];
    }
```

1130. Minimum Cost Tree From Leaf Values

Given an array `arr` of positive integers, consider all binary trees such that:

- Each node has either 0 or 2 children;
- The values of `arr` correspond to the values of each **leaf** in an in-order traversal of the tree. *(Recall that a node is a leaf if and only if it has 0 children.)*
- The value of each non-leaf node is equal to the product of the largest leaf value in its left and right subtree respectively.

Among all possible binary trees considered, return the smallest possible sum of the values of each non-leaf node. It is guaranteed this sum fits into a 32-bit integer.

Example 1:

```
Input: arr = [6,2,4]
Output: 32
Explanation:
There are two possible trees.  The first has non-leaf node sum 36, and the second has n

    24            24
   /  \          /  \
  12   4        6    8
 /  \               / \
6    2             2   4
```

Constraints:

- `2 <= arr.length <= 40`
- `1 <= arr[i] <= 15`
- It is guaranteed that the answer fits into a 32-bit signed integer (ie. it is less than `2^31`).

How to identify DP in this problem? ans: because ques asks for optimal binary tree which gives the solution as making all binary tree is not feasible therefore to find the optimal solution we have to use dp

which type of dp? ans: In this question we have to find solution for the best solution in [L,R] so this is dp on segments solved by filling the table in diagonal manner.

```
int n= arr.size();

        //we will solve it using dp
        int dp[n][n];
        //This dp is called dp on segments
        for(int j=0;j<n;j++){
            for(int i=0,k=j;i<n && k<n;i++,k++){
                if(k-i==0)
                    dp[i][k]=0;
                else if(k-i==1)
                    dp[i][k]=arr[i]*arr[k];
                else{
                    dp[i][k]=INT_MAX;
                    for(int p=i;p<k;p++){
                        //cal max in [i,p] ans [p+1][k];
                        // int mx1= *max_element(arr.begin()+i,arr.begin()+p+1);
                        // int mx2= *max_element(arr.begin()+p+1,arr.begin()+k+1);
                        int mx1= INT_MIN,mx2= INT_MIN;
                        for(int a=i;a<=p;a++)
                            mx1= max(mx1,arr[a]);
                        for(int a=p+1;a<=k;a++)
                            mx2= max(mx2,arr[a]);
                        // cout<<mx1<<" "<<mx2<<endl;
                        dp[i][k]= min(dp[i][k],dp[i][p]+dp[p+1][k]+ mx1*mx2);
                    }
                }
            }
        }
        // for(int i=0;i<n;i++){
        //     for(int j=0;j<n;j++){
        //         cout<<dp[i][j]<<" ";
        //     }
        //     cout<<endl;
        // }
        return dp[0][n-1];
    }
```

1248. Count Number of Nice Subarrays

Given an array of integers `nums` and an integer `k`. A continuous subarray is called **nice** if there are `k` odd numbers on it.

Return *the number of **nice** sub-arrays*.

Example 1:

```
Input: nums = [1,1,2,1,1], k = 3
Output: 2
Explanation: The only sub-arrays with 3 odd numbers are [1,1,2,1] and [1,2,1,1].
```

Example 2:

```
Input: nums = [2,4,6], k = 1
Output: 0
Explanation: There is no odd numbers in the array.
```

Example 3:

```
Input: nums = [2,2,2,1,2,2,1,2,2,2], k = 2
Output: 16
```

Constraints:

- `1 <= nums.length <= 50000`
- `1 <= nums[i] <= 10^5`
- `1 <= k <= nums.length`

This is interesting question has nice approaches

#approach 1: using hashing convert odds->1 and evens->0 find the number of subarrays having sum exacktly k

```
int numberOfSubarrays(vector<int>& nums, int k) {

        int n= nums.size();
        for(int i=0;i<n;i++){
            if(nums[i]%2==0)
                nums[i]=0;
            else
                nums[i]=1;
        }

        //now find number of subarray havins sum=k
        int ans=0;
        unordered_map<long long,int> mp;
        mp[0]=1;
        long long sm=0;
        for(int i=0;i<n;i++){
            sm+=nums[i];
            ans+= mp[sm-k];
            mp[sm]++;
        }
        return ans;
    }
```

approach 2: using sliding window use the relation exactly(k)= atmost(k) -atmost(k-1) number of subarrays with k odds= #arrays with k odds - #arrays with k-1 odds

```
int atmost(int k, vector<int> &nums)
    {
        int ct=0;
        int ans=0;
        int l=0,r=0;
        while(r<nums.size()){
            if(nums[r]%2==1)
                ct++;

            if(ct<=k){
                ans+= (r-l+1);
                r++;
            }
            else{
                while(l<=r && ct>k){
                    if(nums[l]%2==1)
                        ct--;
                    l++;
                }
                ans+=(r-l+1);
                r++;
            }
        }
        return ans;
    }
    int numberOfSubarrays(vector<int>& nums, int k) {

        int ans= atmost(k,nums)-atmost(k-1,nums);
        return ans;
    }
```

1326. Minimum Number of Taps to Open to Water a Garden

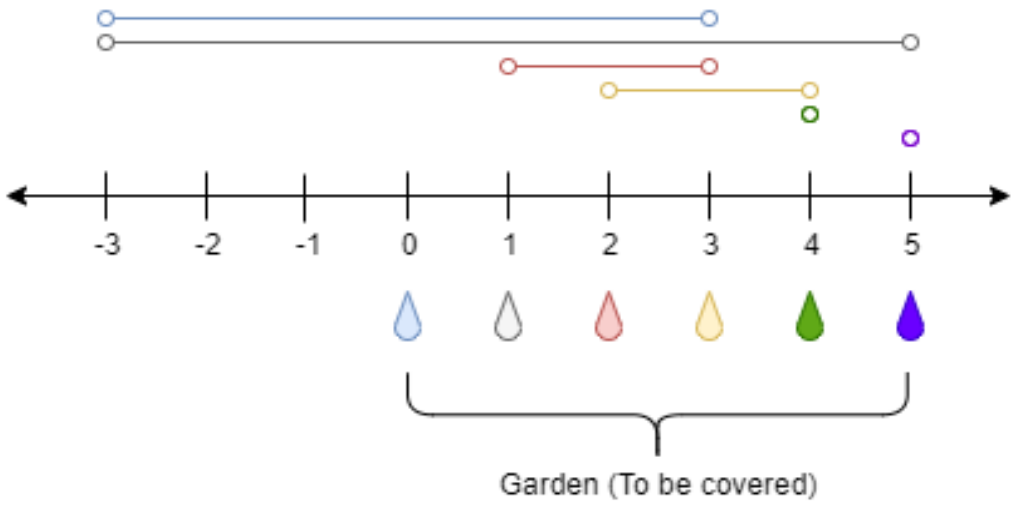
There is a one-dimensional garden on the x-axis. The garden starts at the point `0` and ends at the point `n`. (i.e The length of the garden is `n`).

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.
```

Example 3:

```
Input: n = 7, ranges = [1,2,1,0,2,1,0,1]
Output: 3
```

Example 4:

```
Input: n = 8, ranges = [4,0,0,0,0,0,0,0,4]
Output: 2
```

Example 5:

```
Input: n = 8, ranges = [4,0,0,0,4,0,0,0,4]
Output: 1
```

Constraints:

- `1 <= n <= 10^4`
- `ranges.length == n + 1`
- `0 <= ranges[i] <= 100`

This is the type of question that needed to remebered This is a little similer to jump game II where we use ladder stairs approach

only variation is along with maximizing right bound we have to ensure left bound is also maintained

```
//n^2 + nlogn solution
        vector<array<int,2>> vec;
        for(int i=0;i<=n;i++){
            vec.push_back({max(0,i-ranges[i]),min(n,i+ranges[i])});
        }

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

        int mxrange=0;
        int mnrange=0;
        int idx=0;
        int ans=0;
        while(idx<=n && mxrange<n){

            int sp;
            for(int i=idx;i<=n && vec[i][0]<=mnrange;i++){
                if(vec[i][1]>mxrange){
                    mxrange= vec[i][1];
                    sp=i;
                }
            }

            if(mxrange<=mnrange)
                return -1;

            mnrange= mxrange;
            idx=sp+1;
            ans++;
        }
        return ans;
```

ANOTHER APPROACH

one awesome trick to solve this problem is to convert the question in min jump array question the fountains can be converted into jumps for [L,R] is fountain range then the we can image it as jump from L to R and , ans is minimum number of jums also consider the case when the end is not reachable

```
    int sz= ranges.size();
    vector<int> jumps(n+1,0);

    for(int i=0;i<=n;i++){
        int l= max(0,i-ranges[i]);
        int r= min(n,i+ranges[i]);

        jumps[l]= max(jumps[l],r-l);
    }

    for(int i=0;i<=n;i++)
        cout<<jumps[i]<<" ";
    cout<<endl;
    //run the ladder stairs loop
    int ans=0;
    int mxrange=0;
    int idx=0;
    while(idx<n){
        mxrange= max(mxrange,idx+jumps[idx]);
        int steps= mxrange-idx;
        //cout<<mxrange<<" "<<idx<<endl;
        if(steps==0)
            return -1;
        else
            ans++;

        int tar=mxrange;
        while(idx<tar){
            mxrange= max(mxrange,idx+jumps[idx]);
            idx++;
        }
    }
    //cout<<mxrange<<endl;
    return ans;
}
```

PLEASE CRAMP THIS FOR OR RE ITERATE JUMP GAME II BECAUSE THIS IS TRICKY AND THERE IS POSSOBILITY THAT YOU MIGHT GET STRUCK AT LOOP

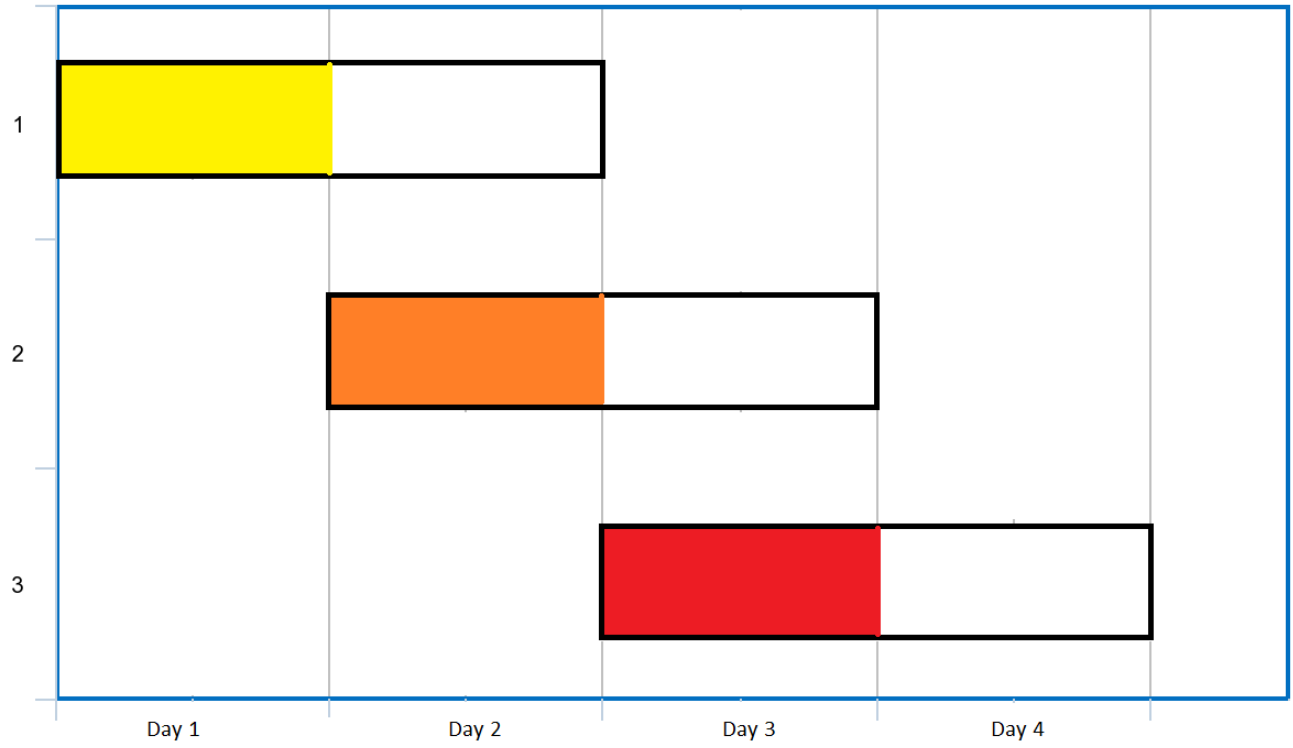
1353. Maximum Number of Events That Can Be Attended

Given an array of `events` where `events[i] = [startDay_i, endDay_i]`. Every event `i` starts at `startDay_i` and ends at `endDay_i`.

You can attend an event `i` at any day `d` where `startTime_i <= d <= endTime_i`. Notice that you can only attend one event at any time `d`.

Return *the maximum number of events* you can attend.

Example 1:




```
Input: events = [[1,2],[2,3],[3,4]]
Output: 3
Explanation: You can attend all the three events.
One way to attend them all is as shown.
Attend the first event on day 1.
Attend the second event on day 2.
Attend the third event on day 3.
```

Example 2:

```
Input: events= [[1,2],[2,3],[3,4],[1,2]]
Output: 4
```

Example 3:

```
Input: events = [[1,4],[4,4],[2,2],[3,4],[1,1]]
Output: 4
```

Example 4:

```
Input: events = [[1,100000]]
Output: 1
```

Example 5:

```
Input: events = [[1,1],[1,2],[1,3],[1,4],[1,5],[1,6],[1,7]]
Output: 7
```

Constraints:

- `1 <= events.length <= 10^5`
- `events[i].length == 2`
- `1 <= startDay_i <= endDay_i <= 10^5`

---

bdiya question hain scheduling ka approach 1 TLE greedy sort acc to end day fill events from 1st free spot from start

```
int maxEvents(vector<vector<int>>& events) {

        int n= events.size();
        vector<pair<int,int>> vec;
        for(int i=0;i<n;i++){
            vec.push_back({events[i][1],events[i][0]});
        }

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

        int ans=0;
        unordered_map<int,int> mp;

        for(int i=0;i<n;i++){
            for(int j= vec[i].second;j<=vec[i].first;j++){
                if(mp.find(j)==mp.end()){
                    mp[j]++;
                    ans++;
                    break;
                }
            }
        }
        return ans;
    }
```

approach 2: greedy + pq

1466. Reorder Routes to Make All Paths Lead to the City Zero

There are `n` cities numbered from `0` to `n - 1` and `n - 1` roads such that there is only one way to travel between two different cities (this network form a tree). Last year, The ministry of transport decided to orient the roads in one direction because they are too narrow.

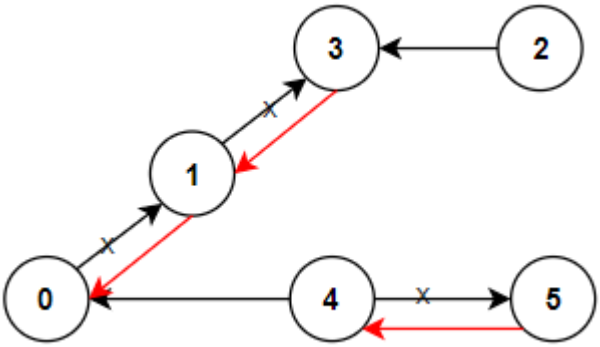
Roads are represented by `connections` where `connections[i] = [a_i, b_i]` represents a road from city `a_i` to city `b_i`.

This year, there will be a big event in the capital (city `0`), and many people want to travel to this city.

Your task consists of reorienting some roads such that each city can visit the city `0`. Return the **minimum** number of edges changed.

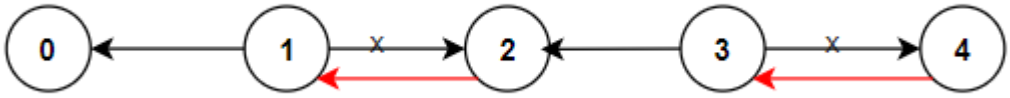
It's **guaranteed** that each city can reach city `0` after reorder.

Example 1:




```
Input: n = 6, connections = [[0,1],[1,3],[2,3],[4,0],[4,5]]
Output: 3
Explanation: Change the direction of edges show in red such that each node can reach th
```

Example 2:




```
Input: n = 5, connections = [[1,0],[1,2],[3,2],[3,4]]
Output: 2
Explanation: Change the direction of edges show in red such that each node can reach th
```

Example 3:

```
Input: n = 3, connections = [[1,0],[2,0]]
Output: 0
```

Constraints:

- $2 <= n <= 5 * 10^4$
- connections.length == n - 1
- connections[i].length == 2
- $0 <= a_i, b_i <= n - 1$
- $a_i != b_i$

** nice implementation on incomming and outgoing edges**

```
unordered_map<int,vector<int>> outgoing,incomming;
    bool vis[100001];
    int ans;
    void dfs(int u)
    {
        vis[u]=true;

        for(int v: outgoing[u])
            if(!vis[v]){
                ans++;
                dfs(v);
            }

        //also move for incomming edges
        for(int v: incomming[u])
            if(!vis[v])
                dfs(v);

        return;
    }

    int minReorder(int n, vector<vector<int>>& connections) {

        //see the graph as undirected graph
        //and check if an edge is outgoing or incomming
        //update ct

        int m= connections.size();
        outgoing.clear(),incomming.clear();

        for(int i=0;i<m;i++){
            outgoing[connections[i][0]].push_back(connections[i][1]);
            incomming[connections[i][1]].push_back(connections[i][0]);
        }

        ans=0;
        dfs(0);
        return ans;
    }
```

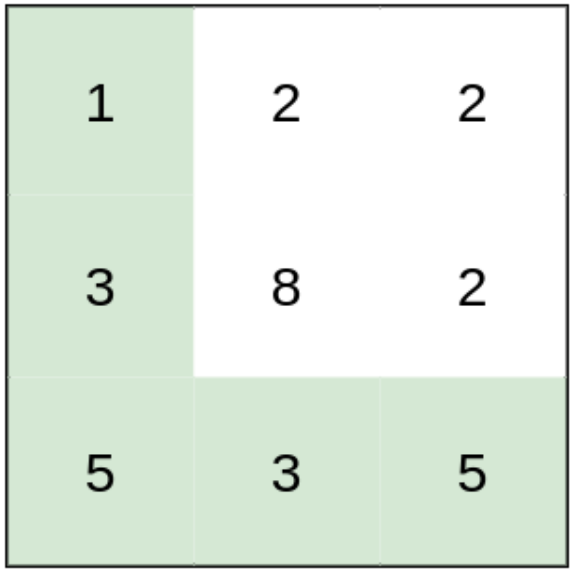
1631. Path With Minimum Effort

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.

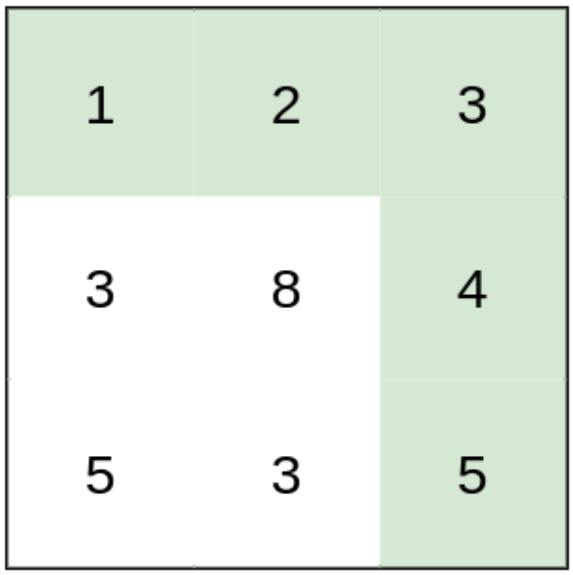
Return *the minimum **effort** required to travel from the top-left cell to the bottom-right cell.*

Example 1:



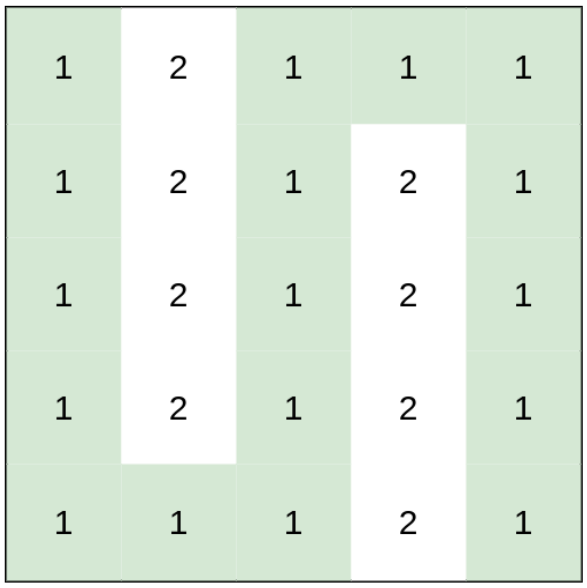

```
Input: heights = [[1,2,2],[3,8,2],[5,3,5]]
Output: 2
Explanation: The route of [1,3,5,3,5] has a maximum absolute difference of 2 in consecu
This is better than the route of [1,2,2,2,5], where the maximum absolute difference is
```

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 consecu
```

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] <= 10^6

---

nice implementation of dijkstra

```
int dir4[2][4]= {{0,0,1,-1},{1,-1,0,0}};
    bool check(int x, int y, int n, int m)
    {
        return x>=0 && x<n && y>=0 && y<m;
    }

    int minimumEffortPath(vector<vector<int>>& heights) {

        int n= heights.size();
        int m= heights[0].size();
        //using priority_queue
        vector<vector<int>> dp(n,vector<int>(m,INT_MAX));

        priority_queue<array<int,3>,vector<array<int,3>>,greater<array<int,3>>> pq;
        pq.push({0,0,0});
        dp[0][0]=0;
        while(!pq.empty())
        {
            auto u= pq.top();
            pq.pop();

            int currans=u[0],x=u[1],y=u[2];

            if(currans> dp[x][y])
                continue;

            for(int i=0;i<4;i++){
                int xx= x+dir4[0][i];
                int yy= y+dir4[1][i];

                if(check(xx,yy,n,m)){
                    //relax xx,yy
                    if(max(currans,abs(heights[xx][yy]-heights[x][y]))<dp[xx][yy]){
                        dp[xx][yy]= max(currans,abs(heights[xx][yy]-heights[x][y]));
                        pq.push({dp[xx][yy],xx,yy});
                    }
                }
            }
        }
        return dp[n-1][m-1];
```

This problem can also be solved using Binary search as we have minimize the maximum someting

```
private int[] d = {0, 1, 0, -1, 0};
    public int minimumEffortPath(int[][] heights) {
        int lo = 0, hi = 1_000_000;
        while (lo < hi) {
            int effort = lo + (hi - lo) / 2;
            if (isPath(heights, effort)) {
                hi = effort;
            }else {
                lo = effort + 1;
            }
        }
        return lo;
    }
    private boolean isPath(int[][] h, int effort) {
        int m = h.length, n = h[0].length;
        Queue<int[]> q = new LinkedList<>();
        q.offer(new int[2]);
        Set<Integer> seen = new HashSet<>();
        seen.add(0);
        while (!q.isEmpty()) {
            int[] cur = q.poll();
            int x = cur[0], y = cur[1];
            if (x == m - 1 && y == n - 1) {
                return true;
            }
            for (int k = 0; k < 4; ++k) {
                int r = x + d[k], c = y + d[k + 1];
                if (0 <= r && r < m && 0 <= c && c < n && effort >= Math.abs(h[r][c]
- h[x][y]) && seen.add(r * n + c)) {
                    q.offer(new int[]{r, c});
                }
            }
        }
        return false;
    }
```