32. Longest Valid Parentheses

Given a string containing just the characters `'('` and `')'`, find the length of the longest valid (well-formed) parentheses substring.

Example 1:

```
Input: s = "(()"
Output: 2
Explanation: The longest valid parentheses substring is "()".
```

Example 2:

```
Input: s = ")()())"
Output: 4
Explanation: The longest valid parentheses substring is "()()".
```

Example 3:

```
Input: s = ""
Output: 0
```

Constraints:

- `0 <= s.length <= 3 * ` 10^4
- `s[i]` is `'('`, or `')'`.

Code with linear dp approach :

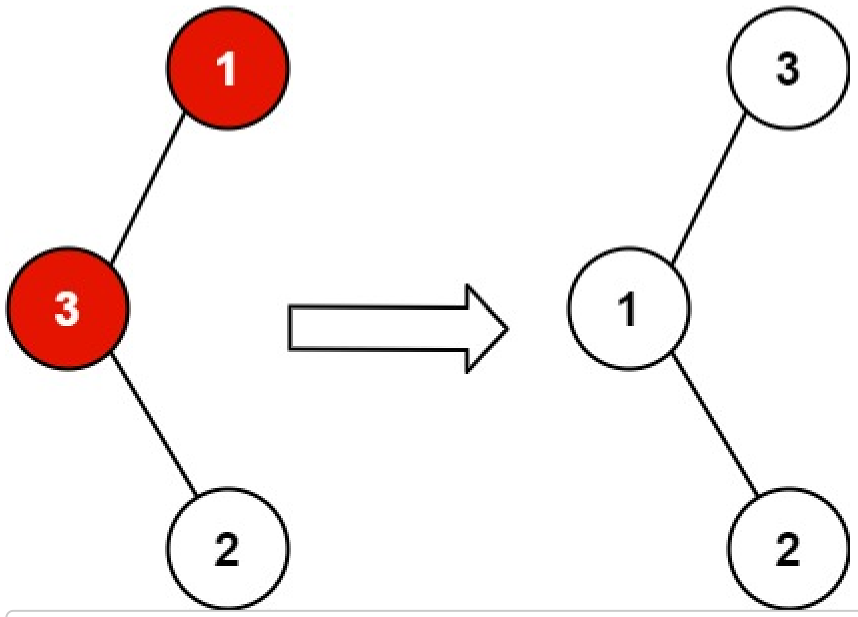
```
class Solution {
public:
    int longestValidParentheses(string s) {
        int n=s.size();
        vector<int> dp(n,0);
        int ans=0;

        for(int i=0;i<s.size();i++){
            if(s[i]=='(')
                continue;
            else if(i-1>=0 && i-dp[i-1]-1>=0 && s[i-dp[i-1]-1]=='(')
                {
                    dp[i]=dp[i-1]+2;
                    if(i-dp[i-1]-2>=0)
                        dp[i]+=dp[i-dp[i-1]-2];
                    ans=max(ans,dp[i]);
                }

        }
        return ans;
    }
};
```

99. Recover Binary Search Tree

You are given the `root` of a binary search tree (BST), where exactly two nodes of the tree were swapped by mistake. *Recover the tree without changing its structure*.

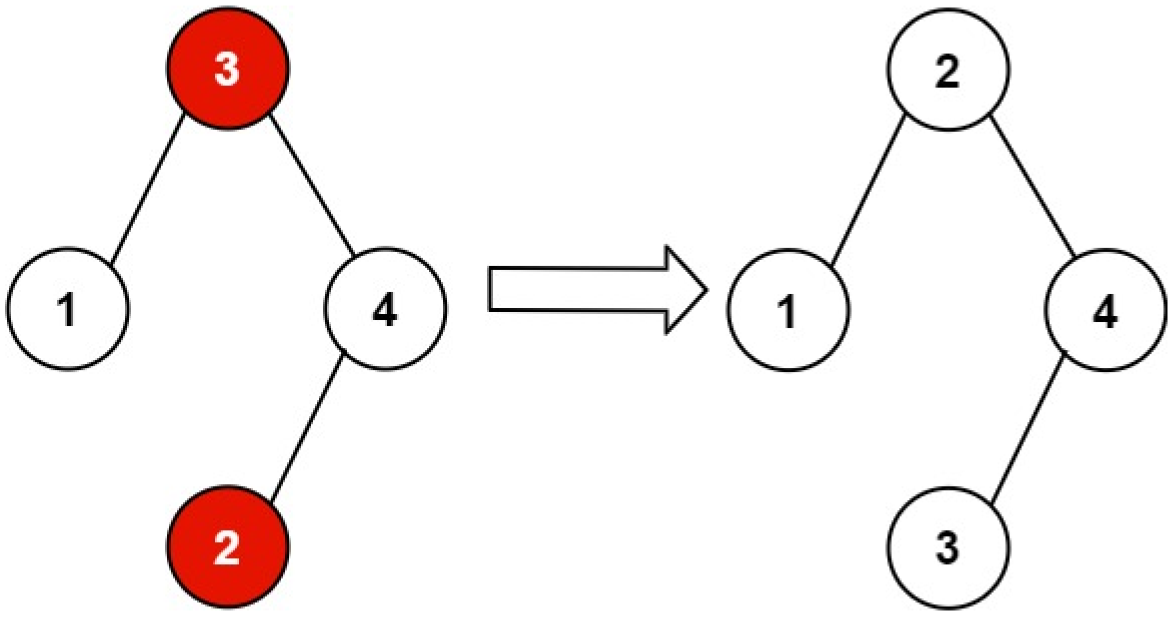
Follow up: A solution using `O(n)` space is pretty straight forward. Could you devise a constant space solution?

Example 1:



```
Input: root = [1,3,null,null,2]
Output: [3,1,null,null,2]
Explanation: 3 cannot be a left child of 1 because 3 > 1. Swapping 1 and 3 makes the BS
```

Example 2:



```
Input: root = [3,1,4,null,null,2]
Output: [2,1,4,null,null,3]
Explanation: 2 cannot be in the right subtree of 3 because 2 < 3. Swapping 2 and 3 make
```

Constraints:

- The number of nodes in the tree is in the range `[2, 1000]`.
- -2^{31} `<= Node.val <=` 2^{31} `- 1`

Code using inorder :

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(r
ight) {}
 * };
 */
class Solution {
public:
    //passing by reference is important
    void inorder(TreeNode* root,TreeNode*& mx,TreeNode*& mn,bool& f)
    {
        if(!root)
            return;

        inorder(root->left,mx,mn,f);

        if(!mx)
            mx = root;
        else if(root->val > mx->val && f==false)
            mx = root;
        else if(root->val < mx->val)
        {
            mn = root;
            f = true;
        }

        inorder(root->right,mx,mn,f);
    }
    void recoverTree(TreeNode* root) {
        TreeNode* mx = NULL,*mn = NULL;
        bool f = false;
        inorder(root,mx,mn,f);

        swap(mx->val,mn->val);
    }
};
```

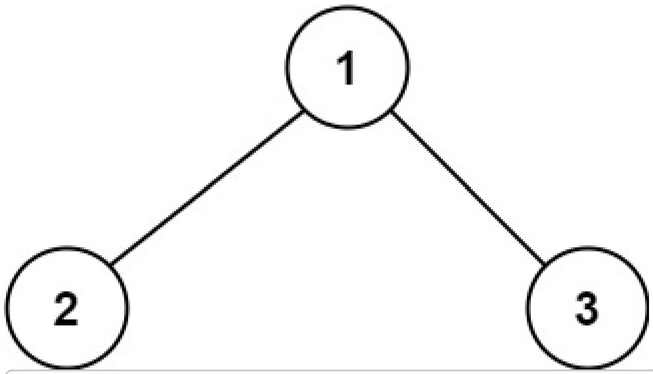
Note : O(1) space and O(n) Time complexity

124. Binary Tree Maximum Path Sum

Given a **non-empty** binary tree, find the maximum path sum.

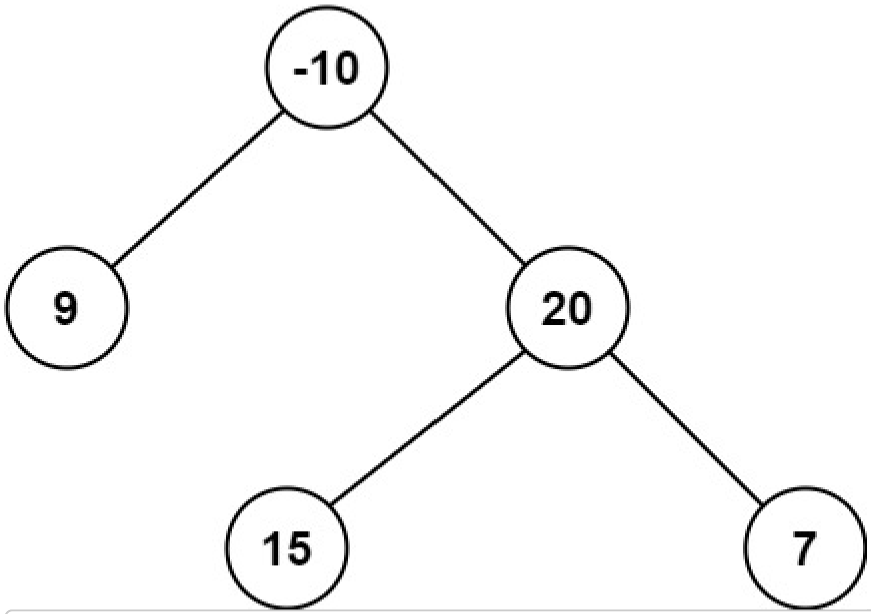
For this problem, a path is defined as any node sequence from some starting node to any node in the tree along the parent-child connections. The path must contain **at least one node** and does not need to go through the root.

Example 1:



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

Example 2:



```
Input: root = [-10,9,20,null,null,15,7]
Output: 42
```

Constraints:

- The number of nodes in the tree is in the range `[0, 3 * 10^4]`.
- `-1000 <= Node.val <= 1000`

Code using dfs :

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(r
ight) {}
 * };
 */
class Solution {
public:
    int res = INT_MIN;
    int dfs(TreeNode* root)
    {
        if(!root)
            return 0;

        int p = max(0,dfs(root->left));
        int q = max(0,dfs(root->right));

        res = max(res,p+q+root->val);
        return max(p,q) + root->val;
    }
    int maxPathSum(TreeNode* root) {
        dfs(root);
        return res;

    }
};
```

134. Gas Station

There are *N* gas stations along a circular route, where the amount of gas at station *i* is `gas[i]`.

You have a car with an unlimited gas tank and it costs `cost[i]` of gas to travel from station *i* to its next station (*i*+1). You begin the journey with an empty tank at one of the gas stations.

Return the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return -1.

Note:

- If there exists a solution, it is guaranteed to be unique.
- Both input arrays are non-empty and have the same length.
- Each element in the input arrays is a non-negative integer.

Example 1:

```
Input:
gas  = [1,2,3,4,5]
cost = [3,4,5,1,2]

Output: 3

Explanation:
Start at station 3 (index 3) and fill up with 4 unit of gas. Your tank = 0 + 4 = 4
Travel to station 4. Your tank = 4 - 1 + 5 = 8
Travel to station 0. Your tank = 8 - 2 + 1 = 7
Travel to station 1. Your tank = 7 - 3 + 2 = 6
Travel to station 2. Your tank = 6 - 4 + 3 = 5
Travel to station 3. The cost is 5. Your gas is just enough to travel back to station 3
Therefore, return 3 as the starting index.
```

Example 2:

```
Input:
gas  = [2,3,4]
cost = [3,4,3]

Output: -1

Explanation:
You can't start at station 0 or 1, as there is not enough gas to travel to the next sta
Let's start at station 2 and fill up with 4 unit of gas. Your tank = 0 + 4 = 4
Travel to station 0. Your tank = 4 - 3 + 2 = 3
Travel to station 1. Your tank = 3 - 3 + 3 = 3
You cannot travel back to station 2, as it requires 4 unit of gas but you only have 3.
Therefore, you can't travel around the circuit once no matter where you start.
```

Code using brute force :

```
class Solution {
public:
    int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
        int n = gas.size();

        for(int i=0;i<n;i++)
        {
            int g = gas[i]-cost[i];  //Gas remaining at (i+1)'th index
            int j=i+1;               //reached at j'th station with gas = g
            while(j<=n)
            {
                if(g<0)              //Check if gas was finished in the way
                    break;
                if(j==n)             //Make it circular path
                    j=0;
                if(j==i)             //If reached the starting station return it
                    return i;
                g = (g+gas[j]) - cost[j];  //Update the remaining gas
                j++;
            }
        }
        return -1;              //If no possible circular route
    }
};
```

Optimized code O(N) :

```
class Solution {
public:
    int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
       int n=gas.size();
        int prev=0;
        int restoreGas=0;
        int start=0;
        for(int i=0;i<n;i++)
        {
            restoreGas+=gas[i]-cost[i];
            if(restoreGas<0)
            {
                prev+=restoreGas;
                start=i+1;
                restoreGas=0;
            }
        }
        return prev+restoreGas>=0?start:-1;
    }
};
```

209. Minimum Size Subarray Sum

Given an array of **n** positive integers and a positive integer **s**, find the minimal length of a **contiguous** subarray of which the sum ≥ **s**. If there isn't one, return 0 instead.

Example:

```
Input: s = 7, nums = [2,3,1,2,4,3]
Output: 2
Explanation: the subarray [4,3] has the minimal length under the problem constraint.
```

Follow up:
If you have figured out the *O*(*n*) solution, try coding another solution of which the time complexity is *O*(*n* log *n*).

Code with sliding window approach :

```
class Solution {
public:
    int minSubArrayLen(int s, vector<int>& nums) {
        int i=0,j=0,n=nums.size();
        long long int sum=0;
        int res=INT_MAX;
        while(j<n)
        {
            sum += nums[j];
            if(sum>=s)
            {
               while(sum>=s)
               {
                   sum -= nums[i];
                    i++;
               }
                res = min(res,j-i+2);
            }
            j++;
        }
        if(res == INT_MAX)
            return 0;
        else
            return res;

    }
};
```

code with binary search approach:

Note : Binary search is applied on length of subarray.

```
class Solution {
public:
    //Using sliding window to find out if subarray of a particular length has sum gre
ater than or equal to the target.

    bool check(vector<int>& v,int len,int s)
    {
        int i=0,j=len;
        while(j<v.size())
        {
            if(v[j]-v[i] >= s)
                return true;
            j++;
            i++;
        }
        return false;
    }
    int minSubArrayLen(int s, vector<int>& nums) {
        int n = nums.size();
        vector<int> v(n+1,0);          //Array to store the cumulative sum;

        for(int i=1;i<=n;i++)
            v[i] = v[i-1] + nums[i-1];

        int low=1,high=n;
        while(low<high)
        {
            int len = (low+high)/2;
            if(check(v,len,s))
                high=len;
            else
                low=len+1;
        }

        //when low is equal to high check for the highest length also, if that also r
eturns false then there is no subarray with sum >= s.

        if(low==high && check(v,high,s)==false)
            return 0;
        return high;
    }
};
```

215. Kth Largest Element in an Array

Find the **k**th largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the kth distinct element.

Example 1:

```
Input: [3,2,1,5,6,4] and k = 2
Output: 5
```

Example 2:

```
Input: [3,2,3,1,2,4,5,5,6] and k = 4
Output: 4
```

Note:
You may assume k is always valid, 1 ≤ k ≤ array's length.

Code using multiset :

```
//Time complexity : O(nlogn)

class Solution {
public:
    int findKthLargest(vector<int>& nums, int k) {
        multiset<int> s;
        for(int i:nums)
            s.insert(i);
        auto it = s.rbegin();
        while(--k)
            it++;
        return *it;
    }
};
```

239. Sliding Window Maximum

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

Return *the max sliding window*.

Example 1:

```
Input: nums = [1,3,-1,-3,5,3,6,7], k = 3
Output: [3,3,5,5,6,7]
Explanation:
Window position                Max
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

Example 2:

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

Example 3:

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

Example 4:

```
Input: nums = [9,11], k = 2
Output: [11]
```

Example 5:

```
Input: nums = [4,-2], k = 2
Output: [4]
```

Constraints:

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

Code using deque :

Note :

Deque is helpful when we need to either insert or delete from both the ends of the queue

```
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        deque<int> q;  //stores the indices of max. values within the window in decre
asing order
        vector<int> res;
        int start = 0,end=0,n=nums.size();
        //insert the index with highest value in window from 0th to (k-2)th index.
        while(end<k-1)
        {
            while(!q.empty() && nums[q.back()]<nums[end])
                q.pop_back();

            q.push_back(end++);
        }
        //Now keep moving your window and push the front element in deque to your res
ult every time
        while(end<n)
        {

            while(!q.empty() && nums[q.back()]<nums[end])
                q.pop_back();

            q.push_back(end++);

            //For each window update the result vector with the front value of the qu
eue
            res.push_back(nums[q.front()]);

            //Indices stored in the deque should be maintained to be within the windo
w frame.
            if(q.front() == start++)
                q.pop_front();
        }
        return res;
    }
};
```

Code using multiset :

```
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        multiset<int> s;
        vector<int> res;
        int start = 0,end=0,n=nums.size();
        while(end<n)
        {
            s.insert(nums[end]);
            if(s.size()<k)
            {
                end++;
                continue;
            }
            else
            {

                res.push_back(*s.rbegin());
                s.erase(s.find(nums[start]));
                start++;
                end++;
            }
        }
        return res;
    }
};
```

310. Minimum Height Trees

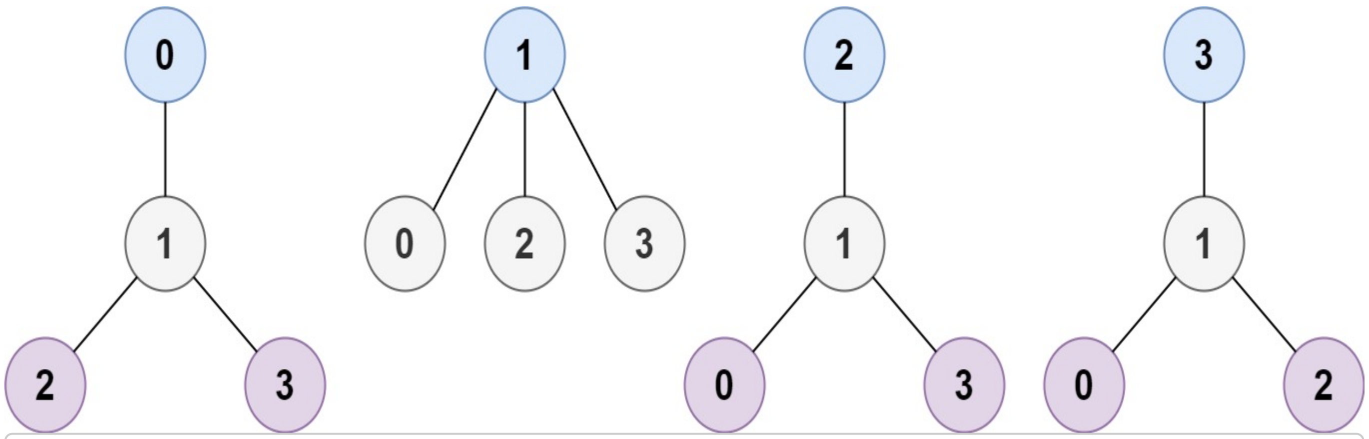
A tree is an undirected graph in which any two vertices are connected by *exactly* one path. In other words, any connected graph without simple cycles is a tree.

Given a tree of `n` nodes labelled from `0` to `n - 1`, and an array of `n - 1` `edges` where `edges[i] = [a_i, b_i]` indicates that there is an undirected edge between the two nodes a_i and b_i in the tree, you can choose any node of the tree as the root. When you select a node `x` as the root, the result tree has height `h`. Among all possible rooted trees, those with minimum height (i.e. `min(h)`) are called **minimum height trees** (MHTs).

Return *a list of all **MHTs'** root labels*. You can return the answer in **any order**.

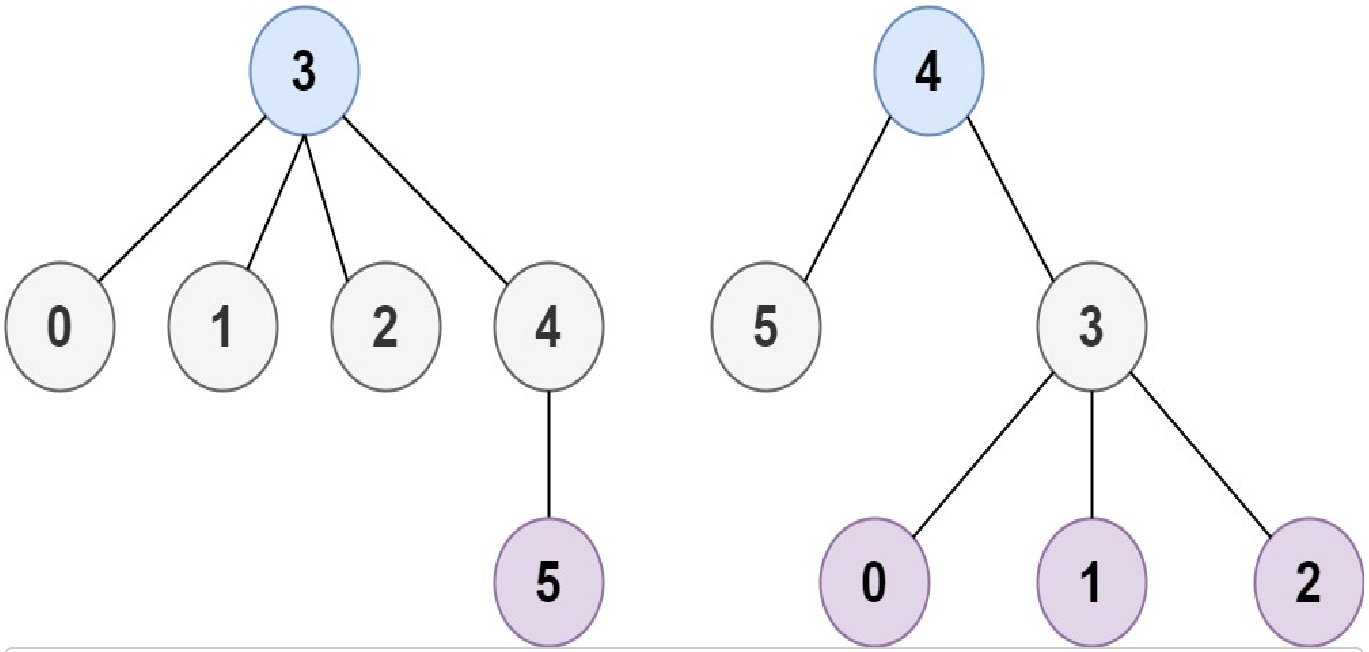
The **height** of a rooted tree is the number of edges on the longest downward path between the root and a leaf.

Example 1:



```
Input: n = 4, edges = [[1,0],[1,2],[1,3]]
Output: [1]
Explanation: As shown, the height of the tree is 1 when the root is the node with label
```

Example 2:



```
Input: n = 6, edges = [[3,0],[3,1],[3,2],[3,4],[5,4]]
Output: [3,4]
```

Example 3:

```
Input: n = 1, edges = []
Output: [0]
```

Example 4:

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

Constraints:

- `1 <= n <= 2 * 10`^{4}
- `edges.length == n - 1`
- `0 <=` a_i, b_i `< n`
- a_i `!=` b_i
- All the pairs (a_i, b_i) are distinct.
- The given input is **guaranteed** to be a tree and there will be **no repeated** edges.

Code using DFS + DP :

```
class Solution {
public:
    //dp[{i,j}] means if i reach j from i then waht is the maximum height that we get
    map<pair<int,int>,int> dp;
    int height(vector<vector<int>>& v,vector<bool>& vis,int i)
    {
        vis[i] = true;
        int temp = 0;
        for(int j=0;j<v[i].size();j++)
        {
            if(vis[v[i][j]] == false)
            {
                if(!(dp.count({i,j})>0))
                    dp[{i,j}] = 1 + height(v,vis,v[i][j]);

                temp = max(temp,dp[{i,j}]);
            }
        }
        return temp;
    }

    vector<int> findMinHeightTrees(int n, vector<vector<int>>& edges) {
        vector<vector<int>> v(n);
        for(int i=0;i<edges.size();i++)
        {
            v[edges[i][0]].push_back(edges[i][1]);
            v[edges[i][1]].push_back(edges[i][0]);
        }

        int min_h = INT_MAX;
        vector<int> res;
        for(int i=0;i<n;i++)
        {
            vector<bool> vis(n,false);
            int h = height(v,vis,i);

            if(h < min_h)
            {
                res.clear();
                min_h = h;
                res.push_back(i);
            }
            else if(h == min_h)
                res.push_back(i);

        }
        return res;
    }
};
```

424. Longest Repeating Character Replacement

Given a string `s` that consists of only uppercase English letters, you can perform at most `k` operations on that string.

In one operation, you can choose **any** character of the string and change it to any other uppercase English character.

Find the length of the longest sub-string containing all repeating letters you can get after performing the above operations.

Note:
Both the string's length and *k* will not exceed 10^4.

Example 1:

```
Input:
s = "ABAB", k = 2

Output:
4

Explanation:
Replace the two 'A's with two 'B's or vice versa.
```

Example 2:

```
Input:
s = "AABABBA", k = 1

Output:
4

Explanation:
Replace the one 'A' in the middle with 'B' and form "AABBBBA".
The substring "BBBB" has the longest repeating letters, which is 4.
```

Code with sliding window approach :

```
class Solution {
public:
    int characterReplacement(string s, int k) {
        int n = s.length();
        int start=0,end=0;
        int ans=0;
        int mx_char=0;  // stores the maximum occurence of characters in current wind
ow
        vector<int> v(26,0);
        while(end<n)
        {
            //check if the number of occurence of the current character is maximum in
the current window

            mx_char = max(mx_char,++v[s[end]-'A']);

            //if window size is greater than the length of character with max occuren
ce plus k, then contract the window
            while(mx_char + k < end-start+1)
                --v[s[start++]-'A'];

            ans = max(ans,end-start+1);
            end++;
        }
        return ans;
    }
};
```

Note :

- The idea is that if we are in a particular window, the maximum possible answer can be equal to the size of window if it is equal to the maximum occuring character plus k.

---

446. Arithmetic Slices II - Subsequence

A sequence of numbers is called arithmetic if it consists of at least three elements and if the difference between any two consecutive elements is the same.

For example, these are arithmetic sequences:

```
1, 3, 5, 7, 9
7, 7, 7, 7
3, -1, -5, -9
```

The following sequence is not arithmetic.

```
1, 1, 2, 5, 7
```

A zero-indexed array A consisting of N numbers is given. A **subsequence** slice of that array is any sequence of integers (P_0, P_1, ..., P_k) such that $0 \le P_0 < P_1 < ... < P_k < N$.

A **subsequence** slice (P_0, P_1, ..., P_k) of array A is called arithmetic if the sequence A[P_0], A[P_1], ..., A[P_{k-1}], A[P_k] is arithmetic. In particular, this means that $k \ge 2$.

The function should return the number of arithmetic subsequence slices in the array A.

The input contains N integers. Every integer is in the range of -2^{31} and $2^{31}-1$ and $0 \le N \le 1000$. The output is guaranteed to be less than $2^{31}-1$.

Example:

```
Input: [2, 4, 6, 8, 10]

Output: 7

Explanation:
All arithmetic subsequence slices are:
[2,4,6]
[4,6,8]
[6,8,10]
[2,4,6,8]
[4,6,8,10]
[2,4,6,8,10]
[2,6,10]
```

Here is the code :

```
class Solution {
public:
    int numberOfArithmeticSlices(vector<int>& A) {
        int n=A.size();
        int res=0;
        vector<unordered_map<long long,int>> dp(n);
        for(int i=1;i<n;i++)
        {
            for(int j=0;j<i;j++)
            {
                long long diff = (long long)A[i]-A[j];
                dp[i][diff]++;
                if(dp[j].count(diff))
                    dp[i][diff] += dp[j][diff];
                res += dp[j][diff];
            }
        }
        return res;
    }
};
```

My observations:

1. Since we need to generate all the possible sets(sequences), we need to look for DP solution.

1.

---

448. Find All Numbers Disappeared in an Array

Given an array of integers where 1 ≤ a[i] ≤ *n* (*n* = size of array), some elements appear twice and others appear once.

Find all the elements of [1, *n*] inclusive that do not appear in this array.

Could you do it without extra space and in O(*n*) runtime? You may assume the returned list does not count as extra space.

Example:

```
Input:
[4,3,2,7,8,2,3,1]

Output:
[5,6]
```

In-place code :

```
class Solution {
public:
    vector<int> findDisappearedNumbers(vector<int>& nums) {
        vector<int> res;
        int n=nums.size();

                //Update each position with new values if they are present, only thos
e positions will not be updated which do not correspond to any element in the array

        for(int i=0;i<n;i++)
        {
            int ind = (nums[i]%(n+1))-1;
            nums[ind] = nums[ind] + (n+1);
        }
                //Alternate for loop for above
                //for(int i=0;i<n;i++)
        //    nums[nums[i]%(n+1) - 1] += (n+1);

                //Those positions which are not updated add them to the res vector

        for(int i=0;i<n;i++)
            if(nums[i]<(n+1))
                res.push_back(i+1);
        return res;
    }
};
```

474. Ones and Zeroes

You are given an array of binary strings `strs` and two integers `m` and `n`.

Return *the size of the largest subset of* `strs` *such that there are* ***at most*** `m` `0`*'s and* `n` `1`*'s in the subset.*

A set `x` is a **subset** of a set `y` if all elements of `x` are also elements of `y`.

Example 1:

```
Input: strs = ["10","0001","111001","1","0"], m = 5, n = 3
Output: 4
Explanation: The largest subset with at most 5 0's and 3 1's is {"10", "0001", "1", "0"
Other valid but smaller subsets include {"0001", "1"} and {"10", "1", "0"}.
{"111001"} is an invalid subset because it contains 4 1's, greater than the maximum of
```

Example 2:

```
Input: strs = ["10","0","1"], m = 1, n = 1
Output: 2
Explanation: The largest subset is {"0", "1"}, so the answer is 2.
```

Constraints:

- `1 <= strs.length <= 600`
- `1 <= strs[i].length <= 100`
- `strs[i]` consists only of digits `'0'` and `'1'` .
- `1 <= m, n <= 100`

```
class Solution {
public:
    int findMaxForm(vector<string>& strs, int m, int n) {
        vector<vector<int>> dp(m+1,vector<int>(n+1,0));

        for(string s : strs)
        {
            int z = 0,o=0;
            for(int i=0;i<s.length();i++)
                if(s[i]=='0')
                    z++;
                else
                    o++;

            for(int i=m;i>=z;i--)
                for(int j=n;j>=o;j--)
                    dp[i][j] = max(dp[i][j],dp[i-z][j-o]+1);
        }
        return dp[m][n];
    }
};
```

516. Longest Palindromic Subsequence

Given a string s, find the longest palindromic subsequence's length in s. You may assume that the maximum length of s is 1000.

Example 1:
Input:

```
"bbbab"
```

Output:

```
4
```

One possible longest palindromic subsequence is "bbbb".

Example 2:
Input:

```
"cbbd"
```

Output:

```
2
```

One possible longest palindromic subsequence is "bb".

Constraints:

- `1 <= s.length <= 1000`
- `s` consists only of lowercase English letters.

Code:

```
class Solution {
public:
    int longestPalindromeSubseq(string s) {
        int n=s.length();
        vector<vector<int>> dp(n,vector<int>(n,0));
        
        for(int len=0;len<n;len++)
        {
            int j=0,k=len;
            while(j<n && k<n)
            {
                if(j==k)
                    dp[j][k]=1;
                else if(s[j]==s[k])
                    dp[j][k] = 2+dp[j+1][k-1];
                else
                    dp[j][k] = max(dp[j+1][k],dp[j][k-1]);
                j++;
                k++;
            }
        }
        return dp[0][n-1];
    }
};
```

Notes:

- We use tabulation method(Dynamic Programming).
- DP table is filled in diagonal fashion, which means we trying to find the optimal answer for smaller length substrings, and using them to find out the answer for larger subsequences contained within the substrings of larger lengths.
- if we are at j'th row and k'th column :
  Check:

  if string[j] == string[k]

  dp[j][k] = 2 + dp[j+1][k-1] (largest possible palindrome in the substring between j'th and k'th element)

  else

  dp[j][k] = max(dp[j][k-1],dp[j-1][k]) which means whatever was the largest

  palindrome possible for 1 smaller length contained in the substring starting from j to k.

673. Number of Longest Increasing Subsequence

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

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

Example 1:

```
Input: nums = [1,3,5,4,7]
Output: 2
Explanation: The two longest increasing subsequences are [1, 3, 4, 7] and [1, 3, 5, 7].
```

Example 2:

```
Input: nums = [2,2,2,2,2]
Output: 5
Explanation: The length of longest continuous increasing subsequence is 1, and there ar
```

Constraints:

- `1 <= nums.length <= 2000`
- -10^6 `<= nums[i] <=` 10^6

Code with dp approach :

```
class Solution {
public:
    int findNumberOfLIS(vector<int>& nums) {
        int n = nums.size(),ans = INT_MIN,res=0;
        if(n<2)
            return n;
        vector<int> dp(n,1), count(n,1);
        //Applying dp for the LIS ending at particular index
        for(int i=0;i<n;i++)
        {
            int mx = INT_MIN, ind = -1;
            for(int j=0;j<i;j++)
            {
                if(dp[j]>mx && nums[j]<nums[i])
                {
                    ind = j;
                    mx = dp[j];
                }
            }
            if(ind != -1)
                dp[i] += dp[ind];
            ans = max(ans,dp[i]);
        }

        //Applying dp for the number of times a particular length can be achieved
        for(int i=0;i<n;i++)
        {
            int sum = 0;
            for(int j=0;j<i;j++)
                if(dp[j] == dp[i]-1 && nums[j]<nums[i])
                    sum+=count[j];

            if(sum>0)
                count[i] = sum;
            if(dp[i] == ans)
                res+=count[i];
        }
        return res;
    }
};
```

904. Fruit Into Baskets

In a row of trees, the `i` -th tree produces fruit with type `tree[i]` .

You **start at any tree of your choice**, then repeatedly perform the following steps:

1. Add one piece of fruit from this tree to your baskets. If you cannot, stop.
2. Move to the next tree to the right of the current tree. If there is no tree to the right, stop.

Note that you do not have any choice after the initial choice of starting tree: you must perform step 1, then step 2, then back to step 1, then step 2, and so on until you stop.

You have two baskets, and each basket can carry any quantity of fruit, but you want each basket to only carry one type of fruit each.

What is the total amount of fruit you can collect with this procedure?

Example 1:

```
Input: [1,2,1]
Output: 3
Explanation: We can collect [1,2,1].
```

Example 2:

```
Input: [0,1,2,2]
Output: 3
Explanation: We can collect [1,2,2].
If we started at the first tree, we would only collect [0, 1].
```

Example 3:

```
Input: [1,2,3,2,2]
Output: 4
Explanation: We can collect [2,3,2,2].
If we started at the first tree, we would only collect [1, 2].
```

Example 4:

```
Input: [3,3,3,1,2,1,1,2,3,3,4]
Output: 5
Explanation: We can collect [1,2,1,1,2].
If we started at the first tree or the eighth tree, we would only collect 4 fruits.
```

Note:

1. `1 <= tree.length <= 40000`
2. `0 <= tree[i] < tree.length`

Code with sliding window approach :

```
class Solution {
public:
    int totalFruit(vector<int>& tree) {
        int n = tree.size();
        pair<int,int> f1 = {-1,0},f2 = {-1,0};  //first element stores the numbers un
der consideration... second element stores the number of times they appear in the cur
rent window.
        int start = 0, end = 0,ans=0;
        while(end<n)
        {
            //If the current element is one of the two under consideration update the
ir count.
            if(tree[end] == f1.first)
                f1.second++;
            else if(tree[end] == f2.first)
                f2.second++;
            else if(f1.first == -1)
            {
                f1.first = tree[end];
                f1.second++;
            }
            else if(f2.first == -1)
            {
                f2.first = tree[end];
                f2.second++;
            }
            //If the current element is neither of the two under consideration
            //if last element is equal to f1.first destroy the f2 to accomodate the n
ew element
            else if(f1.first == tree[end-1])
            {
                ans = max(ans,end-start);
                while(start<n && f2.second>0)
                {
                    if(tree[start] == f2.first)
                        f2.second--;
                    if(tree[start] == f1.first)
                        f1.second--;
                    start++;
                }
                f2.first = tree[end];
                f2.second=1;
            }
            //if last element is equal to f2.first destroy the f1 to accomodate the n
ew element
            else if(f2.first == tree[end-1])
            {
                ans = max(ans,end-start);
                while(start<n && f1.second>0)
                {
                    if(tree[start] == f1.first)
                        f1.second--;
                    if(tree[start] == f2.first)
                        f2.second--;
                    start++;
                }
                f1.first = tree[end];
                f1.second=1;
            }
            end++;
        }
        return max(ans,end-start);
    }
};
```

930. Binary Subarrays With Sum

In an array `A` of `0` s and `1` s, how many **non-empty** subarrays have sum `S` ?

Example 1:

```
Input: A = [1,0,1,0,1], S = 2
Output: 4
Explanation:
The 4 subarrays are bolded below:
[1,0,1,0,1]
[1,0,1,0,1]
[1,0,1,0,1]
[1,0,1,0,1]
```

Note:

1. `A.length <= 30000`
2. `0 <= S <= A.length`
3. `A[i]` is either `0` or `1` .

Code with sliding window approach :

```
class Solution {
public:
    int numSubarraysWithSum(vector<int>& A, int S) {
        int n=A.size();
        int start = 0,end = 0;
        int ans = 0,sum=0;
        //special case : when S = 0. Count the continuous number of 0s and add the to
tal number of possible subarrays to the answer.
        if(S==0)
        {
            while(end<n)
            {
                //count of continuous 0s
                sum = 0;
                while(end<n && A[end] == 0)
                {
                    sum++;
                    end++;
                }
                ans += (sum * (sum+1))/2;

                //skip all 1s
                while(end<n && A[end] == 1)
                    end++;
            }
            return ans;
        }

        while(end<n)
        {
            sum += A[end++];
            if(sum>S)
            {
                sum--;
                start++;
            }
            else if(sum == S)
            {
                int x = 0;  //Count of continuous 0's before 1st 1 in the window
                int y = 1;  //Count of continous 0's after the last 1 in the window
                while(start<end && sum == S)
                {
                    sum-=A[start];
                    x++;
                    start++;
                }
                while(end<n && A[end] == 0)
                {
                    y++;
                    end++;
                }
                //add the total number of subarrays possible in this window without e
xcluding S
                ans += x * y;
            }
        }
        return ans;
    }
};
```

948. Bag of Tokens

You have an initial **power** of `P`, an initial **score** of `0`, and a bag of `tokens` where `tokens[i]` is the value of the `i`th token (0-indexed).

Your goal is to maximize your total **score** by potentially playing each token in one of two ways:

- If your current **power** is at least `tokens[i]`, you may play the `i`th token face up, losing `tokens[i]` **power** and gaining `1` **score**.
- If your current **score** is at least `1`, you may play the `i`th token face down, gaining `tokens[i]` **power** and losing `1` **score**.

Each token may be played **at most** once and **in any order**. You do **not** have to play all the tokens.

Return *the largest possible* ***score*** *you can achieve after playing any number of tokens*.

Example 1:

```
Input: tokens = [100], P = 50
Output: 0
Explanation: Playing the only token in the bag is impossible because you either have to
```

Example 2:

```
Input: tokens = [100,200], P = 150
Output: 1
Explanation: Play the 0th token (100) face up, your power becomes 50 and score becomes 1
There is no need to play the 1st token since you cannot play it face up to add to your s
```

Example 3:

```
Input: tokens = [100,200,300,400], P = 200
Output: 2
Explanation: Play the tokens in this order to get a score of 2:
1. Play the 0th token (100) face up, your power becomes 100 and score becomes 1.
2. Play the 3rd token (400) face down, your power becomes 500 and score becomes 0.
3. Play the 1st token (200) face up, your power becomes 300 and score becomes 1.
4. Play the 2nd token (300) face up, your power becomes 0 and score becomes 2.
```

Constraints:

- `0 <= tokens.length <= 1000`
- `0 <= tokens[i], P <` 10^4

Code with greedy approach :

```
class Solution {
public:
    int bagOfTokensScore(vector<int>& v, int P) {
        int n = v.size();
        sort(v.begin(),v.end());

        int i=0,j=n-1,s=0;
        while(i<=j)
        {
                //untill current minimum token is playable keep playing
            if(P>=v[i])
            {
                P-=v[i];
                s++;
                i++;
            }
                        //if only less than 3 elements are left and we do not have su
fficient power left then break
            else if(j<=i+1)
                break;
                        //if more than 2 elements are there and score is at least 1 t
hen play face down with the maximum token
            else if(s>0)
            {
                P+=v[j];
                s--;
                j--;
            }
            else
                break;
        }
        return s;
    }
};
```

Note : We could use a variable to keep track of maximum till now

1004. Max Consecutive Ones III

Given an array `A` of 0s and 1s, we may change up to `K` values from 0 to 1.

Return the length of the longest (contiguous) subarray that contains only 1s.

Example 1:

```
Input: A = [1,1,1,0,0,0,1,1,1,1,0], K = 2
Output: 6
Explanation: 
[1,1,1,0,0,1,1,1,1,1,1]
Bolded numbers were flipped from 0 to 1.  The longest subarray is underlined.
```

Example 2:

```
Input: A = [0,0,1,1,0,0,1,1,1,0,1,1,0,0,0,1,1,1,1], K = 3
Output: 10
Explanation: 
[0,0,1,1,1,1,1,1,1,1,1,1,0,0,0,1,1,1,1]
Bolded numbers were flipped from 0 to 1.  The longest subarray is underlined.
```

Note:

1. `1 <= A.length <= 20000`
2. `0 <= K <= A.length`
3. `A[i]` is `0` or `1`

Code with sliding window approach :

```
class Solution {
public:
    int longestOnes(vector<int>& A, int k) {
        int n = A.size();
        int start = 0, end=0;   //start and end of the window
        int c=0;     //stores the number of zeroes in the currrent window
        int ans = 0;

        while(end<n)
        {
            if(A[end] == 0)
            {
                c++;
                //If the number of zeroes in the current window exceeds K contract th
e window
                while(start<n && c>k)
                {
                    if(A[start] == 0)
                        c--;
                    start++;
                }
            }
            ans = max(ans,end-start+1);
            end++;
        }
        return ans;
    }
};
```

1091. Shortest Path in Binary Matrix

In an N by N square grid, each cell is either empty (0) or blocked (1).

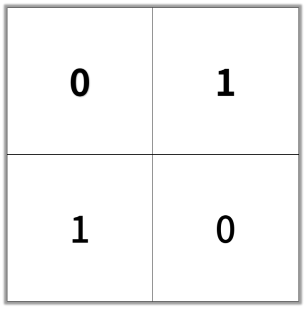
A *clear path from top-left to bottom-right* has length `k` if and only if it is composed of cells `C_1, C_2, ..., C_k` such that:

- Adjacent cells `C_i` and `C_{i+1}` are connected 8-directionally (ie., they are different and share an edge or corner)
- `C_1` is at location `(0, 0)` (ie. has value `grid[0][0]`)
- `C_k` is at location `(N-1, N-1)` (ie. has value `grid[N-1][N-1]`)
- If `C_i` is located at `(r, c)`, then `grid[r][c]` is empty (ie. `grid[r][c] == 0`).

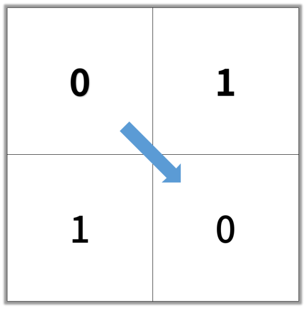
Return the length of the shortest such clear path from top-left to bottom-right. If such a path does not exist, return -1.

Example 1:

```
Input: [[0,1],[1,0]]
```

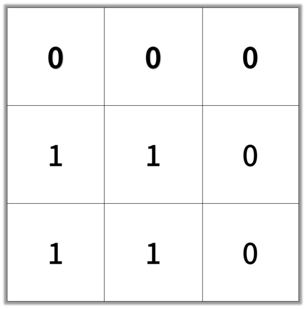


```
Output: 2
```

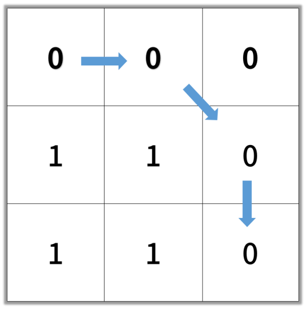


Example 2:

```
Input: [[0,0,0],[1,1,0],[1,1,0]]
```



```
Output: 4
```



Note:

1. `1 <= grid.length == grid[0].length <= 100`
2. `grid[r][c]` is `0` or `1`

Code using BFS :

```
class Solution {
public:

    int shortestPathBinaryMatrix(vector<vector<int>>& grid) {
        int n = grid.size();
        if(n==0)
            return 0;
        if(grid[0][0] == 1 || grid[n-1][n-1] == 1)
            return -1;
        vector<vector<bool>> vis(n,vector<bool>(n,false));

        queue<vector<int>> q;
        vis[0][0] = true;
        q.push({0,0});
        int ans = 0;
        while(!q.empty())
        {
            ans++;
            int qs = q.size();
            for(int x=0;x<qs;x++)
            {
                auto v = q.front();
                q.pop();

                if(v[0]==n-1 && v[1]==n-1)
                    return ans;
                for(int i=v[0]-1;i<=v[0]+1;i++)
                {
                    for(int j=v[1]-1;j<=v[1]+1;j++)
                    {
                        if(i<0 || j<0 || i>=n || j>=n || vis[i][j]==true || grid[i]
[j]==1)
                            continue;
                        vis[i][j] = true;
                        q.push({i,j});
                    }
                }
            }
        }
        return -1;
    }
};
```

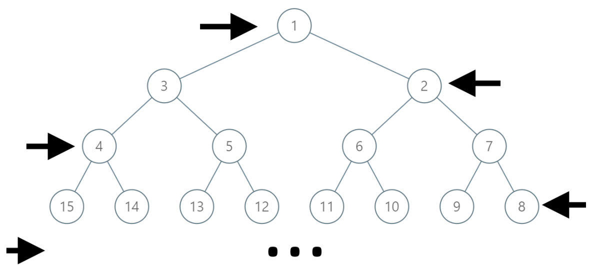
Note :

Whenever we need to find the shortest path from a source to destination, we use BFS.

1104. Path In Zigzag Labelled Binary Tree

In an infinite binary tree where every node has two children, the nodes are labelled in row order.

In the odd numbered rows (ie., the first, third, fifth,...), the labelling is left to right, while in the even numbered rows (second, fourth, sixth,...), the labelling is right to left.



Given the `label` of a node in this tree, return the labels in the path from the root of the tree to the node with that `label` .

Example 1:

```
Input: label = 14
Output: [1,3,4,14]
```

Example 2:

```
Input: label = 26
Output: [1,2,6,10,26]
```

Constraints:

- `1 <= label <= 10^6`

Code with recursive approach :

```
class Solution {
public:
    // l is the current label, h is the current height
    void route(vector<int>& res,int l,int h)
    {
        if(h < 1)
            return;
        res.insert(res.begin(),l);
        int d,pos;

        //If height is even
        if(h%2 == 0)
        {
            //find the distance of the label from left to right
            d = pow(2,h)-l;

            //find the position of the label from right to left i.e; pos
            pos = (pow(2,h-1) - d)/2 + 1;

            //subtract the value which was added to the ancestor to get the current l
abel to find its ancestor
            if(d%2 == 0)
                l = l-(3*pos - 2);
            else
                l = l-(3*pos - 1);

            //recursively call for its' ancestors
            route(res,l,h-1);
        }
        else               //If height is odd
        {
            d = l - pow(2,h-1) + 1;
            pos = (d+1)/2;
            if(d%2 == 0)
                l = l - (3*pos - 1);
            else
                l = l - (3*pos - 2);
            route(res,l,h-1);
        }
    }
    vector<int> pathInZigZagTree(int label) {
        vector<int> res;
        int h=0;      //finding out the height at which the label exists
        while(label >= pow(2,h))
            h++;
        route(res,label,h);
        return res;
    }
};
```

Note :

Time complexity : O(logn)

1627. Graph Connectivity With Threshold

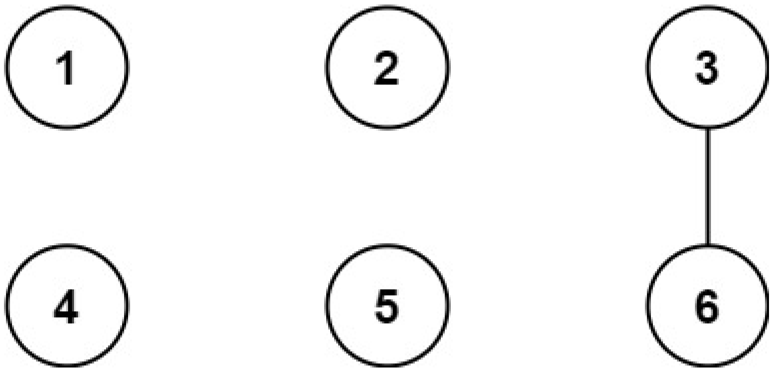
We have `n` cities labeled from `1` to `n`. Two different cities with labels `x` and `y` are directly connected by a bidirectional road if and only if `x` and `y` share a common divisor **strictly greater** than some `threshold`. More formally, cities with labels `x` and `y` have a road between them if there exists an integer `z` such that all of the following are true:

- `x % z == 0`,
- `y % z == 0`, and
- `z > threshold`.

Given the two integers, `n` and `threshold`, and an array of `queries`, you must determine for each `queries[i] = [`a_i`, `b_i`]` if cities a_i and b_i are connected (i.e. there is some path between them).

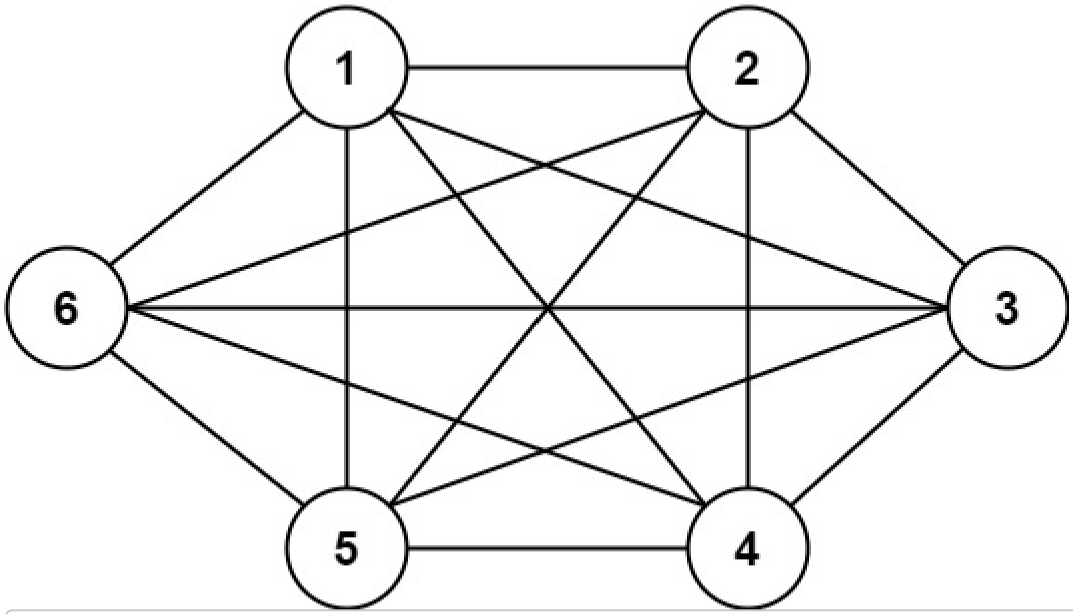
Return *an array* `answer`*, where* `answer.length == queries.length` *and* `answer[i]` *is* `true` *if for the* i^{th} *query, there is a path between* a_i *and* b_i*, or* `answer[i]` *is* `false` *if there is no path.*

Example 1:



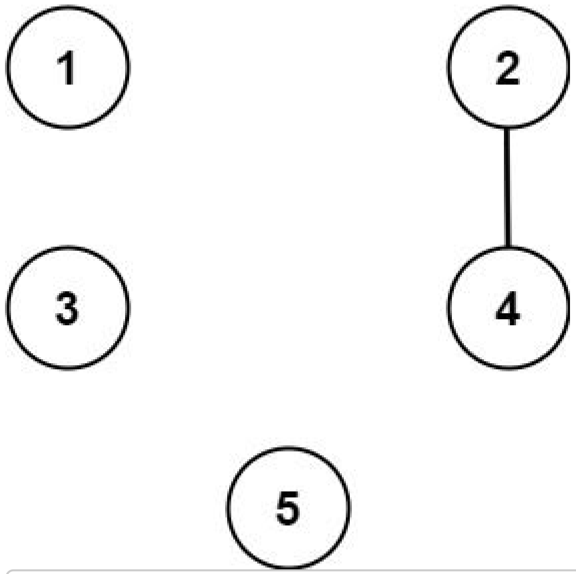
```
Input: n = 6, threshold = 2, queries = [[1,4],[2,5],[3,6]]
Output: [false,false,true]
Explanation: The divisors for each number:
1:   1
2:   1, 2
3:   1, 3
4:   1, 2, 4
5:   1, 5
6:   1, 2, 3, 6
Using the underlined divisors above the threshold, only cities 3 and 6 share a common d
only ones directly connected. The result of each query:
[1,4]   1 is not connected to 4
[2,5]   2 is not connected to 5
[3,6]   3 is connected to 6 through path 3--6
```

Example 2:



```
Input: n = 6, threshold = 0, queries = [[4,5],[3,4],[3,2],[2,6],[1,3]]
Output: [true,true,true,true,true]
Explanation: The divisors for each number are the same as the previous example. However
all divisors can be used. Since all numbers share 1 as a divisor, all cities are connec
```

Example 3:



```
Input: n = 5, threshold = 1, queries = [[4,5],[4,5],[3,2],[2,3],[3,4]]
Output: [false,false,false,false,false]
Explanation: Only cities 2 and 4 share a common divisor 2 which is strictly greater tha
Please notice that there can be multiple queries for the same pair of nodes [x, y], and
```

Constraints:

- `2 <= n <= 10`^{4}
- `0 <= threshold <= n`
- `1 <= queries.length <= 10`^{5}
- `queries[i].length == 2`
- `1 <=` a_i, b_i `<= cities`
- a_i `!=` b_i

Code using Union-Find approach :

```
class Solution {
public:
    int _find(vector<int>& par,int x)
    {
        if(par[x] != -1)
            return _find(par,par[x]);
        return x;
    }
    vector<bool> areConnected(int n, int t, vector<vector<int>>& q) {
        int m = q.size();
        vector<bool> res(m,true);
        if(t == 0)
            return res;
        vector<int> par(10005,-1);

        for(int i=t+1;i<=n;i++)
        {
            int j = i;
            //all multiples of any number should have same parent
            while(j+i <= n)
            {
                int x = _find(par,j);
                int y = _find(par,j+i);
                if(x != y)
                    par[y] = x;
                j+=i;
            }
        }
        //If parents are same they are connected directly or indirectly
        for(int i=0;i<m;i++)
            res[i] = (_find(par,q[i][0]) == _find(par,q[i][1]));
        return res;
    }
};
```

1208. Get Equal Substrings Within Budget

You are given two strings `s` and `t` of the same length. You want to change `s` to `t`. Changing the `i`-th character of `s` to `i`-th character of `t` costs `|s[i] - t[i]|` that is, the absolute difference between the ASCII values of the characters.

You are also given an integer `maxCost`.

Return the maximum length of a substring of `s` that can be changed to be the same as the corresponding substring of `t` with a cost less than or equal to `maxCost`.

If there is no substring from `s` that can be changed to its corresponding substring from `t`, return `0`.

Example 1:

```
Input: s = "abcd", t = "bcdf", maxCost = 3
Output: 3
Explanation: "abc" of s can change to "bcd". That costs 3, so the maximum length is 3.
```

Example 2:

```
Input: s = "abcd", t = "cdef", maxCost = 3
Output: 1
Explanation: Each character in s costs 2 to change to charactor in t, so the maximum
length is 1.
```

Example 3:

```
Input: s = "abcd", t = "acde", maxCost = 0
Output: 1
Explanation: You can't make any change, so the maximum length is 1.
```

Constraints:

- `1 <= s.length, t.length <= 10^5`
- `0 <= maxCost <= 10^6`
- `s` and `t` only contain lower case English letters.

Code with sliding window approach :

```
class Solution {
public:
    int equalSubstring(string s, string t, int maxCost) {
        int ans=0;
        int start = 0, end = 0 , cost = 0;

        while(end<s.length())
        {
            cost += abs(s[end]-t[end]);
            while(start<s.length() && cost>maxCost)
            {
                cost -= abs(s[start] - t[start]);
                start++;
            }
            end++;
            ans = max(ans,end-start);
        }
        return ans;
    }
};
```

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`

Code with sliding window approach :

```
class Solution {
public:
    int numberOfSubarrays(vector<int>& nums, int k) {
        int n=nums.size();
        int ans=0;
        int start=0,end=0,odd=0;
        while(end<n)
        {
            if(nums[end]%2)         //move the end until odd becomes equal to k
                odd++;
            if(odd==k)
            {
                int x=1,y=1;
                //x is the number of even numbers beyond the last odd number and next
odd to that number. (in the window).
                end++;
                while(end<n && nums[end]%2 == 0)
                {
                    x++;
                    end++;
                }
                //y is the number of even numbers before the first odd number and pre
vious odd to that number (in the window).

                while(start<n && nums[start]%2 == 0)
                {
                    y++;
                    start++;
                }
                start++;
                odd-=1;
                ans += x*y;   // Add the number of possible combinations (subarrays),
to the answer.
            }
            else
                end++;
        }
        return ans;
    }
};
```

Note :

- The idea behind the above approach is as follows:
- Let's consider the following testcase :

```
nums =  [ 2, 2, 2,1, 2, 2, 1, 2,  2, 2]     and     k = 2
```

- Now we know that since k=2, we must include subarray temp = [1, 2, 2, 1].
- If we had no other element in the array this would be the only possible subarray and the answer would be 1.
- Here we can include the whole array as an answer or we can delete one from last or from first or we can delete both and so on until we are including the "temp" ....
- So total number of such possibilities are = (x+1) * (y+1) where x = 3(first 3 even numbers(2's here)) and y = 3(last 3 even numbers(2's here)).
- So we get 16 combinations possible here.

  This is the basic idea we are using to solve this problem.

1438. Longest Continuous Subarray With Absolute Diff Less Than or Equal to Limit

Given an array of integers `nums` and an integer `limit`, return the size of the longest **non-empty** subarray such that the absolute difference between any two elements of this subarray is less than or equal to `limit`.

Example 1:

```
Input: nums = [8,2,4,7], limit = 4
Output: 2
Explanation: All subarrays are:
[8] with maximum absolute diff |8-8| = 0 <= 4.
[8,2] with maximum absolute diff |8-2| = 6 > 4.
[8,2,4] with maximum absolute diff |8-2| = 6 > 4.
[8,2,4,7] with maximum absolute diff |8-2| = 6 > 4.
[2] with maximum absolute diff |2-2| = 0 <= 4.
[2,4] with maximum absolute diff |2-4| = 2 <= 4.
[2,4,7] with maximum absolute diff |2-7| = 5 > 4.
[4] with maximum absolute diff |4-4| = 0 <= 4.
[4,7] with maximum absolute diff |4-7| = 3 <= 4.
[7] with maximum absolute diff |7-7| = 0 <= 4.
Therefore, the size of the longest subarray is 2.
```

Example 2:

```
Input: nums = [10,1,2,4,7,2], limit = 5
Output: 4
Explanation: The subarray [2,4,7,2] is the longest since the maximum absolute diff is |
```

Example 3:

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

Constraints:

- `1 <= nums.length <= 10^5`
- `1 <= nums[i] <= 10^9`
- `0 <= limit <= 10^9`

Code with sliding window approach using map :

```
class Solution {
public:
    int longestSubarray(vector<int>& nums, int limit) {
        int n=nums.size(), ans = 1,start = 0;
        if(n==0)
            return 0;
        map<int,int> m;
        for(int end=0;end<n;end++)
        {
            m[nums[end]]++;    //include the current element
            
            //Delete all the occurences of elements until the condition is satisfied.
That is until we either reach the max value or the min value in the current window by
shifting the start.
            while(m.size()>1 && (m.rbegin()->first - m.begin()->first)>limit)
            {
                m[nums[start]]--;
                if(m[nums[start]]==0)
                    m.erase(nums[start]);
                start++;
            }
            ans = max(ans,end-start+1);
        }
        return ans;
    }
};
```

Code using multiset :

```
class Solution {
public:
    int longestSubarray(vector<int>& nums, int limit) {
        int n=nums.size(), ans = 1,start = 0,end;
        if(n==0)
            return 0;
        multiset<int> s;
        for(end=0;end<n;end++)
        {
            s.insert(nums[end]);    //include the current element

            //Delete the start element if the difference between the maximum and the
minimum element is greater than the limit
            if(*s.rbegin()-*s.begin()>limit)
                s.erase(s.find(nums[start++]));
        }
        return end-start;
    }
};
```

1510. Stone Game IV

Alice and Bob take turns playing a game, with Alice starting first.

Initially, there are `n` stones in a pile. On each player's turn, that player makes a *move* consisting of removing **any** non-zero **square number** of stones in the pile.

Also, if a player cannot make a move, he/she loses the game.

Given a positive integer `n`. Return `True` if and only if Alice wins the game otherwise return `False`, assuming both players play optimally.

Example 1:

```
Input: n = 1
Output: true
Explanation: Alice can remove 1 stone winning the game because Bob doesn't have any mov
```

Example 2:

```
Input: n = 2
Output: false
Explanation: Alice can only remove 1 stone, after that Bob removes the last one winning
```

Example 3:

```
Input: n = 4
Output: true
Explanation: n is already a perfect square, Alice can win with one move, removing 4 sto
```

Example 4:

```
Input: n = 7
Output: false
Explanation: Alice can't win the game if Bob plays optimally.
If Alice starts removing 4 stones, Bob will remove 1 stone then Alice should remove onl
If Alice starts removing 1 stone, Bob will remove 4 stones then Alice only can remove 1
```

Example 5:

```
Input: n = 17
Output: false
Explanation: Alice can't win the game if Bob plays optimally.
```

Constraints:

- `1 <= n <= 10^5`

Code with DP approach :

```
class Solution {
public:
    bool winnerSquareGame(int n) {
        vector<bool> dp(100010,false);

        dp[1] = true;
        for(int i=3;i<n+5;i++)
        {
            for(int j=1;j<=sqrt(i);j++)
            {
                if(dp[i-(j*j)] == false)
                {
                    dp[i]=true;
                    break;
                }
            }
        }
        return dp[n];
    }
};
```

1626. Best Team With No Conflicts

You are the manager of a basketball team. For the upcoming tournament, you want to choose the team with the highest overall score. The score of the team is the **sum** of scores of all the players in the team.

However, the basketball team is not allowed to have **conflicts**. A **conflict** exists if a younger player has a **strictly higher** score than an older player. A conflict does **not** occur between players of the same age.

Given two lists, `scores` and `ages`, where each `scores[i]` and `ages[i]` represents the score and age of the `i`th player, respectively, return *the highest overall score of all possible basketball teams*.

Example 1:

```
Input: scores = [1,3,5,10,15], ages = [1,2,3,4,5]
Output: 34
Explanation: You can choose all the players.
```

Example 2:

```
Input: scores = [4,5,6,5], ages = [2,1,2,1]
Output: 16
Explanation: It is best to choose the last 3 players. Notice that you are allowed to ch
```

Example 3:

```
Input: scores = [1,2,3,5], ages = [8,9,10,1]
Output: 6
Explanation: It is best to choose the first 3 players.
```

Constraints:

- `1 <= scores.length, ages.length <= 1000`
- `scores.length == ages.length`
- `1 <= scores[i] <=` 10^6
- `1 <= ages[i] <= 1000`

Code using 1-D DP approach :

```
class Solution {
public:
    int bestTeamScore(vector<int>& scores, vector<int>& ages) {
        int n = scores.size(),res = 0;
        vector<pair<int,int>> v;
        for(int i=0;i<n;i++)
            v.push_back({ages[i],scores[i]});

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

        vector<int> dp(1005);

        for(int i=0;i<n;i++)
        {
            dp[i] = v[i].second;
            for(int j=0;j<i;j++)
            {
                if(v[j].second > v[i].second)
                    continue;
                dp[i] = max(dp[i], dp[j]+v[i].second);
            }
            res = max(res,dp[i]);
        }
        return res;
    }
};
```

1642. Furthest Building You Can Reach

You are given an integer array `heights` representing the heights of buildings, some `bricks`, and some `ladders`.

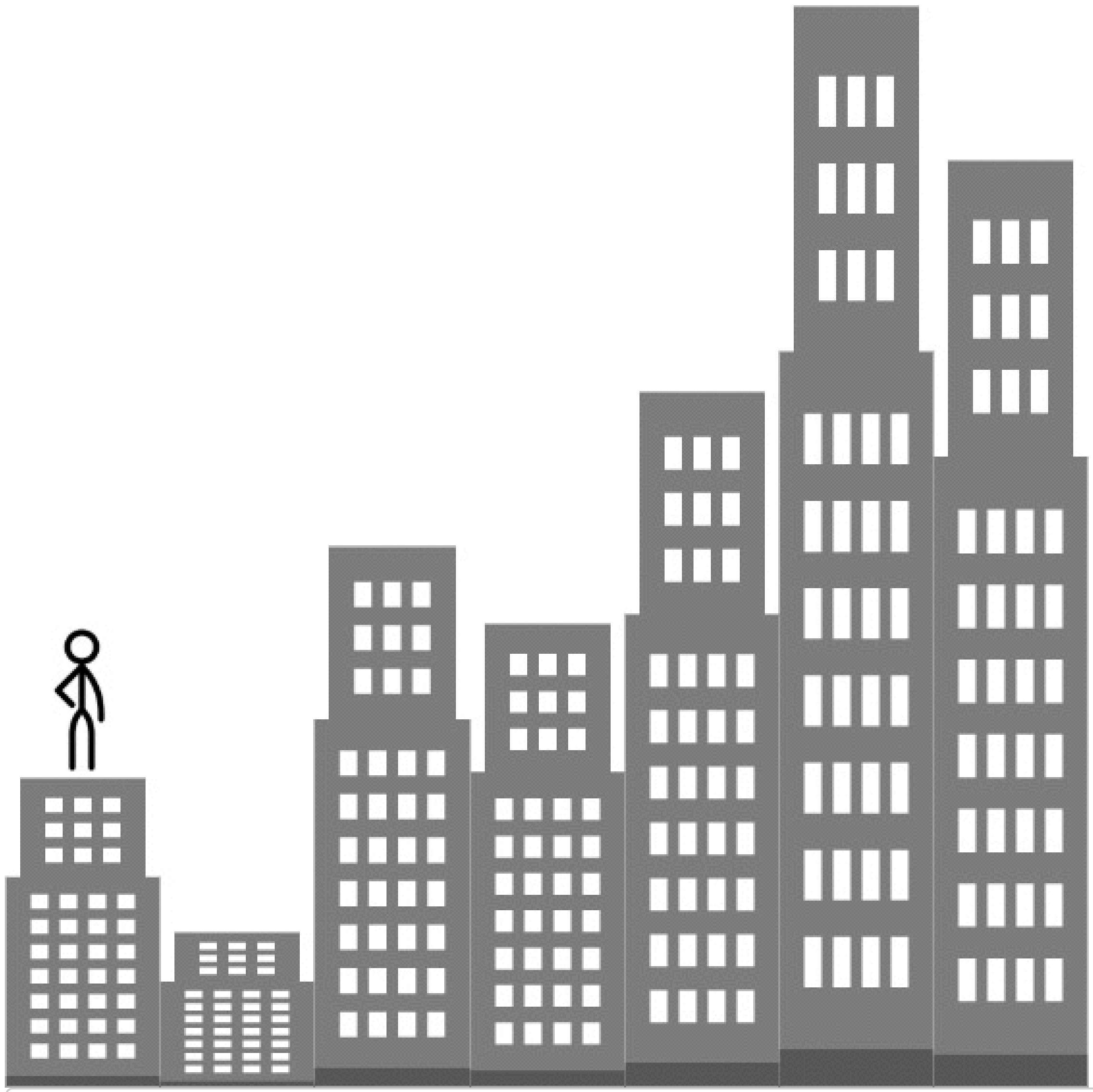
You start your journey from building `0` and move to the next building by possibly using bricks or ladders.

While moving from building `i` to building `i+1` (**0-indexed**),

- If the current building's height is **greater than or equal** to the next building's height, you do **not** need a ladder or bricks.
- If the current building's height is **less than** the next building's height, you can either use **one ladder** or `(h[i+1] - h[i])` **bricks**.

Return the furthest building index (0-indexed) you can reach if you use the given ladders and bricks optimally.

Example 1:



```
Input: heights = [4,2,7,6,9,14,12], bricks = 5, ladders = 1
Output: 4
Explanation: Starting at building 0, you can follow these steps:
- Go to building 1 without using ladders nor bricks since 4 >= 2.
- Go to building 2 using 5 bricks. You must use either bricks or ladders because 2 < 7.
- Go to building 3 without using ladders nor bricks since 7 >= 6.
- Go to building 4 using your only ladder. You must use either bricks or ladders becaus
It is impossible to go beyond building 4 because you do not have any more bricks or lad
```

Example 2:

```
Input: heights = [4,12,2,7,3,18,20,3,19], bricks = 10, ladders = 2
Output: 7
```

Example 3:

```
Input: heights = [14,3,19,3], bricks = 17, ladders = 0
Output: 3
```

Constraints:

- `1 <= heights.length <= 10`^{5}
- `1 <= heights[i] <= 10`^{6}
- `0 <= bricks <= 10`^{9}
- `0 <= ladders <= heights.length`

Code using priority queue :

```
class Solution {
public:
    int furthestBuilding(vector<int>& h, int bricks, int ladders) {
        priority_queue<int> pq;
        int i=1;
        for(i=1;i<h.size();i++)
        {
            int d = h[i]-h[i-1];
            if(d>0)
                pq.push(-d);
            if(pq.size()>ladders)
            {
                bricks+=pq.top();
                pq.pop();
                if(bricks<0)
                    return i-1;
            }
        }
        return i-1;
    }
};
```

Code using multiset :

```
class Solution {
public:
    int furthestBuilding(vector<int>& h, int bricks, int ladders) {
        multiset<int> m;
        int i=1;
        for(i=1;i<h.size();i++)
        {
            int d = h[i]-h[i-1];
            if(d>0)
                m.insert(d);
            if(m.size()>ladders)
            {
                bricks-=*m.begin();
                m.erase(m.begin());
                if(bricks<0)
                    return i-1;
            }
        }
        return i-1;
    }
};
```