

My LeetCode Submissions - @Prabhjeetsandhu010

[Download PDF](#)[Follow @TheShubham99 on GitHub](#)[Star on GitHub](#)[View Source Code](#)

[199 Binary Tree Right Side View \(link\)](#)

Description

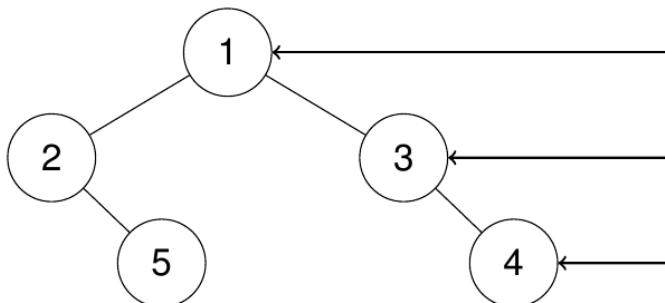
Given the root of a binary tree, imagine yourself standing on the **right side** of it, return *the values of the nodes you can see ordered from top to bottom.*

Example 1:

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

Output: [1,3,4]

Explanation:

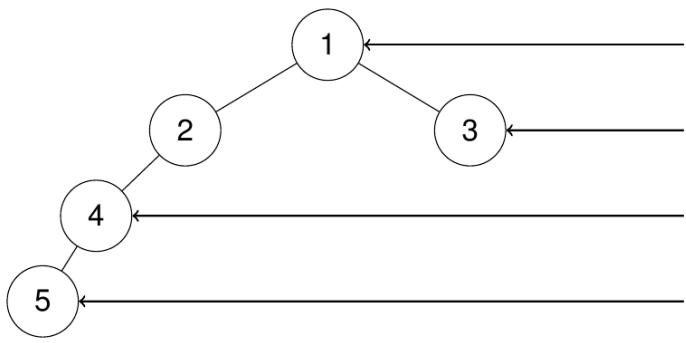


Example 2:

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

Output: [1,3,4,5]

Explanation:



Example 3:

Input: root = [1,null,3]

Output: [1,3]

Example 4:

Input: root = []

Output: []

Constraints:

- The number of nodes in the tree is in the range [0, 100].
- $-100 \leq \text{Node.val} \leq 100$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * 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(right) {}
 * };
 */
class Solution {
public:
    vector<int> rightSideView(TreeNode* root) {
        vector<int>ans;
        ans = get_right_view(root);
        return ans;
    }
private:
    vector<int>get_right_view(TreeNode* &root){
        if(!root) return {};
        vector<int>ans;
        queue<TreeNode*>level_queue;
        level_queue.push(root);

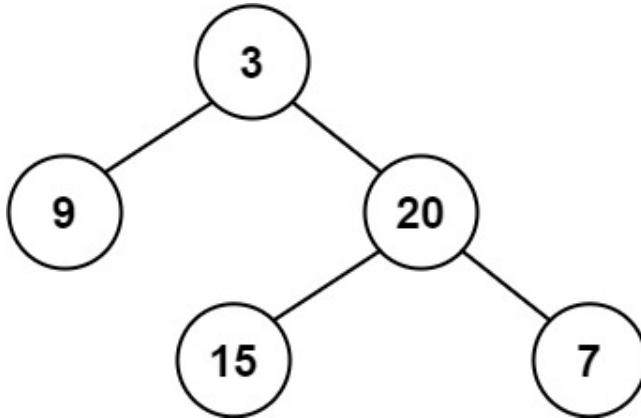
        while(!level_queue.empty()){
            int level_queue_size = level_queue.size();
            int right_most_element = INT_MIN;
            for(int i=0; i<level_queue_size; i++){
                TreeNode* front = level_queue.front();
                level_queue.pop();
                if(front->left)level_queue.push(front->left);
                if(front->right)level_queue.push(front->right);
                right_most_element = front->val;
            }
            ans.push_back(right_most_element);
        }
        return ans;
    }
};
```

[110 Balanced Binary Tree \(link\)](#)

Description

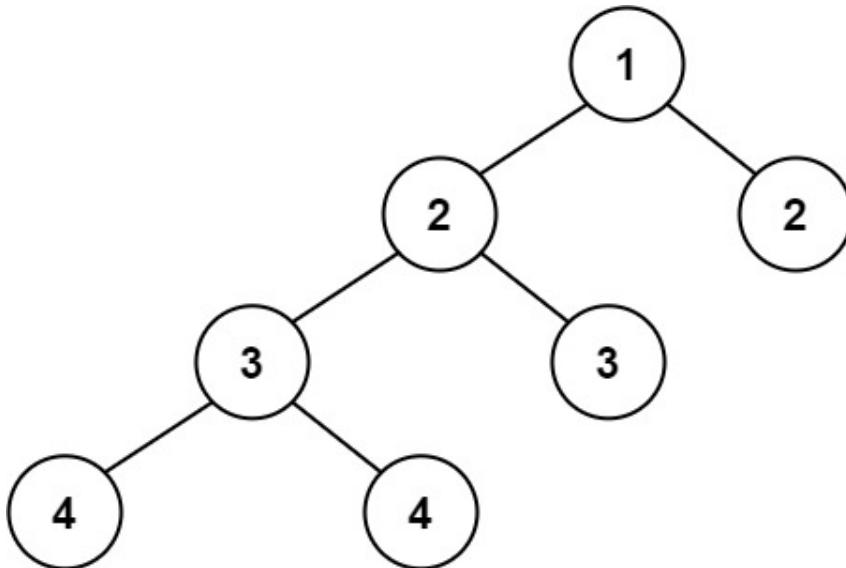
Given a binary tree, determine if it is **height-balanced**.

Example 1:



```
Input: root = [3,9,20,null,null,15,7]  
Output: true
```

Example 2:



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

Example 3:

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

Constraints:

- The number of nodes in the tree is in the range $[0, 5000]$.
- $-10^4 \leq \text{Node.val} \leq 10^4$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * 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(right) {}
 * };
 */
class Solution {
public:
    bool isBalanced(TreeNode* root) {
        return check_balanced(root).is_height_balanced;
    }
    struct Balance{
        int height;
        bool is_height_balanced;
    };
private:
    Balance check_balanced(TreeNode* root){
        if(!root) return {0,true};
        Balance left = check_balanced(root->left);
        Balance right = check_balanced(root->right);

        bool is_balanced = left.is_height_balanced && right.is_height_balanced && abs(left.height - right.height) <= 1;
        return {max(left.height,right.height)+1,is_balanced};
    }
};
```

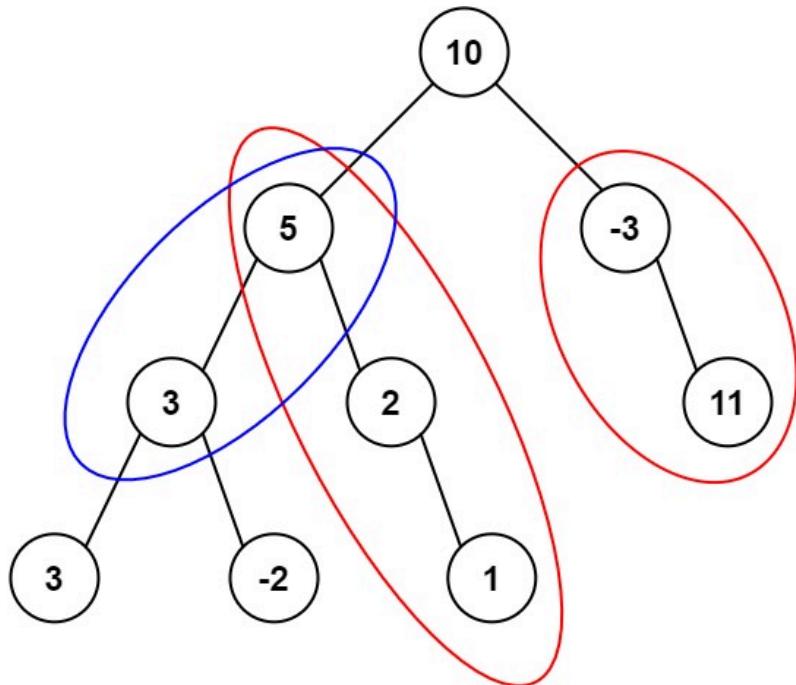
[437 Path Sum III \(link\)](#)

Description

Given the root of a binary tree and an integer `targetSum`, return *the number of paths where the sum of the values along the path equals targetSum*.

The path does not need to start or end at the root or a leaf, but it must go downwards (i.e., traveling only from parent nodes to child nodes).

Example 1:



Input: root = [10,5,-3,3,2,null,11,3,-2,null,1], targetSum = 8

Output: 3

Explanation: The paths that sum to 8 are shown.

Example 2:

Input: root = [5,4,8,11,null,13,4,7,2,null,null,5,1], targetSum = 22

Output: 3

Constraints:

- The number of nodes in the tree is in the range $[0, 1000]$.
- $-10^9 \leq \text{Node.val} \leq 10^9$
- $-1000 \leq \text{targetSum} \leq 1000$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * 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(right) {}
 * };
 */
class Solution {
public:
    int pathSum(TreeNode* root, int targetSum) {
        unordered_map<long long,int>path_sum_count_map;
        long long path_sum=0;
        int count_path = 0;
        path_sum_count_map[0]++;
        get_count_path(root,targetSum,path_sum,count_path,path_sum_count_map);
        return count_path;
    }

private:
    int get_count_path(TreeNode*& root,int targetSum,long long path_sum, int &count_path,unordered_map<long long,int>path_sum_count_map){
        if(!root){
            return 0;
        }
        path_sum += root->val;

        if(path_sum_count_map.count( path_sum - targetSum)){
            count_path += path_sum_count_map[ path_sum - targetSum ];
        }

        path_sum_count_map[path_sum]++; // It will be after checking in map
        get_count_path(root->left,targetSum,path_sum,count_path,path_sum_count_map);
        get_count_path(root->right,targetSum,path_sum,count_path,path_sum_count_map);
        path_sum_count_map[path_sum]--;
        return count_path;
    }
};
```

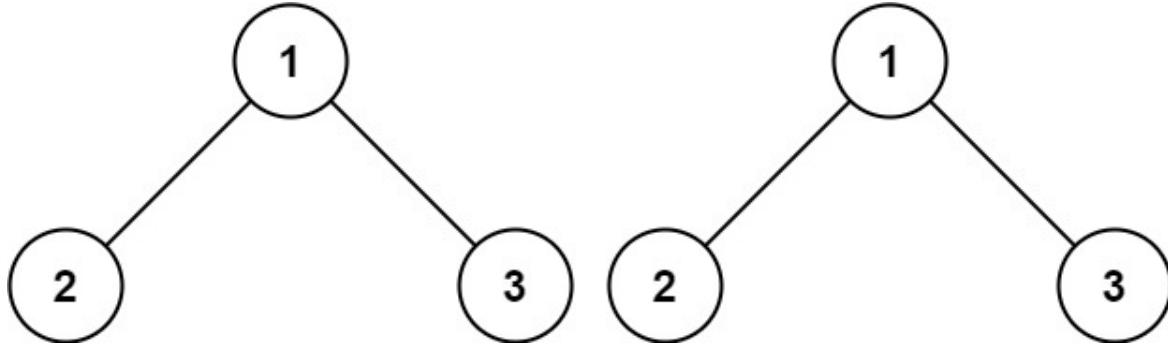
100 Same Tree (link)

Description

Given the roots of two binary trees p and q , write a function to check if they are the same or not.

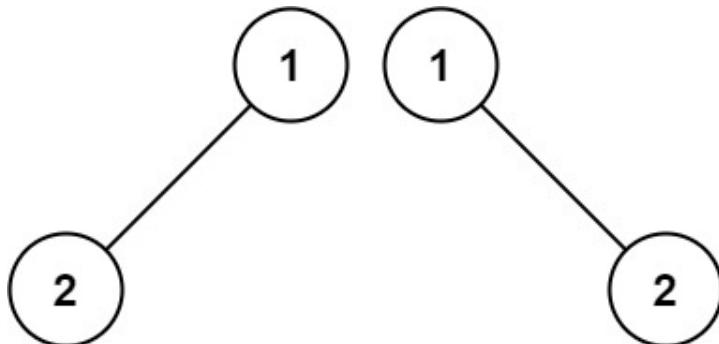
Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

Example 1:



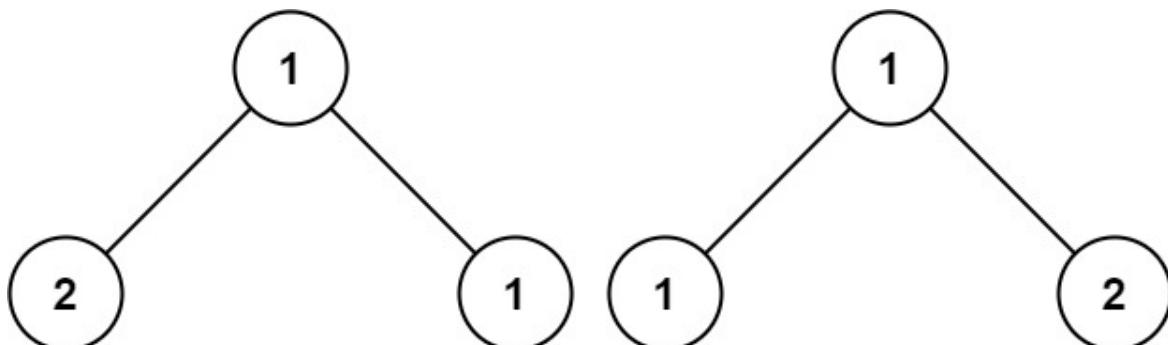
Input: $p = [1,2,3]$, $q = [1,2,3]$
Output: true

Example 2:



Input: $p = [1,2]$, $q = [1,null,2]$
Output: false

Example 3:



Input: $p = [1,2,1]$, $q = [1,1,2]$
Output: false

Constraints:

- The number of nodes in both trees is in the range $[0, 100]$.
- $-10^4 \leq \text{Node.val} \leq 10^4$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * 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(right) {}
 * };
 */
class Solution {
public:
    bool isSameTree(TreeNode* p, TreeNode* q) {
        if(!p && !q) return true;
        if((!p && q) || (p && !q)) return false;
        if(p->val != q->val) return false;

        return isSameTree(p->left,q->left) && isSameTree(p->right,q->right);
    }
};
```

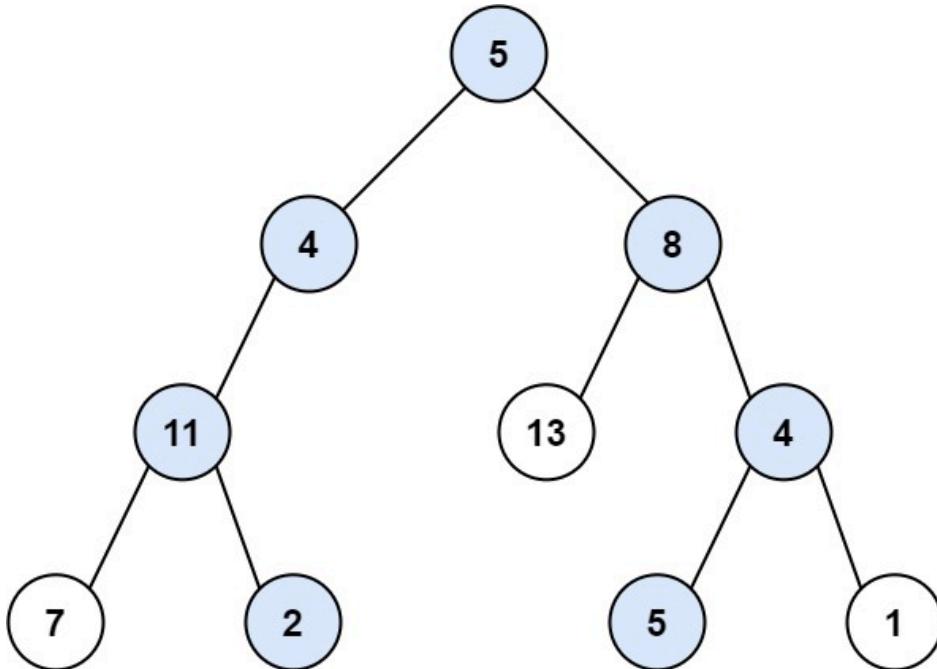
[113 Path Sum II \(link\)](#)

Description

Given the root of a binary tree and an integer `targetSum`, return *all root-to-leaf paths where the sum of the node values in the path equals targetSum*. Each path should be returned as a list of the node **values**, not node references.

A **root-to-leaf** path is a path starting from the root and ending at any leaf node. A **leaf** is a node with no children.

Example 1:



Input: root = [5,4,8,11,null,13,4,7,2,null,null,5,1], targetSum = 22

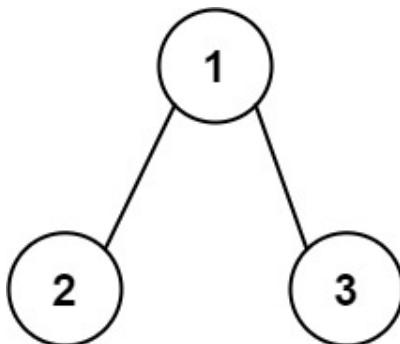
Output: [[5,4,11,2],[5,8,4,5]]

Explanation: There are two paths whose sum equals targetSum:

$$5 + 4 + 11 + 2 = 22$$

$$5 + 8 + 4 + 5 = 22$$

Example 2:



Input: root = [1,2,3], targetSum = 5

Output: []

Example 3:

Input: root = [1,2], targetSum = 0

Output: []

Constraints:

- The number of nodes in the tree is in the range [0, 5000].
- $-1000 \leq \text{Node.val} \leq 1000$
- $-1000 \leq \text{targetSum} \leq 1000$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * 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(right) {}
 * };
 */
class Solution {
public:
    vector<vector<int>> pathSum(TreeNode* root, int targetSum) {
        vector<vector<int>> ans;
        vector<int> curr_path;
        long long curr_sum = 0;
        get_target_path(root, targetSum, curr_sum, curr_path, ans);

        return ans;
    }

private:
    void get_target_path(TreeNode*& root, int targetSum, long long curr_sum,
                         vector<int>& curr_path, vector<vector<int>>& ans) {
        if (!root){
            return;
        }

        curr_sum += root->val;
        curr_path.push_back(root->val);

        if (!root->left && !root->right) {
            // LEAF
            if (curr_sum == targetSum) {
                ans.push_back(curr_path);
            }
        }

        get_target_path(root->left, targetSum, curr_sum, curr_path, ans);
        get_target_path(root->right, targetSum, curr_sum, curr_path, ans);
        curr_sum -= root->val;
        curr_path.pop_back();
    }
};
```

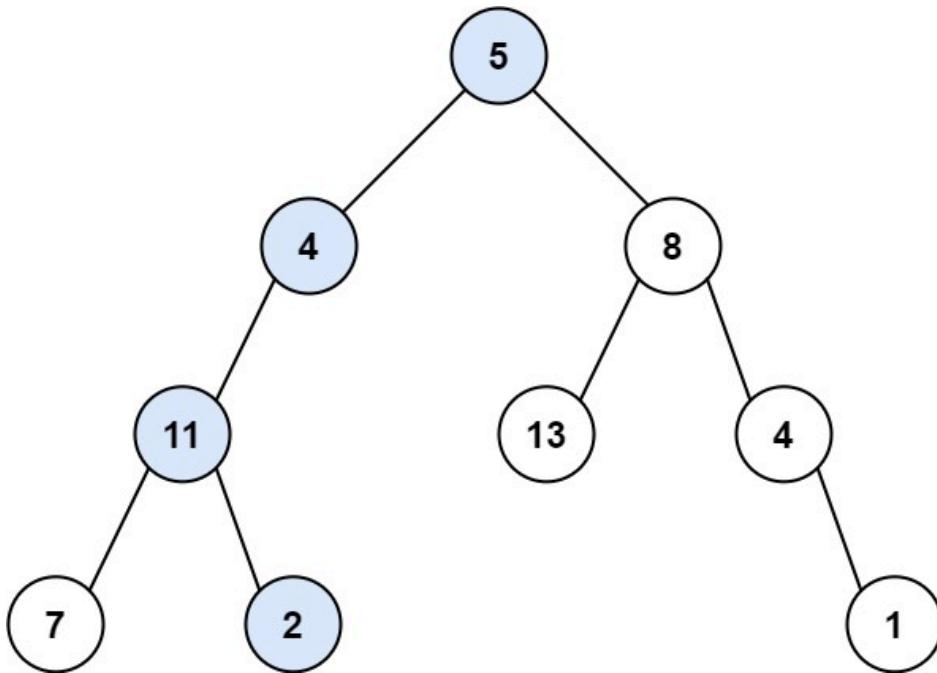
[112 Path Sum \(link\)](#)

Description

Given the root of a binary tree and an integer `targetSum`, return `true` if the tree has a **root-to-leaf** path such that adding up all the values along the path equals `targetSum`.

A **leaf** is a node with no children.

Example 1:

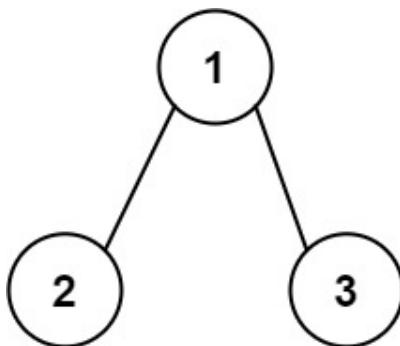


Input: `root = [5,4,8,11,null,13,4,7,2,null,null,null,1]`, `targetSum = 22`

Output: `true`

Explanation: The root-to-leaf path with the target sum is shown.

Example 2:



Input: `root = [1,2,3]`, `targetSum = 5`

Output: `false`

Explanation: There are two root-to-leaf paths in the tree:

(1 --> 2): The sum is 3.

(1 --> 3): The sum is 4.

There is no root-to-leaf path with sum = 5.

Example 3:

Input: root = [], targetSum = 0

Output: false

Explanation: Since the tree is empty, there are no root-to-leaf paths.

Constraints:

- The number of nodes in the tree is in the range [0, 5000].
- $-1000 \leq \text{Node.val} \leq 1000$
- $-1000 \leq \text{targetSum} \leq 1000$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * 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(right) {}
 * };
 */
class Solution {
public:
    bool hasPathSum(TreeNode* root, int targetSum) {
        if(!root) return false;
        if(!root->left && !root->right){
            if(targetSum == root->val) return true;
            else return false;
        }

        bool left = hasPathSum(root->left,targetSum - root->val);
        bool right = hasPathSum(root->right,targetSum - root->val);

        return left || right;
    }
};
```

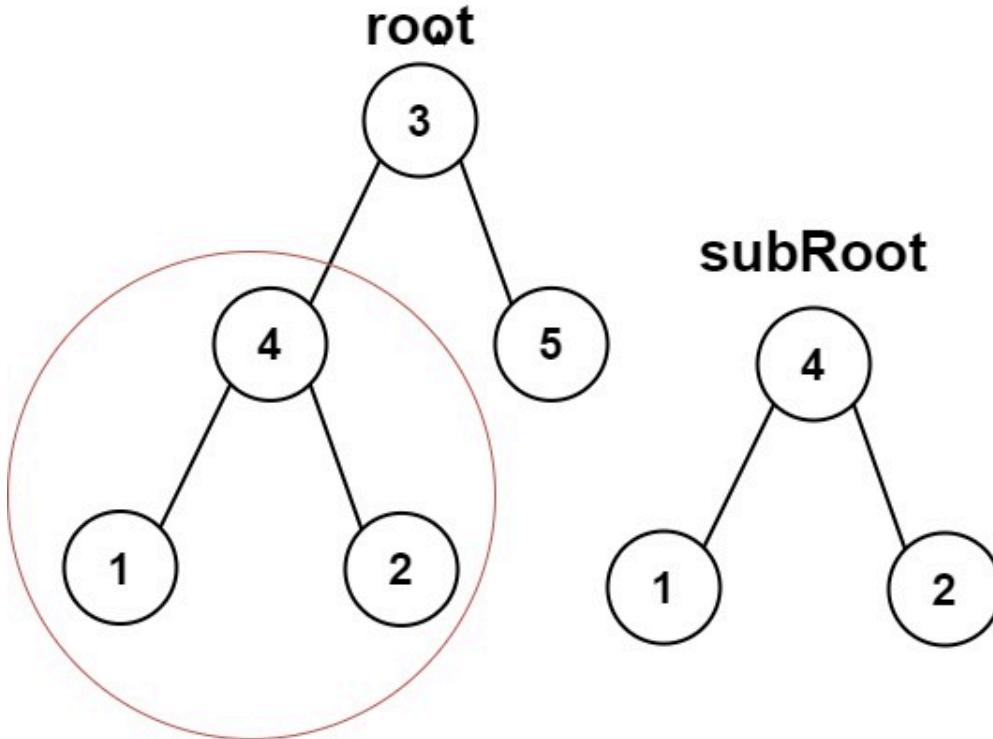
572 Subtree of Another Tree (link)

Description

Given the roots of two binary trees `root` and `subRoot`, return `true` if there is a subtree of `root` with the same structure and node values of `subRoot` and `false` otherwise.

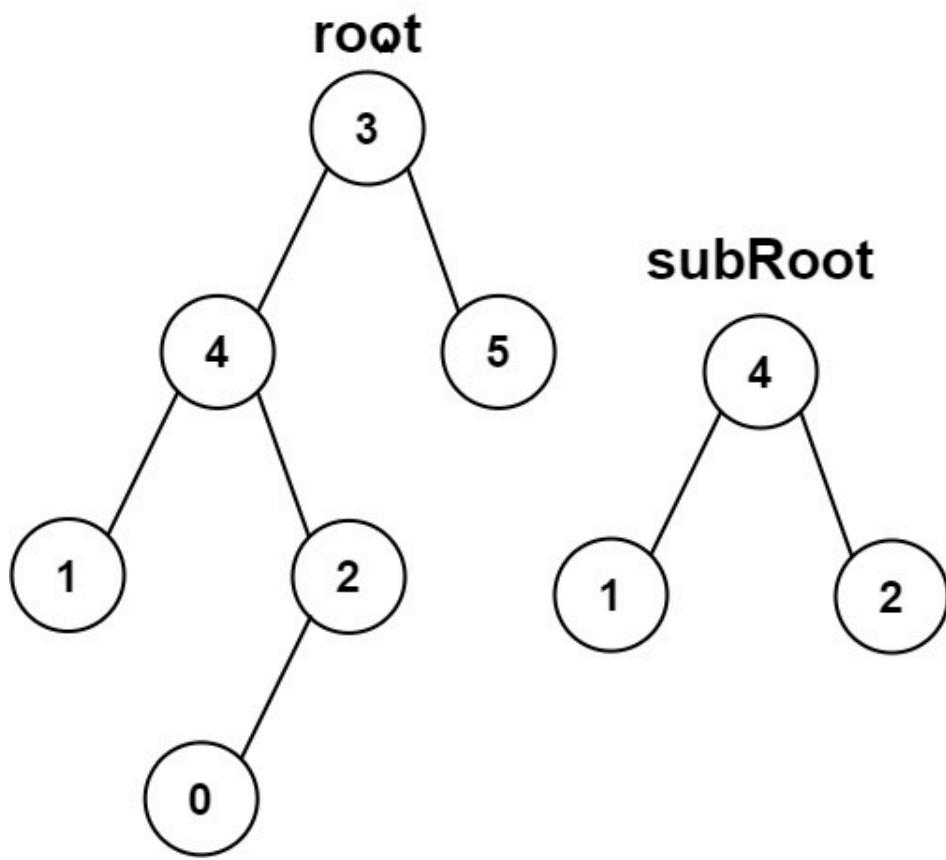
A subtree of a binary tree `tree` is a tree that consists of a node in `tree` and all of this node's descendants. The tree `tree` could also be considered as a subtree of itself.

Example 1:



Input: `root = [3,4,5,1,2]`, `subRoot = [4,1,2]`
Output: `true`

Example 2:



```
Input: root = [3,4,5,1,2,null,null,null,null,0], subRoot = [4,1,2]  
Output: false
```

Constraints:

- The number of nodes in the root tree is in the range [1, 2000].
- The number of nodes in the subRoot tree is in the range [1, 1000].
- $-10^4 \leq \text{root.val} \leq 10^4$
- $-10^4 \leq \text{subRoot.val} \leq 10^4$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * 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(right) {}
 * };
 */
class Solution {
public:
    bool isSubtree(TreeNode* root, TreeNode* subRoot) {
        if(!root) return false;

        bool same_check = is_same(root,subRoot);
        if(same_check) return true;

        return isSubtree(root->left,subRoot) || isSubtree(root->right,subRoot);
    }
private:
    bool is_same(TreeNode*root,TreeNode*sub_root){
        if(!root && !sub_root) return true;

        if(( !root && sub_root ) || (root && !sub_root))return false;

        if(root->val != sub_root->val) return false;

        bool left = is_same(root->left,sub_root->left);
        bool right = is_same(root->right, sub_root->right);

        return left && right; // This important [Both must give true for tree to be identical]
    }
};
```

450 Delete Node in a BST (link)

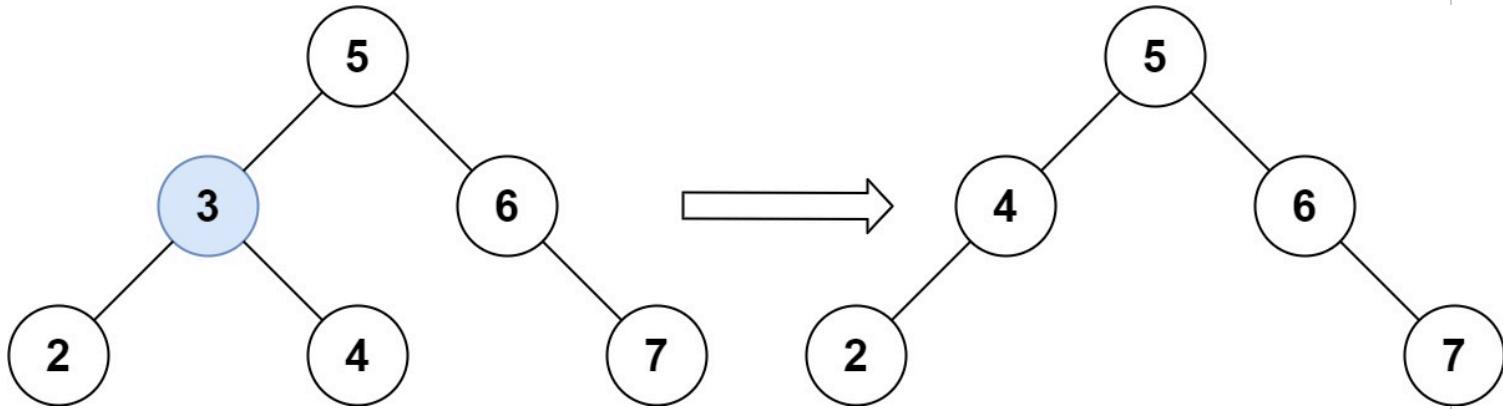
Description

Given a root node reference of a BST and a key, delete the node with the given key in the BST. Return *the root node reference (possibly updated) of the BST*.

Basically, the deletion can be divided into two stages:

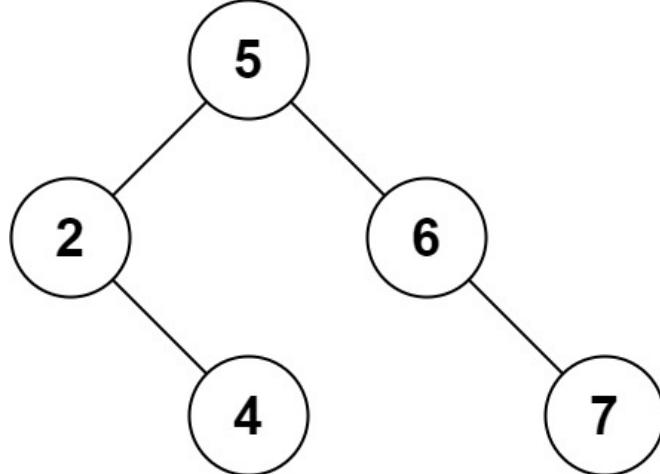
1. Search for a node to remove.
2. If the node is found, delete the node.

Example 1:



Input: root = [5,3,6,2,4,null,7], key = 3
Output: [5,4,6,2,null,null,7]

Explanation: Given key to delete is 3. So we find the node with value 3 and delete it. One valid answer is [5,4,6,2,null,null,7], shown in the above BST. Please notice that another valid answer is [5,2,6,null,4,null,7] and it's also accepted.



Example 2:

Input: root = [5,3,6,2,4,null,7], key = 0
Output: [5,3,6,2,4,null,7]
Explanation: The tree does not contain a node with value = 0.

Example 3:

Input: root = [], key = 0
Output: []

Constraints:

- The number of nodes in the tree is in the range $[0, 10^4]$.
- $-10^5 \leq \text{Node.val} \leq 10^5$
- Each node has a **unique** value.
- root is a valid binary search tree.
- $-10^5 \leq \text{key} \leq 10^5$

Follow up: Could you solve it with time complexity $O(\text{height of tree})$?

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * 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(right) {}
 * };
 */
class Solution {
public:
    TreeNode* deleteNode(TreeNode* root, int key) {
        if(!root) return nullptr;
        if(root->val == key){
            return delete_from_BST(root, key);
        }
        if(root->val < key){
            root->right = deleteNode(root->right, key);
        }else if(root->val > key){
            root->left = deleteNode(root->left, key);
        }

        return root;
    }

private:
    TreeNode* delete_from_BST(TreeNode*& root, int key){
        if(!root->left && !root->right){
            delete root;
            return nullptr;
        }
        if(!root->left || !root->right){
            // 1 Child
            if(!root->left){
                TreeNode* remaining_tree = root->right;
                delete root;
                return remaining_tree;
            }
            else{
                TreeNode* remaining_tree = root->left;
                delete root;
                return remaining_tree;
            }
        }
        else{
            // 2 Children
            TreeNode* inorder_successor = get_inorder_successor(root->right);
            root->val = inorder_successor->val;
            root->right = deleteNode(root->right, inorder_successor->val);
            return root;
        }
    }

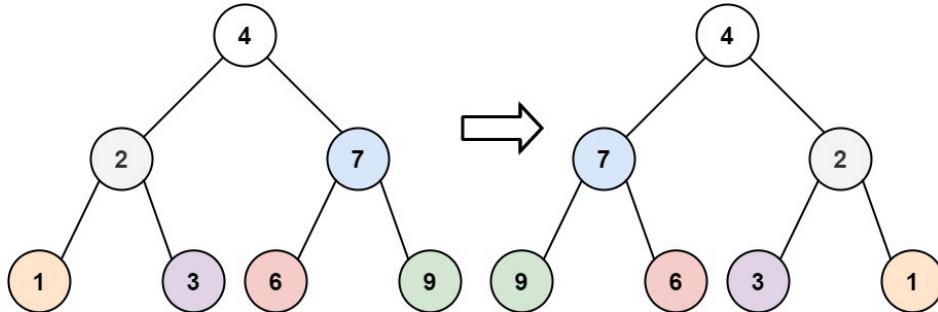
    TreeNode* get_inorder_successor(TreeNode* &root){
        if(!root) return nullptr;
        if(!root->left) return root;
        return get_inorder_successor(root->left);
    }
};
```

[226 Invert Binary Tree \(link\)](#)

Description

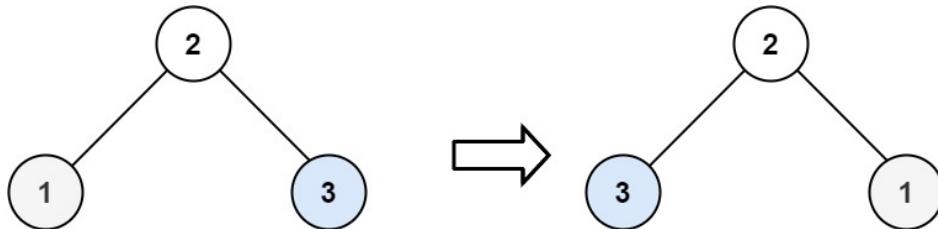
Given the root of a binary tree, invert the tree, and return *its root*.

Example 1:



```
Input: root = [4,2,7,1,3,6,9]  
Output: [4,7,2,9,6,3,1]
```

Example 2:



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

Example 3:

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

Constraints:

- The number of nodes in the tree is in the range $[0, 100]$.
- $-100 \leq \text{Node.val} \leq 100$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

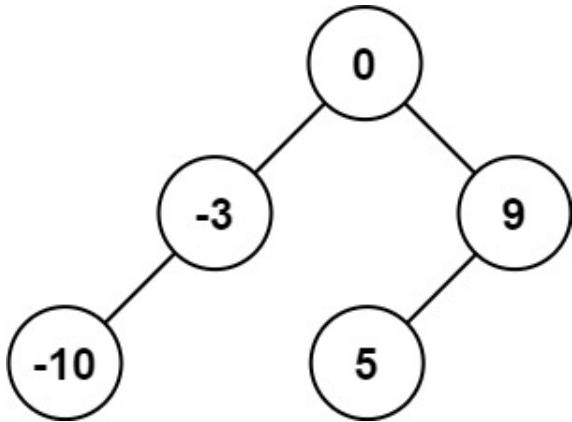
```
/*
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode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

[108 Convert Sorted Array to Binary Search Tree \(link\)](#)

Description

Given an integer array `nums` where the elements are sorted in **ascending order**, convert *it to a height-balanced binary search tree*.

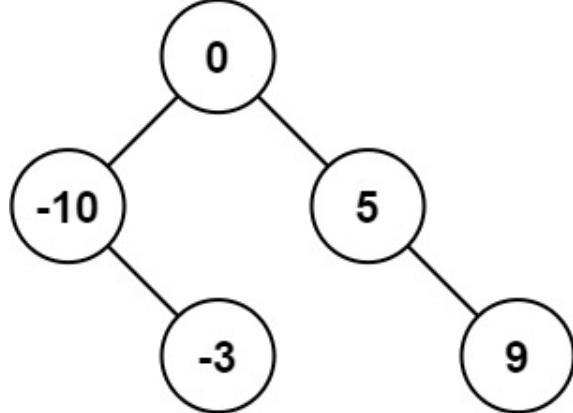
Example 1:



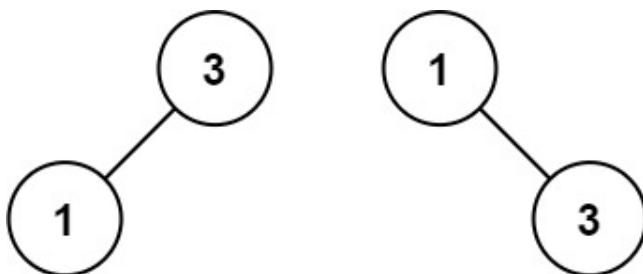
Input: `nums = [-10,-3,0,5,9]`

Output: `[0,−3,9,−10,null,5]`

Explanation: `[0,−10,5,null,−3,null,9]` is also accepted:



Example 2:



Input: `nums = [1,3]`

Output: `[3,1]`

Explanation: `[1,null,3]` and `[3,1]` are both height-balanced BSTs.

Constraints:

- $1 \leq \text{nums.length} \leq 10^4$

- $-10^4 \leq \text{nums}[i] \leq 10^4$
- `nums` is sorted in a **strictly increasing** order.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * 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(right) {}
 * };
 */
class Solution {
public:
    TreeNode* sortedArrayToBST(vector<int>& nums) {
        int start = 0, end = nums.size()-1;
        int mid = start + (end - start)/2;
        TreeNode* root = get_BST(nums,start,end);
        return root;
    }

private:
    TreeNode* get_BST(vector<int>& nums, int start, int end){
        if(start > end) return nullptr;

        int mid = start + (end - start)/2;
        TreeNode* root = new TreeNode(nums[mid]);
        root->left = get_BST(nums,start,mid-1);
        root->right = get_BST(nums,mid+1,end);

        return root;
    }
};
```

297 Serialize and Deserialize Binary Tree (link)

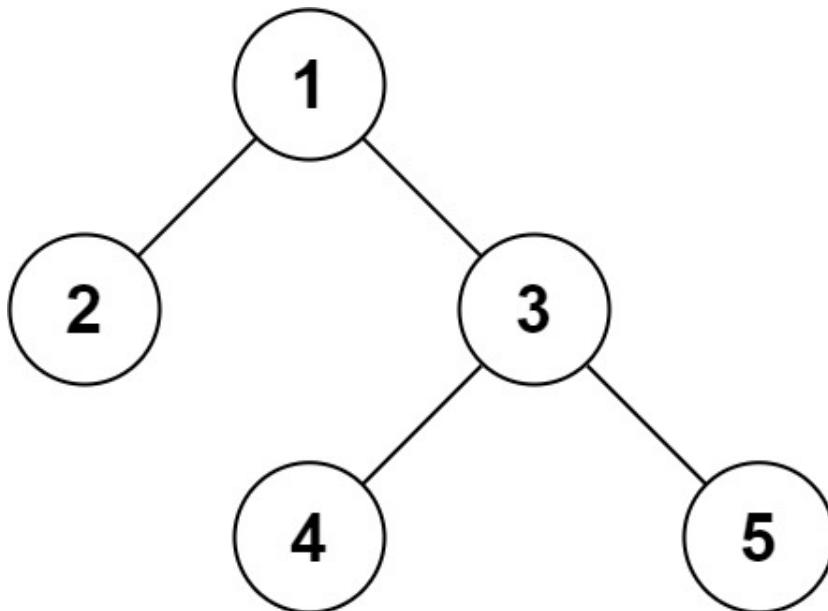
Description

Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.

Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure.

Clarification: The input/output format is the same as [how LeetCode serializes a binary tree](#). You do not necessarily need to follow this format, so please be creative and come up with different approaches yourself.

Example 1:



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

Example 2:

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

Constraints:

- The number of nodes in the tree is in the range $[0, 10^4]$.
- $-1000 \leq \text{Node.val} \leq 1000$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Codec {
public:

    // Encodes a tree to a single string.
    string serialize(TreeNode* root) {
        string encoded_str = "";
        // PREORDER BECAUSE NATURAL FLOW
        calculate_preorder(root,encoded_str);
        return encoded_str;
    }

    // Decodes your encoded data to tree.
    TreeNode* deserialize(string data) {
        int curr = 0;
        TreeNode* root = build_tree(data,curr);
        return root;
    }

private:

    TreeNode* build_tree(string &data,int &curr_index){
        if(curr_index >= data.size())return nullptr;

        int next_space = data.find(' ',curr_index);
        string curr_node = data.substr(curr_index,next_space-curr_index);
        curr_index = next_space+1;

        if(curr_node == "N"){
            return nullptr;
        }
        TreeNode* node = new TreeNode(stoi(curr_node));
        node->left = build_tree(data,curr_index);
        node->right = build_tree(data,curr_index);
        return node;
    }

    void calculate_preorder(TreeNode* root,string &encoded_str){
        if(!root){
            encoded_str += "N ";
            return;
        }
        encoded_str += to_string(root->val) + " ";
        calculate_preorder(root->left,encoded_str);
        calculate_preorder(root->right,encoded_str);
    }
};

// Your Codec object will be instantiated and called as such:
// Codec ser, deser;
// TreeNode* ans = deser.deserialize(ser.serialize(root));
```

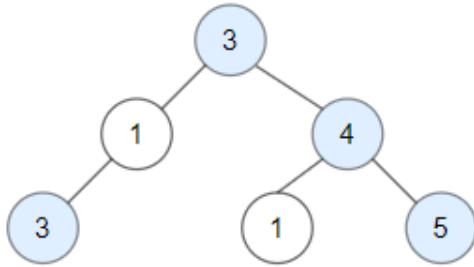
[1544 Count Good Nodes in Binary Tree \(link\)](#)

Description

Given a binary tree root, a node X in the tree is named **good** if in the path from root to X there are no nodes with a value *greater than* X.

Return the number of **good** nodes in the binary tree.

Example 1:



Input: root = [3,1,4,3,null,1,5]

Output: 4

Explanation: Nodes in blue are **good**.

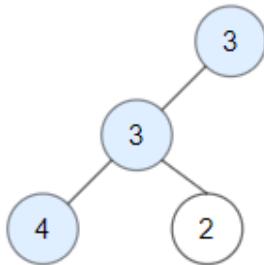
Root Node (3) is always a good node.

Node 4 → (3,4) is the maximum value in the path starting from the root.

Node 5 → (3,4,5) is the maximum value in the path

Node 3 → (3,1,3) is the maximum value in the path.

Example 2:



Input: root = [3,3,null,4,2]

Output: 3

Explanation: Node 2 → (3, 3, 2) is not good, because "3" is higher than it.

Example 3:

Input: root = [1]

Output: 1

Explanation: Root is considered as **good**.

Constraints:

- The number of nodes in the binary tree is in the range [1, 10^5].
- Each node's value is between [-10^4, 10^4].

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * 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(right) {}
 * };
 */
class Solution {
public:
    int goodNodes(TreeNode* root) {
        int count_good_nodes = 0, max_till_now = INT_MIN;
        get_count_good_nodes(root, count_good_nodes, max_till_now);
        return count_good_nodes;
    }

private:
    int get_count_good_nodes(TreeNode*& root, int &count_good_nodes, int max_till_now) {
        if(!root) return 0;
        if(root->val >= max_till_now) count_good_nodes++;
        max_till_now = max(max_till_now, root->val);
        int left_count = get_count_good_nodes(root->left, count_good_nodes, max_till_now);
        int right_count = get_count_good_nodes(root->right, count_good_nodes, max_till_now);

        return left_count + right_count;
    }
};
```

[543 Diameter of Binary Tree \(link\)](#)

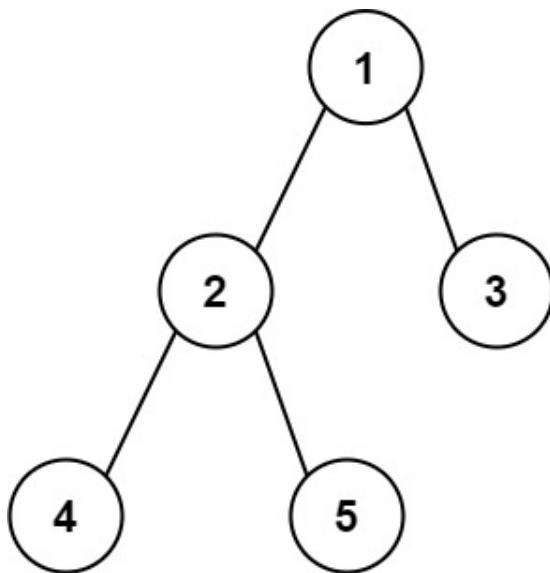
Description

Given the root of a binary tree, return *the length of the diameter of the tree*.

The **diameter** of a binary tree is the **length** of the longest path between any two nodes in a tree. This path may or may not pass through the root.

The **length** of a path between two nodes is represented by the number of edges between them.

Example 1:



Input: root = [1,2,3,4,5]

Output: 3

Explanation: 3 is the length of the path [4,2,1,3] or [5,2,1,3].

Example 2:

Input: root = [1,2]

Output: 1

Constraints:

- The number of nodes in the tree is in the range $[1, 10^4]$.
- $-100 \leq \text{Node.val} \leq 100$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * 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(right) {}
 * };
 */
class Solution {
public:
    int diameterOfBinaryTree(TreeNode* root) {
        if(!root) return 0;
        int max_diameter = 0;
        get_max_diameter(root, max_diameter);
        return max_diameter;
    }
private:
    int get_max_diameter(TreeNode* root, int &max_diameter){
        if(!root) return 0;
        int left_height = get_max_diameter(root->left, max_diameter);
        int right_height = get_max_diameter(root->right, max_diameter);
        max_diameter = max(max_diameter, left_height + right_height);
        return 1+max(left_height, right_height);
    }
};
```

[124 Binary Tree Maximum Path Sum \(link\)](#)

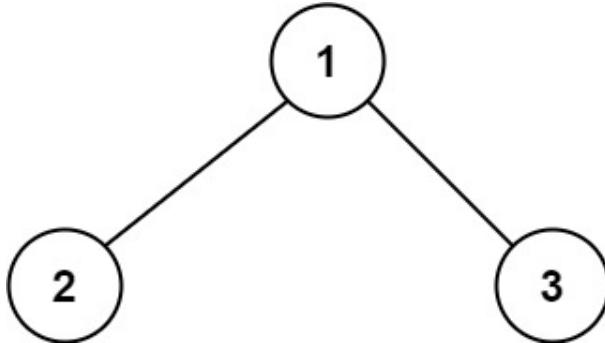
Description

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 non-empty path*.

Example 1:

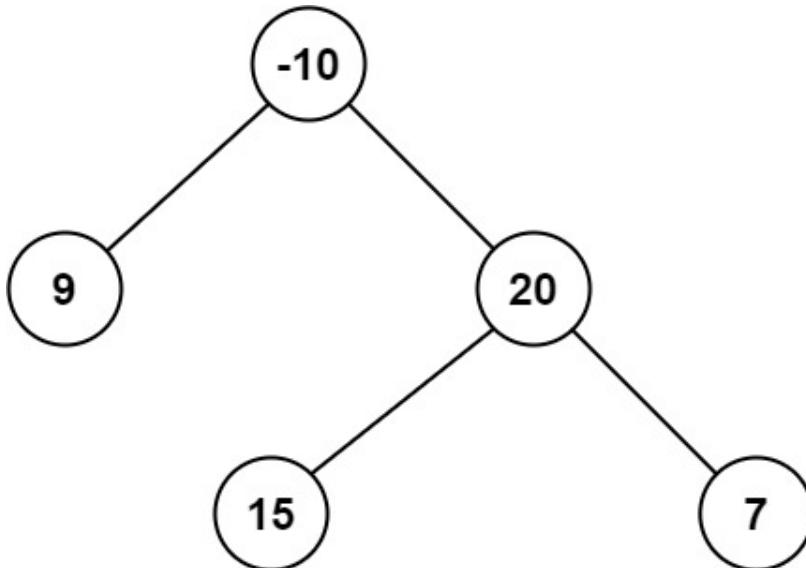


Input: root = [1,2,3]

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 $[1, 3 * 10^4]$.
- $-1000 \leq \text{Node.val} \leq 1000$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

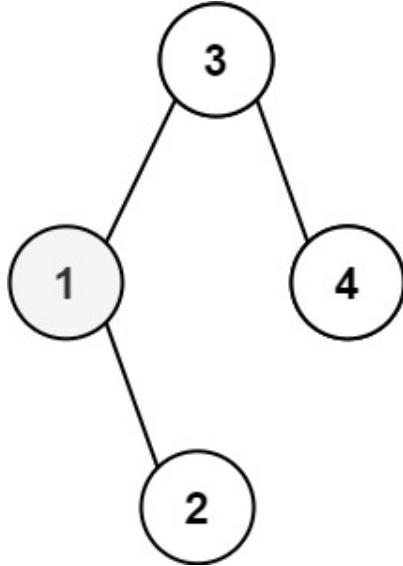
```
/*
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    int maxPathSum(TreeNode* root) {
        int max_sum = INT_MIN, curr_sum = 0;
        curr_sum = calculate_sum(root, max_sum);
        return max_sum;
    }
private:
    int calculate_sum(TreeNode* root, int &max_sum){
        if(!root) return 0;
        int left_sum = calculate_sum(root->left, max_sum);
        int right_sum = calculate_sum(root->right, max_sum);
        if(left_sum < 0) left_sum = 0;
        if(right_sum < 0) right_sum = 0;
        max_sum = max(max_sum, left_sum + right_sum + root->val);
        return root->val + max(left_sum, right_sum); // THis IMPORTANT
    }
};
```

230 Kth Smallest Element in a BST (link)

Description

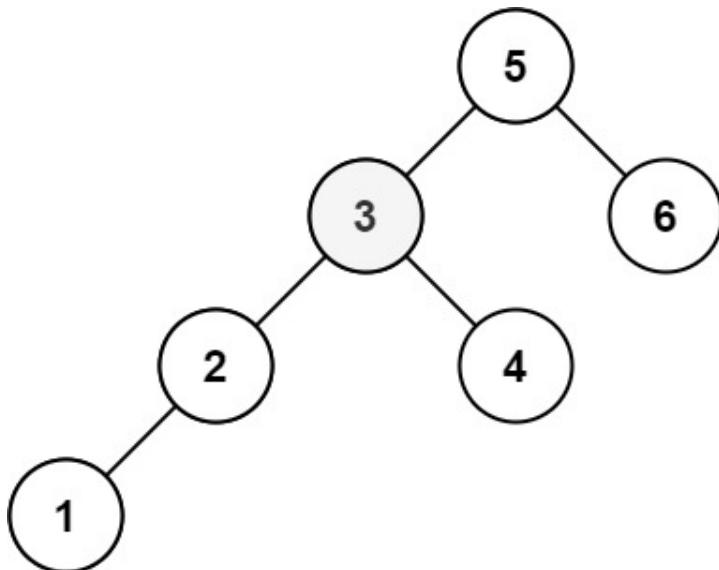
Given the root of a binary search tree, and an integer k, return *the kth smallest value (1-indexed) of all the values of the nodes in the tree.*

Example 1:



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

Example 2:



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

Constraints:

- The number of nodes in the tree is n.
- $1 \leq k \leq n \leq 10^4$

- $0 \leq \text{Node.val} \leq 10^4$

Follow up: If the BST is modified often (i.e., we can do insert and delete operations) and you need to find the k th smallest frequently, how would you optimize?

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * 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(right) {}
 * };
 */
class Solution {
public:
    int kthSmallest(TreeNode* root, int k) {
        int current_count = 0;
        int ans = -1;
        inorder_calculate_k(root, k, current_count,ans);
        return ans;
    }

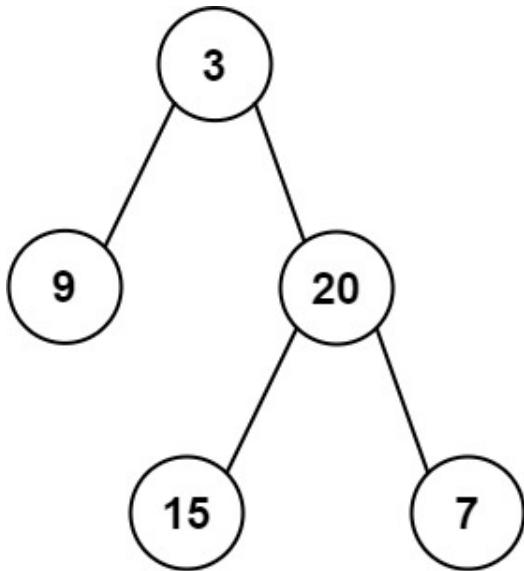
private:
    void inorder_calculate_k(TreeNode*& root, int k, int& current_count,int &ans) {
        if (!root)
            return;
        inorder_calculate_k(root->left, k, current_count,ans);
        current_count++;
        if (current_count == k) {
            ans = root->val;
            return;
        }
        inorder_calculate_k(root->right, k, current_count,ans);
    }
};
```

105 Construct Binary Tree from Preorder and Inorder Traversal [\(link\)](#)

Description

Given two integer arrays `preorder` and `inorder` where `preorder` is the preorder traversal of a binary tree and `inorder` is the inorder traversal of the same tree, construct and return *the binary tree*.

Example 1:



```
Input: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
Output: [3,9,20,null,null,15,7]
```

Example 2:

```
Input: preorder = [-1], inorder = [-1]
Output: [-1]
```

Constraints:

- $1 \leq \text{preorder.length} \leq 3000$
- $\text{inorder.length} == \text{preorder.length}$
- $-3000 \leq \text{preorder}[i], \text{inorder}[i] \leq 3000$
- `preorder` and `inorder` consist of **unique** values.
- Each value of `inorder` also appears in `preorder`.
- `preorder` is **guaranteed** to be the preorder traversal of the tree.
- `inorder` is **guaranteed** to be the inorder traversal of the tree.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * 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(right) {}
 * };
 */
class Solution {
public:
    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        unordered_map<int, int> inorder_index;
        for (int i = 0; i < inorder.size(); i++) {
            inorder_index[inorder[i]] = i;
        }
        TreeNode* root =
            create_tree(preorder, inorder, inorder_index, 0,
                       preorder.size() - 1, 0, inorder.size() - 1);

        return root;
    }

private:
    TreeNode* create_tree(vector<int>& preorder, vector<int>& inorder,
                          unordered_map<int, int>& inorder_index, int pre_start,
                          int pre_end, int in_start, int in_end) {
        if (pre_start > pre_end || in_start > in_end) {
            return nullptr;
        }
        TreeNode* root_node = new TreeNode(preorder[pre_start]);
        int root_inorder_index = inorder_index[preorder[pre_start]];

        int left_inorder_start = in_start; // 
        int left_inorder_end = root_inorder_index - 1; //
        int left_inorder_size = left_inorder_end - left_inorder_start + 1;
        int left_preorder_start = pre_start + 1; //
        int left_preorder_end = left_preorder_start + left_inorder_size - 1;

        int right_inorder_start = root_inorder_index + 1; //
        int right_inorder_end = in_end; //
        int right_preorder_start = left_preorder_end + 1;
        int right_preorder_end = pre_end;

        root_node->left = create_tree(preorder, inorder, inorder_index,
                                       left_preorder_start, left_preorder_end,
                                       left_inorder_start, left_inorder_end);

        root_node->right = create_tree(preorder, inorder, inorder_index,
                                       right_preorder_start, right_preorder_end,
                                       right_inorder_start, right_inorder_end);
        return root_node;
    }
};
```

[93 Restore IP Addresses](#) ([link](#))

Description

A **valid IP address** consists of exactly four integers separated by single dots. Each integer is between 0 and 255 (**inclusive**) and cannot have leading zeros.

- For example, "0.1.2.201" and "192.168.1.1" are **valid** IP addresses, but "0.011.255.245", "192.168.1.312" and "192.168@1.1" are **invalid** IP addresses.

Given a string s containing only digits, return *all possible valid IP addresses that can be formed by inserting dots into s* . You are **not** allowed to reorder or remove any digits in s . You may return the valid IP addresses in **any** order.

Example 1:

```
Input: s = "25525511135"
Output: ["255.255.11.135", "255.255.111.35"]
```

Example 2:

```
Input: s = "0000"
Output: ["0.0.0.0"]
```

Example 3:

```
Input: s = "101023"
Output: ["1.0.10.23", "1.0.102.3", "10.1.0.23", "10.10.2.3", "101.0.2.3"]
```

Constraints:

- $1 \leq s.length \leq 20$
- s consists of digits only.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    vector<string> restoreIpAddresses(string s) {
        vector<string> final_ans;
        int curr_index = 0, curr_segments = 0;
        string curr_str = "";

        solve(s, curr_index, curr_segments, curr_str, final_ans);
        return final_ans;
    }

private:
    void solve(string& s, int curr_index, int& curr_segments, string& curr_str,
               vector<string>& final_ans) {
        if (curr_segments >= 4 && curr_index >= s.size()) {
            final_ans.push_back(curr_str);
            return;
        }
        if (curr_segments >= 4 || curr_index >= s.size()) {
            return;
        }
        // Only 3 length each sub IP Address
        for (int i = curr_index; i < s.size() && i < curr_index+3; i++) {
            string ip_substr = s.substr(curr_index, i - curr_index + 1);

            if (is_valid(ip_substr)) {
                int size = curr_str.size();
                if (curr_str.size() > 0) {
                    curr_str += "." + ip_substr;
                } else {
                    curr_str += ip_substr;
                }
                curr_segments++; // This incremented regardless
                solve(s, i+1, curr_segments, curr_str, final_ans);
                curr_segments--;
                curr_str.erase(size);
            }
        }
    }

    bool is_valid(string& str) {
        if (str.size() > 1 && str[0] == '0')
            return false;
        if (stoi(str) < 0 || stoi(str) > 255)
            return false;

        return true;
    }
};
```

[89 Gray Code \(link\)](#)

Description

An **n-bit gray code sequence** is a sequence of 2^n integers where:

- Every integer is in the **inclusive** range $[0, 2^n - 1]$,
- The first integer is **0**,
- An integer appears **no more than once** in the sequence,
- The binary representation of every pair of **adjacent** integers differs by **exactly one bit**, and
- The binary representation of the **first** and **last** integers differs by **exactly one bit**.

Given an integer n , return *any valid n-bit gray code sequence*.

Example 1:

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

Explanation:

The binary representation of $[0, 1, 3, 2]$ is $[00, 01, 11, 10]$.

- 00 and 01 differ by one bit
- 01 and 11 differ by one bit
- 11 and 10 differ by one bit
- 10 and 00 differ by one bit

$[0, 2, 3, 1]$ is also a valid gray code sequence, whose binary representation is $[00, 10, 11, 01]$.

- 00 and 10 differ by one bit
- 10 and 11 differ by one bit
- 11 and 01 differ by one bit
- 01 and 00 differ by one bit

Example 2:

Input: $n = 1$
Output: $[0, 1]$

Constraints:

- $1 \leq n \leq 16$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    vector<int> grayCode(int n) {
        vector<int>ans;
        solve(n,ans);
        return ans;
    }

private:
    void solve(int n, vector<int>&ans){
        if(n<=1){
            ans.push_back(0);
            ans.push_back(1);
            return;
        }

        solve(n-1,ans);
        vector<int>copy(ans);
        reverse(copy.begin(),copy.end());
        for(auto &element: copy){
            element = element | 1 << (n-1);
        }
        for(auto &el:copy){
            ans.push_back(el);
        }
    }
};
```

[77 Combinations \(link\)](#)

Description

Given two integers n and k , return *all possible combinations of k numbers chosen from the range $[1, n]$.*

You may return the answer in **any order**.

Example 1:

Input: $n = 4$, $k = 2$

Output: $\{[1,2], [1,3], [1,4], [2,3], [2,4], [3,4]\}$

Explanation: There are 4 choose $2 = 6$ total combinations.

Note that combinations are unordered, i.e., $[1,2]$ and $[2,1]$ are considered to be the same comb:

Example 2:

Input: $n = 1$, $k = 1$

Output: $\{[1]\}$

Explanation: There is 1 choose $1 = 1$ total combination.

Constraints:

- $1 \leq n \leq 20$
- $1 \leq k \leq n$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    vector<vector<int>> combine(int n, int k) {
        vector<vector<int>>final_ans;
        vector<int>curr;

        find_combo(n,k,1,final_ans,curr);

        return final_ans;
    }
private:
    void find_combo(int n,int k, int index,vector<vector<int>>&final_ans,vector<int>&curr){
        if(curr.size() >= k ){
            final_ans.push_back(curr);
            return;
        }

        for(int i=index; i<=n; i++){
            curr.push_back(i);
            find_combo(n,k,i+1,final_ans,curr);
            curr.pop_back();
        }
    }
};
```

140 Word Break II ([link](#))

Description

Given a string s and a dictionary of strings wordDict , add spaces in s to construct a sentence where each word is a valid dictionary word. Return all such possible sentences in **any order**.

Note that the same word in the dictionary may be reused multiple times in the segmentation.

Example 1:

```
Input: s = "catsanddog", wordDict = ["cat","cats","and","sand","dog"]
Output: ["cats and dog","cat sand dog"]
```

Example 2:

```
Input: s = "pineapplepenapple", wordDict = ["apple","pen","applepen","pine","pineapple"]
Output: ["pine apple pen apple","pineapple pen apple","pine applepen apple"]
Explanation: Note that you are allowed to reuse a dictionary word.
```

Example 3:

```
Input: s = "catsandog", wordDict = ["cats","dog","sand","and","cat"]
Output: []
```

Constraints:

- $1 \leq s.length \leq 20$
- $1 \leq \text{wordDict.length} \leq 1000$
- $1 \leq \text{wordDict}[i].length \leq 10$
- s and $\text{wordDict}[i]$ consist of only lowercase English letters.
- All the strings of wordDict are **unique**.
- Input is generated in a way that the length of the answer doesn't exceed 10^5 .

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    vector<string> wordBreak(string s, vector<string>& wordDict) {
        vector<string>final_ans;
        unordered_set<string>wordDictSet(wordDict.begin(),wordDict.end());
        string curr_str = "";
        find_word_break(s,wordDictSet,0,curr_str,final_ans);
        return final_ans;
    }
private:
    void find_word_break(string s,unordered_set<string>& wordDict, int index,string& curr_str,\n        if(index >= s.size()){
            final_ans.push_back(curr_str);
            return;
        }
        for(int i=index;i <s.size(); i++){
            string curr_substr = s.substr(index,i-index+1);
            if(wordDict.count(curr_substr)){
                int curr_str_size = curr_str.size();
                curr_str.size() == 0 ? curr_str += curr_substr : curr_str += " " + curr_substr;
                find_word_break(s,wordDict,i+1,curr_str,final_ans);
                curr_str.erase(curr_str_size);
            }
        }
    };
};
```

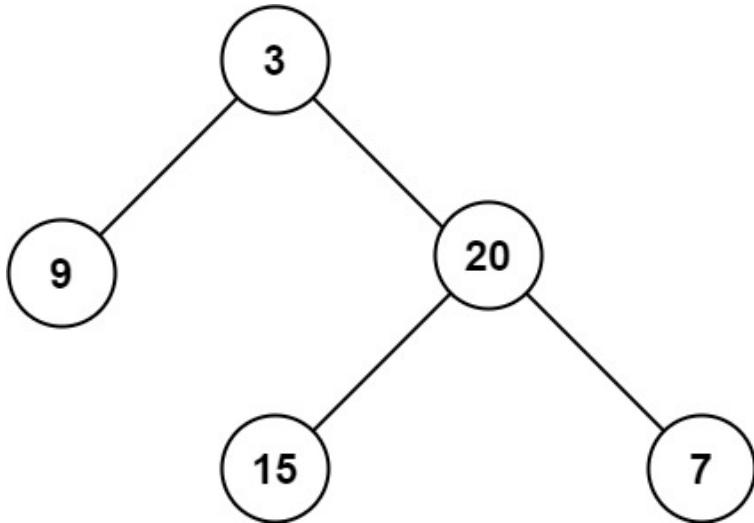
104 Maximum Depth of Binary Tree ([link](#))

Description

Given the root of a binary tree, return *its maximum depth*.

A binary tree's **maximum depth** is the number of nodes along the longest path from the root node down to the farthest leaf node.

Example 1:



```
Input: root = [3,9,20,null,null,15,7]
Output: 3
```

Example 2:

```
Input: root = [1,null,2]
Output: 2
```

Constraints:

- The number of nodes in the tree is in the range $[0, 10^4]$.
- $-100 \leq \text{Node.val} \leq 100$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * 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(right) {}
 * };
 */
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if(!root) return 0;
        int left_depth = maxDepth(root->left);
        int right_depth = maxDepth(root->right);

        return 1 + max(left_depth,right_depth);
    }
};
```

[526 Beautiful Arrangement \(link\)](#)

Description

Suppose you have n integers labeled 1 through n . A permutation of those n integers perm (**1-indexed**) is considered a **beautiful arrangement** if for every i ($1 \leq i \leq n$), **either** of the following is true:

- $\text{perm}[i]$ is divisible by i .
- i is divisible by $\text{perm}[i]$.

Given an integer n , return *the number of the beautiful arrangements that you can construct.*

Example 1:

Input: $n = 2$

Output: 2

Explanation:

The first beautiful arrangement is [1,2]:

- $\text{perm}[1] = 1$ is divisible by $i = 1$
- $\text{perm}[2] = 2$ is divisible by $i = 2$

The second beautiful arrangement is [2,1]:

- $\text{perm}[1] = 2$ is divisible by $i = 1$
- $i = 2$ is divisible by $\text{perm}[2] = 1$

Example 2:

Input: $n = 1$

Output: 1

Constraints:

- $1 \leq n \leq 15$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int countArrangement(int n) {
        int count = 0;
        vector<bool>visited(n+1, false); // n+1 because directly 1 based indexing used
        int curr_pos = 1;
        count_beauty(n, count, visited, curr_pos);
        return count;
    }

private:
    void count_beauty(int n, int &count, vector<bool>&visited, int curr_pos){
        if( curr_pos > n){
            count++;
            return ;
        }
        // curr_pos -> slot, i -> value at slot
        for(int i=1; i<=n; i++){
            if(visited[i])continue;
            visited[i]=1;
            if(( curr_pos % i == 0 ) || (i % curr_pos ==0)){
                count_beauty(n, count, visited, curr_pos+1);
            }
            visited[i]=0;
        }
    }
};
```

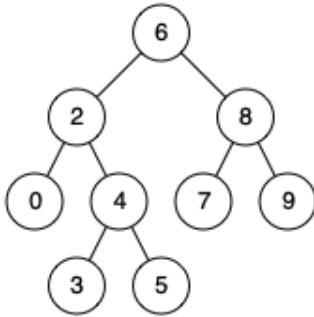
[235 Lowest Common Ancestor of a Binary Search Tree \(link\)](#)

Description

Given a binary search tree (BST), find the lowest common ancestor (LCA) node of two given nodes in the BST.

According to the [definition of LCA on Wikipedia](#): “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself).”

Example 1:

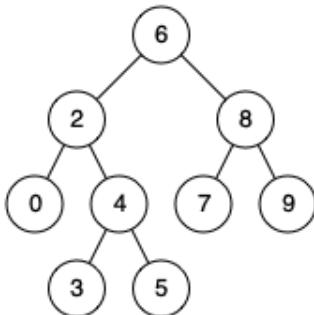


Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8

Output: 6

Explanation: The LCA of nodes 2 and 8 is 6.

Example 2:



Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4

Output: 2

Explanation: The LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself according to the problem statement.

Example 3:

Input: root = [2,1], p = 2, q = 1

Output: 2

Constraints:

- The number of nodes in the tree is in the range $[2, 10^5]$.
- $-10^9 \leq \text{Node.val} \leq 10^9$

- All `Node.val` are **unique**.
- $p \neq q$
- p and q will exist in the BST.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */

class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(!root) return root;

        if(p->val < root->val && q->val < root->val) return lowestCommonAncestor(root->left, p, q);
        if(p->val > root->val && q->val > root->val) return lowestCommonAncestor(root->right, p, q);
        else return root;
    }
};
```

[216 Combination Sum III \(link\)](#)

Description

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

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

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

Example 1:

```
Input: k = 3, n = 7
Output: [[1,2,4]]
Explanation:
1 + 2 + 4 = 7
There are no other valid combinations.
```

Example 2:

```
Input: k = 3, n = 9
Output: [[1,2,6],[1,3,5],[2,3,4]]
Explanation:
1 + 2 + 6 = 9
1 + 3 + 5 = 9
2 + 3 + 4 = 9
There are no other valid combinations.
```

Example 3:

```
Input: k = 4, n = 1
Output: []
Explanation: There are no valid combinations.
Using 4 different numbers in the range [1,9], the smallest sum we can get is 1+2+3+4 = 10 and s
```

Constraints:

- $2 \leq k \leq 9$
- $1 \leq n \leq 60$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    vector<vector<int>> combinationSum3(int k, int n) {
        vector<vector<int>> final_ans;
        vector<int> curr_combo;
        long long curr_sum = 0;
        solve(k, n, final_ans, curr_combo, curr_sum, 1);
        return final_ans;
    }

private:
    // k -> size, n -> target
    void solve(int k_size, int target_sum, vector<vector<int>>& final_ans,
               vector<int>& curr_combo, long long& curr_sum, int start_index) {
        if (curr_combo.size() >= k_size && curr_sum == target_sum) {
            final_ans.push_back(curr_combo);
            return;
        } else if (curr_combo.size() >= k_size && curr_sum != target_sum) {
            return;
        }

        for (int j = start_index; j <= 9; j++) {
            curr_sum += j;
            curr_combo.push_back(j);
            solve(k_size, target_sum, final_ans, curr_combo, curr_sum, j + 1);
            // Backtrack
            curr_sum -= j;
            curr_combo.pop_back();
        }
    }
};
```

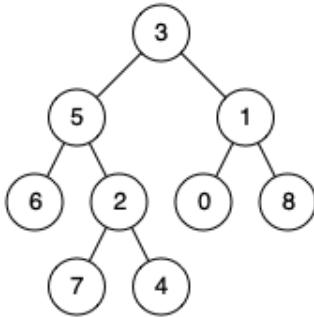
236 Lowest Common Ancestor of a Binary Tree (link)

Description

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

According to the [definition of LCA on Wikipedia](#): “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself).”

Example 1:

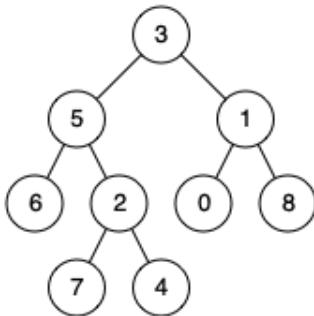


Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

Output: 3

Explanation: The LCA of nodes 5 and 1 is 3.

Example 2:



Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4

Output: 5

Explanation: The LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself according to the problem statement.

Example 3:

Input: root = [1,2], p = 1, q = 2
Output: 1

Constraints:

- The number of nodes in the tree is in the range $[2, 10^5]$.
- $-10^9 \leq \text{Node.val} \leq 10^9$
- All `Node.val` are **unique**.

- $p \neq q$
- p and q will exist in the tree.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(!root) return nullptr;
        if(root == p || root == q) return root;

        TreeNode* left_tree = lowestCommonAncestor(root->left,p,q);
        TreeNode* right_tree = lowestCommonAncestor(root->right,p,q);

        if(left_tree && right_tree) return root;
        if(left_tree && !right_tree){
            // In Left Part
            return left_tree;
        }
        if(right_tree && !left_tree){
            // in right part
            return right_tree;
        }

        return nullptr;
    }
};
```

[37 Sudoku Solver \(link\)](#)

Description

Write a program to solve a Sudoku puzzle by filling the empty cells.

A sudoku solution must satisfy **all** of the following rules:

1. Each of the digits 1–9 must occur exactly once in each row.
 2. Each of the digits 1–9 must occur exactly once in each column.
 3. Each of the digits 1–9 must occur exactly once in each of the 9 3×3 sub-boxes of the grid.

The '...' character indicates empty cells.

Example 1:

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

Input: board = [[5, 3, '.', '.', '7', '.', '.', '.', '9', '1', '5', '.', '.', '.', '.'], [6, '.', '.', '.', '8', '2', '1', '9', '5', '3', '4', '8', '1', '9', '2'], [1, '9', '5', '3', '4', '8', '2', '1', '9', '5', '3', '4', '8', '1', '9', '2']]
Output: [[5, 3, 4, 6, 7, 8, 9, 1, 2], [6, 7, 2, 1, 9, 5, 3, 4, 8], [1, 9, 5, 3, 4, 8, 2, 1, 9], [5, 3, 4, 8, 2, 1, 9, 5, 3], [4, 8, 1, 9, 2, 5, 3, 6, 7], [8, 1, 9, 2, 5, 3, 6, 7, 4], [2, 5, 3, 6, 7, 4, 8, 1, 9], [7, 4, 8, 1, 9, 2, 5, 3, 6], [9, 1, 2, 5, 3, 6, 7, 4, 8]]
Explanation: The input board is shown above and the only valid solution is shown below:

Explanation: The input board is shown above and the only valid solution is shown below:

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

Constraints:

- `board.length == 9`
 - `board[i].length == 9`
 - `board[i][j]` is a digit or '.'.
 - It is **guaranteed** that the input board has only one solution.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    void solveSudoku(vector<vector<char>>& board) {
        solve(board);
    }
private:
    bool solve(vector<vector<char>>&board){
        for(int i=0; i<9; i++){
            for(int j=0; j<9; j++){
                if(board[i][j]=='.'){
                    for(int k=1; k<=9; k++){
                        if(is_valid(board,i,j,k)){
                            board[i][j] = k + '0';
                            if(solve(board)){
                                return true;
                            }
                            board[i][j]='.';
                        }
                    }
                    return false; // invalid path
                }
            }
        }
        return true;
    }

    bool is_valid(vector<vector<char>>&board, int row, int col, int curr_digit){
        for(int i=0; i<9; i++){
            if(board[row][i]==curr_digit+'0')return false;
            if(board[i][col]==curr_digit+'0')return false;
            if(board[3* (row/3) + i/3][3* (col/3) + i%3]==curr_digit+'0')return false;
        }
        return true;
    }
};
```

[98 Validate Binary Search Tree \(link\)](#)

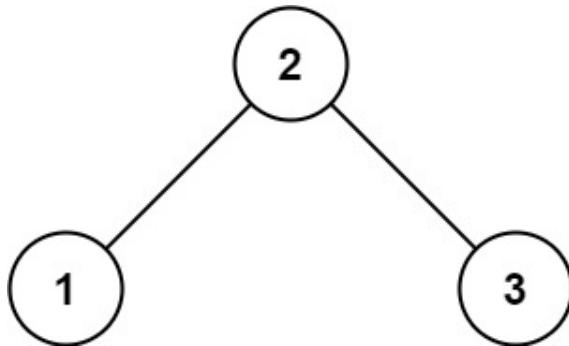
Description

Given the root of a binary tree, *determine if it is a valid binary search tree (BST)*.

A **valid BST** is defined as follows:

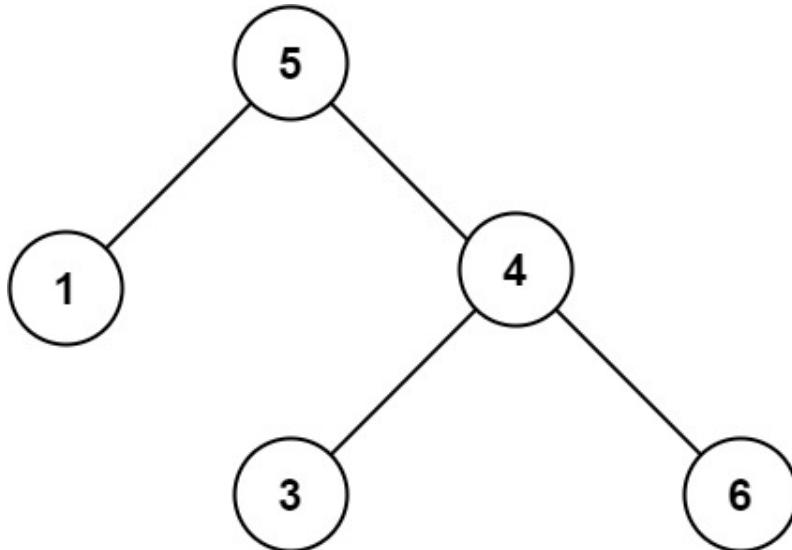
- The left subtree of a node contains only nodes with keys **strictly less than** the node's key.
- The right subtree of a node contains only nodes with keys **strictly greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.

Example 1:



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

Example 2:



```
Input: root = [5,1,4,null,null,3,6]
Output: false
Explanation: The root node's value is 5 but its right child's value is 4.
```

Constraints:

- The number of nodes in the tree is in the range $[1, 10^4]$.
- $-2^{31} \leq \text{Node.val} \leq 2^{31} - 1$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * 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(right) {}
 * };
 */
class Solution {
public:
    bool isValidBST(TreeNode* root) {
        TreeNode* prev = nullptr;
        return calculate_inorder(root, prev);
    }
private:
    bool calculate_inorder(TreeNode*& root, TreeNode*& prev){
        if(!root) return true;
        bool left = calculate_inorder(root->left, prev);
        if(prev && prev->val >= root->val) return false;
        prev = root;
        bool right = calculate_inorder(root->right, prev);
        return left & right;
    }
};
```

[3946 Find Maximum Balanced XOR Subarray Length \(link\)](#)

Description

Given an integer array `nums`, return the **length** of the **longest subarray** that has a bitwise XOR of zero and contains an **equal** number of **even** and **odd** numbers. If no such subarray exists, return 0.

Example 1:

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

Output: 4

Explanation:

The subarray `[1, 3, 2, 0]` has bitwise XOR $1 \text{ XOR } 3 \text{ XOR } 2 \text{ XOR } 0 = 0$ and contains 2 even and 2 odd numbers.

Example 2:

Input: `nums = [3,2,8,5,4,14,9,15]`

Output: 8

Explanation:

The whole array has bitwise XOR 0 and contains 4 even and 4 odd numbers.

Example 3:

Input: `nums = [0]`

Output: 0

Explanation:

No non-empty subarray satisfies both conditions.

Constraints:

- $1 \leq \text{nums.length} \leq 10^5$
- $0 \leq \text{nums}[i] \leq 10^9$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int maxBalancedSubarray(vector<int>& nums) {
        map<pair<int, int>, int> fo;
        fo[{0,0}]=-1;
        int cx = 0, cb=0, ml=0;

        for(int i=0; i<nums.size(); i++){
            cx += nums[i];
            if(nums[i] % 2 != 0) cb+=1;
            else cb-=1;

            if(fo.count({cx,cb})){
                int prev = fo[{cx,cb}];
                ml = max(ml, i-prev);
            }else fo[{cx,cb}] = i;
        }

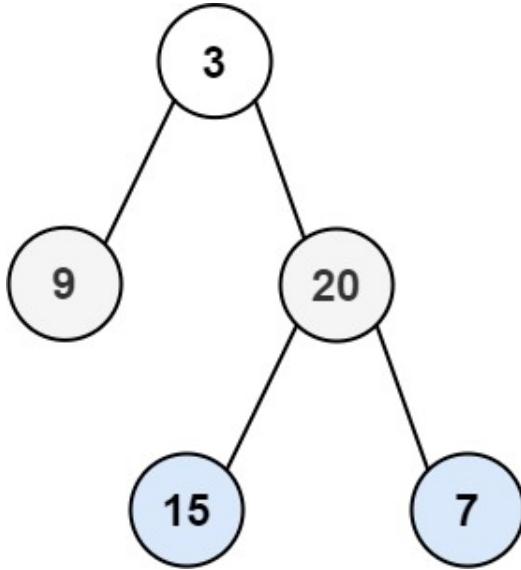
        return ml;
    }
};
```

[102 Binary Tree Level Order Traversal \(link\)](#)

Description

Given the root of a binary tree, return *the level order traversal of its nodes' values*. (i.e., from left to right, level by level).

Example 1:



```
Input: root = [3,9,20,null,null,15,7]
Output: [[3],[9,20],[15,7]]
```

Example 2:

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

Example 3:

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

Constraints:

- The number of nodes in the tree is in the range $[0, 2000]$.
- $-1000 \leq \text{Node.val} \leq 1000$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * 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(right) {}
 * };
 */
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>>ans;
        if(!root) return ans;
        queue<TreeNode*>queue;
        queue.push(root);
        while(!queue.empty()){
            int size = queue.size();
            vector<int>curr;
            for(int i=0; i<size; i++){
                TreeNode* front_node = queue.front();queue.pop();
                curr.push_back(front_node->val);
                if(front_node->left)queue.push(front_node->left);
                if(front_node->right)queue.push(front_node->right);
            }
            ans.push_back(curr);
        }
        return ans;
    }
};
```

[1331 Path with Maximum Gold \(link\)](#)

Description

In a gold mine grid of size $m \times n$, each cell in this mine has an integer representing the amount of gold in that cell, 0 if it is empty.

Return the maximum amount of gold you can collect under the conditions:

- Every time you are located in a cell you will collect all the gold in that cell.
- From your position, you can walk one step to the left, right, up, or down.
- You can't visit the same cell more than once.
- Never visit a cell with 0 gold.
- You can start and stop collecting gold from **any** position in the grid that has some gold.

Example 1:

Input: grid = [[0,6,0],[5,8,7],[0,9,0]]

Output: 24

Explanation:

```
[[0,6,0],  
 [5,8,7],  
 [0,9,0]]
```

Path to get the maximum gold, 9 → 8 → 7.

Example 2:

Input: grid = [[1,0,7],[2,0,6],[3,4,5],[0,3,0],[9,0,20]]

Output: 28

Explanation:

```
[[1,0,7],  
 [2,0,6],  
 [3,4,5],  
 [0,3,0],  
 [9,0,20]]
```

Path to get the maximum gold, 1 → 2 → 3 → 4 → 5 → 6 → 7.

Constraints:

- $m == \text{grid.length}$
- $n == \text{grid[i].length}$
- $1 \leq m, n \leq 15$
- $0 \leq \text{grid}[i][j] \leq 100$
- There are at most **25** cells containing gold.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int getMaximumGold(vector<vector<int>>& grid) {
        int row_size = grid.size(), col_size = grid[0].size();
        vector<vector<bool>> visited(row_size, vector<bool>(col_size, false));
        int max_sum = INT_MIN, curr_sum = 0;
        for(int i=0; i<row_size; i++) {
            for(int j=0; j<col_size; j++) {
                if(grid[i][j]==0) continue;
                solve(grid, visited, i, j, curr_sum, max_sum);
            }
        }
        return max_sum == INT_MIN ? 0 : max_sum;
    }

private:
    void solve(vector<vector<int>>&grid, vector<vector<bool>>&visited, int row, int col, int &curr_sum) {
        if(row < 0 || col < 0 || row >= grid.size() || col >= grid[0].size()){
            return;
        }
        if(visited[row][col] == true || grid[row][col] == 0) return;

        // DLRU
        vector<int> move_row = {1,0,0,-1};
        vector<int> move_col = {0,-1,1,0};

        for(int i=0; i<4; i++){
            visited[row][col] = true;
            curr_sum += grid[row][col];
            max_sum = max(max_sum, curr_sum);
            solve(grid, visited, row+move_row[i], col + move_col[i], curr_sum, max_sum);
            curr_sum -= grid[row][col];
            visited[row][col] = false;
        }
    }
};
```

[4134 Number of Effective Subsequences \(link\)](#)

Description

You are given an integer array `nums`.

The **strength** of the array is defined as the **bitwise OR** of all its elements.

A **subsequence** is considered **effective** if removing that subsequence **strictly decreases** the strength of the remaining elements.

Return the number of **effective subsequences** in `nums`. Since the answer may be large, return it **modulo** $10^9 + 7$.

The bitwise OR of an empty array is 0.

Example 1:

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

Output: 3

Explanation:

- The Bitwise OR of the array is $1 \text{ OR } 2 \text{ OR } 3 = 3$.
- Subsequences that are effective are:
 - $[1, 3]$: The remaining element $[2]$ has a Bitwise OR of 2.
 - $[2, 3]$: The remaining element $[1]$ has a Bitwise OR of 1.
 - $[1, 2, 3]$: The remaining elements $[]$ have a Bitwise OR of 0.
- Thus, the total number of effective subsequences is 3.

Example 2:

Input: `nums = [7,4,6]`

Output: 4

Explanation:

- The Bitwise OR of the array is $7 \text{ OR } 4 \text{ OR } 6 = 7$.
- Subsequences that are effective are:
 - $[7]$: The remaining elements $[4, 6]$ have a Bitwise OR of 6.
 - $[7, 4]$: The remaining element $[6]$ has a Bitwise OR of 6.
 - $[7, 6]$: The remaining element $[4]$ has a Bitwise OR of 4.
 - $[7, 4, 6]$: The remaining elements $[]$ have a Bitwise OR of 0.
- Thus, the total number of effective subsequences is 4.

Example 3:

Input: `nums = [8,8]`

Output: 1

Explanation:

- The Bitwise OR of the array is $8 \text{ OR } 8 = 8$.
- Only the subsequence $[8, 8]$ is effective since removing it leaves $[]$ which has a Bitwise OR of 0.
- Thus, the total number of effective subsequences is 1.

Example 4:

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

Output: 5

Explanation:

- The Bitwise OR of the array is $2 \text{ OR } 2 \text{ OR } 1 = 3$.
- Subsequences that are effective are:
 - [1]: The remaining elements [2, 2] have a Bitwise OR of 2.
 - [2, 1] (using `nums[0]`, `nums[2]`): The remaining element [2] has a Bitwise OR of 2.
 - [2, 1] (using `nums[1]`, `nums[2]`): The remaining element [2] has a Bitwise OR of 2.
 - [2, 2]: The remaining element [1] has a Bitwise OR of 1.
 - [2, 2, 1]: The remaining elements [] have a Bitwise OR of 0.
- Thus, the total number of effective subsequences is 5.

Constraints:

- $1 \leq \text{nums.length} \leq 10^5$
- $1 \leq \text{nums}[i] \leq 10^6$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int countEffective(vector<int>& nums) {
        long long MOD = 1e9+7;
        int n = nums.size(), s=0;
        for(auto num:nums)s = s | num;

        int nb = 0;
        while((1<<nb)<=s)nb++;

        int limit = 1<<nb;
        vector<long long> fi(limit,0);

        for(auto x:nums)fi[x]++;
        for(int i=0; i<nb; i++){
            for(int j=0; j<limit; j++){
                if(j & (1<<i))fi[j] += fi[j^(1<<i)];
            }
        }

        vector<long long> power_2(n+1);
        power_2[0]=1;
        for(int i=1; i<n+1;i++)power_2[i] = (power_2[i-1]*2)%MOD;

        long long count = 0;
        int pops = __builtin_popcount(s);

        for(int i=0; i<s+1; i++){
            if((i | s)==s){
                int pop = __builtin_popcount(i);
                long long difference = pops - pop;

                long long paths = power_2[fi[i]];

                if(difference % 2 == 1)count = (count - paths + MOD) % MOD;
                else count = (count+paths) % MOD;
            }
        }

        long long all_sets = power_2[nums.size()],final_ans = (all_sets - count + MOD)%MOD;
        return final_ans;
    }
};
```

[2662 Check Knight Tour Configuration \(link\)](#)

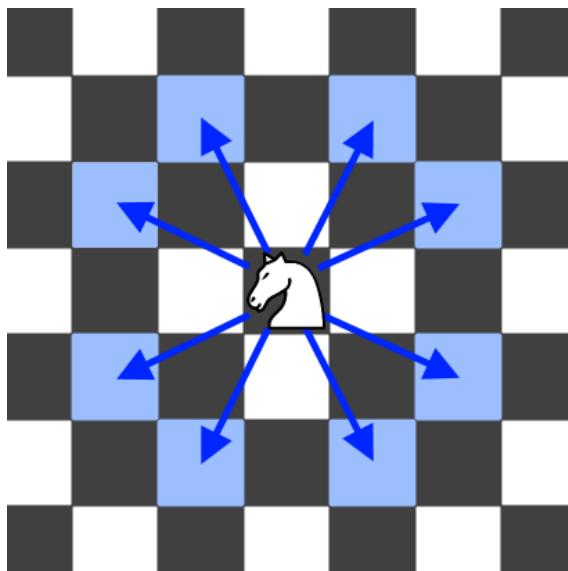
Description

There is a knight on an $n \times n$ chessboard. In a valid configuration, the knight starts **at the top-left cell** of the board and visits every cell on the board **exactly once**.

You are given an $n \times n$ integer matrix `grid` consisting of distinct integers from the range $[0, n * n - 1]$ where `grid[row][col]` indicates that the cell (row, col) is the `grid[row][col]th` cell that the knight visited. The moves are **0-indexed**.

Return `true` if `grid` represents a valid configuration of the knight's movements or `false` otherwise.

Note that a valid knight move consists of moving two squares vertically and one square horizontally, or two squares horizontally and one square vertically. The figure below illustrates all the possible eight moves of a knight from some cell.



Example 1:

0	11	16	5	20
17	4	19	10	15
12	1	8	21	6
3	18	23	14	9
24	13	2	7	22

Input: `grid = [[0,11,16,5,20],[17,4,19,10,15],[12,1,8,21,6],[3,18,23,14,9],[24,13,2,7,22]]`
Output: `true`

Explanation: The above diagram represents the grid. It can be shown that it is a valid configuration.

Example 2:

0	3	6
5	8	1
2	7	4

Input: grid = [[0,3,6],[5,8,1],[2,7,4]]

Output: false

Explanation: The above diagram represents the grid. The 8th move of the knight is not valid con

Constraints:

- $n == \text{grid.length} == \text{grid[i].length}$
- $3 \leq n \leq 7$
- $0 \leq \text{grid[row][col]} < n * n$
- All integers in grid are **unique**.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    bool checkValidGrid(vector<vector<int>>& grid) {
        bool is_possible = false;
        int n = grid.size();
        solve(grid, is_possible, 0, 0, 0, n);
        return is_possible;
    }

private:
    void solve(vector<vector<int>>&grid, bool &is_possible, int index, int row, int col, int n){
        if(row < 0 || row >= n || col < 0 || col >= n){
            return;
        }
        if(grid[row][col] != index){
            return;
        }
        if(index >= n*n-1){
            is_possible = true;
            return;
        }
        vector<int>move_row = {-2,-2,-1,1,2,2,1,-1};
        vector<int>move_col = {-1,1,2,2,1,-1,-2,-2};

        for(int i=0; i<8; i++){
            solve(grid, is_possible, index+1, row + move_row[i], col + move_col[i], n);
        }
    };
};
```

[4136 Concatenate Non-Zero Digits and Multiply by Sum II \(link\)](#)

Description

You are given a string s of length m consisting of digits. You are also given a 2D integer array queries , where $\text{queries}[i] = [l_i, r_i]$.

For each $\text{queries}[i]$, extract the **substring** $s[l_i..r_i]$. Then, perform the following:

- Form a new integer x by concatenating all the **non-zero digits** from the substring in their original order. If there are no non-zero digits, $x = 0$.
- Let sum be the **sum of digits** in x . The answer is $x * \text{sum}$.

Return an array of integers answer where $\text{answer}[i]$ is the answer to the i^{th} query.

Since the answers may be very large, return them **modulo** $10^9 + 7$.

Example 1:

Input: $s = "10203004"$, $\text{queries} = [[0,7],[1,3],[4,6]]$

Output: [12340, 4, 9]

Explanation:

- $s[0..7] = "10203004"$
 - $x = 1234$
 - $\text{sum} = 1 + 2 + 3 + 4 = 10$
 - Therefore, answer is $1234 * 10 = 12340$.
- $s[1..3] = "020"$
 - $x = 2$
 - $\text{sum} = 2$
 - Therefore, the answer is $2 * 2 = 4$.
- $s[4..6] = "300"$
 - $x = 3$
 - $\text{sum} = 3$
 - Therefore, the answer is $3 * 3 = 9$.

Example 2:

Input: $s = "1000"$, $\text{queries} = [[0,3],[1,1]]$

Output: [1, 0]

Explanation:

- $s[0..3] = "1000"$
 - $x = 1$
 - $\text{sum} = 1$
 - Therefore, the answer is $1 * 1 = 1$.
- $s[1..1] = "0"$
 - $x = 0$
 - $\text{sum} = 0$
 - Therefore, the answer is $0 * 0 = 0$.

Example 3:

Input: $s = "9876543210"$, $\text{queries} = [[0,9]]$

Output: [444444137]

Explanation:

- $s[0..9] = "9876543210"$
 - $x = 987654321$
 - $\text{sum} = 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = 45$
 - Therefore, the answer is $987654321 * 45 = 44444444445$.
 - We return $4444444445 \bmod (10^9 + 7) = 444444137$.

Constraints:

- $1 \leq m == s.length \leq 10^5$
- s consists of digits only.
- $1 \leq \text{queries.length} \leq 10^5$
- $\text{queries}[i] = [l_i, r_i]$
- $0 \leq l_i \leq r_i < m$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    vector<int> sumAndMultiply(string s, vector<vector<int>>& queries) {
        int m = s.length();
        long long MOD = 1e9+7;

        vector<long long> ps(m+1,0), pc(m+1,0), pv(m+1,0), pow(m+1,1);
        vector<int> result;

        for(int i=0; i<m; i++){
            int d = s[i] - '0';
            ps[i+1] = ps[i] + d;

            if(d!= 0){
                pc[i+1] = pc[i]+1;
                pv[i+1] = (pv[i]*10+d)%MOD;
            }
            else{
                pc[i+1] = pc[i];
                pv[i+1] = pv[i];
            }
            pow[i+1] = (pow[i]*10)%MOD;
        }

        for(auto q:queries){
            int li = q[0], ri=q[1];
            long long cs = ps[ri+1]-ps[li], cnt = pc[ri+1]-pc[li];
            if(cnt == 0){
                result.push_back(0);continue;
            }

            long long t = (pv[li]*pow[cnt])%MOD;
            long long cx = (pv[ri+1]-t + MOD)%MOD;
            long long ans = (cx*(cs%MOD))%MOD;
            result.push_back((int)ans);
        }
        return result;
    }
};
```

[4135 Concatenate Non-Zero Digits and Multiply by Sum I \(link\)](#)

Description

You are given an integer n .

Form a new integer x by concatenating all the **non-zero digits** of n in their original order. If there are no **non-zero** digits, $x = 0$.

Let sum be the **sum of digits** in x .

Return an integer representing the value of $x * \text{sum}$.

Example 1:

Input: $n = 10203004$

Output: 12340

Explanation:

- The non-zero digits are 1, 2, 3, and 4. Thus, $x = 1234$.
- The sum of digits is $\text{sum} = 1 + 2 + 3 + 4 = 10$.
- Therefore, the answer is $x * \text{sum} = 1234 * 10 = 12340$.

Example 2:

Input: $n = 1000$

Output: 1

Explanation:

- The non-zero digit is 1, so $x = 1$ and $\text{sum} = 1$.
- Therefore, the answer is $x * \text{sum} = 1 * 1 = 1$.

Constraints:

- $0 \leq n \leq 10^9$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    long long sumAndMultiply(int n) {
        vector<int> digits;
        long long sum = 0;
        string non_zero = "";
        while(n!=0){
            if(n%10 != 0){
                sum += n%10;
                non_zero += to_string(n%10);
            }
            n/=10;
        }
        if(non_zero.empty())return 0;
        reverse(non_zero.begin(),non_zero.end());
        return sum * stoll(non_zero);
    }
};
```

[282 Expression Add Operators \(link\)](#)

Description

Given a string `num` that contains only digits and an integer `target`, return ***all possibilities to insert the binary operators '+', '-' , and/or '*' between the digits of num so that the resultant expression evaluates to the target value.***

Note that operands in the returned expressions **should not** contain leading zeros.

Note that a number can contain multiple digits.

Example 1:

```
Input: num = "123", target = 6
Output: ["1*2*3", "1+2+3"]
Explanation: Both "1*2*3" and "1+2+3" evaluate to 6.
```

Example 2:

```
Input: num = "232", target = 8
Output: ["2*3+2", "2+3*2"]
Explanation: Both "2*3+2" and "2+3*2" evaluate to 8.
```

Example 3:

```
Input: num = "3456237490", target = 9191
Output: []
Explanation: There are no expressions that can be created from "3456237490" to evaluate to 9191
```

Constraints:

- $1 \leq \text{num.length} \leq 10$
- `num` consists of only digits.
- $-2^{31} \leq \text{target} \leq 2^{31} - 1$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    vector<string> addOperators(string num, int target) {
        vector<string> ans;
        string curr_str = "";
        long long curr_value = 0;
        long long last_operand = 0;
        solve(num, target, 0, ans, curr_str, curr_value, last_operand);

        return ans;
    }

private:
    void solve(string& nums, int target, int index, vector<string>& ans,
               string curr_str, long long curr_value, long long last_operand) {
        if (index >= nums.size()) {
            if (curr_value == target) {
                ans.push_back(curr_str);
            }
            return;
        }

        for (int i = index; i < nums.size(); i++) {
            if (i > index && nums[index] == '0') {
                break;
            }
            string curr_substr = nums.substr(index, i - index + 1);
            long long curr_digit = stoll(curr_substr);

            if (index == 0) {
                // Don't Add Operator at beginning
                solve(nums, target, i + 1, ans, curr_str + curr_substr,
                      curr_value + curr_digit, curr_digit);

            } else {
                // ADD
                solve(nums, target, i + 1, ans, curr_str + "+" + curr_substr,
                      curr_value + curr_digit, curr_digit);
                // Subtract
                solve(nums, target, i + 1, ans, curr_str + "-" + curr_substr,
                      curr_value - curr_digit, -curr_digit);

                // Multiply (UNDO & MULTIPLY)
                solve(nums, target, i + 1, ans, curr_str + "*" + curr_substr,
                      (curr_value - last_operand) + (last_operand * curr_digit),
                      curr_digit * last_operand);
            }
        }
    }
};
```

[4128 Total Waviness of Numbers in Range II \(link\)](#)

Description

You are given two integers `num1` and `num2` representing an **inclusive** range `[num1, num2]`.

The **waviness** of a number is defined as the total count of its **peaks** and **valleys**:

- A digit is a **peak** if it is **strictly greater** than both of its immediate neighbors.
- A digit is a **valley** if it is **strictly less** than both of its immediate neighbors.
- The first and last digits of a number **cannot** be peaks or valleys.
- Any number with fewer than 3 digits has a waviness of 0.

Return the total sum of waviness for all numbers in the range `[num1, num2]`.

Example 1:

Input: `num1 = 120, num2 = 130`

Output: 3

Explanation:

In the range `[120, 130]`:

- 120: middle digit 2 is a peak, waviness = 1.
- 121: middle digit 2 is a peak, waviness = 1.
- 130: middle digit 3 is a peak, waviness = 1.
- All other numbers in the range have a waviness of 0.

Thus, total waviness is $1 + 1 + 1 = 3$.

Example 2:

Input: `num1 = 198, num2 = 202`

Output: 3

Explanation:

In the range `[198, 202]`:

- 198: middle digit 9 is a peak, waviness = 1.
- 201: middle digit 0 is a valley, waviness = 1.
- 202: middle digit 0 is a valley, waviness = 1.
- All other numbers in the range have a waviness of 0.

Thus, total waviness is $1 + 1 + 1 = 3$.

Example 3:

Input: `num1 = 4848, num2 = 4848`

Output: 2

Explanation:

Number 4848: the second digit 8 is a peak, and the third digit 4 is a valley, giving a waviness of 2.

Constraints:

- $1 \leq \text{num1} \leq \text{num2} \leq 10^{15}$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    long long totalWaviness(long long num1, long long num2) {
        long long answer = solve(num2) - solve(num1-1);
        return answer;
    }
private:
    string s;
    struct Result{
        long long count,wavy;
    }dp[17][11][4][2][2];

    Result dp_function(int index,int prev,int tr, bool ti, bool le){
        if(index == s.size())return {1,0};
        if(dp[index][prev][tr][ti][le].count != -1)
            return dp[index][prev][tr][ti][le];

        long long tc = 0,tw=0,limit = ti ?( s[index] - '0'):9;

        for(int d = 0; d<= limit; d++){
            bool nt = ti && (d == limit),nl = le && !d;

            if(nl){
                Result res_obj = dp_function(index+1,10,0,nt,1);
                tc += res_obj.count;
                tw += res_obj.wavy;
            }else{
                int ntr = 0;
                int peak_or_valley = 0;
                if(prev != 10){
                    if(prev > d)ntr = 2;
                    else if(prev < d)ntr = 1;
                    else ntr= 3;
                }

                if(tr == 1 && prev > d || tr == 2 && prev < d)peak_or_valley = 1;
            }

            Result res_obj = dp_function(index+1,d,ntr,nt,0);
            tc += res_obj.count;
            tw += res_obj.wavy + (res_obj.count * peak_or_valley);
        }
        return dp[index][prev][tr][ti][le] = {tc,tw};
    }
    long long solve(long long n){
        if(n<=0) return 0;
        s = to_string(n);
        for(int i=0; i<17;i++)
            for(int j=0; j<11; j++)
                for(int k=0; k<4; k++)
                    for(int l=0; l<2; l++)
                        for(int m=0; m<2; m++)
                            dp[i][j][k][l][m]={-1,-1};

        return dp_function(0,10,0,1,1).wavy;
    }
};
```

4077 Lexicographically Smallest Negated Permutation that Sums to Target (link)

Description

You are given a positive integer n and an integer target .

Return the **lexicographically smallest** array of integers of size n such that:

- The **sum** of its elements equals target .
- The **absolute values** of its elements form a **permutation** of size n .

If no such array exists, return an empty array.

A **permutation** of size n is a rearrangement of integers $1, 2, \dots, n$.

Example 1:

Input: $n = 3$, $\text{target} = 0$

Output: $[-3, 1, 2]$

Explanation:

The arrays that sum to 0 and whose absolute values form a permutation of size 3 are:

- $[-3, 1, 2]$
- $[-3, 2, 1]$
- $[-2, -1, 3]$
- $[-2, 3, -1]$
- $[-1, -2, 3]$
- $[-1, 3, -2]$
- $[1, -3, 2]$
- $[1, 2, -3]$
- $[2, -3, 1]$
- $[2, 1, -3]$
- $[3, -2, -1]$
- $[3, -1, -2]$

The lexicographically smallest one is $[-3, 1, 2]$.

Example 2:

Input: $n = 1$, $\text{target} = 10000000000$

Output: $[]$

Explanation:

There are no arrays that sum to 10000000000 and whose absolute values form a permutation of size 1. Therefore, the answer is $[]$.

Constraints:

- $1 \leq n \leq 10^5$
- $-10^{10} \leq \text{target} \leq 10^{10}$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    vector<int> lexSmallestNegatedPerm(int n, long long target) {
        long long ts = (long long) n*(n+1)/2, diff = ts - target;
        if(diff % 2 != 0 || diff < 0) return {};
        long long k = diff/2;
        vector<int> res;
        vector<bool> is_neg(n+1, 0);
        for(int i=n; i>0; i--){
            if(k>=i){
                k-=i;
                is_neg[i] = 1;
                res.push_back(-i);
            }
        }
        if(k>0) return {};
        for(int i=1; i<n+1; i++){
            if(!is_neg[i]) res.push_back(i);
        }
        return res;
    }
};
```

[4057 Total Waviness of Numbers in Range I \(link\)](#)

Description

You are given two integers `num1` and `num2` representing an **inclusive** range `[num1, num2]`.

The **waviness** of a number is defined as the total count of its **peaks** and **valleys**:

- A digit is a **peak** if it is **strictly greater** than both of its immediate neighbors.
- A digit is a **valley** if it is **strictly less** than both of its immediate neighbors.
- The first and last digits of a number **cannot** be peaks or valleys.
- Any number with fewer than 3 digits has a waviness of 0.

Return the total sum of waviness for all numbers in the range `[num1, num2]`.

Example 1:

Input: `num1 = 120, num2 = 130`

Output: 3

Explanation:

In the range `[120, 130]`:

- 120: middle digit 2 is a peak, waviness = 1.
- 121: middle digit 2 is a peak, waviness = 1.
- 130: middle digit 3 is a peak, waviness = 1.
- All other numbers in the range have a waviness of 0.

Thus, total waviness is $1 + 1 + 1 = 3$.

Example 2:

Input: `num1 = 198, num2 = 202`

Output: 3

Explanation:

In the range `[198, 202]`:

- 198: middle digit 9 is a peak, waviness = 1.
- 201: middle digit 0 is a valley, waviness = 1.
- 202: middle digit 0 is a valley, waviness = 1.
- All other numbers in the range have a waviness of 0.

Thus, total waviness is $1 + 1 + 1 = 3$.

Example 3:

Input: `num1 = 4848, num2 = 4848`

Output: 2

Explanation:

Number 4848: the second digit 8 is a peak, and the third digit 4 is a valley, giving a waviness of 2.

Constraints:

- $1 \leq \text{num1} \leq \text{num2} \leq 10^5$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int totalWaviness(int num1, int num2) {
        int peak = 0, valley = 0, wavy = 0;

        for(int i=num1; i<=num2; i++){
            wavy += find_peak_valley(i);
        }
        return wavy;
    }
private:
    int find_peak_valley(int num){
        vector<int>peak_arr;
        int count_peak = 0, count_valley=0;
        while(num!=0){
            peak_arr.push_back(num%10);
            num/=10;
        }

        for(int i=1;i<peak_arr.size()-1; i++){
            if(peak_arr[i] > peak_arr[i-1] && peak_arr[i] > peak_arr[i+1])count_peak++;
            else if(peak_arr[i] < peak_arr[i-1] && peak_arr[i] < peak_arr[i+1])count_valley++;
        }
        return count_peak + count_valley;
    }
};
```

[4126 Minimum Number of Flips to Reverse Binary String \(link\)](#)

Description

You are given a **positive** integer n .

Let s be the **binary representation** of n without leading zeros.

The **reverse** of a binary string s is obtained by writing the characters of s in the opposite order.

You may flip any bit in s (change $0 \rightarrow 1$ or $1 \rightarrow 0$). Each flip affects **exactly** one bit.

Return the **minimum** number of flips required to make s equal to the reverse of its original form.

Example 1:

Input: $n = 7$

Output: 0

Explanation:

The binary representation of 7 is "111". Its reverse is also "111", which is the same. Hence, no flips are needed.

Example 2:

Input: $n = 10$

Output: 4

Explanation:

The binary representation of 10 is "1010". Its reverse is "0101". All four bits must be flipped to make them equal. Thus, the minimum number of flips required is 4.

Constraints:

- $1 \leq n \leq 10^9$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int minimumFlips(int n) {
        string reversed = "";
        while(n!=0){
            string rem = to_string(n%2);
            n/=2;
            reversed += rem;
        }
        string original = reversed;
        reverse(original.begin(),original.end());
        int ans = 0;
        for(int i = 0; i<reversed.size(); i++){
            if(reversed[i] != original[i])ans++;
        }
        return ans;
    }
};
```

131 Palindrome Partitioning (link)

Description

Given a string s , partition s such that every substring of the partition is a **palindrome**. Return *all possible palindrome partitioning of s* .

Example 1:

```
Input: s = "aab"
Output: [[["a","a","b"], ["aa","b"]]]
```

Example 2:

```
Input: s = "a"
Output: [[["a"]]]
```

Constraints:

- $1 \leq s.length \leq 16$
- s contains only lowercase English letters.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    vector<vector<string>> partition(string s) {
        vector<vector<string>> final_ans;
        vector<string> curr_string;

        find_palindrome_partition(s, 0, curr_string, final_ans);

        return final_ans;
    }

private:
    void find_palindrome_partition(string s, int index, vector<string>& curr_string, vector<vector<string>>& final_ans) {
        if(index >= s.size()){
            final_ans.push_back(curr_string);
            return;
        }

        for(int i=index; i<s.size(); i++){
            if(is_palindrome(s, index, i)){
                curr_string.push_back(s.substr(index, i-index+1));
                find_palindrome_partition(s, i+1, curr_string, final_ans);
                curr_string.pop_back();
            }
        }
    }

    bool is_palindrome(string s, int start, int end){
        while(start < end){
            if(s[start] != s[end]){
                return false;
            }
            start++;end--;
        }
        return true;
    }
};
```

[4110 Count Stable Subarrays \(link\)](#)

Description

You are given an integer array `nums`.

A **subarray** of `nums` is called **stable** if it contains **no inversions**, i.e., there is no pair of indices $i < j$ such that `nums[i] > nums[j]`.

You are also given a **2D integer array** `queries` of length q , where each `queries[i] = [li, ri]` represents a query. For each query $[l_i, r_i]$, compute the number of **stable subarrays** that lie entirely within the segment `nums[li..ri]`.

Return an integer array `ans` of length q , where `ans[i]` is the answer to the i^{th} query.

Note:

- A single element subarray is considered stable.

Example 1:

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

Output: `[2,3,4]`

Explanation:

- For `queries[0] = [0, 1]`, the subarray is `[nums[0], nums[1]] = [3, 1]`.
 - The stable subarrays are `[3]` and `[1]`. The total number of stable subarrays is 2.
- For `queries[1] = [1, 2]`, the subarray is `[nums[1], nums[2]] = [1, 2]`.
 - The stable subarrays are `[1]`, `[2]`, and `[1, 2]`. The total number of stable subarrays is 3.
- For `queries[2] = [0, 2]`, the subarray is `[nums[0], nums[1], nums[2]] = [3, 1, 2]`.
 - The stable subarrays are `[3]`, `[1]`, `[2]`, and `[1, 2]`. The total number of stable subarrays is 4.

Thus, `ans = [2, 3, 4]`.

Example 2:

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

Output: `[3,1]`

Explanation:

- For `queries[0] = [0, 1]`, the subarray is `[nums[0], nums[1]] = [2, 2]`.
 - The stable subarrays are `[2]`, `[2]`, and `[2, 2]`. The total number of stable subarrays is 3.
- For `queries[1] = [0, 0]`, the subarray is `[nums[0]] = [2]`.
 - The stable subarray is `[2]`. The total number of stable subarrays is 1.

Thus, `ans = [3, 1]`.

Constraints:

- $1 \leq \text{nums.length} \leq 10^5$
- $1 \leq \text{nums}[i] \leq 10^5$
- $1 \leq \text{queries.length} \leq 10^5$
- `queries[i] = [li, ri]`
- $0 \leq l_i \leq r_i \leq \text{nums.length} - 1$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
    long long countSubarray(long long n){
        return n*(n+1)/2;
    }
public:
    vector<long long> countStableSubarrays(vector<int>& nums,
                                            vector<vector<int>>& queries) {
        vector<long long> ans;

        vector<int> last(nums.size());

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

        int b_size = (int)sqrt(nums.size()) + 1;

        vector<int> hop(nums.size());
        vector<long long> sum(nums.size(), 0);

        for (int i = nums.size() - 1; i >= 0; i--) {
            int next_bs = last[i] + 1;

            if (next_bs < nums.size() && (i / b_size) == (next_bs / b_size)) {
                hop[i] = hop[next_bs];
                long long c_bl = last[i] - i + 1;
                sum[i] = countSubarray(c_bl) + sum[next_bs];
            } else {
                hop[i] = next_bs;
                long long c_bl = last[i] - i + 1;
                sum[i] = countSubarray(c_bl);
            }
        }

        for (const auto& q : queries) {
            int l = q[0], r = q[1];
            long long tc = 0;
            int curr = l;
            while (curr <= r) {
                if ((curr / b_size) != (r / b_size) && hop[curr] <= r) {
                    tc += sum[curr];
                    curr = hop[curr];
                } else {
                    int be = last[curr];
                    int ie = min(be, r);
                    long long len = ie - curr + 1;
                    tc += countSubarray(len);
                    curr = ie + 1;
                }
            }
            ans.push_back(tc);
        }
    }
};
```

[4054 Count Distinct Integers After Removing Zeros](#) ([link](#))

Description

You are given a **positive** integer n .

For every integer x from 1 to n , we write down the integer obtained by removing all zeros from the decimal representation of x .

Return an integer denoting the number of **distinct** integers written down.

Example 1:

Input: $n = 10$

Output: 9

Explanation:

The integers we wrote down are 1, 2, 3, 4, 5, 6, 7, 8, 9, 1. There are 9 distinct integers (1, 2, 3, 4, 5, 6, 7, 8, 9).

Example 2:

Input: $n = 3$

Output: 3

Explanation:

The integers we wrote down are 1, 2, 3. There are 3 distinct integers (1, 2, 3).

Constraints:

- $1 \leq n \leq 10^{15}$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
    long long dp[17];

    long long get_ans(int k) {
        if (k == 0) return 1;

        if (dp[k] != 0) return dp[k];

        dp[k] = 9 * get_ans(k - 1);

        return dp[k];
    }

    long long solve(string& s, int i, bool is_tight) {
        if (i == s.length()) {
            return 1;
        }

        long long count = 0;

        int limit = is_tight ? (s[i] - '0') : 9;

        for (int d = 1; d <= 9; ++d) {
            if (d > limit) {
                break;
            }
            if (d < limit) {
                count += get_ans(s.length() - i - 1);
            } else {
                count += solve(s, i + 1, is_tight);
            }
        }
        return count;
    }

public:
    long long countDistinct(long long n) {
        string str = to_string(n);
        int length = str.length();
        long long tc = 0;
        for(int i=0; i<17; i++)dp[i]=0;

        for(int i=1; i<length; i++){
            tc += get_ans(i);
        }
        tc+= solve(str,0,true);
        return tc;
    }
};
```

[4116 Minimum Moves to Equal Array Elements III \(link\)](#)

Description

You are given an integer array `nums`.

In one move, you may **increase** the value of any single element `nums[i]` by 1.

Return the **minimum total** number of **moves** required so that all elements in `nums` become **equal**.

Example 1:

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

Output: 3

Explanation:

To make all elements equal:

- Increase `nums[0] = 2` by 1 to make it 3.
- Increase `nums[1] = 1` by 1 to make it 2.
- Increase `nums[1] = 2` by 1 to make it 3.

Now, all elements of `nums` are equal to 3. The minimum total moves is 3.

Example 2:

Input: `nums = [4,4,5]`

Output: 2

Explanation:

To make all elements equal:

- Increase `nums[0] = 4` by 1 to make it 5.
- Increase `nums[1] = 4` by 1 to make it 5.

Now, all elements of `nums` are equal to 5. The minimum total moves is 2.

Constraints:

- $1 \leq \text{nums.length} \leq 100$
- $1 \leq \text{nums}[i] \leq 100$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int minMoves(vector<int>& nums) {
        if(nums.size()==1) return 0;
        int ans = 0;
        int maxi = *max_element(nums.begin(),nums.end());

        for(auto& element:nums){
            ans += abs(element - maxi);
        }

        return ans;
    }
};
```

[79 Word Search \(link\)](#)

Description

Given an $m \times n$ grid of characters board and a string word, return true if word exists in the grid.

The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.

Example 1:

A	B	C	E
S	F	C	S
A	D	E	E

```
Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCED"
Output: true
```

Example 2:

A	B	C	E
S	F	C	S
A	D	E	E

```
Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "SEE"
Output: true
```

Example 3:

A	B	C	E
S	F	C	S
A	D	E	E

Input: board = [["A", "B", "C", "E"], ["S", "F", "C", "S"], ["A", "D", "E", "E"]], word = "ABCB"

Output: false

Constraints:

- m == board.length
- n = board[i].length
- 1 <= m, n <= 6
- 1 <= word.length <= 15
- board and word consists of only lowercase and uppercase English letters.

Follow up: Could you use search pruning to make your solution faster with a larger board?

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
    bool find_word(vector<vector<char>>& board, string word,
                  vector<vector<bool>>& visited, int row, int col, int index) {

        if (index == word.size())
            return true;
        if (row < 0 || row >= board.size() || col < 0 ||
            col >= board[0].size()) {
            return false;
        }
        if (word[index] != board[row][col])
            return false;
        if (visited[row][col])
            return false;

        visited[row][col] = true;
        bool left = find_word(board, word, visited, row - 1, col, index + 1);
        bool top = find_word(board, word, visited, row, col - 1, index + 1);
        bool right = find_word(board, word, visited, row + 1, col, index + 1);
        bool down = find_word(board, word, visited, row, col + 1, index + 1);
        visited[row][col] = false;

        return left || top || right || down;
    }

public:
    bool exist(vector<vector<char>>& board, string word) {
        vector<vector<bool>> visited(board.size(),
                                       vector<bool>(board[0].size(), 0));
        bool word_exists = false;
        // Anywhere from the grid
        for (int i = 0; i < board.size(); i++) {
            for (int j = 0; j < board[0].size(); j++) {
                if (board[i][j] != word[0]) continue;
                word_exists = find_word(board, word, visited, i, j, 0);
                if (word_exists) return true;
            }
        }
        return word_exists;
    }
};
```

[4090 Minimum String Length After Balanced Removals \(link\)](#)

Description

You are given a string s consisting only of the characters 'a' and 'b'.

You are allowed to repeatedly remove **any substring** where the number of 'a' characters is equal to the number of 'b' characters. After each removal, the remaining parts of the string are concatenated together without gaps.

Return an integer denoting the **minimum possible length** of the string after performing any number of such operations.

Example 1:

Input: $s = \text{"aabbab"}$

Output: 0

Explanation:

The substring "aabbab" has three 'a' and three 'b'. Since their counts are equal, we can remove the entire string directly. The minimum length is 0.

Example 2:

Input: $s = \text{"aaaa"}$

Output: 4

Explanation:

Every substring of "aaaa" contains only 'a' characters. No substring can be removed as a result, so the minimum length remains 4.

Example 3:

Input: $s = \text{"aaabb"}$

Output: 1

Explanation:

First, remove the substring "ab", leaving "aab". Next, remove the new substring "ab", leaving "a". No further removals are possible, so the minimum length is 1.

Constraints:

- $1 \leq s.\text{length} \leq 10^5$
- $s[i]$ is either 'a' or 'b'.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int minLengthAfterRemovals(string s) {
        stack<char> stack;
        int total_length = s.size(), count_a = 0, count_b=0;
        for (int i = 0; i < s.size(); i++) {
            if(s[i]=='a')count_a++;
            else count_b++;
        }

        return abs(count_a-count_b);
    }
};
```

[4112 Maximize Expression of Three Elements \(link\)](#)

Description

You are given an integer array `nums`.

Choose three elements `a`, `b`, and `c` from `nums` at **distinct** indices such that the value of the expression `a + b - c` is maximized.

Return an integer denoting the **maximum possible value** of this expression.

Example 1:

Input: `nums = [1,4,2,5]`

Output: 8

Explanation:

We can choose `a = 4`, `b = 5`, and `c = 1`. The expression value is $4 + 5 - 1 = 8$, which is the maximum possible.

Example 2:

Input: `nums = [-2,0,5,-2,4]`

Output: 11

Explanation:

We can choose `a = 5`, `b = 4`, and `c = -2`. The expression value is $5 + 4 - (-2) = 11$, which is the maximum possible.

Constraints:

- $3 \leq \text{nums.length} \leq 100$
- $-100 \leq \text{nums}[i] \leq 100$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int maximizeExpressionOfThree(vector<int>& nums) {
        sort(nums.begin(),nums.end());
        int size = nums.size();
        int ans = (nums[size-1] + nums[size-2]) - nums[0];
        return ans;
    }
};
```

[47 Permutations II \(link\)](#)

Description

Given a collection of numbers, `nums`, that might contain duplicates, return *all possible unique permutations in any order*.

Example 1:

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

Example 2:

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

Constraints:

- $1 \leq \text{nums.length} \leq 8$
- $-10 \leq \text{nums}[i] \leq 10$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
    void solve_permutations(vector<int>& nums,
                           vector<vector<int>>& final_permutations,
                           vector<bool>& visited,
                           vector<int>& current_permutations) {

        if (current_permutations.size() == nums.size()) {
            final_permutations.push_back(current_permutations);
            return;
        }

        // Do For Each Slot
        for (int i = 0; i < nums.size(); i++) {
            if (i > 0 && nums[i] == nums[i - 1] && !visited[i - 1]) {
                // Duplicate Logic ==> Take the first element of duplicates
                // !visited[i-1] -> if earlier is not picked then this means
                // this is a duplicate path
                continue;
            }
            if (visited[i]) // Already Visited Slot
                continue;
            current_permutations.push_back(nums[i]);
            visited[i] = true;
            solve_permutations(nums, final_permutations, visited,
                               current_permutations);
            current_permutations.pop_back();
            visited[i] = false;
        }
    }

public:
    vector<vector<int>> permuteUnique(vector<int>& nums) {
        vector<vector<int>> final_permutations;
        vector<bool> visited(nums.size(), 0);
        vector<int> current_permutations;

        sort(nums.begin(), nums.end());
        solve_permutations(nums, final_permutations, visited,
                           current_permutations);

        return final_permutations;
    }
};
```

[46 Permutations \(link\)](#)

Description

Given an array `nums` of distinct integers, return all the possible permutations. You can return the answer in **any order**.

Example 1:

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

Example 2:

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

Example 3:

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

Constraints:

- $1 \leq \text{nums.length} \leq 6$
- $-10 \leq \text{nums}[i] \leq 10$
- All the integers of `nums` are **unique**.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
    void calculate_permutations(vector<int>&nums, vector<vector<int>>&final_permutations, vector<bool>visited) {
        if(current_permutations.size() == nums.size()){
            final_permutations.push_back(current_permutations);
            return;
        }

        for(int i=0; i<nums.size(); i++){
            // Because Every slot must have all numbers at least once
            if(visited[i]) continue; // which slots filled
            current_permutations.push_back(nums[i]);
            visited[i] = true;
            calculate_permutations(nums, final_permutations, current_permutations, visited);
            current_permutations.pop_back();
            visited[i] = false;
        }
    }

public:
    vector<vector<int>> permute(vector<int>& nums) {
        vector<vector<int>>final_permutations;
        vector<int>current_permutations;
        vector<bool>visited(nums.size(),0);

        calculate_permutations(nums, final_permutations, current_permutations, visited);

        return final_permutations;
    }
};
```

17 Letter Combinations of a Phone Number (link)

Description

Given a string containing digits from 2–9 inclusive, return all possible letter combinations that the number could represent. Return the answer in **any order**.

A mapping of digits to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters.



Example 1:

```
Input: digits = "23"
Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]
```

Example 2:

```
Input: digits = "2"
Output: ["a", "b", "c"]
```

Constraints:

- $1 \leq \text{digits.length} \leq 4$
- $\text{digits}[i]$ is a digit in the range $['2', '9']$.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
    void all_combo(int curr_digit_index, string digits, string& curr_string,
                  vector<string>& ans,unordered_map<char,string>&num_alpha) {
        if (curr_digit_index >= digits.size()) {
            ans.push_back(curr_string);
            return;
        }

        int curr_digit = stoi(to_string(digits[curr_digit_index]));
        for(int i=0; i<num_alpha[curr_digit].size(); i++){
            curr_string += num_alpha[curr_digit][i];
            all_combo(curr_digit_index+1, digits, curr_string, ans,num_alpha);
            curr_string.pop_back();
        }
    }

public:
    vector<string> letterCombinations(string digits) {
        vector<string> ans;
        string curr_string = "";
        unordered_map<char, string> num_alpha = {
            {'2', "abc"},{'3", "def"},{'4', "ghi"},{'5', "jkl"},{'6', "mno"},{'7', "pqrs"},{'8', "tuv"},{'9', "wxyz"}
        };
        all_combo(0, digits, curr_string, ans,num_alpha);
        return ans;
    }
};
```

22 Generate Parentheses ([link](#))

Description

Given n pairs of parentheses, write a function to *generate all combinations of well-formed parentheses*.

Example 1:

```
Input: n = 3
Output: ["((()))","(()())","(())()","(())()","()()()"]
```

Example 2:

```
Input: n = 1
Output: ["()"]
```

Constraints:

- $1 \leq n \leq 8$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
    void get_all_strings(int open_count, int close_count, string curr,
                         vector<string>& final_ans, int n) {
        if (open_count + close_count == 2 * n) {
            final_ans.push_back(curr);
            return;
        }

        // Add open bracket
        if (open_count < n)
            get_all_strings(open_count + 1, close_count, curr + "(", final_ans,
                           n);

        if (close_count < open_count) {
            get_all_strings(open_count, close_count + 1, curr + ")", final_ans,
                           n);
        }
    }

public:
    vector<string> generateParenthesis(int n) {
        int open_count = 0, close_count = 0;
        vector<string> final_ans;
        string curr = "";
        get_all_strings(open_count, close_count, curr, final_ans, n);
        return final_ans;
    }
};
```

[40 Combination Sum II \(link\)](#)

Description

Given a collection of candidate numbers (candidates) and a target number (target), find all unique combinations in candidates where the candidate numbers sum to target.

Each number in candidates may only be used **once** in the combination.

Note: The solution set must not contain duplicate combinations.

Example 1:

```
Input: candidates = [10,1,2,7,6,1,5], target = 8
Output:
[
[1,1,6],
[1,2,5],
[1,7],
[2,6]
]
```

Example 2:

```
Input: candidates = [2,5,2,1,2], target = 5
Output:
[
[1,2,2],
[5]
]
```

Constraints:

- $1 \leq \text{candidates.length} \leq 100$
- $1 \leq \text{candidates}[i] \leq 50$
- $1 \leq \text{target} \leq 30$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
    void find_combination_candidates(vector<int>& candidates, int target, int index, int current_sum) {
        if(current_sum == target){
            final_ans.push_back(current_candidates);
            return;
        }else if(current_sum > target){
            return;
        }
        if(index >= candidates.size()){
            return;
        }

        // pick
        current_candidates.push_back(candidates[index]);
        find_combination_candidates(candidates,target,index+1,current_sum + candidates[index],current_candidates.pop_back());

        // Skip Duplicates
        while(index < candidates.size()-1 && candidates[index] == candidates[index+1]){
            index++;
        }

        // not pick
        find_combination_candidates(candidates,target,index+1,current_sum,current_candidates,final_ans);
    }
}

public:
    vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
        vector<vector<int>>final_ans;
        vector<int>current_candidates;

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

        find_combination_candidates(candidates,target,0,0,current_candidates,final_ans);

        return final_ans;
    }
};
```

[51 N-Queens \(link\)](#)

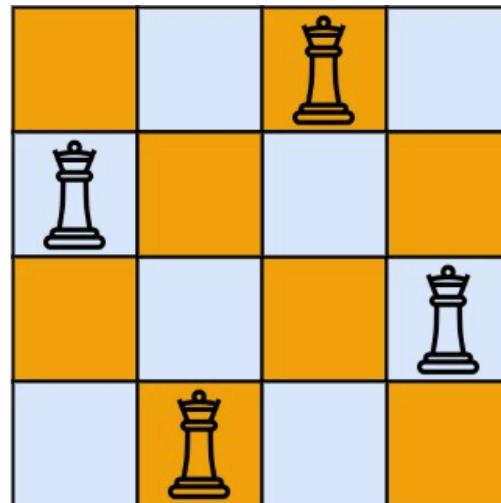
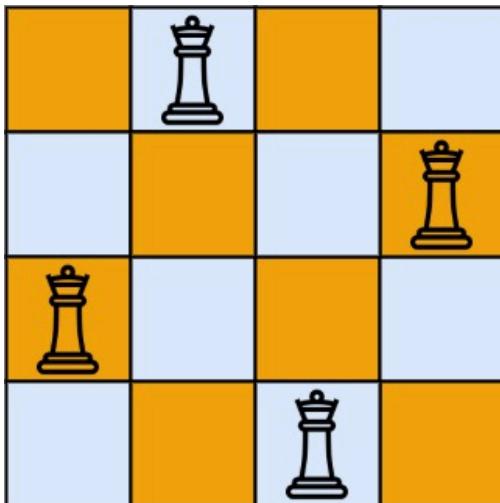
Description

The **n-queens** puzzle is the problem of placing n queens on an $n \times n$ chessboard such that no two queens attack each other.

Given an integer n , return *all distinct solutions to the n-queens puzzle*. You may return the answer in **any order**.

Each solution contains a distinct board configuration of the n-queens' placement, where 'Q' and '.' both indicate a queen and an empty space, respectively.

Example 1:



Input: $n = 4$

Output: `[["Q..","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]`

Explanation: There exist two distinct solutions to the 4-queens puzzle as shown above

Example 2:

Input: $n = 1$

Output: `[["Q"]]`

Constraints:

- $1 \leq n \leq 9$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {

    void can_place(int row, int col, unordered_set<int>& left_right,
                  unordered_set<int>& right_left, unordered_set<int>& curr_col,
                  int n, int queens_placed, vector<vector<string>>& final_ans,
                  vector<string>& curr_board) {

        if (queens_placed == n) {
            final_ans.push_back(curr_board);
            return;
        }
        if (row >= n || col >= n) {
            return;
        }

        for (int i = 0; i < n; i++) {
            bool already_queen = left_right.count(row - i) ||
                                 right_left.count(row + i) ||
                                 curr_col.count(i);
            if (already_queen) {
                continue;
            }
            curr_col.insert(i);
            left_right.insert(row - i);
            right_left.insert(row + i);
            curr_board[row][i] = 'Q';
            can_place(row + 1, i, left_right, right_left, curr_col, n,
                      queens_placed + 1, final_ans, curr_board);
            curr_board[row][i] = '.';
            curr_col.erase(i);
            left_right.erase(row - i);
            right_left.erase(row + i);
        }
    }

public:
    vector<vector<string>> solveNQueens(int n) {
        vector<vector<string>> final_ans;
        string initial = "";
        for (int i = 0; i < n; i++)
            initial += ".";
        vector<string> curr_board(n, initial);

        unordered_set<int> curr_col;
        unordered_set<int> left_right;
        unordered_set<int> right_left;
        can_place(0, 0, left_right, right_left, curr_col, n, 0, final_ans,
                  curr_board);

        return final_ans;
    }
};
```

[39 Combination Sum \(link\)](#)

Description

Given an array of **distinct** integers candidates and a target integer target, return a *list of all unique combinations* of candidates where the chosen numbers sum to target. You may return the combinations in **any order**.

The **same** number may be chosen from candidates an **unlimited number of times**. Two combinations are unique if the frequency of at least one of the chosen numbers is different.

The test cases are generated such that the number of unique combinations that sum up to target is less than 150 combinations for the given input.

Example 1:

```
Input: candidates = [2,3,6,7], target = 7
Output: [[2,2,3],[7]]
```

Explanation:

2 and 3 are candidates, and $2 + 2 + 3 = 7$. Note that 2 can be used multiple times.

7 is a candidate, and $7 = 7$.

These are the only two combinations.

Example 2:

```
Input: candidates = [2,3,5], target = 8
Output: [[2,2,2,2],[2,3,3],[3,5]]
```

Example 3:

```
Input: candidates = [2], target = 1
Output: []
```

Constraints:

- $1 \leq \text{candidates.length} \leq 30$
- $2 \leq \text{candidates}[i] \leq 40$
- All elements of candidates are **distinct**.
- $1 \leq \text{target} \leq 40$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
    void find_target_sum(vector<int>& candidates, int index, int target,
                        int current_sum, vector<int>& current_sum_arr,
                        vector<vector<int>> &final_ans) {
        if (index >= candidates.size()) {
            return;
        }
        if (current_sum == target) {
            final_ans.push_back(current_sum_arr);
            return;
        } else if (current_sum > target) {
            return;
        }

        // pick
        // Pass index for Re-Pick
        current_sum_arr.push_back(candidates[index]);
        find_target_sum(candidates, index, target,
                        current_sum + candidates[index], current_sum_arr,
                        final_ans);
        current_sum_arr.pop_back();

        // not pick
        find_target_sum(candidates, index+1, target,
                        current_sum, current_sum_arr,
                        final_ans);
    }

public:
    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
        vector<vector<int>> final_ans;
        vector<int> current_sum_arr;
        int current_sum = 0;
        find_target_sum(candidates, 0, target, current_sum, current_sum_arr,
                        final_ans);

        return final_ans;
    }
};
```

[90 Subsets II \(link\)](#)

Description

Given an integer array `nums` that may contain duplicates, return *all possible subsets (the power set)*.

The solution set **must not** contain duplicate subsets. Return the solution in **any order**.

Example 1:

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

Example 2:

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

Constraints:

- $1 \leq \text{nums.length} \leq 10$
- $-10 \leq \text{nums}[i] \leq 10$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
    void calculate_subsets(vector<int>nums, int index, vector<vector<int>>&final_ans, vector<int>current_subset) {
        if(index >= nums.size()){
            final_ans.push_back(current_subset);
            return;
        }

        // pick
        current_subset.push_back(nums[index]);
        calculate_subsets(nums, index+1, final_ans, current_subset);
        current_subset.pop_back();

        // skip duplicates
        while(index < nums.size()-1 && nums[index+1] == nums[index]){
            index++;
        }
        // not pick
        calculate_subsets(nums, index+1, final_ans, current_subset);
    }
public:
    vector<vector<int>> subsetsWithDup(vector<int>& nums) {
        vector<vector<int>>final_ans;
        sort(nums.begin(), nums.end());
        vector<int>current_subset;
        calculate_subsets(nums, 0, final_ans, current_subset);
        return final_ans;
    }
};
```

[3986 Maximum Path Score in a Grid \(link\)](#)

Description

You are given an $m \times n$ grid where each cell contains one of the values 0, 1, or 2. You are also given an integer k .

You start from the top-left corner $(0, 0)$ and want to reach the bottom-right corner $(m - 1, n - 1)$ by moving only **right** or **down**.

Each cell contributes a specific score and incurs an associated cost, according to their cell values:

- 0: adds 0 to your score and costs 0.
- 1: adds 1 to your score and costs 1.
- 2: adds 2 to your score and costs 1.

Return the **maximum** score achievable without exceeding a total cost of k , or -1 if no valid path exists.

Note: If you reach the last cell but the total cost exceeds k , the path is invalid.

Example 1:

Input: grid = [[0, 1], [2, 0]], k = 1

Output: 2

Explanation:

The optimal path is:

Cell	grid[i][j]	Score	Total Score	Cost	Total Cost
(0, 0)	0	0	0	0	0
(1, 0)	2	2	2	1	1
(1, 1)	0	0	2	0	1

Thus, the maximum possible score is 2.

Example 2:

Input: grid = [[0, 1], [1, 2]], k = 1

Output: -1

Explanation:

There is no path that reaches cell $(1, 1)$ without exceeding cost k . Thus, the answer is -1.

Constraints:

- $1 \leq m, n \leq 200$
- $0 \leq k \leq 10^3$
- $\text{grid}[0][0] == 0$
- $0 \leq \text{grid}[i][j] \leq 2$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int maxPathScore(vector<vector<int>>& grid, int k) {
        int curr_row = grid.size();
        int curr_col = grid[0].size();

        // dp[r][c][third] = max_score to reach cell (r, c) with exactly 'third' cost
        vector<vector<vector<int>>> path_scores(
            curr_row, vector<vector<int>>(
                curr_col, vector<int>(k + 1, -1)
            )
        );
        path_scores[0][0][0] = 0;

        for (int r = 0; r < curr_row; ++r) {
            for (int c = 0; c < curr_col; ++c) {

                int cell_val = grid[r][c];
                int cell_score = 0;
                int cell_cost = 0;

                if (cell_val == 1) {
                    cell_score = 1;
                    cell_cost = 1;
                } else if (cell_val == 2) {
                    cell_score = 2;
                    cell_cost = 1;
                }

                for (int third = 0; third <= k; ++third) {

                    if (r == 0 && c == 0) {
                        if (third > 0) path_scores[r][c][third] = -1;
                        continue;
                    }

                    int cost_from_prev = third - cell_cost;
                    if (cost_from_prev < 0) {
                        continue;
                    }

                    int score_from_up = (r > 0) ? path_scores[r - 1][c][cost_from_prev] : -1;
                    int score_from_left = (c > 0) ? path_scores[r][c - 1][cost_from_prev] : -1;
                    if (score_from_up != -1 || score_from_left != -1) {
                        path_scores[r][c][third] = max(score_from_up, score_from_left) + cell_s
                    }
                }
            }
        }

        const auto& last_cell_costs = path_scores[curr_row - 1][curr_col - 1];
        int max_score_at_end = *max_element(last_cell_costs.begin(), last_cell_costs.end());
        return max_score_at_end;
    }
};
```

[4119 Minimum Distance Between Three Equal Elements II \(link\)](#)

Description

You are given an integer array `nums`.

A tuple (i, j, k) of 3 **distinct** indices is **good** if $\text{nums}[i] == \text{nums}[j] == \text{nums}[k]$.

The **distance** of a **good** tuple is $\text{abs}(i - j) + \text{abs}(j - k) + \text{abs}(k - i)$, where $\text{abs}(x)$ denotes the **absolute value** of x .

Return an integer denoting the **minimum** possible **distance** of a **good** tuple. If no **good** tuples exist, return -1 .

Example 1:

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

Output: 6

Explanation:

The minimum distance is achieved by the good tuple $(0, 2, 3)$.

$(0, 2, 3)$ is a good tuple because $\text{nums}[0] == \text{nums}[2] == \text{nums}[3] == 1$. Its distance is $\text{abs}(0 - 2) + \text{abs}(2 - 3) + \text{abs}(3 - 0) = 2 + 1 + 3 = 6$.

Example 2:

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

Output: 8

Explanation:

The minimum distance is achieved by the good tuple $(2, 4, 6)$.

$(2, 4, 6)$ is a good tuple because $\text{nums}[2] == \text{nums}[4] == \text{nums}[6] == 2$. Its distance is $\text{abs}(2 - 4) + \text{abs}(4 - 6) + \text{abs}(6 - 2) = 2 + 2 + 4 = 8$.

Example 3:

Input: `nums = [1]`

Output: -1

Explanation:

There are no good tuples. Therefore, the answer is -1 .

Constraints:

- $1 \leq n == \text{nums.length} \leq 10^5$
- $1 \leq \text{nums}[i] \leq n$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int minimumDistance(vector<int>& nums) {
        unordered_map<int, vector<int>> freq;
        for (int i = 0; i < nums.size(); i++) {
            freq[nums[i]].push_back(i);
        }

        int min_distance = INT_MAX;

        for (auto& pair : freq) {
            vector<int>& index = pair.second;

            if (index.size() < 3) {
                continue;
            }

            for (int i = 0; i <= index.size() - 3; i++) {
                int i_val = index[i];
                int k_val = index[i+2];

                int dist = 2 * (k_val - i_val);
                min_distance = min(min_distance, dist);
            }
        }

        return min_distance == INT_MAX ? -1 : min_distance;
    }
};
```

[4115 Minimum Distance Between Three Equal Elements I \(link\)](#)

Description

You are given an integer array `nums`.

A tuple (i, j, k) of 3 **distinct** indices is **good** if $\text{nums}[i] == \text{nums}[j] == \text{nums}[k]$.

The **distance** of a **good** tuple is $\text{abs}(i - j) + \text{abs}(j - k) + \text{abs}(k - i)$, where $\text{abs}(x)$ denotes the **absolute value** of x .

Return an integer denoting the **minimum** possible **distance** of a **good** tuple. If no **good** tuples exist, return -1.

Example 1:

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

Output: 6

Explanation:

The minimum distance is achieved by the good tuple $(0, 2, 3)$.

$(0, 2, 3)$ is a good tuple because $\text{nums}[0] == \text{nums}[2] == \text{nums}[3] == 1$. Its distance is $\text{abs}(0 - 2) + \text{abs}(2 - 3) + \text{abs}(3 - 0) = 2 + 1 + 3 = 6$.

Example 2:

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

Output: 8

Explanation:

The minimum distance is achieved by the good tuple $(2, 4, 6)$.

$(2, 4, 6)$ is a good tuple because $\text{nums}[2] == \text{nums}[4] == \text{nums}[6] == 2$. Its distance is $\text{abs}(2 - 4) + \text{abs}(4 - 6) + \text{abs}(6 - 2) = 2 + 2 + 4 = 8$.

Example 3:

Input: `nums = [1]`

Output: -1

Explanation:

There are no good tuples. Therefore, the answer is -1.

Constraints:

- $1 \leq n == \text{nums.length} \leq 100$
- $1 \leq \text{nums}[i] \leq n$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int minimumDistance(vector<int>& nums) {
        // unordered_map<int, vector<int>> freq;
        // for (int i = 0; i < nums.size(); i++) {
        //     freq[nums[i]].push_back(i);
        // }

        int min_distance = INT_MAX;
        int curr_distance;

        // for (auto& element : freq) {
        //     if (element.second.size() >= 3) {
        //         vector<int> nums = element.second;
        //         int first = element.second[0];
        //         int second = element.second[1];
        //         int third = element.second[2];
        //     }
        // }
        for (int i = 0; i < nums.size(); i++) {
            for (int j = i + 1; j < nums.size(); j++) {
                for (int k = j + 1; k < nums.size(); k++) {
                    if ((nums[i] == nums[j]) && (nums[j] == nums[k])) {
                        curr_distance = abs(i-j) +
                            abs(j-k) +
                            abs(i-k);

                        min_distance = min(min_distance, curr_distance);
                    }
                }
            }
        }
        return min_distance == INT_MAX ? -1 : min_distance;
    };
};
```

[78 Subsets \(link\)](#)

Description

Given an integer array `nums` of **unique** elements, return *all possible subsets (the power set)*.

The solution set **must not** contain duplicate subsets. Return the solution in **any order**.

Example 1:

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

Example 2:

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

Constraints:

- $1 \leq \text{nums.length} \leq 10$
- $-10 \leq \text{nums}[i] \leq 10$
- All the numbers of `nums` are **unique**.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
    void calculate_subsets(int index, vector<int>& nums,
                          vector<vector<int>>& final_ans,
                          vector<int> &current_subset) {
        // Base Case
        if (index >= nums.size()) {
            final_ans.push_back(current_subset);
            return;
        }

        // not pick
        calculate_subsets(index + 1, nums, final_ans, current_subset);
        // pick
        current_subset.push_back(nums[index]);
        calculate_subsets(index + 1, nums, final_ans, current_subset);
        current_subset.pop_back();
    }

public:
    vector<vector<int>> subsets(vector<int>& nums) {
        vector<vector<int>> final_ans;
        vector<int> current_subset;
        calculate_subsets(0, nums, final_ans, current_subset);

        return final_ans;
    }
};
```

[4075 Count Subarrays With Majority Element II \(link\)](#)

Description

You are given an integer array `nums` and an integer target.

Return the number of **subarrays** of `nums` in which target is the **majority element**.

The **majority element** of a subarray is the element that appears **strictly more than half** of the times in that subarray.

Example 1:

Input: `nums` = [1,2,2,3], target = 2

Output: 5

Explanation:

Valid subarrays with target = 2 as the majority element:

- `nums[1..1]` = [2]
- `nums[2..2]` = [2]
- `nums[1..2]` = [2,2]
- `nums[0..2]` = [1,2,2]
- `nums[1..3]` = [2,2,3]

So there are 5 such subarrays.

Example 2:

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

Output: 10

Explanation:

All 10 subarrays have 1 as the majority element.

Example 3:

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

Output: 0

Explanation:

target = 4 does not appear in `nums` at all. Therefore, there cannot be any subarray where 4 is the majority element. Hence the answer is 0.

Constraints:

- $1 \leq \text{nums.length} \leq 10^5$
- $1 \leq \text{nums}[i] \leq 10^9$
- $1 \leq \text{target} \leq 10^9$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    long long countMajoritySubarrays(vector<int>& nums, int target) {
        long long count = 0;
        long long balance = 0;
        long long valid_here = 0;
        unordered_map<long long,int> freq;

        freq[0] = 1;

        for(auto &element:nums){
            if(element == target){
                // here valid
                balance++;
                valid_here += freq[balance-1];
            }else{
                // not valid
                balance--;
                valid_here -= freq[balance];
            }
            count+=valid_here;
            freq[balance]++;
        }
        return count;
    }
};
```

[735 Asteroid Collision \(link\)](#)

Description

We are given an array `asteroids` of integers representing asteroids in a row. The indices of the asteroid in the array represent their relative position in space.

For each asteroid, the absolute value represents its size, and the sign represents its direction (positive meaning right, negative meaning left). Each asteroid moves at the same speed.

Find out the state of the asteroids after all collisions. If two asteroids meet, the smaller one will explode. If both are the same size, both will explode. Two asteroids moving in the same direction will never meet.

Example 1:

```
Input: asteroids = [5,10,-5]
Output: [5,10]
```

Explanation: The 10 and -5 collide resulting in 10. The 5 and 10 never collide.

Example 2:

```
Input: asteroids = [8,-8]
Output: []
```

Explanation: The 8 and -8 collide exploding each other.

Example 3:

```
Input: asteroids = [10,2,-5]
Output: [10]
```

Explanation: The 2 and -5 collide resulting in -5. The 10 and -5 collide resulting in 10.

Example 4:

```
Input: asteroids = [3,5,-6,2,-1,4]
Output: [-6,2,4]
```

Explanation: The asteroid -6 makes the asteroid 3 and 5 explode, and then continues going left.

Constraints:

- $2 \leq \text{asteroids.length} \leq 10^4$
- $-1000 \leq \text{asteroids}[i] \leq 1000$
- $\text{asteroids}[i] \neq 0$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    vector<int> asteroidCollision(vector<int>& asteroids) {
        stack<int> surviving_asteroids_stack;

        for(int i=0; i<asteroids.size();i++){
            // Only condition because there is collision
            // if already present asteroid Left->right(+) and new_asteroid Left<-Right(-)
            bool asteroid_survives = true;
            while(!surviving_asteroids_stack.empty() && asteroids[i] < 0 && surviving_asteroids_stack.top() > 0){
                if(abs(asteroids[i]) == abs(surviving_asteroids_stack.top())){
                    // both explode
                    surviving_asteroids_stack.pop();
                    asteroid_survives = false;
                }else if(abs(asteroids[i]) < abs(surviving_asteroids_stack.top())){
                    // new asteroid explode
                    asteroid_survives = false;
                }else{
                    // old asteroid explode
                    surviving_asteroids_stack.pop();
                }
                if(!asteroid_survives)break;
            }
            if(asteroid_survives){
                surviving_asteroids_stack.push(asteroids[i]);
            }
        }
        vector<int>ans;
        while(!surviving_asteroids_stack.empty()){
            ans.push_back(surviving_asteroids_stack.top());
            surviving_asteroids_stack.pop();
        }

        reverse(ans.begin(),ans.end());
        return ans;
    }
};
```

[4074 Count Subarrays With Majority Element I \(link\)](#)

Description

You are given an integer array `nums` and an integer target.

Return the number of **subarrays** of `nums` in which target is the **majority element**.

The **majority element** of a subarray is the element that appears **strictly more than half** of the times in that subarray.

Example 1:

Input: `nums` = [1,2,2,3], target = 2

Output: 5

Explanation:

Valid subarrays with target = 2 as the majority element:

- `nums[1..1]` = [2]
- `nums[2..2]` = [2]
- `nums[1..2]` = [2,2]
- `nums[0..2]` = [1,2,2]
- `nums[1..3]` = [2,2,3]

So there are 5 such subarrays.

Example 2:

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

Output: 10

Explanation:

All 10 subarrays have 1 as the majority element.

Example 3:

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

Output: 0

Explanation:

target = 4 does not appear in `nums` at all. Therefore, there cannot be any subarray where 4 is the majority element. Hence the answer is 0.

Constraints:

- $1 \leq \text{nums.length} \leq 1000$
- $1 \leq \text{nums}[i] \leq 10^9$
- $1 \leq \text{target} \leq 10^9$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int countMajoritySubarrays(vector<int>& nums, int target) {
        int count = 0;

        for(int i=0; i<nums.size(); i++){
            unordered_map<int, int> freq;
            for(int j=i; j<nums.size(); j++){
                freq[nums[j]]++;
                if(freq[target] > ((j-i+1)/2)){
                    count++;
                }
            }
        }

        return count;
    }
};
```

[224 Basic Calculator \(link\)](#)

Description

Given a string s representing a valid expression, implement a basic calculator to evaluate it, and return *the result of the evaluation*.

Note: You are **not** allowed to use any built-in function which evaluates strings as mathematical expressions, such as `eval()`.

Example 1:

```
Input: s = "1 + 1"
Output: 2
```

Example 2:

```
Input: s = " 2-1 + 2 "
Output: 3
```

Example 3:

```
Input: s = "(1+(4+5+2)-3)+(6+8)"
Output: 23
```

Constraints:

- $1 \leq s.length \leq 3 * 10^5$
- s consists of digits, '+', '-', '(', ')', and ' '.
- s represents a valid expression.
- '+' is **not** used as a unary operation (i.e., "+1" and "+(2 + 3)" is invalid).
- '-' could be used as a unary operation (i.e., "-1" and "-(2 + 3)" is valid).
- There will be no two consecutive operators in the input.
- Every number and running calculation will fit in a signed 32-bit integer.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int calculate(string s) {
        stack<long long> result_sign_stack;
        long long current_result = 0, current_number = 0, current_sign = 1;

        for (int i = 0; i < s.size(); i++) {
            if (s[i] == ' ')
                continue;

            if (s[i] - '0' >= 0 && s[i] - '0' <= 9) {
                // Number or multi-digit number
                current_number = current_number * 10 + (s[i] - '0');
            } else if (s[i] == '(') {
                current_result += (current_sign * current_number);
                result_sign_stack.push(current_result);
                result_sign_stack.push(current_sign);
                current_result = 0, current_number = 0, current_sign = 1;
            } else if (s[i] == ')') {
                current_result += current_sign * current_number;
                current_number = 0;
                int prev_sign = result_sign_stack.top();
                result_sign_stack.pop();
                long long prev_result = result_sign_stack.top();
                result_sign_stack.pop();

                current_result = prev_result + (prev_sign * current_result);
            } else {
                // Sign
                current_result += (current_sign * current_number);
                current_number = 0;
                if (s[i] == '+') {
                    current_sign = 1;
                } else if (s[i] == '-') {
                    current_sign = -1;
                }
            }
        }

        current_result = current_result + (current_sign * current_number);

        return current_result;
    }
};
```

[394 Decode String \(link\)](#)

Description

Given an encoded string, return its decoded string.

The encoding rule is: $k [encoded_string]$, where the `encoded_string` inside the square brackets is being repeated exactly k times. Note that k is guaranteed to be a positive integer.

You may assume that the input string is always valid; there are no extra white spaces, square brackets are well-formed, etc. Furthermore, you may assume that the original data does not contain any digits and that digits are only for those repeat numbers, k . For example, there will not be input like `3a` or `2[4]`.

The test cases are generated so that the length of the output will never exceed 10^5 .

Example 1:

```
Input: s = "3[a]2[bc]"  
Output: "aaabcbc"
```

Example 2:

```
Input: s = "3[a2[c]]"  
Output: "accaccacc"
```

Example 3:

```
Input: s = "2[abc]3[cd]ef"  
Output: "abcabccdcdef"
```

Constraints:

- $1 \leq s.length \leq 30$
- s consists of lowercase English letters, digits, and square brackets '`[]`'.
- s is guaranteed to be a **valid** input.
- All the integers in s are in the range $[1, 300]$.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    string decodeString(string s) {
        int current_number = 0;
        string current_string = "";

        stack<int> count_stack;
        stack<string> string_stack;

        for (int i = 0; i < s.size(); i++) {
            if (s[i] - '0' >= 0 && s[i] - '0' <= 9) {
                current_number = (current_number * 10) + (s[i] - '0');
                continue;
            }
            else if(s[i] == '['){
                count_stack.push(current_number);
                string_stack.push(current_string);
                current_number=0;current_string="";
            }
            else if(s[i]==']'){
                int repeat = count_stack.top();
                count_stack.pop();
                string prev_string = string_stack.top();
                string_stack.pop();
                string new_current_string = prev_string;

                for(int i=0; i<repeat;i++){
                    new_current_string+=current_string;
                }
                current_string = new_current_string;
            }else{
                current_string += s[i];
            }
        }
        return current_string;
    }
};
```

[4048 Minimum Time to Complete All Deliveries \(link\)](#)

Description

You are given two integer arrays of size 2: $d = [d_1, d_2]$ and $r = [r_1, r_2]$.

Two delivery drones are tasked with completing a specific number of deliveries. Drone i must complete d_i deliveries.

Each delivery takes **exactly** one hour and **only one** drone can make a delivery at any given hour.

Additionally, both drones require recharging at specific intervals during which they cannot make deliveries. Drone i must recharge every r_i hours (i.e. at hours that are multiples of r_i).

Return an integer denoting the **minimum** total time (in hours) required to complete all deliveries.

Example 1:

Input: $d = [3, 1]$, $r = [2, 3]$

Output: 5

Explanation:

- The first drone delivers at hours 1, 3, 5 (recharges at hours 2, 4).
- The second drone delivers at hour 2 (recharges at hour 3).

Example 2:

Input: $d = [1, 3]$, $r = [2, 2]$

Output: 7

Explanation:

- The first drone delivers at hour 3 (recharges at hours 2, 4, 6).
- The second drone delivers at hours 1, 5, 7 (recharges at hours 2, 4, 6).

Example 3:

Input: $d = [2, 1]$, $r = [3, 4]$

Output: 3

Explanation:

- The first drone delivers at hours 1, 2 (recharges at hour 3).
- The second drone delivers at hour 3.

Constraints:

- $d = [d_1, d_2]$
- $1 \leq d_i \leq 10^9$
- $r = [r_1, r_2]$
- $2 \leq r_i \leq 3 * 10^4$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
    long long gcd(long long a, long long b) {
        while (b) {
            a %= b;
            swap(a, b);
        }
        return a;
    }

    long long lcm(long long a, long long b) {
        if (a == 0 || b == 0) return 0;
        return (a / gcd(a, b)) * b;
    }

public:
    long long minimumTime(vector<int>& d, vector<int>& r) {
        vector<vector<int>> arr = {d, r};

        long long d1 = arr[0][0];
        long long d2 = arr[0][1], r1 = arr[1][0], r2 = arr[1][1];

        long long low = d1 + d2;
        long long high = 4000000000LL;
        long long ans = high;

        while (low <= high) {
            long long total = low + (high - low) / 2;
            long long l = lcm(r1, r2);

            long long d1_drone = (total / r2) - (total / l);
            long long d2_drone = (total / r1) - (total / l);
            long long both_drone = total - (total / r1) - (total / r2) + (total / l);

            bool can_d1 = (d1 <= d1_drone + both_drone);
            bool can_d2 = (d2 <= d2_drone + both_drone);
            bool can_total = ((d1 + d2) <= (d1_drone + d2_drone + both_drone));

            if (can_d1 && can_d2 && can_total) {
                ans = total;
                high = total - 1;
            } else {
                low = total + 1;
            }
        }

        return ans;
    }
};
```

4101 Maximum Product of Three Elements After One Replacement (link)

Description

You are given an integer array `nums`.

You **must** replace **exactly one** element in the array with **any** integer value in the range $[-10^5, 10^5]$ (inclusive).

After performing this single replacement, determine the **maximum possible product of any three elements at distinct indices** from the modified array.

Return an integer denoting the **maximum product** achievable.

Example 1:

Input: `nums = [-5,7,0]`

Output: 3500000

Explanation:

Replacing 0 with -10^5 gives the array $[-5, 7, -10^5]$, which has a product $(-5) * 7 * (-10^5) = 3500000$. The maximum product is 3500000.

Example 2:

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

Output: 1200000

Explanation:

Two ways to achieve the maximum product include:

- $[-4, -2, -3] \rightarrow$ replace -2 with $10^5 \rightarrow$ product = $(-4) * 10^5 * (-3) = 1200000$.
- $[-4, -1, -3] \rightarrow$ replace -1 with $10^5 \rightarrow$ product = $(-4) * 10^5 * (-3) = 1200000$.

The maximum product is 1200000.

Example 3:

Input: `nums = [0,10,0]`

Output: 0

Explanation:

There is no way to replace an element with another integer and not have a 0 in the array. Hence, the product of all three elements will always be 0, and the maximum product is 0.

Constraints:

- $3 \leq \text{nums.length} \leq 10^5$
- $-10^5 \leq \text{nums}[i] \leq 10^5$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    long long maxProduct(vector<int>& nums) {
        sort(nums.begin(),nums.end());
        int max1 = 0, max2 = 0, min = 0;
        // case 1
        int size = nums.size();
        long long product1 = (long long)nums[size-1] * nums[size-2];
        long long product2 = (long long)nums[0] * nums[1];
        long long max_product = (long long)INT_MIN;
        int index1=0, index2=0;
        if(abs(product1) > abs(product2)){
            max_product = product1;
            index1 = size-1;
            index2 = size-2;
        }else{
            max_product = product2;
            index1 = 0;
            index2 = 1;
        }

        int i=0,j=size-1;
        while(i<j){
            if(abs((long long)nums[i] * nums[j]) > abs(max_product)){
                index1 = i;
                index2 = j;
                max_product = (long long)nums[i]*nums[j];
            }
            if(nums[i] < nums[j]){
                i++;
            }else{
                j--;
            }
        }

        if(max_product < 0){
            return (long long)max_product * -1e5;
        }else{
            return (long long)max_product * 1e5;
        }
    }
};
```

[4107 Find Missing Elements \(link\)](#)

Description

You are given an integer array `nums` consisting of **unique** integers.

Originally, `nums` contained **every integer** within a certain range. However, some integers might have gone **missing** from the array.

The **smallest** and **largest** integers of the original range are still present in `nums`.

Return a **sorted** list of all the missing integers in this range. If no integers are missing, return an **empty** list.

Example 1:

Input: `nums` = [1,4,2,5]

Output: [3]

Explanation:

The smallest integer is 1 and the largest is 5, so the full range should be [1,2,3,4,5]. Among these, only 3 is missing.

Example 2:

Input: `nums` = [7,8,6,9]

Output: []

Explanation:

The smallest integer is 6 and the largest is 9, so the full range is [6,7,8,9]. All integers are already present, so no integer is missing.

Example 3:

Input: `nums` = [5,1]

Output: [2,3,4]

Explanation:

The smallest integer is 1 and the largest is 5, so the full range should be [1,2,3,4,5]. The missing integers are 2, 3, and 4.

Constraints:

- $2 \leq \text{nums.length} \leq 100$
- $1 \leq \text{nums}[i] \leq 100$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    vector<int> findMissingElements(vector<int>& nums) {
        sort(nums.begin(),nums.end());
        int start = nums[0];
        int end = nums[nums.size()-1];
        int current = 1;
        vector<int>ans;
        for(int i=start+1;i<end;i++){
            auto found = find(nums.begin(),nums.end(),i);
            if(found == nums.end()){
                ans.push_back(i);
            }
        }
        return ans;
    }
};
```

225 Implement Stack using Queues (link)

Description

Implement a last-in-first-out (LIFO) stack using only two queues. The implemented stack should support all the functions of a normal stack (push, top, pop, and empty).

Implement the MyStack class:

- void push(int x) Pushes element x to the top of the stack.
- int pop() Removes the element on the top of the stack and returns it.
- int top() Returns the element on the top of the stack.
- boolean empty() Returns true if the stack is empty, false otherwise.

Notes:

- You must use **only** standard operations of a queue, which means that only push to back, peek/pop from front, size and is empty operations are valid.
- Depending on your language, the queue may not be supported natively. You may simulate a queue using a list or deque (double-ended queue) as long as you use only a queue's standard operations.

Example 1:

```
Input
["MyStack", "push", "push", "top", "pop", "empty"]
[], [1], [2], [], [2], []]
```

```
Output
[null, null, null, 2, 2, false]
```

Explanation

```
MyStack myStack = new MyStack();
myStack.push(1);
myStack.push(2);
myStack.top(); // return 2
myStack.pop(); // return 2
myStack.empty(); // return False
```

Constraints:

- $1 \leq x \leq 9$
- At most 100 calls will be made to push, pop, top, and empty.
- All the calls to pop and top are valid.

Follow-up: Can you implement the stack using only one queue?

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class MyStack {
    queue<int>queue;
public:
    MyStack() {

    }

    void push(int x) {
        int element_to_push = x;
        if(queue.empty()){
            queue.push(element_to_push);
            return;
        }
        int size = queue.size();
        queue.push(element_to_push);
        for(int i=0;i<size;i++){
            int front_element = queue.front();
            queue.pop();
            queue.push(front_element);
        }
    }

    int pop() {
        if(queue.empty())return -1;
        int popped_element = queue.front();
        queue.pop();
        return popped_element;
    }

    int top() {
        if(queue.empty())return -1;
        int front_element = queue.front();
        return front_element;
    }

    bool empty() {
        return queue.empty();
    }
};

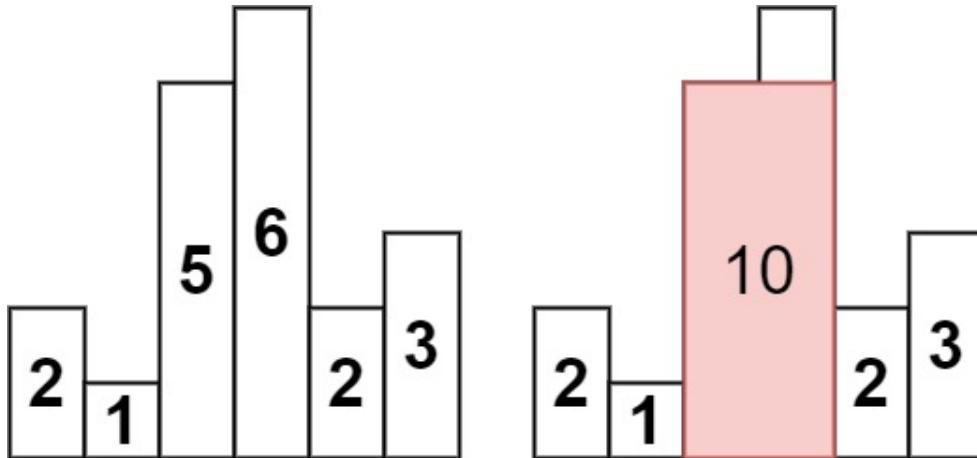
/***
 * Your MyStack object will be instantiated and called as such:
 * MyStack* obj = new MyStack();
 * obj->push(x);
 * int param_2 = obj->pop();
 * int param_3 = obj->top();
 * bool param_4 = obj->empty();
 */
```

[84 Largest Rectangle in Histogram \(link\)](#)

Description

Given an array of integers heights representing the histogram's bar height where the width of each bar is 1, return *the area of the largest rectangle in the histogram*.

Example 1:



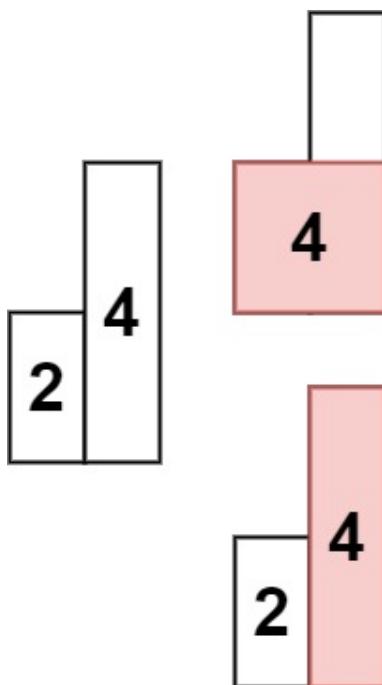
Input: heights = [2,1,5,6,2,3]

Output: 10

Explanation: The above is a histogram where width of each bar is 1.

The largest rectangle is shown in the red area, which has an area = 10 units.

Example 2:



Input: heights = [2,4]

Output: 4

Constraints:

- $1 \leq \text{heights.length} \leq 10^5$
- $0 \leq \text{heights}[i] \leq 10^4$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int largestRectangleArea(vector<int>& heights) {
        heights.push_back(0); // to calculate all remaining heights left in the stack
        stack<int>stack;
        int max_area = 0;

        for(int i=0; i<heights.size(); i++){
            // current_height < actual_height_to_calculate
            while(!stack.empty() && heights[i] < heights[stack.top()]){
                int actual_height = heights[stack.top()];
                stack.pop();
                int right_boundary_index = i;
                int left_boundary_index = stack.empty() ? -1 : stack.top();
                // area = height * width
                int current_area = actual_height * (right_boundary_index - left_boundary_index);
                max_area = max(max_area, current_area);
            }
            stack.push(i); // wait for current height to get its right boundary
        }
        return max_area;
    }
};
```

[883 Car Fleet \(link\)](#)

Description

There are n cars at given miles away from the starting mile 0, traveling to reach the mile target.

You are given two integer arrays `position` and `speed`, both of length n , where `position[i]` is the starting mile of the i^{th} car and `speed[i]` is the speed of the i^{th} car in miles per hour.

A car cannot pass another car, but it can catch up and then travel next to it at the speed of the slower car.

A **car fleet** is a single car or a group of cars driving next to each other. The speed of the car fleet is the **minimum speed** of any car in the fleet.

If a car catches up to a car fleet at the mile target, it will still be considered as part of the car fleet.

Return the number of car fleets that will arrive at the destination.

Example 1:

Input: target = 12, position = [10,8,0,5,3], speed = [2,4,1,1,3]

Output: 3

Explanation:

- The cars starting at 10 (speed 2) and 8 (speed 4) become a fleet, meeting each other at 12. The fleet forms at target.
- The car starting at 0 (speed 1) does not catch up to any other car, so it is a fleet by itself.
- The cars starting at 5 (speed 1) and 3 (speed 3) become a fleet, meeting each other at 6. The fleet moves at speed 1 until it reaches target.

Example 2:

Input: target = 10, position = [3], speed = [3]

Output: 1

Explanation:

There is only one car, hence there is only one fleet.

Example 3:

Input: target = 100, position = [0,2,4], speed = [4,2,1]

Output: 1

Explanation:

- The cars starting at 0 (speed 4) and 2 (speed 2) become a fleet, meeting each other at 4. The car starting at 4 (speed 1) travels to 5.
- Then, the fleet at 4 (speed 2) and the car at position 5 (speed 1) become one fleet, meeting each other at 6. The fleet moves at speed 1 until it reaches target.

Constraints:

- $n == \text{position.length} == \text{speed.length}$
- $1 \leq n \leq 10^5$
- $0 < \text{target} \leq 10^6$

- $0 \leq position[i] < target$
- All the values of position are **unique**.
- $0 < speed[i] \leq 10^6$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
    bool static custom_operator(const pair<int, int> pair1,
                               const pair<int, int> pair2) {
        return pair1.first > pair2.first;
    }

public:
    int carFleet(int target, vector<int>& position, vector<int>& speed) {
        vector<pair<int, int>> position_speed;

        for (int i = 0; i < position.size(); i++) {
            position_speed.push_back({position[i], speed[i]});
        }

        sort(position_speed.begin(), position_speed.end(), custom_operator);

        vector<double> time;
        for (int i=0;i<position_speed.size();i++){
            double current_time = (double)(target - position_speed[i].first)/(double)position_
            time.push_back(current_time);
        }

        stack<double> fleets;
        for (int i=0; i<time.size();i++){
            if(fleets.empty()){
                fleets.push(time[i]);
                continue;
            }
            if(time[i] > fleets.top()){
                fleets.push(time[i]);
            }
        }

        return fleets.size();
    };
}
```

[150 Evaluate Reverse Polish Notation \(link\)](#)

Description

You are given an array of strings `tokens` that represents an arithmetic expression in a [Reverse Polish Notation](#).

Evaluate the expression. Return *an integer that represents the value of the expression*.

Note that:

- The valid operators are '+', '-', '*', and '/'.
- Each operand may be an integer or another expression.
- The division between two integers always **truncates toward zero**.
- There will not be any division by zero.
- The input represents a valid arithmetic expression in a reverse polish notation.
- The answer and all the intermediate calculations can be represented in a **32-bit** integer.

Example 1:

```
Input: tokens = ["2","1","+","3","*"]
Output: 9
Explanation: ((2 + 1) * 3) = 9
```

Example 2:

```
Input: tokens = ["4","13","5","/","+"]
Output: 6
Explanation: (4 + (13 / 5)) = 6
```

Example 3:

```
Input: tokens = ["10","6","9","3","+","-11","*","/","*","17","+", "5", "+"]
Output: 22
Explanation: ((10 * (6 / ((9 + 3) * -11))) + 17) + 5
= ((10 * (6 / (12 * -11))) + 17) + 5
= ((10 * (6 / -132)) + 17) + 5
= ((10 * 0) + 17) + 5
= (0 + 17) + 5
= 17 + 5
= 22
```

Constraints:

- $1 \leq \text{tokens.length} \leq 10^4$
- `tokens[i]` is either an operator: "+", "-", "*", or "/", or an integer in the range [-200, 200].

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int evalRPN(vector<string>& tokens) {
        int final_ans = 0;
        stack<string> stack;
        stack.push(tokens[0]);
        for (int i = 1; i < tokens.size(); i++) {
            if (tokens[i] == "+" || tokens[i] == "-" || tokens[i] == "/" || tokens[i] == "*") {
                int second_number = stoi(stack.top());
                stack.pop();
                int first_number = stoi(stack.top());
                stack.pop();
                if (tokens[i] == "+") {
                    stack.push(to_string(first_number + second_number));
                }
                else if (tokens[i] == "-") {
                    stack.push(to_string(first_number - second_number));
                }
                else if (tokens[i] == "*") {
                    stack.push(to_string(first_number * second_number));
                }
                else if (tokens[i] == "/") {
                    stack.push(to_string(first_number / second_number));
                }
            }else{
                stack.push(tokens[i]);
            }
        }
        if(!stack.empty())final_ans = stoi(stack.top());
        return final_ans;
    }
};
```

[232 Implement Queue using Stacks \(link\)](#)

Description

Implement a first in first out (FIFO) queue using only two stacks. The implemented queue should support all the functions of a normal queue (push, peek, pop, and empty).

Implement the MyQueue class:

- void push(int x) Pushes element x to the back of the queue.
- int pop() Removes the element from the front of the queue and returns it.
- int peek() Returns the element at the front of the queue.
- boolean empty() Returns true if the queue is empty, false otherwise.

Notes:

- You must use **only** standard operations of a stack, which means only push to top, peek/pop from top, size, and is empty operations are valid.
- Depending on your language, the stack may not be supported natively. You may simulate a stack using a list or deque (double-ended queue) as long as you use only a stack's standard operations.

Example 1:

Input
["MyQueue", "push", "push", "peek", "pop", "empty"]
[], [1], [2], [], [], []]

Output
[null, null, null, 1, 1, false]

Explanation

```
MyQueue myQueue = new MyQueue();
myQueue.push(1); // queue is: [1]
myQueue.push(2); // queue is: [1, 2] (leftmost is front of the queue)
myQueue.peek(); // return 1
myQueue.pop(); // return 1, queue is [2]
myQueue.empty(); // return false
```

Constraints:

- $1 \leq x \leq 9$
- At most 100 calls will be made to push, pop, peek, and empty.
- All the calls to pop and peek are valid.

Follow-up: Can you implement the queue such that each operation is **amortized** $O(1)$ time complexity? In other words, performing n operations will take overall $O(n)$ time even if one of those operations may take longer.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class MyQueue {
    stack<int>input_stack;
    stack<int>output_stack;
public:
    MyQueue() {

    }

    void push(int x) {
        input_stack.push(x);
    }

    int pop() {
        if(output_stack.empty()){
            // Repeatedly pop from input and store in output to show FIFO
            while(!input_stack.empty()){
                int top_input_stack = input_stack.top();
                input_stack.pop();
                output_stack.push(top_input_stack);
            }
            int popped_element = output_stack.top();
            output_stack.pop();
            return popped_element;
        }else{
            int top_output_stack = output_stack.top();
            output_stack.pop();
            return top_output_stack;
        }
    }

    int peek() {
        if(output_stack.empty()){
            // Repeatedly pop from input and store in output to show FIFO
            while(!input_stack.empty()){
                int top_input_stack = input_stack.top();
                input_stack.pop();
                output_stack.push(top_input_stack);
            }
            return output_stack.top();
        }else{
            int top_output_stack = output_stack.top();
            return top_output_stack;
        }
    }

    bool empty() {
        if(output_stack.empty() && input_stack.empty())return true;
        return false;
    }
};

/***
 * Your MyQueue object will be instantiated and called as such:
 * MyQueue* obj = new MyQueue();
 * obj->push(x);
 * int param_2 = obj->pop();
 * int param_3 = obj->peek();
 * bool param_4 = obj->empty();
 */
```

[739 Daily Temperatures \(link\)](#)

Description

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

Example 1:

```
Input: temperatures = [73,74,75,71,69,72,76,73]
Output: [1,1,4,2,1,1,0,0]
```

Example 2:

```
Input: temperatures = [30,40,50,60]
Output: [1,1,1,0]
```

Example 3:

```
Input: temperatures = [30,60,90]
Output: [1,1,0]
```

Constraints:

- $1 \leq \text{temperatures.length} \leq 10^5$
- $30 \leq \text{temperatures}[i] \leq 100$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    vector<int> dailyTemperatures(vector<int>& temperatures) {
        stack<int> stack; // colder_to_warmer
        vector<int> ans(temperatures.size(), 0);

        for (int i = temperatures.size() - 1; i >= 0; i--) {
            if (stack.empty()) {
                stack.push(i);
                ans[i] = 0;
                continue;
            }
            while (!stack.empty() && temperatures[stack.top()] <= temperatures[i]) {
                // colder day at stack.top() will never be required anymore
                // because current is more warmer
                stack.pop();
            }
            if(!stack.empty())ans[i] = stack.top() - i;
            else ans[i]=0;
            stack.push(i);
        }
        return ans;
    }
};
```

155 Min Stack (link)

Description

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the `MinStack` class:

- `MinStack()` initializes the stack object.
- `void push(int val)` pushes the element `val` onto the stack.
- `void pop()` removes the element on the top of the stack.
- `int top()` gets the top element of the stack.
- `int getMin()` retrieves the minimum element in the stack.

You must implement a solution with $O(1)$ time complexity for each function.

Example 1:

Input
["MinStack","push","push","push","getMin","pop","top","getMin"]
[[],[-2],[0],[-3],[],[],[],[]]

Output
[null,null,null,null,-3,null,0,-2]

Explanation
`MinStack minStack = new MinStack();`
`minStack.push(-2);`
`minStack.push(0);`
`minStack.push(-3);`
`minStack.getMin(); // return -3`
`minStack.pop();`
`minStack.top(); // return 0`
`minStack.getMin(); // return -2`

Constraints:

- $-2^{31} \leq \text{val} \leq 2^{31} - 1$
- Methods `pop`, `top` and `getMin` operations will always be called on **non-empty** stacks.
- At most $3 * 10^4$ calls will be made to `push`, `pop`, `top`, and `getMin`.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class MinStack {
    // current element, min_element_so_far
    stack<pair<int,int>>stack;
public:
    MinStack() {

    }

    void push(int val) {
        if(stack.empty()){
            stack.push({val,val});
            return;
        }
        int min_element_till_now = min(val,stack.top().second);
        stack.push({val,min_element_till_now});
    }

    void pop() {
        if(!stack.empty())stack.pop();
    }

    int top() {
        if(stack.empty())return -1;
        return stack.top().first;
    }

    int getMin() {
        if(stack.empty())return -1;
        return stack.top().second;
    }
};

/**
 * Your MinStack object will be instantiated and called as such:
 * MinStack* obj = new MinStack();
 * obj->push(val);
 * obj->pop();
 * int param_3 = obj->top();
 * int param_4 = obj->getMin();
 */
```

24 Swap Nodes in Pairs (link)

Description

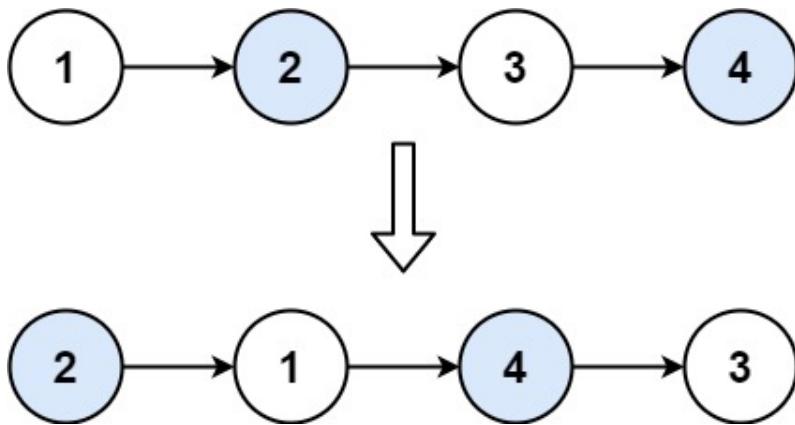
Given a linked list, swap every two adjacent nodes and return its head. You must solve the problem without modifying the values in the list's nodes (i.e., only nodes themselves may be changed.)

Example 1:

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

Output: [2,1,4,3]

Explanation:



Example 2:

Input: head = []

Output: []

Example 3:

Input: head = [1]

Output: [1]

Example 4:

Input: head = [1,2,3]

Output: [2,1,3]

Constraints:

- The number of nodes in the list is in the range [0, 100].
- $0 \leq \text{Node.val} \leq 100$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* swapPairs(ListNode* head) {
        if(!head || !head->next){
            return head;
        }

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

        while(curr){
            if(!curr->next || !curr->next->next){
                break;
            }
            ListNode* firstNode = curr->next;
            ListNode* secondNode = curr->next->next;
            ListNode* remainingLL = curr->next->next->next;

            secondNode->next = firstNode;
            firstNode->next = remainingLL;
            curr->next = secondNode;
            curr = firstNode;
        }

        ListNode* finalHead = dummyHead->next;
        delete dummyHead;
        return finalHead;
    }
};
```

[20 Valid Parentheses \(link\)](#)

Description

Given a string s containing just the characters `'(`, `)`, `{`, `}`, `[` and `]`, determine if the input string is valid.

An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.
3. Every close bracket has a corresponding open bracket of the same type.

Example 1:

Input: $s = "()"$

Output: true

Example 2:

Input: $s = "()[]{}"$

Output: true

Example 3:

Input: $s = "[]"$

Output: false

Example 4:

Input: $s = "(())"$

Output: true

Example 5:

Input: $s = "([])"$

Output: false

Constraints:

- $1 \leq s.length \leq 10^4$
- s consists of parentheses only `'()' [] {}'`.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    bool isValid(string s) {
        stack<char>brackets;
        if(s.size()%2==1) return false;

        for(int i=0; i<s.size();i++){
            char currentCh = s[i];
            if(currentCh == '(' ||currentCh == '[' ||currentCh == '{'){
                brackets.push(s[i]);
            }else{
                if(brackets.empty())return false;
                char topBracket = brackets.top();
                brackets.pop();
                if( currentCh == ')' && topBracket != '(' ){
                    return false;
                }else if( currentCh == '}' && topBracket != '{'){
                    return false;
                }else if( currentCh == ']' && topBracket != '[' ){
                    return false;
                }
            }
        }
        if(!brackets.empty())return false;
        return true;
    }
};
```

[287 Find the Duplicate Number \(link\)](#)

Description

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 using 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 = [3,3,3,3,3]
Output: 3
```

Constraints:

- $1 \leq n \leq 10^5$
- `nums.length == n + 1`
- $1 \leq \text{nums}[i] \leq n$
- 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?

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int findDuplicate(vector<int>& nums) {
        int slow = nums[0];
        int fast = nums[0];

        while(fast < nums.size()){
            slow = nums[slow];
            fast = nums[nums[fast]];
            if(slow == fast){
                slow = nums[0];
                while(slow != fast){
                    slow = nums[slow];
                    fast = nums[fast];
                }
                break;
            }
        }
        return slow;
    }
};
```

25 Reverse Nodes in k-Group (link)

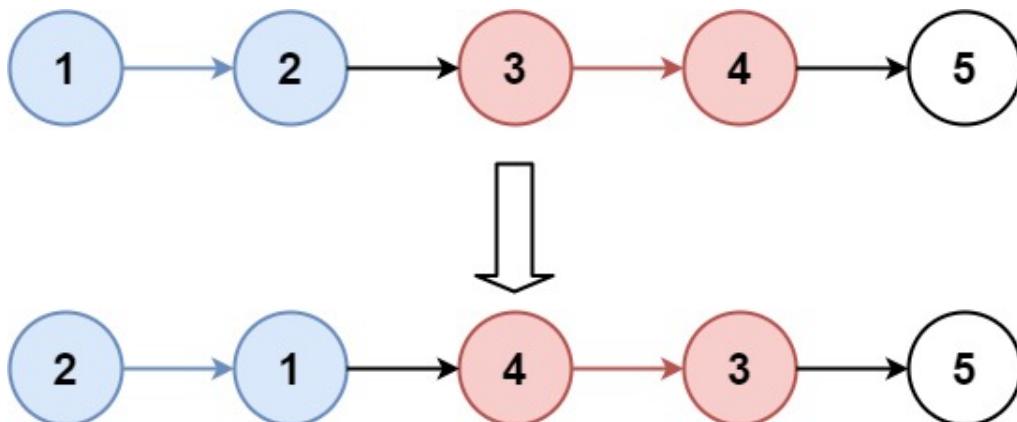
Description

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

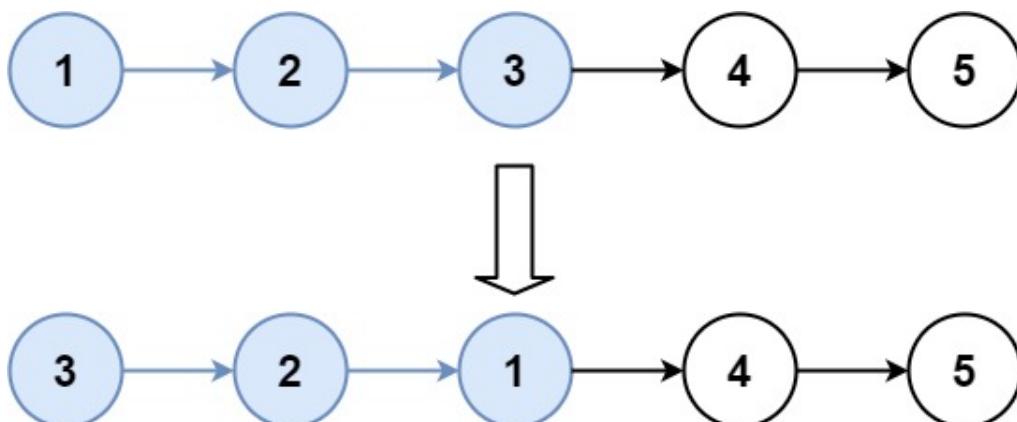
Example 1:



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]

Constraints:

- The number of nodes in the list is n .
- $1 \leq k \leq n \leq 5000$
- $0 \leq \text{Node.val} \leq 1000$

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

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
    ListNode* reverseKNodes(ListNode* head){
        if(!head || !head->next) return head;

        ListNode* prev = nullptr;
        ListNode* curr = head;
        ListNode* nextNode = head->next;

        while(curr){
            curr->next = prev;
            prev = curr;
            curr = nextNode;
            if(nextNode) nextNode = nextNode->next;
        }
        return prev;
    }
public:
    ListNode* reverseKGroup(ListNode* head, int k) {
        ListNode* dummy = new ListNode(-1);
        dummy->next = head;
        ListNode* curr = dummy;

        while(curr->next){
            ListNode* prev = curr;
            ListNode* kGroupEnd = curr->next;
            int i=0;
            while(i<k && curr){
                curr = curr->next;
                i++;
            }
            if(!curr) break;

            ListNode* kGroupStart = curr;
            prev->next = kGroupStart;
            ListNode* nextGroup = curr->next;
            curr->next = nullptr;
            ListNode* lastNodeGroup = kGroupEnd;
            kGroupEnd = reverseKNodes(kGroupEnd);
            lastNodeGroup->next = nextGroup;
            curr = lastNodeGroup;
        }

        return dummy->next;
    }
};
```

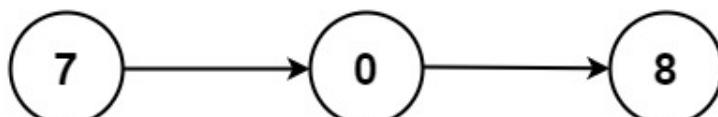
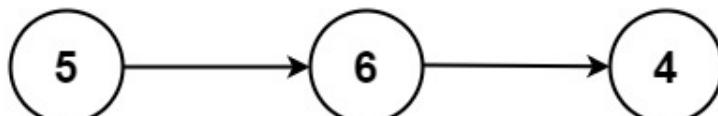
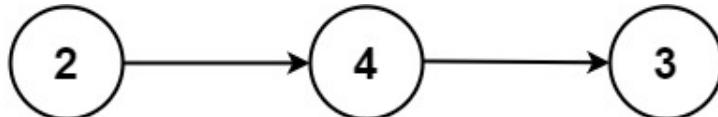
[2 Add Two Numbers \(link\)](#)

Description

You are given two **non-empty** linked lists representing two non-negative integers. The digits are stored in **reverse order**, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

Example 1:



Input: l1 = [2,4,3], l2 = [5,6,4]

Output: [7,0,8]

Explanation: 342 + 465 = 807.

Example 2:

Input: l1 = [0], l2 = [0]

Output: [0]

Example 3:

Input: l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9]

Output: [8,9,9,9,0,0,0,1]

Constraints:

- The number of nodes in each linked list is in the range [1, 100].
- $0 \leq \text{Node.val} \leq 9$
- It is guaranteed that the list represents a number that does not have leading zeros.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {

    ListNode* reverseLL(ListNode* head) {
        if (!head || !head->next) {
            return head;
        }

        ListNode* prevNode = nullptr;
        ListNode* currNode = head;
        ListNode* nextNode = head->next;

        while (currNode != nullptr) {
            currNode->next = prevNode;
            prevNode = currNode;
            currNode = nextNode;
            if (nextNode->next)
                nextNode = nextNode->next;
        }
        return prevNode;
    }

public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        int carry = 0;
        ListNode* curr1 = l1;
        ListNode* curr2 = l2;
        ListNode* newHead = new ListNode(-1); // Dummy Head
        ListNode* currAns = newHead;
        while (curr1 != nullptr || curr2 != nullptr || carry == 1) {
            int currentDigitInteger = carry, sum = 0;
            if (curr1 && curr2) {
                sum = curr1->val + curr2->val;
                currentDigitInteger = (sum+ carry) % 10;
            } else if (curr1 && !curr2) {
                sum = curr1->val;
                currentDigitInteger = (sum+ carry) % 10;
            } else if (!curr1 && curr2) {
                sum = curr2->val;
                currentDigitInteger = (sum+ carry) % 10;
            }
            ListNode* currDigit = new ListNode(currentDigitInteger);
            currAns->next = currDigit;
            currAns = currAns->next;
            if (curr1)
                curr1 = curr1->next;
            if (curr2)
                curr2 = curr2->next;

            if (sum + carry > 9 )
                carry = 1;
            else
                carry = 0;
        }

        return newHead->next;
    }
}
```

}; }

[4073 Lexicographically Smallest String After Reverse \(link\)](#)

Description

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

You must perform **exactly** one operation by choosing any integer k such that $1 \leq k \leq n$ and either:

- reverse the **first** k characters of s , or
- reverse the **last** k characters of s .

Return the **lexicographically smallest** string that can be obtained after **exactly** one such operation.

Example 1:

Input: $s = "dcab"$

Output: "acdb"

Explanation:

- Choose $k = 3$, reverse the first 3 characters.
- Reverse "dca" to "acd", resulting string $s = "acdb"$, which is the lexicographically smallest string achievable.

Example 2:

Input: $s = "abba"$

Output: "aabb"

Explanation:

- Choose $k = 3$, reverse the last 3 characters.
- Reverse "bba" to "abb", so the resulting string is "aabb", which is the lexicographically smallest string achievable.

Example 3:

Input: $s = "zxy"$

Output: "xzy"

Explanation:

- Choose $k = 2$, reverse the first 2 characters.
- Reverse "zx" to "xz", so the resulting string is "xzy", which is the lexicographically smallest string achievable.

Constraints:

- $1 \leq n == s.length \leq 1000$
- s consists of lowercase English letters.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    string lexSmallest(string s) {
        string ans = s;
        int n = s.size();

        for (int k = 1; k <= n; k++) {
            string temp = s;
            reverse(temp.begin(), temp.begin() + k);
            ans = min(ans, temp);
        }

        for (int k = 1; k <= n; k++) {
            string temp = s;
            reverse(temp.end() - k, temp.end());
            ans = min(ans, temp);
        }

        return ans;
    }
};
```

23 Merge k Sorted Lists (link)

Description

You are given an array of k linked-lists `lists`, each linked-list is sorted in ascending order.

Merge all the linked-lists into one sorted linked-list and return it.

Example 1:

```
Input: lists = [[1,4,5],[1,3,4],[2,6]]
Output: [1,1,2,3,4,4,5,6]
Explanation: The linked-lists are:
[
    1->4->5,
    1->3->4,
    2->6
]
merging them into one sorted linked list:
1->1->2->3->4->4->5->6
```

Example 2:

```
Input: lists = []
Output: []
```

Example 3:

```
Input: lists = [[]]
Output: []
```

Constraints:

- $k == \text{lists.length}$
- $0 \leq k \leq 10^4$
- $0 \leq \text{lists}[i].length \leq 500$
- $-10^4 \leq \text{lists}[i][j] \leq 10^4$
- **lists[i]** is sorted in **ascending order**.
- The sum of `lists[i].length` will not exceed 10^4 .

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
    class customComparator {
public:
    bool operator()(ListNode* a, ListNode* b) {
        return a->val > b->val; // 3,5
    }
};

public:
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        priority_queue<ListNode*, vector<ListNode*>, customComparator> minHeap;

        for (int i = 0; i < lists.size(); i++) {
            if (lists[i] != nullptr) {
                minHeap.push(lists[i]);
            }
        }

        ListNode* ans = new ListNode(-1); // Dummy Head
        ListNode* curr = ans;

        while (!minHeap.empty()) {
            ListNode* minNode = minHeap.top();
            minHeap.pop();
            ListNode* minNodeNext = minNode->next;
            curr->next = minNode;
            curr = curr->next;
            if(minNodeNext != nullptr)minHeap.push(minNodeNext);
        }

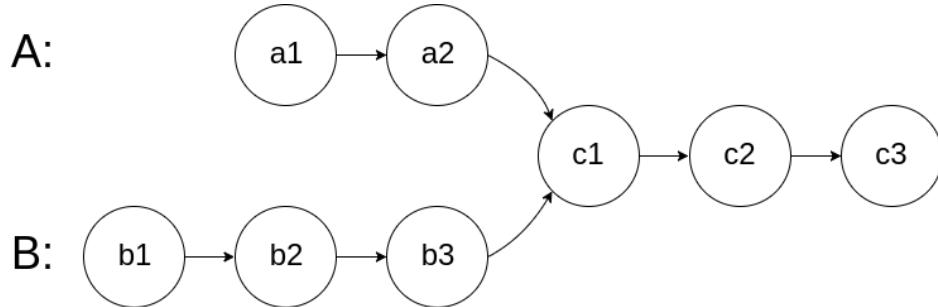
        return ans->next;
    }
};
```

160 Intersection of Two Linked Lists (link)

Description

Given the heads of two singly linked-lists headA and headB, return *the node at which the two lists intersect*. If the two linked lists have no intersection at all, return `null`.

For example, the following two linked lists begin to intersect at node c1:



The test cases are generated such that there are no cycles anywhere in the entire linked structure.

Note that the linked lists must **retain their original structure** after the function returns.

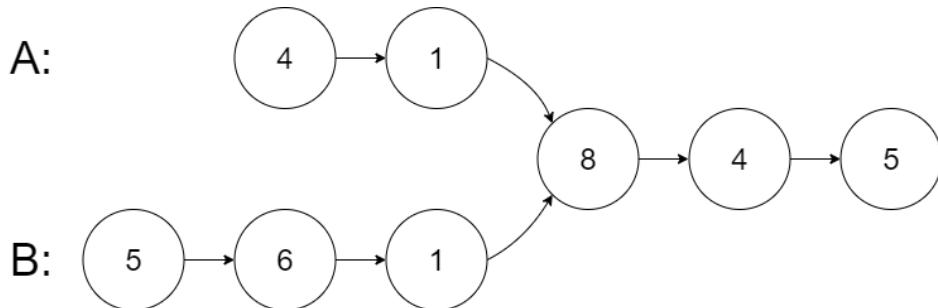
Custom Judge:

The inputs to the **judge** are given as follows (your program is **not** given these inputs):

- `intersectVal` - The value of the node where the intersection occurs. This is `0` if there is no intersected node.
- `listA` - The first linked list.
- `listB` - The second linked list.
- `skipA` - The number of nodes to skip ahead in `listA` (starting from the head) to get to the intersected node.
- `skipB` - The number of nodes to skip ahead in `listB` (starting from the head) to get to the intersected node.

The judge will then create the linked structure based on these inputs and pass the two heads, `headA` and `headB` to your program. If you correctly return the intersected node, then your solution will be **accepted**.

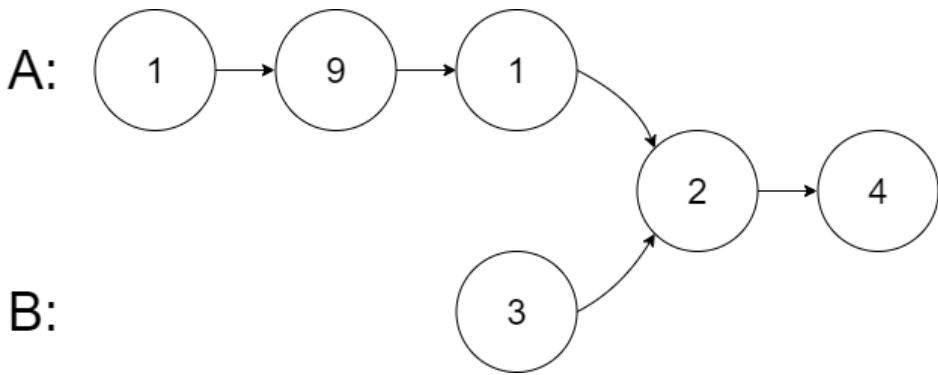
Example 1:



Input: `intersectVal = 8, listA = [4,1,8,4,5], listB = [5,6,1,8,4,5], skipA = 2, skipB = 3`
Output: Intersected at '8'

Explanation: The intersected node's value is 8 (note that this must not be 0 if the two lists : From the head of A, it reads as [4,1,8,4,5]. From the head of B, it reads as [5,6,1,8,4,5]. The - Note that the intersected node's value is not 1 because the nodes with value 1 in A and B (2nd and 3rd) are different).

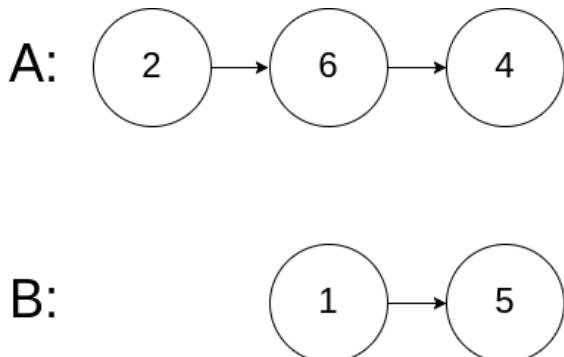
Example 2:



Input: intersectVal = 2, listA = [1,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1
Output: Intersected at '2'

Explanation: The intersected node's value is 2 (note that this must not be 0 if the two lists : From the head of A, it reads as [1,9,1,2,4]. From the head of B, it reads as [3,2,4]. There are

Example 3:



Input: intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2
Output: No intersection

Explanation: From the head of A, it reads as [2,6,4]. From the head of B, it reads as [1,5]. So they do not intersect.

Explanation: The two lists do not intersect, so return null.

Constraints:

- The number of nodes of listA is in the m.
- The number of nodes of listB is in the n.
- $1 \leq m, n \leq 3 * 10^4$
- $1 \leq \text{Node.val} \leq 10^5$
- $0 \leq \text{skipA} \leq m$
- $0 \leq \text{skipB} \leq n$
- intersectVal is 0 if listA and listB do not intersect.
- intersectVal == listA[skipA] == listB[skipB] if listA and listB intersect.

Follow up: Could you write a solution that runs in $O(m + n)$ time and use only $O(1)$ memory?

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
    int calculateLengthLL(ListNode* head){
        ListNode* curr = head;
        if(!head) return 0;
        if(!head->next) return 1;
        int lengthLL = 0;
        while(curr!=nullptr){
            lengthLL++;
            curr = curr->next;
        }
        return lengthLL;
    }
public:
    ListNode* getIntersectionNode(ListNode* headA, ListNode* headB) {
        int length1 = 0, length2 = 0;
        length1 = calculateLengthLL(headA);
        length2 = calculateLengthLL(headB);

        int difference = abs(length1 - length2);

        ListNode* curr1 = headA;
        ListNode* curr2 = headB;

        while (length1 > length2) {
            curr1 = curr1->next;
            length1--;
        }
        while (length1 < length2) {
            curr2 = curr2->next;
            length2--;
        }

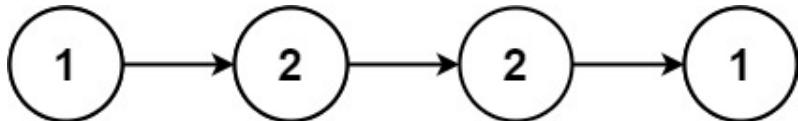
        while (curr1 != nullptr && curr2 != nullptr) {
            if (curr1 == curr2)
                return curr1;
            curr1 = curr1->next;
            curr2 = curr2->next;
        }
        return nullptr;
    }
};
```

234 Palindrome Linked List ([link](#))

Description

Given the head of a singly linked list, return `true` *if it is a palindrome or false otherwise.*

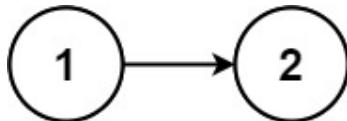
Example 1:



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

Output: true

Example 2:



Input: head = [1,2]

Output: false

Constraints:

- The number of nodes in the list is in the range $[1, 10^5]$.
- $0 \leq \text{Node.val} \leq 9$

Follow up: Could you do it in $O(n)$ time and $O(1)$ space?

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
    ListNode* reverseLL(ListNode* head) {
        if (!head || !head->next)
            return head;
        ListNode* currNode = head;
        ListNode* prevNode = nullptr;
        ListNode* nextNode = head->next;

        while (currNode != nullptr) {
            currNode->next = prevNode;
            prevNode = currNode;
            currNode = nextNode;
            if (nextNode)
                nextNode = nextNode->next;
        }
        return prevNode;
    }

    void printLL(ListNode* head){
        ListNode* curr = head;
        while(curr){
            cout<<curr->val<<" -> ";
            curr=curr->next;
        }
        cout<<"\n";
    }

public:
    bool isPalindrome(ListNode* head) {
        if (!head)
            return false;
        if (!head->next)
            return true;
        ListNode* fast = head;
        ListNode* slow = head;

        while (fast != nullptr && fast->next != nullptr) {
            fast = fast->next->next;
            slow = slow->next;
        }
        ListNode* middle = slow;
        ListNode* reversedMidHead = reverseLL(slow);

        ListNode* first = head;
        ListNode* second = reversedMidHead;
        while (first != slow && second != nullptr) {
            if (first->val != second->val) {
                return false;
            }
            first = first->next;
            second = second->next;
        }
        return true;
    }
};
```

138 Copy List with Random Pointer (link)

Description

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** 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 $X.\text{random} \rightarrow Y$, then for the corresponding two nodes x and y in the copied list, $x.\text{random} \rightarrow y$.

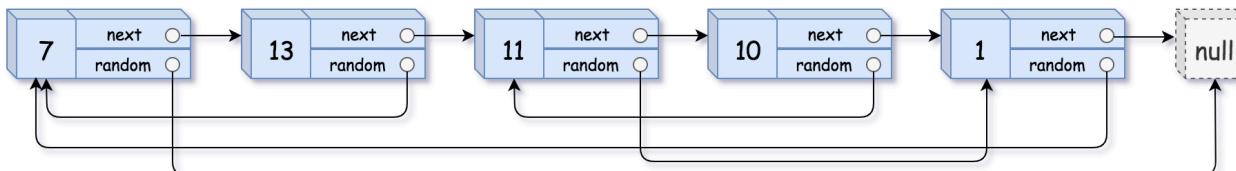
Return *the head of the copied linked list*.

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.

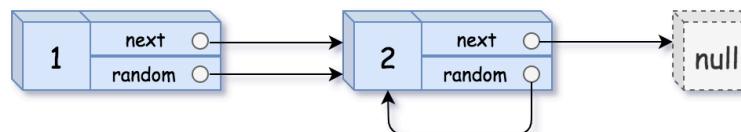
Your code will **only** be given the head of the original linked list.

Example 1:



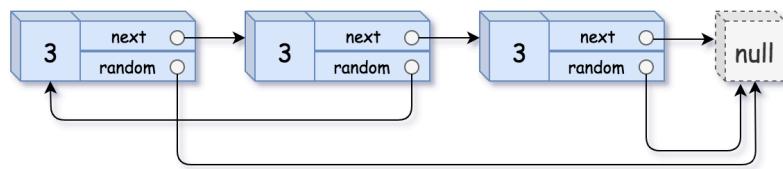
```
Input: head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
Output: [[7,null],[13,0],[11,4],[10,2],[1,0]]
```

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]]
```

Constraints:

- $0 \leq n \leq 1000$
- $-10^4 \leq \text{Node.val} \leq 10^4$
- `Node.random` is null or is pointing to some node in the linked list.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
// Definition for a Node.
class Node {
public:
    int val;
    Node* next;
    Node* random;

    Node(int _val) {
        val = _val;
        next = NULL;
        random = NULL;
    }
};

class Solution {
public:
    Node* copyRandomList(Node* head) {
        Node* originalNode = head;
        while (originalNode != nullptr) {
            Node* originalNodeNext = originalNode->next;
            originalNode->next = new Node(originalNode->val);
            originalNode = originalNode->next;
            originalNode->next = originalNodeNext;
            originalNode = originalNode->next;
        }

        originalNode = head;
        while (originalNode != nullptr) {
            if (originalNode->random != nullptr) {
                originalNode->next->random = originalNode->random->next;
            }
            originalNode = originalNode->next->next;
        }

        Node* newHead = new Node(-1);
        originalNode = head;
        Node* newCurr = newHead;

        while (originalNode != nullptr) {
            Node* originalNodeNext = originalNode->next;
            originalNode->next = originalNode->next->next;
            newCurr->next = originalNodeNext;
            newCurr = newCurr->next;
            originalNode = originalNode->next;
        }

        // Node* curr = newHead;
        // while(curr!=nullptr){
        //     cout<<curr->val<<" ";
        //     curr = curr->next;
        // }

        return newHead->next;
    }
};
```

143 Reorder List (link)

Description

You are given the head of a singly linked-list. The list can be represented as:

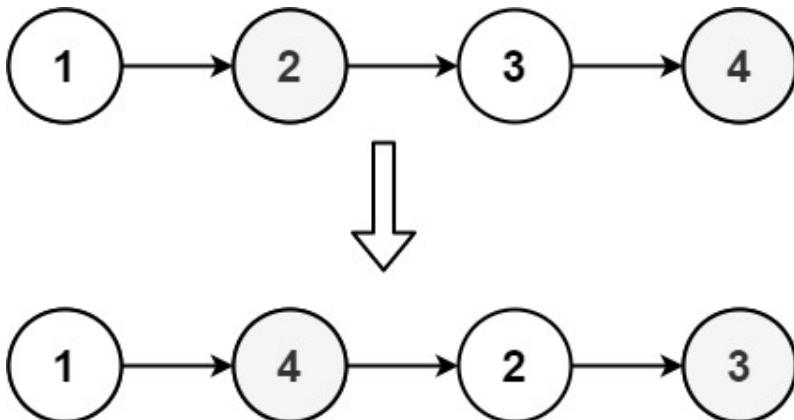
```
L0 → L1 → ... → Ln - 1 → Ln
```

Reorder the list to be on the following form:

```
L0 → Ln → L1 → Ln - 1 → L2 → Ln - 2 → ...
```

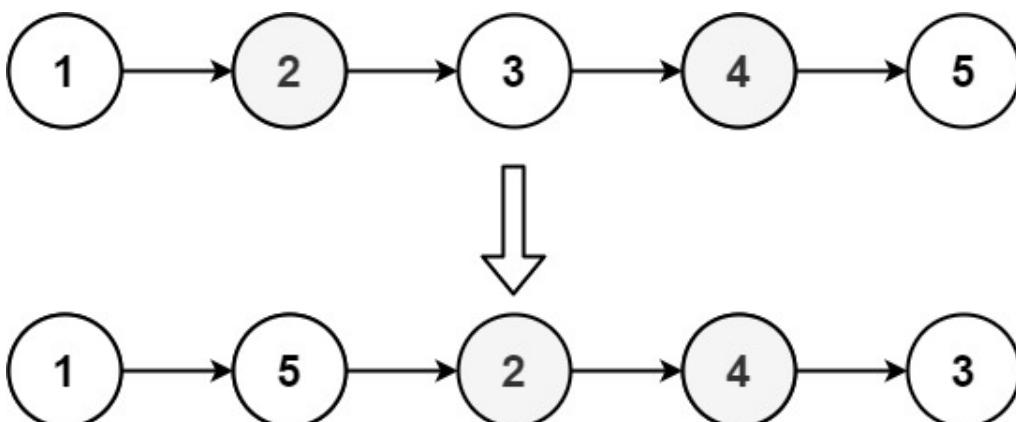
You may not modify the values in the list's nodes. Only nodes themselves may be changed.

Example 1:



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

Example 2:



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

Constraints:

- The number of nodes in the list is in the range $[1, 5 * 10^4]$.
- $1 \leq \text{Node.val} \leq 1000$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
    ListNode* reverseLL(ListNode*& node) {
        if (node == nullptr || node->next == nullptr) {
            return node;
        }
        ListNode* currNode = node;
        ListNode* prevNode = nullptr;
        ListNode* nextNode = node->next;

        while (currNode != nullptr) {
            currNode->next = prevNode;
            prevNode = currNode;
            currNode = nextNode;
            if (nextNode != nullptr)
                nextNode = nextNode->next;
        }

        return prevNode;
    }

public:
    void reorderList(ListNode* head) {
        if (head == nullptr || head->next == nullptr || head->next->next == nullptr) {
            return;
        }
        ListNode* slow = head;
        ListNode* fast = head;
        ListNode* firstListEnd = nullptr;

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

        if (firstListEnd != nullptr)
            firstListEnd->next = nullptr;

        ListNode* firstHead = head;
        ListNode* secondHead = reverseLL(slow);
        while (secondHead != nullptr) {

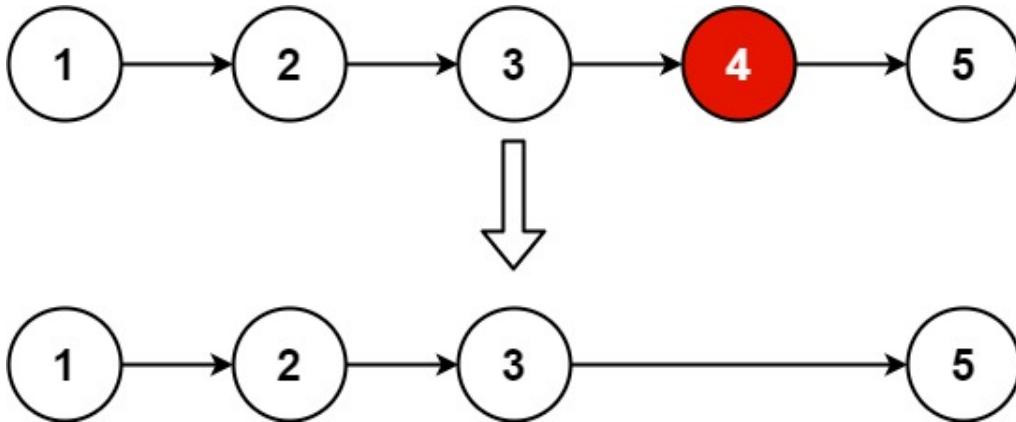
            ListNode* firstHeadNext = firstHead->next;
            ListNode* secondHeadNext = secondHead->next;
            firstHead->next = secondHead;
            if (firstHeadNext == nullptr)
                break;
            secondHead->next = firstHeadNext;
            secondHead = secondHeadNext;
            firstHead = firstHeadNext;
        }
    }
};
```

19 Remove Nth Node From End of List ([link](#))

Description

Given the head of a linked list, remove the n^{th} node from the end of the list and return its head.

Example 1:



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

Example 2:

Input: head = [1], n = 1
Output: []

Example 3:

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

Constraints:

- The number of nodes in the list is sz .
 - $1 \leq sz \leq 30$
 - $0 \leq \text{Node.val} \leq 100$
 - $1 \leq n \leq sz$

Follow up: Could you do this in one pass?

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode* dummyHead = new ListNode(-1);
        dummyHead->next = head;
        int count = 1;

        ListNode* slow = dummyHead;
        ListNode* fast = head;

        while(fast != nullptr && fast->next != nullptr && count < n){
            count++;
            fast = fast->next;
        }

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

        ListNode* nodeToRemove = slow->next;
        slow->next = slow->next->next;
        delete nodeToRemove;

        return dummyHead->next;
    }
};
```

146 LRU Cache ([link](#))

Description

Design a data structure that follows the constraints of a [Least Recently Used \(LRU\) cache](#).

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:

- $1 \leq \text{capacity} \leq 3000$
- $0 \leq \text{key} \leq 10^4$
- $0 \leq \text{value} \leq 10^5$
- At most $2 * 10^5$ calls will be made to get and put.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
struct DLL{
    int storedKey;
    int storedVal;
    DLL *next;
    DLL* prev;
    DLL(int key,int val){
        this->storedKey = key;
        this->storedVal = val;
        this->next = nullptr;
        this->prev = nullptr;
    }
};

class LRUCache {
    unordered_map<int,DLL*>keyMap;
    int currentCapacity;
    int maxCapacity;
    DLL* head;
    DLL* tail;

    void updateTail(DLL *&tail,DLL *&currNode){
        tail->prev->next = currNode;
        currNode->prev = tail->prev;
        currNode->next = tail;
        tail->prev = currNode;
    }

    void removeNode(DLL * &tail,DLL* &nodeToDelete){
        nodeToDelete->prev->next = nodeToDelete->next;
        nodeToDelete->next->prev = nodeToDelete->prev;
    }
public:
    LRUCache(int capacity) {
        head = new DLL(-1,-1);
        tail = new DLL(-1,-1);
        head->next = tail;
        tail->prev = head;
        this->maxCapacity = capacity;
        currentCapacity = 0;
    }

    int get(int key) {
        if(!keyMap.count(key)){
            return -1;
        }
        DLL* currNode = keyMap[key];
        removeNode(tail,currNode);
        updateTail(tail,currNode);
        return currNode->storedVal;
    }

    void put(int key, int value) {
        if(!keyMap.count(key)){
            // Not Found
            if(currentCapacity < maxCapacity){
                // Valid cache size
                DLL* newNode = new DLL(key,value);
                currentCapacity++;
                keyMap[key] = newNode;
                updateTail(tail,newNode);
            }else{
                // Invalid cache Size
                DLL* leastRecentlyUsedNode = head->next;
                removeNode(tail,leastRecentlyUsedNode);
                keyMap.erase(leastRecentlyUsedNode->storedKey); // HERE Delete Stored key not r
                delete leastRecentlyUsedNode;
            }
        }
        DLL* newNode = new DLL(key,value);
        keyMap[key] = newNode;
    }
}
```

```
        updateTail(tail,newNode);
    }
} else{
    // Found
    DLL* currNode = keyMap[key];
    currNode->storedVal = value;
    removeNode(tail,currNode);
    updateTail(tail,currNode);
}
};

/***
 * 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);
 */
```

[4045 Longest Balanced Subarray I \(link\)](#)

Description

You are given an integer array `nums`.

A **subarray** is called **balanced** if the number of **distinct even** numbers in the subarray is equal to the number of **distinct odd** numbers.

Return the length of the **longest** balanced subarray.

Example 1:

Input: `nums = [2,5,4,3]`

Output: 4

Explanation:

- The longest balanced subarray is `[2, 5, 4, 3]`.
- It has 2 distinct even numbers `[2, 4]` and 2 distinct odd numbers `[5, 3]`. Thus, the answer is 4.

Example 2:

Input: `nums = [3,2,2,5,4]`

Output: 5

Explanation:

- The longest balanced subarray is `[3, 2, 2, 5, 4]`.
- It has 2 distinct even numbers `[2, 4]` and 3 distinct odd numbers `[3, 5]`. Thus, the answer is 5.

Example 3:

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

Output: 3

Explanation:

- The longest balanced subarray is `[2, 3, 2]`.
- It has 1 distinct even number `[2]` and 1 distinct odd number `[3]`. Thus, the answer is 3.

Constraints:

- $1 \leq \text{nums.length} \leq 1500$
- $1 \leq \text{nums}[i] \leq 10^5$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int longestBalanced(vector<int>& nums) {
        int maxLength = 0;

        for(int i=0; i<nums.size(); i++){
            unordered_set<int>odd;
            unordered_set<int>even;

            for(int j=i; j<nums.size(); j++){
                int currNumber = nums[j];

                if(currNumber % 2 == 0){
                    even.insert(currNumber);
                }else{
                    odd.insert(currNumber);
                }

                if(even.size() == odd.size()){
                    int currSize = j-i+1;
                    maxLength = max(maxLength, currSize);
                }
            }
        }

        return maxLength;
    }
};
```

[4080 Smallest Missing Multiple of K \(link\)](#)

Description

Given an integer array `nums` and an integer `k`, return the **smallest positive multiple** of `k` that is **missing** from `nums`.

A **multiple** of `k` is any positive integer divisible by `k`.

Example 1:

Input: `nums` = [8,2,3,4,6], `k` = 2

Output: 10

Explanation:

The multiples of `k` = 2 are 2, 4, 6, 8, 10, 12... and the smallest multiple missing from `nums` is 10.

Example 2:

Input: `nums` = [1,4,7,10,15], `k` = 5

Output: 5

Explanation:

The multiples of `k` = 5 are 5, 10, 15, 20... and the smallest multiple missing from `nums` is 5.

Constraints:

- `1 <= nums.length <= 100`
- `1 <= nums[i] <= 100`
- `1 <= k <= 100`

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int missingMultiple(vector<int>& nums, int k) {
        unordered_set<int> set(nums.begin(), nums.end());

        for(int i=k; i<=nums.size()*k + k; i+=k){
            if(!set.count(i)){
                return i;
            }
        }
        return k;
    }
};
```

21 Merge Two Sorted Lists ([link](#))

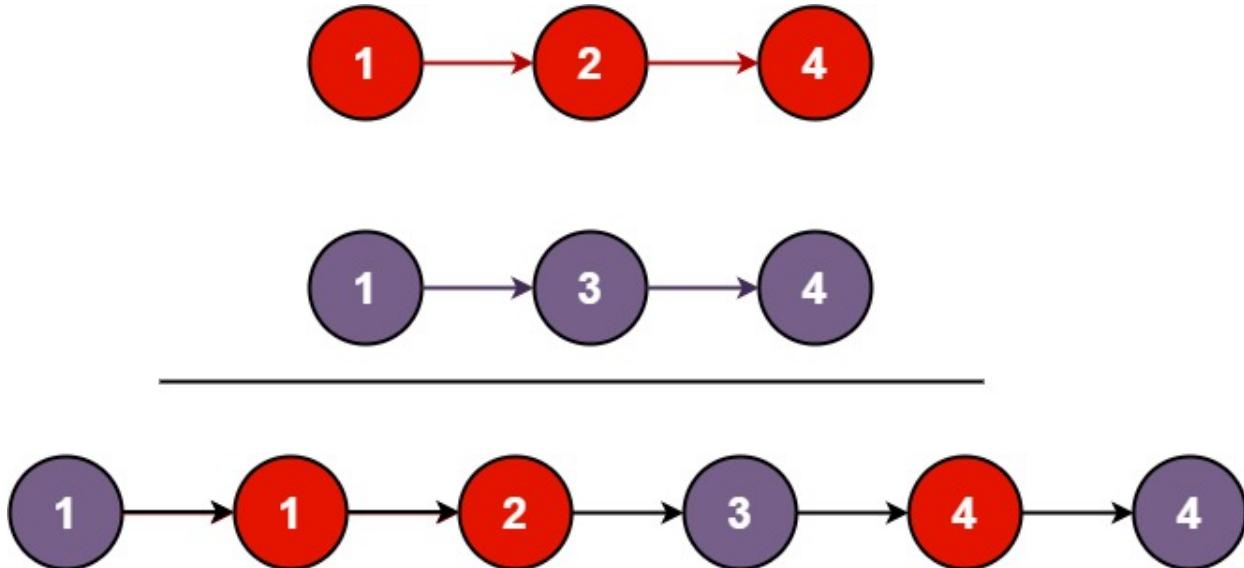
Description

You are given the heads of two sorted linked lists `list1` and `list2`.

Merge the two lists into one **sorted** list. The list should be made by splicing together the nodes of the first two lists.

Return *the head of the merged linked list*.

Example 1:



```
Input: list1 = [1,2,4], list2 = [1,3,4]
Output: [1,1,2,3,4,4]
```

Example 2:

```
Input: list1 = [], list2 = []
Output: []
```

Example 3:

```
Input: list1 = [], list2 = [0]
Output: [0]
```

Constraints:

- The number of nodes in both lists is in the range $[0, 50]$.
- $-100 \leq \text{Node.val} \leq 100$
- Both `list1` and `list2` are sorted in **non-decreasing** order.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
        ListNode* list3 = new ListNode(-1); // Dummy Node
        ListNode* newList = list3;
        while(list1 != nullptr && list2 != nullptr){
            if(list1->val < list2->val){
                newList->next = list1;
                newList = newList->next;
                list1 = list1->next;
            }else{
                newList->next = list2;
                newList = newList->next;
                list2 = list2->next;
            }
        }

        if(list1 != nullptr){
            newList->next = list1;
        }
        if(list2 != nullptr){
            newList->next = list2;
        }

        return list3->next;
    }
};
```

141 Linked List Cycle (link)

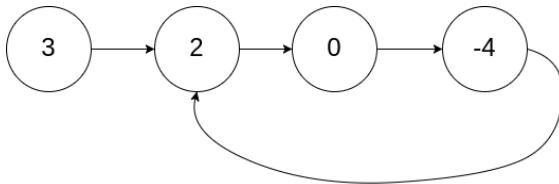
Description

Given head, the head of a linked list, determine if the linked list has a cycle in it.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. **Note that pos is not passed as a parameter.**

Return true if there is a cycle in the linked list. Otherwise, return false.

Example 1:

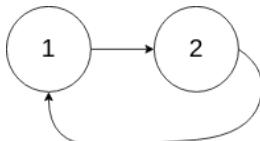


Input: head = [3,2,0,-4], pos = 1

Output: true

Explanation: There is a cycle in the linked list, where the tail connects to the 1st node (0-indexed).

Example 2:



Input: head = [1,2], pos = 0

Output: true

Explanation: There is a cycle in the linked list, where the tail connects to the 0th node.

Example 3:



Input: head = [1], pos = -1

Output: false

Explanation: There is no cycle in the linked list.

Constraints:

- The number of the nodes in the list is in the range $[0, 10^4]$.
- $-10^5 \leq \text{Node.val} \leq 10^5$
- pos is -1 or a valid index in the linked-list.

Follow up: Can you solve it using O(1) (i.e. constant) memory?

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    bool hasCycle(ListNode* head) {
        if (head == nullptr || head->next == nullptr)
            return false;

        ListNode* slow = head;
        ListNode* fast = head;

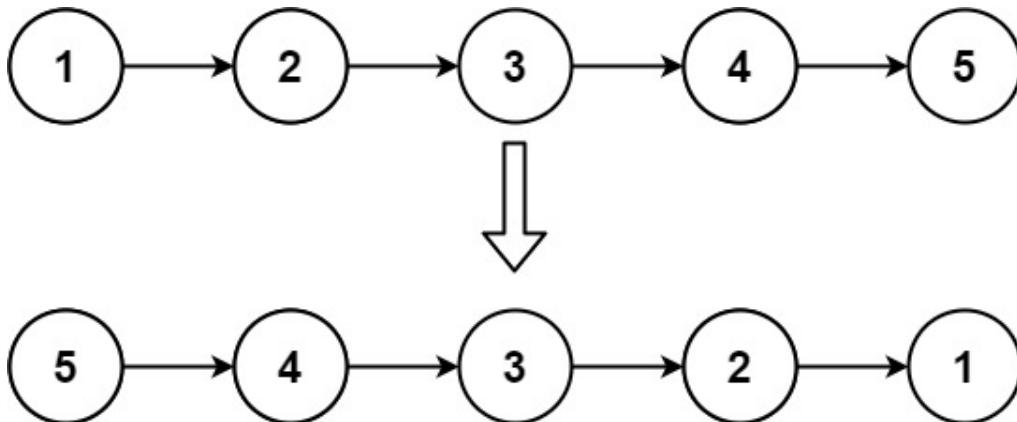
        while (fast != nullptr && fast->next != nullptr) {
            slow = slow->next;
            fast = fast->next->next;
            if (slow == fast) {
                return true;
            }
        }
        return false;
    }
};
```

206 Reverse Linked List (link)

Description

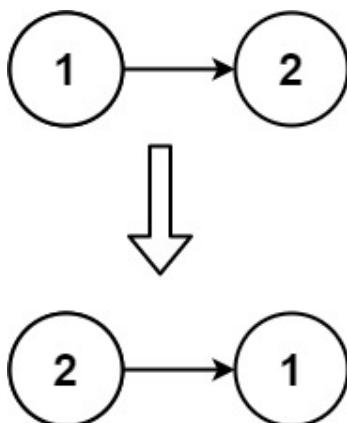
Given the head of a singly linked list, reverse the list, and return *the reversed list*.

Example 1:



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

Example 2:



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

Example 3:

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

Constraints:

- The number of nodes in the list is in the range $[0, 5000]$.
- $-5000 \leq \text{Node.val} \leq 5000$

Follow up: A linked list can be reversed either iteratively or recursively. Could you implement both?

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        if(head == nullptr || head->next == nullptr){
            return head;
        }
        ListNode* prevNode = nullptr;
        ListNode* currNode = head;
        ListNode* nextNode = head->next;

        while(currNode != nullptr){
            currNode->next = prevNode;
            prevNode = currNode;
            currNode = nextNode;
            if(nextNode != nullptr)nextNode = nextNode->next;
        }
        return prevNode;
    }
};
```

[57 Insert Interval \(link\)](#)

Description

You are given an array of non-overlapping intervals `intervals` where `intervals[i] = [starti, endi]` represent the start and the end of the i^{th} interval and `intervals` is sorted in ascending order by `starti`. You are also given an interval `newInterval = [start, end]` that represents the start and end of another interval.

Insert `newInterval` into `intervals` such that `intervals` is still sorted in ascending order by `starti` and `intervals` still does not have any overlapping intervals (merge overlapping intervals if necessary).

Return `intervals` after the insertion.

Note that you don't need to modify `intervals` in-place. You can make a new array and return it.

Example 1:

```
Input: intervals = [[1,3],[6,9]], newInterval = [2,5]
Output: [[1,5],[6,9]]
```

Example 2:

```
Input: intervals = [[1,2],[3,5],[6,7],[8,10],[12,16]], newInterval = [4,8]
Output: [[1,2],[3,10],[12,16]]
Explanation: Because the new interval [4,8] overlaps with [3,5],[6,7],[8,10].
```

Constraints:

- $0 \leq \text{intervals.length} \leq 10^4$
- $\text{intervals}[i].length == 2$
- $0 \leq \text{start}_i \leq \text{end}_i \leq 10^5$
- `intervals` is sorted by `starti` in **ascending** order.
- `newInterval.length == 2`
- $0 \leq \text{start} \leq \text{end} \leq 10^5$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    vector<vector<int>> insert(vector<vector<int>>& intervals,
                                vector<int>& newInterval) {
        vector<vector<int>> mergedIntervals;
        int i = 0;
        while (i < intervals.size() && intervals[i][1] < newInterval[0]) {
            mergedIntervals.push_back(intervals[i]);
            i++;
        }

        // EDGE CASE
        while (i < intervals.size() && intervals[i][0] <= newInterval[1]) {
            newInterval[0] = min(newInterval[0], intervals[i][0]);
            newInterval[1] = max(newInterval[1], intervals[i][1]);
            i++;
        }

        mergedIntervals.push_back(newInterval);

        while (i < intervals.size()) {
            mergedIntervals.push_back(intervals[i]);
            i++;
        }

        return mergedIntervals;
    }
};
```

121 Best Time to Buy and Sell Stock ([link](#))

Description

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

You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.

Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return 0.

Example 1:

Input: `prices = [7,1,5,3,6,4]`

Output: 5

Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = $6 - 1 = 5$. Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

Example 2:

Input: `prices = [7,6,4,3,1]`

Output: 0

Explanation: In this case, no transactions are done and the max profit = 0.

Constraints:

- $1 \leq \text{prices.length} \leq 10^5$
- $0 \leq \text{prices}[i] \leq 10^4$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int maxProfit = 0, minPrice = INT_MAX;
        for(int i=0;i<prices.size();i++){
            minPrice = min(minPrice,prices[i]);
            maxProfit = max(maxProfit,prices[i] - minPrice);
        }
        return maxProfit;
    }
};
```

5 Longest Palindromic Substring (link)

Description

Given a string s , return *the longest palindromic substring* in s .

Example 1:

```
Input: s = "babad"
Output: "bab"
Explanation: "aba" is also a valid answer.
```

Example 2:

```
Input: s = "cbbd"
Output: "bb"
```

Constraints:

- $1 \leq s.length \leq 1000$
- s consist of only digits and English letters.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
    pair<int,int> getMaxSubstr(string &s, int left, int right){
        while(left >=0 && right < s.size() && s[left] == s[right]){
            left--;
            right++;
        }
        return {left+1, (right-1) - (left+1) + 1};
    }
public:
    string longestPalindrome(string s) {
        int maxLength = 0;
        string longestStr = "";

        for(int currentCharIndex=0;currentCharIndex<s.size();currentCharIndex++){
            // <startIdx, lengthSubstr>
            pair<int,int> evenLength = getMaxSubstr(s,currentCharIndex,currentCharIndex);
            pair<int,int> oddLength = getMaxSubstr(s,currentCharIndex,currentCharIndex+1);
            int currentLength = max(evenLength.second,oddLength.second);
            if(maxLength < currentLength){
                maxLength = currentLength;
                if(evenLength.second == maxLength){
                    longestStr = s.substr(evenLength.first,evenLength.second);
                }else{
                    longestStr = s.substr(oddLength.first,oddLength.second);
                }
            }
        }
        return longestStr;
    }
};
```

153 Find Minimum in Rotated Sorted Array ([link](#))

Description

Suppose an array of length n sorted in ascending order is **rotated** between 1 and n times. For example, the array `nums = [0,1,2,4,5,6,7]` might become:

- `[4,5,6,7,0,1,2]` if it was rotated 4 times.
- `[0,1,2,4,5,6,7]` if it was rotated 7 times.

Notice that **rotating** an array `[a[0], a[1], a[2], ..., a[n-1]]` 1 time results in the array `[a[n-1], a[0], a[1], a[2], ..., a[n-2]]`.

Given the sorted rotated array `nums` of **unique** elements, return *the minimum element of this array*.

You must write an algorithm that runs in $O(\log n)$ time.

Example 1:

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

Output: 1

Explanation: The original array was `[1,2,3,4,5]` rotated 3 times.

Example 2:

Input: `nums = [4,5,6,7,0,1,2]`

Output: 0

Explanation: The original array was `[0,1,2,4,5,6,7]` and it was rotated 4 times.

Example 3:

Input: `nums = [11,13,15,17]`

Output: 11

Explanation: The original array was `[11,13,15,17]` and it was rotated 4 times.

Constraints:

- $n == \text{nums.length}$
- $1 \leq n \leq 5000$
- $-5000 \leq \text{nums}[i] \leq 5000$
- All the integers of `nums` are **unique**.
- `nums` is sorted and rotated between 1 and n times.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int findMin(vector<int>& nums) {
        int start = 0, end = nums.size()-1;

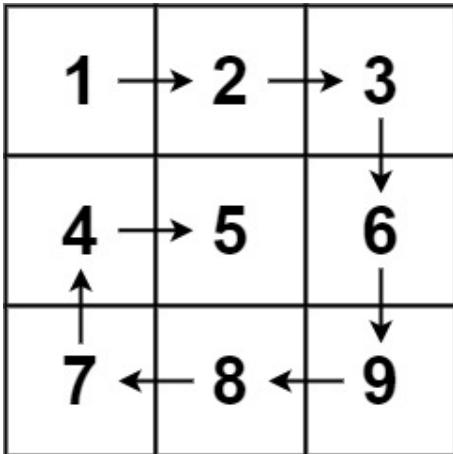
        while(start < end){
            int mid = start + (end-start)/2;
            if(nums[mid] < nums[end]){
                end = mid;
            }else if(nums[mid] >= nums[end]){
                start = mid+1;
            }
        }
        return nums[end];
    }
};
```

[54 Spiral Matrix \(link\)](#)

Description

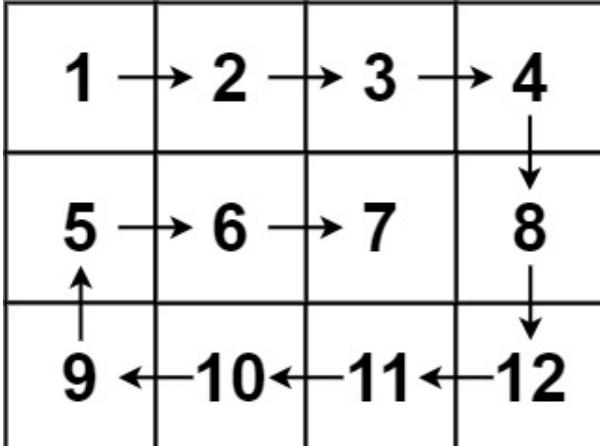
Given an $m \times n$ matrix, return *all elements of the matrix in spiral order*.

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:

- $m == \text{matrix.length}$
- $n == \text{matrix}[i].length$
- $1 \leq m, n \leq 10$
- $-100 \leq \text{matrix}[i][j] \leq 100$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    vector<int> spiralOrder(vector<vector<int>>& matrix) {
        int startRow = 0, endRow = matrix.size() - 1, startCol = 0,
            endCol = matrix[0].size() - 1;
        vector<int> ans;
        while (startRow <= endRow && startCol <= endCol) {
            for (int i = startCol; i <= endCol; i++) {
                ans.push_back(matrix[startRow][i]);
            }
            startRow++;
            if(startRow <= endRow && startCol <= endCol){
                for (int i = startRow; i <= endRow; i++) {
                    ans.push_back(matrix[i][endCol]);
                }
            endCol--;
            if(startRow <= endRow && startCol <= endCol){
                for (int i = endCol; i >= startCol; i--) {
                    ans.push_back(matrix[endRow][i]);
                }
            endRow--;
            if(startRow <= endRow && startCol <= endCol){
                for (int i = endRow; i >= startRow; i--) {
                    ans.push_back(matrix[i][startCol]);
                }
            startCol++;
            }
            return ans;
        }
    };
};
```

[125 Valid Palindrome \(link\)](#)

Description

A phrase is a **palindrome** if, after converting all uppercase letters into lowercase letters and removing all non-alphanumeric characters, it reads the same forward and backward. Alphanumeric characters include letters and numbers.

Given a string s , return `true` *if it is a palindrome*, or `false` *otherwise*.

Example 1:

```
Input: s = "A man, a plan, a canal: Panama"
Output: true
Explanation: "amanaplanacanalpanama" is a palindrome.
```

Example 2:

```
Input: s = "race a car"
Output: false
Explanation: "raceacar" is not a palindrome.
```

Example 3:

```
Input: s = " "
Output: true
Explanation: s is an empty string "" after removing non-alphanumeric characters.
Since an empty string reads the same forward and backward, it is a palindrome.
```

Constraints:

- $1 \leq s.length \leq 2 * 10^5$
- s consists only of printable ASCII characters.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    bool isPalindrome(string s) {
        int left = 0, right = s.size()-1;
        for(int i=0; i<s.size();i++){
            if(s[i] >= 'A' && s[i] <= 'Z'){
                s[i] = (char)((int)(s[i] - 'A') + 'a');
            }
        }

        while(left < right){
            while(left < right && !isalnum(s[left]))left++;
            while(right > left && !isalnum(s[right]))right--;
            if(s[left] != s[right])return false;
            left++;right--;
        }

        return true;
    }
};
```

[424 Longest Repeating Character Replacement \(link\)](#)

Description

You are given a string s and an integer k . You can choose any character of the string and change it to any other uppercase English character. You can perform this operation at most k times.

Return *the length of the longest substring containing the same letter you can get after performing the above operations.*

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 "AABBBA".
The substring "BBBB" has the longest repeating letters, which is 4.
There may exists other ways to achieve this answer too.
```

Constraints:

- $1 \leq s.length \leq 10^5$
- s consists of only uppercase English letters.
- $0 \leq k \leq s.length$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int characterReplacement(string s, int k) {
        int maxFreqChar = 0, maxLengthSubstr = 0, currWindowSize = 0, left=0;
        unordered_map<char,int> freqMap; // WORST CASE -> 26 size O(1)

        for(int right = 0; right < s.size(); right++){
            freqMap[s[right]]++;
            currWindowSize++;

            maxFreqChar = max(maxFreqChar, freqMap[s[right]]);

            while(currWindowSize - maxFreqChar > k){
                // INVALID WINDOW
                currWindowSize--;
                freqMap[s[left]]--;
                left++;
            }

            if(currWindowSize - maxFreqChar <= k){
                maxLengthSubstr = max(maxLengthSubstr, currWindowSize);
            }
        }
        return maxLengthSubstr;
    };
};
```

[209 Minimum Size Subarray Sum \(link\)](#)

Description

Given an array of positive integers `nums` and a positive integer `target`, return *the minimal length of a subarray whose sum is greater than or equal to target*. If there is no such subarray, return `0` instead.

Example 1:

```
Input: target = 7, nums = [2,3,1,2,4,3]
Output: 2
```

Explanation: The subarray [4,3] has the minimal length under the problem constraint.

Example 2:

```
Input: target = 4, nums = [1,4,4]
Output: 1
```

Example 3:

```
Input: target = 11, nums = [1,1,1,1,1,1,1,1]
Output: 0
```

Constraints:

- $1 \leq \text{target} \leq 10^9$
- $1 \leq \text{nums.length} \leq 10^5$
- $1 \leq \text{nums}[i] \leq 10^4$

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

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int minSubArrayLen(int target, vector<int>& nums) {
        int left = 0, right = 0, currentSum = 0, minLength = INT_MAX;

        while(right < nums.size()){
            currentSum += nums[right];
            if(currentSum < target){
                right++; continue;
            }
            while(left <= right && currentSum >= target){
                minLength = min(minLength, right-left+1);
                currentSum -= nums[left];
                left++;
            }
            right++;
        }

        if(minLength == INT_MAX) return 0;
        return minLength;
    }
};
```

[73 Set Matrix Zeroes \(link\)](#)

Description

Given an $m \times n$ integer matrix `matrix`, if an element is 0, set its entire row and column to 0's.

You must do it [in place](#).

Example 1:

1	1	1
1	0	1
1	1	1

1	0	1
0	0	0
1	0	1

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

Example 2:

0	1	2	0
3	4	5	2
1	3	1	5

0	0	0	0
0	4	5	0
0	3	1	0

```
Input: matrix = [[0,1,2,0],[3,4,5,2],[1,3,1,5]]
Output: [[0,0,0,0],[0,4,5,0],[0,3,1,0]]
```

Constraints:

- $m == \text{matrix.length}$
- $n == \text{matrix[0].length}$
- $1 \leq m, n \leq 200$
- $-2^{31} \leq \text{matrix}[i][j] \leq 2^{31} - 1$

Follow up:

- A straightforward solution using $O(mn)$ space is probably a bad idea.
- A simple improvement uses $O(m + n)$ space, but still not the best solution.
- Could you devise a constant space solution?

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    void setZeroes(vector<vector<int>>& matrix) {
        bool row0 = false, col0 = false;
        for (int col = 0; col < matrix[0].size(); col++) {
            if (matrix[0][col] == 0){
                row0 = true;
                break;
            }
        }
        for (int row = 0; row < matrix.size(); row++) {
            if (matrix[row][0] == 0){
                col0 = true;
                break;
            }
        }

        for(int row = 1; row<matrix.size();row++){
            for(int col = 1; col<matrix[0].size();col++){
                if(matrix[row][col] == 0){
                    matrix[0][col] = 0;
                    matrix[row][0] = 0;
                }
            }
        }

        for(int i=1;i<matrix.size();i++){
            for(int j=1; j<matrix[0].size();j++){
                if(matrix[i][0]==0 || matrix[0][j]==0){
                    matrix[i][j]=0;
                }
            }
        }

        if(row0){
            for(int i=0;i<matrix[0].size();i++){
                matrix[0][i] = 0;
            }
        }
        if(col0){
            for(int i=0;i<matrix.size();i++){
                matrix[i][0] = 0;
            }
        }

    }
};
```

[49 Group Anagrams \(\[link\]\(#\)\)](#)

Description

Given an array of strings `strs`, group the anagrams together. You can return the answer in **any order**.

Example 1:

Input: `strs = ["eat", "tea", "tan", "ate", "nat", "bat"]`

Output: `[["bat"], ["nat", "tan"], ["ate", "eat", "tea"]]`

Explanation:

- There is no string in `strs` that can be rearranged to form "bat".
- The strings "nat" and "tan" are anagrams as they can be rearranged to form each other.
- The strings "ate", "eat", and "tea" are anagrams as they can be rearranged to form each other.

Example 2:

Input: `strs = [""]`

Output: `[[""]]`

Example 3:

Input: `strs = ["a"]`

Output: `[["a"]]`

Constraints:

- $1 \leq \text{strs.length} \leq 10^4$
- $0 \leq \text{strs[i].length} \leq 100$
- `strs[i]` consists of lowercase English letters.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    vector<vector<string>> groupAnagrams(vector<string>& strs) {
        unordered_map<string, vector<string>>anagramList;

        for(int i=0; i<strs.size();i++){
            string currentString = strs[i];
            sort(strs[i].begin(),strs[i].end());
            anagramList[strs[i]].push_back(currentString);
        }
        vector<vector<string>>ans;
        for(auto &iterator:anagramList){
            ans.push_back(iterator.second);
        }
        return ans;
    }
};
```

56 Merge Intervals (link)

Description

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

Example 1:

Input: $\text{intervals} = [[1,3], [2,6], [8,10], [15,18]]$

Output: $[[1,6], [8,10], [15,18]]$

Explanation: Since intervals $[1,3]$ and $[2,6]$ overlap, merge them into $[1,6]$.

Example 2:

Input: $\text{intervals} = [[1,4], [4,5]]$

Output: $[[1,5]]$

Explanation: Intervals $[1,4]$ and $[4,5]$ are considered overlapping.

Example 3:

Input: $\text{intervals} = [[4,7], [1,4]]$

Output: $[[1,7]]$

Explanation: Intervals $[1,4]$ and $[4,7]$ are considered overlapping.

Constraints:

- $1 \leq \text{intervals.length} \leq 10^4$
- $\text{intervals}[i].length == 2$
- $0 \leq \text{start}_i \leq \text{end}_i \leq 10^4$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        vector<vector<int>> ans;
        int currentInterval = 0, finalMax = 0;
        sort(intervals.begin(), intervals.end());

        while (currentInterval < intervals.size()) {
            if (ans.empty()) {
                ans.push_back(intervals[0]);
                currentInterval++;
                continue;
            }
            if (ans.back()[1] >= intervals[currentInterval][0]) {
                while (currentInterval < intervals.size() &&
                    ans.back()[1] >= intervals[currentInterval][0]) {

                    ans.back()[1] = max(ans.back()[1], intervals[currentInterval][1]);
                    currentInterval++;
                }
            } else {
                ans.push_back(intervals[currentInterval]);
                currentInterval++;
            }
        }
        return ans;
    }
};
```

33 Search in Rotated Sorted Array (link)

Description

There is an integer array `nums` sorted in ascending order (with **distinct** values).

Prior to being passed to your function, `nums` is **possibly left rotated** at an unknown index `k` ($1 \leq k < \text{nums.length}$) such that the resulting array is `[nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]` (**0-indexed**). For example, `[0,1,2,4,5,6,7]` might be left rotated by 3 indices and become `[4,5,6,7,0,1,2]`.

Given the array `nums` **after** the possible rotation and an integer target, return *the index of target if it is in `nums`, or -1 if it is not in `nums`*.

You must write an algorithm with $O(\log n)$ runtime complexity.

Example 1:

```
Input: nums = [4,5,6,7,0,1,2], target = 0
Output: 4
```

Example 2:

```
Input: nums = [4,5,6,7,0,1,2], target = 3
Output: -1
```

Example 3:

```
Input: nums = [1], target = 0
Output: -1
```

Constraints:

- $1 \leq \text{nums.length} \leq 5000$
- $-10^4 \leq \text{nums}[i] \leq 10^4$
- All values of `nums` are **unique**.
- `nums` is an ascending array that is possibly rotated.
- $-10^4 \leq \text{target} \leq 10^4$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        // 4 5 6 7 0 1 2
        int start = 0, end = nums.size() - 1;

        while(start <= end){
            int mid = start + (end - start)/2;

            if(nums[mid]==target) return mid;
            else if(nums[mid] <= nums[end]){
                if(nums[mid] <= target && target <= nums[end]){
                    start = mid+1;
                }else{
                    end = mid-1;
                }
            }else{
                if(nums[start] <= target && target <= nums[mid]){
                    end = mid-1;
                }else{
                    start = mid+1;
                }
            }
        }

        return -1;
    }
};
```

[4068 Sum of Elements With Frequency Divisible by K \(link\)](#)

Description

You are given an integer array `nums` and an integer `k`.

Return an integer denoting the **sum** of all elements in `nums` whose **frequency** is divisible by `k`, or 0 if there are no such elements.

Note: An element is included in the sum **exactly** as many times as it appears in the array if its total frequency is divisible by `k`.

Example 1:

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

Output: 16

Explanation:

- The number 1 appears once (odd frequency).
- The number 2 appears twice (even frequency).
- The number 3 appears four times (even frequency).
- The number 4 appears once (odd frequency).

So, the total sum is $2 + 2 + 3 + 3 + 3 + 3 = 16$.

Example 2:

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

Output: 0

Explanation:

There are no elements that appear an even number of times, so the total sum is 0.

Example 3:

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

Output: 12

Explanation:

- The number 1 appears once.
- The number 2 appears once.
- The number 3 appears once.
- The number 4 appears three times.

So, the total sum is $4 + 4 + 4 = 12$.

Constraints:

- $1 \leq \text{nums.length} \leq 100$
- $1 \leq \text{nums}[i] \leq 100$
- $1 \leq k \leq 100$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int sumDivisibleByK(vector<int>& nums, int k) {
        unordered_map<int,int> freq;
        for(int i=0; i<nums.size();i++){
            freq[nums[i]]++;
        }
        int totalSum=0;
        for(auto &it:freq){
            if(it.second % k == 0){
                totalSum += (it.first * it.second);
            }
        }
        return totalSum;
    }
};
```

[4003 Longest Fibonacci Subarray \(link\)](#)

Description

You are given an array of **positive** integers `nums`.

A **Fibonacci** array is a contiguous sequence whose third and subsequent terms each equal the sum of the two preceding terms.

Return the length of the longest **Fibonacci subarray** in `nums`.

Note: Subarrays of length 1 or 2 are always **Fibonacci**.

Example 1:

Input: `nums` = [1,1,1,1,2,3,5,1]

Output: 5

Explanation:

The longest Fibonacci subarray is `nums[2..6] = [1, 1, 2, 3, 5]`.

[1, 1, 2, 3, 5] is Fibonacci because $1 + 1 = 2$, $1 + 2 = 3$, and $2 + 3 = 5$.

Example 2:

Input: `nums` = [5,2,7,9,16]

Output: 5

Explanation:

The longest Fibonacci subarray is `nums[0..4] = [5, 2, 7, 9, 16]`.

[5, 2, 7, 9, 16] is Fibonacci because $5 + 2 = 7$, $2 + 7 = 9$, and $7 + 9 = 16$.

Example 3:

Input: `nums` = [1000000000,1000000000,1000000000]

Output: 2

Explanation:

The longest Fibonacci subarray is `nums[1..2] = [1000000000, 1000000000]`.

[1000000000, 1000000000] is Fibonacci because its length is 2.

Constraints:

- $3 \leq \text{nums.length} \leq 10^5$
- $1 \leq \text{nums}[i] \leq 10^9$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int longestSubarray(vector<int>& nums) {
        if(nums.size()<=2) return nums.size();
        int maxCount = 0, count=2;

        for(int i=2;i<nums.size();i++){
            if(nums[i-1]+nums[i-2] == nums[i]) count++;
            else{
                count=2;
            }
            maxCount = max(maxCount, count);
        }
        return maxCount;
    }
};
```

[4052 Equal Score Substrings \(link\)](#)

Description

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

The **score** of a string is the sum of the positions of its characters in the alphabet, where ' a ' = 1, ' b ' = 2, ..., ' z ' = 26.

Determine whether there exists an index i such that the string can be split into two **non-empty substrings** $s[0..i]$ and $s[(i + 1)..(n - 1)]$ that have **equal** scores.

Return `true` if such a split exists, otherwise return `false`.

Example 1:

Input: $s = \text{"adcb"}$

Output: `true`

Explanation:

Split at index $i = 1$:

- Left substring = $s[0..1] = \text{"ad"}$ with score = $1 + 4 = 5$
- Right substring = $s[2..3] = \text{"cb"}$ with score = $3 + 2 = 5$

Both substrings have equal scores, so the output is `true`.

Example 2:

Input: $s = \text{"bace"}$

Output: `false`

Explanation:

No split produces equal scores, so the output is `false`.

Constraints:

- $2 \leq s.\text{length} \leq 100$
- s consists of lowercase English letters.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    bool scoreBalance(string s) {
        vector<int>prefix(s.size());
        vector<int>suffix(s.size());

        for(int i=0;i<s.size();i++){
            if(i==0){
                prefix[i]=(s[i]-'a')+1;continue;
            }
            prefix[i]=((s[i]-'a')+1)+prefix[i-1];
        }
        for(int i=s.size()-1;i>=0;i--){
            if(i==s.size()-1){
                suffix[i]=(s[i]-'a')+1;
                continue;
            }
            suffix[i]=((s[i]-'a')+1)+suffix[i+1];
        }

        for(int i=0; i<s.size();i++){
            if(i==0){
                if(prefix[i]==suffix[i+1])return true;
                continue;
            }
            if(i==s.size()-1){
                if(prefix[i]==0){return true;}
                continue;
            }
            if(prefix[i]==suffix[i+1])return true;
        }

        return false;
    }
};
```

560 Subarray Sum Equals K (link)

Description

Given an array of integers `nums` and an integer `k`, return *the total number of subarrays whose sum equals to k*.

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

Example 1:

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

Example 2:

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

Constraints:

- $1 \leq \text{nums.length} \leq 2 * 10^4$
- $-1000 \leq \text{nums}[i] \leq 1000$
- $-10^7 \leq k \leq 10^7$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int subarraySum(vector<int>& nums, int k) {
        unordered_map<int,int>prefixSum;
        prefixSum[0]++;
        int count=0;
        int currentSum = 0;
        for(int i=0; i<nums.size();i++){
            currentSum += nums[i];
            if(prefixSum.count(currentSum - k)){
                count += prefixSum[currentSum - k];
            }
            prefixSum[currentSum]++;
        }
        return count;
    }
};
```

[48 Rotate Image \(link\)](#)

Description

You are given an $n \times n$ 2D matrix representing an image, rotate the image by **90** degrees (clockwise).

You have to rotate the image **in-place**, which means you have to modify the input 2D matrix directly. **DO NOT** allocate another 2D matrix and do the rotation.

Example 1:

1	2	3
4	5	6
7	8	9

→

7	4	1
8	5	2
9	6	3

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

Example 2:

5	1	9	11
2	4	8	10
13	3	6	7
15	14	12	16

→

15	13	2	5
14	3	4	1
12	6	8	9
16	7	10	11

```
Input: matrix = [[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,12,16]]  
Output: [[15,13,2,5],[14,3,4,1],[12,6,8,9],[16,7,10,11]]
```

Constraints:

- $n == \text{matrix.length} == \text{matrix[i].length}$
- $1 \leq n \leq 20$
- $-1000 \leq \text{matrix[i][j]} \leq 1000$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    void rotate(vector<vector<int>>& matrix) {
        for(int row=0;row<matrix.size();row++){
            for(int col = row; col <matrix[0].size(); col++){
                swap(matrix[row][col],matrix[col][row]);
            }
        }
        for(int row = 0; row<matrix.size();row++){
            reverse(matrix[row].begin(),matrix[row].end());
        }
    }
};
```

[238 Product of Array Except Self \(link\)](#)

Description

Given an integer array `nums`, return *an array answer such that answer[i] is equal to the product of all the elements of nums except nums[i]*.

The product of any prefix or suffix of `nums` is **guaranteed** to fit in a **32-bit** integer.

You must write an algorithm that runs in $O(n)$ time and without using the division operation.

Example 1:

```
Input: nums = [1,2,3,4]
Output: [24,12,8,6]
```

Example 2:

```
Input: nums = [-1,1,0,-3,3]
Output: [0,0,9,0,0]
```

Constraints:

- $2 \leq \text{nums.length} \leq 10^5$
- $-30 \leq \text{nums}[i] \leq 30$
- The input is generated such that `answer[i]` is **guaranteed** to fit in a **32-bit** integer.

Follow up: Can you solve the problem in $O(1)$ extra space complexity? (The output array **does not** count as extra space for space complexity analysis.)

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    vector<int> productExceptSelf(vector<int>& nums) {
        int size = nums.size();
        vector<int>ans(size);

        // Store Prefix in ans
        for(int i=0;i<size;i++){
            if(i==0){
                ans[i] = nums[i];
                continue;
            }
            ans[i] = ans[i-1]*nums[i];
        }

        int suffix = 1;
        for(int i=size-1;i>=0;i--){
            if(i==size-1){
                ans[i] = ans[i-1];
                suffix = nums[i];
                continue;
            }
            if(i==0){
                ans[i]=suffix;
                break;
            }
            ans[i] = ans[i-1]*suffix;
            suffix = suffix * nums[i];
        }
        return ans;
    }
};
```

3 Longest Substring Without Repeating Characters (link)

Description

Given a string s , find the length of the **longest substring** without duplicate characters.

Example 1:

Input: $s = \text{"abcabcbb"}$

Output: 3

Explanation: The answer is "abc", with the length of 3. Note that "bca" and "cab" are also correct.

Example 2:

Input: $s = \text{"bbbbbb"}$

Output: 1

Explanation: The answer is "b", with the length of 1.

Example 3:

Input: $s = \text{"pwwkew"}$

Output: 3

Explanation: The answer is "wke", with the length of 3.

Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

Constraints:

- $0 \leq s.\text{length} \leq 5 * 10^4$
- s consists of English letters, digits, symbols and spaces.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        if(s.size()<=1) return s.size();
        int left = 0, right=0;
        unordered_set<int>set;
        int maxLength=-1;

        while(right < s.size()){
            while(set.find(s[right]) != set.end()){
                // FOUND IN SET
                set.erase(s[left]);
                left++;
            }
            set.insert(s[right]);
            maxLength = max(maxLength,right-left+1);
            right++;
        }
        return maxLength;
    }
};
```

11 Container With Most Water (link)

Description

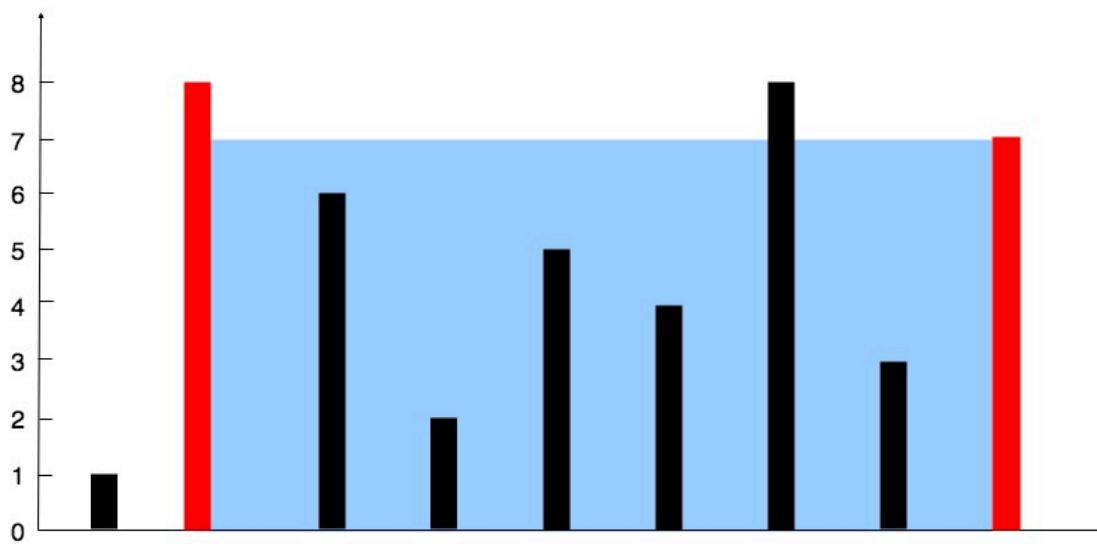
You are given an integer array `height` of length `n`. There are `n` vertical lines drawn such that the two endpoints of the i^{th} line are $(i, 0)$ and $(i, \text{height}[i])$.

Find two lines that together with the x-axis form a container, such that the container contains the most water.

Return *the maximum amount of water a container can store*.

Notice that you may not slant the container.

Example 1:



Input: `height = [1,8,6,2,5,4,8,3,7]`

Output: 49

Explanation: The above vertical lines are represented by array `[1,8,6,2,5,4,8,3,7]`. In this case, the maximum amount of water that can be stored is 49 units.

Example 2:

Input: `height = [1,1]`

Output: 1

Constraints:

- $n == \text{height.length}$
- $2 \leq n \leq 10^5$
- $0 \leq \text{height}[i] \leq 10^4$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int maxArea(vector<int>& height) {
        // Formula: minHeight * (rightIndex - leftIndex)
        int maxArea = -1;
        int start = 0 ,end = height.size()-1;

        while(start < end){
            int currentArea = min(height[start],height[end]) * (end - start);
            maxArea = max(maxArea,currentArea);
            if(height[start] < height[end] ){
                start++;
            }else{
                end--;
            }
        }
        return maxArea;
    }
};
```

1 Two Sum ([link](#))

Description

Given an array of integers `nums` and an integer `target`, return *indices of the two numbers such that they add up to target*.

You may assume that each input would have **exactly one solution**, and you may not use the *same* element twice.

You can return the answer in any order.

Example 1:

```
Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Explanation: Because nums[0] + nums[1] == 9, we return [0, 1].
```

Example 2:

```
Input: nums = [3,2,4], target = 6
Output: [1,2]
```

Example 3:

```
Input: nums = [3,3], target = 6
Output: [0,1]
```

Constraints:

- $2 \leq \text{nums.length} \leq 10^4$
- $-10^9 \leq \text{nums}[i] \leq 10^9$
- $-10^9 \leq \text{target} \leq 10^9$
- **Only one valid answer exists.**

Follow-up: Can you come up with an algorithm that is less than $O(n^2)$ time complexity?

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        unordered_map<int,int>map;
        map[nums[0]]=0;

        for(int i=1; i<nums.size(); i++){
            if(map.find(target - nums[i]) != map.end()){
                return {map[target-nums[i]],i};
            }
            map[nums[i]] = i;
        }
        return {-1,-1};
    }
};
```

[4033 Longest Subsequence With Non-Zero Bitwise XOR \(link\)](#)

Description

You are given an integer array `nums`.

Return the length of the **longest subsequence** in `nums` whose bitwise **XOR** is **non-zero**. If no such **subsequence** exists, return 0.

Example 1:

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

Output: 2

Explanation:

One longest subsequence is `[2, 3]`. The bitwise XOR is computed as $2 \text{ XOR } 3 = 1$, which is non-zero.

Example 2:

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

Output: 3

Explanation:

The longest subsequence is `[2, 3, 4]`. The bitwise XOR is computed as $2 \text{ XOR } 3 \text{ XOR } 4 = 5$, which is non-zero.

Constraints:

- $1 \leq \text{nums.length} \leq 10^5$
- $0 \leq \text{nums}[i] \leq 10^9$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int longestSubsequence(vector<int>& nums) {
        if(nums.size() < 1){
            return 0;
        }
        int totalXOR = 0, totalOR=0;
        for(int i=0; i<nums.size();i++){
            totalXOR ^= nums[i];
            totalOR |= nums[i];
        }
        if(totalXOR != 0){
            return nums.size();
        }
        if(totalOR == 0) return 0;
        return nums.size()-1;
    }
};
```

[4058 Compute Alternating Sum \(link\)](#)

Description

You are given an integer array `nums`.

The **alternating sum** of `nums` is the value obtained by **adding** elements at even indices and **subtracting** elements at odd indices. That is, `nums[0] - nums[1] + nums[2] - nums[3]...`

Return an integer denoting the alternating sum of `nums`.

Example 1:

Input: `nums = [1,3,5,7]`

Output: `-4`

Explanation:

- Elements at even indices are `nums[0] = 1` and `nums[2] = 5` because 0 and 2 are even numbers.
- Elements at odd indices are `nums[1] = 3` and `nums[3] = 7` because 1 and 3 are odd numbers.
- The alternating sum is `nums[0] - nums[1] + nums[2] - nums[3] = 1 - 3 + 5 - 7 = -4`.

Example 2:

Input: `nums = [100]`

Output: `100`

Explanation:

- The only element at even indices is `nums[0] = 100` because 0 is an even number.
- There are no elements on odd indices.
- The alternating sum is `nums[0] = 100`.

Constraints:

- `1 <= nums.length <= 100`
- `1 <= nums[i] <= 100`

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int alternatingSum(vector<int>& nums) {
        if(nums.size() == 1){
            return nums[0];
        }

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

[4005 Maximum Total Subarray Value I \(link\)](#)

Description

You are given an integer array `nums` of length `n` and an integer `k`.

You need to choose **exactly** `k` non-empty subarrays `nums[l..r]` of `nums`. Subarrays may overlap, and the exact same subarray (same `l` and `r`) **can** be chosen more than once.

The **value** of a subarray `nums[l..r]` is defined as: `max(nums[l..r]) - min(nums[l..r])`.

The **total value** is the sum of the **values** of all chosen subarrays.

Return the **maximum** possible total value you can achieve.

Example 1:

Input: `nums = [1,3,2]`, `k = 2`

Output: 4

Explanation:

One optimal approach is:

- Choose `nums[0..1] = [1, 3]`. The maximum is 3 and the minimum is 1, giving a value of $3 - 1 = 2$.
- Choose `nums[0..2] = [1, 3, 2]`. The maximum is still 3 and the minimum is still 1, so the value is also $3 - 1 = 2$.

Adding these gives $2 + 2 = 4$.

Example 2:

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

Output: 12

Explanation:

One optimal approach is:

- Choose `nums[0..3] = [4, 2, 5, 1]`. The maximum is 5 and the minimum is 1, giving a value of $5 - 1 = 4$.
- Choose `nums[0..3] = [4, 2, 5, 1]`. The maximum is 5 and the minimum is 1, so the value is also 4.
- Choose `nums[2..3] = [5, 1]`. The maximum is 5 and the minimum is 1, so the value is again 4.

Adding these gives $4 + 4 + 4 = 12$.

Constraints:

- $1 \leq n == \text{nums.length} \leq 5 * 10^4$
- $0 \leq \text{nums}[i] \leq 10^9$
- $1 \leq k \leq 10^5$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    long long maxTotalValue(vector<int>& nums, int k) {
        long long sum = 0;
        if(nums.size() == 1){return 0;}

        long long maxi = *max_element(nums.begin(),nums.end());
        long long mini = *min_element(nums.begin(),nums.end());

        sum = (maxi - mini) * k;

        return sum;
    }
};
```

[4039 Compute Decimal Representation \(link\)](#)

Description

You are given a **positive** integer n .

A positive integer is a **base-10 component** if it is the product of a single digit from 1 to 9 and a non-negative power of 10. For example, 500, 30, and 7 are **base-10 components**, while 537, 102, and 11 are not.

Express n as a sum of **only** base-10 components, using the **fewest** base-10 components possible.

Return an array containing these **base-10 components** in **descending** order.

Example 1:

Input: $n = 537$

Output: [500,30,7]

Explanation:

We can express 537 as $500 + 30 + 7$. It is impossible to express 537 as a sum using fewer than 3 base-10 components.

Example 2:

Input: $n = 102$

Output: [100,2]

Explanation:

We can express 102 as $100 + 2$. 102 is not a base-10 component, which means 2 base-10 components are needed.

Example 3:

Input: $n = 6$

Output: [6]

Explanation:

6 is a base-10 component.

Constraints:

- $1 \leq n \leq 10^9$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    vector<int> decimalRepresentation(int n) {
        if(n>=1 && n<=9) return {n};
        int power = 0;
        vector<int>ans;
        while(n!=0){
            int remainder = n%10;
            if(remainder != 0){
                if(power == 0){
                    ans.push_back(remainder);
                }else{
                    int pushArr = remainder * (pow(10,power));
                    ans.push_back(pushArr);
                }
            }
            power++;
            n/=10;
        }
        reverse(ans.begin(),ans.end());
        return ans;
    }
};
```

[4009 Bitwise OR of Even Numbers in an Array](#)[\(link\)](#)

Description

You are given an integer array `nums`.

Return the bitwise **OR** of all **even** numbers in the array.

If there are no even numbers in `nums`, return 0.

Example 1:

Input: `nums` = [1,2,3,4,5,6]

Output: 6

Explanation:

The even numbers are 2, 4, and 6. Their bitwise OR equals 6.

Example 2:

Input: `nums` = [7,9,11]

Output: 0

Explanation:

There are no even numbers, so the result is 0.

Example 3:

Input: `nums` = [1,8,16]

Output: 24

Explanation:

The even numbers are 8 and 16. Their bitwise OR equals 24.

Constraints:

- `1 <= nums.length <= 100`
- `1 <= nums[i] <= 100`

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int evenNumberBitwise0Rs(vector<int>& nums) {
        int orn = 0;
        for(auto &el:nums){
            if(el%2==0){
                orn = orn | el;
            }
        }
        return orn;
    }
};
```

1890 Sum of Beauty of All Substrings ([link](#))

Description

The **beauty** of a string is the difference in frequencies between the most frequent and least frequent characters.

- For example, the beauty of "abaacc" is $3 - 1 = 2$.

Given a string s , return *the sum of beauty of all of its substrings*.

Example 1:

Input: $s = \text{"aabcb"}$

Output: 5

Explanation: The substrings with non-zero beauty are $[\text{"aab"}, \text{"aabc"}, \text{"aabcb"}, \text{"abcb"}, \text{"bcb"}]$, each

Example 2:

Input: $s = \text{"aabcbbaa"}$

Output: 17

Constraints:

- $1 \leq s.length \leq 500$
- s consists of only lowercase English letters.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
    int getCurrentBeauty(int freq[])
    {
        int maxFreq = INT_MIN, minFreq = INT_MAX;
        for (int i = 0; i < 26; i++)
        {
            if (freq[i] == 0)
                continue;
            maxFreq = max(maxFreq, freq[i]);
            minFreq = min(minFreq, freq[i]);
        }
        return maxFreq - minFreq;
    }

public:
    int beautySum(string s) {
        int beautyAll = 0;

        for (int i = 0; i < s.size(); i++) {
            int freq[26] = {0};
            for (int j = i; j < s.size(); j++) {
                freq[s[j] - 'a']++;
                string curr = s.substr(i, j - i + 1);
                int currentBeauty = getCurrentBeauty(freq);
                if (currentBeauty > 0) {
                    beautyAll += currentBeauty;
                }
            }
        }
        return beautyAll;
    }
};
```

[4053 Majority Frequency Characters \(link\)](#)

Description

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

The **frequency group** for a value k is the set of characters that appear exactly k times in s .

The **majority frequency group** is the frequency group that contains the largest number of **distinct** characters.

Return a string containing all characters in the majority frequency group, in **any** order. If two or more frequency groups tie for that largest size, pick the group whose frequency k is **larger**.

Example 1:

Input: $s = \text{"aaabbbccddde"}$

Output: "ab"

Explanation:

Frequency (k)	Distinct characters in group	Group size	Majority?
4	{d}	1	No
3	{a, b}	2	Yes
2	{c}	1	No
1	{e}	1	No

Both characters 'a' and 'b' share the same frequency 3, they are in the majority frequency group. "ba" is also a valid answer.

Example 2:

Input: $s = \text{"abcd"}$

Output: "abcd"

Explanation:

Frequency (k)	Distinct characters in group	Group size	Majority?
1	{a, b, c, d}	4	Yes

All characters share the same frequency 1, they are all in the majority frequency group.

Example 3:

Input: $s = \text{"pfpfgi"}$

Output: "fp"

Explanation:

Frequency (k)	Distinct characters in group	Group size	Majority?
2	{p, f}	2	Yes
1	{g, i}	2	No (tied size, lower frequency)

Both characters 'p' and 'f' share the same frequency 2, they are in the majority frequency group. There is a tie in group size with frequency 1, but we pick the higher frequency: 2.

Constraints:

- $1 \leq s.length \leq 100$
- s consists only of lowercase English letters.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    string majorityFrequencyGroup(string s) {
        if(s.size() == 1) return s;
        int freq[26] = {0};
        for(int i=0; i<s.size(); i++){
            freq[s[i] - 'a']++;
        }
        string ans = "";
        map<int,string>map;
        for(int i=0; i<26; i++){
            if(freq[i] == 0) continue;
            map[freq[i]]+= (char)(i+'a');
        }
        unsigned maxLen = 0;
        for(auto &it: map){
            if(it.second.size() >= maxLen ){
                maxLen = it.second.size();
                ans = it.second;
            }
        }

        return ans;
    }
};
```

908 Middle of the Linked List (link)

Description

Given the head of a singly linked list, return *the middle node of the linked list*.

If there are two middle nodes, return **the second middle** node.

Example 1:

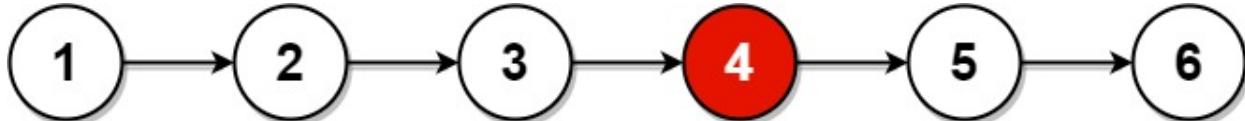


Input: head = [1,2,3,4,5]

Output: [3,4,5]

Explanation: The middle node of the list is node 3.

Example 2:



Input: head = [1,2,3,4,5,6]

Output: [4,5,6]

Explanation: Since the list has two middle nodes with values 3 and 4, we return the second one.

Constraints:

- The number of nodes in the list is in the range [1, 100].
- $1 \leq \text{Node.val} \leq 100$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* middleNode(ListNode* head) {
        if(head->next == nullptr){
            return head;
        }
        ListNode* slow = head;
        ListNode* fast = head;
        while(fast != nullptr && fast->next != nullptr){
            slow = slow->next;
            fast = fast->next->next;
        }
        return slow;
    }
};
```

8 String to Integer (atoi) (link)

Description

Implement the `myAtoi(string s)` function, which converts a string to a 32-bit signed integer.

The algorithm for `myAtoi(string s)` is as follows:

1. **Whitespace:** Ignore any leading whitespace (" ").
2. **Signedness:** Determine the sign by checking if the next character is '-' or '+', assuming positivity if neither present.
3. **Conversion:** Read the integer by skipping leading zeros until a non-digit character is encountered or the end of the string is reached. If no digits were read, then the result is 0.
4. **Rounding:** If the integer is out of the 32-bit signed integer range $[-2^{31}, 2^{31} - 1]$, then round the integer to remain in the range. Specifically, integers less than -2^{31} should be rounded to -2^{31} , and integers greater than $2^{31} - 1$ should be rounded to $2^{31} - 1$.

Return the integer as the final result.

Example 1:

Input: `s = "42"`

Output: 42

Explanation:

The underlined characters are what is read in and the caret is the current reader position.
Step 1: "42" (no characters read because there is no leading whitespace)
Step 2: "42" (no characters read because there is neither a '-' nor '+')
Step 3: "42" ("42" is read in)

Example 2:

Input: `s = "-042"`

Output: -42

Explanation:

Step 1: " -042" (leading whitespace is read and ignored)
Step 2: " -042" ('-' is read, so the result should be negative)
Step 3: " -042" ("042" is read in, leading zeros ignored in the result)

Example 3:

Input: `s = "1337c0d3"`

Output: 1337

Explanation:

Step 1: "1337c0d3" (no characters read because there is no leading whitespace)
Step 2: "1337c0d3" (no characters read because there is neither a '-' nor '+')

Step 3: "1337c0d3" ("1337" is read in; reading stops because the next character is a non-digit)

Example 4:

Input: s = "0-1"

Output: 0

Explanation:

Step 1: "0-1" (no characters read because there is no leading whitespace)

Step 2: "0-1" (no characters read because there is neither a '-' nor '+')

Step 3: "0-1" ("0" is read in; reading stops because the next character is a non-digit)

Example 5:

Input: s = "words and 987"

Output: 0

Explanation:

Reading stops at the first non-digit character 'w'.

Constraints:

- $0 \leq s.length \leq 200$
- s consists of English letters (lower-case and upper-case), digits (0-9), ' ', '+', '−', and '.'.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int myAtoi(string s) {
        int idx = 0;
        long long ans = 0;

        for (auto& ch : s) {
            if (ch != ' ') {
                break;
            }
            idx++;
        }

        bool isPositive = true;
        if (s[idx] == '-') {
            isPositive = false;
            idx++;
        } else if (s[idx] == '+') {
            idx++;
        }
        int sign = isPositive ? +1 : -1;

        while (idx < s.size()) {
            if (s[idx] - '0' >= 0 && s[idx] - '0' <= 9) {
                ans = (ans * 10) + (s[idx] - '0') * sign ;
                if(ans > INT_MAX)ans = INT_MAX;
                if(ans < INT_MIN)ans = INT_MIN;
                idx++;
            }else{
                break;
            }
        }
        return ans;
    }
};
```

237 Delete Node in a Linked List (link)

Description

There is a singly-linked list head and we want to delete a node `node` in it.

You are given the node to be deleted `node`. You will **not be given access** to the first node of head.

All the values of the linked list are **unique**, and it is guaranteed that the given node `node` is not the last node in the linked list.

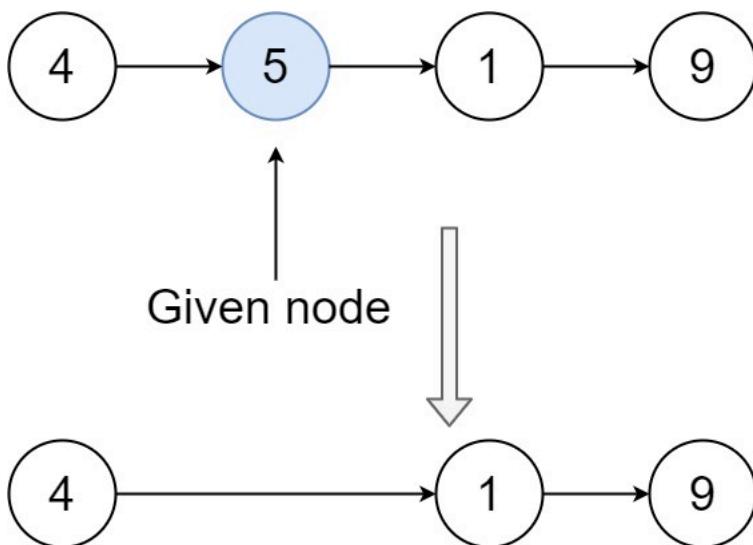
Delete the given node. Note that by deleting the node, we do not mean removing it from memory. We mean:

- The value of the given node should not exist in the linked list.
- The number of nodes in the linked list should decrease by one.
- All the values before `node` should be in the same order.
- All the values after `node` should be in the same order.

Custom testing:

- For the input, you should provide the entire linked list head and the node to be given `node`. `node` should not be the last node of the list and should be an actual node in the list.
- We will build the linked list and pass the node to your function.
- The output will be the entire list after calling your function.

Example 1:

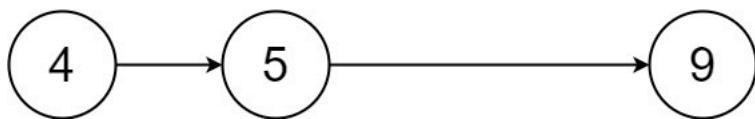
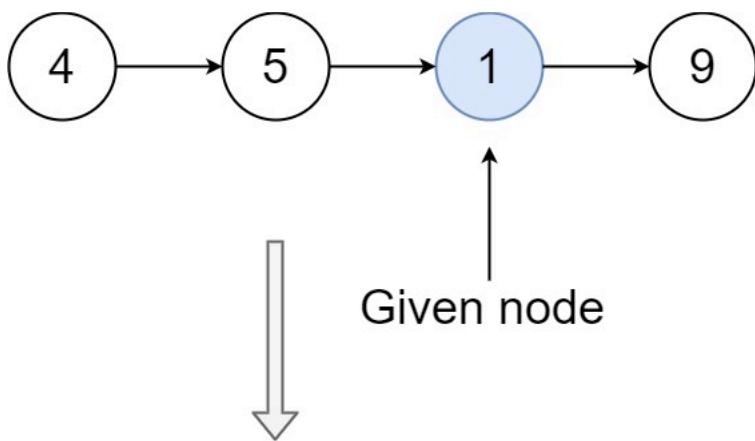


Input: head = [4,5,1,9], node = 5

Output: [4,1,9]

Explanation: You are given the second node with value 5, the linked list should become 4 → 1

Example 2:



Input: head = [4,5,1,9], node = 1

Output: [4,5,9]

Explanation: You are given the third node with value 1, the linked list should become 4 → 5 →

Constraints:

- The number of the nodes in the given list is in the range [2, 1000].
- $-1000 \leq \text{Node.val} \leq 1000$
- The value of each node in the list is **unique**.
- The node to be deleted is **in the list** and is **not a tail node**.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    void deleteNode(ListNode* node) {
        ListNode* nextElement = node->next;
        node->val = nextElement->val;
        node->next = nextElement->next;
        delete nextElement;
    }
};
```

12 Integer to Roman ([link](#))

Description

Seven different symbols represent Roman numerals with the following values:

Symbol Value

I	1
V	5
X	10
L	50
C	100
D	500
M	1000

Roman numerals are formed by appending the conversions of decimal place values from highest to lowest. Converting a decimal place value into a Roman numeral has the following rules:

- If the value does not start with 4 or 9, select the symbol of the maximal value that can be subtracted from the input, append that symbol to the result, subtract its value, and convert the remainder to a Roman numeral.
- If the value starts with 4 or 9 use the **subtractive form** representing one symbol subtracted from the following symbol, for example, 4 is 1 (I) less than 5 (V): IV and 9 is 1 (I) less than 10 (X): IX. Only the following subtractive forms are used: 4 (IV), 9 (IX), 40 (XL), 90 (XC), 400 (CD) and 900 (CM).
- Only powers of 10 (I, X, C, M) can be appended consecutively at most 3 times to represent multiples of 10. You cannot append 5 (V), 50 (L), or 500 (D) multiple times. If you need to append a symbol 4 times use the **subtractive form**.

Given an integer, convert it to a Roman numeral.

Example 1:

Input: num = 3749

Output: "MMMDCCXLIX"

Explanation:

3000 = MMM as 1000 (M) + 1000 (M) + 1000 (M)

700 = DCC as 500 (D) + 100 (C) + 100 (C)

40 = XL as 10 (X) less of 50 (L)

9 = IX as 1 (I) less of 10 (X)

Note: 49 is not 1 (I) less of 50 (L) because the conversion is based on decimal places

Example 2:

Input: num = 58

Output: "LVIII"

Explanation:

50 = L

8 = VIII

Example 3:

Input: num = 1994

Output: "MCMXCV"

Explanation:

```
1000 = M  
900 = CM  
90 = XC  
4 = IV
```

Constraints:

- $1 \leq \text{num} \leq 3999$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    string intToRoman(int num) {
        vector<pair<int, string>> mapArr = {
            {1, "I"}, {4, "IV"}, {5, "V"}, {9, "IX"}, {10, "X"}, {40, "XL"}, {50, "L"}, {90, "XC"}, {100, "C"}, {400, "CD"}, {500, "D"}, {900, "CM"}, {1000, "M"}, };
        int justSmallerNumberIndex = mapArr.size()-1;
        string ans = "";
        while(num > 0 && justSmallerNumberIndex >= 0){
            if(num >= mapArr[justSmallerNumberIndex].first){
                ans += mapArr[justSmallerNumberIndex].second;
                num -= mapArr[justSmallerNumberIndex].first;
            }else{
                // Move to lesser roman number
                justSmallerNumberIndex--;
            }
        }
        return ans;
    }
};
```

13 Roman to Integer ([link](#))

Description

Roman numerals are represented by seven different symbols: I, V, X, L, C, D and M.

Symbol	Value
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

For example, 2 is written as II in Roman numeral, just two ones added together. 12 is written as XII, which is simply X + II. The number 27 is written as XXVII, which is XX + V + II.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not IIII. Instead, the number four is written as IV. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as IX. There are six instances where subtraction is used:

- I can be placed before V (5) and X (10) to make 4 and 9.
- X can be placed before L (50) and C (100) to make 40 and 90.
- C can be placed before D (500) and M (1000) to make 400 and 900.

Given a roman numeral, convert it to an integer.

Example 1:

```
Input: s = "III"
Output: 3
Explanation: III = 3.
```

Example 2:

```
Input: s = "LVIII"
Output: 58
Explanation: L = 50, V = 5, III = 3.
```

Example 3:

```
Input: s = "MCMXCIV"
Output: 1994
Explanation: M = 1000, CM = 900, XC = 90 and IV = 4.
```

Constraints:

- $1 \leq s.length \leq 15$
- s contains only the characters ('I', 'V', 'X', 'L', 'C', 'D', 'M').
- It is guaranteed that s is a valid roman numeral in the range [1, 3999].

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int romanToInt(string s) {
        int romanToInt = 0;
        unordered_map<char, int> map = {{'I', 1},
                                         {'V', 5},
                                         {'X', 10},
                                         {'L', 50},
                                         {'C', 100},
                                         {'D', 500},
                                         {'M', 1000}};
        for (int i = 0; i < s.size(); i++)
        {
            if (i + 1 < s.size() && map[s[i]] < map[s[i + 1]])
            {
                // Subtract
                romanToInt -= map[s[i]];
            }
            else
            {
                // Add
                romanToInt += map[s[i]];
            }
        }
        return romanToInt;
    }
};
```

[3997 Maximize Sum of At Most K Distinct Elements \(link\)](#)

Description

You are given a **positive** integer array `nums` and an integer `k`.

Choose at most `k` elements from `nums` so that their sum is maximized. However, the chosen numbers must be **distinct**.

Return an array containing the chosen numbers in **strictly descending** order.

Example 1:

Input: `nums = [84,93,100,77,90]`, `k = 3`

Output: `[100,93,90]`

Explanation:

The maximum sum is 283, which is attained by choosing 93, 100 and 90. We rearrange them in strictly descending order as `[100, 93, 90]`.

Example 2:

Input: `nums = [84,93,100,77,93]`, `k = 3`

Output: `[100,93,84]`

Explanation:

The maximum sum is 277, which is attained by choosing 84, 93 and 100. We rearrange them in strictly descending order as `[100, 93, 84]`. We cannot choose 93, 100 and 93 because the chosen numbers must be distinct.

Example 3:

Input: `nums = [1,1,1,2,2,2]`, `k = 6`

Output: `[2,1]`

Explanation:

The maximum sum is 3, which is attained by choosing 1 and 2. We rearrange them in strictly descending order as `[2, 1]`.

Constraints:

- `1 <= nums.length <= 100`
- `1 <= nums[i] <= 109`
- `1 <= k <= nums.length`

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
#include<bits/stdc++.h>
class Solution {
    static bool customOperator(int &a, int &b){
        return a>b;
    }
public:
    vector<int> maxKDistinct(vector<int>& nums, int k) {
        sort(nums.begin(), nums.end(), customOperator);
        int currK = 0;
        if(k==1) return {nums[0]};
        vector<int> ans;
        ans.push_back(nums[0]);
        for(int i=1; i<nums.size(); i++){
            if(nums[i-1]==nums[i]) continue;
            ans.push_back(nums[i]);
            if(ans.size() == k) break;
        }
        return ans;
    }
};
```

4012 Earliest Time to Finish One Task ([link](#))

Description

You are given a 2D integer array tasks where $\text{tasks}[i] = [s_i, t_i]$.

Each $[s_i, t_i]$ in tasks represents a task with start time s_i that takes t_i units of time to finish.

Return the earliest time at which at least one task is finished.

Example 1:

Input: tasks = [[1,6],[2,3]]

Output: 5

Explanation:

The first task starts at time $t = 1$ and finishes at time $1 + 6 = 7$. The second task finishes at time $2 + 3 = 5$. You can finish one task at time 5.

Example 2:

Input: tasks = [[100,100],[100,100],[100,100]]

Output: 200

Explanation:

All three tasks finish at time $100 + 100 = 200$.

Constraints:

- $1 \leq \text{tasks.length} \leq 100$
- $\text{tasks}[i] = [s_i, t_i]$
- $1 \leq s_i, t_i \leq 100$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int earliestTime(vector<vector<int>>& tasks) {
        int minTime = INT_MAX;
        if(tasks.size() == 1) return tasks[0][0] + tasks[0][1];

        for(auto &row:tasks){
            minTime = min(minTime, row[0] + row[1]);
        }
        return minTime;
    }
};
```

[4011 Smallest Absent Positive Greater Than Average](#) ([link](#))

Description

You are given an integer array `nums`.

Return the **smallest absent positive** integer in `nums` such that it is **strictly greater** than the **average** of all elements in `nums`.

The **average** of an array is defined as the sum of all its elements divided by the number of elements.

Example 1:

Input: `nums` = [3,5]

Output: 6

Explanation:

- The average of `nums` is $(3 + 5) / 2 = 8 / 2 = 4$.
- The smallest absent positive integer greater than 4 is 6.

Example 2:

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

Output: 3

Explanation:

- The average of `nums` is $(-1 + 1 + 2) / 3 = 2 / 3 = 0.667$.
- The smallest absent positive integer greater than 0.667 is 3.

Example 3:

Input: `nums` = [4,-1]

Output: 2

Explanation:

- The average of `nums` is $(4 + (-1)) / 2 = 3 / 2 = 1.50$.
- The smallest absent positive integer greater than 1.50 is 2.

Constraints:

- $1 \leq \text{nums.length} \leq 100$
- $-100 \leq \text{nums}[i] \leq 100$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int smallestAbsent(vector<int>& nums) {
        int sumArr = accumulate(nums.begin(),nums.end(),0);
        int avg = floor((double)sumArr / (double)nums.size() );
        if(avg <=0)avg = 0;
        for(int i = avg+1;i<=INT_MAX;i++){
            auto it = find(nums.begin(),nums.end(),i);
            if(it == nums.end()){
                return i;
            }
        }
        return 0;
    }
};
```

[1737 Maximum Nesting Depth of the Parentheses \(link\)](#)

Description

Given a **valid parentheses string** s , return the **nesting depth** of s . The nesting depth is the **maximum** number of nested parentheses.

Example 1:

Input: $s = "(1+(2*3)+((8)/4))+1"$

Output: 3

Explanation:

Digit 8 is inside of 3 nested parentheses in the string.

Example 2:

Input: $s = "(1)+((2))+(((3)))"$

Output: 3

Explanation:

Digit 3 is inside of 3 nested parentheses in the string.

Example 3:

Input: $s = "()((())(((0)))"$

Output: 3

Constraints:

- $1 \leq s.length \leq 100$
- s consists of digits 0–9 and characters '+', '−', '*', '/', '(', and ')'.
• It is guaranteed that parentheses expression s is a VPS.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int maxDepth(string s) {
        int maxCount=0,currentCount=0;
        for(auto character:s){
            if(character == '('){
                currentCount++;
            }else if(character == ')'){
                currentCount--;
            }
            maxCount = max(currentCount,maxCount);
        }
        return maxCount;
    }
};
```

[451 Sort Characters By Frequency \(link\)](#)

Description

Given a string s , sort it in **decreasing order** based on the **frequency** of the characters. The **frequency** of a character is the number of times it appears in the string.

Return *the sorted string*. If there are multiple answers, return *any of them*.

Example 1:

Input: $s = \text{"tree"}$

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: $s = \text{"cccaaa"}$

Output: "aaaccc"

Explanation: Both 'c' and 'a' appear three times, so both "cccaaa" and "aaaccc" are valid answers. Note that "cacaca" is incorrect, as the same characters must be together.

Example 3:

Input: $s = \text{"Aabb"}$

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:

- $1 \leq s.\text{length} \leq 5 * 10^5$
- s consists of uppercase and lowercase English letters and digits.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
    static bool customOperator(pair<char, int>& s1, pair<char, int>& s2) {
        if (s1.second > s2.second)
            return true;
        if (s1.second < s2.second)
            return false;
        return s1.first < s2.first; // Return smaller Occuring Character
    }

public:
    string frequencySort(string s) {
        if(s.size() <=2) return s;
        vector<pair<char, int>> frequency(256, {NULL, 0});
        for (auto& ch : s) {
            frequency[(int)ch].first = ch;
            frequency[(int)ch].second += 1;
        }
        sort(frequency.begin(), frequency.end(), customOperator);
        string ans = "";
        for (auto& element : frequency) {
            if (element.second != 0) {
                for (int i = 0; i < element.second; i++)
                    ans += element.first;
            }
        }
        return ans;
    }
};
```

[242 Valid Anagram \(link\)](#)

Description

Given two strings s and t, return true if t is an anagram of s, and false otherwise.

Example 1:

Input: s = "anagram", t = "nagaram"

Output: true

Example 2:

Input: s = "rat", t = "car"

Output: false

Constraints:

- $1 \leq s.length, t.length \leq 5 * 10^4$
- s and t consist of lowercase English letters.

Follow up: What if the inputs contain Unicode characters? How would you adapt your solution to such a case?

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    bool isAnagram(string s, string t) {
        int freq[256]={0};

        for(int i=0 ;i <s.size();i++){
            freq[s[i]]++;
        }
        for(int i=0 ;i <t.size();i++){
            freq[t[i]]--;
        }
        for(auto &element:freq){
            if(element != 0 ){
                return false;
            }
        }
        return true;
    }
};
```

[812 Rotate String \(link\)](#)

Description

Given two strings s and $goal$, return `true` if and only if s can become $goal$ after some number of **shifts** on s .

A **shift** on s consists of moving the leftmost character of s to the rightmost position.

- For example, if $s = "abcde"$, then it will be $"bcdea"$ after one shift.

Example 1:

```
Input: s = "abcde", goal = "cdeab"  
Output: true
```

Example 2:

```
Input: s = "abcde", goal = "abced"  
Output: false
```

Constraints:

- $1 \leq s.length, goal.length \leq 100$
- s and $goal$ consist of lowercase English letters.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
    bool isRotateSame(string &temp, string &goal,int index){
        reverse(temp.begin(),temp.begin()+index);
        reverse(temp.begin()+index,temp.end());
        reverse(temp.begin(),temp.end());
        if(temp == goal){
            return true;
        }
        return false;
    }
public:
    bool rotateString(string s, string goal) {
        if(s.size() != goal.size())return false;
        for(int i=0; i<goal.size(); i++){
            string temp = s;
            if(isRotateSame(temp,goal,i))return true;
        }
        return false;
    }
};
```

[205 Isomorphic Strings \(link\)](#)

Description

Given two strings s and t, *determine if they are isomorphic*.

Two strings s and t are isomorphic if the characters in s can be replaced to get t.

All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character, but a character may map to itself.

Example 1:

Input: s = "egg", t = "add"

Output: true

Explanation:

The strings s and t can be made identical by:

- Mapping 'e' to 'a'.
- Mapping 'g' to 'd'.

Example 2:

Input: s = "foo", t = "bar"

Output: false

Explanation:

The strings s and t can not be made identical as 'o' needs to be mapped to both 'a' and 'r'.

Example 3:

Input: s = "paper", t = "title"

Output: true

Constraints:

- $1 \leq s.length \leq 5 * 10^4$
- $t.length == s.length$
- s and t consist of any valid ascii character.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    bool isIsomorphic(string s, string t) {
        vector<int> arr1(256, -1), arr2(256, -1);

        for (int i = 0; i < s.size(); i++) {
            if (arr1[s[i]] != arr2[t[i]])
                return false;
            arr1[s[i]] = i ;
            arr2[t[i]] = i ;
        }
        return true;
    }
};
```

14 Longest Common Prefix ([link](#))

Description

Write a function to find the longest common prefix string amongst an array of strings.

If there is no common prefix, return an empty string "".

Example 1:

```
Input: strs = ["flower","flow","flight"]
Output: "fl"
```

Example 2:

```
Input: strs = ["dog","racecar","car"]
Output: ""
Explanation: There is no common prefix among the input strings.
```

Constraints:

- $1 \leq \text{strs.length} \leq 200$
- $0 \leq \text{strs[i].length} \leq 200$
- strs[i] consists of only lowercase English letters if it is non-empty.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    string longestCommonPrefix(vector<string>& strs) {
        if(strs.size() == 1) return strs[0];
        sort(strs.begin(), strs.end());

        string first = strs[0];
        string last = strs[strs.size() - 1];

        string ans = "";
        for (int i = 0; i < min(first.size(), last.size()); i++) {
            if (first[i] != last[i]) {
                break;
            }
            ans += first[i];
        }
        return ans;
    }
};
```

[2032 Largest Odd Number in String \(link\)](#)

Description

You are given a string `num`, representing a large integer. Return *the largest-valued odd integer (as a string) that is a non-empty substring of num, or an empty string "" if no odd integer exists.*

A **substring** is a contiguous sequence of characters within a string.

Example 1:

Input: num = "52"

Output: "5"

Explanation: The only non-empty substrings are "5", "2", and "52". "5" is the only odd number.

Example 2:

Input: num = "4206"

Output: ""

Explanation: There are no odd numbers in "4206".

Example 3:

Input: num = "35427"

Output: "35427"

Explanation: "35427" is already an odd number.

Constraints:

- $1 \leq \text{num.length} \leq 10^5$
- num only consists of digits and does not contain any leading zeros.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    string largestOddNumber(string num) {
        int indexOdd = -1;
        for(int i=num.size()-1; i>=0; i--){
            char ch = num[i];
            int currentDigit = stoi(to_string(ch));
            if(currentDigit % 2==1){
                return num.substr(0,i+1);
            }
        }
        return "";
};
```

151 Reverse Words in a String ([link](#))

Description

Given an input string s , reverse the order of the **words**.

A **word** is defined as a sequence of non-space characters. The **words** in s will be separated by at least one space.

Return a *string of the words in reverse order concatenated by a single space*.

Note that s may contain leading or trailing spaces or multiple spaces between two words. The returned string should only have a single space separating the words. Do not include any extra spaces.

Example 1:

```
Input: s = "the sky is blue"  
Output: "blue is sky the"
```

Example 2:

```
Input: s = " hello world "  
Output: "world hello"  
Explanation: Your reversed string should not contain leading or trailing spaces.
```

Example 3:

```
Input: s = "a good example"  
Output: "example good a"  
Explanation: You need to reduce multiple spaces between two words to a single space in the rever
```

Constraints:

- $1 \leq s.length \leq 10^4$
- s contains English letters (upper-case and lower-case), digits, and spaces ' '.
- There is **at least one** word in s .

Follow-up: If the string data type is mutable in your language, can you solve it **in-place** with $O(1)$ extra space?

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    string reverseWords(string s) {
        int size = s.size();
        string ans = "";
        int startSpace=0,endSpace=0;
        for(int i=0;i<size;i++){
            if (s[i] != ' '){
                startSpace = i;
                break;
            }
        }
        s.erase(0,startSpace);
        for(int i=s.size()-1;i>=0;i--){
            if (s[i] != ' '){
                endSpace = i;
                break;
            }
        }
        s.erase(endSpace+1,s.size()-endSpace-1);
        size = s.size();
        int left = size - 1, right = size - 1;
        while (left >= 0) {
            char ch = s[left];
            if (left != size-1 && ch == ' ' && s[left + 1] == ' ') {
                left--;
                continue;
            }
            if (ch == ' ' && left != size - 1) {
                ans += s.substr(left + 1, right - (left + 1) + 1) + " ";
            } else if (ch != ' ' && left != size - 1 && s[left + 1] == ' ') {
                right = left;
            }
            left--;
        }
        ans += s.substr(0, right + 1);
        return ans;
    }
};
```

[1078 Remove Outermost Parentheses \(link\)](#)

Description

A valid parentheses string is either empty "", "(" + A + ")", or A + B, where A and B are valid parentheses strings, and + represents string concatenation.

- For example, "", "()", "((()))", and "(((())))" are all valid parentheses strings.

A valid parentheses string s is primitive if it is nonempty, and there does not exist a way to split it into $s = A + B$, with A and B nonempty valid parentheses strings.

Given a valid parentheses string s, consider its primitive decomposition: $s = P_1 + P_2 + \dots + P_k$, where P_i are primitive valid parentheses strings.

Return s *after removing the outermost parentheses of every primitive string in the primitive decomposition of s*.

Example 1:

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

Output: "())()

Explanation:

The input string is "((())())()", with primitive decomposition "((())") + "(()())". After removing outer parentheses of each part, this is "())" + ")" = "())()".

Example 2:

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

Output: "())()()()

Explanation:

The input string is "((())())((())())", with primitive decomposition "((())") + "(()())" + "(()())". After removing outer parentheses of each part, this is "())" + ")" + "())()" = "())()()()

Example 3:

Input: s = "())()

Output: ""

Explanation:

The input string is "())()", with primitive decomposition ")" + ")". After removing outer parentheses of each part, this is "" + "" = "".

Constraints:

- $1 \leq s.length \leq 10^5$
- $s[i]$ is either '(' or ')'.
• s is a valid parentheses string.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    string removeOuterParentheses(string s) {
        string ans = "";
        int parenthesisCount = 0;

        for (char& ch : s) {
            if (ch == '(') {
                if (parenthesisCount != 0)
                    ans += "(";
                parenthesisCount++;
            } else {
                if (parenthesisCount > 1)
                    ans += ")";
                parenthesisCount--;
            }
        }
        return ans;
    }
};
```

36 Valid Sudoku ([link](#))

Description

Determine if a 9×9 Sudoku board is valid. Only the filled cells need to be validated **according to the following rules**:

1. Each row must contain the digits 1–9 without repetition.
2. Each column must contain the digits 1–9 without repetition.
3. Each of the nine 3×3 sub-boxes of the grid must contain the digits 1–9 without repetition.

Note:

- A Sudoku board (partially filled) could be valid but is not necessarily solvable.
- Only the filled cells need to be validated according to the mentioned rules.

Example 1:

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

```
Input: board =  
[[ "5", "3", ".", ".", "7", ".", ".", ".", "."]]  
,[ "6", ".", ".", "1", "9", "5", ".", ".", "."]]  
,[ ".", "9", "8", ".", ".", ".", ".", "6", "."]]  
,[ "8", ".", ".", "6", ".", ".", ".", ".", "3"]]  
,[ "4", ".", ".", "8", ".", "3", ".", ".", "1"]]  
,[ "7", ".", ".", "2", ".", ".", ".", ".", "6"]]  
,[ ".", "6", ".", ".", "2", "8", ".", "."]]  
,[ ".", ".", "4", "1", "9", ".", ".", "5"]]  
,[ ".", ".", "8", ".", ".", "7", "9"]]  
Output: true
```

Example 2:

```
Input: board =  
[[ "8", "3", ".", ".", "7", ".", ".", ".", "."]]  
,[ "6", ".", ".", "1", "9", "5", ".", ".", "."]]  
,[ ".", "9", "8", ".", ".", ".", ".", "6", "."]]  
,[ "8", ".", ".", "6", ".", ".", ".", ".", "3"]]  
,[ "4", ".", ".", "8", ".", "3", ".", ".", "1"]]  
,[ "7", ".", ".", "2", ".", ".", ".", ".", "6"]]  
,[ ".", "6", ".", ".", "2", "8", ".", "."]]  
,[ ".", ".", "4", "1", "9", ".", ".", "5"]]  
,[ ".", ".", "8", ".", ".", "7", "9"]]  
Output: false
```

Explanation: Same as Example 1, except with the 5 in the top left corner being modified to 8. ↴

Constraints:

- `board.length == 9`
- `board[i].length == 9`
- `board[i][j]` is a digit 1–9 or '.'.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

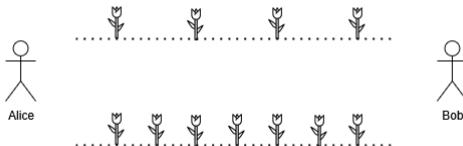
```
class Solution {
    void isPossible(vector<vector<char>> &board, int row, int col,
                  bool& isValid) {
        for (int i = 0; i < 9; i++) {
            if (i==row || col==i)
                continue;
            if (board[row][col] == board[row][i] ||
                board[row][col] == board[i][col]){
                isValid = false;return;}
        }
        int boxRow,boxCol;
        if(row >=0 && row < 3)boxRow=0;
        if(row >=3 && row < 6)boxRow=3;
        if(row >=6 && row < 9)boxRow=6;
        if(col >=0 && col < 3)boxCol=0;
        if(col >=3 && col < 6)boxCol=3;
        if(col >=6 && col < 9)boxCol=6;
        int cnt=0;
        for(int i=boxRow;i<boxRow+3;i++){
            for(int j=boxCol;j<boxCol+3;j++){
                if(board[row][col] == board[i][j]){
                    cnt++;
                }
                if(cnt>=2){isValid=false;return;}
            }
        }
        isValid = true;
    }

public:
    bool isValidSudoku(vector<vector<char>>& board) {
        bool isValid = true;
        for (int i = 0; i < board.size(); i++) {
            for (int j = 0; j < board[0].size(); j++) {
                if (board[i][j] - '0' >= 0 && board[i][j] - '0' <= 9) {
                    isPossible(board, i, j, isValid);
                    if (!isValid)
                        return false;
                }
            }
        }
        return isValid;
    }
};
```

[3279 Alice and Bob Playing Flower Game \(link\)](#)

Description

Alice and Bob are playing a turn-based game on a field, with two lanes of flowers between them. There are x flowers in the first lane between Alice and Bob, and y flowers in the second lane between them.



The game proceeds as follows:

1. Alice takes the first turn.
2. In each turn, a player must choose either one of the lane and pick one flower from that side.
3. At the end of the turn, if there are no flowers left at all in either lane, the **current** player captures their opponent and wins the game.

Given two integers, n and m , the task is to compute the number of possible pairs (x, y) that satisfy the conditions:

- Alice must win the game according to the described rules.
- The number of flowers x in the first lane must be in the range $[1, n]$.
- The number of flowers y in the second lane must be in the range $[1, m]$.

Return *the number of possible pairs (x, y) that satisfy the conditions mentioned in the statement.*

Example 1:

Input: $n = 3, m = 2$

Output: 3

Explanation: The following pairs satisfy conditions described in the statement: $(1,2)$, $(3,2)$,

Example 2:

Input: $n = 1, m = 1$

Output: 0

Explanation: No pairs satisfy the conditions described in the statement.

Constraints:

- $1 \leq n, m \leq 10^5$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    long long flowerGame(int n, int m) {
        int nOdd = (n+1)/2, nEven = n/2, mOdd=(m+1)/2, mEven=m/2;
        return 1ll*nOdd*mEven + 1ll*mOdd*nEven;
    }
};
```

[1605 Minimum Number of Days to Make m Bouquets \(link\)](#)

Description

You are given an integer array `bloomDay`, an integer `m` and an integer `k`.

You want to make `m` bouquets. To make a bouquet, you need to use `k` **adjacent flowers** from the garden.

The garden consists of `n` flowers, the `ith` flower will bloom in the `bloomDay[i]` and then can be used in **exactly one** bouquet.

Return *the minimum number of days you need to wait to be able to make `m` bouquets from the garden*. If it is impossible to make `m` bouquets return `-1`.

Example 1:

```
Input: bloomDay = [1,10,3,10,2], m = 3, k = 1
```

```
Output: 3
```

Explanation: Let us see what happened in the first three days. `x` means flower bloomed and `_` means we need 3 bouquets each should contain 1 flower.

After day 1: `[x, _, _, _, _]` // we can only make one bouquet.

After day 2: `[x, _, _, _, x]` // we can only make two bouquets.

After day 3: `[x, _, x, _, x]` // we can make 3 bouquets. The answer is 3.

Example 2:

```
Input: bloomDay = [1,10,3,10,2], m = 3, k = 2
```

```
Output: -1
```

Explanation: We need 3 bouquets each has 2 flowers, that means we need 6 flowers. We only have

Example 3:

```
Input: bloomDay = [7,7,7,7,12,7,7], m = 2, k = 3
```

```
Output: 12
```

Explanation: We need 2 bouquets each should have 3 flowers.

Here is the garden after the 7 and 12 days:

After day 7: `[x, x, x, x, _, x, x]`

We can make one bouquet of the first three flowers that bloomed. We cannot make another bouquet.

After day 12: `[x, x, x, x, x, x, x]`

It is obvious that we can make two bouquets in different ways.

Constraints:

- `bloomDay.length == n`
- $1 \leq n \leq 10^5$
- $1 \leq \text{bloomDay}[i] \leq 10^9$
- $1 \leq m \leq 10^6$
- $1 \leq k \leq n$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
    void findMaxMin(vector<int>&bloomDay, int &maxi, int &mini){
        for(auto &element:bloomDay){
            maxi = max(maxi,element);
            mini = min(mini,element);
        }
    }
public:
    int minDays(vector<int>& bloomDay, int m, int k) {
        int maxi = INT_MIN,mini = INT_MAX,minDays=-1;
        findMaxMin(bloomDay,maxi,mini);
        int size = bloomDay.size();
        if(size < (long long)m*k){
            return -1;
        }

        int start = mini,end = maxi;
        while(start <= end){
            int mid = start + (end-start)/2;
            int count=0,bouquetsFormed= 0;
            for(int i=0; i<bloomDay.size();i++){
                if(bloomDay[i] <= mid)count++;
                else{
                    bouquetsFormed += count / k ;
                    count=0;
                }
            }
            bouquetsFormed += count / k ;
            if(bouquetsFormed >= m){
                minDays = mid;
                end = mid-1;
            }else{
                start = mid+1;
            }
        }
        return minDays;
    }
};
```

[907 Koko Eating Bananas \(link\)](#)

Description

Koko loves to eat bananas. There are n piles of bananas, the i^{th} pile has $\text{piles}[i]$ bananas. The guards have gone and will come back in h hours.

Koko can decide her bananas-per-hour eating speed of k . Each hour, she chooses some pile of bananas and eats k bananas from that pile. If the pile has less than k bananas, she eats all of them instead and will not eat any more bananas during this hour.

Koko likes to eat slowly but still wants to finish eating all the bananas before the guards return.

Return *the minimum integer k such that she can eat all the bananas within h hours.*

Example 1:

```
Input: piles = [3,6,7,11], h = 8
Output: 4
```

Example 2:

```
Input: piles = [30,11,23,4,20], h = 5
Output: 30
```

Example 3:

```
Input: piles = [30,11,23,4,20], h = 6
Output: 23
```

Constraints:

- $1 \leq \text{piles.length} \leq 10^4$
- $\text{piles.length} \leq h \leq 10^9$
- $1 \leq \text{piles}[i] \leq 10^9$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int minEatingSpeed(vector<int>& piles, int h) {
        int size=piles.size(),maxElement = INT_MIN,ans=-1;

        for(int i=0; i<size;i++)maxElement = max(maxElement,piles[i]);
        int start = 1,end = maxElement;
        while(start <= end){
            int mid = start + (end - start)/2;
            long long currentSum=0;
            for(auto &element:piles){
                currentSum += ceil((double(element)/double(mid)));
            }
            if(currentSum <= h){
                ans = mid;
                end = mid-1;
            }else{
                start = mid+1;
            }
        }
        return ans;
    }
};
```

[162 Find Peak Element \(link\)](#)

Description

A peak element is an element that is strictly greater than its neighbors.

Given a **0-indexed** integer array `nums`, find a peak element, and return its index. If the array contains multiple peaks, return the index to **any of the peaks**.

You may imagine that `nums[-1] = nums[n] = -∞`. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array.

You must write an algorithm that runs in $O(\log n)$ time.

Example 1:

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

Output: 2

Explanation: 3 is a peak element and your function should return the index number 2.

Example 2:

Input: `nums = [1,2,1,3,5,6,4]`

Output: 5

Explanation: Your function can return either index number 1 where the peak element is 2, or index number 5 where the peak element is 6.

Constraints:

- $1 \leq \text{nums.length} \leq 1000$
- $-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$
- $\text{nums}[i] \neq \text{nums}[i + 1]$ for all valid i .

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int findPeakElement(vector<int>& nums) {
        int start = 1, size = nums.size(), end = size-2;
        if(size == 1 || nums[0]>nums[1])return 0;
        if(nums[size-1]>nums[size-2])return size-1;

        while(start <= end){
            int mid = start + (end - start)/2;
            if(nums[mid-1]<nums[mid] && nums[mid] > nums[mid+1]){
                return mid;
            }
            if(nums[mid-1]<nums[mid] && nums[mid] < nums[mid+1]){
                start=mid+1;
            }else{
                end = mid-1;
            }
        }
        return -1;
    }
};
```

540 Single Element in a Sorted Array [\(link\)](#)

Description

You are given a sorted array consisting of only integers where every element appears exactly twice, except for one element which appears exactly once.

Return *the single element that appears only once*.

Your solution must run in $O(\log n)$ time and $O(1)$ space.

Example 1:

```
Input: nums = [1,1,2,3,3,4,4,8,8]
Output: 2
```

Example 2:

```
Input: nums = [3,3,7,7,10,11,11]
Output: 10
```

Constraints:

- $1 \leq \text{nums.length} \leq 10^5$
- $0 \leq \text{nums}[i] \leq 10^5$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int singleNonDuplicate(vector<int>& nums) {
        if(nums.size()==1) return nums[0];
        if(nums[0]!= nums[1]) return nums[0];
        if(nums[nums.size()-1]!=nums[nums.size()-2]) return nums[nums.size()-1];

        int start = 1,end = nums.size()-2;

        while(start <=end){
            int mid = start + (end-start)/2;
            if(nums[mid]!=nums[mid-1] && nums[mid]!=nums[mid+1]) return nums[mid];
            if(((mid%2==0) && nums[mid]==nums[mid+1]) || ((mid%2==1) && nums[mid]==nums[mid-1]))
                else end = mid-1;
        }
        return -1;
    }
};
```

81 Search in Rotated Sorted Array II ([link](#))

Description

There is an integer array `nums` sorted in non-decreasing order (not necessarily with **distinct** values).

Before being passed to your function, `nums` is **rotated** at an unknown pivot index `k` ($0 \leq k < \text{nums.length}$) such that the resulting array is `[nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]` (**0-indexed**). For example, `[0,1,2,4,4,4,5,6,6,7]` might be rotated at pivot index 5 and become `[4,5,6,6,7,0,1,2,4,4]`.

Given the array `nums` **after** the rotation and an integer target, return true *if target is in nums, or false if it is not in nums.*

You must decrease the overall operation steps as much as possible.

Example 1:

```
Input: nums = [2,5,6,0,0,1,2], target = 0
Output: true
```

Example 2:

```
Input: nums = [2,5,6,0,0,1,2], target = 3
Output: false
```

Constraints:

- $1 \leq \text{nums.length} \leq 5000$
- $-10^4 \leq \text{nums}[i] \leq 10^4$
- `nums` is guaranteed to be rotated at some pivot.
- $-10^4 \leq \text{target} \leq 10^4$

Follow up: This problem is similar to [Search in Rotated Sorted Array](#), but `nums` may contain **duplicates**. Would this affect the runtime complexity? How and why?

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    bool search(vector<int>& nums, int target) {
        int start = 0, end = nums.size()-1;
        bool ans = false;

        while(start <= end){
            int mid = start + (end - start)/2;
            if(nums[mid] == target){
                return true;
            }
            if(nums[start] == nums[mid] && nums[mid] == nums[end]){
                start++;end--;continue;
            }
            if(nums[start] <= nums[mid]){
                if(nums[start] <= target && target <= nums[mid]){
                    end = mid - 1;
                }else{
                    start = mid+1;
                }
            }else{
                if(nums[mid]<=target && target <= nums[end]){
                    start = mid+1;
                }else{
                    end = mid-1;
                }
            }
        }
        return false;
    }
};
```

34 Find First and Last Position of Element in Sorted Array ([link](#))

Description

Given an array of integers `nums` sorted in non-decreasing order, find the starting and ending position of a given target value.

If target is not found in the array, return `[-1, -1]`.

You must write an algorithm with $O(\log n)$ runtime complexity.

Example 1:

```
Input: nums = [5,7,7,8,8,10], target = 8
Output: [3,4]
```

Example 2:

```
Input: nums = [5,7,7,8,8,10], target = 6
Output: [-1,-1]
```

Example 3:

```
Input: nums = [], target = 0
Output: [-1,-1]
```

Constraints:

- $0 \leq \text{nums.length} \leq 10^5$
- $-10^9 \leq \text{nums}[i] \leq 10^9$
- `nums` is a non-decreasing array.
- $-10^9 \leq \text{target} \leq 10^9$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
    void find_first(vector<int>&arr, int start, int end, int &firstOcc, int target){
        while(start <= end){
            int mid = start + (end - start)/2;
            if(arr[mid]==target){
                firstOcc = mid;
                end = mid - 1;
            }else if(arr[mid] > target){
                end = mid-1;
            }else{
                start = mid+1;
            }
        }
    }

    void find_last(vector<int>&arr, int start, int end, int &lastOcc, int target){
        while(start <= end){
            int mid = start + (end - start)/2;
            if(arr[mid]==target){
                lastOcc = mid;
                start = mid + 1;
            }else if(arr[mid] > target){
                end = mid-1;
            }else{
                start = mid+1;
            }
        }
    }

public:
    vector<int> searchRange(vector<int>& nums, int target) {
        int size = nums.size();
        int start = 0, end = size - 1, firstOcc = -1, lastOcc = -1;
        if(size == 0){
            return {-1,-1};
        }
        find_first(nums,start,end,firstOcc,target);
        find_last(nums,start,end,lastOcc,target);

        return {firstOcc,lastOcc};
    }
};
```

35 Search Insert Position ([link](#))

Description

Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You must write an algorithm with $O(\log n)$ runtime complexity.

Example 1:

```
Input: nums = [1,3,5,6], target = 5
Output: 2
```

Example 2:

```
Input: nums = [1,3,5,6], target = 2
Output: 1
```

Example 3:

```
Input: nums = [1,3,5,6], target = 7
Output: 4
```

Constraints:

- $1 \leq \text{nums.length} \leq 10^4$
- $-10^4 \leq \text{nums}[i] \leq 10^4$
- **nums** contains **distinct** values sorted in **ascending** order.
- $-10^4 \leq \text{target} \leq 10^4$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int searchInsert(vector<int>& nums, int target) {
        int start = 0, end = nums.size() - 1;
        int ansIndex = nums.size();

        while(start <= end){
            int mid = start + (end - start)/2;
            if(nums[mid] >= target){
                ansIndex = mid;
                end = mid - 1;
            }else{
                start = mid + 1;
            }
        }
        return ansIndex;
    }
};
```

[792 Binary Search \(link\)](#)

Description

Given an array of integers `nums` which is sorted in ascending order, and an integer `target`, write a function to search `target` in `nums`. If `target` exists, then return its index. Otherwise, return `-1`.

You must write an algorithm with $O(\log n)$ runtime complexity.

Example 1:

```
Input: nums = [-1,0,3,5,9,12], target = 9
Output: 4
Explanation: 9 exists in nums and its index is 4
```

Example 2:

```
Input: nums = [-1,0,3,5,9,12], target = 2
Output: -1
Explanation: 2 does not exist in nums so return -1
```

Constraints:

- $1 \leq \text{nums.length} \leq 10^4$
- $-10^4 < \text{nums}[i], \text{target} < 10^4$
- All the integers in `nums` are **unique**.
- `nums` is sorted in ascending order.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int start = 0, end = nums.size() - 1;

        while(start <= end){
            int mid = start + (end - start)/2;
            if(nums[mid] == target){
                return mid;
            }else if(nums[mid] < target){
                start = mid+1;
            }else{
                end = mid - 1;
            }
        }
        return -1;
    }
};
```

152 Maximum Product Subarray (link)

Description

Given an integer array `nums`, find a subarray that has the largest product, and return *the product*.

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

Note that the product of an array with a single element is the value of that element.

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:

- $1 \leq \text{nums.length} \leq 2 * 10^4$
- $-10 \leq \text{nums}[i] \leq 10$
- The product of any subarray of `nums` is **guaranteed** to fit in a **32-bit** integer.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int maxProduct(vector<int>& nums) {
        long long prefixProduct = nums[0], suffixProduct = nums[nums.size()-1],
               maxProduct = max(prefixProduct,suffixProduct);
        bool allZero = true;
        for (int i = 1; i < nums.size(); i++) {
            if (prefixProduct == 0) {
                prefixProduct = 1;
            }
            if (suffixProduct == 0) {
                suffixProduct = 1;
            }
            allZero = false;
            prefixProduct *= nums[i];
            suffixProduct *= nums[nums.size() - i - 1];
            maxProduct = max(maxProduct, max(suffixProduct, prefixProduct));
        }

        if (allZero && nums.size() > 1) {
            return 0;
        }
        return maxProduct;
    }
};
```

128 Longest Consecutive Sequence ([link](#))

Description

Given an unsorted array of integers `nums`, return *the length of the longest consecutive elements sequence*.

You must write an algorithm that runs in $O(n)$ time.

Example 1:

Input: `nums = [100, 4, 200, 1, 3, 2]`

Output: 4

Explanation: The longest consecutive elements sequence is `[1, 2, 3, 4]`. Therefore its length is 4.

Example 2:

Input: `nums = [0, 3, 7, 2, 5, 8, 4, 6, 0, 1]`

Output: 9

Example 3:

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

Output: 3

Constraints:

- $0 \leq \text{nums.length} \leq 10^5$
- $-10^9 \leq \text{nums}[i] \leq 10^9$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int longestConsecutive(vector<int>& nums) {
        unordered_set<int> set;
        for (auto& it : nums)
            set.insert(it);

        int currentSequence = 1, longestSequence = 1;
        if (nums.size() <= 1) {
            return nums.size();
        }

        for (auto setIterator: set) {
            if (set.find(setIterator - 1) == set.end()) {
                int cnt = 1;
                currentSequence = 1;
                while (set.find(setIterator + cnt) != set.end()) {
                    currentSequence++;
                    longestSequence = max(longestSequence, currentSequence);
                    cnt++;
                }
            }
        }

        return longestSequence;
    }
};
```

31 Next Permutation (link)

Description

A **permutation** of an array of integers is an arrangement of its members into a sequence or linear order.

- For example, for arr = [1,2,3], the following are all the permutations of arr: [1,2,3], [1,3,2], [2, 1, 3], [2, 3, 1], [3,1,2], [3,2,1].

The **next permutation** of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the **next permutation** of that array is the permutation that follows it in the sorted container. If such arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order).

- For example, the next permutation of arr = [1,2,3] is [1,3,2].
- Similarly, the next permutation of arr = [2,3,1] is [3,1,2].
- While the next permutation of arr = [3,2,1] is [1,2,3] because [3,2,1] does not have a lexicographical larger rearrangement.

Given an array of integers nums, *find the next permutation* of nums.

The replacement must be **in place** and use only constant extra memory.

Example 1:

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

Example 2:

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

Example 3:

```
Input: nums = [1,1,5]
Output: [1,5,1]
```

Constraints:

- $1 \leq \text{nums.length} \leq 100$
- $0 \leq \text{nums}[i] \leq 100$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    void nextPermutation(vector<int>& nums) {
        int breakPoint = -1;

        for(int i = nums.size()-1; i>0; i--){
            if(nums[i-1] < nums[i]){
                breakPoint = i-1;
                break;
            }
        }

        if(breakPoint == -1){
            reverse(nums.begin(),nums.end());
            return;
        }

        for(int i=nums.size()-1; i>breakPoint; i--){
            if(nums[breakPoint] < nums[i]){
                swap(nums[breakPoint],nums[i]);
                break;
            }
        }

        reverse(nums.begin()+breakPoint+1,nums.end());
    }
};
```

[3974 XOR After Range Multiplication Queries I \(link\)](#)

Description

You are given an integer array `nums` of length `n` and a 2D integer array `queries` of size `q`, where `queries[i] = [li, ri, ki, vi]`.

For each query, you must apply the following operations in order:

- Set `idx = li`.
- While `idx <= ri`:
 - Update: `nums[idx] = (nums[idx] * vi) % (109 + 7)`
 - Set `idx += ki`.

Return the **bitwise XOR** of all elements in `nums` after processing all queries.

Example 1:

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

Output: 4

Explanation:

- A single query `[0, 2, 1, 4]` multiplies every element from index 0 through index 2 by 4.
- The array changes from `[1, 1, 1]` to `[4, 4, 4]`.
- The XOR of all elements is $4 \wedge 4 \wedge 4 = 4$.

Example 2:

Input: `nums = [2,3,1,5,4]`, `queries = [[1,4,2,3],[0,2,1,2]]`

Output: 31

Explanation:

- The first query `[1, 4, 2, 3]` multiplies the elements at indices 1 and 3 by 3, transforming the array to `[2, 9, 1, 15, 4]`.
- The second query `[0, 2, 1, 2]` multiplies the elements at indices 0, 1, and 2 by 2, resulting in `[4, 18, 2, 15, 4]`.
- Finally, the XOR of all elements is $4 \wedge 18 \wedge 2 \wedge 15 \wedge 4 = 31$.

Constraints:

- $1 \leq n == \text{nums.length} \leq 10^3$
- $1 \leq \text{nums}[i] \leq 10^9$
- $1 \leq q == \text{queries.length} \leq 10^3$
- `queries[i] = [li, ri, ki, vi]`
- $0 \leq l_i \leq r_i < n$
- $1 \leq k_i \leq n$
- $1 \leq v_i \leq 10^5$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int xorAfterQueries(vector<int>& nums, vector<vector<int>>& queries) {
        int mod = 1e9 + 7;

        for(int i=0; i<queries.size(); i++){
            long long left = queries[i][0];
            long long right = queries[i][1];
            long long k = queries[i][2];
            long long v = queries[i][3];
            while(left <= right){
                nums[left] = (nums[left] % mod * v % mod) % mod;
                left += k;
            }
        }
        int xorN = 0;
        for(int i=0; i<nums.size(); i++){
            xorN ^= nums[i];
        }
        return xorN;
    }
};
```

[2271 Rearrange Array Elements by Sign \(link\)](#)

Description

You are given a **0-indexed** integer array `nums` of **even** length consisting of an **equal** number of positive and negative integers.

You should return the array of `nums` such that the array follows the given conditions:

1. Every **consecutive pair** of integers have **opposite signs**.
2. For all integers with the same sign, the **order** in which they were present in `nums` is **preserved**.
3. The rearranged array begins with a positive integer.

Return *the modified array after rearranging the elements to satisfy the aforementioned conditions*.

Example 1:

Input: `nums = [3,1,-2,-5,2,-4]`
Output: `[3,-2,1,-5,2,-4]`

Explanation:

The positive integers in `nums` are `[3,1,2]`. The negative integers are `[-2,-5,-4]`.
The only possible way to rearrange them such that they satisfy all conditions is `[3,-2,1,-5,2,-4]`.
Other ways such as `[1,-2,2,-5,3,-4]`, `[3,1,2,-2,-5,-4]`, `[-2,3,-5,1,-4,2]` are incorrect because they do not preserve the original order of elements with the same sign.

Example 2:

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

Explanation:

1 is the only positive integer and -1 the only negative integer in `nums`.
So `nums` is rearranged to `[1,-1]`.

Constraints:

- $2 \leq \text{nums.length} \leq 2 * 10^5$
- `nums.length` is **even**
- $1 \leq |\text{nums}[i]| \leq 10^5$
- `nums` consists of **equal** number of positive and negative integers.

It is not required to do the modifications in-place.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    vector<int> rearrangeArray(vector<int>& nums) {
        vector<int>ans;
        int currentIndex = 0;
        int positiveIndex = 0;
        int negativeIndex = 0;

        while(positiveIndex < nums.size() && negativeIndex < nums.size()){
            while(positiveIndex < nums.size() && nums[positiveIndex] <= 0 ){
                positiveIndex++;
            }
            while(negativeIndex < nums.size() && nums[negativeIndex] > 0 ){
                negativeIndex++;
            }

            if(currentIndex %2 == 0){
                ans.push_back(nums[positiveIndex]);
                positiveIndex++;
                currentIndex++;
            } else{
                ans.push_back(nums[negativeIndex]);
                negativeIndex++;
                currentIndex++;
            }
        }

        while(positiveIndex < nums.size() && nums[positiveIndex] >= 0){
            ans.push_back(nums[positiveIndex]);
            positiveIndex++;
            currentIndex++;
        }
        while(negativeIndex < nums.size() && nums[negativeIndex] < 0){
            ans.push_back(nums[negativeIndex]);
            negativeIndex++;
            currentIndex++;
        }
    }
    return ans;
};

};
```

[136 Single Number \(link\)](#)

Description

Given a **non-empty** array of integers `nums`, every element appears *twice* except for one. Find that single one.

You must implement a solution with a linear runtime complexity and use only constant extra space.

Example 1:

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

Output: 1

Example 2:

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

Output: 4

Example 3:

Input: `nums = [1]`

Output: 1

Constraints:

- $1 \leq \text{nums.length} \leq 3 * 10^4$
- $-3 * 10^4 \leq \text{nums}[i] \leq 3 * 10^4$
- Each element in the array appears twice except for one element which appears only once.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        int xorArr = 0;

        for(int i=0; i<nums.size(); i++){
            xorArr ^= nums[i];
        }

        return xorArr;
    }
};
```

[485 Max Consecutive Ones \(link\)](#)

Description

Given a binary array `nums`, return *the maximum number of consecutive 1's in the array*.

Example 1:

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

Explanation: The first two digits or the last three digits are consecutive 1s. The maximum number of consecutive 1s is 3.

Example 2:

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

Constraints:

- $1 \leq \text{nums.length} \leq 10^5$
- `nums[i]` is either 0 or 1.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int findMaxConsecutiveOnes(vector<int>& nums) {
        int currentOne = 0;
        int maxOne = 0;

        for(int i=0; i<nums.size(); i++){
            if(nums[i] == 0){
                currentOne = 0;
                continue;
            }
            currentOne++;
            maxOne = max(maxOne,currentOne);
        }
        return maxOne;
    }
};
```

[268 Missing Number \(link\)](#)

Description

Given an array `nums` containing n distinct numbers in the range $[0, n]$, return *the only number in the range that is missing from the array*.

Example 1:

Input: `nums = [3,0,1]`

Output: 2

Explanation:

$n = 3$ since there are 3 numbers, so all numbers are in the range $[0, 3]$. 2 is the missing number in the range since it does not appear in `nums`.

Example 2:

Input: `nums = [0,1]`

Output: 2

Explanation:

$n = 2$ since there are 2 numbers, so all numbers are in the range $[0, 2]$. 2 is the missing number in the range since it does not appear in `nums`.

Example 3:

Input: `nums = [9,6,4,2,3,5,7,0,1]`

Output: 8

Explanation:

$n = 9$ since there are 9 numbers, so all numbers are in the range $[0, 9]$. 8 is the missing number in the range since it does not appear in `nums`.

Constraints:

- $n == \text{nums.length}$
- $1 \leq n \leq 10^4$
- $0 \leq \text{nums}[i] \leq n$
- All the numbers of `nums` are **unique**.

Follow up: Could you implement a solution using only $O(1)$ extra space complexity and $O(n)$ runtime complexity?

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int missingNumber(vector<int>& nums) {
        int missingNumber = 0;
        int xorN = 0;
        int xorArr = 0;

        for(int i=0; i<nums.size(); i++){
            xorArr ^= nums[i];
        }

        for(int i=0; i<= nums.size(); i++){
            xorN ^= i;
        }

        missingNumber = xorN ^ xorArr;
        return missingNumber;
    }
};
```

53 Maximum Subarray (link)

Description

Given an integer array `nums`, find the subarray with the largest sum, and return *its sum*.

Example 1:

```
Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: The subarray [4,-1,2,1] has the largest sum 6.
```

Example 2:

```
Input: nums = [1]
Output: 1
Explanation: The subarray [1] has the largest sum 1.
```

Example 3:

```
Input: nums = [5,4,-1,7,8]
Output: 23
Explanation: The subarray [5,4,-1,7,8] has the largest sum 23.
```

Constraints:

- $1 \leq \text{nums.length} \leq 10^5$
- $-10^4 \leq \text{nums}[i] \leq 10^4$

Follow up: If you have figured out the $O(n)$ solution, try coding another solution using the **divide and conquer** approach, which is more subtle.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int maxi = INT_MIN;
        int sum = 0;
        for(int i=0; i<nums.size(); i++){
            sum+=nums[i];

            if(sum > maxi){
                maxi = sum;
            }
            if(sum < 0 ){
                sum = 0;
            }
        }
        return maxi;
    }
};
```

[75 Sort Colors \(link\)](#)

Description

Given an array `nums` with n objects colored red, white, or blue, sort them [in-place](#) 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]
```

Constraints:

- $n == \text{nums.length}$
- $1 \leq n \leq 300$
- $\text{nums}[i]$ is either 0, 1, or 2.

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

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    void sortColors(vector<int>& nums) {
        if(nums.size() == 1){
            return;
        }

        int low=0,mid=0,high=nums.size()-1;

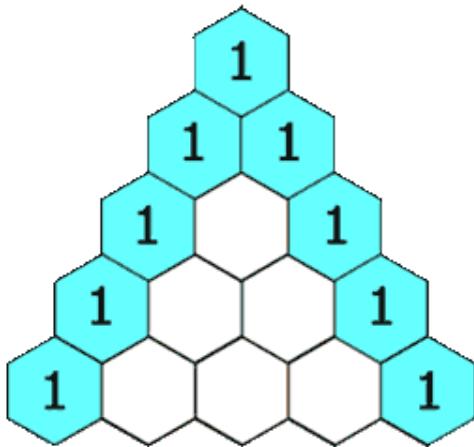
        while(mid<=high){
            if(nums[mid] == 0){
                swap(nums[low],nums[mid]);
                low++;
                mid++;
            }
            else if(nums[mid]==1){
                mid++;
            }
            else if(nums[mid]==2){
                swap(nums[mid],nums[high]);
                high--;
            }
        }
    }
};
```

[118 Pascal's Triangle \(link\)](#)

Description

Given an integer numRows, return the first numRows of **Pascal's triangle**.

In **Pascal's triangle**, each number is the sum of the two numbers directly above it as shown:



Example 1:

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

Example 2:

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

Constraints:

- $1 \leq \text{numRows} \leq 30$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
    vector<int> calculatePascalRow(int numRow) {
        long long val = 1;
        vector<int>ans;
        ans.push_back(val);
        for(int i=1; i<numRow; i++){
            val = val * (numRow-i)/i;
            ans.push_back(val);
        }
        return ans;
    }
public:
    vector<vector<int>> generate(int numRows) {
        vector<vector<int>>result;
        for(int i=1; i<=numRows; i++){
            result.push_back(calculatePascalRow(i));
        }
        return result;
    }
};
```

[144 Binary Tree Preorder Traversal \(link\)](#)

Description

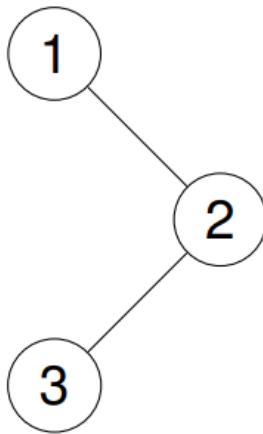
Given the root of a binary tree, return *the preorder traversal of its nodes' values.*

Example 1:

Input: root = [1,null,2,3]

Output: [1,2,3]

Explanation:

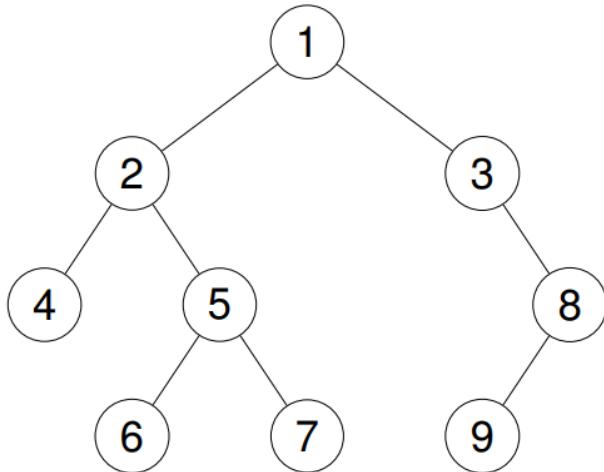


Example 2:

Input: root = [1,2,3,4,5,null,8,null,null,6,7,9]

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

Explanation:



Example 3:

Input: root = []

Output: []

Example 4:

Input: root = [1]

Output: [1]

Constraints:

- The number of nodes in the tree is in the range $[0, 100]$.
- $-100 \leq \text{Node.val} \leq 100$

Follow up: Recursive solution is trivial, could you do it iteratively?

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * 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(right) {}
 * };
 */
class Solution {
    void preorder(vector<int>&ans,TreeNode* root){
        if(root == NULL){
            return;
        }
        ans.push_back(root->val);
        preorder(ans,root->left);
        preorder(ans,root->right);
    }
public:
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int>ans;
        preorder(ans,root);
        return ans;
    }
};
```

[94 Binary Tree Inorder Traversal \(link\)](#)

Description

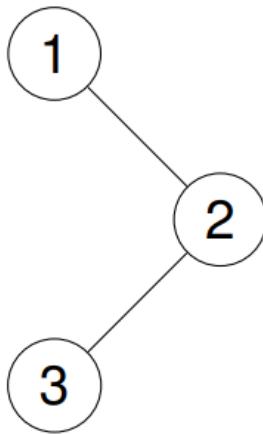
Given the root of a binary tree, return *the inorder traversal of its nodes' values.*

Example 1:

Input: root = [1,null,2,3]

Output: [1,3,2]

Explanation:

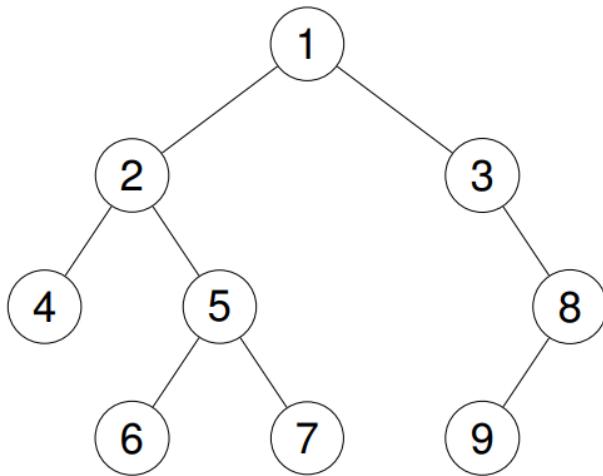


Example 2:

Input: root = [1,2,3,4,5,null,8,null,null,6,7,9]

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

Explanation:



Example 3:

Input: root = []

Output: []

Example 4:

Input: root = [1]

Output: [1]

Constraints:

- The number of nodes in the tree is in the range $[0, 100]$.
- $-100 \leq \text{Node.val} \leq 100$

Follow up: Recursive solution is trivial, could you do it iteratively?

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * 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(right) {}
 * };
 */
class Solution {
    void findInorder(TreeNode* root, vector<int>&ans){
        if(root == NULL){
            return;
        }
        findInorder(root->left,ans);
        ans.push_back(root->val);
        findInorder(root->right,ans);
    }
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int>ans;
        if(root == NULL){
            return ans;
        }
        findInorder(root,ans);
        return ans;
    }
};
```

[733 Flood Fill \(link\)](#)

Description

You are given an image represented by an $m \times n$ grid of integers `image`, where `image[i][j]` represents the pixel value of the image. You are also given three integers `sr`, `sc`, and `color`. Your task is to perform a **flood fill** on the image starting from the pixel `image[sr][sc]`.

To perform a **flood fill**:

1. Begin with the starting pixel and change its color to `color`.
2. Perform the same process for each pixel that is **directly adjacent** (pixels that share a side with the original pixel, either horizontally or vertically) and shares the **same color** as the starting pixel.
3. Keep **repeating** this process by checking neighboring pixels of the *updated* pixels and modifying their color if it matches the original color of the starting pixel.
4. The process **stops** when there are **no more** adjacent pixels of the original color to update.

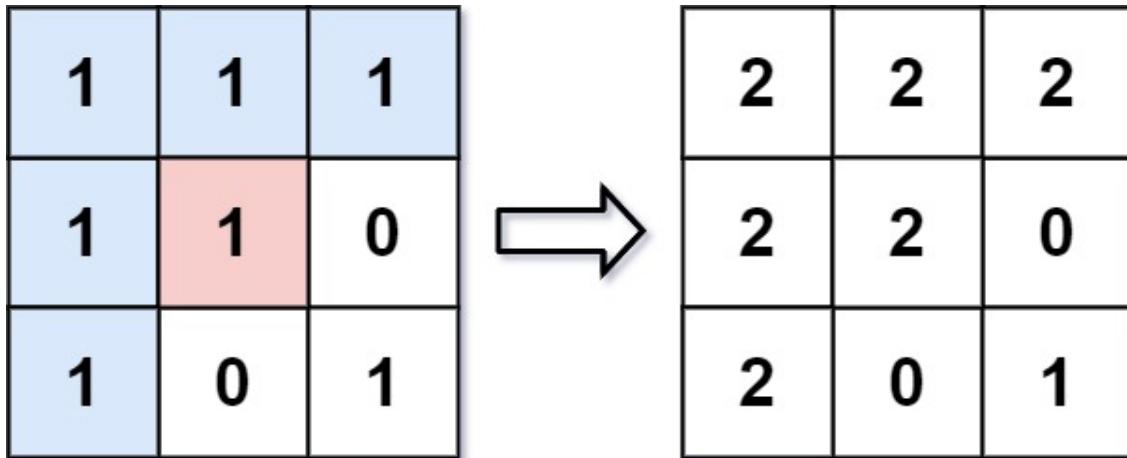
Return the **modified** image after performing the flood fill.

Example 1:

Input: `image = [[1,1,1],[1,1,0],[1,0,1]]`, `sr = 1`, `sc = 1`, `color = 2`

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

Explanation:



From the center of the image with position $(sr, sc) = (1, 1)$ (i.e., the red pixel), all pixels connected by a path of the same color as the starting pixel (i.e., the blue pixels) are colored with the new color.

Note the bottom corner is **not** colored 2, because it is not horizontally or vertically connected to the starting pixel.

Example 2:

Input: `image = [[0,0,0],[0,0,0]]`, `sr = 0`, `sc = 0`, `color = 0`

Output: `[[0,0,0],[0,0,0]]`

Explanation:

The starting pixel is already colored with 0, which is the same as the target color. Therefore, no changes are made to the image.

Constraints:

- $m == \text{image.length}$
- $n == \text{image[i].length}$
- $1 \leq m, n \leq 50$
- $0 \leq \text{image}[i][j], \text{color} < 2^{16}$
- $0 \leq sr < m$
- $0 \leq sc < n$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
    void dfs(vector<vector<int>>& ans, int sr, int sc, vector<int>& directionx,
             vector<int>& directiony, int color, int initialColor)
    {
        ans[sr][sc] = color;
        int n = ans.size();
        int m = ans[0].size();

        for(int i=0; i<4; i++){
            int nrow = sr + directionx[i];
            int ncol = sc + directiony[i];

            if(nrow<=0 && nrow < n && ncol<=0 && ncol<m
               && ans[nrow][ncol]==initialColor && ans[nrow][ncol]!=color){
                dfs(ans, nrow, ncol, directionx, directiony, color, initialColor);
            }
        }
    }

public:
    vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc,
                                      int color) {
        vector<vector<int>> ans = image;
        int initialColor = ans[sr][sc];

        vector<int> directionx = {0, -1, 0, 1}; // LURD
        vector<int> directiony = {-1, 0, 1, 0}; // LURD

        dfs(ans, sr, sc, directionx, directiony, color, initialColor);

        return ans;
    }
};
```

200 Number of Islands (link)

Description

Given an $m \times n$ 2D binary grid `grid` which represents a map of '1's (land) and '0's (water), return *the number of islands*.

An **island** is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:

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

Example 2:

```
Input: grid = [
    ["1","1","0","0","0"],
    ["1","1","0","0","0"],
    ["0","0","1","0","0"],
    ["0","0","0","1","1"]
]
Output: 3
```

Constraints:

- $m == \text{grid.length}$
- $n == \text{grid}[i].length$
- $1 \leq m, n \leq 300$
- $\text{grid}[i][j]$ is '0' or '1'.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
    void bfs(int row, int col, vector<vector<bool>>& visited,
             vector<vector<char>>& grid, int n, int m) {
        visited[row][col] = true;
        queue<pair<int, int>> q;
        q.push({row, col});

        while (!q.empty()) {
            auto front = q.front();
            q.pop();
            int qrow = front.first;
            int qcol = front.second;
            vector<int> directionRow = {0, -1, 0, 1}; // LURD
            vector<int> directionCol = {-1, 0, 1, 0};

            for (int i = 0; i < 4; i++) {
                int nrow = qrow + directionRow[i];
                int ncol = qcol + directionCol[i];

                if (nrow >= 0 && nrow < n && ncol >= 0 && ncol < m &&
                    visited[nrow][ncol] == false && grid[nrow][ncol] == '1') {
                    q.push({nrow, ncol});
                    visited[nrow][ncol] = true;
                }
            }
        }
    }

public:
    int numIslands(vector<vector<char>>& grid) {
        int n = grid.size();
        int m = grid[0].size();
        int cnt = 0;
        vector<vector<bool>> visited(n, vector<bool>(m, false));
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                if (!visited[i][j] && grid[i][j] == '1') {
                    cnt++;
                    bfs(i, j, visited, grid, n, m);
                }
            }
        }
        return cnt;
    }
};
```

1300 Critical Connections in a Network (link)

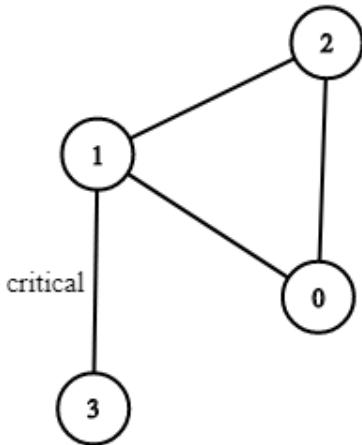
Description

There are n servers numbered from 0 to $n - 1$ connected by undirected server-to-server connections forming a network where $\text{connections}[i] = [a_i, b_i]$ represents a connection between servers a_i and b_i . Any server can reach other servers directly or indirectly through the network.

A *critical connection* is a connection that, if removed, will make some servers unable to reach some other server.

Return all critical connections in the network in any order.

Example 1:



```
Input: n = 4, connections = [[0,1],[1,2],[2,0],[1,3]]  
Output: [[1,3]]  
Explanation: [[3,1]] is also accepted.
```

Example 2:

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

Constraints:

- $2 \leq n \leq 10^5$
- $n - 1 \leq \text{connections.length} \leq 10^5$
- $0 \leq a_i, b_i \leq n - 1$
- $a_i \neq b_i$
- There are no repeated connections.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {

    void dfs(int node, int timer, int parent, vector<int>& discovery, vector<bool>& visited, vector<int>& low, vector<vector<int>>& adj, vector<vector<int>>& result) {
        visited[node] = true;
        discovery[node] = low[node] = timer++;

        for(auto neighbour: adj[node]){
            if(neighbour == parent){
                continue;
            }
            if(!visited[neighbour]){
                dfs(neighbour, timer, node, discovery, visited, low, adj, result);
                low[node] = min(low[node], low[neighbour]);
                if(low[neighbour] > discovery[node]){
                    result.push_back({node, neighbour});
                }
            } else{
                //Backedge
                low[node] = min(low[node], discovery[neighbour]);
            }
        }
    }

public:
    vector<vector<int>> criticalConnections(int n, vector<vector<int>>& connections) {
        vector<vector<int>>adj(n);
        for(int i=0; i<connections.size(); i++){
            int u = connections[i][0];
            int v = connections[i][1];

            adj[u].push_back(v);
            adj[v].push_back(u);
        }

        vector<int> discovery(n, -1);
        vector<int> low(n, -1);
        vector<bool> visited(n, false);
        int parent = -1;
        int timer=0;

        vector<vector<int>>result;

        for(int i=0; i<n; i++){
            if(!visited[i]){
                dfs(i, timer, parent, discovery, visited, low, adj, result);
            }
        }
        return result;
    }
};
```

1972 Rotating the Box (link)

Description

You are given an $m \times n$ matrix of characters `boxGrid` representing a side-view of a box. Each cell of the box is one of the following:

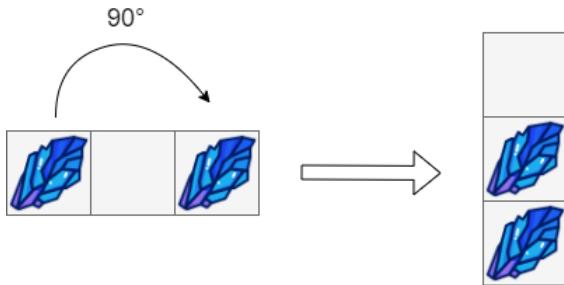
- A stone '#'
- A stationary obstacle '*'
- Empty '..'

The box is rotated **90 degrees clockwise**, causing some of the stones to fall due to gravity. Each stone falls down until it lands on an obstacle, another stone, or the bottom of the box. Gravity **does not** affect the obstacles' positions, and the inertia from the box's rotation **does not** affect the stones' horizontal positions.

It is **guaranteed** that each stone in `boxGrid` rests on an obstacle, another stone, or the bottom of the box.

Return an $n \times m$ matrix representing the box after the rotation described above.

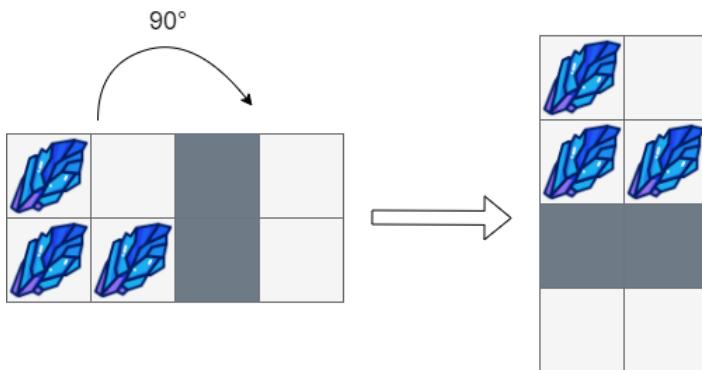
Example 1:



Input: `boxGrid = [["#", ".", "#"]]`

Output: `[[".."],
["#"],
["#"]]`

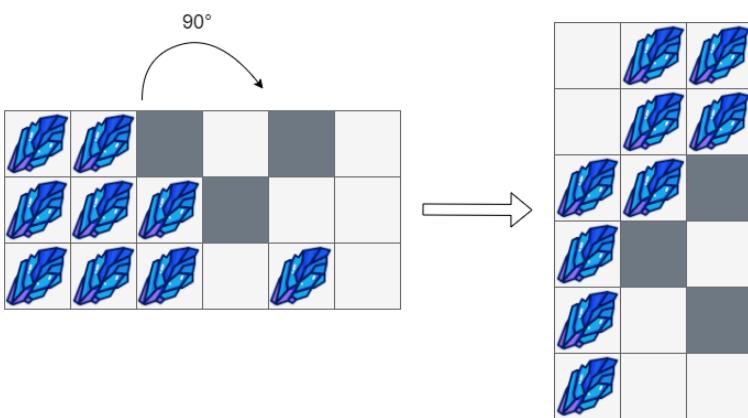
Example 2:



Input: `boxGrid = [["#", ".", "*", "."],
["#", "#", "*", "."]]`

Output: `[["#", "."],
["#", "#"],
["*", "*"],
["..", ".."]]`

Example 3:



Input: boxGrid = [[#, #, *, ., *, .],
[#, #, #, *, ., .],
[#, #, #, ., #, .]]

Output: [[., #, #],
[., #, #],
[#, #, *],
[#, *, .],
[#, ., *],
[#, ., .]]

Constraints:

- m == boxGrid.length
- n == boxGrid[i].length
- 1 <= m, n <= 500
- boxGrid[i][j] is either '#', '*', or '.'.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    vector<vector<char>> rotateTheBox(vector<vector<char>>& box) {
        int m = box.size();
        int n = box[0].size();

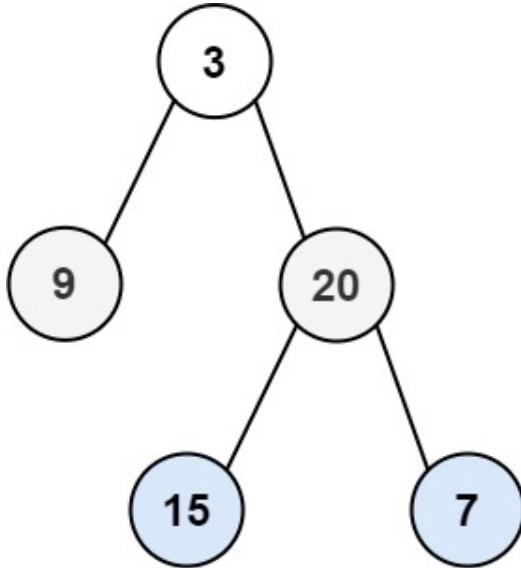
        vector<vector<char>> res(n, vector<char>(m));
        // Trasposing the matrix
        for (int i = 0; i < n; i++) {
            for (int j = m - 1; j >= 0; j--) {
                res[i][m - 1 - j] = box[j][i];
            }
        }
        // Applying the gravity
        for (int i = 0; i < m; i++) {
            int start = n - 1;
            for (int j = start; j >= 0; j--) {
                if (res[j][i] == '*') {
                    start = j - 1;
                } else if (res[j][i] == '#') {
                    res[j][i] = '.';
                    res[start][i] = '#';
                    start--;
                }
            }
        }
        return res;
    }
};
```

[103 Binary Tree Zigzag Level Order Traversal \(link\)](#)

Description

Given the root of a binary tree, return *the zigzag level order traversal of its nodes' values*. (i.e., from left to right, then right to left for the next level and alternate between).

Example 1:



```
Input: root = [3,9,20,null,null,15,7]
Output: [[3],[20,9],[15,7]]
```

Example 2:

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

Example 3:

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

Constraints:

- The number of nodes in the tree is in the range $[0, 2000]$.
- $-100 \leq \text{Node.val} \leq 100$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * 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(right) {}
 * };
 */
class Solution {
public:
    vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
        vector<vector<int>> finalans;
        queue<TreeNode*> q;
        if(root==NULL){
            return finalans;
        }
        q.push(root);
        bool leftToRight = true;

        while(!q.empty()){
            int size=q.size();
            int index=0;
            vector<int> ans(size);

            for(int i=0; i<size; i++){
                TreeNode* temp=q.front();
                q.pop();

                if(leftToRight){
                    index = i;
                }else{
                    index = size-i-1;
                }
                ans[index]=temp->val;
                if(temp->left){
                    q.push(temp->left);
                }
                if(temp->right){
                    q.push(temp->right);
                }
            }
            leftToRight=!leftToRight;
            finalans.push_back(ans);
        }
        return finalans;
    }
};
```

142 Linked List Cycle II (link)

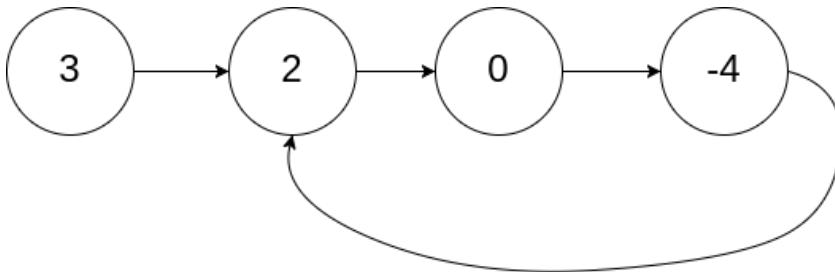
Description

Given the head of a linked list, return *the node where the cycle begins. If there is no cycle, return null.*

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to (**0-indexed**). It is **-1** if there is no cycle. **Note that pos is not passed as a parameter.**

Do not modify the linked list.

Example 1:

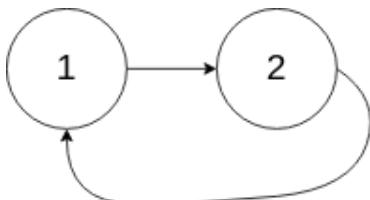


Input: head = [3,2,0,-4], pos = 1

Output: tail connects to node index 1

Explanation: There is a cycle in the linked list, where tail connects to the second node.

Example 2:



Input: head = [1,2], pos = 0

Output: tail connects to node index 0

Explanation: There is a cycle in the linked list, where tail connects to the first node.

Example 3:



Input: head = [1], pos = -1

Output: no cycle

Explanation: There is no cycle in the linked list.

Constraints:

- The number of the nodes in the list is in the range $[0, 10^4]$.
- $-10^5 \leq \text{Node.val} \leq 10^5$
- pos is **-1** or a **valid index** in the linked-list.

Follow up: Can you solve it using $O(1)$ (i.e. constant) memory?

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
/*
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        ListNode* fast=head;
        ListNode* slow=head;
        while(fast!=NULL && fast->next!=NULL && slow!=NULL){
            fast=fast->next->next;
            slow=slow->next;
            if(slow==fast){
                slow=head;
                while(fast!=slow){
                    slow=slow->next;
                    fast=fast->next;
                }
                return slow;
            }
        }
        return NULL;
    }
};
```

[240 Search a 2D Matrix II \(link\)](#)

Description

Write an efficient algorithm that searches for a value target in an $m \times n$ integer matrix matrix. This matrix has the following properties:

- Integers in each row are sorted in ascending from left to right.
- Integers in each column are sorted in ascending from top to bottom.

Example 1:

1	4	7	11	15
2	5	8	12	19
3	6	9	16	22
10	13	14	17	24
18	21	23	26	30

```
Input: matrix = [[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],[18,21,23,26,30]],  
Output: true
```

Example 2:

1	4	7	11	15
2	5	8	12	19
3	6	9	16	22
10	13	14	17	24
18	21	23	26	30

```
Input: matrix = [[1,4,7,11,15],[2,5,8,12,19],[3,6,9,16,22],[10,13,14,17,24],[18,21,23,26,30]],  
Output: false
```

Constraints:

- $m == \text{matrix.length}$
- $n == \text{matrix}[i].length$
- $1 \leq n, m \leq 300$
- $-10^9 \leq \text{matrix}[i][j] \leq 10^9$
- All the integers in each row are **sorted** in ascending order.
- All the integers in each column are **sorted** in ascending order.
- $-10^9 \leq \text{target} \leq 10^9$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        int row=matrix.size();
        int col=matrix[0].size();

        int s=0,e=col-1;
        while(s<row && e>=0){
            int element=matrix[s][e];
            if(element == target){
                return 1;
            }
            if(element>target){
                e--;
            }
            else{
                s++;
            }
        }
        return 0;
    }
};
```

[74 Search a 2D Matrix \(link\)](#)

Description

You are given an $m \times n$ integer matrix `matrix` with the following two properties:

- Each row is sorted in non-decreasing order.
- The first integer of each row is greater than the last integer of the previous row.

Given an integer target, return true if target is in matrix or false otherwise.

You must write a solution in $O(\log(m * n))$ time complexity.

Example 1:

1	3	5	7
10	11	16	20
23	30	34	60

```
Input: matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3
Output: true
```

Example 2:

1	3	5	7
10	11	16	20
23	30	34	60

```
Input: matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 13
Output: false
```

Constraints:

- $m == \text{matrix.length}$
- $n == \text{matrix[i].length}$
- $1 \leq m, n \leq 100$

- $-10^4 \leq \text{matrix}[i][j], \text{target} \leq 10^4$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        int row=matrix.size();
        int col=matrix[0].size();

        int s=0,e=row*col-1;

        while(s<=e){
            int mid=s+(e-s)/2;
            int element=matrix[mid/col][mid%col];
            if(element==target){
                return 1;
            }if(element>target){
                e=mid-1;
            }else{
                s=mid+1;
            }
        }

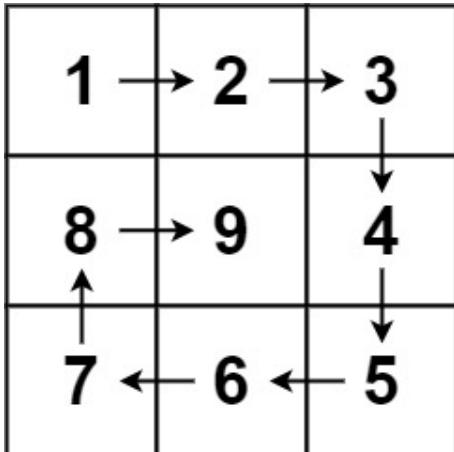
        return 0;
    }
};
```

[59 Spiral Matrix II \(link\)](#)

Description

Given a positive integer n , generate an $n \times n$ matrix filled with elements from 1 to n^2 in spiral order.

Example 1:



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

Example 2:

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

Constraints:

- $1 \leq n \leq 20$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    vector<vector<int>> generateMatrix(int n) {

        vector<vector<int>> ans(n, vector<int>(n, 0));
        int startingRow=0;
        int endingRow=n-1;
        int startingCol=0;
        int endingCol=n-1;
        int total=n*n;
        int count=1;
        while(count<=total){
            if(count<=total){
                for(int i=startingCol; i<=endingCol; i++){
                    ans[startingRow][i]=count;count++;
                }startingRow++;
            }
            if(count<=total){
                for(int i=startingRow; i<=endingRow; i++){
                    ans[i][endingCol]=count;count++;
                }endingCol--;
            }
            if(count<=total){
                for(int i=endingCol; i>=startingCol;i--){
                    ans[endingRow][i]=count;count++;
                }endingRow--;
            }
            if(count<=total){
                for(int i=endingRow; i>=startingRow; i--){
                    ans[i][startingCol]=count;count++;
                }startingCol++;
            }
        }
        // endingCol=
    }

    return ans;
};

};
```

[443 String Compression \(link\)](#)

Description

Given an array of characters `chars`, compress it using the following algorithm:

Begin with an empty string `s`. For each group of **consecutive repeating characters** in `chars`:

- If the group's length is 1, append the character to `s`.
- Otherwise, append the character followed by the group's length.

The compressed string `s` **should not be returned separately**, but instead, be stored **in the input character array `chars`**. Note that group lengths that are 10 or longer will be split into multiple characters in `chars`.

After you are done **modifying the input array**, return *the new length of the array*.

You must write an algorithm that uses only constant extra space.

Note: The characters in the array beyond the returned length do not matter and should be ignored.

Example 1:

Input: `chars = ["a","a","b","b","c","c","c"]`

Output: Return 6, and the first 6 characters of the input array should be: `["a","2","b","2","c","3"]`

Explanation: The groups are "aa", "bb", and "ccc". This compresses to "a2b2c3".

Example 2:

Input: `chars = ["a"]`

Output: Return 1, and the first character of the input array should be: `["a"]`

Explanation: The only group is "a", which remains uncompressed since it's a single character.

Example 3:

Input: `chars = ["a","b","b","b","b","b","b","b","b","b","b"]`

Output: Return 4, and the first 4 characters of the input array should be: `["a","b","1","2"]`.

Explanation: The groups are "a" and "bbbbbbbbbb". This compresses to "ab12".

Constraints:

- $1 \leq \text{chars.length} \leq 2000$
- `chars[i]` is a lowercase English letter, uppercase English letter, digit, or symbol.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int compress(vector<char>& chars) {
        int i=0, index=0;
        while(i<chars.size()){
            int j=i+1;
            while(j<chars.size() && chars[i]==chars[j]){
                j++;
            }

            chars[index++]=chars[i];
            int count = j-i;

            if(count>1){
                string cnt=to_string(count);

                for(char ch:cnt){
                    chars[index++]=ch;
                }
            }

            i=j;
        }
        return index;
    }
};
```

[1128 Remove All Adjacent Duplicates In String \(link\)](#)

Description

You are given a string s consisting of lowercase English letters. A **duplicate removal** consists of choosing two **adjacent** and **equal** letters and removing them.

We repeatedly make **duplicate removals** on s until we no longer can.

Return *the final string after all such duplicate removals have been made*. It can be proven that the answer is **unique**.

Example 1:

Input: $s = "abbaca"$

Output: "ca"

Explanation:

For example, in "abbaca" we could remove "bb" since the letters are adjacent and equal, and th:

Example 2:

Input: $s = "azxxzy"$

Output: "ay"

Constraints:

- $1 \leq s.length \leq 10^5$
- s consists of lowercase English letters.

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    string removeDuplicates(string s) {
        int i=0;
        int n=s.length();
        while(i<n){
            if(s[i]==s[i+1]){
                s.erase(i,2);
                i=0;
            }else{
                i++;
            }
        }
        return s;
    }
};
```

[344 Reverse String \(link\)](#)

Description

Write a function that reverses a string. The input string is given as an array of characters s .

You must do this by modifying the input array [in-place](#) with $O(1)$ extra memory.

Example 1:

```
Input: s = ["h","e","l","l","o"]
Output: ["o","l","l","e","h"]
```

Example 2:

```
Input: s = ["H","a","n","n","a","h"]
Output: ["h","a","n","n","a","H"]
```

Constraints:

- $1 \leq s.length \leq 10^5$
- $s[i]$ is a [printable ascii character](#).

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    void reverseString(vector<char>& s) {
        int i=0,j=s.size()-1;
        while(i<j){
            swap(s[i++],s[j--]);
        }
    };
};
```

[1878 Check if Array Is Sorted and Rotated \(link\)](#)

Description

Given an array `nums`, return `true` if the array was originally sorted in non-decreasing order, then rotated some number of positions (including zero). Otherwise, return `false`.

There may be **duplicates** in the original array.

Note: An array `A` rotated by x positions results in an array `B` of the same length such that $B[i] == A[(i+x) \% A.length]$ for every valid index i .

Example 1:

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

Output: `true`

Explanation: `[1,2,3,4,5]` is the original sorted array.

You can rotate the array by $x = 2$ positions to begin on the element of value 3: `[3,4,5,1,2]`.

Example 2:

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

Output: `false`

Explanation: There is no sorted array once rotated that can make `nums`.

Example 3:

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

Output: `true`

Explanation: `[1,2,3]` is the original sorted array.

You can rotate the array by $x = 0$ positions (i.e. no rotation) to make `nums`.

Constraints:

- $1 \leq \text{nums.length} \leq 100$
- $1 \leq \text{nums}[i] \leq 100$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    bool check(vector<int>& nums) {
        int i=0,c=0;
        while(i<nums.size()-1){
            if(nums[i]>nums[i+1]){
                c++;
            }
            i++;
        }
        if(nums.size()-1>nums[0]){
            c++;
        }
        if(c<=1){
            return true;
        }else{
            return false;
        }
    }
};
```

189 Rotate Array ([link](#))

Description

Given an integer array `nums`, rotate the array to the right by `k` steps, where `k` is non-negative.

Example 1:

```
Input: nums = [1,2,3,4,5,6,7], k = 3
Output: [5,6,7,1,2,3,4]
Explanation:
rotate 1 steps to the right: [7,1,2,3,4,5,6]
rotate 2 steps to the right: [6,7,1,2,3,4,5]
rotate 3 steps to the right: [5,6,7,1,2,3,4]
```

Example 2:

```
Input: nums = [-1,-100,3,99], k = 2
Output: [3,99,-1,-100]
Explanation:
rotate 1 steps to the right: [99,-1,-100,3]
rotate 2 steps to the right: [3,99,-1,-100]
```

Constraints:

- $1 \leq \text{nums.length} \leq 10^5$
- $-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$
- $0 \leq k \leq 10^5$

Follow up:

- Try to come up with as many solutions as you can. There are at least **three** different ways to solve this problem.
- Could you do it in-place with $O(1)$ extra space?

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    void rotate(vector<int>& nums, int k) {
        k=k%nums.size();
        reverse(nums.begin(),nums.begin()+(nums.size()-k));
        reverse(nums.begin()+(nums.size()-k),nums.end());
        reverse(nums.begin(),nums.end());
    };
};
```

[283 Move Zeroes \(link\)](#)

Description

Given an integer array `nums`, move all `0`'s to the end of it while maintaining the relative order of the non-zero elements.

Note that you must do this in-place without making a copy of the array.

Example 1:

```
Input: nums = [0,1,0,3,12]
Output: [1,3,12,0,0]
```

Example 2:

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

Constraints:

- $1 \leq \text{nums.length} \leq 10^4$
- $-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$

Follow up: Could you minimize the total number of operations done?

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    void moveZeroes(vector<int>& nums) {
        int i=0;
        for(int j=0; j<nums.size(); j++){
            if(nums[j]!=0){
                swap(nums[j],nums[i]);
                i++;
            }
        }
    }
};
```

88 Merge Sorted Array (link)

Description

You are given two integer arrays `nums1` and `nums2`, sorted in **non-decreasing order**, and two integers `m` and `n`, representing the number of elements in `nums1` and `nums2` respectively.

Merge `nums1` and `nums2` into a single array sorted in **non-decreasing order**.

The final sorted array should not be returned by the function, but instead be *stored inside the array* `nums1`. To accommodate this, `nums1` has a length of `m + n`, where the first `m` elements denote the elements that should be merged, and the last `n` elements are set to `0` and should be ignored. `nums2` has a length of `n`.

Example 1:

Input: `nums1 = [1,2,3,0,0,0]`, `m = 3`, `nums2 = [2,5,6]`, `n = 3`

Output: `[1,2,2,3,5,6]`

Explanation: The arrays we are merging are `[1,2,3]` and `[2,5,6]`.

The result of the merge is `[1,2,2,3,5,6]` with the underlined elements coming from `nums1`.

Example 2:

Input: `nums1 = [1]`, `m = 1`, `nums2 = []`, `n = 0`

Output: `[1]`

Explanation: The arrays we are merging are `[1]` and `[]`.

The result of the merge is `[1]`.

Example 3:

Input: `nums1 = [0]`, `m = 0`, `nums2 = [1]`, `n = 1`

Output: `[1]`

Explanation: The arrays we are merging are `[]` and `[1]`.

The result of the merge is `[1]`.

Note that because `m = 0`, there are no elements in `nums1`. The `0` is only there to ensure the merge

Constraints:

- `nums1.length == m + n`
- `nums2.length == n`
- `0 <= m, n <= 200`
- `1 <= m + n <= 200`
- `-109 <= nums1[i], nums2[j] <= 109`

Follow up: Can you come up with an algorithm that runs in $O(m + n)$ time?

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
        int i = m - 1;
        int j = n - 1;
        int k = m + n - 1;
        while (j >= 0) {
            if (i >= 0 && nums1[i] > nums2[j]) {
                nums1[k--] = nums1[i--];
            } else {
                nums1[k--] = nums2[j--];
            }
        }
    }
};
```

[724 Find Pivot Index \(link\)](#)

Description

Given an array of integers `nums`, calculate the **pivot index** of this array.

The **pivot index** is the index where the sum of all the numbers **strictly** to the left of the index is equal to the sum of all the numbers **strictly** to the index's right.

If the index is on the left edge of the array, then the left sum is 0 because there are no elements to the left. This also applies to the right edge of the array.

Return *the leftmost pivot index*. If no such index exists, return -1 .

Example 1:

```
Input: nums = [1,7,3,6,5,6]
Output: 3
Explanation:
The pivot index is 3.
Left sum = nums[0] + nums[1] + nums[2] = 1 + 7 + 3 = 11
Right sum = nums[4] + nums[5] = 5 + 6 = 11
```

Example 2:

```
Input: nums = [1,2,3]
Output: -1
Explanation:
There is no index that satisfies the conditions in the problem statement.
```

Example 3:

```
Input: nums = [2,1,-1]
Output: 0
Explanation:
The pivot index is 0.
Left sum = 0 (no elements to the left of index 0)
Right sum = nums[1] + nums[2] = 1 + -1 = 0
```

Constraints:

- $1 \leq \text{nums.length} \leq 10^4$
- $-1000 \leq \text{nums}[i] \leq 1000$

Note: This question is the same as 1991: <https://leetcode.com/problems/find-the-middle-index-in-array/>

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int pivotIndex(vector<int>& nums) {
        int lsum = 0, rsum = 0, i = 0;
        for(int i=0; i<nums.size(); i++){
            rsum+=nums[i];
        }
        while (i < nums.size()) {
            if (lsum == (rsum-nums[i])) {
                return i;
            }else{
                lsum+=nums[i];
                rsum-=nums[i];
            }
            i++;
        }
        return -1;
    }
};
```

[342 Power of Four \(link\)](#)

Description

Given an integer n , return *true if it is a power of four. Otherwise, return false.*

An integer n is a power of four, if there exists an integer x such that $n == 4^x$.

Example 1:

```
Input: n = 16
Output: true
```

Example 2:

```
Input: n = 5
Output: false
```

Example 3:

```
Input: n = 1
Output: true
```

Constraints:

- $-2^{31} \leq n \leq 2^{31} - 1$

Follow up: Could you solve it without loops/recursion?

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    bool isPowerOfFour(int n) {
        int i=0;
        while(i<=30){
            if(pow(4,i)==n){
                return 1;
            }
            i++;
        }
        return 0;
    }
};
```

[326 Power of Three \(link\)](#)

Description

Given an integer n , return *true if it is a power of three. Otherwise, return false.*

An integer n is a power of three, if there exists an integer x such that $n == 3^x$.

Example 1:

```
Input: n = 27
Output: true
Explanation: 27 = 33
```

Example 2:

```
Input: n = 0
Output: false
Explanation: There is no x where 3x = 0.
```

Example 3:

```
Input: n = -1
Output: false
Explanation: There is no x where 3x = (-1).
```

Constraints:

- $-2^{31} \leq n \leq 2^{31} - 1$

Follow up: Could you solve it without loops/recursion?

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    bool isPowerOfThree(int n) {
        for(int i=0;i<31;i++){
            if(pow(3,i)==n){
                return true;
            }
        }
        return false;
    }
};
```

[231 Power of Two \(link\)](#)

Description

Given an integer n , return *true if it is a power of two. Otherwise, return false.*

An integer n is a power of two, if there exists an integer x such that $n == 2^x$.

Example 1:

```
Input: n = 1
Output: true
Explanation: 20 = 1
```

Example 2:

```
Input: n = 16
Output: true
Explanation: 24 = 16
```

Example 3:

```
Input: n = 3
Output: false
```

Constraints:

- $-2^{31} \leq n \leq 2^{31} - 1$

Follow up: Could you solve it without loops/recursion?

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    bool isPowerOfTwo(int n) {
        for(int i=0;i<31;i++){
            if(pow(2,i)==n){
                return true;
            }
        }
        return false;
    }
};
```

1054 Complement of Base 10 Integer ([link](#))

Description

The **complement** of an integer is the integer you get when you flip all the 0's to 1's and all the 1's to 0's in its binary representation.

- For example, The integer 5 is "101" in binary and its **complement** is "010" which is the integer 2.

Given an integer n , return *its complement*.

Example 1:

Input: $n = 5$

Output: 2

Explanation: 5 is "101" in binary, with complement "010" in binary, which is 2 in base-10.

Example 2:

Input: $n = 7$

Output: 0

Explanation: 7 is "111" in binary, with complement "000" in binary, which is 0 in base-10.

Example 3:

Input: $n = 10$

Output: 5

Explanation: 10 is "1010" in binary, with complement "0101" in binary, which is 5 in base-10.

Constraints:

- $0 \leq n < 10^9$

Note: This question is the same as 476: <https://leetcode.com/problems/number-complement/>

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int bitwiseComplement(int n) {
        int mask = 0;
        int m = n;
        if (n == 0) {
            return 1;
        }
        while (m != 0) {
            m = m >> 1;
            mask = (mask << 1) | 1;
        }

        int ans;
        ans = (~n) & mask;
        return ans;
    }
};
```

7 Reverse Integer (link)

Description

Given a signed 32-bit integer x , return x *with its digits reversed*. If reversing x causes the value to go outside the signed 32-bit integer range $[-2^{31}, 2^{31} - 1]$, then return 0.

Assume the environment does not allow you to store 64-bit integers (signed or unsigned).

Example 1:

```
Input: x = 123
Output: 321
```

Example 2:

```
Input: x = -123
Output: -321
```

Example 3:

```
Input: x = 120
Output: 21
```

Constraints:

- $-2^{31} \leq x \leq 2^{31} - 1$

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int reverse(int x) {
        int rev = 0;
        while (x != 0) {

            int rem = x % 10;
            if ((rev > INT_MAX / 10) || (rev < INT_MIN / 10)) {
                return 0;
            }
            rev = (rev * 10) + rem;

            x = x / 10;
        }

        return rev;
    }
};
```

[169 Majority Element \(link\)](#)

Description

Given an array `nums` of size `n`, return *the majority element*.

The majority element is the element that appears more than $\lfloor n / 2 \rfloor$ times. You may assume that the majority element always exists in the array.

Example 1:

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

Example 2:

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

Constraints:

- `n == nums.length`
- $1 \leq n \leq 5 * 10^4$
- $-10^9 \leq \text{nums}[i] \leq 10^9$
- The input is generated such that a majority element will exist in the array.

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

(scroll down for solution)

Solution

Language: cpp

Status: Accepted

```
class Solution {
public:
    int majorityElement(vector<int>& nums) {
        unordered_map <int,int> mpp;
        int maj=0;
        int num=nums.size()/2;
        for(auto a:nums){
            mpp[a]++;
        }
        int ans=0;
        for(auto it : mpp){
            if(it.second>num){
                ans= it.first;
            }
        }
        return ans;
    };
};
```