MICROSOFT

By Jani Ahamed Habeeb Mohamed

Table of Contents

1. Problems

1.1 Minimum moves to spread stones over grid

You are given a 0-indexed 2D integer matrix grid of size 3 * 3, representing the number of stones in each cell. The grid contains exactly 9 stones, and there can be multiple stones in a single cell.

In one move, you can move a single stone from its current cell to any other cell if the two cells share a side.

Return *the minimum number of moves required to place one stone in each cell*.

Example 1:

Input: grid = [[1,1,0],[1,1,1],[1,2,1]]
Output: 3
Explanation: One possible sequence of moves to place one stone in each cell is:
1- Move one stone from cell (2,1) to cell (2,2).
2- Move one stone from cell (2,2) to cell (1,2).
3- Move one stone from cell (1,2) to cell (0,2).
In total, it takes 3 moves to place one stone in each cell of the grid.
It can be shown that 3 is the minimum number of moves required to place one stone in each cell.

Example 2:

Input: grid = [[1,3,0],[1,0,0],[1,0,3]]
Output: 4
Explanation: One possible sequence of moves to place one stone in each cell is:
1- Move one stone from cell (0,1) to cell (0,2).
2- Move one stone from cell (0,1) to cell (1,1).
3- Move one stone from cell (2,2) to cell (1,2).
4- Move one stone from cell (2,2) to cell (2,1).
In total, it takes 4 moves to place one stone in each cell of the grid.
It can be shown that 4 is the minimum number of moves required to place one stone in each cell.

C:
```c
#define MAX_SIZE 3

int min(int a, int b) {
    return a < b ? a : b;
}

int minimumMoves(int** grid, int gridSize, int* gridColSize) {
```

```
    int ans = INT_MAX;

    for (int i = 0; i < MAX_SIZE; ++i) {
        for (int j = 0; j < MAX_SIZE; ++j) {
            if (grid[i][j]) {
                continue;
            }

            for (int a = 0; a < MAX_SIZE; ++a) {
                for (int b = 0; b < MAX_SIZE; ++b) {
                    if (grid[a][b] < 2) {
                        continue;
                    }

                    int res = abs(i - a) + abs(j - b);
                    --grid[a][b];
                    ++grid[i][j];
                    res += minimumMoves(grid, gridSize, gridColSize);
                    ++grid[a][b];
                    --grid[i][j];
                    ans = min(ans, res);
                }
            }
        }
    }

    return (ans == INT_MAX ? 0 : ans);
}
```

1.2   LRU Cache

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:

C :

```c
#define MAX_KEYS 10000

typedef struct Node {
    int key;
    int value;
    struct Node* next;
    struct Node* prev;
} Node;

// Define the LRUCache structure
typedef struct {
    int capacity;
    int size;
    Node* head;
    Node* tail;
    Node cache[MAX_KEYS]; // Static cache array for quick lookups
} LRUCache;

// Function to create an LRUCache
LRUCache* lRUCacheCreate(int capacity) {
    LRUCache* obj = (LRUCache*)malloc(sizeof(LRUCache));
    obj->capacity = capacity;
    obj->size = 0;
    obj->head = NULL;
    obj->tail = NULL;

    // Initialize the cache array
    for (int i = 0; i < MAX_KEYS; ++i) {
        obj->cache[i].key = -1; // Mark unused entries with -1
        obj->cache[i].next = NULL;
        obj->cache[i].prev = NULL;
    }
```

```c
        return obj;
}

// Function to get the value of a key from the LRUCache
int lRUCacheGet(LRUCache* obj, int key) {
    if (obj->cache[key].key == -1) {
        return -1; // Key not found
    }

    // Move the accessed node to the front
    Node* current = &obj->cache[key];
    if (current != obj->head) {
        if (current == obj->tail) {
            obj->tail = current->prev;
        } else {
            current->next->prev = current->prev;
        }

        current->prev->next = current->next;
        current->next = obj->head;
        current->prev = NULL;
        obj->head->prev = current;
        obj->head = current;
    }

    return current->value;
}

// Function to put a key-value pair into the LRUCache
void lRUCachePut(LRUCache* obj, int key, int value) {
    if (obj->cache[key].key != -1) {
        // Key exists, update the value and move the node to the front
        obj->cache[key].value = value;
        lRUCacheGet(obj, key); // Move the node to the front
    } else {
        // Key doesn't exist, create a new node and add it to the front
        Node* newNode = &obj->cache[key];
        newNode->key = key;
        newNode->value = value;
        newNode->next = obj->head;
        newNode->prev = NULL;

        if (obj->head != NULL) {
            obj->head->prev = newNode;
        }

        obj->head = newNode;

        if (obj->tail == NULL) {
```

```
            obj->tail = newNode;
        }

        // If the cache is full, remove the least recently used element
        if (obj->size == obj->capacity) {
            Node* tailPrev = obj->tail->prev;
            obj->cache[obj->tail->key].key = -1; // Mark the entry as unused
            obj->tail = tailPrev;

            if (obj->tail != NULL) {
                obj->tail->next = NULL;
            }
        } else {
            obj->size++;
        }
    }
}

// Function to free the memory used by the LRUCache
void lRUCacheFree(LRUCache* obj) {
    // No dynamic memory to free in this implementation
    free(obj);
}
```

1.3   Inorder Successor of Binary Search Tree

Given the root of a binary search tree and a node p in it, return *the in-order successor of that node in the BST*. If the given node has no in-order successor in the tree, return null.

The successor of a node p is the node with the smallest key greater than p.val.

Example 1:
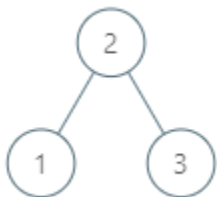
Input: root = [2,1,3], p = 1
Output: 2
Explanation: 1's in-order successor node is 2. Note that both p and the return value is of TreeNode type.

Example 2:
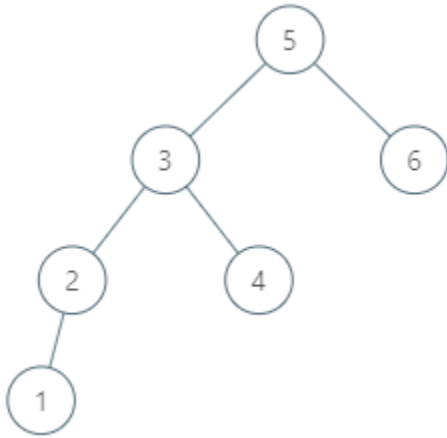
Input: root = [5,3,6,2,4,null,null,1], p = 6
Output: null
Explanation: There is no in-order successor of the current node, so the answer is null.

C:

```c
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */
struct TreeNode* inorderSuccessor(struct TreeNode* root, struct TreeNode* p) {
    struct TreeNode *successor = NULL;
    while(root != NULL) {
        if (p->val < root->val) {
            successor = root;
            root = root->left;
        } else {
            root = root->right;
        }
    }
    return successor;
}
```

1.4   Merge Intervals

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

Example 1:

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

Example 2:

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

C :

```c
/**
 * Return an array of arrays of size *returnSize.
 * The sizes of the arrays are returned as *returnColumnSizes array.
 * Note: Both returned array and *columnSizes array must be malloced, assume caller
calls free().
 */
// Definition for an interval.
typedef struct {
    int start;
    int end;
} Interval;

int compareIntervals(const void* a, const void* b) {
    return ((Interval*)a)->start - ((Interval*)b)->start;
}

int** merge(int** intervals, int intervalsSize, int* intervalsColSize, int* returnSize,
int** returnColumnSizes) {
    if (intervalsSize <= 0) {
        *returnSize = 0;
        *returnColumnSizes = NULL;
        return NULL;
    }

    // Convert the input to Interval structures
    Interval* intervalsStruct = (Interval*)malloc(intervalsSize * sizeof(Interval));
    for (int i = 0; i < intervalsSize; i++) {
        intervalsStruct[i].start = intervals[i][0];
        intervalsStruct[i].end = intervals[i][1];
    }

    // Sort intervals based on the start time
    qsort(intervalsStruct, intervalsSize, sizeof(Interval), compareIntervals);

    // Initialize result array
    int** result = (int**)malloc(intervalsSize * sizeof(int*));
    *returnSize = 0;
    *returnColumnSizes = (int*)malloc(intervalsSize * sizeof(int));
```

```c
        // Merge overlapping intervals
        result[*returnSize] = (int*)malloc(2 * sizeof(int));
        result[*returnSize][0] = intervalsStruct[0].start;
        result[*returnSize][1] = intervalsStruct[0].end;

        for (int i = 1; i < intervalsSize; i++) {
            if (result[*returnSize][1] >= intervalsStruct[i].start) {
                // Merge overlapping intervals
                result[*returnSize][1] = (result[*returnSize][1] > intervalsStruct[i].end) ?
result[*returnSize][1] : intervalsStruct[i].end;
            } else {
                // Move to the next non-overlapping interval
                (*returnSize)++;
                result[*returnSize] = (int*)malloc(2 * sizeof(int));
                result[*returnSize][0] = intervalsStruct[i].start;
                result[*returnSize][1] = intervalsStruct[i].end;
            }
        }

        (*returnSize)++;
        *returnColumnSizes = (int*)realloc(*returnColumnSizes, *returnSize * sizeof(int));
        for (int i = 0; i < *returnSize; i++) {
            (*returnColumnSizes)[i] = 2;
        }

        // Clean up
        free(intervalsStruct);

        return result;
}
```

1.5   Merge Sorted Array

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 result can fit in nums1.

C :

```c
void merge(int* nums1, int nums1Size, int m, int* nums2, int nums2Size, int n){
    int i = m - 1;
    int j = n - 1;
    int k = m + n - 1;

    while (i >= 0 && j >= 0) {
        if (nums1[i] < nums2[j]) {
            nums1[k--] = nums2[j--];
        } else {
            nums1[k--] = nums1[i--];
        }
    }

    // If there are remaining elements in nums2, copy them to nums1
    while (i >= 0) {
        nums1[k--] = nums1[i--];
    }
    while (j >= 0) {
        nums1[k--] = nums2[j--];
    }
}
```

1.6   Serialize and Deserialize Binary Tree

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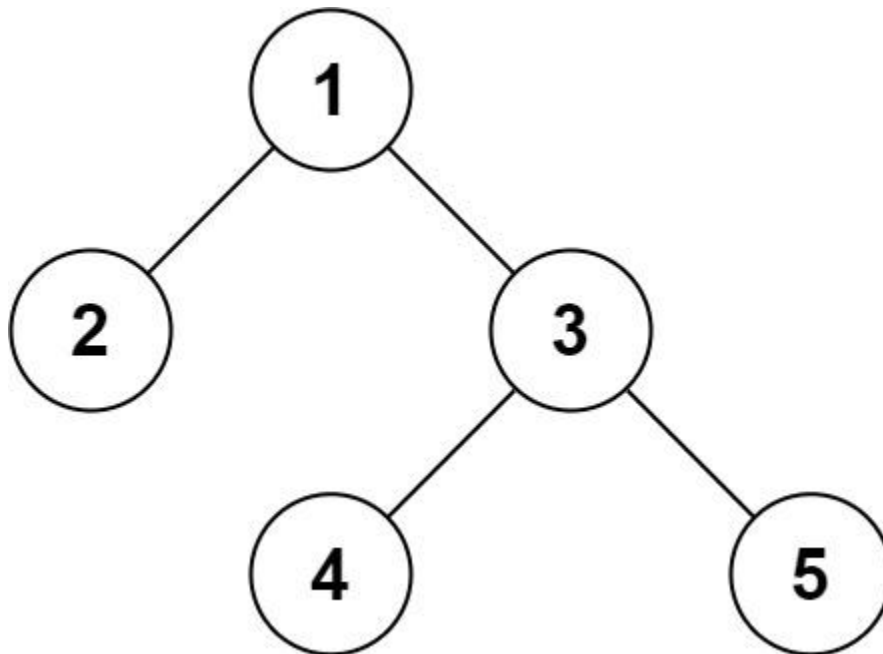
Input: root = [1,2,3,null,null,4,5]
Output: [1,2,3,null,null,4,5]

Example 2:

Input: root = []
Output: []

C :

```c
// Serialize helper function
void serializeHelper(struct TreeNode* root, char* result, int* index) {
    if (root == NULL) {
        // Append "null," to the result string
        sprintf(result + *index, "null,");
        (*index) += strlen("null,");
    } else {
```

```c
        // Convert the node value to a string and append it to the result
        sprintf(result + *index, "%d,", root->val);
        (*index) += strlen(result + *index);

        // Recursively serialize the left and right subtrees
        serializeHelper(root->left, result, index);
        serializeHelper(root->right, result, index);
    }
}

// Deserialize helper function
struct TreeNode* deserializeHelper(char* data, int* index) {
    if (data[*index] == '\0' || strncmp(data + *index, "null", 4) == 0) {
        // End of string or "null" node
        *index += 5;  // Move index to the next position after "null,"
        return NULL;
    }

    // Extract the node value from the data string
    int val;
    sscanf(data + *index, "%d,", &val);
    (*index) += numDigits(val) + 1;  // Move index to the next position after the comma

    // Create a new tree node with the extracted value
    struct TreeNode* newNode = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    newNode->val = val;

    // Recursively deserialize the left and right subtrees
    newNode->left = deserializeHelper(data, index);
    newNode->right = deserializeHelper(data, index);

    return newNode;
}

// Serialize function
char* serialize(struct TreeNode* root) {
    // Allocate memory for the result string
    char* result = (char*)malloc(1024 * sizeof(char));
    int index = 0;

    // Call the serializeHelper to populate the result string
    serializeHelper(root, result, &index);

    // Null-terminate the result string
    result[index] = '\0';

    // Resize the result buffer to the actual size needed
    result = (char*)realloc(result, (index + 1) * sizeof(char));
```

```
        return result;
}

// Deserialize function
struct TreeNode* deserialize(char* data) {
    int index = 0;
    return deserializeHelper(data, &index);
}

// Utility function to calculate the number of digits in an integer
int numDigits(int num) {
    int count = 0;
    while (num != 0) {
        num /= 10;
        count++;
    }
    return count;
}
```

1.7   Remove All Adjacent Duplicates In String

You are given a string s and an integer k, a k **duplicate removal** consists of choosing k adjacent and equal letters from s and removing them, causing the left and the right side of the deleted substring to concatenate together.

We repeatedly make k **duplicate removals** on s until we no longer can.

Return *the final string after all such duplicate removals have been made*. It is guaranteed that the answer is **unique**.

Example 1:

Input: s = "abcd", k = 2
Output: "abcd"
Explanation: There's nothing to delete.

Example 2:

Input: s = "deeedbbcccbdaa", k = 3
Output: "aa"
Explanation:
First delete "eee" and "ccc", get "ddbbbdaa"
Then delete "bbb", get "dddaa"
Finally delete "ddd", get "aa"

Example 3:

Input: s = "pbbcggttciiippooaais", k = 2

C :
```c
struct Stack {
    int *arr;
    int top;
    int capacity;
};

// Function to initialize a stack
struct Stack* createStack(int capacity) {
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->arr = (int*)malloc(stack->capacity * sizeof(int));
    return stack;
}

// Function to check if the stack is empty
int isEmpty(struct Stack* stack) {
    return stack->top == -1;
}

// Function to push an element onto the stack
void push(struct Stack* stack, int item) {
    stack->arr[++stack->top] = item;
}

// Function to pop an element from the stack
int pop(struct Stack* stack) {
    if (isEmpty(stack)) {
        return -1; // Stack underflow
    }
    return stack->arr[stack->top--];
}

// Function to remove duplicates in a string
char* removeDuplicates(char* s, int k) {
    int length = strlen(s);
    struct Stack* counts = createStack(length);

    for (int i = 0; i < length; ++i) {
        if (i == 0 || s[i] != s[i - 1]) {
            push(counts, 1);
        } else {
            int incremented = pop(counts) + 1;
            if (incremented == k) {
                memmove(s + i - k + 1, s + i + 1, length - i);
                i = i - k;
                length = length - k;
```

```
        } else {
            push(counts, incremented);
        }
    }
}

    return s;
}
```

1.8   Number Of Islands

Given an m x 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","1","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

C :

```c
void dfs(char** grid, int gridSize, int gridColSize, int crow, int ccol) {
    if (crow < 0 || crow >= gridSize || ccol < 0 || ccol >= gridColSize ||
grid[crow][ccol] == '0') {
        return;
    }
    grid[crow][ccol] = '0';
    dfs(grid,gridSize, gridColSize,crow-1,ccol);
    dfs(grid,gridSize, gridColSize,crow+1,ccol);
    dfs(grid,gridSize, gridColSize,crow,ccol+1);
    dfs(grid,gridSize, gridColSize,crow,ccol-1);
```

```c
}
int numIslands(char** grid, int gridSize, int* gridColSize) {
    if (grid == NULL || gridSize == 0 || gridColSize == NULL || *gridColSize == 0) {
        return 0;
    }
    int num_of_islands = 0;
    for (int i = 0; i < gridSize; i++) {
        for (int j = 0; j < *gridColSize; j++) {
            if (grid[i][j] == '1') {
                num_of_islands++;
                dfs(grid, gridSize, *gridColSize, i, j);
            }
        }
    }
    return num_of_islands;
}
```

1.9   Copy List With Random Pointer

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

Construct a **deep copy** of the list. The deep copy should consist of exactly n **brand new** nodes, where each new node has its value set to the value of its corresponding original node. Both the next and random pointer of the new nodes should point to new nodes in the copied list such that the pointers in the original list and copied list represent the same list state. **None of the pointers in the new list should point to nodes in the original list**.

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

Return *the head of the copied linked list*.

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

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

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

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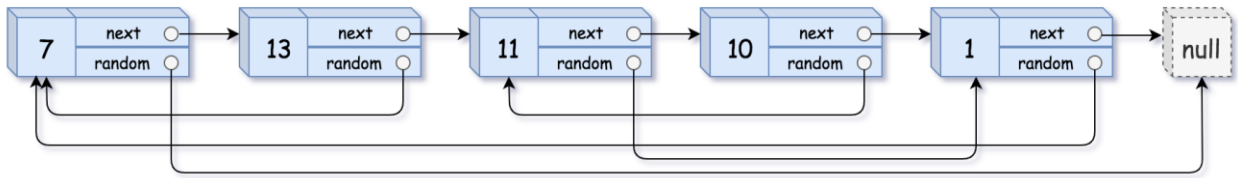
Input: head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
Output: [[7,null],[13,0],[11,4],[10,2],[1,0]]

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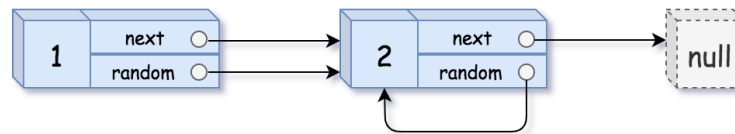
Input: head = [[1,1],[2,1]]
Output: [[1,1],[2,1]]

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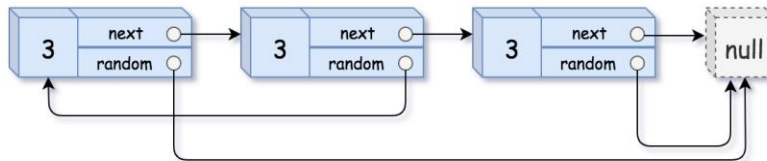
Input: head = [[3,null],[3,0],[3,null]]
Output: [[3,null],[3,0],[3,null]]

C :

```c
/**
 * Definition for a Node.
 * struct Node {
 *     int val;
 *     struct Node *next;
 *     struct Node *random;
 * };
 */



struct Node* copyRandomList(struct Node* head) {
    if (head == NULL) {
        return NULL;
```

```c
    }
    struct Node *curr = head;
    while (curr != NULL) {
        struct Node* new = (struct Node*) malloc (sizeof(struct Node));
        new->val = curr->val;
        new->next = curr->next;
        new->random = NULL;
        curr->next = new;
        curr = new->next;
    }
    curr = head;
    while (curr != NULL) {
        if (curr->random != NULL) {
            curr->next->random = curr->random->next;
        }
        curr = curr->next->next;
    }
    curr= head;
    struct Node* new_head = head->next;
    struct Node *new_curr = new_head;
    while (curr != NULL) {
        curr->next = new_curr->next;
        curr = curr->next;
        if (curr != NULL) {
            new_curr->next = curr->next;
            new_curr = new_curr->next;
        }
    }
    return new_head;
}
```

1.10 Two Sum

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].

C :

```c
typedef struct {
    int key;
    int value;
} hashEntry_t;

typedef struct {
    hashEntry_t *table;
    int capacity;
} hashTable_t;

void init(hashTable_t *t, int capacity) {
    t->table = (hashEntry_t *)malloc(sizeof(hashEntry_t) * capacity);
    t->capacity = capacity;
}
bool contains(hashTable_t *t, int complement, int *index) {
    for (int i = 0; i < t->capacity; i++) {
        if (t->table[i].key == complement) {
            *index = t->table[i].value;
            return true;
        }
    }
    return false;
}

int insert(hashTable_t *t, int number, int number_index, int target) {
    int index = 0;
    int complement = target - number;
    if (contains(t, complement, &index)) {
        return index;
    } else {

        t->table[number_index].key = number;
        t->table[number_index].value = number_index;
    }
    return -1;
}
```

```c
int* twoSum(int* nums, int numsSize, int target, int* returnSize) {
    hashTable_t t;
    init(&t, numsSize);
    int *result = (int *)malloc(sizeof(int) * 2);

    for (int i = 0; i < numsSize; i++) {
        int j = insert(&t, nums[i], i, target);
        if (j != -1) {
            result[0] = i;
            result[1] = j;
            *returnSize = 2;
            return result;
        }
    }

    free(t.table);
    return NULL;
}
```

1.11 Flood Fill

An image is represented by an m x n integer grid image where image[i][j] represents the pixel value of the image.

You are also given three integers sr, sc, and color. You should perform a **flood fill** on the image starting from the pixel image[sr][sc].

To perform a **flood fill**, consider the starting pixel, plus any pixels connected **4-directionally** to the starting pixel of the same color as the starting pixel, plus any pixels connected **4-directionally** to those pixels (also with the same color), and so on. Replace the color of all of the aforementioned pixels with color.

Return *the modified image after performing the flood fill*.

Example 1:
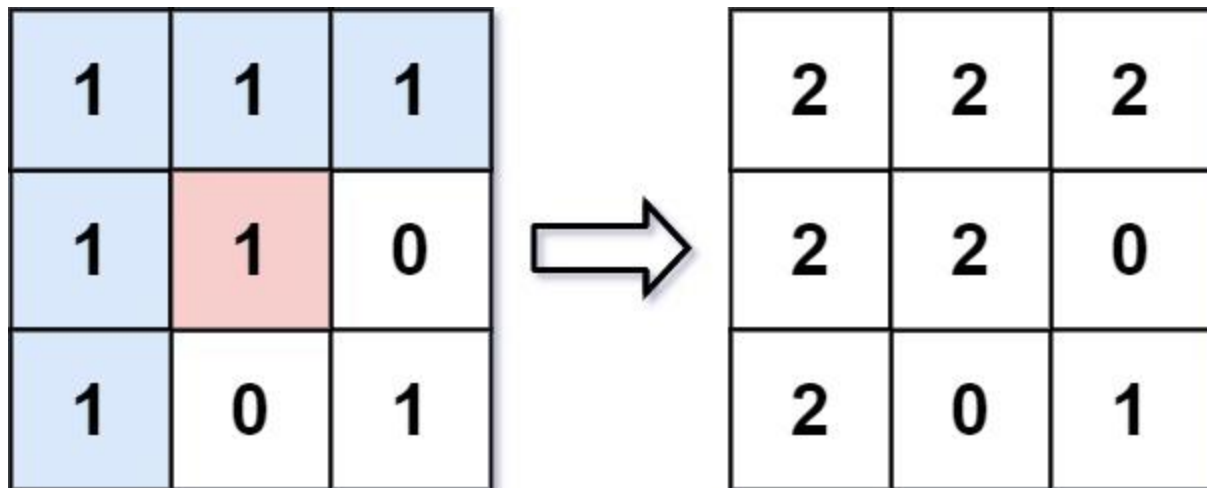
Input: image = [[1,1,1],[1,1,0],[1,0,1]], sr = 1, sc = 1, color = 2
Output: [[2,2,2],[2,2,0],[2,0,1]]
Explanation: From the center of the image with position (sr, sc) = (1, 1) (i.e., the red pixel), all pixels connected by a path of the same color as the starting pixel (i.e., the blue pixels) are colored with the new color.
Note the bottom corner is not colored 2, because it is not 4-directionally 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 0, so no changes are made to the image.

C:

```c
/**
 * Return an array of arrays of size *returnSize.
 * The sizes of the arrays are returned as *returnColumnSizes array.
 * Note: Both returned array and *columnSizes array must be malloced, assume caller
calls free().
 */
void dfs(int** image, int imageSize, int* imageColSize, int sr, int sc, int
originalColor, int newColor) {
    // Base case: check if the current pixel is out of bounds or already colored with
the new color
    if (sr < 0 || sr >= imageSize || sc < 0 || sc >= imageColSize[sr] || image[sr][sc]
!= originalColor) {
        return;
    }

    // Change the color of the current pixel
    image[sr][sc] = newColor;

    // Recursive calls for the neighboring pixels
```

```c
    dfs(image, imageSize, imageColSize, sr - 1, sc, originalColor, newColor); // Up
    dfs(image, imageSize, imageColSize, sr + 1, sc, originalColor, newColor); // Down
    dfs(image, imageSize, imageColSize, sr, sc - 1, originalColor, newColor); // Left
    dfs(image, imageSize, imageColSize, sr, sc + 1, originalColor, newColor); // Right
}

int** floodFill(int** image, int imageSize, int* imageColSize, int sr, int sc, int
color, int* returnSize, int** returnColumnSizes) {
    if (image == NULL || imageSize == 0 || imageColSize == NULL) {
        *returnSize = 0;
        return NULL;
    }

    int originalColor = image[sr][sc];

    // Check if the fill color is the same as the original color
    if (originalColor == color) {
        *returnSize = imageSize;
        *returnColumnSizes = imageColSize;
        return image;
    }

    // Allocate memory for the result image
    int** result = (int**)malloc(imageSize * sizeof(int*));
    *returnColumnSizes = (int*)malloc(imageSize * sizeof(int));

    for (int i = 0; i < imageSize; i++) {
        result[i] = (int*)malloc(imageColSize[i] * sizeof(int));
        (*returnColumnSizes)[i] = imageColSize[i];
    }

    // Copy the original image to the result image
    for (int i = 0; i < imageSize; i++) {
        for (int j = 0; j < imageColSize[i]; j++) {
            result[i][j] = image[i][j];
        }
    }

    // Apply the flood fill algorithm
    dfs(result, imageSize, *returnColumnSizes, sr, sc, originalColor, color);

    *returnSize = imageSize;
    return result;
}
```

1.12 Maximum SubArray

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.

C :

```c
int maxSubArray(int* nums, int numsSize) {
        if (numsSize == 0) {
        // Empty array case
        return 0;
    }

    int maxSum = nums[0];  // Initialize maxSum with the first element
    int currentSum = nums[0];  // Initialize currentSum with the first element

    for (int i = 1; i < numsSize; i++) {
        // Update currentSum to either continue the current subarray or start a new one
        currentSum = (currentSum > 0) ? currentSum + nums[i] : nums[i];

        // Update maxSum if the current subarray sum is greater
        if (currentSum > maxSum) {
            maxSum = currentSum;
        }
    }

    return maxSum;
}
```

1.13 Sort Colors

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]

C :

```c
void sortColors(int* nums, int numsSize) {
    int low = 0;            // Index for 0
    int high = numsSize - 1;  // Index for 2

    for (int i = 0; i <= high;) {
        if (nums[i] == 0) {
            // Swap current element with the one at the low index
            int temp = nums[i];
            nums[i] = nums[low];
            nums[low] = temp;

            // Move low and i to the right
            low++;
            i++;
        } else if (nums[i] == 2) {
            // Swap current element with the one at the high index
            int temp = nums[i];
            nums[i] = nums[high];
            nums[high] = temp;

            // Move high to the left
            high--;
        } else {
            // Element is 1, move i to the right
            i++;
        }
    }
}
```

1.14   Median Of Two Sorted Arrays

Given two sorted arrays nums1 and nums2 of size m and n respectively, return **the median** of the two sorted arrays.

The overall run time complexity should be O(log (m+n)).

Input: nums1 = [1,3], nums2 = [2]
Output: 2.00000
Explanation: merged array = [1,2,3] and median is 2.

Input: nums1 = [1,2], nums2 = [3,4]
Output: 2.50000
Explanation: merged array = [1,2,3,4] and median is (2 + 3) / 2 = 2.5.

C :

```c
double findMedianSortedArrays(int* nums1, int nums1Size, int* nums2, int nums2Size) {
    int totalSize = nums1Size + nums2Size;
    int merged[totalSize];

    // Merge the two sorted arrays
    int i = 0, j = 0, k = 0;
    while (i < nums1Size && j < nums2Size) {
        if (nums1[i] < nums2[j]) {
            merged[k++] = nums1[i++];
        } else {
            merged[k++] = nums2[j++];
        }
    }

    // Add remaining elements from both arrays
    while (i < nums1Size) {
        merged[k++] = nums1[i++];
    }

    while (j < nums2Size) {
        merged[k++] = nums2[j++];
    }

    // Calculate the median
    if (totalSize % 2 == 0) {
        return (double)(merged[totalSize / 2 - 1] + merged[totalSize / 2]) / 2.0;
    } else {
        return (double)merged[totalSize / 2];
    }
}
```

1.15 Generate Parantheses

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: ["()"]

C :

```c
/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
void backtracking(char*** result, int* returnSize, char* curString, int leftCount, int rightCount, int n, int index) {
    if (index == 2 * n) {
        curString[index] = '\0';
        (*result)[*returnSize] = strdup(curString);
        (*returnSize)++;
        return;
    }

    if (leftCount < n) {
        curString[index] = '(';
        backtracking(result, returnSize, curString, leftCount + 1, rightCount, n, index + 1);
    }

    if (leftCount > rightCount) {
        curString[index] = ')';
        backtracking(result, returnSize, curString, leftCount, rightCount + 1, n, index + 1);
    }
}

char** generateParenthesis(int n, int* returnSize) {
    char** result = (char**)malloc(1000 * sizeof(char*));
    *returnSize = 0;

    char* curString = (char*)malloc((2 * n + 1) * sizeof(char));
    curString[0] = '\0';

    backtracking(&result, returnSize, curString, 0, 0, n, 0);

    free(curString);
```

```
        return result;
}
```

1.16   Meeting Rooms II – Minimum Rooms Required

Given an array of meeting time intervals intervals where intervals[i] = [start₁, end₁], return *the minimum number of conference rooms required*.

Example 1:

Input: intervals = [[0,30],[5,10],[15,20]]
Output: 2

Example 2:

Input: intervals = [[7,10],[2,4]]
Output: 1

C :
```c
int compare(const void* a, const void* b) {
    return ((int**)a)[0][0] - ((int**)b)[0][0];
}
int minMeetingRooms(int** intervals, int intervalsSize, int* intervalsColSize) {
    if (intervals == NULL || intervalsSize == 0) {
        return 0;
    }

    // Sort the intervals based on start times
    qsort(intervals, intervalsSize, sizeof(int*), compare);

    // Priority queue to store end times of meetings
    int* endTimes = (int*)malloc(sizeof(int) * intervalsSize);
    int rooms = 0;

    // Iterate through the sorted intervals
    for (int i = 0; i < intervalsSize; i++) {
        int start = intervals[i][0];
        int end = intervals[i][1];

        // Check if there is an available room
        int j;
        for (j = 0; j < rooms; j++) {
            if (endTimes[j] <= start) {
                endTimes[j] = end;
                break;
```

```
            }
        }

        // If no available room, allocate a new room
        if (j == rooms) {
            endTimes[rooms++] = end;
        }
    }

    free(endTimes);

    return rooms;
}
```

1.17   Reverse Linked List

Given the head of a singly linked list, reverse the list, and return *the reversed list*.
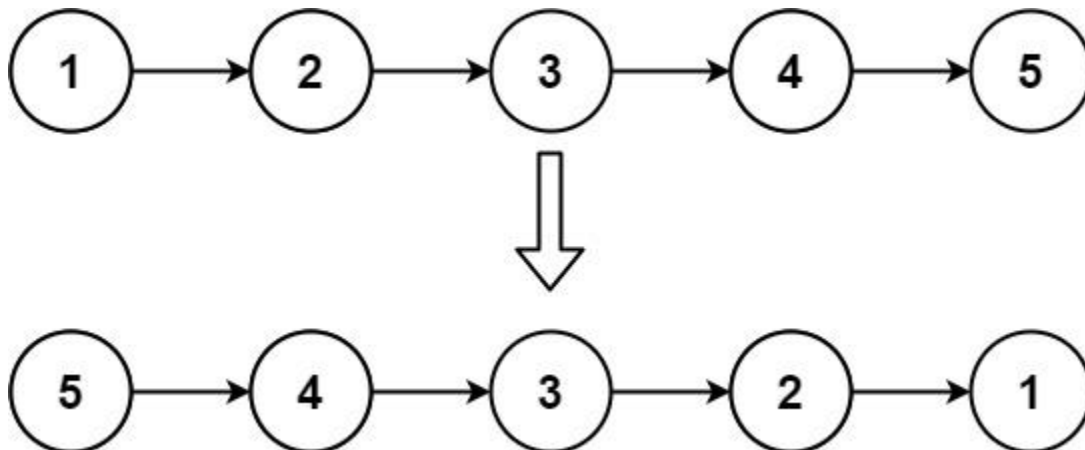
Example 1:

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

Example 2:
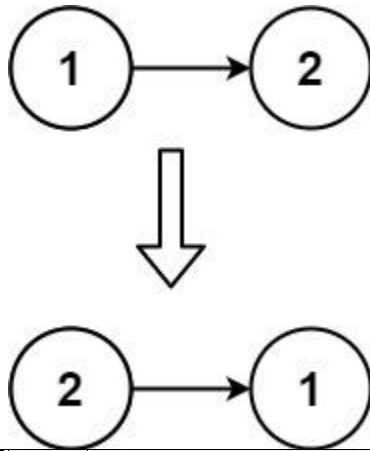
| **Input:** head = [1,2] |
| **Output:** [2,1] |

| **Example 3:** |

| **Input:** head = [] |
| **Output:** [] |

C :

```c
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
struct ListNode* reverseList(struct ListNode* head){
    struct ListNode *prev = NULL;
    struct ListNode *curr = head;
    struct ListNode *next = NULL;
    while (curr != NULL) {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}
```

1.18   Implement Queue Using Stack

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

C:

```c
typedef struct {
    int* stack1;  // Stack for enqueue operation (push)
    int* stack2;  // Stack for dequeue operation (pop, peek)
    int top1;     // Top index for stack1
    int top2;     // Top index for stack2
    int capacity; // Capacity of the stacks
} MyQueue;

MyQueue* myQueueCreate() {
    MyQueue* queue = (MyQueue*)malloc(sizeof(MyQueue));
    queue->capacity = 100; // You can adjust the capacity based on your needs
    queue->stack1 = (int*)malloc(sizeof(int) * queue->capacity);
    queue->stack2 = (int*)malloc(sizeof(int) * queue->capacity);
    queue->top1 = -1;
    queue->top2 = -1;
    return queue;
```

```c
}

void enqueue(MyQueue* obj, int x) {
    if (obj->top1 == obj->capacity - 1) {
        // Resize stack1 if needed
        obj->capacity *= 2;
        obj->stack1 = realloc(obj->stack1, sizeof(int) * obj->capacity);
    }
    obj->stack1[++(obj->top1)] = x;
}

void transferStacks(MyQueue* obj) {
    // Transfer elements from stack1 to stack2
    while (obj->top1 != -1) {
        obj->stack2[++(obj->top2)] = obj->stack1[(obj->top1)--];
    }
}

void myQueuePush(MyQueue* obj, int x) {
    enqueue(obj, x);
}

int myQueuePop(MyQueue* obj) {
    if (obj->top2 == -1) {
        // If stack2 is empty, transfer elements from stack1
        transferStacks(obj);
    }

    if (obj->top2 == -1) {
        return -1; // Queue is empty
    }

    return obj->stack2[(obj->top2)--];
}

int myQueuePeek(MyQueue* obj) {
    if (obj->top2 == -1) {
        // If stack2 is empty, transfer elements from stack1
        transferStacks(obj);
    }

    if (obj->top2 == -1) {
        return -1; // Queue is empty
    }

    return obj->stack2[obj->top2];
}

bool myQueueEmpty(MyQueue* obj) {
```

```
    return (obj->top1 == -1) && (obj->top2 == -1);
}

void myQueueFree(MyQueue* obj) {
    free(obj->stack1);
    free(obj->stack2);
    free(obj);
}
```

1.19 Implement Stack Using Queue

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], [], [], []]
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

C :

```c
typedef struct {
    int* queue;
    int front;
    int rear;
} MyStack;

MyStack* myStackCreate() {
    MyStack* obj = (MyStack*)malloc(sizeof(MyStack));
    obj->queue = (int*)malloc(1000 * sizeof(int));  // Adjust the size according to your
requirements
    obj->front = obj->rear = -1;
    return obj;
}

void enqueue(int* queue, int* rear, int x) {
    queue[++(*rear)] = x;
}

int dequeue(int* queue, int* front) {
    return queue[(*front)++];
}

void myStackPush(MyStack* obj, int x) {
    enqueue(obj->queue, &(obj->rear), x);

    // Rotate the queue to make the recently pushed element at the front
    int size = obj->rear - obj->front + 1;
    for (int i = 0; i < size - 1; ++i) {
        enqueue(obj->queue, &(obj->rear), dequeue(obj->queue, &(obj->front)));
    }
}

int myStackPop(MyStack* obj) {
    if (obj == NULL || obj->front > obj->rear) {
        // Stack is empty
        return -1;
    }

    return dequeue(obj->queue, &(obj->front));
}

int myStackTop(MyStack* obj) {
    if (obj == NULL || obj->front > obj->rear) {
        // Stack is empty
        return -1;
    }

    return obj->queue[obj->front];
}
```

```c
bool myStackEmpty(MyStack* obj) {
    return obj == NULL || obj->front > obj->rear;
}

void myStackFree(MyStack* obj) {
    free(obj->queue);
    free(obj);
}
/**
 * Your MyStack struct will be instantiated and called as such:
 * MyStack* obj = myStackCreate();
 * myStackPush(obj, x);

 * int param_2 = myStackPop(obj);

 * int param_3 = myStackTop(obj);

 * bool param_4 = myStackEmpty(obj);

 * myStackFree(obj);
*/
```

1.20 Trapping Rain Water

Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it can trap after raining.

Example 1:

Input: height = [0,1,0,2,1,0,1,3,2,1,2,1]
Output: 6
Explanation: The above elevation map (black section) is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped.

Example 2:

Input: height = [4,2,0,3,2,5]
Output: 9

C :

```c
int trap(int* height, int heightSize) {
    if (heightSize <= 2) {
        return 0;  // Not enough bars to trap water
    }

    int leftMax[heightSize];
    int rightMax[heightSize];
    int result = 0;

    // Calculate the maximum height to the left of each bar
    leftMax[0] = height[0];
    for (int i = 1; i < heightSize; ++i) {
        leftMax[i] = (height[i] > leftMax[i - 1]) ? height[i] : leftMax[i - 1];
    }

    // Calculate the maximum height to the right of each bar
    rightMax[heightSize - 1] = height[heightSize - 1];
    for (int i = heightSize - 2; i >= 0; --i) {
        rightMax[i] = (height[i] > rightMax[i + 1]) ? height[i] : rightMax[i + 1];
    }

    // Calculate the trapped water at each bar
    for (int i = 1; i < heightSize - 1; ++i) {
        int minHeight = (leftMax[i] < rightMax[i]) ? leftMax[i] : rightMax[i];
        result += (minHeight - height[i] > 0) ? minHeight - height[i] : 0;
    }

    return result;
}
```

1.21   Longest SubString Without Repeating Characters

Given a string s, find the length of the **longest**

substring
 without repeating characters.

Example 1:

Input: s = "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the length of 3.

C:

```c
int lengthOfLongestSubstring(char* s) {
    int n = strlen(s);
    if (n == 0) {
        return 0;
    }

    int* charIndex = (int*)malloc(sizeof(int) * 128);
    for (int i = 0; i < 128; i++) {
        charIndex[i] = -1;
    }

    int maxLength = 0;
    int start = 0;

    for (int end = 0; end < n; end++) {
        if (charIndex[s[end]] != -1) {
            start = fmax(charIndex[s[end]] + 1, start);
        }

        maxLength = fmax(maxLength, end - start + 1);
        charIndex[s[end]] = end;
    }

    free(charIndex);
    return maxLength;
}
class Solution {
    public int lengthOfLongestSubstring(String s) {
        if (s.length() == 0) return 0;
        else if (s.length() == 1) return 1;
        else {
            HashSet <Character> hs = new HashSet<Character>();
```

```java
            int max_length = 0;
            for (int k = 0; k < s.length(); k++) {
                hs.add(s.charAt(k));
                for (int i = k+1; i < s.length(); i++) {
                    if (!hs.contains(s.charAt(i))) {
                        hs.add(s.charAt(i));
                        if (i == s.length()-1) {
                            max_length = Math.max(max_length,hs.size());
                            hs.clear();
                        }
                    }
                    else {
                        max_length = Math.max(max_length,hs.size());
                        hs.clear();
                        break;
                    }
                }
            }
            return max_length;
        }
    }
}
```

1.22 SubArray Sum Equals K

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

C :

```c
int subarraySum(int* nums, int numsSize, int k) {
    int count = 0;
    int sum = 0;

    // Create a hashmap to store cumulative sum frequencies
```

```
    int* hashmap = (int*)malloc(sizeof(int) * (numsSize + 1));
    for (int i = 0; i <= numsSize; i++) {
        hashmap[i] = 0;
    }
    hashmap[0] = 1;

    for (int i = 0; i < numsSize; i++) {
        sum += nums[i];

        int diff = sum - k;

        if (hashmap[diff] > 0) {
            count += hashmap[diff];
        }

        hashmap[sum]++;
    }

    free(hashmap);

    // Debugging output
    printf("Count: %d\n", count);
    return count;
}
```

1.23   Search In A Rotated Sorted Array

There is an integer array nums sorted in ascending order (with distinct values).

Prior to being passed to your function, nums is possibly rotated at an unknown pivot index k (1 <= k < 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 rotated at pivot index 3 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

<u>**C :**</u>

```c
int search(int* nums, int numsSize, int target) {
        int n = numsSize;
        int left = 0, right = n - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;

            // Case 1: find target
            if (nums[mid] == target) {
                return mid;
            }

            // Case 2: subarray on mid's left is sorted
            else if (nums[mid] >= nums[left]) {
                if (target >= nums[left] && target < nums[mid]) {
                    right = mid - 1;
                } else {
                    left = mid + 1;
                }
            }

            // Case 3: subarray on mid's right is sorted
            else {
                if (target <= nums[right] && target > nums[mid]) {
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            }
        }
        return -1;
}
```

1.24 Top K Frequent Elements

Given an integer array nums and an integer k, return *the* k *most frequent elements*. You may return the answer in **any order**.

Example 1:

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

Example 2:

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

C :

```c
typedef struct {
    int value;
    int frequency;
} Element;

int compare(const void *a, const void *b) {
    return ((Element *)b)->frequency - ((Element *)a)->frequency;
}

int* topKFrequent(int* nums, int numsSize, int k, int* returnSize) {
    // Initialize a hash table with calloc
    int* hashTable = (int*)calloc(1001, sizeof(int));

    // Count the frequency of each element and store it in the hash table
    for (int i = 0; i < numsSize; ++i) {
        hashTable[nums[i]]++;
    }

    // Populate the elements array with values and frequencies from the hash table
    Element* elements = (Element*)malloc(1001 * sizeof(Element));
    int count = 0;
    for (int i = 0; i <= 1000; ++i) {
        if (hashTable[i] > 0) {
            elements[count++] = (Element){i, hashTable[i]};
        }
    }

    // Sort the elements array based on frequency in descending order
    qsort(elements, count, sizeof(Element), compare);

    // Allocate memory for the result array
    int* result = (int*)malloc(k * sizeof(int));

    // Copy the top k frequent elements to the result array
    for (int i = 0; i < k; ++i) {
        result[i] = elements[i].value;
    }

    // Update the returnSize
    *returnSize = k;
```

```
    // Free allocated memory
    free(elements);
    free(hashTable);

    return result;
}
```
1.25   Rotate Image

You are given an n x 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:
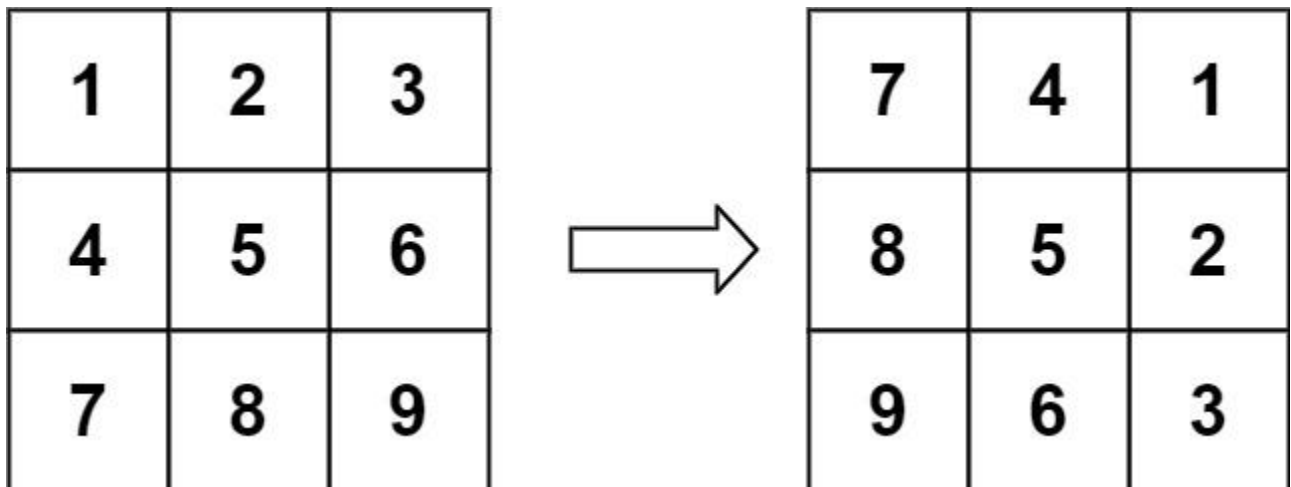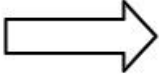
Input: matrix = [[1,2,3],[4,5,6],[7,8,9]]
Output: [[7,4,1],[8,5,2],[9,6,3]]

Example 2:

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]]

C :

```c
void transpose(int** matrix, int matrixSize, int matrixcolSize) {
    for (int i = 0; i < matrixSize; i++) {
        for (int j = i+1; j < matrixcolSize; j++) {
            int temp = matrix[i][j];
            matrix[i][j] = matrix[j][i];
            matrix[j][i] = temp;
        }
    }
}
void reverse(int** matrix, int matrixSize, int matrixcolSize) {
    for (int i = 0; i < matrixSize; i++) {
        for (int j = 0; j < matrixcolSize/2; j++) {
            int temp = matrix[i][j];
            matrix[i][j] = matrix[i][matrixcolSize-j-1];
            matrix[i][matrixcolSize-j-1] = temp;
        }
    }
}
void rotate(int** matrix, int matrixSize, int* matrixColSize){
    transpose(matrix, matrixSize, matrixSize);
    reverse(matrix, matrixSize, matrixSize);
}
```

1.26 Gas Station

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

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

Given two integer arrays gas and cost, return *the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return* -1. If there exists a solution, it is **guaranteed** to be **unique**

Example 1:

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

Example 2:

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

C :

```c
int canCompleteCircuit(int* gas, int gasSize, int* cost, int costSize) {
    int totalGas = 0;
    int totalCost = 0;
    int currentGas = 0;
    int startStation = 0;

    for (int i = 0; i < gasSize; ++i) {
        totalGas += gas[i];
        totalCost += cost[i];
        currentGas += gas[i] - cost[i];

        // If the current sum becomes negative, reset the starting station to the next station
        if (currentGas < 0) {
            currentGas = 0;
            startStation = i + 1;
        }
    }
```

```
    // If the total gas is less than the total cost, there is no solution
    if (totalGas < totalCost) {
        return -1;
    }

    return startStation;
}
```

1.27   Delete Node in BST

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:
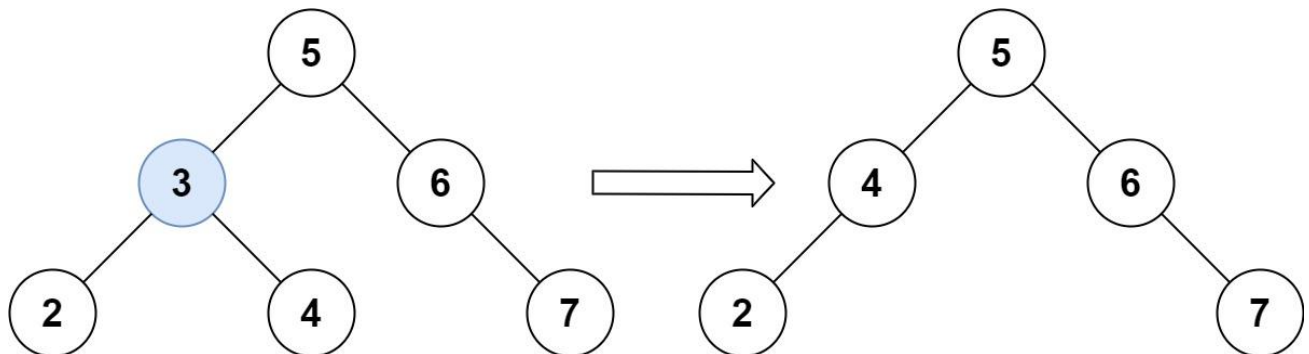
Input: root = [5,3,6,2,4,null,7], key = 3
Output: [5,4,6,2,null,null,7]
Explanation: Given key to delete is 3. So we find the node with value 3 and delete it.
One valid answer is [5,4,6,2,null,null,7], shown in the above BST.
Please notice that another valid answer is [5,2,6,null,4,null,7] and it's also accepted.
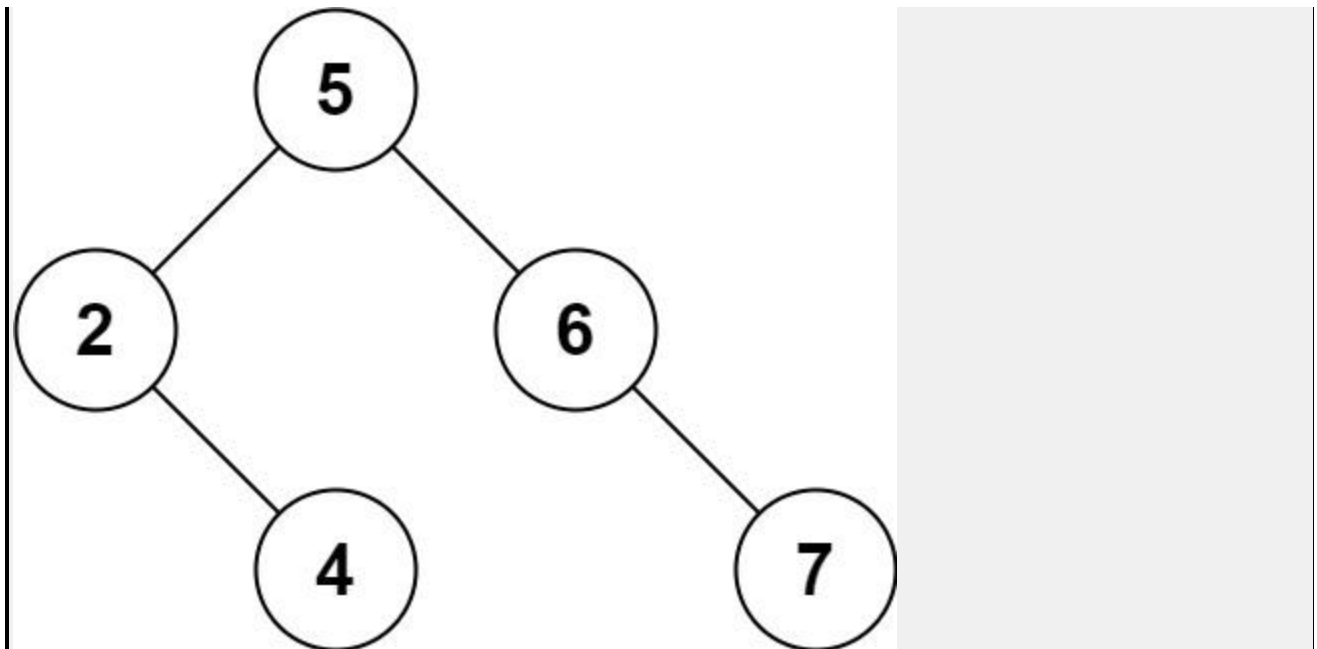
Input: root = [5,3,6,2,4,null,7], key = 0
Output: [5,3,6,2,4,null,7]
Explanation: The tree does not contain a node with value = 0.

Example 3:

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

Constraints:

C :

```c
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */

struct TreeNode* findMin(struct TreeNode* node) {
    while (node->left != NULL) {
        node = node->left;
    }
```

```c
        return node;
}

struct TreeNode* deleteNode(struct TreeNode* root, int key) {
    if (root == NULL) {
        return NULL;
    }

    if (key < root->val) {
        root->left = deleteNode(root->left, key);
    } else if (key > root->val) {
        root->right = deleteNode(root->right, key);
    } else {
        if (root->left == NULL) {
            struct TreeNode* temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct TreeNode* temp = root->left;
            free(root);
            return temp;
        }

        struct TreeNode* temp = findMin(root->right);
        root->val = temp->val;
        root->right = deleteNode(root->right, temp->val);
    }

    return root;
}

// Function to create a new TreeNode
struct TreeNode* createNode(int val) {
    struct TreeNode* newNode = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    newNode->val = val;
    newNode->left = newNode->right = NULL;
    return newNode;
}

// Function to print the inorder traversal of the tree
void inorderTraversal(struct TreeNode* root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->val);
        inorderTraversal(root->right);
    }
}
```

1.28   Word Search

Given an m x 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:

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

Example 2:

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

C :

```c
bool backtrack(char** board, int row, int col, int boardSize, int* boardColSize, char*
word, int index) {
    if (index == strlen(word)) {
        // All characters of the word are found
        return true;
    }

    if (row < 0 || col < 0 || row >= boardSize || col >= boardColSize[row] ||
board[row][col] != word[index]) {
        // Out of bounds or the current cell does not match the current character of the
word
        return false;
    }

    // Mark the current cell as visited
    char original = board[row][col];
    board[row][col] = '\0';

    // Explore neighbors in all four directions
    bool found = (backtrack(board, row - 1, col, boardSize, boardColSize, word, index +
1) ||
                  backtrack(board, row + 1, col, boardSize, boardColSize, word, index +
1) ||
                  backtrack(board, row, col - 1, boardSize, boardColSize, word, index +
1) ||
                  backtrack(board, row, col + 1, boardSize, boardColSize, word, index +
1));

    // Restore the original value of the current cell
    board[row][col] = original;

    return found;
```

```
}

bool exist(char** board, int boardSize, int* boardColSize, char* word) {
    for (int i = 0; i < boardSize; i++) {
        for (int j = 0; j < boardColSize[i]; j++) {
            if (backtrack(board, i, j, boardSize, boardColSize, word, 0)) {
                // If the word is found starting from (i, j)
                return true;
            }
        }
    }

    return false;
}
```

1.29 Integer To Roman

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 one's 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 an integer, convert it to a roman numeral.

Example 1:

Input: num = 3
Output: "III"
Explanation: 3 is represented as 3 ones.

Input: num = 58
Output: "LVIII"
Explanation: L = 50, V = 5, III = 3.

Example 3:

Input: num = 1994
Output: "MCMXCIV"
Explanation: M = 1000, CM = 900, XC = 90 and IV = 4.

C :

```c
char* intToRoman(int num) {
    const char* thousands[] = {"", "M", "MM", "MMM"};
    const char* hundreds[] = {"", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC",
"CM"};
    const char* tens[] = {"", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"};
    const char* ones[] = {"", "I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"};
    char *result = (char *) malloc (sizeof (char) * 16);
    result[0] = '\0';
    strcat(result, thousands[(num / 1000)]);
    strcat(result, hundreds[(num % 1000)/100]);
    strcat(result, tens[(num % 100)/10]);
    strcat(result, ones[(num % 10)]);

    return result;
}
```

1.30 Merge String Alternately

You are given two strings word1 and word2. Merge the strings by adding letters in alternating order, starting with word1. If a string is longer than the other, append the additional letters onto the end of the merged string.

Return *the merged string.*

Example 1:

Input: word1 = "abc", word2 = "pqr"
Output: "apbqcr"
Explanation: The merged string will be merged as so:
word1: a   b   c
word2:   p   q   r
merged: a p b q c r

Example 2:

Input: word1 = "ab", word2 = "pqrs"
Output: "apbqrs"
Explanation: Notice that as word2 is longer, "rs" is appended to the end.
word1: a   b
word2:   p q r s
merged: a p b q   r s

Example 3:

Input: word1 = "abcd", word2 = "pq"
Output: "apbqcd"
Explanation: Notice that as word1 is longer, "cd" is appended to the end.
word1: a   b   c   d
word2:   p q
merged: a p b q c   d

C :

```c
char * mergeAlternately(char * word1, char * word2){
    int i = 0;
    int j = 0;
    int k = 0;
    int len1 = strlen(word1);
    int len2 = strlen(word2);
    int len = len1 + len2;
    char *result = (char*)malloc(sizeof(char)*(len+1));
    memset(result, 0, sizeof(char) * (len + 1));
    while (i < len1 || j < len2) {
        if (i < len1) result[k++] = word1[i++];
        if (j < len2) result[k++] = word2[j++];
    }
    return result;
}
```

1.31   Reverse Linked List II

Given the head of a singly linked list and two integers left and right where left <= right, reverse the nodes of the list from position left to position right, and return *the reversed list*.
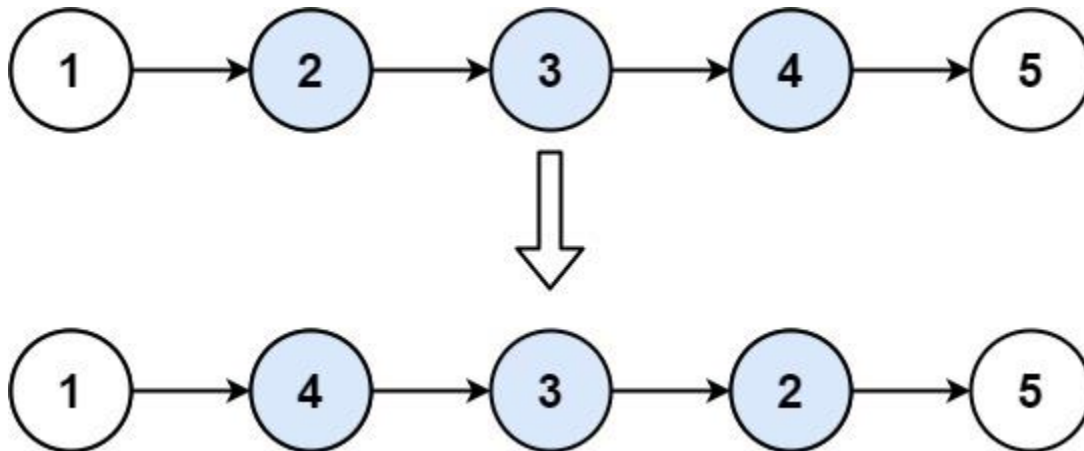
Example 1:

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

Example 2:

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

C :

```c
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
struct ListNode* reverseBetween(struct ListNode* head, int left, int right) {

    if (head == NULL || left == right) {
        return head;
    }

    struct ListNode* dummy = (struct ListNode *)malloc(sizeof(struct ListNode));
    dummy->val = 0;
    dummy->next = head;
    struct ListNode* prev = dummy;

    // Move to the node before the left-th node
    for (int i = 1; i < left; ++i) {
        prev = prev->next;
    }

    struct ListNode* current = prev->next;
    struct ListNode* next = NULL;
```

```
    // Reverse the subsegment from left to right
    for (int i = left; i <= right; ++i) {
        struct ListNode* temp = current->next;
        current->next = next;
        next = current;
        current = temp;
    }

    // Connect the reversed subsegment back to the list
    prev->next->next = current;
    prev->next = next;

    return dummy->next;
}
```

1.32   Implement Trie – Prefix Tree

A **trie** (pronounced as "try") or **prefix tree** is a tree data structure used to efficiently store and retrieve keys in a dataset of strings. There are various applications of this data structure, such as autocomplete and spellchecker.

Implement the Trie class:

- Trie() Initializes the trie object.
- void insert(String word) Inserts the string word into the trie.
- boolean search(String word) Returns true if the string word is in the trie (i.e., was inserted before), and false otherwise.
- boolean startsWith(String prefix) Returns true if there is a previously inserted string word that has the prefix prefix, and false otherwise.

Example 1:

Input
["Trie", "insert", "search", "search", "startsWith", "insert", "search"]
[[], ["apple"], ["apple"], ["app"], ["app"], ["app"], ["app"]]
Output
[null, null, true, false, true, null, true]

Explanation
Trie trie = new Trie();
trie.insert("apple");
trie.search("apple");   // return True
trie.search("app");   // return False
trie.startsWith("app"); // return True
trie.insert("app");
trie.search("app");   // return True

C :

```c
// Define TrieNode structure
typedef struct TrieNode {
    struct TrieNode* children[26];
    bool isEndOfWord;
} TrieNode;

// Define Trie structure
typedef struct {
    TrieNode* root;
} Trie;

// Function to create a new TrieNode
TrieNode* createNode() {
    TrieNode* newNode = (TrieNode*)malloc(sizeof(TrieNode));
    for (int i = 0; i < 26; i++) {
        newNode->children[i] = NULL;
    }
    newNode->isEndOfWord = false;
    return newNode;
}

// Initialize the Trie
Trie* trieCreate() {
    Trie* trie = (Trie*)malloc(sizeof(Trie));
    trie->root = createNode();
    return trie;
}

// Insert a word into the Trie
void trieInsert(Trie* obj, char* word) {
    TrieNode* node = obj->root;
    while (*word) {
        int index = *word - 'a';
        if (node->children[index] == NULL) {
            node->children[index] = createNode();
        }
        node = node->children[index];
        word++;
    }
    node->isEndOfWord = true;
}

// Search for a word in the Trie
```

```c
bool trieSearch(Trie* obj, char* word) {
    TrieNode* node = obj->root;
    while (*word) {
        int index = *word - 'a';
        if (node->children[index] == NULL) {
            return false;
        }
        node = node->children[index];
        word++;
    }
    return node->isEndOfWord;
}

// Check if a prefix exists in the Trie
bool trieStartsWith(Trie* obj, char* prefix) {
    TrieNode* node = obj->root;
    while (*prefix) {
        int index = *prefix - 'a';
        if (node->children[index] == NULL) {
            return false;
        }
        node = node->children[index];
        prefix++;
    }
    return true;
}

// Free the Trie memory
void trieFree(Trie* obj) {
    // Use recursive helper function to free Trie nodes
    void freeNode(TrieNode* node) {
        if (node == NULL) {
            return;
        }
        for (int i = 0; i < 26; i++) {
            freeNode(node->children[i]);
        }
        free(node);
    }

    freeNode(obj->root);
    free(obj);
}

/**
 * Your Trie struct will be instantiated and called as such:
 * Trie* obj = trieCreate();
 * trieInsert(obj, word);
```

```
 * bool param_2 = trieSearch(obj, word);

 * bool param_3 = trieStartsWith(obj, prefix);

 * trieFree(obj);
*/
```

1.33   Odd Even Linked List

Given the head of a singly linked list, group all the nodes with odd indices together followed by the nodes with even indices, and return the reordered list.

The **first** node is considered **odd**, and the **second** node is **even**, and so on.

Note that the relative order inside both the even and odd groups should remain as it was in the input.

You must solve the problem in O(1) extra space complexity and O(n) time complexity.
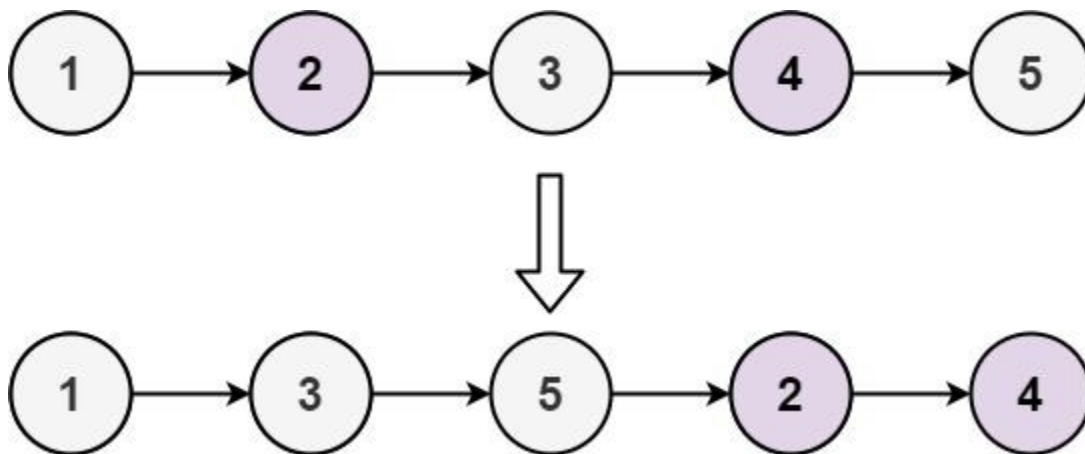
Example 1:

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

Example 2:
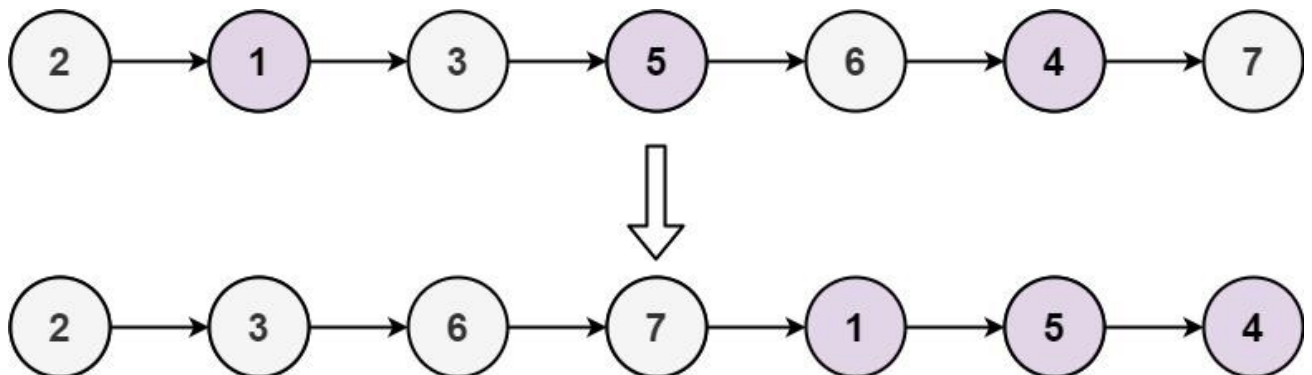
C :

```c
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
struct ListNode* oddEvenList(struct ListNode* head) {
    if (head == NULL || head->next == NULL) {
        return head; // No or only one node, no reordering needed
    }

    struct ListNode* oddHead = head;      // Head of the odd indices list
    struct ListNode* evenHead = head->next; // Head of the even indices list
    struct ListNode* odd = oddHead;       // Pointer for iterating over odd indices
    struct ListNode* even = evenHead;     // Pointer for iterating over even indices

    while (even != NULL && even->next != NULL) {
        // Connect odd indices
        odd->next = even->next;
        odd = odd->next;

        // Connect even indices
        even->next = odd->next;
        even = even->next;
    }

    // Combine the odd and even lists
    odd->next = evenHead;

    return oddHead;
}
```

1.34   Best Time To Buy And Sell Stock

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

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

C :

```c
int maxProfit(int* prices, int pricesSize) {
    if (pricesSize <= 1) {
        return 0; // No transactions can be done with less than 2 prices
    }

    int minPrice = INT_MAX; // Initialize minimum price to maximum possible value
    int maxProfit = 0;

    for (int i = 0; i < pricesSize; i++) {
        // Update the minimum price if the current price is smaller
        if (prices[i] < minPrice) {
            minPrice = prices[i];
        }
        // Update the maximum profit if selling at the current price yields a higher
profit
        else if (prices[i] - minPrice > maxProfit) {
            maxProfit = prices[i] - minPrice;
        }
    }

    return maxProfit;
}
```

1.35   Longest Consecutive Sequence

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

C :

```c
// HashSet structure
typedef struct {
    int* array;
    int size;
} HashSet;

// Function to initialize a HashSet
HashSet* initHashSet(int size) {
    HashSet* set = (HashSet*)malloc(sizeof(HashSet));
    set->array = (int*)calloc(size, sizeof(int));
    set->size = size;
    return set;
}

// Function to insert an element into the HashSet
void insertHashSet(HashSet* set, int value) {
    set->array[value] = 1;
}

// Function to check if an element exists in the HashSet
int containsHashSet(HashSet* set, int value) {
    return set->array[value];
}

// Function to calculate the length of the longest consecutive sequence
int longestConsecutive(int* nums, int numsSize) {
    if (numsSize <= 1) {
        return numsSize; // No consecutive sequence with one or zero elements
    }

    // Initialize the HashSet
    int min = nums[0], max = nums[0];
    for (int i = 1; i < numsSize; i++) {
        if (nums[i] < min) {
            min = nums[i];
        } else if (nums[i] > max) {
            max = nums[i];
        }
    }
```

```
    HashSet* set = initHashSet(max - min + 1);

    // Insert all elements into the HashSet
    for (int i = 0; i < numsSize; i++) {
        insertHashSet(set, nums[i] - min);
    }

    // Calculate the length of the longest consecutive sequence
    int longestStreak = 0, currentStreak = 0;
    for (int i = 0; i <= max - min; i++) {
        if (containsHashSet(set, i)) {
            currentStreak++;
        } else {
            longestStreak = currentStreak > longestStreak ? currentStreak :
longestStreak;
            currentStreak = 0;
        }
    }

    // Free memory allocated for the HashSet
    free(set->array);
    free(set);

    return longestStreak > currentStreak ? longestStreak : currentStreak;
}
```

1.36   Spiral Matrix

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

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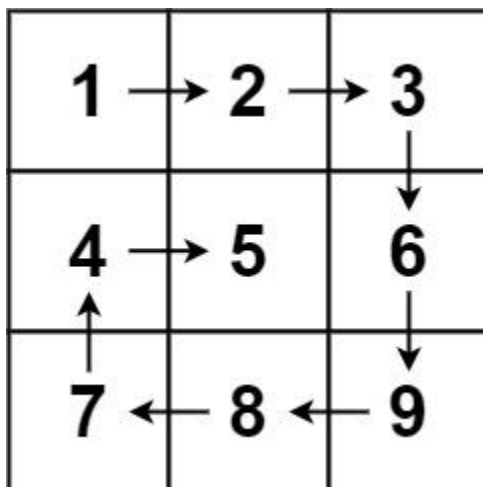
Input: matrix = [[1,2,3],[4,5,6],[7,8,9]]
Output: [1,2,3,6,9,8,7,4,5]

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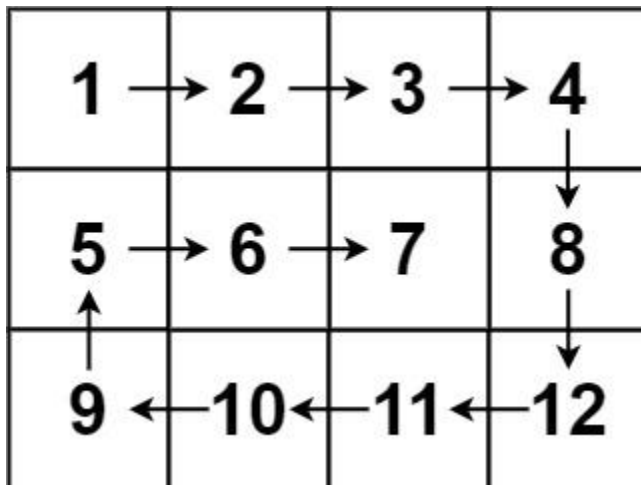
Input: matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]
Output: [1,2,3,4,8,12,11,10,9,5,6,7]

C:

```c
/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
int* spiralOrder(int** matrix, int matrixSize, int* matrixColSize, int* returnSize) {
    if (matrixSize == 0 || matrixColSize[0] == 0) {
        *returnSize = 0;
        return NULL;
    }

    int totalElements = matrixSize * matrixColSize[0];
    int* result = (int*)malloc(sizeof(int) * totalElements);

    *returnSize = totalElements;

    int top = 0, bottom = matrixSize - 1;
    int left = 0, right = matrixColSize[0] - 1;
    int index = 0;

    while (top <= bottom && left <= right) {
        // Traverse top row
        for (int i = left; i <= right; i++) {
            result[index++] = matrix[top][i];
        }
        top++;
```

```
            // Traverse right column
            for (int i = top; i <= bottom; i++) {
                result[index++] = matrix[i][right];
            }
            right--;

            // Check if there's a bottom row to traverse
            if (top <= bottom) {
                // Traverse bottom row
                for (int i = right; i >= left; i--) {
                    result[index++] = matrix[bottom][i];
                }
                bottom--;
            }

            // Check if there's a left column to traverse
            if (left <= right) {
                // Traverse left column
                for (int i = bottom; i >= top; i--) {
                    result[index++] = matrix[i][left];
                }
                left++;
            }
        }
    }

    return result;
}
```

1.37   Populating Next Right Pointer In Each Node

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

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

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

Initially, all next pointers are set to NULL.

Example 1:
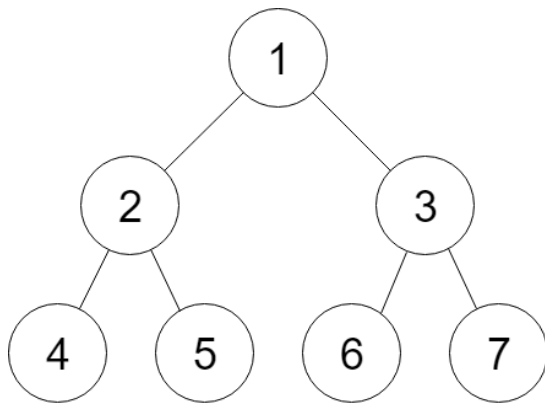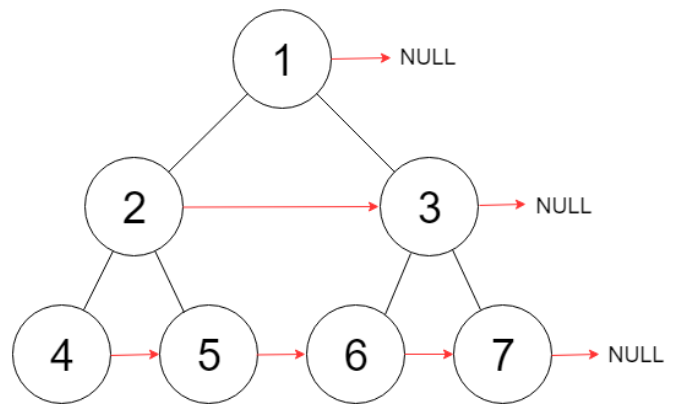
Figure A   Figure B

Input: root = [1,2,3,4,5,6,7]
Output: [1,#,2,3,#,4,5,6,7,#]
Explanation: Given the above perfect binary tree (Figure A), your function should populate each next pointer to point to its next right node, just like in Figure B. The serialized output is in level order as connected by the next pointers, with '#' signifying the end of each level.

Example 2:

Input: root = []
Output: []

C :

```c
/**
 * Definition for a Node.
 * struct Node {
 *     int val;
 *     struct Node *left;
 *     struct Node *right;
 *     struct Node *next;
 * };
 */

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

    struct Node* leftmost = root; // Start from the leftmost node in the current level

    while (leftmost->left != NULL) {
        struct Node* current = leftmost;

        // Traverse the current level and connect nodes in the next level
        while (current != NULL) {
            current->left->next = current->right;

            if (current->next != NULL) {
```

```
                    current->right->next = current->next->left;
            }

            current = current->next; // Move to the next node in the current level
        }

        leftmost = leftmost->left; // Move to the leftmost node in the next level
    }

    return root;
}
```

1.38 String Compression

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.

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","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 "bbbbbbbbbbbb". This compresses to "ab12".

C :

```c
int compress(char* chars, int charsSize) {
    if (charsSize <= 1) {
        return charsSize;
    }

    int writeIndex = 0, readIndex = 0;

    while (readIndex < charsSize) {
        char currentChar = chars[readIndex];
        int count = 0;

        while (readIndex < charsSize && chars[readIndex] == currentChar) {
            readIndex++;
            count++;
        }

        if (count == 1) {
            chars[writeIndex++] = currentChar;
        } else {
            chars[writeIndex++] = currentChar;

            // Process the count and update the array
            int tempCount = count;
            while (tempCount > 0) {
                chars[writeIndex++] = '0' + tempCount % 10;
                tempCount /= 10;
            }

            // Reverse the count characters in the array
            int start = writeIndex - count + 1;
            int end = writeIndex - 1;
            while (start < end) {
                char temp = chars[start];
                chars[start] = chars[end];
                chars[end] = temp;
                start++;
                end--;
            }
        }
    }

    return writeIndex;
}
```

1.39 Construct Binary Tree From Inorder and post Order Traversal

Given two integer arrays inorder and postorder where inorder is the inorder traversal of a binary tree and postorder is the postorder traversal of the same tree, construct and return *the binary tree*.
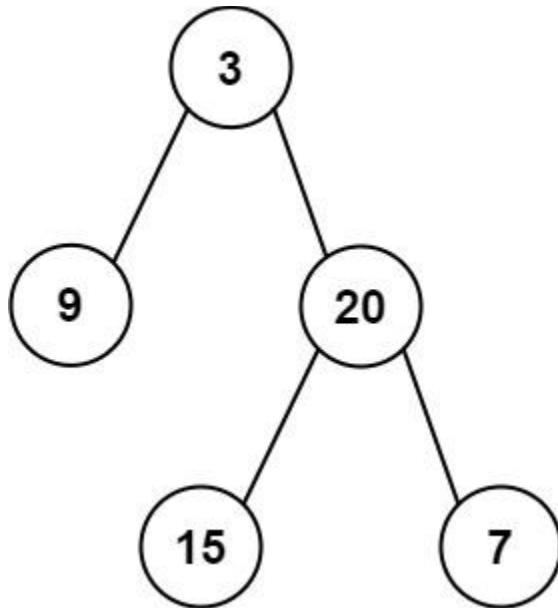
Example 1:

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

Example 2:

Input: inorder = [-1], postorder = [-1]
Output: [-1]

Constraints:

- 1 <= inorder.length <= 3000
- postorder.length == inorder.length
- -3000 <= inorder[i], postorder[i] <= 3000
- inorder and postorder consist of **unique** values.
- Each value of postorder also appears in inorder.
- inorder is **guaranteed** to be the inorder traversal of the tree.
- postorder is **guaranteed** to be the postorder traversal of the tree.

C :

```c
struct TreeNode* createNode(int val) {
    struct TreeNode* node = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    node->val = val;
    node->left = NULL;
    node->right = NULL;
    return node;
}

struct TreeNode* helper(int* inorder, int in_left, int in_right, int* post_idx, int*
postorder) {
    // if there are no elements to construct subtrees
    if (in_left > in_right)
        return NULL;

    // pick up post_idx element as a root
    int root_val = postorder[(*post_idx)];
    struct TreeNode* root = createNode(root_val);

    // root splits inorder list
    // into left and right subtrees
    int index = 0;
    for (int i = in_left; i <= in_right; i++) {
        if (inorder[i] == root_val) {
            index = i;
            break;
        }
    }

    // recursion
    (*post_idx)--;
    // build the right subtree
    root->right = helper(inorder, index + 1, in_right, post_idx, postorder);
    // build the left subtree
    root->left = helper(inorder, in_left, index - 1, post_idx, postorder);
    return root;
}

struct TreeNode* buildTree(int* inorder, int inorderSize, int* postorder, int
postorderSize) {
    int post_idx = postorderSize - 1;
    return helper(inorder, 0, inorderSize - 1, &post_idx, postorder);
}
```

1.40   Valid Sudoku

Determine if a 9 x 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 x 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:

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

C:

```c
bool isValidSudoku(char** board, int boardSize, int* boardColSize) {
    // Arrays to check for duplicate digits in rows, columns, and sub-boxes
    bool row[9][9] = {false};
    bool col[9][9] = {false};
    bool box[9][9] = {false};

    for (int i = 0; i < boardSize; i++) {
        for (int j = 0; j < *boardColSize; j++) {
            if (board[i][j] != '.') {
                int digit = board[i][j] - '1';

                // Check for duplicate in the same row
                if (row[i][digit]) {
                    return false;
                }
                row[i][digit] = true;

                // Check for duplicate in the same column
                if (col[j][digit]) {
                    return false;
                }
                col[j][digit] = true;

                // Check for duplicate in the same 3x3 sub-box
                int boxIndex = (i / 3) * 3 + j / 3;
                if (box[boxIndex][digit]) {
                    return false;
                }
                box[boxIndex][digit] = true;
            }
        }
    }

    return true;
}
```

1.41 Path Sum

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.
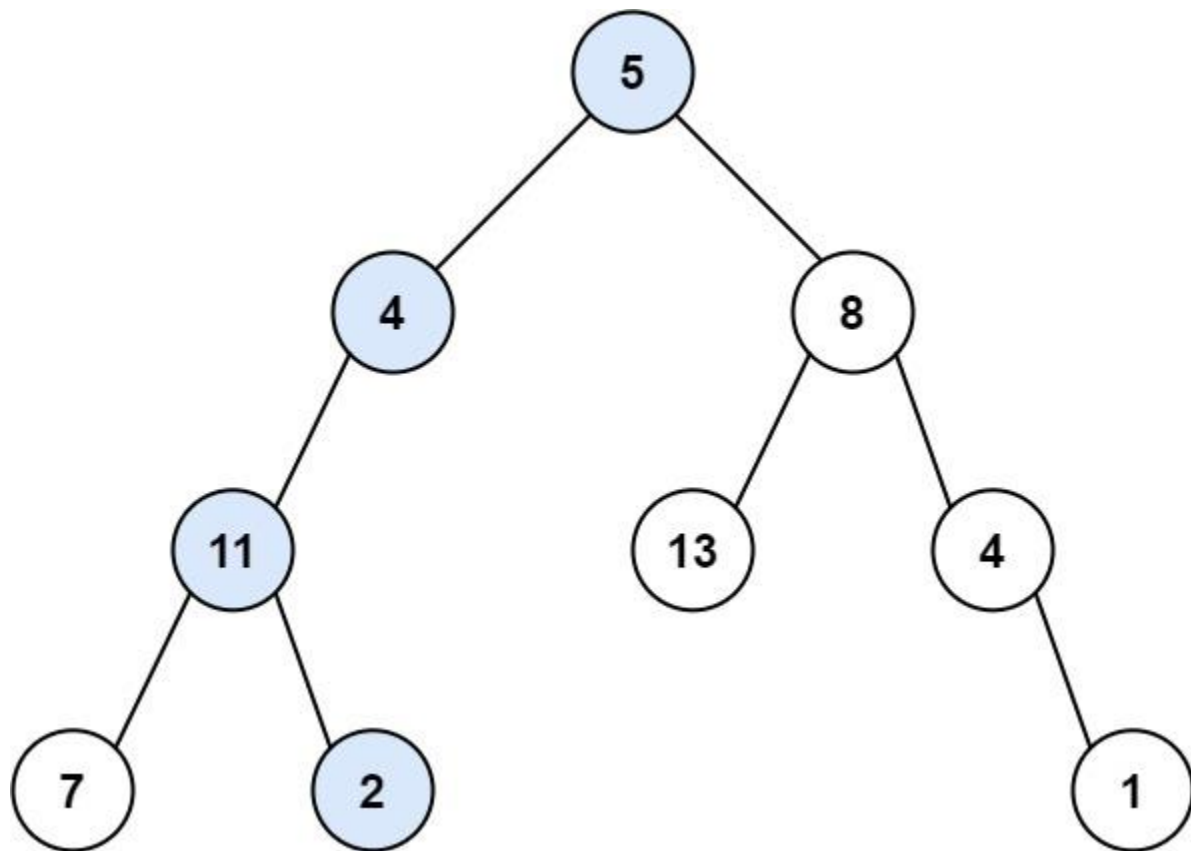
Input: root = [5,4,8,11,null,13,4,7,2,null,null,null,1], targetSum = 22
Output: true
Explanation: The root-to-leaf path with the target sum is shown.

Example 2:
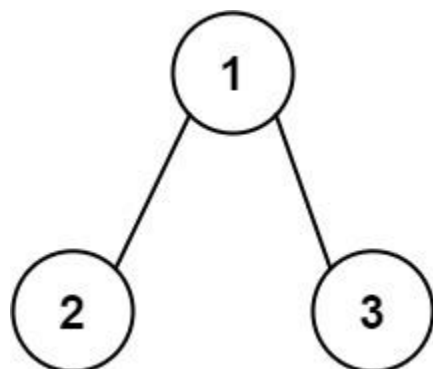
Input: root = [1,2,3], targetSum = 5
Output: false
Explanation: There two root-to-leaf paths in the tree:
(1 --> 2): The sum is 3.
(1 --> 3): The sum is 4.
There is no root-to-leaf path with sum = 5.

Input: root = [], targetSum = 0
Output: false
Explanation: Since the tree is empty, there are no root-to-leaf paths.

C :

```c
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */
bool hasPathSum(struct TreeNode* root, int targetSum) {
    // Base case: If the current node is NULL, return false.
    if (root == NULL) {
        return false;
    }

    // Subtract the value of the current node from the target sum.
    targetSum -= root->val;

    // If the current node is a leaf node and the remaining sum is 0, return true.
    if (root->left == NULL && root->right == NULL) {
        return targetSum == 0;
    }

    // Recursively check the left and right subtrees.
    bool leftResult = hasPathSum(root->left, targetSum);
    bool rightResult = hasPathSum(root->right, targetSum);

    // Return true if either the left or right subtree has a path sum equal to the
target sum.
    return leftResult || rightResult;
}
```

1.42 Design Add and Search Words DataStructure

Design a data structure that supports adding new words and finding if a string matches any previously added string.

Implement the WordDictionary class:

- WordDictionary() Initializes the object.
- void addWord(word) Adds word to the data structure, it can be matched later.

- bool search(word) Returns true if there is any string in the data structure that matches word or false otherwise. word may contain dots '.' where dots can be matched with any letter.

Example:

Input
["WordDictionary","addWord","addWord","addWord","search","search","search","search"]
[[],["bad"],["dad"],["mad"],["pad"],["bad"],[".ad"],["b.."]]
Output
[null,null,null,null,false,true,true,true]

Explanation
WordDictionary wordDictionary = new WordDictionary();
wordDictionary.addWord("bad");
wordDictionary.addWord("dad");
wordDictionary.addWord("mad");
wordDictionary.search("pad"); // return False
wordDictionary.search("bad"); // return True
wordDictionary.search(".ad"); // return True
wordDictionary.search("b.."); // return True

C :

```c
// Definition for a trie node
typedef struct TrieNode {
    bool isEnd;
    struct TrieNode* children[26];
} TrieNode;

// Definition for WordDictionary
typedef struct {
    TrieNode* root;
} WordDictionary;

// Helper function to create a new trie node
TrieNode* createTrieNode() {
    TrieNode* node = (TrieNode*)malloc(sizeof(TrieNode));
    node->isEnd = false;
    memset(node->children, 0, sizeof(node->children));
    return node;
}

WordDictionary* wordDictionaryCreate() {
    WordDictionary* obj = (WordDictionary*)malloc(sizeof(WordDictionary));
    obj->root = createTrieNode();
    return obj;
}
```

```c
void addWordToTrie(TrieNode* root, char* word) {
    TrieNode* node = root;
    for (int i = 0; i < strlen(word); ++i) {
        int index = word[i] - 'a';
        if (node->children[index] == NULL) {
            node->children[index] = createTrieNode();
        }
        node = node->children[index];
    }
    node->isEnd = true;
}

void wordDictionaryAddWord(WordDictionary* obj, char* word) {
    addWordToTrie(obj->root, word);
}

bool searchInTrie(TrieNode* node, char* word) {
    for (int i = 0; i < strlen(word); ++i) {
        if (word[i] == '.') {
            for (int j = 0; j < 26; ++j) {
                if (node->children[j] != NULL &&
                    searchInTrie(node->children[j], word + i + 1)) {
                    return true;
                }
            }
            return false;
        } else {
            int index = word[i] - 'a';
            if (node->children[index] == NULL) {
                return false;
            }
            node = node->children[index];
        }
    }
    return node->isEnd;
}

bool wordDictionarySearch(WordDictionary* obj, char* word) {
    return searchInTrie(obj->root, word);
}

void freeTrie(TrieNode* node) {
    if (node == NULL) {
        return;
    }
    for (int i = 0; i < 26; ++i) {
        freeTrie(node->children[i]);
    }
```

```
        free(node);
}

void wordDictionaryFree(WordDictionary* obj) {
    freeTrie(obj->root);
    free(obj);
}

/**
 * Your WordDictionary struct will be instantiated and called as such:
 * WordDictionary* obj = wordDictionaryCreate();
 * wordDictionaryAddWord(obj, word);

 * bool param_2 = wordDictionarySearch(obj, word);

 * wordDictionaryFree(obj);
*/
```

1.43   Sudoku Solver

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 3x3 sub-boxes of the grid.

The '.' character indicates empty cells.

Example 1:

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:
[["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"]]

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

C:

```c
#define SIZE 9

bool isSafe(char** board, int row, int col, char num) {
    // Check if 'num' is not already present in the current row and column
    for (int x = 0; x < SIZE; x++) {
        if (board[row][x] == num || board[x][col] == num) {
            return false;
        }
    }

    // Check if 'num' is not already present in the current 3x3 sub-grid
    int startRow = row - row % 3;
    int startCol = col - col % 3;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (board[i + startRow][j + startCol] == num) {
                return false;
            }
        }
    }

    return true;
}

bool findUnassignedLocation(char** board, int* row, int* col) {
```

```c
    for (*row = 0; *row < SIZE; (*row)++) {
        for (*col = 0; *col < SIZE; (*col)++) {
            if (board[*row][*col] == '.') {
                return true;
            }
        }
    }
    return false;
}

bool solve(char** board) {
    int row, col;

    if (!findUnassignedLocation(board, &row, &col)) {
        // No unassigned location, puzzle is solved
        return true;
    }

    for (char num = '1'; num <= '9'; num++) {
        if (isSafe(board, row, col, num)) {
            // Try placing 'num' in the current position
            board[row][col] = num;

            // Recursively solve the rest of the puzzle
            if (solve(board)) {
                return true;
            }

            // If placing 'num' didn't lead to a solution, backtrack
            board[row][col] = '.';
        }
    }

    // No number can be placed in the current position
    return false;
}

void solveSudoku(char** board, int boardSize, int* boardColSize) {
    solve(board);
}
```

1.44 Max Area Of Island

You are given an m x n binary matrix grid. An island is a group of 1's (representing land) connected **4-directionally** (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

The **area** of an island is the number of cells with a value 1 in the island.

Return *the maximum **area** of an island in* grid. If there is no island, return 0.

0	0	1	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	1	1	1	0	0	0
0	1	1	0	1	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	0	1	0	1	0	0
0	1	0	0	1	1	0	0	1	1	1	0	0
0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	1	1	1	0	0	0
0	0	0	0	0	0	0	1	1	0	0	0	0

Input: grid =
[[0,0,1,0,0,0,0,1,0,0,0,0,0],[0,0,0,0,0,0,0,1,1,1,0,0,0],[0,1,1,0,1,0,0,0,0,0,0,0,0],[0,1,0,0,1,1,0,0,1,0,1,0,0],[0,1,0,0,1,1,0,0,1,1,1,0,0],[0,0,0,0,0,0,0,0,0,0,1,0,0],[0,0,0,0,0,0,0,1,1,1,0,0,0],[0,0,0,0,0,0,0,1,1,0,0,0,0]]
Output: 6
Explanation: The answer is not 11, because the island must be connected 4-directionally.

Example 2:

Input: grid = [[0,0,0,0,0,0,0,0]]
Output: 0

C :

```c
int maxAreaOfIsland(int** grid, int gridSize, int* gridColSize) {
    int maxArea = 0;

    if (grid == NULL || gridSize == 0) {
        return maxArea;
    }

    for (int i = 0; i < gridSize; i++) {
        for (int j = 0; j < gridColSize[i]; j++) {
            if (grid[i][j] == 1) {
                int area = evaluateGrid(grid, i, j, gridSize, gridColSize[i]);
                maxArea = area > maxArea ? area : maxArea;
```

```
                }
            }
        }

        return maxArea;
}

int evaluateGrid(int** grid, int i, int j, int row, int col) {
    if (i < 0 || j < 0 || i >= row || j >= col || grid[i][j] != 1) {
        return 0;
    }

    grid[i][j] = 2;
    int area = 1; // Current cell contributes to the area

    // Explore neighbors
    area += evaluateGrid(grid, i - 1, j, row, col);
    area += evaluateGrid(grid, i + 1, j, row, col);
    area += evaluateGrid(grid, i, j - 1, row, col);
    area += evaluateGrid(grid, i, j + 1, row, col);

    return area;
}
```

1.45   Valid Parantheses

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 = "(]"

C :

```c
bool isValid(char* s) {
    int length = 0;
    char* stack = (char*)malloc(strlen(s) * sizeof(char));

    for (int i = 0; s[i] != '\0'; i++) {
        if (s[i] == '(' || s[i] == '[' || s[i] == '{') {
            stack[length++] = s[i];
        } else {
            if (length == 0) {
                free(stack);
                return false; // No corresponding open bracket
            }

            char openBracket = stack[--length];

            if ((s[i] == ')' && openBracket != '(') ||
                (s[i] == ']' && openBracket != '[') ||
                (s[i] == '}' && openBracket != '{')) {
                free(stack);
                return false; // Mismatched brackets
            }
        }
    }

    bool result = (length == 0); // Stack should be empty for a valid string
    free(stack);
    return result;
}
```

1.46   Merge Two Sorted Lists

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:
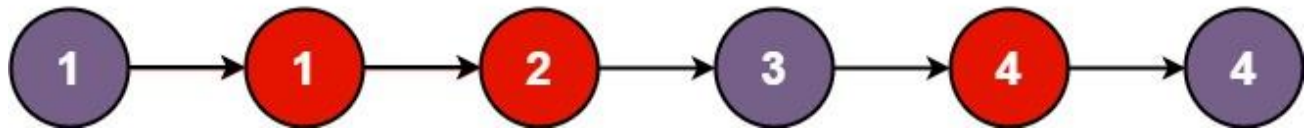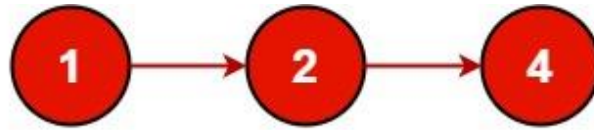
Input: list1 = [1,2,4], list2 = [1,3,4]
Output: [1,1,2,3,4,4]

Example 2:

Input: list1 = [], list2 = []
Output: []

Example 3:

Input: list1 = [], list2 = [0]
Output: [0]

C :

```c
struct ListNode* mergeTwoLists(struct ListNode* list1, struct ListNode* list2){
    struct ListNode *dummy = (struct ListNode *)malloc(sizeof(struct ListNode));
    dummy->next = NULL;
    struct ListNode *curr = dummy;
    while (list1 != NULL && list2 != NULL) {
        if (list1->val < list2->val) {
            dummy->next = list1;
            list1 = list1->next;
            dummy = dummy->next;
        } else {
            dummy->next = list2;
            list2 = list2->next;
            dummy = dummy->next;
        }
    }
    while (list1 != NULL) {
        dummy->next = list1;
        list1 = list1->next;
        dummy = dummy->next;
    }
```

```
    while (list2 != NULL) {
        dummy->next = list2;
        list2 = list2->next;
        dummy = dummy->next;
    }

    return curr->next;
}
```

1.47   Container With Most Water

You are given an integer array height of length n. There are n vertical lines drawn such that the two endpoints of the ith line are (i, 0) and (i, 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:
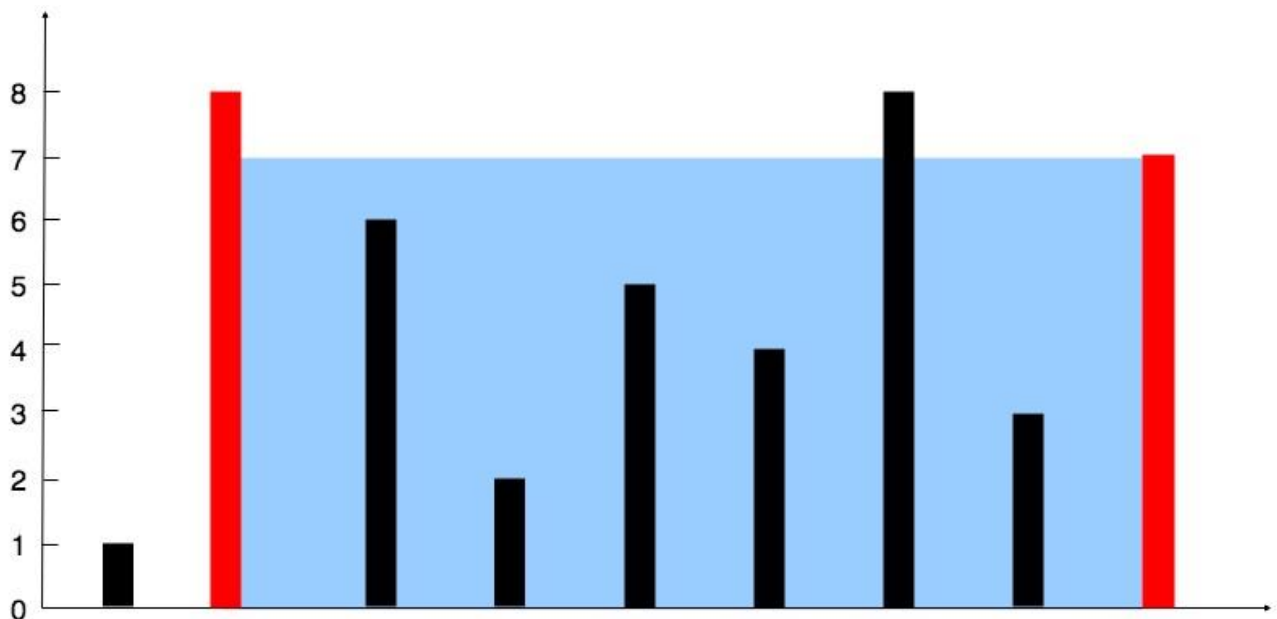
Input: height = [1,8,6,2,5,4,8,3,7]
Output: 49
Explanation: The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this case, the max area of water (blue section) the container can contain is 49.

Example 2:

Input: height = [1,1]
Output: 1

C :

```c
#define MAX(a,b) ((a) > (b)?(a):(b))
#define MIN(a,b) ((a) > (b)?(b):(a))
int maxArea(int* height, int heightSize) {
    int max_area = INT_MIN;
    int left = 0;
    int right = heightSize-1;
    while (left < right) {
        max_area = MAX(max_area,(MIN(height[left],height[right])*(abs(left-right))));
        if (height[left] < height[right]) left++;
        else right--;
    }
    return max_area;
}
```

1.48   Pascals Triangle

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:
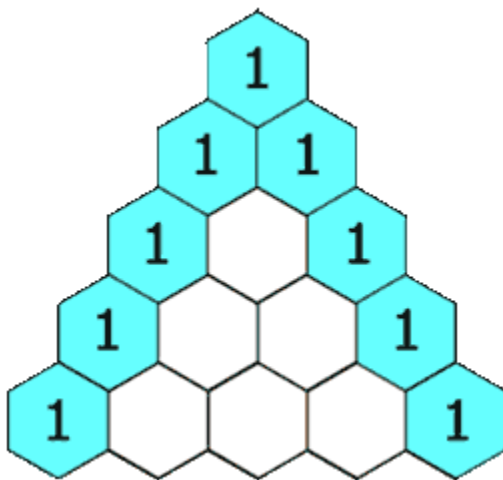
Example 1:

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]]

C:

```c
/**
 * Return an array of arrays of size *returnSize.
 * The sizes of the arrays are returned as *returnColumnSizes array.
 * Note: Both returned array and *columnSizes array must be malloced, assume caller
calls free().
 */
int** generate(int numRows, int* returnSize, int** returnColumnSizes) {
    if (numRows <= 0) {
        *returnSize = 0;
        *returnColumnSizes = NULL;
        return NULL;
    }

    // Allocate memory for the result
    int** result = (int**)malloc(sizeof(int*) * numRows);
    *returnColumnSizes = (int*)malloc(sizeof(int) * numRows);

    for (int i = 0; i < numRows; i++) {
        result[i] = (int*)malloc(sizeof(int) * (i + 1));
        (*returnColumnSizes)[i] = i + 1;

        // Set the first and last elements to 1
        result[i][0] = result[i][i] = 1;

        // Calculate the middle elements based on the previous row
        for (int j = 1; j < i; j++) {
            result[i][j] = result[i - 1][j - 1] + result[i - 1][j];
        }
    }

    *returnSize = numRows;
    return result;
}
```

1.49   Valid Anagram

Given two strings s and t, return true *if* t *is an anagram of* s*, and* false *otherwise*.

An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

Example 1:

Input: s = "anagram", t = "nagaram"
Output: true

Example 2:

Input: s = "rat", t = "car"
Output: false

C :

```c
bool isAnagram(char* s, char* t) {
    // Check if the lengths of both strings are equal
    int lenS = 0, lenT = 0;
    while (s[lenS] != '\0') {
        lenS++;
    }
    while (t[lenT] != '\0') {
        lenT++;
    }
    if (lenS != lenT) {
        return false;
    }

    // Initialize an array to store the frequency of characters
    int frequency[26] = {0};  // Assuming input only contains lowercase English letters

    // Increment frequency for characters in string s
    for (int i = 0; i < lenS; i++) {
        frequency[s[i] - 'a']++;
    }

    // Decrement frequency for characters in string t
    for (int i = 0; i < lenT; i++) {
        frequency[t[i] - 'a']--;
    }

    // Check if all frequencies are zero
    for (int i = 0; i < 26; i++) {
        if (frequency[i] != 0) {
            return false;
        }
    }

    // If all checks pass, the strings are anagrams
    return true;
}
```

1.50   Unique Paths

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

Given the two integers m and n, return *the number of possible unique paths that the robot can take to reach the bottom-right corner*.

The test cases are generated so that the answer will be less than or equal to $2 * 10^9$.

Example 1:

Input: m = 3, n = 7
Output: 28

Example 2:

Input: m = 3, n = 2
Output: 3
Explanation: From the top-left corner, there are a total of 3 ways to reach the bottom-right corner:
1. Right -> Down -> Down
2. Down -> Down -> Right
3. Down -> Right -> Down

C:

```c
int uniquePaths(int m, int n) {
    // Create a 2D array to store the number of unique paths for each position.
    int dp[m][n];

    // Initialize the leftmost column (only one way to reach any position in the
leftmost column).
    for (int i = 0; i < m; i++) {
        dp[i][0] = 1;
```

```
    }

    // Initialize the top row (only one way to reach any position in the top row).
    for (int j = 0; j < n; j++) {
        dp[0][j] = 1;
    }

    // Fill in the rest of the array by summing the number of unique paths from the top
and left.
    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
        }
    }

    // The result is stored in the bottom-right corner of the array.
    return dp[m - 1][n - 1];
}
```

1.51 Longest Increasing Subsequence

Given an integer array nums, return *the length of the longest **strictly increasing***

subsequence.

Example 1:

Input: nums = [10,9,2,5,3,7,101,18]
Output: 4
Explanation: The longest increasing subsequence is [2,3,7,101], therefore the length is 4.

Example 2:

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

Example 3:

Input: nums = [7,7,7,7,7,7,7]
Output: 1

C:
```
int lengthOfLIS(int* nums, int numsSize) {
    if (numsSize == 0) {
```

```
        return 0;
    }

    // Initialize an array to store the elements of the increasing subsequence
    int sub[numsSize];
    sub[0] = nums[0];
    int subSize = 1;

    for (int i = 1; i < numsSize; i++) {
        int num = nums[i];
        if (num > sub[subSize - 1]) {
            sub[subSize] = num;
            subSize++;
        } else {
            // Find the first element in sub that is greater than or equal to num
            int j = 0;
            while (num > sub[j]) {
                j++;
            }

            sub[j] = num;
        }
    }

    return subSize;
}
```

1.52 Group Anagrams

Given an array of strings strs, group **the anagrams** together. You can return the answer in **any order**.

An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

Example 1:

Input: strs = ["eat","tea","tan","ate","nat","bat"]
Output: [["bat"],["nat","tan"],["ate","eat","tea"]]

Example 2:

Input: strs = [""]
Output: [[""]]

Example 3:

Input: strs = ["a"]

C :

```java
class Solution {
    public List<List<String>> groupAnagrams(String[] strs) {
        HashMap<String, List<String>> hm = new HashMap<String, List<String>>();
        int[] count_map = new int[26];
        for (int i = 0; i < strs.length; i++) {
            Arrays.fill(count_map, 0);
            for (int j = 0; j < strs[i].length(); j++) {
                count_map[strs[i].charAt(j)-'a']++;
            }
            StringBuilder sb = new StringBuilder();
            for (int m = 0; m < 26; m++) {
                sb.append("*");
                sb.append(count_map[m]);
            }
            if (!hm.containsKey(sb.toString())) {
                hm.put(sb.toString(), new ArrayList());
            }
            hm.get(sb.toString()).add(strs[i]);
            System.out.println(sb.toString());
        }
        return new ArrayList(hm.values());
    }
}
```

1.53   Longest Palindromic SubString

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"

C :

```c
char* longestPalindrome(char* s) {
    int n = strlen(s);

    // Create a new string to insert '#' between characters
    char* modifiedString = (char*)malloc((2 * n + 2) * sizeof(char));
    int j = 0;

    for (int i = 0; i < n; i++) {
        modifiedString[j++] = '#';
        modifiedString[j++] = s[i];
    }
    modifiedString[j++] = '#';
    modifiedString[j] = '\0';

    // Initialize the palindrome array
    int* palindrome = (int*)malloc((2 * n + 2) * sizeof(int));
    memset(palindrome, 0, (2 * n + 2) * sizeof(int));

    int center = 0, right = 0;

    for (int i = 0; i < 2 * n + 1; i++) {
        int mirror = 2 * center - i;

        // Check if the mirror of i is within the right boundary
        if (i < right) {
            palindrome[i] = (right - i < palindrome[mirror]) ? right - i :
palindrome[mirror];
        }

        // Expand around the center
        int a = i + (1 + palindrome[i]);
        int b = i - (1 + palindrome[i]);

        while (a < 2 * n + 1 && b >= 0 && modifiedString[a] == modifiedString[b]) {
            palindrome[i]++;
            a++;
            b--;
        }

        // If palindrome[i] reaches the boundary of the current rightmost palindrome,
        // update the center and right boundary accordingly
        if (i + palindrome[i] > right) {
            center = i;
            right = i + palindrome[i];
        }
    }
```

```c
    // Find the maximum element in the palindrome array
    int maxLen = 0, centerIndex = 0;

    for (int i = 0; i < 2 * n + 1; i++) {
        if (palindrome[i] > maxLen) {
            maxLen = palindrome[i];
            centerIndex = i;
        }
    }

    // Extract the longest palindrome from the modified string
    int start = (centerIndex - maxLen) / 2;
    char* result = (char*)malloc((maxLen + 1) * sizeof(char));
    strncpy(result, s + start, maxLen);
    result[maxLen] = '\0';

    // Clean up
    free(modifiedString);
    free(palindrome);

    return result;
}
```

1.54 Three Sum

Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, and j != k, and nums[i] + nums[j] + nums[k] == 0.

Notice that the solution set must not contain duplicate triplets.

Example 1:

Input: nums = [-1,0,1,2,-1,-4]
Output: [[-1,-1,2],[-1,0,1]]
Explanation:
nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0.
nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0.
nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0.
The distinct triplets are [-1,0,1] and [-1,-1,2].
Notice that the order of the output and the order of the triplets does not matter.

Example 2:

Input: nums = [0,1,1]
Output: []
Explanation: The only possible triplet does not sum up to 0.

Example 3:

Input: nums = [0,0,0]
Output: [[0,0,0]]
Explanation: The only possible triplet sums up to 0.

C :

```c
/**
 * Return an array of arrays of size *returnSize.
 * The sizes of the arrays are returned as *returnColumnSizes array.
 * Note: Both returned array and *columnSizes array must be malloced, assume caller
calls free().
 */
int compare(const void *a, const void *b) {
    return (*(int *)a - *(int *)b);
}
int** threeSum(int* nums, int numsSize, int* returnSize, int** returnColumnSizes) {
     // Sort the array to simplify the solution
    qsort(nums, numsSize, sizeof(int), compare);

    // Allocate memory for the result
    int** result = NULL;
    *returnSize = 0;
    *returnColumnSizes = NULL;

    // Iterate through the array
    for (int i = 0; i < numsSize - 2; ++i) {
        // Skip duplicates to avoid duplicate triplets
        if (i > 0 && nums[i] == nums[i - 1]) {
            continue;
        }

        int left = i + 1;
        int right = numsSize - 1;

        while (left < right) {
            int sum = nums[i] + nums[left] + nums[right];
            if (sum == 0) {
                // Found a triplet, allocate memory for it
                result = (int**)realloc(result, (*returnSize + 1) * sizeof(int*));
                result[*returnSize] = (int*)malloc(3 * sizeof(int));

                // Assign values to the triplet
                result[*returnSize][0] = nums[i];
                result[*returnSize][1] = nums[left];
                result[*returnSize][2] = nums[right];

                // Increment the result size
                (*returnSize)++;
```

```
                // Skip duplicates for left pointer
                while (left < right && nums[left] == nums[left + 1]) {
                    left++;
                }

                // Skip duplicates for right pointer
                while (left < right && nums[right] == nums[right - 1]) {
                    right--;
                }

                // Move pointers
                left++;
                right--;
            } else if (sum < 0) {
                left++;
            } else {
                right--;
            }
        }
    }

    // Allocate memory for column sizes
    *returnColumnSizes = (int*)malloc(*returnSize * sizeof(int));
    for (int i = 0; i < *returnSize; ++i) {
        (*returnColumnSizes)[i] = 3;
    }

    return result;
}
class Solution {
    public List<List<Integer>> threeSum(int[] nums) {
        HashSet<Integer> hs = new HashSet<Integer>();
        HashMap<Integer,Integer> hm = new HashMap<Integer,Integer>();
        HashSet<List<Integer>> result = new HashSet<List<Integer>>();

        for (int i = 0; i < nums.length; i++) {
            if (hs.add(nums[i])) {
                for (int j = i+1; j < nums.length; j++) {
                    int target = -nums[i]-nums[j];
                    if (hm.containsKey(target) && hm.get(target) == i) {
                        List<Integer> entry = new ArrayList<Integer>();
                        entry.add(nums[i]);
                        entry.add(nums[j]);
                        entry.add(target);
                        Collections.sort(entry);
                        result.add(entry);
                    } else {
                        hm.put(nums[j],i);
                    }
```

```
                }
            }
        }
        return new ArrayList(result);
    }
}
/*
[-1,0,1,2,-1,-4]
duplicates: - 1 , hm : 0:0 and complement = 1
duplicates: -1 , hm: 0:0 and 2:0 complement = -1
*/
```

1.55   Single Element In A Sorted Array

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

C :
```c
int singleNonDuplicate(int* nums, int numsSize) {
    int r = 0;
    for (int i = 0; i < numsSize; i++)
    r = r ^ nums[i];

    return r;
}
```
1.56   Letter Combinations Of A Phone Number

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 = ""
Output: []

Example 3:

Input: digits = "2"

C :

```c
/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
// Helper function to append a character to a string
void append(char* str, char c) {
    int len = strlen(str);
    str[len] = c;
    str[len + 1] = '\0';
}


// Helper function to perform backtracking
void backtrack(char** result, char* digits, char* current, char** mapping, int index,
int* returnSize) {
    // If the current combination is complete, add it to the result
    if (digits[index] == '\0') {
        result[*returnSize] = strdup(current);
        (*returnSize)++;
        return;
    }

    // Get the letters corresponding to the current digit
    char* letters = mapping[digits[index] - '0'];

    // Iterate through the letters and backtrack
    for (int i = 0; letters[i] != '\0'; i++) {
        append(current, letters[i]);
        backtrack(result, digits, current, mapping, index + 1, returnSize);
        // Remove the last character to backtrack
        current[strlen(current) - 1] = '\0';
    }
}

// Main function to generate letter combinations
char** letterCombinations(char* digits, int* returnSize) {
    // Define the mapping of digits to letters
    char* mapping[] = {"", "", "abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv",
"wxyz"};

    // Allocate space for the result
    char** result = (char**)malloc(sizeof(char*) * 1000);
    *returnSize = 0;

    // Check if the input is empty
    if (digits == NULL || *digits == '\0') {
        return result;
    }
```

```java
    // Start the backtracking process
    char current[1000] = "";
    backtrack(result, digits, current, mapping, 0, returnSize);

    return result;
}
class Solution {
    HashMap<Character, String> hm = new HashMap<Character, String>();


    public void generate_words(String current_word, String digits, int current_index,
List<String> result) {
        if (current_index == digits.length()) {
            result.add(current_word);
            return;
        }

        String seq = hm.get(digits.charAt(current_index));

        for (int i = 0; i < seq.length(); i++) {
            generate_words(current_word+seq.charAt(i), digits, current_index+1, result);
        }

    }
    public void helper(String digits, List<String> result) {
        generate_words("", digits, 0, result);
    }
    public List<String> letterCombinations(String digits) {
        List<String> result = new ArrayList<String>();
        hm.put('2',"abc");
        hm.put('3',"def");
        hm.put('4',"ghi");
        hm.put('5',"jkl");
        hm.put('6',"mno");
        hm.put('7',"pqrs");
        hm.put('8',"tuv");
        hm.put('9',"wxyz");
        if (digits.length() == 0) return result;
        helper(digits,result);
        return result;
    }
}
```

1.57   Find The Index Of The First Occurrence In The String

Given two strings needle and haystack, return the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

Input: haystack = "sadbutsad", needle = "sad"
Output: 0
Explanation: "sad" occurs at index 0 and 6.
The first occurrence is at index 0, so we return 0.

Input: haystack = "leetcode", needle = "leeto"
Output: -1
Explanation: "leeto" did not occur in "leetcode", so we return -1.

C:

```c
int strStr(char * haystack, char * needle){
    int i = 0;
    int j = 0;
    int start = -1;
    while (i < strlen(haystack)) {
        if (haystack[i] == needle[j]) {
            if (start == -1) {
                start = i;
            }
            if (j == strlen(needle)-1) {
                return start;
            }
            i++;
            j++;
        }
        else {
            if (i == strlen(haystack)-1) {
                break;
            }
            i=start+1;
            start = i;
            j = 0;

        }
    }
    return -1;
}
```

1.58 Best Time to Buy And Sell Stock II (Buy Sell Any Times)

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

On each day, you may decide to buy and/or sell the stock. You can only hold **at most one** share of the stock at any time. However, you can buy it then immediately sell it on the **same day**.

Find and return *the **maximum** profit you can achieve*.

Example 1:

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

Example 2:

Input: prices = [1,2,3,4,5]
Output: 4
Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 = 4.
Total profit is 4.

Example 3:

Input: prices = [7,6,4,3,1]
Output: 0
Explanation: There is no way to make a positive profit, so we never buy the stock to achieve the maximum profit of 0.

C :

```c
int maxProfit(int* prices, int pricesSize){
    int  i = 0;
    int result = 0;
    for (i = 1; i < pricesSize; i++) {
        if (prices[i] > prices[i-1]) result+=(prices[i]-prices[i-1]) ;
    }
    return result;
}
```

1.59 Intersection Of Two Arrays

Given two integer arrays nums1 and nums2, return *an array of their intersection*. Each element in the result must be **unique** and you may return the result in **any order**.

Example 1:

Input: nums1 = [1,2,2,1], nums2 = [2,2]
Output: [2]

Example 2:

Input: nums1 = [4,9,5], nums2 = [9,4,9,8,4]

C :

```c
int compare(const void* a, const void* b) {
    return (*(int*)a - *(int*)b);
}

int* intersection(int* nums1, int nums1Size, int* nums2, int nums2Size, int* returnSize)
{
    qsort(nums1, nums1Size, sizeof(int), compare);
    qsort(nums2, nums2Size, sizeof(int), compare);

    int* result = (int*)malloc(sizeof(int) * (nums1Size + nums2Size));
    *returnSize = 0;

    int i = 0, j = 0;

    while (i < nums1Size && j < nums2Size) {
        if (nums1[i] == nums2[j]) {
            // Add the common element to the result if it is not already present
            if (*returnSize == 0 || result[*returnSize - 1] != nums1[i]) {
                result[(*returnSize)++] = nums1[i];
            }
            i++;
            j++;
        } else if (nums1[i] < nums2[j]) {
            i++;
        } else {
            j++;
        }
    }

    return result;
}
```

1.60   Climbing Stairs

You are climbing a staircase. It takes n steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Example 1:

Input: n = 2
Output: 2
Explanation: There are two ways to climb to the top.
1. 1 step + 1 step

2. 2 steps

Example 2:

Input: n = 3
Output: 3
Explanation: There are three ways to climb to the top.
1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step

C :

```c
int climbStairs(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    if (n == 2) return 2;
    int *result = (int *)malloc(sizeof(int)*(n+1));
    int i = 0;
    result[0] = 0;
    result[1] = 1;
    result[2] = 2;
    for (i = 3; i <= n; i++) {
        result[i] = result[i-1] + result[i-2];
    }
    return result[n];

}
```

1.61 Convert Sorted Array To Binary Search Tree

binary search tree.

Example 1:
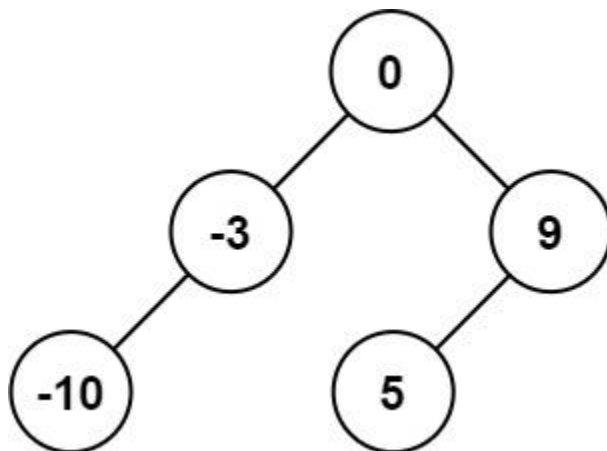
Input: nums = [-10,-3,0,5,9]
Output: [0,-3,9,-10,null,5]
Explanation: [0,-10,5,null,-3,null,9] is also accepted:
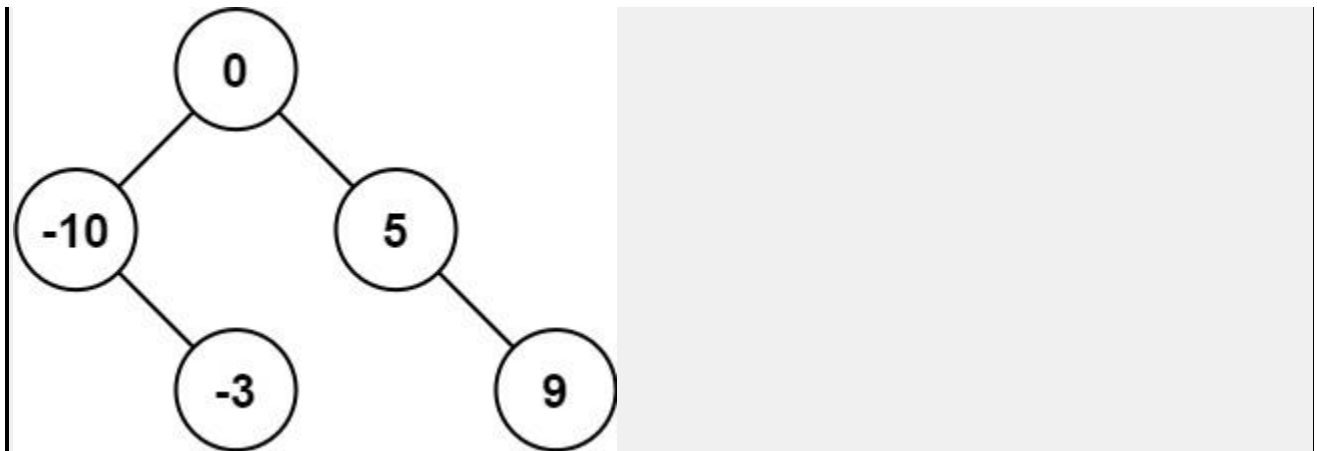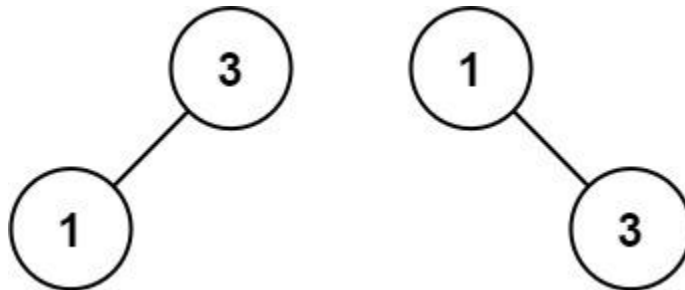
Input: nums = [1,3]
Output: [3,1]
Explanation: [1,null,3] and [3,1] are both height-balanced BSTs.

C :

```c
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */
struct TreeNode* helper(int* nums, int left, int right) {
    if (left > right)
        return NULL;

    int middle = (left + right) / 2;
    struct TreeNode* result = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    result->val = nums[middle];
    result->left = helper(nums, left, middle - 1);
    result->right = helper(nums, middle + 1, right);

    return result;
```

```
}

struct TreeNode* sortedArrayToBST(int* nums, int numsSize) {
    return helper(nums, 0, numsSize - 1);
}
```

1.62   Kth Largest Element In Array

Given an integer array nums and an integer k, return *the* kth *largest element in the array*.

Note that it is the kth largest element in the sorted order, not the kth distinct element.

Can you solve it without sorting?

Example 1:

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

Example 2:

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

C :

```
class Solution {
    public int findKthLargest(int[] nums, int k) {
        Queue<Integer> q = new PriorityQueue<Integer>();
        for (int n : nums) {
            q.add(n);
            if (q.size() > k) q.poll();
        }
        System.out.println(q);
        return q.poll();
    }
}
```

1.63   Binary Tree ZigZag Level Order Traversal

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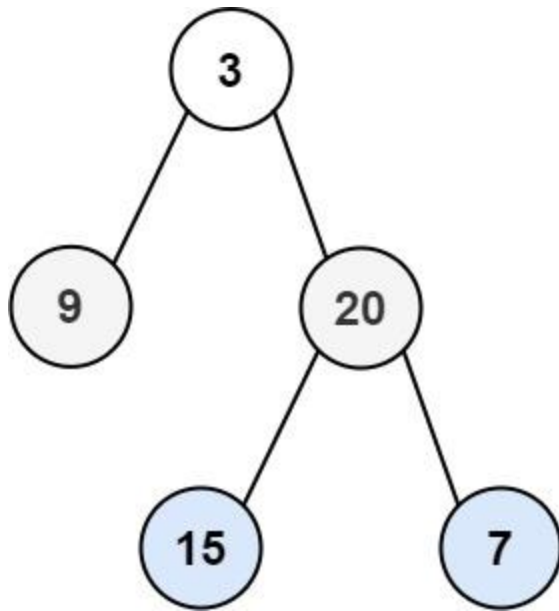
Input: root = [3,9,20,null,null,15,7]
Output: [[3],[20,9],[15,7]]

Example 2:

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

Example 3:

Input: root = []
Output: []

C :

```c
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
```

```java
class Solution {
    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        List<List<Integer>> levels = new ArrayList<List<Integer>>();
        if (root == null) return levels;

        Queue<TreeNode> queue = new LinkedList<TreeNode>();
        queue.add(root);
        int level = 0;
        while ( !queue.isEmpty() ) {
            // start the current level
            levels.add(new ArrayList<Integer>());

            // number of elements in the current level
            int level_length = queue.size();

            for(int i = 0; i < level_length; ++i) {
                TreeNode node = queue.remove();

                // fulfill the current level
                levels.get(level).add(node.val);

                // add child nodes of the current level
                // in the queue for the next level
                if (node.left != null) queue.add(node.left);
                if (node.right != null) queue.add(node.right);
            }
            // go to next level
            level++;
        }
        for (int i = 0; i < levels.size(); i++) {
            if (i % 2 == 1) {
                Collections.reverse(levels.get(i));
            }
        }
        return levels;
    }
}
```

1.64 Find Minimum In Rotated Sorted Array

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.

C:
```c
int findMin(int* nums, int numsSize) {
    // If the list has just one element then return that element.
    if (numsSize == 1) {
        return nums[0];
    }

    // Initializing left and right pointers.
    int left = 0, right = numsSize - 1;

    // If the last element is greater than the first element, then there is no rotation.
    // Already sorted array. Hence the smallest element is the first element. A[0]
    if (nums[right] > nums[0]) {
        return nums[0];
    }

    // Binary search
    while (left <= right) {
        // Find the mid element
        int mid = left + (right - left) / 2;

        // If the mid element is greater than its next element, then mid+1 element is
the smallest.
        // This point would be the point of change. From higher to lower value.
        if (nums[mid] > nums[mid + 1]) {
            return nums[mid + 1];
```

```
        }

        // If the mid element is lesser than its previous element, then mid element is
the smallest.
        if (nums[mid - 1] > nums[mid]) {
            return nums[mid];
        }

        // If the mid element's value is greater than the 0th element, this means
        // the least value is still somewhere to the right, as we are still dealing with
        // elements greater than nums[0].
        if (nums[mid] > nums[0]) {
            left = mid + 1;
        } else {
            // If nums[0] is greater than the mid value, then this means the smallest
value
            // is somewhere to the left.
            right = mid - 1;
        }
    }
    return INT_MAX;
}
```

1.65 Binary Search

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

C:

```
int search(int* nums, int numsSize, int target) {
        int left = 0;
    int right = numsSize - 1;
```

```
    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (nums[mid] == target) {
            return mid; // Target found
        } else if (nums[mid] < target) {
            left = mid + 1; // Target may be in the right half
        } else {
            right = mid - 1; // Target may be in the left half
        }
    }
return -1;
}
```

1.66 Construct Binary Tree From PreOrder And Inorder Traversal

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*.
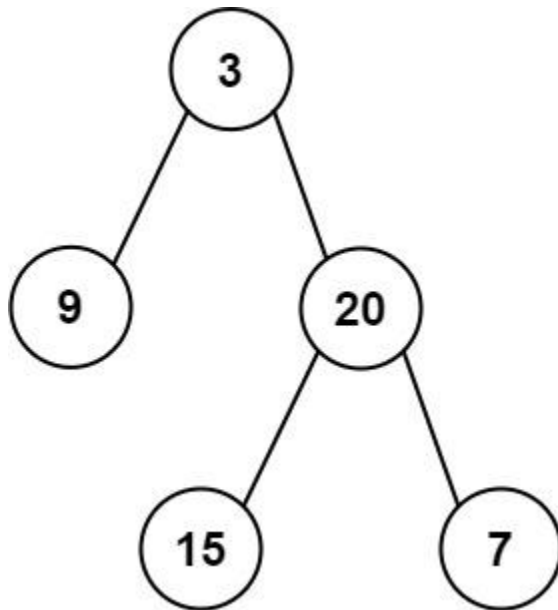
Example 1:

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]

```c
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */
struct TreeNode* buildTreeHelper(int* preorder, int preStart, int preEnd, int* inorder,
int inStart, int inEnd) {
    if (preStart > preEnd || inStart > inEnd) {
        return NULL;
    }

    // The first element in preorder is the root of the current subtree
    struct TreeNode* root = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    root->val = preorder[preStart];
    root->left = root->right = NULL;

    // Find the index of the root in inorder
    int rootIndex;
    for (rootIndex = inStart; rootIndex <= inEnd; rootIndex++) {
        if (inorder[rootIndex] == root->val) {
            break;
        }
    }

    // Calculate the number of elements in the left subtree
    int leftSubtreeSize = rootIndex - inStart;

    // Recursively build the left and right subtrees
    root->left = buildTreeHelper(preorder, preStart + 1, preStart + leftSubtreeSize,
inorder, inStart, rootIndex - 1);
    root->right = buildTreeHelper(preorder, preStart + leftSubtreeSize + 1, preEnd,
inorder, rootIndex + 1, inEnd);

    return root;
}

struct TreeNode* buildTree(int* preorder, int preorderSize, int* inorder, int
inorderSize) {
    return buildTreeHelper(preorder, 0, preorderSize - 1, inorder, 0, inorderSize - 1);
}
```

1.67 Remove Nth Node From End Of List

Given the head of a linked list, remove the nth node from the end of the list and return its head.

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]

C:

```c
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
struct ListNode* removeNthFromEnd(struct ListNode* head, int n){
    struct ListNode *curr = head;
    struct ListNode *d = head;

    for (int i = 0; i < n; i++) {
        curr = curr->next;
    }
    if (curr == NULL) return head->next;
```

```c
    while (curr->next != NULL) {
        d = d->next;
        curr = curr->next;
    }
    struct ListNode *del = d->next;
    d->next = d->next->next;
    free(del);
    return head;
    /*
        struct ListNode *fast = head, *slow = head;
        for (int i = 0; i < n; i++) fast = fast->next;
        if (fast == NULL) return head->next;
        while (fast->next) fast = fast->next, slow = slow->next;
        slow->next = slow->next->next;
        return head;
    */
}
```

1.68   Jump Game

You are given an integer array nums. You are initially positioned at the array's **first index**, and each element in the array represents your maximum jump length at that position.

Return true *if you can reach the last index, or* false *otherwise*.

Example 1:

Input: nums = [2,3,1,1,4]
Output: true
Explanation: Jump 1 step from index 0 to 1, then 3 steps to the last index.

Example 2:

Input: nums = [3,2,1,0,4]
Output: false
Explanation: You will always arrive at index 3 no matter what. Its maximum jump length is 0, which makes it impossible to reach the last index.

```java
class Solution {
    public boolean canJump(int[] nums) {
        int target = nums.length-1;
        for (int i = nums.length-2; i >= 0; i--) {
            if (i+nums[i] >= target) target = i;
        }
        return (target == 0);
    }
}
```
```c
bool canJump(int* nums, int numsSize) {
    int maxReach = 0;
```

```
    for (int i = 0; i < numsSize; i++) {
        // If the current index is beyond the furthest position we can reach
        if (i > maxReach) {
            return false;
        }

        // Update the furthest position we can reach
        maxReach = (i + nums[i]) > maxReach ? (i + nums[i]) : maxReach;

        // If we can reach the last index, return true
        if (maxReach >= numsSize - 1) {
            return true;
        }
    }

    return false;
}
```

1.69   Insert Delete GetRandom O(1)

Implement the RandomizedSet class:

- RandomizedSet() Initializes the RandomizedSet object.
- bool insert(int val) Inserts an item val into the set if not present. Returns true if the item was not present, false otherwise.
- bool remove(int val) Removes an item val from the set if present. Returns true if the item was present, false otherwise.
- int getRandom() Returns a random element from the current set of elements (it's guaranteed that at least one element exists when this method is called). Each element must have the **same probability** of being returned.

You must implement the functions of the class such that each function works in **average** O(1) time complexity.

Example 1:

Input
["RandomizedSet", "insert", "remove", "insert", "getRandom", "remove", "insert", "getRandom"]
[[], [1], [2], [2], [], [1], [2], []]
Output
[null, true, false, true, 2, true, false, 2]

Explanation
RandomizedSet randomizedSet = new RandomizedSet();
randomizedSet.insert(1); // Inserts 1 to the set. Returns true as 1 was inserted successfully.
randomizedSet.remove(2); // Returns false as 2 does not exist in the set.
randomizedSet.insert(2); // Inserts 2 to the set, returns true. Set now contains [1,2].

```c
// Define the structure for RandomizedSet
typedef struct {
    int* dict;  // Map (dictionary) to store the index of each value
    int* list;  // List to store the elements
    int size;   // Current size of the list
    int capacity;  // Capacity of the list
} RandomizedSet;

// Function to initialize RandomizedSet
RandomizedSet* randomizedSetCreate() {
    RandomizedSet* set = (RandomizedSet*)malloc(sizeof(RandomizedSet));
    set->dict = (int*)calloc(1000, sizeof(int));  // Assuming values are in the range
[0, 999]
    set->list = (int*)malloc(sizeof(int) * 1000);  // Assuming initial capacity is 1000
    set->size = 0;
    set->capacity = 1000;
    return set;
}

// Function to insert a value into RandomizedSet
bool randomizedSetInsert(RandomizedSet* obj, int val) {
    if (obj->dict[val] != 0) {
        return false;  // Value already exists
    }

    if (obj->size == obj->capacity) {
        // Resize the list if needed
        obj->capacity *= 2;
        obj->list = realloc(obj->list, sizeof(int) * obj->capacity);
    }

    obj->dict[val] = obj->size + 1;  // Adding 1 to differentiate from zero (0 indicates
not present)
    obj->list[obj->size] = val;
    obj->size++;
    return true;
}

// Function to remove a value from RandomizedSet
bool randomizedSetRemove(RandomizedSet* obj, int val) {
    if (obj->dict[val] == 0) {
        return false;  // Value not present
    }

    int lastElement = obj->list[obj->size - 1];
    int idx = obj->dict[val] - 1;  // Subtracting 1 to get the actual index
```

```
        obj->list[idx] = lastElement;
        obj->dict[lastElement] = idx + 1;   // Update the index of the swapped value
        obj->size--;
        obj->dict[val] = 0;   // Marking the value as not present
        return true;
}

// Function to get a random element from RandomizedSet
int randomizedSetGetRandom(RandomizedSet* obj) {
        int randomIndex = rand() % obj->size;
        return obj->list[randomIndex];
}

// Function to free the memory used by RandomizedSet
void randomizedSetFree(RandomizedSet* obj) {
        free(obj->dict);
        free(obj->list);
        free(obj);
}

/**
 * Your RandomizedSet struct will be instantiated and called as such:
 * RandomizedSet* obj = randomizedSetCreate();
 * bool param_1 = randomizedSetInsert(obj, val);

 * bool param_2 = randomizedSetRemove(obj, val);

 * int param_3 = randomizedSetGetRandom(obj);

 * randomizedSetFree(obj);
 */
```

1.70   Daily Temperatures

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

Example 1:

Input: temperatures = [73,74,75,71,69,72,76,73]
Output: [1,1,4,2,1,1,0,0]

Example 2:

Input: temperatures = [30,40,50,60]
Output: [1,1,1,0]

Input: temperatures = [30,60,90]
Output: [1,1,0]

```c
/**
 * Note: The returned array must be malloced, assume caller calls free().
 */

int* dailyTemperatures(int* temperatures, int temperaturesSize, int* returnSize) {
    int* result = (int*)malloc(sizeof(int) * temperaturesSize);
    *returnSize = temperaturesSize;

    int* stack = (int*)malloc(sizeof(int) * temperaturesSize);
    int top = -1;

    for (int i = 0; i < temperaturesSize; i++) {
        // Check if the current temperature is greater than the temperature at the top
of the stack
        while (top >= 0 && temperatures[i] > temperatures[stack[top]]) {
            int prevIndex = stack[top--];
            result[prevIndex] = i - prevIndex;
        }

        // Push the current index onto the stack
        stack[++top] = i;
    }

    // Remaining elements in the stack have no warmer days
    while (top >= 0) {
        result[stack[top--]] = 0;
    }

    free(stack);
    return result;
}
```

1.71   Reverse Integer

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
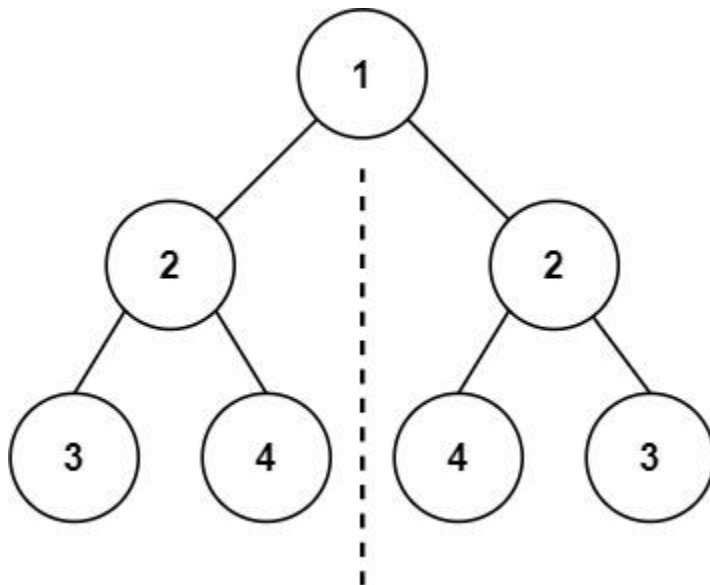
```java
class Solution {
    public int reverse(int x) {
        int rev = 0;
        while (x != 0) {
            int pop = x % 10;
            x /= 10;
            if (rev > Integer.MAX_VALUE/10 || (rev == Integer.MAX_VALUE / 10 && pop > 7)) return 0;
            if (rev < Integer.MIN_VALUE/10 || (rev == Integer.MIN_VALUE / 10 && pop < -8)) return 0;
            rev = rev * 10 + pop;
        }
        return rev;
    }
}
```
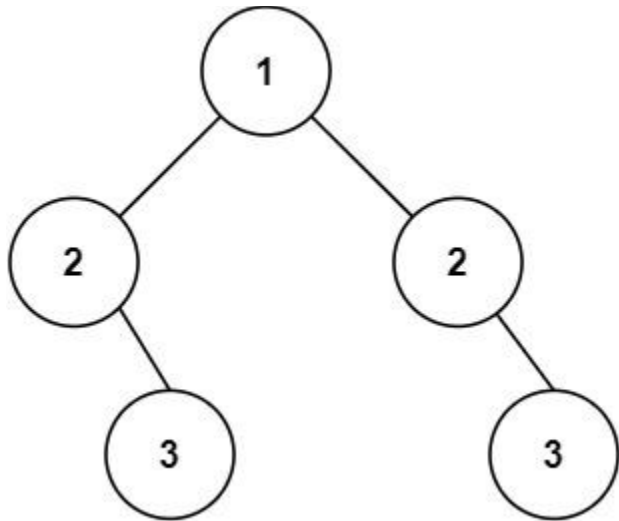
1.72   Symmetric Tree

Given the root of a binary tree, check whether it is a mirror of itself (i.e., symmetric around its center).

Example 1:

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

Example 2:

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

```c
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */
bool isSymmetric(struct TreeNode* root){
 if (root == NULL) {
        return true;
    }

    struct TreeNode* stack[200];
    int top = -1;
    stack[++top] = root;
    stack[++top] = root;

    while (top >= 0) {
        struct TreeNode* left = stack[top--];
        struct TreeNode* right = stack[top--];

        if (left == NULL && right == NULL) {
            continue;
        }
```

```
        if (left == NULL || right == NULL) {
            return false;
        }

        if (left->val != right->val) {
            return false;
        }

        stack[++top] = left->left;
        stack[++top] = right->right;
        stack[++top] = left->right;
        stack[++top] = right->left;
    }

    return true;
}
```
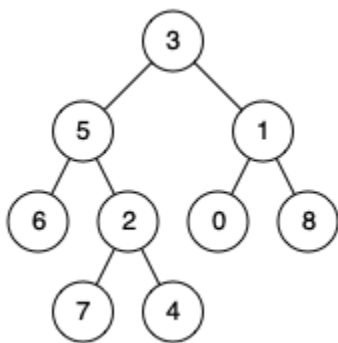## 1.73   Lowest Common Ancestor Of A binary tree

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)."
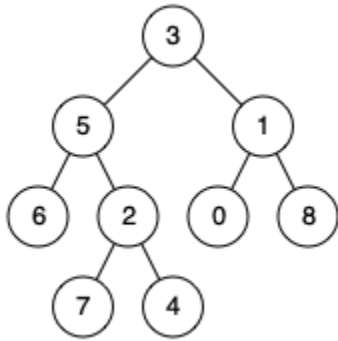
Example 1:

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 LCA definition.	

Example 3:

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

```c
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */
struct TreeNode* lowestCommonAncestor(struct TreeNode* root, struct TreeNode* p, struct TreeNode* q) {
    // Base case: If the current node is null or matches either p or q, return the
current node
    if (!root || root == p || root == q) {
        return root;
    }

    // Recursive calls to find the LCA in the left and right subtrees
    struct TreeNode* leftLCA = lowestCommonAncestor(root->left, p, q);
    struct TreeNode* rightLCA = lowestCommonAncestor(root->right, p, q);

    // If both left and right subtrees have a result, the current node is the LCA
    if (leftLCA && rightLCA) {
        return root;
    }

    // If only one subtree has a result, propagate it upward
    return (leftLCA != NULL) ? leftLCA : rightLCA;
}
```

1.74 Merge K Sorted Lists

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 list:
1->1->2->3->4->4->5->6

Example 2:

Input: lists = []
Output: []

Example 3:

Input: lists = [[]]
Output: []

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
```

```c
struct ListNode* mergeKLists(struct ListNode** lists, int listsSize){
    int i = 0;
    int count = 0;
    int arr[10000];
    int temp = 0;
    int j = 0;

    for (i = 0; i < listsSize; i++) {
        while(lists[i]!=NULL) {
            arr[count] = lists[i]->val;
            count++;
            lists[i] = lists[i]->next;
        }
    }
    if (count == 0) return NULL;
    for(i = 0 ; i < count; i++) {
        for (j = i+1; j < count; j++) {
            if (arr[i] > arr[j]) {
                temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }
    struct ListNode *result = (struct ListNode*)malloc(sizeof(struct ListNode));
    struct ListNode *head;
    head = result;

    for(i = 0 ; i < count; i++) {

        result->val = arr[i];
        result->next = NULL;

        if (i!=count-1) {
            result->next = (struct ListNode*)malloc(sizeof(struct ListNode));
            result = result->next;

        }
        else {
            result->next = NULL;
        }
    }

    return head;
}
```
1.75   SubSets

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]]

```c
// Function to count the number of set bits in an integer
int countSetBits(int n) {
    int count = 0;
    while (n) {
        count += n & 1;
        n >>= 1;
    }
    return count;
}

int** subsets(int* nums, int numsSize, int* returnSize, int** returnColumnSizes) {
    // Calculate the total number of subsets (2^n)
    int totalSubsets = 1 << numsSize;

    // Allocate memory for the result array
    int** result = (int**)malloc(totalSubsets * sizeof(int*));
    *returnColumnSizes = (int*)malloc(totalSubsets * sizeof(int));
    *returnSize = 0;

    // Iterate through all numbers from 0 to 2^n - 1
    for (int i = 0; i < totalSubsets; i++) {
        // Calculate the size of the current subset
        int subsetSize = countSetBits(i);

        // Allocate memory for the current subset
        result[*returnSize] = (int*)malloc(subsetSize * sizeof(int));
        (*returnColumnSizes)[*returnSize] = subsetSize;

        // Fill in the subset based on set bits in the binary representation of i
        int index = 0;
        for (int j = 0; j < numsSize; j++) {
            if ((i >> j) & 1) {
                result[*returnSize][index++] = nums[j];
```

```
            }
        }

        (*returnSize)++;
    }

    return result;
}
```

1.76   Can Place Flowers

You have a long flowerbed in which some of the plots are planted, and some are not. However, flowers cannot be planted in **adjacent** plots.

Given an integer array flowerbed containing 0's and 1's, where 0 means empty and 1 means not empty, and an integer n, return true *if* n *new flowers can be planted in the* flowerbed *without violating the no-adjacent-flowers rule and* false *otherwise*.

Example 1:

Input: flowerbed = [1,0,0,0,1], n = 1
Output: true

Example 2:

Input: flowerbed = [1,0,0,0,1], n = 2
Output: false

```
bool canPlaceFlowers(int* flowerbed, int flowerbedSize, int n) {
    int count = 0;
    if (flowerbedSize == 0) return false;
    for (int i = 0; i < flowerbedSize; i++) {
        if (flowerbed[i] == 0) {
            if (i == 0) {
                if (i == flowerbedSize-1) {
                    count++;
                    break;
                }
                if (flowerbed[i+1] != 1) {
                    flowerbed[i] = 1;
                    count++;
                    //i++;
                }
            } else if (i != flowerbedSize-1) {
                if (flowerbed[i-1] != 1 && flowerbed[i+1] != 1) {
                    flowerbed[i] = 1;
                    count++;
                    //i++;
```

```
                }
            } else {
                if (flowerbed[i-1] != 1) {
                    flowerbed[i] = 1;
                    count++;
                    //i++;
                }
            }
        }
    }
    if (count >= n) return true;
    else return false;
}
```

1.77   Roman To Integer

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:

```c
int romanToInt(char * s){
    int result = 0;
    int i = 0;

    // Create a lookup table for Roman numerals
    char roman[7] = {'I', 'V', 'X', 'L', 'C', 'D', 'M'};
    int value[7] = {1, 5, 10, 50, 100, 500, 1000};
    int length = strlen(s);

    for (i = 0; i < length; i++) {
        // Find the corresponding value for the current Roman numeral
        int current = 0;
        for (int j = 0; j < 7; j++) {
            if (s[i] == roman[j]) {
                current = value[j];
                break;
            }
        }

        // Check if subtraction is required
        if (i < length - 1) {
            int next = 0;
            for (int j = 0; j < 7; j++) {
                if (s[i+1] == roman[j]) {
                    next = value[j];
                    break;
                }
            }

            if (current < next) {
                result += (next - current);
                i++;
                continue;
            }
        }

        // Add the current value to the result
        result += current;
    }

    return result;
}
```

1.78   Find The Duplicate Number

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

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

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

Example 1:

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

Example 2:

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

```c
int findDuplicate(int* nums, int numsSize) {
        if (nums == NULL || numsSize <= 1) {
        // Handle invalid input
        return -1;
    }

    int slow = nums[0];
    int fast = nums[0];

    // Phase 1: Find the intersection point of the two pointers
    do {
        slow = nums[slow];
        fast = nums[nums[fast]];
    } while (slow != fast);

    // Phase 2: Find the entrance to the cycle
    slow = nums[0];
    while (slow != fast) {
        slow = nums[slow];
        fast = nums[fast];
    }

    return slow;
}
```

1.79   Product Of Array Except Self

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]

```c
/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
int* productExceptSelf(int* nums, int numsSize, int* returnSize) {
    if (nums == NULL || numsSize == 0) {
        *returnSize = 0;
        return NULL;
    }

    // Initialize arrays to store products to the left and right of each element
    int* leftProducts = (int*)malloc(numsSize * sizeof(int));
    int* rightProducts = (int*)malloc(numsSize * sizeof(int));


    // Initialize return array
    int* result = (int*)malloc(numsSize * sizeof(int));


    // Calculate products to the left of each element
    int leftProduct = 1;
    for (int i = 0; i < numsSize; i++) {
        leftProducts[i] = leftProduct;
        leftProduct *= nums[i];
    }

    // Calculate products to the right of each element
    int rightProduct = 1;
    for (int i = numsSize - 1; i >= 0; i--) {
        rightProducts[i] = rightProduct;
        rightProduct *= nums[i];
    }
```

```c
    // Calculate the final result by multiplying left and right products
    for (int i = 0; i < numsSize; i++) {
        result[i] = leftProducts[i] * rightProducts[i];
    }

    // Set the return size
    *returnSize = numsSize;

    // Free allocated memory for intermediate arrays
    free(leftProducts);
    free(rightProducts);

    return result;
}
```

1.80   Coin Change

You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.

Return *the fewest number of coins that you need to make up that amount*. If that amount of money cannot be made up by any combination of the coins, return -1.

You may assume that you have an infinite number of each kind of coin.

Example 1:

Input: coins = [1,2,5], amount = 11
Output: 3
Explanation: 11 = 5 + 5 + 1

Example 2:

Input: coins = [2], amount = 3
Output: -1

Example 3:

Input: coins = [1], amount = 0
Output: 0

```c
#define min(a,b) a < b ? a:b
int coinChange(int* coins, int coinsSize, int amount) {
    // Create an array to store the minimum number of coins needed for each amount
    int dp[amount + 1];
```

```
    // Initialize the array with a value larger than the maximum possible number of
coins
    for (int i = 0; i <= amount; i++) {
        dp[i] = INT_MAX;
    }

    // Base case: 0 coins needed for amount 0
    dp[0] = 0;

    // Iterate through each coin denomination
    for (int i = 0; i < coinsSize; i++) {
        // Update dp array for each possible amount
        for (int j = coins[i]; j <= amount; j++) {
            // If using the current coin leads to a smaller number of coins needed,
update dp
            if (dp[j - coins[i]] != INT_MAX) {
                dp[j] = min(dp[j], dp[j - coins[i]] + 1);
            }
        }
    }

    // If dp[amount] is still INT_MAX, no valid combination was found
    return (dp[amount] == INT_MAX) ? -1 : dp[amount];
}
```

1.81   Contains Duplicate

Given an integer array nums, return true if any value appears **at least twice** in the array, and return false if every element is distinct.

Example 1:

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

Example 2:

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

Example 3:

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

```
bool containsDuplicate(int* nums, int numsSize) {
    // Create a hash set using an array of flags
    bool* set = (bool*)malloc(100000 * sizeof(bool));
```

```c
    // Initialize the set to false
    for (int i = 0; i < 100000; i++) {
        set[i] = false;
    }

    // Iterate through the array
    for (int i = 0; i < numsSize; i++) {
        // If the element is already in the set, it's a duplicate
        if (set[nums[i]]) {
            free(set);
            return true;
        }

        // Mark the element as seen in the set
        set[nums[i]] = true;
    }

    // No duplicates found
    free(set);
    return false;
}
```

1.82 Search A 2D Matrix

You are given an m x 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

```c
bool searchMatrix(int** matrix, int matrixSize, int* matrixColSize, int target){
    int row = matrixSize - 1;
    int col = 0;

    while (row >= 0 && col < *matrixColSize) {
        if (matrix[row][col] > target) {
            row--;
        } else if (matrix[row][col] < target) {
            col++;
        } else { //a found it
            return true;
        }
    }
```

```
        }

        return false;

}
```

1.83   Validate Binary Search Tree

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

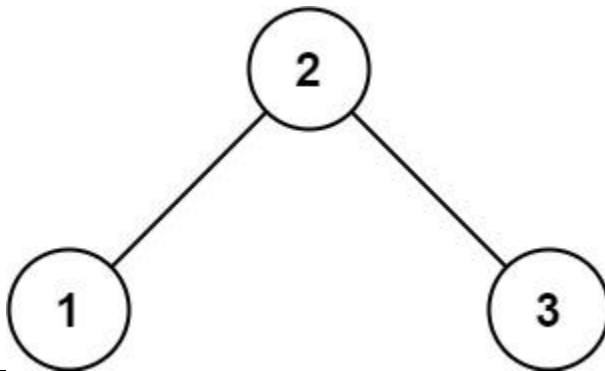A **valid BST** is defined as follows:

- The left

  subtree

   of a node contains only nodes with keys **less than** the node's key.

- The right subtree of a node contains only nodes with keys **greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.
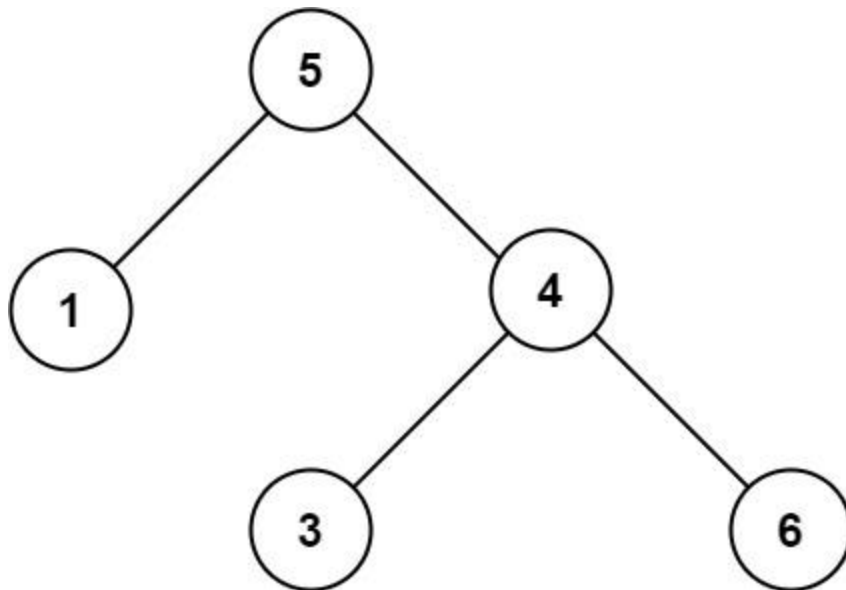
Example 1:

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.

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */
bool isValidBSTHelper(struct TreeNode* node, int *min, int *max) {
    if (node == NULL) return true;
    if ((min != NULL && node->val <= *min) || (max != NULL && node->val >= *max)) return
false;
    return isValidBSTHelper(node->left,min,&node->val) && isValidBSTHelper(node->right,
&node->val, max);
}
bool isValidBST(struct TreeNode* root){
    return isValidBSTHelper(root,NULL,NULL);
}
```

1.84   Plus One

You are given a **large integer** represented as an integer array digits, where each digits[i] is the ith digit of the integer. The digits are ordered from most significant to least significant in left-to-right order. The large integer does not contain any leading 0's.

Increment the large integer by one and return *the resulting array of digits*.

Example 1:

Input: digits = [1,2,3]
Output: [1,2,4]
Explanation: The array represents the integer 123.
Incrementing by one gives 123 + 1 = 124.
Thus, the result should be [1,2,4].

Example 2:

Input: digits = [4,3,2,1]
Output: [4,3,2,2]
Explanation: The array represents the integer 4321.
Incrementing by one gives 4321 + 1 = 4322.
Thus, the result should be [4,3,2,2].

Example 3:

Input: digits = [9]
Output: [1,0]
Explanation: The array represents the integer 9.
Incrementing by one gives 9 + 1 = 10.
Thus, the result should be [1,0].

```c
/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
int* plusOne(int* digits, int digitsSize, int* returnSize) {
    int i = 0;
    int *return_arr = NULL;
    for (i = digitsSize-1; i >= 0; i-- ) {
        if (digits[i] == 9) digits[i] = 0;
        else {
            digits[i]+=1;
            *returnSize = digitsSize;
            return digits;
        }
    }
    return_arr = (int*)malloc((digitsSize+1)*sizeof(int));
    return_arr[0] = 1;
    for (i = 0; i <= digitsSize-1; i++ ) {
      return_arr[i+1] = digits[i];
    }
    *returnSize = digitsSize+1;
    return return_arr;
}
```

1.85 Majority Element

Given an array nums of size n, return *the majority element*.

The majority element is the element that appears more than ⌊n / 2⌋ 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

```c
int majorityElement(int* nums, int numsSize) {
    int candidate = 0;
    int count = 0;
    for (int i = 0; i < numsSize; i++) {
        if (count == 0) candidate = nums[i];
        count+=(nums[i] == candidate) ? 1:-1;
    }
    return candidate;
}
```

1.86 Valid Palindrome

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.

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.

```c
bool isPalindrome(char* s) {
    int start = 0;
    int end = strlen(s);
    while (start < end) {
        if (!isalnum(s[start])) {
            start++;
            continue;
        }
        if (!isalnum(s[end])) {
            end--;
            continue;
        }
        if (tolower(s[start]) != tolower(s[end])) return false;
        else {
            start++;
            end--;
        }
    }
    return true;
}
```

1.87   House Robber

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given an integer array nums representing the amount of money of each house, return *the maximum amount of money you can rob tonight* **without alerting the police**.

Example 1:

Input: nums = [1,2,3,1]
Output: 4
Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).
Total amount you can rob = 1 + 3 = 4.

Example 2:

Input: nums = [2,7,9,3,1]
Output: 12

```c
int max(int a, int b) {
    return (a > b) ? a:b;
}
int rob(int* nums, int numsSize){

    int i = 0;
    if (numsSize == 0) return 0;
    if (numsSize == 1) return nums[0];
    int* result = malloc(numsSize * sizeof(int));
    result[0] = nums[0];
    result[1] = max(nums[0],nums[1]);
    for (i = 2; i < numsSize; i++) {
        result[i] = max(result[i-1],result[i-2]+nums[i]);
    }
    return result[numsSize-1];
}
```

1.88 Next Greater Element

The **next greater element** of some element x in an array is the **first greater** element that is **to the right** of x in the same array.

You are given two **distinct 0-indexed** integer arrays nums1 and nums2, where nums1 is a subset of nums2.

For each 0 <= i < nums1.length, find the index j such that nums1[i] == nums2[j] and determine the **next greater element** of nums2[j] in nums2. If there is no next greater element, then the answer for this query is -1.

Return *an array* ans *of length* nums1.length *such that* ans[i] *is the **next greater element** as described above.*

Example 1:

Input: nums1 = [4,1,2], nums2 = [1,3,4,2]
Output: [-1,3,-1]
Explanation: The next greater element for each value of nums1 is as follows:
- 4 is underlined in nums2 = [1,3,4,2]. There is no next greater element, so the answer is -1.
- 1 is underlined in nums2 = [1,3,4,2]. The next greater element is 3.
- 2 is underlined in nums2 = [1,3,4,2]. There is no next greater element, so the answer is -1.

Example 2:

```java
public class Solution {
    public int[] nextGreaterElement(int[] nums1, int[] nums2) {
        Stack<Integer> stack = new Stack<>();
        HashMap<Integer, Integer> map = new HashMap<>();

        for (int i = 0; i < nums2.length; i++) {
            while (!stack.empty() && nums2[i] > stack.peek())
                map.put(stack.pop(), nums2[i]);
            stack.push(nums2[i]);
        }

        while (!stack.empty())
            map.put(stack.pop(), -1);

        int[] res = new int[nums1.length];
        for (int i = 0; i < nums1.length; i++) {
            res[i] = map.get(nums1[i]);
        }

        return res;
    }
}
```

1.89   Shuffle An Array

Given an integer array nums, design an algorithm to randomly shuffle the array. All permutations of the array should be **equally likely** as a result of the shuffling.

Implement the Solution class:

- Solution(int[] nums) Initializes the object with the integer array nums.
- int[] reset() Resets the array to its original configuration and returns it.
- int[] shuffle() Returns a random shuffling of the array.

Example 1:

Input

["Solution", "shuffle", "reset", "shuffle"]

[[1, 2, 3]], [], [], []]

C:
```c
#include <stdlib.h>
#include <string.h>

typedef struct {
    int* original;
    int* copy;
    int* temp;
    int length;
} Solution;

void swap(int* arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

Solution* solutionCreate(int* nums, int numsSize) {
    Solution* obj = (Solution*)malloc(sizeof(Solution));
    obj->original = (int*)malloc(sizeof(int) * numsSize);
    obj->copy = (int*)malloc(sizeof(int) * numsSize);
    obj->temp = (int*)malloc(sizeof(int) * numsSize);
    obj->length = numsSize;

    memcpy(obj->original, nums, sizeof(int) * numsSize);
    memcpy(obj->copy, nums, sizeof(int) * numsSize);
```

```
        return obj;
}

int* solutionReset(Solution* obj, int* returnSize) {
    memcpy(obj->copy, obj->original, sizeof(int) * obj->length);
    *returnSize = obj->length;
    return obj->original;
}

int* solutionShuffle(Solution* obj, int* returnSize) {
    memcpy(obj->temp, obj->copy, sizeof(int) * obj->length);

    for (int i = 0; i < obj->length; i++) {
        int ra = obj->length - i;
        int j = i + rand() % ra;
        swap(obj->temp, i, j);
    }

    *returnSize = obj->length;
    return obj->temp;
}

void solutionFree(Solution* obj) {
    free(obj->original);
    free(obj->copy);
    free(obj->temp);
    free(obj);
}
```

1.90   MinStack

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

C:
```c
typedef struct {
    int* stack;
    int* minStack;
    int top;
    int minTop;
    int capacity;
} MinStack;

MinStack* minStackCreate() {
    MinStack* stack = (MinStack*)malloc(sizeof(MinStack));
    stack->stack = (int*)malloc(sizeof(int));
    stack->minStack = (int*)malloc(sizeof(int));
    stack->top = -1;
    stack->minTop = -1;
    stack->capacity = 1;
    return stack;
}
```

```
void minStackPush(MinStack* obj, int val) {
    if (obj->top == obj->capacity - 1) {
        obj->capacity *= 2;
        obj->stack = (int*)realloc(obj->stack, sizeof(int) * obj->capacity);
    }
    obj->stack[++obj->top] = val;

    if (obj->minTop == -1 || val <= obj->minStack[obj->minTop]) {
        if (obj->minTop == obj->capacity - 1) {
            obj->minStack = (int*)realloc(obj->minStack, sizeof(int) * obj->capacity);
        }
        obj->minStack[++obj->minTop] = val;
    }
}

void minStackPop(MinStack* obj) {
    if (obj->stack[obj->top] == obj->minStack[obj->minTop]) {
        obj->minTop--;
    }
    obj->top--;
}

int minStackTop(MinStack* obj) {
    return obj->stack[obj->top];
}

int minStackGetMin(MinStack* obj) {
    return obj->minStack[obj->minTop];
}

void minStackFree(MinStack* obj) {
    free(obj->stack);
    free(obj->minStack);
    free(obj);
}
```

1.91   Sum Of Two Integers

Given two integers a and b, return *the sum of the two integers without using the operators* + *and* -.

Example 1:

Example 2:

Input: a = 2, b = 3
Output: 5

```
int getSum(int a, int b){
    while (b != 0) {
        unsigned int sum = a ^ b;
        unsigned int carry = (unsigned int)(a & b) << 1;
        a = sum;
        b = carry;
    }
    return a;
}
```

1.92 Maximum Product SubArray

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.

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.

```
class Solution {
    public int maxProduct(int[] nums) {
        int current_array = nums[0];
        int maximum_array = nums[0];
        for (int i = 1; i < nums.length; i++) {
            current_array = Math.max(current_array*nums[i],nums[i]);
            maximum_array = Math.max(current_array, maximum_array);
        }
        return maximum_array;
    }}
```

1.93   Number Of 1 Bits

Write a function that takes the binary representation of an unsigned integer and returns the number of '1' bits it has (also known as the Hamming weight).

Note:

- Note that in some languages, such as Java, there is no unsigned integer type. In this case, the input will be given as a signed integer type. It should not affect your implementation, as the integer's internal binary representation is the same, whether it is signed or unsigned.
- In Java, the compiler represents the signed integers using 2's complement notation. Therefore, in **Example 3**, the input represents the signed integer. −3.

Example 1:

Input: n = 00000000000000000000000000001011
Output: 3
Explanation: The input binary string **00000000000000000000000000001011** has a total of three '1' bits.

Example 2:

Input: n = 00000000000000000000000010000000
Output: 1
Explanation: The input binary string **00000000000000000000000010000000** has a total of one '1' bit.

Example 3:

Input: n = 11111111111111111111111111111101
Output: 31
Explanation: The input binary string **11111111111111111111111111111101** has a total of thirty one '1' bits.

```
int hammingWeight(uint32_t n) {
    int count = 0;
    while (n != 0) {
        n = (n & n-1);
        count++;
    }
    return count;
}
```

1.94 Counting Bits

Given an integer n, return *an array* ans *of length* n + 1 *such that for each* i (0 <= i <= n), ans[i] *is the **number of** 1 's in the binary representation of* i.

Example 1:

Input: n = 2
Output: [0,1,1]
Explanation:
0 --> 0
1 --> 1
2 --> 10

Example 2:

Input: n = 5
Output: [0,1,1,2,1,2]
Explanation:
0 --> 0
1 --> 1
2 --> 10
3 --> 11
4 --> 100
5 --> 101

```c
/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
int countOnes(int num) {
    int count = 0;
    while (num) {
        num=  num & (num -1);
        //num >>= 1;
        count++;
    }
    return count;
}

int* countBits(int n, int* returnSize) {
    // Allocate memory for the result array
    int* result = (int*)malloc((n + 1) * sizeof(int));

    // Generate count of set bits for each number from 0 to n
    for (int i = 0; i <= n; ++i) {
        result[i] = countOnes(i);
    }

    // Set the return size
```

```
    *returnSize = n + 1;

    return result;
}
```

1.95   Missing Number

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.

```c
int missingNumber(int* nums, int numsSize){
    int sum = 0;
    int actual_sum = 0;
    int i = 0;
    for (i = 0; i < numsSize; i++) {
        sum += nums[i];
    }

    actual_sum = (numsSize*(numsSize+1))/2;
    return actual_sum - sum;
}
```

1.96   Reverse Bits

Reverse bits of a given 32 bits unsigned integer.

Note:

- Note that in some languages, such as Java, there is no unsigned integer type. In this case, both input and output will be given as a signed integer type. They should not affect your implementation, as the integer's internal binary representation is the same, whether it is signed or unsigned.
- In Java, the compiler represents the signed integers using 2's complement notation. Therefore, in **Example 2** above, the input represents the signed integer -3 and the output represents the signed integer -1073741825.

Example 1:

Input: n = 00000010100101000001111010011100
Output: 964176192 (00111001011110000010100101000000)
Explanation: The input binary string **00000010100101000001111010011100** represents the unsigned integer 43261596, so return 964176192 which its binary representation is **00111001011110000010100101000000**.

Example 2:

Input: n = 11111111111111111111111111111101
Output: 3221225471 (10111111111111111111111111111111)
Explanation: The input binary string **11111111111111111111111111111101** represents the unsigned integer 4294967293, so return 3221225471 which its binary representation is **10111111111111111111111111111111**.

```
uint32_t reverseBits(uint32_t n) {
    int result = 0;
    for (int i = 31; i >=0; i--) {
        result |= (((n >> (31-i)) & 0x1) << i);
    }
    return result;
}
```

1.97   Detect Cycle in Linked List

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**.

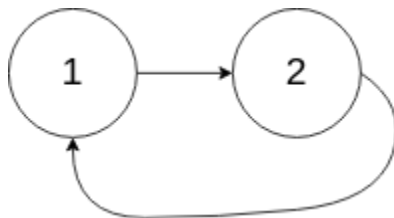Return true *if there is a cycle in the linked list*. Otherwise, return false.
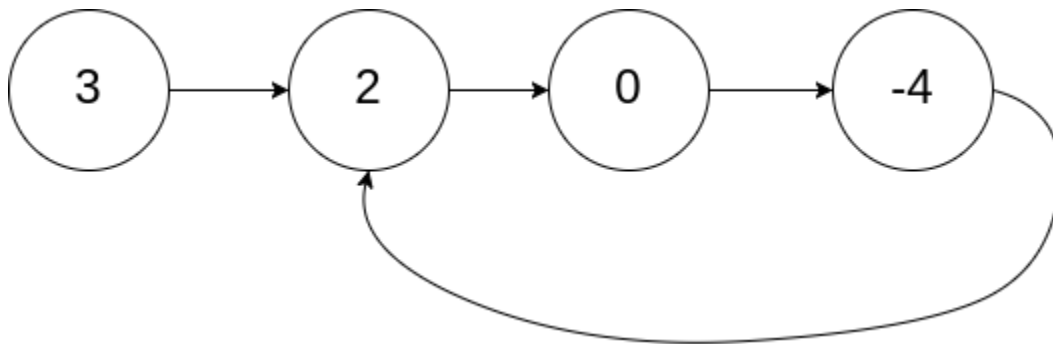
Input: head = [3,2,0,-4], pos = 1
Output: true
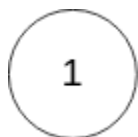**Explanation:** There is a cycle in the linked list, where the tail connects to the 1st node (0-indexed).

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.

Input: head = [1], pos = -1
Output: false
Explanation: There is no cycle in the linked list.

```c
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
bool hasCycle(struct ListNode *head) {

    if (head == NULL) return false;
    struct ListNode *slow = head;
```

```
    struct ListNode *fast = head;

    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) return true;
    }
    return false;
}
```

1.98   Reorder List

You are given the head of a singly linked-list. The list can be represented as:
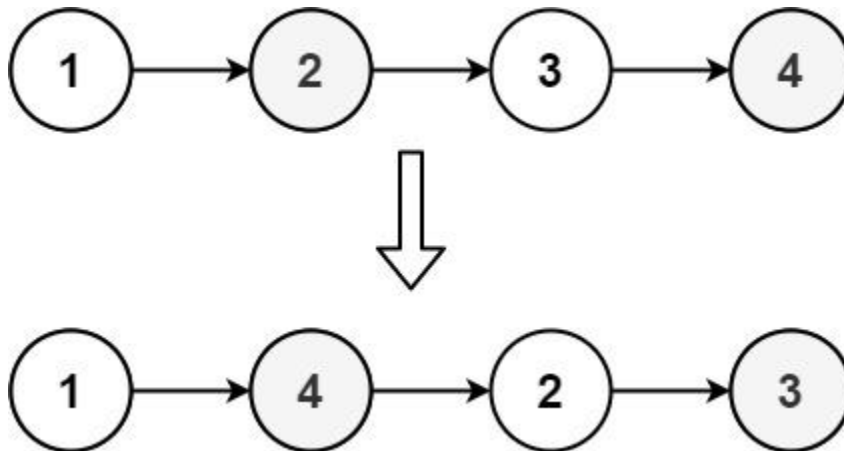
$L_0 \rightarrow L_1 \rightarrow \dots \rightarrow L_{n-1} \rightarrow L_n$

Reorder the list to be on the following form:

$L_0 \rightarrow L_n \rightarrow L_1 \rightarrow L_{n-1} \rightarrow L_2 \rightarrow L_{n-2} \rightarrow \dots$

You may not modify the values in the list's nodes. Only nodes themselves may be changed.
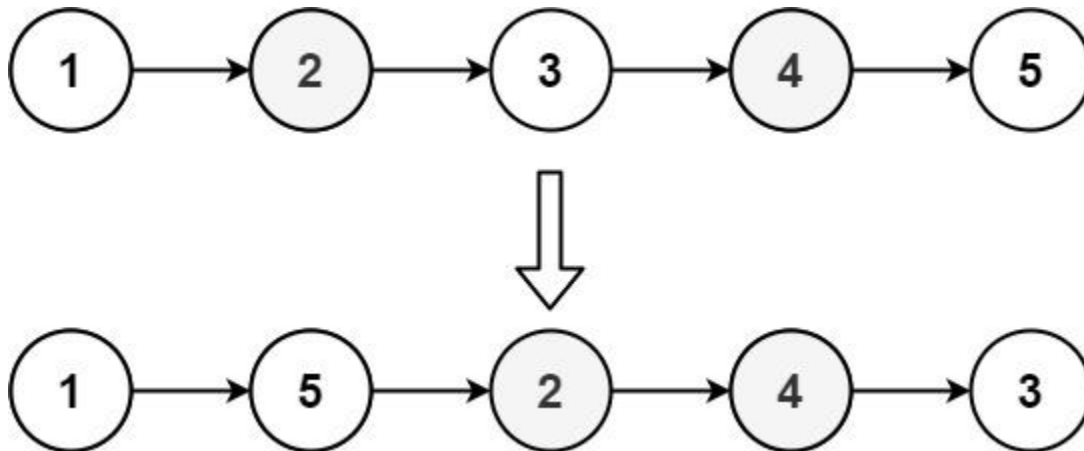
Example 1:

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:

```c
void reorderList(struct ListNode* head){
struct ListNode *stack[50000];
struct ListNode *temp=head;
int i=0;
while(temp)
{
    stack[i]=temp;
    temp=temp->next;
    i++;
}
temp=head;
struct ListNode *sec=NULL;
int k=i;
for(int j=0;j<i/2;j++)
{
 sec=temp->next;
 temp->next=stack[--k];
 temp->next->next=sec;
 temp=sec;
}
temp->next=NULL;
}
class Solution {
 public void reorderList(ListNode head) {
   if (head == null) return;

   // find the middle of linked list [Problem 876]
   // in 1->2->3->4->5->6 find 4
   ListNode slow = head, fast = head;
```

```
    while (fast != null && fast.next != null) {
      slow = slow.next;
      fast = fast.next.next;
    }

    // reverse the second part of the list [Problem 206]
    // convert 1->2->3->4->5->6 into 1->2->3->4 and 6->5->4
    // reverse the second half in-place
    ListNode prev = null, curr = slow, tmp;
    while (curr != null) {
      tmp = curr.next;

      curr.next = prev;
      prev = curr;
      curr = tmp;
    }

    // merge two sorted linked lists [Problem 21]
    // merge 1->2->3->4 and 6->5->4 into 1->6->2->5->3->4
    ListNode first = head, second = prev;
    while (second.next != null) {
      tmp = first.next;
      first.next = second;
      first = tmp;

      tmp = second.next;
      second.next = first;
      second = tmp;
    }
  }
}
```
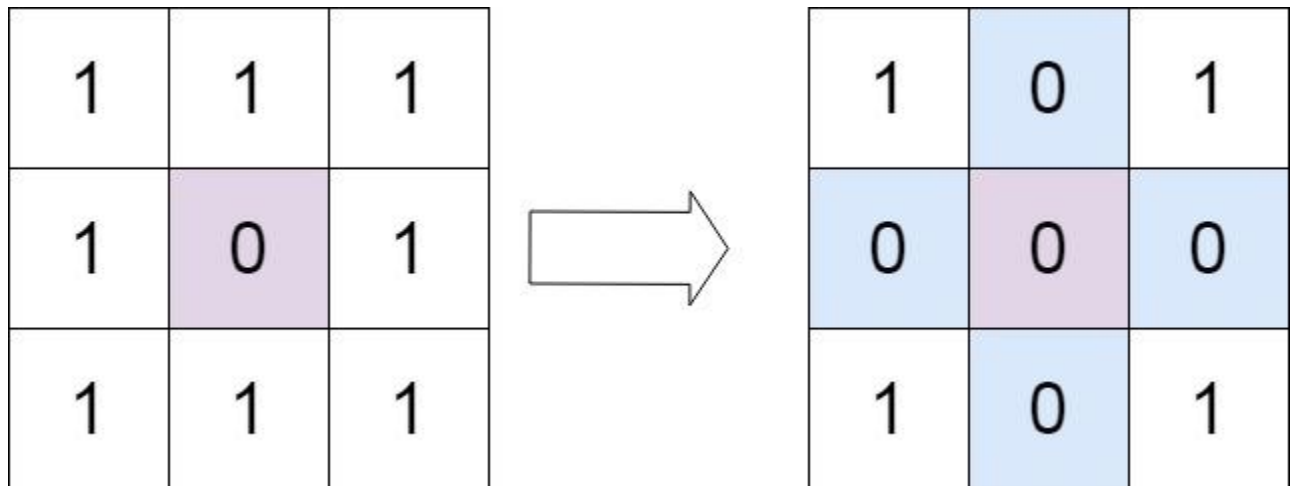
1.99   Set Matrix Zeros

Given an m x n integer matrix matrix, if an element is 0, set its entire row and column to 0's.

You must do it [in place](#).

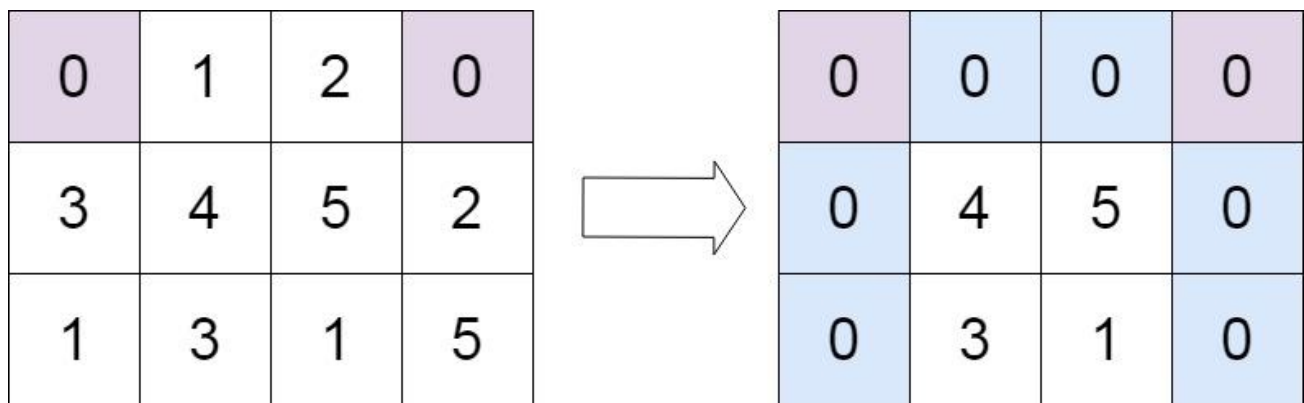**Example 1:**

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

Example 2:

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]]

```java
class Solution {
    public void setZeroes(int[][] matrix) {
        boolean first_col = false;
        boolean first_row = false;

        for (int i = 0; i < matrix.length; i++) {
            if (matrix[i][0] == 0) first_col = true;
        }

        for (int i = 0; i < matrix[0].length; i++) {
            if (matrix[0][i] == 0) first_row = true;
        }
        for (int i = 1; i < matrix.length; i++) {
            for (int j = 1; j < matrix[0].length; j++) {
                if (matrix[i][j] == 0) {
                    matrix[i][0] = 0;
                    matrix[0][j] = 0;
                }
```

```java
            }
        }
        for (int i = 1; i < matrix.length; i++) {
            if(matrix[i][0] == 0) {
                for (int j = 1; j < matrix[0].length; j++) {
                    matrix[i][j] = 0;
                }
            }
        }
        for (int j = 1; j < matrix[0].length; j++) {
            if(matrix[0][j] == 0) {
                for (int i = 1; i < matrix.length; i++) {
                    matrix[i][j] = 0;
                }
            }
        }
        if (first_row == true) {
            for (int j = 0; j < matrix[0].length; j++) {
                matrix[0][j] = 0;
            }
        }
        if (first_col == true) {
            for (int i = 0; i < matrix.length; i++) {
                matrix[i][0] = 0;
            }
        }

    }
}

/*
0 1 2 0    0 0 0 0    0 0 0 0
3 4 5 2    0 4 5 0    3 4 5 0
1 3 1 5    0 3 1 0    1 3 1 0


0 0 0 5    0 0 0 0    0 0 0 0
4 3 1 4    0 0 0 4    0 0 0 4
0 1 1 4    0 0 0 0    0 0 0 4
1 2 1 3    0 0 0 3    0 0 0 3
0 0 1 1    0 0 0 0    0 0 0 0
*/
```

1.100 Insert Intervals

You are given an array of non-overlapping intervals intervals where intervals[i] = [start, end] represent the start and the end of the i interval and intervals is sorted in ascending order by start. 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 start and intervals still does not have any overlapping intervals (merge overlapping intervals if necessary).

Return intervals *after the insertion*.

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].

```
/**
 * Return an array of arrays of size *returnSize.
 * The sizes of the arrays are returned as *returnColumnSizes array.
 * Note: Both returned array and *columnSizes array must be malloced, assume caller
calls free().
 */
/*
 * 57. Insert Interval
 *
 * You are given an array of non-overlapping intervals intervals
 * where intervals[i] = [starti, endi] represent the start and the
 * end of the ith 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.
 *
 * 0 <= intervals.length <= 104
 * intervals[i].length == 2
 * 0 <= starti <= endi <= 105
```

```
 * intervals is sorted by starti in ascending order.
 * newInterval.length == 2
 * 0 <= start <= end <= 10^5
 */

/**
 * Return an array of arrays of size *returnSize.
 * The sizes of the arrays are returned as *returnColumnSizes array.
 * Note: Both returned array and *columnSizes array must be malloced, assume caller
calls free().
 */

#define MAX(a, b) ((a) > (b) ? (a) : (b))

int cmp(const void *a, const void *b)
{
    return (*((int **)a))[0] - (*((int **)b))[0];
}

int** insert(int** intervals, int intervalsSize, int* intervalsColSize, int*
newInterval, int newIntervalSize, int* returnSize, int** returnColumnSizes){

    /*
     * Input:
     *  intervals,
     *  intervalsSize
     *  intervalsColSize
     *  newInterval
     *  newIntervalSize
     */

    int seg_start = newInterval[0], seg_end = newInterval[1];
    int **ans = (int **)malloc(sizeof(int *) * (intervalsSize + 1));
    int **tmp = (int **)malloc(sizeof(int *) * (intervalsSize + 1));
    (*returnColumnSizes) = (int *)malloc(sizeof(int) * (intervalsSize + 1));

    *returnSize = 0;

    /* Copy intervals to tmp buffer */
    for (int i = 0; i < intervalsSize; i++) {
        tmp[i] = (int *)malloc(sizeof(int) * 2);
        tmp[i][0] = intervals[i][0];
        tmp[i][1] = intervals[i][1];
    }

    /* Append newInterval to tmp buffer */
    tmp[intervalsSize] = (int *)malloc(sizeof(int) * 2);
    ans[intervalsSize] = (int *)malloc(sizeof(int) * 2);
    tmp[intervalsSize][0] = newInterval[0];
```

```c
        tmp[intervalsSize][1] = newInterval[1];

    /* Sort by start */
    qsort(tmp, intervalsSize + 1, sizeof(int *), cmp);

    seg_start = tmp[0][0];
    seg_end = tmp[0][1];

    /* Merge the interval */
    for (int i = 0; i < intervalsSize + 1; i++) {
        /* Check if segements can be merged, and store the start and end position of
merged segment */
        if (tmp[i][0] >= seg_start && tmp[i][0] <= seg_end) {
            seg_end = MAX(seg_end, tmp[i][1]);
        }
        /* Complete current segment */
        else {
            ans[*returnSize] = (int *)malloc(sizeof(int) * 2);
            ans[*returnSize][0] = seg_start;
            ans[*returnSize][1] = seg_end;
            (*returnColumnSizes)[*returnSize] = 2;
            *returnSize += 1;
            seg_start = tmp[i][0];
            seg_end = tmp[i][1];
        }

        /* Add last one segment */
        if (i == intervalsSize) {
            ans[*returnSize] = (int *)malloc(sizeof(int) * 2);
            ans[*returnSize][0] = seg_start;
            ans[*returnSize][1] = seg_end;
            (*returnColumnSizes)[*returnSize] = 2;
            *returnSize += 1;
        }
    }

    for (int i = 0; i < intervalsSize + 1; i++) {
        free(tmp[i]);
    }

    free(tmp);

    /*
     * Output:
     *   returnSize
     *   returnColumnSizes
     */

    return ans;
```

```
}
```

1.101 **Non Overlapping Intervals**

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

Example 1:

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

Example 2:

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

Example 3:

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

```c
// Interval structure definition
struct Interval {
    int start;
    int end;
};

// Function to compare intervals for sorting
int compareIntervals(const void* a, const void* b) {
    return ((struct Interval*)a)->end - ((struct Interval*)b)->end;
}

// Function to find the minimum number of intervals to remove
int eraseOverlapIntervals(int** intervals, int intervalsSize, int* intervalsColSize) {
    if (intervals == NULL || intervalsSize <= 1) {
        return 0;
    }

    // Convert 2D array to array of Interval structures
    struct Interval* intervalsArr = (struct Interval*)malloc(intervalsSize *
sizeof(struct Interval));
    for (int i = 0; i < intervalsSize; i++) {
```

```c
        intervalsArr[i].start = intervals[i][0];
        intervalsArr[i].end = intervals[i][1];
    }

    // Sort intervals by end time in ascending order
    qsort(intervalsArr, intervalsSize, sizeof(struct Interval), compareIntervals);

    int ans = 0;
    int k = INT_MIN;

    // Iterate through intervals
    for (int i = 0; i < intervalsSize; i++) {
        int x = intervalsArr[i].start;
        int y = intervalsArr[i].end;

        if (x >= k) {
            // Case 1
            k = y;
        } else {
            // Case 2
            ans++;
        }
    }

    // Free allocated memory
    free(intervalsArr);

    return ans;
}
```

1.102 Meeting Rooms

Given an array of meeting time intervals where intervals[i] = [start, end], determine if a person could attend all meetings.

Example 1:

Input: intervals = [[0,30],[5,10],[15,20]]
Output: false

Example 2:

Input: intervals = [[7,10],[2,4]]
Output: true

```c
int compareIntervals(const void* a, const void* b) {
    return (*((int**)a))[0] - (*((int**)b))[0];
}
```

```
}

int canAttendMeetings(int** intervals, int intervalsSize, int* intervalsColSize) {
    if (intervalsSize <= 1) {
        return 1;  // A person can attend all meetings if there is only one or zero
meetings.
    }

    // Sort intervals based on the start time.
    qsort(intervals, intervalsSize, sizeof(int*), compareIntervals);

    // Check if there is any overlap between consecutive intervals.
    for (int i = 1; i < intervalsSize; i++) {
        if (intervals[i - 1][1] > intervals[i][0]) {
            return 0;  // There is an overlap, so the person cannot attend all meetings.
        }
    }

    return 1;  // No overlap found, person can attend all meetings.
}
```

1.103 Longest Repeating Character Replacement

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 "AABBBBA".
The substring "BBBB" has the longest repeating letters, which is 4.
There may exists other ways to achieve this answer too.

```
int characterReplacement(char *s, int k) {
    int n = strlen(s);
    int max_count = 0;  // Maximum repeating character count in the window
    int start = 0;      // Start index of the window
```

```
    int max_length = 0; // Maximum length of the substring with repeating characters

    // Array to store the count of each character in the window
    int char_count[26] = {0};

    for (int end = 0; end < n; end++) {
        char_count[s[end] - 'A']++;
        max_count = (max_count > char_count[s[end] - 'A']) ? max_count :
char_count[s[end] - 'A'];

        // If the window size exceeds the maximum count + k, shrink the window
        if ((end - start + 1) - max_count > k) {
            char_count[s[start] - 'A']--;
            start++;
        }

        // Update the maximum length
        max_length = (max_length > (end - start + 1)) ? max_length : (end - start + 1);
    }

    return max_length;
}
```

1.104 Minimum Window SubString

Given two strings s and t of lengths m and n respectively, return *the **minimum window***

substring
 of s *such that every character in* t *(**including duplicates**) is included in the window*. If there is no such substring, return *the empty string "".*

The testcases will be generated such that the answer is **unique**.

Example 1:

Input: s = "ADOBECODEBANC", t = "ABC"
Output: "BANC"
Explanation: The minimum window substring "BANC" includes 'A', 'B', and 'C' from string t.

Example 2:

Input: s = "a", t = "a"
Output: "a"
Explanation: The entire string s is the minimum window.

Example 3:

```c
char* minWindow(char* s, char* t) {
    int sLen = strlen(s);
    int tLen = strlen(t);

    // Initialize arrays to store character frequencies
    int sFreq[128] = {0};
    int tFreq[128] = {0};

    // Count frequencies of characters in string t
    for (int i = 0; i < tLen; i++) {
        tFreq[t[i]]++;
    }

    int left = 0; // Left pointer of the sliding window
    int right = 0; // Right pointer of the sliding window
    int minLen = sLen + 1; // Initialize to a value greater than sLen
    int minStart = 0; // Start index of the minimum window

    int requiredChars = tLen; // Number of characters still required to match t

    while (right < sLen) {
        // Expand the window to the right
        if (tFreq[s[right]] > 0) {
            // This character is required
            requiredChars--;
        }
        tFreq[s[right]]--;
        right++;

        // Check if all characters from t are included in the window
        while (requiredChars == 0) {
            // Update the minimum window
            if (right - left < minLen) {
                minLen = right - left;
                minStart = left;
            }

            // Shrink the window from the left
            tFreq[s[left]]++;
            if (tFreq[s[left]] > 0) {
                // This character is no longer covered
                requiredChars++;
            }
            left++;
        }
```

```
    }

    // Check if a valid window was found
    if (minLen == sLen + 1) {
        return "";
    }

    // Allocate memory for the result and copy the substring
    char* result = malloc((minLen + 1) * sizeof(char));
    strncpy(result, s + minStart, minLen);
    result[minLen] = '\0';

    return result;
}
```

1.105 Palindromic Substrings

Given a string s, return *the number of **palindromic substrings** in it*.

A string is a **palindrome** when it reads the same backward as forward.

A **substring** is a contiguous sequence of characters within the string.

Example 1:

Input: s = "abc"
Output: 3
Explanation: Three palindromic strings: "a", "b", "c".

Example 2:

Input: s = "aaa"
Output: 6
Explanation: Six palindromic strings: "a", "a", "a", "aa", "aa", "aaa".

```
    // Function to expand around the center for odd length palindromes
    int expandAroundCenterOdd(int left, int right, int n, char *s) {
        int cnt = 0;
        while (left >= 0 && right < n && s[left] == s[right]) {
            cnt++;
            left--;
            right++;
        }
        return cnt;
    }
```

```
    // Function to expand around the center for even length palindromes
    int expandAroundCenterEven(int left, int right, int n, char *s) {
        int cnt = 0;
        while (left >= 0 && right < n && s[left] == s[right]) {
            cnt++;
            left--;
            right++;
        }
        return cnt;
    }
int countSubstrings(char *s) {
    int n = strlen(s);
    int count = 0;

    for (int i = 0; i < n; i++) {
        // Expand around the center for odd length palindromes
        count += expandAroundCenterOdd(i, i, n, s);

        // Expand around the center for even length palindromes
        count += expandAroundCenterEven(i, i + 1, n, s);
    }

    return count;
}
```

1.106 Encode And Decode Strings

Design an algorithm to encode a list of strings to a string. The encoded string is then sent over the network and is decoded back to the original list of strings.

Machine 1 (sender) has the function:

```
string encode(vector<string> strs) {
  // ... your code
  return encoded_string;
}
```

Machine 2 (receiver) has the function:

```
vector<string> decode(string s) {
  //... your code
  return strs;
}
```

So Machine 1 does:

```
string encoded_string = encode(strs);
```

and Machine 2 does:

```
vector<string> strs2 = decode(encoded_string);
```

strs2 in Machine 2 should be the same as strs in Machine 1.

Implement the encode and decode methods.

You are not allowed to solve the problem using any serialize methods (such as eval).

Example 1:

Input: dummy_input = ["Hello","World"]
Output: ["Hello","World"]
Explanation:
Machine 1:
Codec encoder = new Codec();
String msg = encoder.encode(strs);
Machine 1 ---msg---> Machine 2

Machine 2:
Codec decoder = new Codec();
String[] strs = decoder.decode(msg);

Example 2:

Input: dummy_input = [""]
Output: [""]

```c
/** Encodes a list of strings to a single string */
char* encode(char** strs, int strsSize) {
    char *ret = malloc(strsSize * 200);
    char *ret_start = ret;

    /* Build string in the following format:
        <string_size>#<string> */
    for (int i = 0; i < strsSize; i++) {
        ret += sprintf(ret, "%d#%s", (int) strlen(strs[i]), strs[i]);
    }
    printf("%s",ret_start);
    return ret_start;
}

/**
 * Decodes a single string to a list of strings.
 *
 * Return an array of size *returnSize.
 * Note: The returned array must be malloced, assume caller calls free().
 */
```

```c
char** decode(char* s, int* returnSize) {
    int i = 0;
    int j;
    char **ret = malloc(sizeof (char *) * 200);
    char len[4];
    int curr_char, str_len;
    int curr_str = 0;

    while (s[i] != '\0') {

        /* Find the length of this string */
        j = 0;
        memset(len, 0, sizeof(len));
        while (s[i] != '#') {
            len[j++] = s[i++];
        }
        str_len = atoi(len);
        i++;

        /* Extract the string */
        ret[curr_str] = malloc(str_len + 1);
        curr_char = 0;
        while (str_len > 0) {
            ret[curr_str][curr_char++] = s[i++];
            str_len--;
        }
        ret[curr_str][curr_char] = '\0';
        curr_str++;
    }
    *returnSize = curr_str;
    return ret;
}


// Your functions will be called as such:
// char* s = encode(strs, strsSize);
// decode(s, &returnSize);
```

1.107 Longest Common Subsequence

Given two strings text1 and text2, return *the length of their longest **common subsequence**.* If there is no **common subsequence**, return 0.

A **subsequence** of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

- For example, "ace" is a subsequence of "abcde".

A **common subsequence** of two strings is a subsequence that is common to both strings.

Example 1:

Input: text1 = "abcde", text2 = "ace"
Output: 3
Explanation: The longest common subsequence is "ace" and its length is 3.

Example 2:

Input: text1 = "abc", text2 = "abc"
Output: 3
Explanation: The longest common subsequence is "abc" and its length is 3.

Example 3:

Input: text1 = "abc", text2 = "def"
Output: 0
Explanation: There is no such common subsequence, so the result is 0.

```c
int max(int a, int b) {
    return a > b ? a : b;
}

int longestCommonSubsequence(char* text1, char* text2) {
    int len1 = strlen(text1);
    int len2 = strlen(text2);

    // Create a 2D array to store the lengths of LCS for subproblems
    int dp[len1 + 1][len2 + 1];

    // Initialize the first row and column to 0
    for (int i = 0; i <= len1; i++) {
        for (int j = 0; j <= len2; j++) {
            if (i == 0 || j == 0) {
                dp[i][j] = 0;
            }
        }
    }

    // Fill the dp array to find the length of LCS
    for (int i = 1; i <= len1; i++) {
        for (int j = 1; j <= len2; j++) {
            if (text1[i - 1] == text2[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
```

```
        } else {
            dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
        }
    }
}

// The length of the LCS is stored in the bottom-right cell of dp array
return dp[len1][len2];
}
```

1.108 Word Break

Given a string s and a dictionary of strings wordDict, return true if s can be segmented into a space-separated sequence of one or more dictionary words.

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

Example 1:

Input: s = "leetcode", wordDict = ["leet","code"]
Output: true
Explanation: Return true because "leetcode" can be segmented as "leet code".

Example 2:

Input: s = "applepenapple", wordDict = ["apple","pen"]
Output: true
Explanation: Return true because "applepenapple" can be segmented as "apple pen apple".
Note that you are allowed to reuse a dictionary word.

Example 3:

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

```
class Solution {
    public boolean wordBreak(String s, List<String> wordDict) {
        int n = s.length();
        Set<String> words = new HashSet<>(wordDict);
        boolean[] dp = new boolean[n + 1];
        dp[0] = true;

        for (int i = 1; i <= n; i++) {
            for (int j = 0; j < i; j++) {
```

```
                if (dp[j] && words.contains(s.substring(j, i))) {
                    dp[i] = true;
                    break;
                }
            }
        }

        return dp[n];
    }
}
```

1.109 Combination Sum IV

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

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

Example 1:

Input: nums = [1,2,3], target = 4
Output: 7
Explanation:
The possible combination ways are:
(1, 1, 1, 1)
(1, 1, 2)
(1, 2, 1)
(1, 3)
(2, 1, 1)
(2, 2)
(3, 1)
Note that different sequences are counted as different combinations.

```
class Solution {

    public int combinationSum4(int[] nums, int target) {
        // minor optimization
        // Arrays.sort(nums);
        int[] dp = new int[target + 1];
        dp[0] = 1;

        for (int combSum = 1; combSum < target + 1; ++combSum) {
            for (int num : nums) {
                if (combSum - num >= 0)
                    dp[combSum] += dp[combSum - num];
                // minor optimizaton, early stopping
                // else
                //     break;
            }
        }
```

```
        return dp[target];
    }
}
```

1.110 House Robber II

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed. All houses at this place are **arranged in a circle.** That means the first house is the neighbor of the last one. Meanwhile, adjacent houses have a security system connected, and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given an integer array nums representing the amount of money of each house, return *the maximum amount of money you can rob tonight* **without alerting the police**.

Example 1:

Input: nums = [2,3,2]
Output: 3
Explanation: You cannot rob house 1 (money = 2) and then rob house 3 (money = 2), because they are adjacent houses.

Example 2:

Input: nums = [1,2,3,1]
Output: 4
Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).
Total amount you can rob = 1 + 3 = 4.

Example 3:

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

```
class Solution {
    public int rob(int[] nums) {
        if (nums.length == 0)
            return 0;

        if (nums.length == 1)
            return nums[0];

        int max1 = rob_simple(nums, 0, nums.length - 2);
        int max2 = rob_simple(nums, 1, nums.length - 1);

        return Math.max(max1, max2);
```

```
        }

    public int rob_simple(int[] nums, int start, int end) {
        int t1 = 0;
        int t2 = 0;

        for (int i = start; i <= end; i++) {
            int temp = t1;
            int current = nums[i];
            t1 = Math.max(current + t2, t1);
            t2 = temp;
        }

        return t1;
    }
}
```

1.111 <u>Decode Ways</u>

A message containing letters from A-Z can be encoded into numbers using the following mapping:
'A' -> "1"
'B' -> "2"
...
'Z' -> "26"
To decode an encoded message, all the digits must be grouped then mapped back into letters using the reverse of the mapping above (there may be multiple ways). For example, "11106" can be mapped into:
•   "AAJF" with the grouping (1 1 10 6)
•   "KJF" with the grouping (11 10 6)
Note that the grouping (1 11 06) is invalid because "06" cannot be mapped into 'F' since "6" is different from "06".
Given a string s containing only digits, return the number of ways to decode it.
The test cases are generated so that the answer fits in a 32-bit integer.

Example 1:
Input: s = "12"
Output: 2
Explanation: "12" could be decoded as "AB" (1 2) or "L" (12).
Example 2:
Input: s = "226"
Output: 3
Explanation: "226" could be decoded as "BZ" (2 26), "VF" (22 6), or "BBF" (2 2 6).
Example 3:
Input: s = "06"
Output: 0
Explanation: "06" cannot be mapped to "F" because of the leading zero ("6" is different from "06").
```
class Solution {
    public int numDecodings(String s) {
        if (s.charAt(0) == '0') {
            return 0;
        }

        int n = s.length();
        int twoBack = 1;
        int oneBack = 1;
        for (int i = 1; i < n; i++) {
```

```
        int current = 0;
        if (s.charAt(i) != '0') {
            current = oneBack;
        }
        int twoDigit = Integer.parseInt(s.substring(i - 1, i + 1));
        if (twoDigit >= 10 && twoDigit <= 26) {
            current += twoBack;
        }

        twoBack = oneBack;
        oneBack = current;
    }
    return oneBack;
  }
}
```

1.112 **Find Median From Data Stream**

The **median** is the middle value in an ordered integer list. If the size of the list is even, there is no middle value, and the median is the mean of the two middle values.

- For example, for arr = [2,3,4], the median is 3.
- For example, for arr = [2,3], the median is (2 + 3) / 2 = 2.5.

Implement the MedianFinder class:

- MedianFinder() initializes the MedianFinder object.
- void addNum(int num) adds the integer num from the data stream to the data structure.
- double findMedian() returns the median of all elements so far. Answers within 10^{-5} of the actual answer will be accepted.

Example 1:

Input
["MedianFinder", "addNum", "addNum", "findMedian", "addNum", "findMedian"]
[[], [1], [2], [], [3], []]
Output
[null, null, null, 1.5, null, 2.0]

Explanation
MedianFinder medianFinder = new MedianFinder();
medianFinder.addNum(1);   // arr = [1]
medianFinder.addNum(2);   // arr = [1, 2]
medianFinder.findMedian(); // return 1.5 (i.e., (1 + 2) / 2)
medianFinder.addNum(3);   // arr[1, 2, 3]
medianFinder.findMedian(); // return 2.0

```
class MedianFinder {
    private PriorityQueue<Integer> small = new
PriorityQueue<>(Collections.reverseOrder());
    private PriorityQueue<Integer> large = new PriorityQueue<>();
```

```java
    private boolean even = true;
    public MedianFinder() {

    }

    public void addNum(int num) {
        if (even) {
            large.offer(num);
            small.offer(large.poll());
        } else {
            small.offer(num);
            large.offer(small.poll());
        }
        even = !even;
    }

    public double findMedian() {
        if (even)
            return (small.peek() + large.peek()) / 2.0;
        else
            return small.peek();
    }
}

/**
 * Your MedianFinder object will be instantiated and called as such:
 * MedianFinder obj = new MedianFinder();
 * obj.addNum(num);
 * double param_2 = obj.findMedian();
 */
class MedianFinder {

    ArrayList<Integer> list;

    public MedianFinder() {
        list = new ArrayList<>();
    }

    public void addNum(int num) {
        int i;
        if(list.size() > 0){
            for (i = 0; (i < list.size()  && list.get(i) < num); i++);
            list.add(i , num);
        }else{
            list.add(num);
        }
    }
```

```
    public double findMedian() {
        // System.out.println(list);
        int index = list.size()/2;
        if(list.size() % 2 == 0){
            return (double) (list.get(index) + list.get(index - 1))/2;
        }else{
            return list.get(index);
        }

    }
}
```
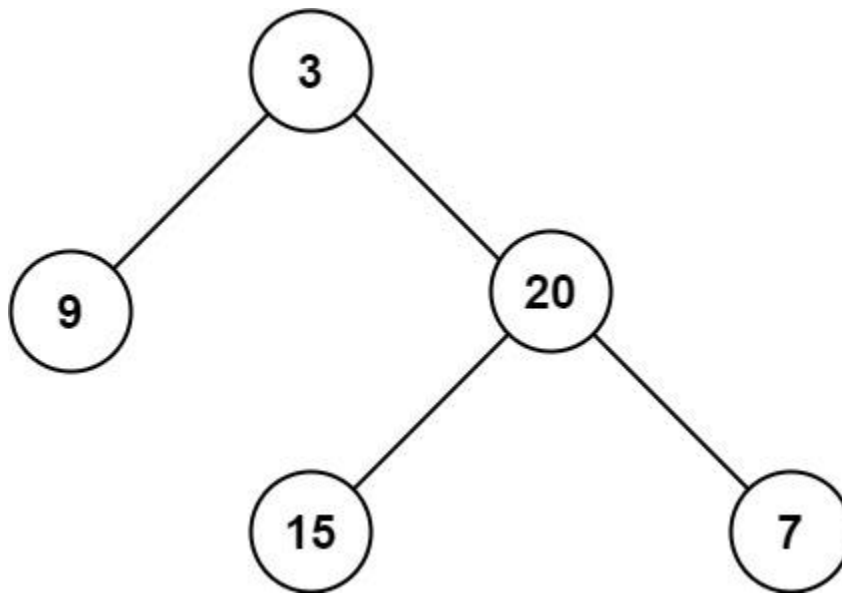
1.113 Maximum Depth Of Binary Tree

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

```c
int max(int a, int b) {
    return (a > b) ? a : b;
}
int maxDepth(struct TreeNode* root){
    if (root == NULL) {
        return 0;
    }
    return max(1 + maxDepth(root -> left), 1 + maxDepth(root -> right));

}
```

1.114 Same Tree

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

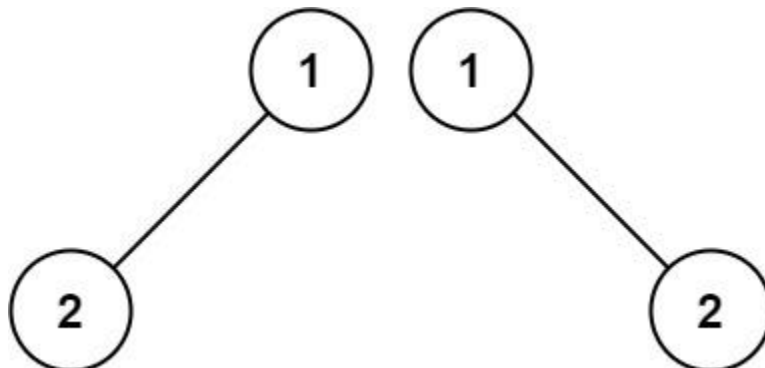Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.
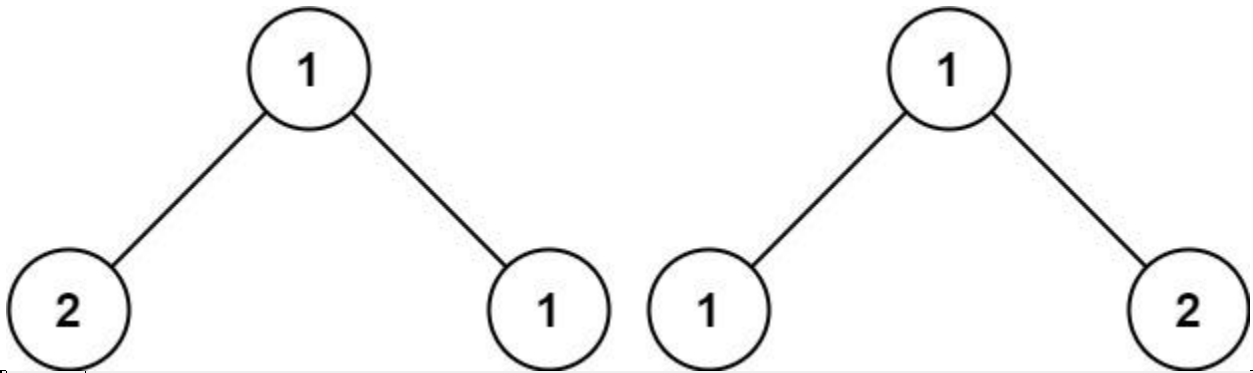
Example 1:
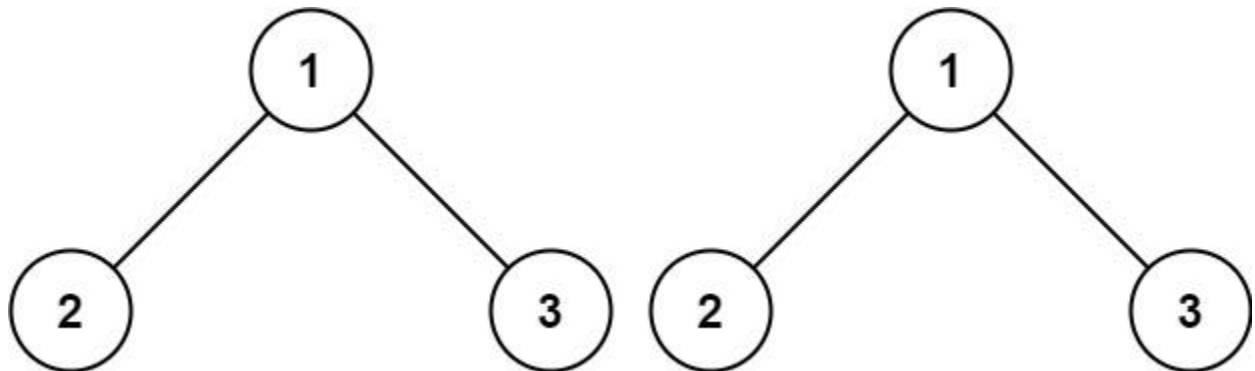
Input: p = [1,2,3], q = [1,2,3]
Output: true

Example 2:


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

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

```c
bool isSameTree(struct TreeNode* p, struct TreeNode* q) {
    if (p == NULL && q == NULL) {
        return true;
    }
    else if ((p == NULL && q != NULL) ||
            (q == NULL && p != NULL)) {
                return false;
            }
    if (p->val != q->val) {
        return false;
    }
    return isSameTree(p->left, q->left) && isSameTree(p->right, q->right);

}
```
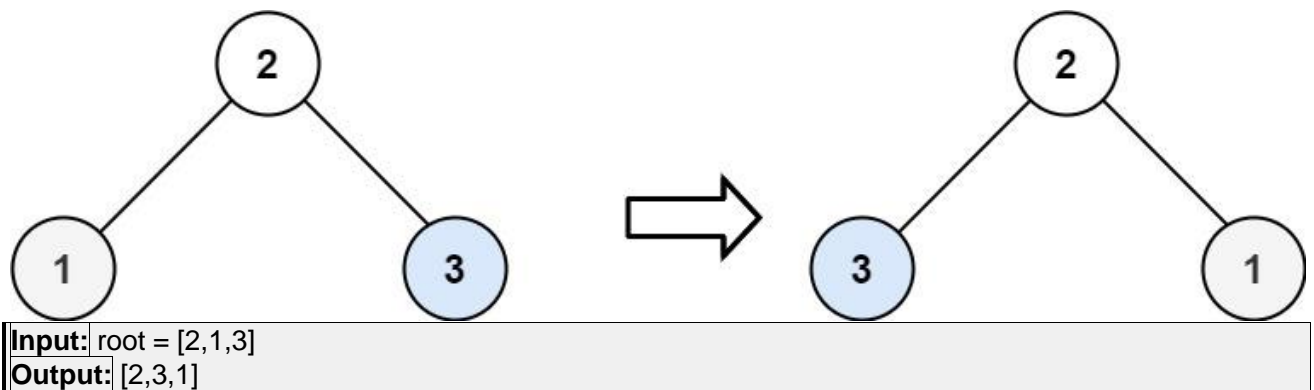
1.115 Invert Binary Tree

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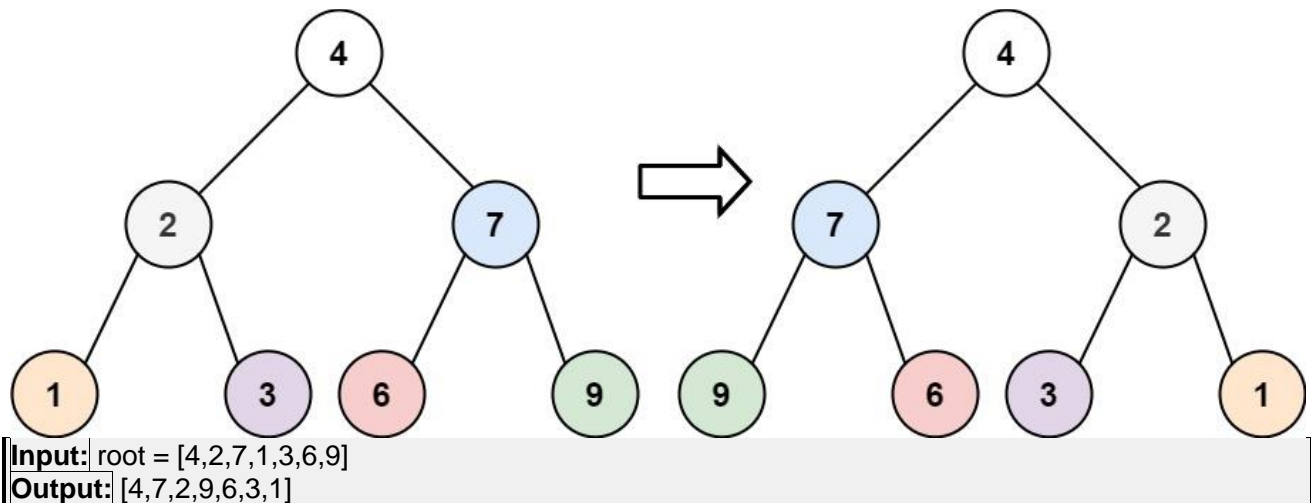
Input: root = [2,1,3]
Output: [2,3,1]

Example 3:

Input: root = []
Output: []

```c
struct TreeNode* invertTree(struct TreeNode* root) {
    if (root == NULL) {
        return NULL;
    }

    // Swap left and right subtrees
    struct TreeNode* temp = root->left;
    root->left = root->right;
    root->right = temp;

    // Invert left and right subtrees recursively
    invertTree(root->left);
    invertTree(root->right);

    return root;
}
```

1.116 Binary Tree Maximum Path Sum

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

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

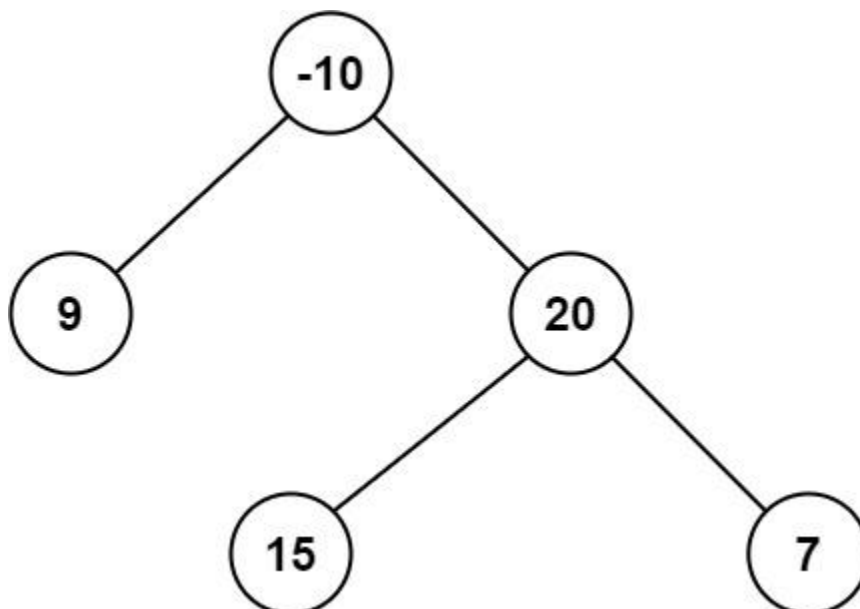Given the root of a binary tree, return *the maximum **path sum** of any **non-empty** path*.

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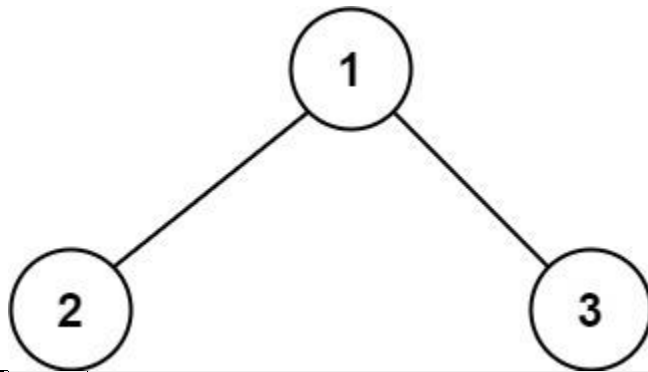
Input: root = [1,2,3]
Output: 6
Explanation: The optimal path is 2 -> 1 -> 3 with a path sum of 2 + 1 + 3 = 6.

Example 2:

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left = left;
 *         this.right = right;
 *     }
 * }
 */
class Solution {

    int max = Integer.MIN_VALUE;

    public int maxPath(TreeNode root) {

        if(root == null) return 0;

        int value = root.val;

        int left_sum = Math.max(maxPath(root.left),0);
        int right_sum = Math.max(maxPath(root.right),0);

        max = Math.max(max, left_sum + right_sum + value);

        return Math.max(left_sum, right_sum) + value;
    }

    public int maxPathSum(TreeNode root) {

        maxPath(root);
        return max;
    }
}
```
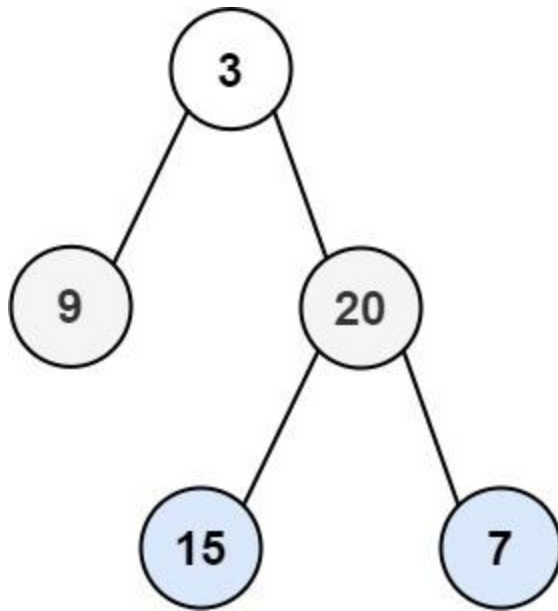
1.117 Binary Tree Level Order Traversal

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:** [] |

```c
int** levelOrder(struct TreeNode* root, int* returnSize, int** returnColumnSizes) {
    // Check if the tree is empty
    if (root == NULL) {
        *returnSize = 0;
        *returnColumnSizes = NULL;
        return NULL;
    }

    // Initialize a queue for BFS
    struct TreeNode** queue = (struct TreeNode**)malloc(sizeof(struct TreeNode*) *
10000);
    int front = 0, rear = -1;

    // Initialize 2D array to store level-order traversal
    int** result = (int**)malloc(sizeof(int*) * 10000);
    *returnColumnSizes = (int*)malloc(sizeof(int) * 10000);
    int levelSize = 0;
```

```c
    *returnSize = 0;

    // Enqueue the root node
    queue[++rear] = root;

    while (front <= rear) {
        levelSize = rear - front + 1;
        (*returnColumnSizes)[*returnSize] = levelSize;

        // Allocate memory for the current level
        result[*returnSize] = (int*)malloc(sizeof(int) * levelSize);

        // Process nodes at the current level
        for (int i = 0; i < levelSize; i++) {
            struct TreeNode* node = queue[front++];
            result[*returnSize][i] = node->val;

            // Enqueue left and right children, if present
            if (node->left) queue[++rear] = node->left;
            if (node->right) queue[++rear] = node->right;
        }

        (*returnSize)++;
    }

    // Free the queue memory
    free(queue);

    return result;
}
```
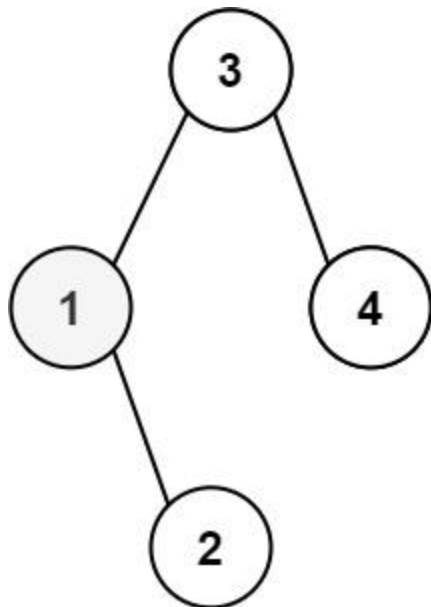
1.118 Kth Smallest Element In BST

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*.
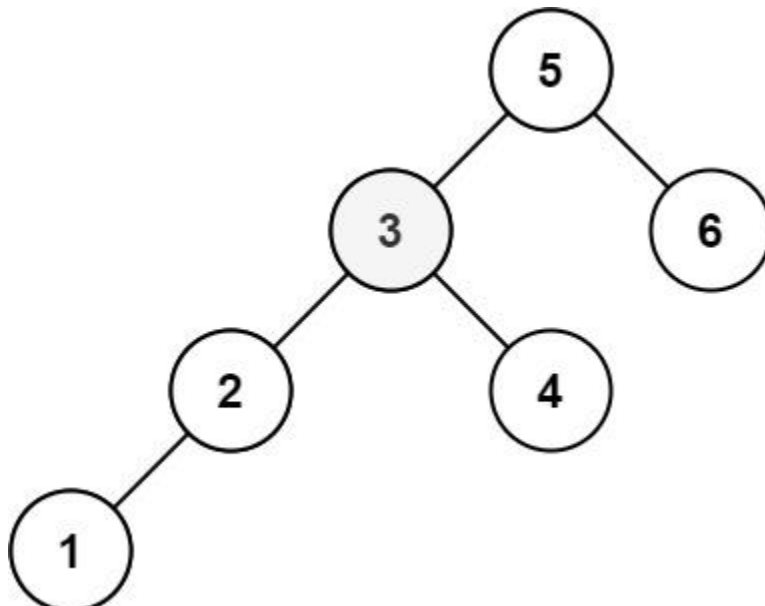
Example 1:

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

```java
class Solution {
    public ArrayList<Integer> inorder(TreeNode root, ArrayList<Integer> arr) {
        if (root == null) return arr;
        inorder(root.left, arr);
        arr.add(root.val);
        inorder(root.right, arr);
        return arr;
    }
```

```java
    public int kthSmallest(TreeNode root, int k) {
        ArrayList<Integer> nums = inorder(root, new ArrayList<Integer>());
        return nums.get(k - 1);
    }
}
```

1.119 SubTree Of Another Tree

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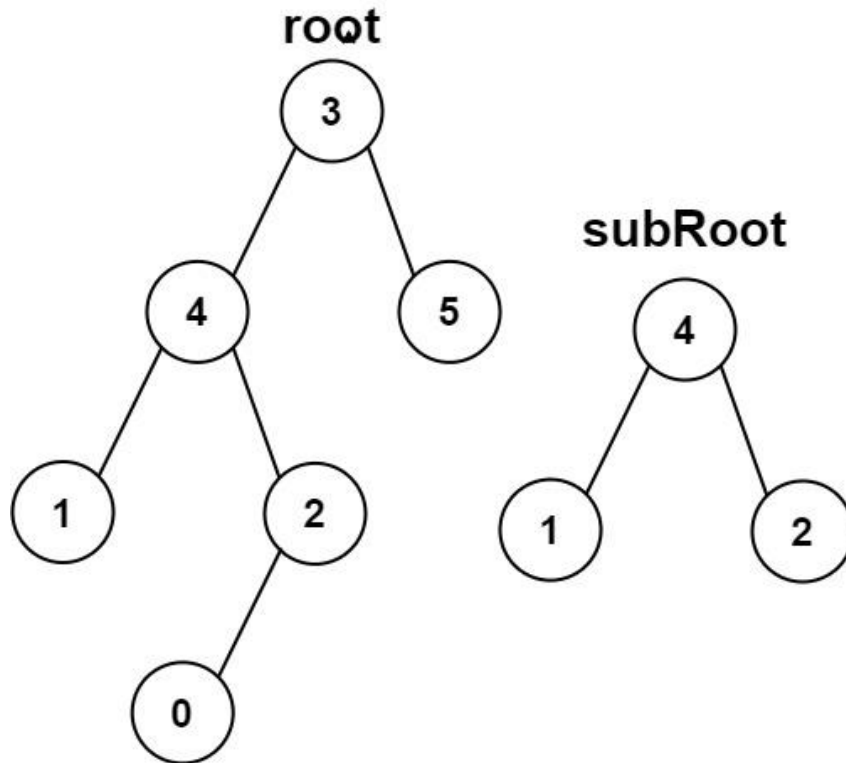
Example 1:
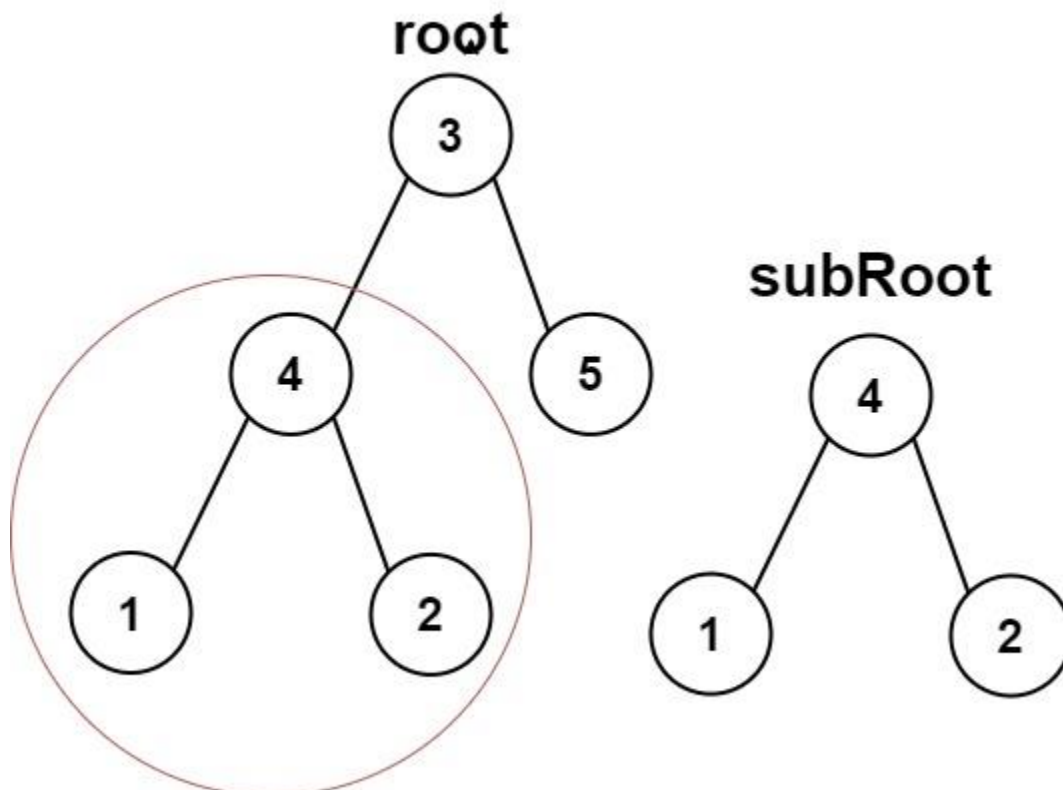
Input: root = [3,4,5,1,2], subRoot = [4,1,2]
Output: true

Example 2:

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

```java
public class Solution {
    public boolean isSubtree(TreeNode s, TreeNode t) {
        if (s == null) return false;
        if (isSame(s, t)) return true;
        return isSubtree(s.left, t) || isSubtree(s.right, t);
    }

    private boolean isSame(TreeNode s, TreeNode t) {
        if (s == null && t == null) return true;
        if (s == null || t == null) return false;

        if (s.val != t.val) return false;

        return isSame(s.left, t.left) && isSame(s.right, t.right);
    }
}
```

1.120 Word Search II

Given an m x n board of characters and a list of strings words, return *all words on the board*.

Each word must 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 in a word.

Example 1:

Input: board = [["o","a","a","n"],["e","t","a","e"],["i","h","k","r"],["i","f","l","v"]], words = ["oath","pea","eat","rain"]
Output: ["eat","oath"]

Example 2:

Input: board = [["a","b"],["c","d"]], words = ["abcb"]
Output: []

```
class TrieNode {
  HashMap<Character, TrieNode> children = new HashMap<Character, TrieNode>();
  String word = null;
  public TrieNode() {}
}

class Solution {
  char[][] _board = null;
```

```
ArrayList<String> _result = new ArrayList<String>();

public List<String> findWords(char[][] board, String[] words) {

  // Step 1). Construct the Trie
  TrieNode root = new TrieNode();
  for (String word : words) {
    TrieNode node = root;

    for (Character letter : word.toCharArray()) {
      if (node.children.containsKey(letter)) {
        node = node.children.get(letter);
      } else {
        TrieNode newNode = new TrieNode();
        node.children.put(letter, newNode);
        node = newNode;
      }
    }
    node.word = word;  // store words in Trie
  }

  this._board = board;
  // Step 2). Backtracking starting for each cell in the board
  for (int row = 0; row < board.length; ++row) {
    for (int col = 0; col < board[row].length; ++col) {
      if (root.children.containsKey(board[row][col])) {
        backtracking(row, col, root);
      }
    }
  }

  return this._result;
}

private void backtracking(int row, int col, TrieNode parent) {
  Character letter = this._board[row][col];
  TrieNode currNode = parent.children.get(letter);

  // check if there is any match
  if (currNode.word != null) {
    this._result.add(currNode.word);
    currNode.word = null;
  }

  // mark the current letter before the EXPLORATION
  this._board[row][col] = '#';
```

```
    // explore neighbor cells in around-clock directions: up, right, down, left
    int[] rowOffset = {-1, 0, 1, 0};
    int[] colOffset = {0, 1, 0, -1};
    for (int i = 0; i < 4; ++i) {
      int newRow = row + rowOffset[i];
      int newCol = col + colOffset[i];
      if (newRow < 0 || newRow >= this._board.length || newCol < 0
          || newCol >= this._board[0].length) {
        continue;
      }
      if (currNode.children.containsKey(this._board[newRow][newCol])) {
        backtracking(newRow, newCol, currNode);
      }
    }

    // End of EXPLORATION, restore the original letter in the board.
    this._board[row][col] = letter;

    // Optimization: incrementally remove the leaf nodes
    if (currNode.children.isEmpty()) {
      parent.children.remove(letter);
    }
  }
}
```

1.121 Clone Graph

Given a reference of a node in a **connected** undirected graph.

Return a **deep copy** (clone) of the graph.

Each node in the graph contains a value (int) and a list (List[Node]) of its neighbors.

```
class Node {
    public int val;
    public List<Node> neighbors;
}
```
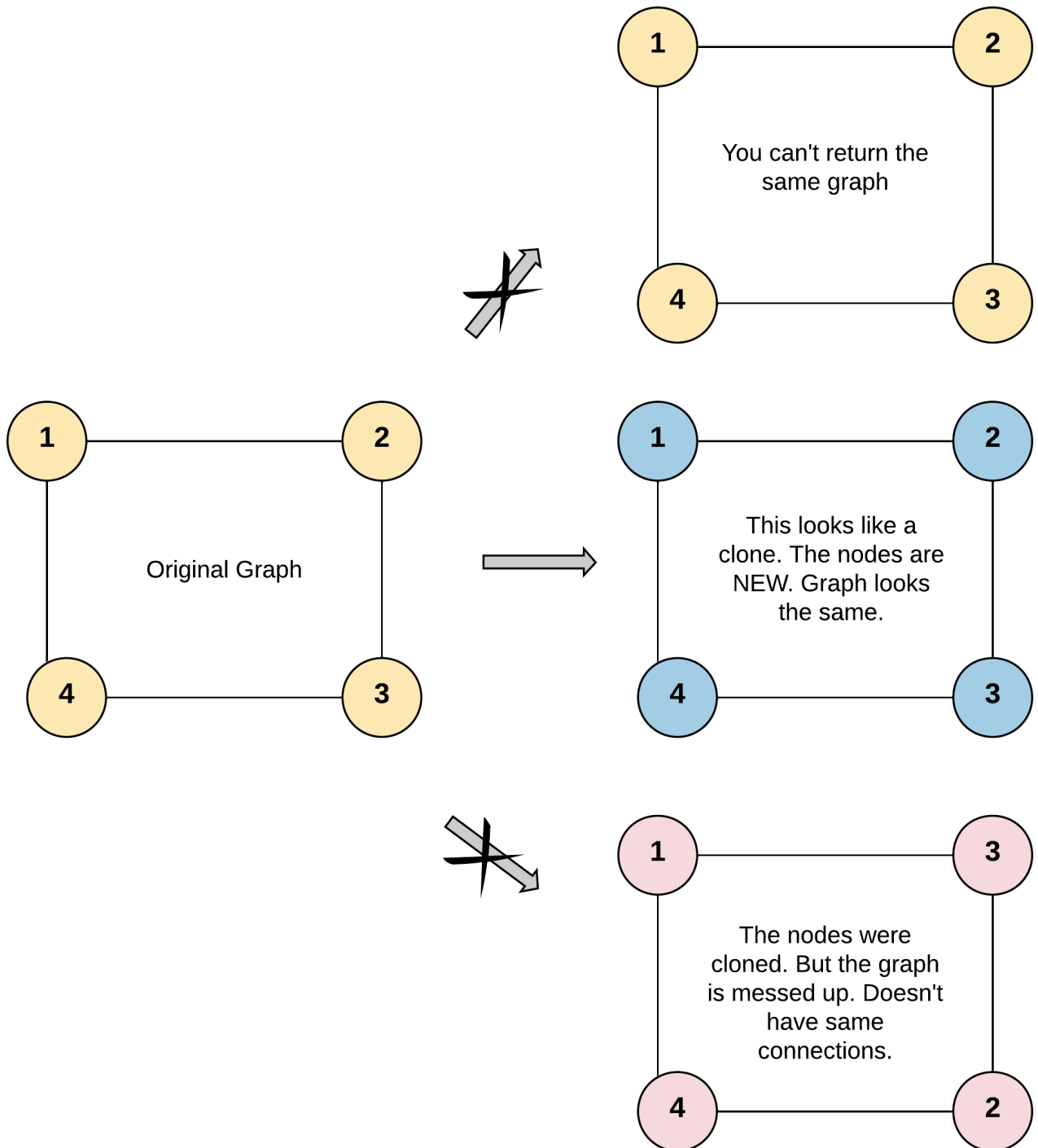
Test case format:

For simplicity, each node's value is the same as the node's index (1-indexed). For example, the first node with val == 1, the second node with val == 2, and so on. The graph is represented in the test case using an adjacency list.

An adjacency list is a collection of unordered **lists** used to represent a finite graph. Each list describes the set of neighbors of a node in the graph.

The given node will always be the first node with val = 1. You must return the **copy of the given node** as a reference to the cloned graph.

You can't return the same graph

Original Graph

This looks like a clone. The nodes are NEW. Graph looks the same.

The nodes were cloned. But the graph is messed up. Doesn't have same connections.

Input: adjList = [[2,4],[1,3],[2,4],[1,3]]
Output: [[2,4],[1,3],[2,4],[1,3]]

Explanation: There are 4 nodes in the graph.
1st node (val = 1)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).
2nd node (val = 2)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).
3rd node (val = 3)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).
4th node (val = 4)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).

Example 2:

Input: adjList = [[]]
Output: [[]]
Explanation: Note that the input contains one empty list. The graph consists of only one node with val = 1 and it does not have any neighbors.

Example 3:

Input: adjList = []
Output: []
Explanation: This an empty graph, it does not have any nodes.

```
class Solution {
    private HashMap <Node, Node> visited = new HashMap <> ();
    public Node cloneGraph(Node node) {
        if (node == null) {
            return node;
        }

        // If the node was already visited before.
        // Return the clone from the visited dictionary.
        if (visited.containsKey(node)) {
            return visited.get(node);
        }

        // Create a clone for the given node.
        // Note that we don't have cloned neighbors as of now, hence [].
        Node cloneNode = new Node(node.val, new ArrayList());
        // The key is original node and value being the clone node.
        visited.put(node, cloneNode);

        // Iterate through the neighbors to generate their clones
        // and prepare a list of cloned neighbors to be added to the cloned node.
        for (Node neighbor: node.neighbors) {
            cloneNode.neighbors.add(cloneGraph(neighbor));
        }
        return cloneNode;
    }
```

```
}
```

1.122 Course Schedule

There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [a, b] indicates that you **must** take course b first if you want to take course a.

- For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.

Return true if you can finish all courses. Otherwise, return false.

Example 1:

Input: numCourses = 2, prerequisites = [[1,0]]
Output: true
Explanation: There are a total of 2 courses to take.
To take course 1 you should have finished course 0. So it is possible.

Example 2:

Input: numCourses = 2, prerequisites = [[1,0],[0,1]]
Output: false
Explanation: There are a total of 2 courses to take.
To take course 1 you should have finished course 0, and to take course 0 you should also have finished course 1. So it is impossible.

```java
class Solution {
    public boolean canFinish(int numCourses, int[][] prerequisites) {
        int[] indegree = new int[numCourses];
        List<List<Integer>> adj = new ArrayList<>(numCourses);

        for (int i = 0; i < numCourses; i++) {
            adj.add(new ArrayList<>());
        }

        for (int[] prerequisite : prerequisites) {
            adj.get(prerequisite[1]).add(prerequisite[0]);
            indegree[prerequisite[0]]++;
        }

        Queue<Integer> queue = new LinkedList<>();
        // Push all the nodes with indegree zero in the queue.
        for (int i = 0; i < numCourses; i++) {
            if (indegree[i] == 0) {
                queue.offer(i);
            }
        }
```

```
        int nodesVisited = 0;
        while (!queue.isEmpty()) {
            int node = queue.poll();
            nodesVisited++;

            for (int neighbor : adj.get(node)) {
                // Delete the edge "node -> neighbor".
                indegree[neighbor]--;
                if (indegree[neighbor] == 0) {
                    queue.offer(neighbor);
                }
            }
        }

        return nodesVisited == numCourses;
    }
}

class Solution {
    public boolean dfs(int node, List<List<Integer>> adj, boolean[] visit, boolean[] inStack) {
        // If the node is already in the stack, we have a cycle.
        if (inStack[node]) {
            return true;
        }
        if (visit[node]) {
            return false;
        }
        // Mark the current node as visited and part of current recursion stack.
        visit[node] = true;
        inStack[node] = true;
        for (int neighbor : adj.get(node)) {
            if (dfs(neighbor, adj, visit, inStack)) {
                return true;
            }
        }
        // Remove the node from the stack.
        inStack[node] = false;
        return false;
    }

    public boolean canFinish(int numCourses, int[][] prerequisites) {
        List<List<Integer>> adj = new ArrayList<>(numCourses);
        for (int i = 0; i < numCourses; i++) {
            adj.add(new ArrayList<>());
        }

        for (int[] prerequisite : prerequisites) {
            adj.get(prerequisite[1]).add(prerequisite[0]);
        }

        boolean[] visit = new boolean[numCourses];
        boolean[] inStack = new boolean[numCourses];
        for (int i = 0; i < numCourses; i++) {
            if (dfs(i, adj, visit, inStack)) {
                return false;
            }
```

```
        }
        return true;
    }
```

1.123 Pacific Atlantic Water Flow

There is an m x n rectangular island that borders both the **Pacific Ocean** and **Atlantic Ocean**.
The **Pacific Ocean** touches the island's left and top edges, and the **Atlantic Ocean** touches the island's right and bottom edges.

The island is partitioned into a grid of square cells. You are given an m x n integer
matrix heights where heights[r][c] represents the **height above sea level** of the cell at coordinate (r, c).

The island receives a lot of rain, and the rain water can flow to neighboring cells directly north, south, east, and west if the neighboring cell's height is **less than or equal to** the current cell's height. Water can flow from any cell adjacent to an ocean into the ocean.

Return *a **2D list** of grid coordinates* result *where* result[i] = [r, c] *denotes that rain water can flow from cell* (r, c) *to **both** the Pacific and Atlantic oceans*.

Example 1:
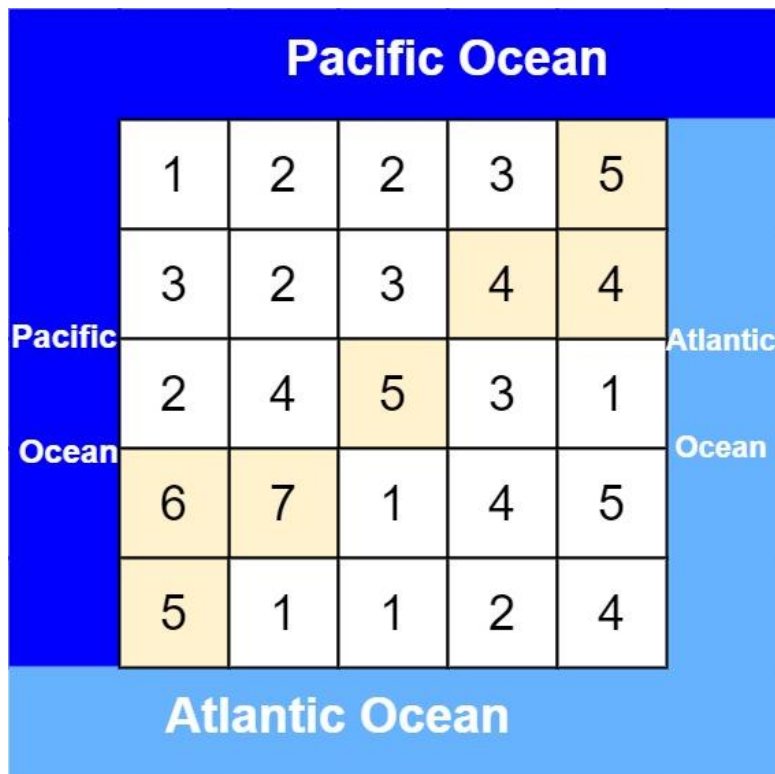
Input: heights = [[1,2,2,3,5],[3,2,3,4,4],[2,4,5,3,1],[6,7,1,4,5],[5,1,1,2,4]]
Output: [[0,4],[1,3],[1,4],[2,2],[3,0],[3,1],[4,0]]
Explanation: The following cells can flow to the Pacific and Atlantic oceans, as shown below:
[0,4]: [0,4] -> Pacific Ocean

```
        [0,4] -> Atlantic Ocean
[1,3]: [1,3] -> [0,3] -> Pacific Ocean
       [1,3] -> [1,4] -> Atlantic Ocean
[1,4]: [1,4] -> [1,3] -> [0,3] -> Pacific Ocean
       [1,4] -> Atlantic Ocean
[2,2]: [2,2] -> [1,2] -> [0,2] -> Pacific Ocean
       [2,2] -> [2,3] -> [2,4] -> Atlantic Ocean
[3,0]: [3,0] -> Pacific Ocean
       [3,0] -> [4,0] -> Atlantic Ocean
[3,1]: [3,1] -> [3,0] -> Pacific Ocean
       [3,1] -> [4,1] -> Atlantic Ocean
[4,0]: [4,0] -> Pacific Ocean
       [4,0] -> Atlantic Ocean
Note that there are other possible paths for these cells to flow to the Pacific and Atlantic oceans.
```

Example 2:

Input: heights = [[1]]
Output: [[0,0]]
Explanation: The water can flow from the only cell to the Pacific and Atlantic oceans.

```java
class Solution {
    private static final int[][] DIRECTIONS = new int[][]{{0, 1}, {1, 0}, {-1, 0}, {0, -1}};
    private int numRows;
    private int numCols;
    private int[][] landHeights;

    public List<List<Integer>> pacificAtlantic(int[][] matrix) {
        // Check if input is empty
        if (matrix.length == 0 || matrix[0].length == 0) {
            return new ArrayList<>();
        }

        // Save initial values to parameters
        numRows = matrix.length;
        numCols = matrix[0].length;
        landHeights = matrix;
        boolean[][] pacificReachable = new boolean[numRows][numCols];
        boolean[][] atlanticReachable = new boolean[numRows][numCols];

        // Loop through each cell adjacent to the oceans and start a DFS
        for (int i = 0; i < numRows; i++) {
            dfs(i, 0, pacificReachable);
            dfs(i, numCols - 1, atlanticReachable);
        }
        for (int i = 0; i < numCols; i++) {
            dfs(0, i, pacificReachable);
            dfs(numRows - 1, i, atlanticReachable);
        }

        // Find all cells that can reach both oceans
        List<List<Integer>> commonCells = new ArrayList<>();
```

```
        for (int i = 0; i < numRows; i++) {
            for (int j = 0; j < numCols; j++) {
                if (pacificReachable[i][j] && atlanticReachable[i][j]) {
                    commonCells.add(List.of(i, j));
                }
            }
        }
        return commonCells;
    }

    private void dfs(int row, int col, boolean[][] reachable) {
        // This cell is reachable, so mark it
        reachable[row][col] = true;
        for (int[] dir : DIRECTIONS) { // Check all 4 directions
            int newRow = row + dir[0];
            int newCol = col + dir[1];
            // Check if new cell is within bounds
            if (newRow < 0 || newRow >= numRows || newCol < 0 || newCol >= numCols) {
                continue;
            }
            // Check that the new cell hasn't already been visited
            if (reachable[newRow][newCol]) {
                continue;
            }
            // Check that the new cell has a higher or equal height,
            // So that water can flow from the new cell to the old cell
            if (landHeights[newRow][newCol] < landHeights[row][col]) {
                continue;
            }
            // If we've gotten this far, that means the new cell is reachable
            dfs(newRow, newCol, reachable);
        }
    }
}
```

1.124 Alien Dictionary

There is a new alien language that uses the English alphabet. However, the order of the letters is unknown to you.

You are given a list of strings words from the alien language's dictionary. Now it is claimed that the strings in words are

sorted lexicographically
 by the rules of this new language.

If this claim is incorrect, and the given arrangement of string in words cannot correspond to any order of letters, return "".

Otherwise, return *a string of the unique letters in the new alien language sorted in **lexicographically increasing order** by the new language's rules.* If there are multiple solutions, return *any of them*.

Example 1:

Input: words = ["wrt","wrf","er","ett","rftt"]
Output: "wertf"

Example 2:

Input: words = ["z","x"]
Output: "zx"

Example 3:

Input: words = ["z","x","z"]
Output: ""
Explanation: The order is invalid, so return "".

```java
class Solution {

    private Map<Character, List<Character>> reverseAdjList = new HashMap<>();
    private Map<Character, Boolean> seen = new HashMap<>();
    private StringBuilder output = new StringBuilder();

    public String alienOrder(String[] words) {

        // Step 0: Put all unique letters into reverseAdjList as keys.
        for (String word : words) {
            for (char c : word.toCharArray()) {
                reverseAdjList.putIfAbsent(c, new ArrayList<>());
            }
        }

        // Step 1: Find all edges and add reverse edges to reverseAdjList.
        for (int i = 0; i < words.length - 1; i++) {
            String word1 = words[i];
            String word2 = words[i + 1];
            // Check that word2 is not a prefix of word1.
            if (word1.length() > word2.length() && word1.startsWith(word2)) {
                return "";
            }
            // Find the first non match and insert the corresponding relation.
            for (int j = 0; j < Math.min(word1.length(), word2.length()); j++) {
                if (word1.charAt(j) != word2.charAt(j)) {
                    reverseAdjList.get(word2.charAt(j)).add(word1.charAt(j));
                    break;
                }
            }
        }

        // Step 2: DFS to build up the output list.
        for (Character c : reverseAdjList.keySet()) {
            boolean result = dfs(c);
            if (!result) return "";
```

```
        }

        return output.toString();
    }

    // Return true iff no cycles detected.
    private boolean dfs(Character c) {
        if (seen.containsKey(c)) {
            return seen.get(c); // If this node was grey (false), a cycle was detected.
        }
        seen.put(c, false);
        for (Character next : reverseAdjList.get(c)) {
            boolean result = dfs(next);
            if (!result) return false;
        }
        seen.put(c, true);
        output.append(c);
        return true;
    }
}
```
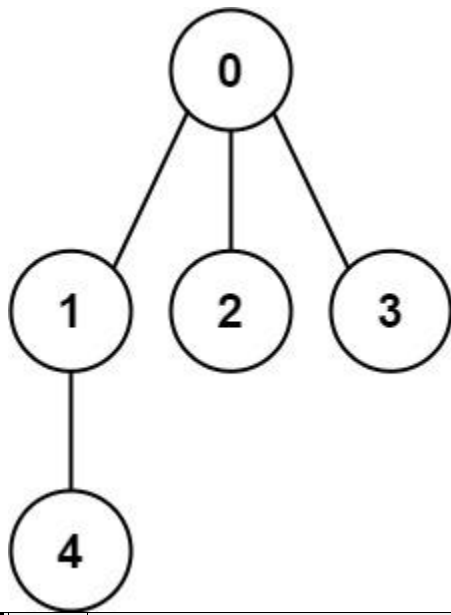
1.125 Graph Valid Tree

You have a graph of n nodes labeled from 0 to n - 1. You are given an integer n and a list
of edges where edges[i] = [a, b] indicates that there is an undirected edge between nodes a and b in the
graph.

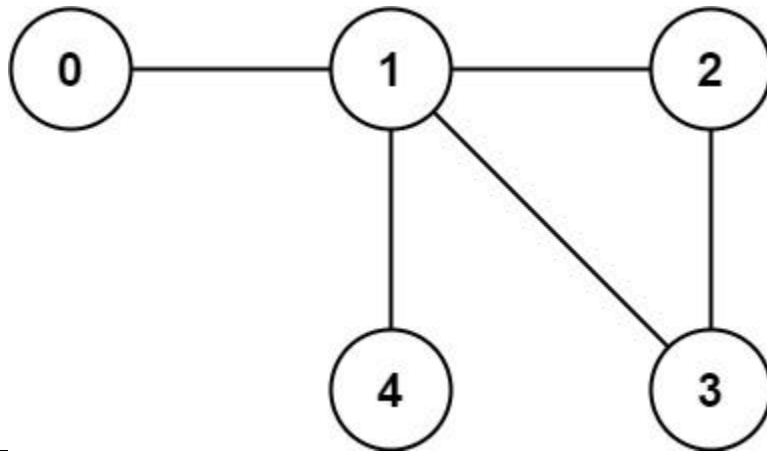Return true *if the edges of the given graph make up a valid tree, and* false *otherwise*.

Example 1:

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

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

```
class Solution {
    // A graph is considered a tree if it is connected and the number of edges = n-1 nodes
    public boolean validTree(int n, int[][] edges) {
        List<List<Integer>> graph = new ArrayList <>();

            //create graph
        for (int i = 0; i < n; i++) {
            graph.add(new ArrayList<>());
        }

            //since this is an undirected graph, make each edge go both ways
        for (int[] edge : edges) {
            graph.get(edge[1]).add(edge[0]);
            graph.get(edge[0]).add(edge[1]);
        }

        HashSet<Integer> visited = new HashSet<>();

            //if graph is connected then we would be able to start from any node and reach all the
other nodes using dfs
        //we can dfs starting from first node since this is guaranteed to exist according to constraints
        dfs(visited, 0, graph);

        //if we see an unvisited node,
        //the graph is not connected and therefore cannot be a tree
        for (int i = 0; i < n; i++) {
            if(!visited.contains(i)) return false;
        }
```

```
      //also check number of edges = n-1 nodes
      return (edges.length == n-1);
   }

   public void dfs(HashSet<Integer> visited, int currNode, List<List<Integer>> graph) {
      visited.add(currNode);
      for (Integer node : graph.get(currNode)) {
         if(!visited.contains(node)) {
            dfs(visited, node, graph);
         }
      }
   }
}
```
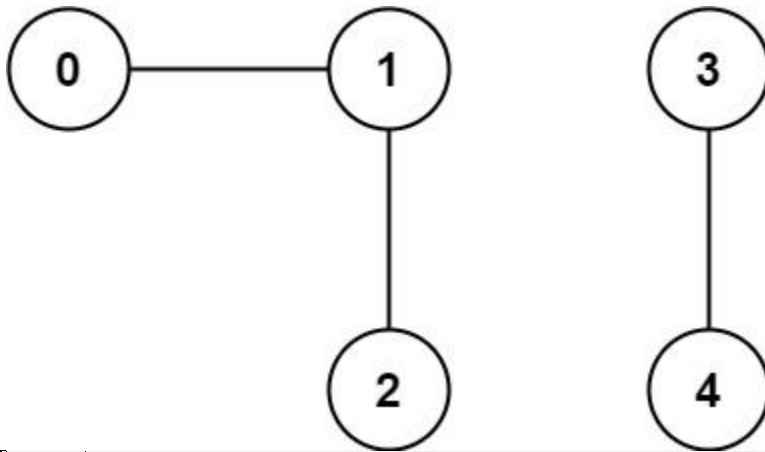
1.126 Number Of Connected Components In an Undirected Graph

You have a graph of n nodes. You are given an integer n and an array edges where edges[i] = [a_i, b_i] indicates that there is an edge between a_i and b_i in the graph.

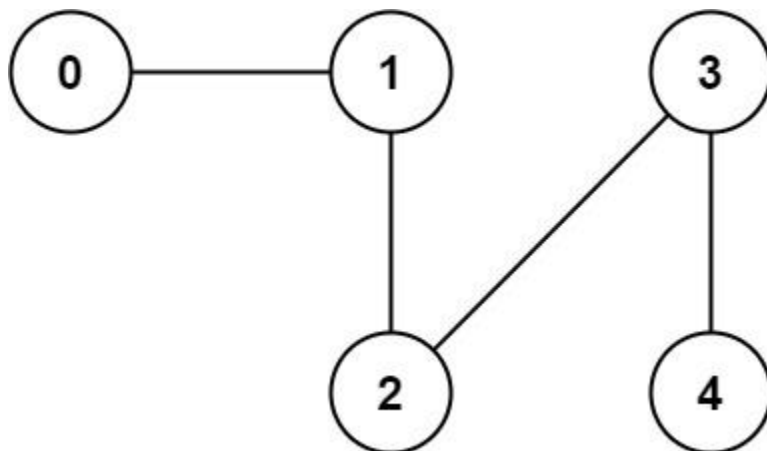Return *the number of connected components in the graph*.

Example 1:

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

Example 2:

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

```java
class Solution {
    public int countComponents(int n, int[][] edges) {
      int[] roots = new int[n];
      for(int i = 0; i < n; i++) roots[i] = i;

      for(int[] e : edges) {
          int root1 = find(roots, e[0]);
          int root2 = find(roots, e[1]);
          if(root1 != root2) {
              roots[root1] = root2;   // union
              n--;
          }
      }
      return n;
}

public int find(int[] roots, int id) {
    while(roots[id] != id) {
        roots[id] = roots[roots[id]];   // optional: path compression
        id = roots[id];
    }
    return id;
}
}
```