BASIC CODING PREPARATION

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) {
```

```c
    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:

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

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

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) {
```

```c
            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 Time Needed To Inform All Employees

```c
typedef struct Node {
    int val;
    struct Node* next;
} Node;

typedef struct {
    Node* head;
} List;

List* createList() {
    List* list = (List*)malloc(sizeof(List));
    list->head = NULL;
    return list;
}
void printList(List* list) {
    Node* current = list->head;

    while (current != NULL) {
        printf("%d ", current->val);
        current = current->next;
    }

    printf("\n");
}
void addToList(List* list, int val) {
```

```c
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->val = val;
    newNode->next = list->head;
    list->head = newNode;
}

int helper(int* informTime, List** list, int curr) {
    printf("\ncurr : %d",curr);
    if (list[curr] == NULL) return 0;

    int min = 0;
    Node* current = list[curr]->head;

    while (current != NULL) {
        min = fmax(helper(informTime, list, current->val), min);
        current = current->next;
    }

    return min + informTime[curr];
}

int numOfMinutes(int n, int headID, int* manager, int managerSize, int* informTime, int
informTimeSize) {
    List** list = (List**)malloc(n * sizeof(List*));

    for (int i = 0; i < n; i++) {
        list[i] = createList();
    }

    for (int i = 0; i < n; i++) {
        if (manager[i] != -1) {
            addToList(list[manager[i]], i);
        }
    }
    for (int i = 0; i < n; i++) {
        printf("\ni = %d",i);
        printList(list[i]);
    }
    int result = helper(informTime, list, headID);

    for (int i = 0; i < n; i++) {
        free(list[i]);
    }

    free(list);

    return result;
}
```

1.4 Distribute Coins In Binary Tree

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

int helper(struct TreeNode* root, int* moves) {
    if (root == NULL)
        return 0;

    int coinsLeft = helper(root->left, moves);
    int coinsRight = helper(root->right, moves);

    int coins = coinsLeft + coinsRight;

    if (root->val == 0)
        coins -= 1;
    else if (root->val == 1)
        coins += 0;
    else
        coins += root->val - 1;

    *moves += abs(coins);

    return coins;
}

int distributeCoins(struct TreeNode* root) {
    int moves = 0;
    helper(root, &moves);
    return moves;
}
```

1.5 Find Original Array From Doubled Array

```
int* findOriginalArray(int* changed, int changedSize, int* returnSize) {
    *returnSize = 0;

    int n = changedSize;
    if (n % 2 == 1) {
        return NULL;
    }

    int* ans = (int*)malloc((n / 2) * sizeof(int));
    int idx = 0;
    int* freq = (int*)calloc(100005, sizeof(int));
```

```c
    for (int i = 0; i < n; i++) {
        freq[changed[i]]++;
    }

    for (int num = 100000; num >= 0; num--) {
        while (freq[num]-- > 0) {
            if ((num & 1) == 1) {
                free(ans);
                free(freq);
                return NULL;
            }

            int div = num / 2;

            if (freq[div] == 0) {
                free(ans);
                free(freq);
                return NULL;
            }

            ans[idx++] = div;
            freq[div]--;
        }
    }

    *returnSize = idx;
    return ans;
}
```

1.6 Nth Digit

```c
int findNthDigit(int n) {
    long base = 9;
    int digits = 1;

    // Step 1: Determine the length of the numbers at each level
    while (n > base * digits) {
        n -= base * digits;
        base *= 10;
        digits++;
    }
    printf("\ndigits = %d n = %d",digits,n);
    // Step 2: Calculate the actual number where the nth digit belongs
    int number = 1;
    for (int i = 1; i < digits; i++) {
        number *= 10;
    }
    printf("\nnumber = %d",number);
```

```
    number += (n - 1) / digits;
    printf("\nnew number = %d",number);

    // Step 3: Find the position of the digit within that number
    int position = (n - 1) % digits;
    printf("\nposition = %d", position);
    // Step 4: Return the digit at the specified position
    for (int i = digits - 1; i > position; i--) {
        number /= 10;
    }
    printf("\nlast number = %d",number);
    return number % 10;
}
```

1.7 Minimize The Maximum Difference Of Pairs

```
// Function to compare integers for qsort
int compare(const void* a, const void* b) {
    return (*(int*)a - *(int*)b);
}

// Function to find the number of valid pairs by greedy approach
int countValidPairs(int* nums, int size, int threshold) {
    int index = 0, count = 0;
    while (index < size - 1) {
        // If a valid pair is found, skip both numbers.
        if (nums[index + 1] - nums[index] <= threshold) {
            count++;
            index++;
        }
        index++;
    }
    return count;
}

// Function to minimize the maximum difference between pairs
int minimizeMax(int* nums, int size, int p) {
    // Sort the array
    qsort(nums, size, sizeof(int), compare);

    int left = 0, right = nums[size - 1] - nums[0];

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

        // If there are enough pairs, look for a smaller threshold.
        // Otherwise, look for a larger threshold.
        if (countValidPairs(nums, size, mid) >= p) {
            right = mid;
```

```
        } else {
            left = mid + 1;
        }
    }
    return left;
}
```

1.8 Reverse Words In String II

```
// Function to reverse a portion of the array
void reverse(char* s, int start, int end) {
    while (start < end) {
        char temp = s[start];
        s[start] = s[end];
        s[end] = temp;
        start++;
        end--;
    }
}

// Function to reverse the order of words in-place
void reverseWords(char* s, int sSize) {
    // Step 1: Reverse the entire array
    reverse(s, 0, sSize - 1);
    //printf("\n s is : %s",s);
    // Step 2: Reverse each individual word
    int start = 0;
    for (int i = 0; i < sSize; i++) {
        if (s[i] == ' ' || i == sSize - 1) {
            // If space is encountered or at the end, reverse the word
            //printf("\nreversing now from start:%d to i:%d",start,i);
            (i == sSize-1) ? reverse(s, start, i):reverse(s, start, i - 1);
            //printf("\nNow s is : %s",s);
            start = i + 1;
            //printf("\n strt is : %d i = %d",start,i);
        } else {
            //printf("\nELSE Now s is : %s",s);
            //printf("\n here strt is : %d i = %d",start,i);
        }
    }
}
```

1.9 Number Of Flowers In Full Bloom

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

```c
// Utility function to perform binary search
int compareInt(const void* a, const void* b) {
    return (*(int*)a - *(int*)b);
}
int binarySearch(int* arr, int size, int target) {
    int left = 0, right = size - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] <= target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return left;
}

int* fullBloomFlowers(int** flowers, int flowersSize, int* flowersColSize, int* people,
int peopleSize, int* returnSize) {
    // Initialize arrays for start and end times
    int* starts = (int*)malloc(flowersSize * sizeof(int));
    int* ends = (int*)malloc(flowersSize * sizeof(int));

    // Populate start and end arrays
    for (int i = 0; i < flowersSize; i++) {
        starts[i] = flowers[i][0];
        ends[i] = flowers[i][1] + 1;
    }

    // Sort the start and end arrays
    qsort(starts, flowersSize, sizeof(int), compareInt);
    qsort(ends, flowersSize, sizeof(int), compareInt);

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

    // Process each person's arrival time
    for (int i = 0; i < peopleSize; i++) {
        int startCount = binarySearch(starts, flowersSize, people[i]);
        int endCount = binarySearch(ends, flowersSize, people[i]);
        result[i] = startCount - endCount;
        printf("\n%d:%d:%d",startCount,endCount,result[i]);
    }

    // Set the return size
    *returnSize = peopleSize;

    // Free allocated memory
    free(starts);
```

```c
    free(ends);

    return result;
}
```

1.10 Restore IP addresses

```c
typedef struct {
    char* data;
    int length;
} String;

void rec(const char* s, int i, char** address, int addressSize, char*** addresses, int*
returnSize);
int isValid(const char* s, int start, int length);

void rec(const char* s, int i, char** address, int addressSize, char*** addresses, int*
returnSize) {
    if (addressSize == 4) {
        if (s[i] == '\0') {
            // If we have reached the end of the input string, add the current address
to the list of valid addresses
            char* current = (char*)malloc(16 * sizeof(char));  // Maximum IP address
length is 15
            snprintf(current, 16, "%s.%s.%s.%s", address[0], address[1], address[2],
address[3]);

            (*returnSize)++;
            *addresses = (char**)realloc(*addresses, (*returnSize) * sizeof(char*));
            (*addresses)[*returnSize - 1] = current;
        }
    } else {
        // Try all possible next segments for the current address
        for (int j = i + 1; j <= i + 3 && s[j - 1] != '\0'; j++) {
            char nextInt[4] = {0};
            strncpy(nextInt, s + i, j - i);
            // Check if the next segment is valid (between 0 and 255, and not starting
with 0 unless it is 0)
            if (isValid(nextInt, 0, j - i)) {
                // Add the next segment to the current address
                address[addressSize] = nextInt;
                // Recursively call the function to generate the next segment
                rec(s, j, address, addressSize + 1, addresses, returnSize);
            }
        }
    }
}

int isValid(const char* s, int start, int length) {
    int num = 0;
```

```c
    for (int i = 0; i < length; i++) {
        num = num * 10 + (s[start + i] - '0');
    }
    return (num <= 255) && (length == 1 || (s[start] != '0'));
}
char** restoreIpAddresses(char* s, int* returnSize) {
    *returnSize = 0;
    char** addresses = NULL;

    char* address[4] = {"", "", "", ""};
    rec(s, 0, address, 0, &addresses, returnSize);

    return addresses;
}
```

1.11 SubArray Sum Divisible By K

```c
int subarraysDivByK(int* nums, int numsSize, int k) {
    int n = numsSize;
    int prefixMod = 0, result = 0;

    // There are k mod groups 0...k-1.
    int* modGroups = (int*)calloc(k, sizeof(int));
    modGroups[0] = 1;

    for (int i = 0; i < n; ++i) {
        // Take modulo twice to avoid negative remainders.
        prefixMod = (prefixMod + nums[i] % k + k) % k;
        // Add the count of subarrays that have the same remainder as the current
        printf("\nprefixMod = %d modgrp:%d",prefixMod, modGroups[prefixMod]);
        // one to cancel out the remainders.
        result += modGroups[prefixMod]++;
        printf("\nresult is: %d modgrp: %d",result,modGroups[prefixMod]);
    }

    free(modGroups);

    return result;
}
```

1.12 Find Missing Observations

```c
/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
int* missingRolls(int* rolls, int rollsSize, int mean, int n, int* returnSize) {
    int orig_sum = 0;

    *returnSize = 0;
    int *result = (int *)malloc(sizeof(int)*n);
```

```c
    for (int i = 0; i < rollsSize; i++) {
        orig_sum += rolls[i];
    }
    printf("\nOrig_sum = %d",orig_sum);
    int expected_sum = (mean*(n+rollsSize));
    printf("\nExpected_sum = %d",expected_sum);
    int missing_sum = expected_sum - orig_sum;
    printf("\nMissing_sum = %d", missing_sum);
    int missing_value = missing_sum/n;
    int remainder = missing_sum % n;
    printf("\nMissing Value = %d",missing_value);
    if ((missing_value > 6) || (missing_sum != 0 && missing_value == 0)|| (missing_sum <
0) || ((missing_value == 6) && (remainder != 0))) return result;
    for (int j = 0; j < n; j++) {
        result[j] = missing_value;
        (*returnSize)++;
    }
    for (int j = 0; j < n; j++) {
        while (remainder != 0) {
            if (missing_value < 6) {
                if (result[j] < 6) {
                    result[j]++;
                    remainder-=1;
                } else {
                    break;
                }
            } else {
                continue;
            }

        }
        //printf("\nResult : %d",result[j]);

    }
    //printf("%d",*returnSize);
    return result;
 }
```

1.13 Largest Palindromic Number

```c
// Function to find the largest palindromic integer
char* largestPalindromic(char* num) {
    int cnt[10] = {0};
    int n = strlen(num);

    // Count occurrences of each digit
    for (int i = 0; i < n; i++) {
        cnt[num[i] - '0']++;
    }
```

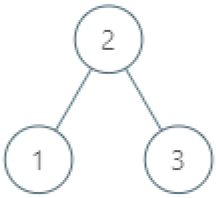
```c
    char* lp = (char*)malloc((n + 1) * sizeof(char));
    char* rp = (char*)malloc((n + 1) * sizeof(char));
    int lpIndex = 0;
    int rpIndex = 0;

    // Build left and right partitions
    for (int i = 0; i < n; i++) {
        for (int j = 9; j >= 0; j--) {
            if (cnt[j] > 1 && (j > 0 || lpIndex > 0)) {
                lp[lpIndex++] = '0' + j;
                rp[rpIndex++] = '0' + j;
                cnt[j] -= 2;
                break;
            }
        }
    }

    // Find the middle digit (if any)
    for (int i = 9; i >= 0; i--) {
        if (cnt[i]) {
            lp[lpIndex++] = '0' + i;
            break;
        }
    }

    // Reverse the right partition
    for (int i = rpIndex - 1; i >= 0; i--) {
        lp[lpIndex++] = rp[i];
    }

    // Null-terminate the result
    lp[lpIndex] = '\0';

    free(rp);

    return lp;
}
```

1.14 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.

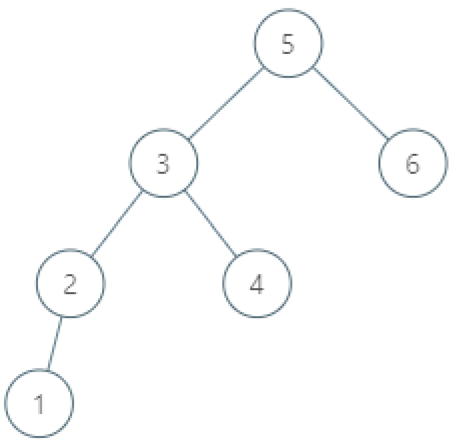
The successor of a node p is the node with the smallest key greater than p.val.

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 :

```
/**
 * 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;
}
```

1.15 Regular Expression Matching

```
bool isMatch(char* s, char* p) {
    int m = strlen(s);
    int n = strlen(p);

    // Create a 2D array to store intermediate results
    bool dp[m + 1][n + 1];
    memset(dp, false, sizeof(dp));

    // Empty pattern matches empty string
    dp[0][0] = true;

    // Handle patterns with '*'
    for (int j = 1; j <= n; j++) {
        if (p[j - 1] == '*') {
            dp[0][j] = dp[0][j - 2];
        }
    }

    // Dynamic programming to fill the rest of the array
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (p[j - 1] == s[i - 1] || p[j - 1] == '.') {
                dp[i][j] = dp[i - 1][j - 1];
            } else if (p[j - 1] == '*') {
                dp[i][j] = dp[i][j - 2] || (dp[i - 1][j] && (s[i - 1] == p[j - 2] || p[j
- 2] == '.'));
            }
        }
    }

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

1.16 Robot Room Cleaner

```
/**
 * // This is the robot's control interface.
 * // You should not implement it, or speculate about its implementation
 * interface Robot {
 *     // Returns true if the cell in front is open and robot moves into the cell.
```

```java
 *     // Returns false if the cell in front is blocked and robot stays in the current
cell.
 *     public boolean move();
 *
 *     // Robot will stay in the same cell after calling turnLeft/turnRight.
 *     // Each turn will be 90 degrees.
 *     public void turnLeft();
 *     public void turnRight();
 *
 *     // Clean the current cell.
 *     public void clean();
 * }
 */


class Solution {
    private final int[][] directions = {{-1, 0}, {0, 1}, {1, 0}, {0, -1}};
    private Set<String> visited = new HashSet<>();

    public void cleanRoom(Robot robot) {
        dfs(robot, 0, 0, 0);
    }

    private void dfs(Robot robot, int x, int y, int dir) {
        visited.add(x + "-" + y);
        robot.clean();

        for (int i = 0; i < 4; i++) {
            int newDir = (dir + i) % 4;
            int newX = x + directions[newDir][0];
            int newY = y + directions[newDir][1];

            if (!visited.contains(newX + "-" + newY) && robot.move()) {
                dfs(robot, newX, newY, newDir);
                goBack(robot);
            }

            robot.turnRight();
        }
    }

    private void goBack(Robot robot) {
        robot.turnRight();
        robot.turnRight();
        robot.move();
        robot.turnRight();
        robot.turnRight();
    }
}
```

1.17 Remove Duplicate Letters

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

    // Array to store the last occurrence index of each character
    int lastIndex[26] = {0};
    for (int i = 0; i < n; i++) {
        lastIndex[s[i] - 'a'] = i;
    }

    // Array to keep track of seen characters
    int seen[26] = {0};

    // Stack to maintain selected characters
    int stack[26];
    int top = -1;

    // Iterate through the string
    for (int i = 0; i < n; i++) {
        int curr = s[i] - 'a';

        // If the character is already seen, continue to the next iteration
        if (seen[curr]) {
            continue;
        }

        // Pop characters from the stack if the current character is smaller,
        // there are remaining occurrences of the characters in the stack,
        // and the character at the top of the stack is greater than the current
character
        while (top >= 0 && curr < stack[top] && i < lastIndex[stack[top]]) {
            seen[stack[top--]] = 0; // Pop out and mark unseen
        }

        // Push the current character onto the stack and mark it as seen
        stack[++top] = curr;
        seen[curr] = 1; // Mark seen
    }

    // Build the result string from the characters in the stack
    char* result = (char*)malloc((top + 2) * sizeof(char));
    for (int i = 0; i <= top; i++) {
        result[i] = stack[i] + 'a';
    }
    result[top + 1] = '\0';

    return result;
}
```

1.18 Score of Parantheses

```
int scoreOfParentheses(char* S) {
    int ans = 0, bal = 0;
    for (int i = 0; S[i] != '\0'; ++i) {
        if (S[i] == '(') {
            bal++;
        } else {
            bal--;
            if (S[i - 1] == '(') {
                ans += 1 << bal;
            }
        }
    }

    return ans;
}
```

1.19 Maximum Number Of Balloons

```
#define min(a,b) (a < b) ? a:b
int maxNumberOfBalloons(char* text) {
    int bCount = 0, aCount = 0, lCount = 0, oCount = 0, nCount = 0;

    // Count the frequency of all the five characters
    for (int i = 0; i < strlen(text); i++) {
        if (text[i] == 'b') {
            bCount++;
        } else if (text[i] == 'a') {
            aCount++;
        } else if (text[i] == 'l') {
            lCount++;
        } else if (text[i] == 'o') {
            oCount++;
        } else if (text[i] == 'n') {
            nCount++;
        }
    }

    // Find the potential of each character.
    // Except for 'l' and 'o', the potential is equal to the frequency.
    lCount = lCount / 2;
    oCount = oCount / 2;
    printf("%d:%d:%d:%d:%d",bCount,aCount,lCount,oCount,nCount);

    // Find the bottleneck.
    int ret = min(oCount,nCount);
    ret = min(ret,lCount);
    ret = min(ret,aCount);
    ret = min(ret,bCount);
    return ret;
```

```
}
```

1.20 Buy Two Chocolates

```
int buyChoco(int* prices, int pricesSize, int money) {
    // Assume minimum and second minimum
    int minimum = prices[0] < prices[1] ? prices[0] : prices[1];
    int secondMinimum = prices[0] > prices[1] ? prices[0] : prices[1];

    // Iterate over the remaining elements
    for (int i = 2; i < pricesSize; i++) {
        if (prices[i] < minimum) {
            secondMinimum = minimum;
            minimum = prices[i];
        } else if (prices[i] < secondMinimum) {
            secondMinimum = prices[i];
        }
    }

    // Minimum Cost
    int minCost = minimum + secondMinimum;

    // We can buy chocolates only if we have enough money
    if (minCost <= money) {
        // Return the Amount of Money Left
        return money - minCost;
    }

    // We cannot buy chocolates. Return the initial amount of money
    return money;
}
```

1.21 Permutations

```
#include <stdio.h>
#include <stdlib.h>

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void permute(int* nums, int start, int end) {
    if (start == end) {
        // Print the current permutation
        for (int i = 0; i <= end; i++) {
            printf("%d ", nums[i]);
        }
        printf("\n");
```

```
        return;
    }

    for (int i = start; i <= end; i++) {
        // Swap elements to create different permutations
        swap(&nums[start], &nums[i]);

        // Recursively generate permutations for the remaining elements
        permute(nums, start + 1, end);

        // Backtrack (undo the swap) to explore other possibilities
        swap(&nums[start], &nums[i]);
    }
}

int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int* nums = (int*)malloc(n * sizeof(int));

    printf("Enter the elements: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &nums[i]);
    }

    printf("Permutations:\n");
    permute(nums, 0, n - 1);

    free(nums);

    return 0;
}
```

1.22 <u>Combinations</u>

```
#include <stdio.h>
#include <stdlib.h>

void combine(int* nums, int n, int k, int start, int* current, int currentSize) {
    if (currentSize == k) {
        // Print the current combination
        for (int i = 0; i < k; i++) {
            printf("%d ", current[i]);
        }
        printf("\n");
        return;
    }
```

```c
    for (int i = start; i < n; i++) {
        current[currentSize] = nums[i];

        // Recursively generate combinations for the remaining elements
        combine(nums, n, k, i + 1, current, currentSize + 1);
    }
}

int main() {
    int n, k;
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int* nums = (int*)malloc(n * sizeof(int));

    printf("Enter the elements: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &nums[i]);
    }

    printf("Enter the size of combinations (k): ");
    scanf("%d", &k);

    if (k > n) {
        printf("Invalid input! Size of combinations (k) cannot be greater than the
number of elements.\n");
        return 1;
    }

    int* current = (int*)malloc(k * sizeof(int));

    printf("Combinations:\n");
    combine(nums, n, k, 0, current, 0);

    free(nums);
    free(current);

    return 0;
}

/**
 * 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 backtrack(int n, int k, int start, int* current, int** result, int* returnSize,
int** returnColumnSizes, int currentSize) {
```

```
    if (currentSize == k) {
        // Add the current combination to the result
        result[*returnSize] = (int*)malloc(k * sizeof(int));
        for (int i = 0; i < k; i++) {
            result[*returnSize][i] = current[i];
        }
        (*returnColumnSizes)[*returnSize] = k;
        (*returnSize)++;
        return;
    }

    for (int i = start; i <= n; i++) {
        current[currentSize] = i;
        // Recursively generate combinations
        backtrack(n, k, i + 1, current, result, returnSize, returnColumnSizes,
currentSize + 1);
    }
}

int** combine(int n, int k, int* returnSize, int** returnColumnSizes) {
    *returnSize = 0;

    // Allocate memory for the result
    int** result = (int**)malloc(10000 * sizeof(int*)); // Assuming a maximum limit for
the returnSize
    *returnColumnSizes = (int*)malloc(10000 * sizeof(int));

    int* current = (int*)malloc(k * sizeof(int));

    // Start backtracking from the first element
    backtrack(n, k, 1, current, result, returnSize, returnColumnSizes, 0);

    free(current);

    return result;
}
```

1.23 Remove K Digits

```
char* removeKdigits(char* num, int k) {
    int n = strlen(num);

    // Check if k is equal to the length of the number
    if (k == n) {
        char* result = (char*)malloc(2 * sizeof(char));
        result[0] = '0';
        result[1] = '\0';
        return result;
    }
```

```c
    // Initialize a stack to keep track of the digits
    char* stack = (char*)malloc((n + 1) * sizeof(char));
    int top = -1;

    // Iterate through the digits of the number
    for (int i = 0; i < n; i++) {
        // Pop elements from the stack if the current digit is smaller than the top of
the stack
        while (k > 0 && top >= 0 && num[i] < stack[top]) {
            top--;
            k--;
        }

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

    // Pop remaining elements from the stack to remove excess digits
    while (k > 0 && top >= 0) {
        top--;
        k--;
    }

    // Find the first non-zero digit in the stack
    int nonZeroIndex = 0;
    while (nonZeroIndex <= top && stack[nonZeroIndex] == '0') {
        nonZeroIndex++;
    }

    // If the resulting number is empty, return "0"
    if (nonZeroIndex > top) {
        char* result = (char*)malloc(2 * sizeof(char));
        result[0] = '0';
        result[1] = '\0';
        free(stack);
        return result;
    }

    // Construct the resulting number
    char* result = (char*)malloc((top - nonZeroIndex + 2) * sizeof(char));
    int resultIndex = 0;
    for (int i = nonZeroIndex; i <= top; i++) {
        result[resultIndex++] = stack[i];
    }
    result[resultIndex] = '\0';

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

1.24 Car Fleet

```
int carFleet(int target, int* position, int positionSize, int* speed, int speedSize) {
      int res = 0;
    double* timeArr = (double*)malloc(target * sizeof(double));

    // Calculate time to destination for each car
    for (int i = 0; i < positionSize; i++) {
        timeArr[position[i]] = (double)(target - position[i]) / speed[i];
    }

    double prev = 0.0;
    for (int i = target - 1; i >= 0; i--) {
        double cur = timeArr[i];
        if (cur > prev) {
            prev = cur;
            res++;
        }
    }

    free(timeArr);
    return res;
}
```

1.25 Match Sticks To Square

```
bool canMake(int index, int bucketsize, int* matchsticks, int* bucket, int
matchstickSize) {
    if (index == matchstickSize)
        return bucket[0] == bucket[1] && bucket[1] == bucket[2] && bucket[2] ==
bucket[3];

    for (int i = 0; i < 4; i++) {
        if (bucket[i] + matchsticks[index] > bucketsize)
            continue;

        int j = i;
        while (--j >= 0)
            if (bucket[i] == bucket[j])
                break;

        if (j != -1)
            continue;

        bucket[i] += matchsticks[index];
        if (canMake(index + 1, bucketsize, matchsticks, bucket, matchstickSize))
            return true;

        bucket[i] -= matchsticks[index];
```

```c
    }

    return false;
}

bool makesquare(int* matchsticks, int matchstickSize) {
    int sum = 0;

    for (int i = 0; i < matchstickSize; i++)
        sum += matchsticks[i];

    if (sum == 0 || sum % 4)
        return false;

    for (int i = 0; i < matchstickSize - 1; i++) {
        for (int j = 0; j < matchstickSize - i - 1; j++) {
            if (matchsticks[j] < matchsticks[j + 1]) {
                // swap
                int temp = matchsticks[j];
                matchsticks[j] = matchsticks[j + 1];
                matchsticks[j + 1] = temp;
            }
        }
    }

    int bucket[4] = {0};
    return canMake(0, sum / 4, matchsticks, bucket, matchstickSize);
}
```

1.26 132 Pattern

```c
bool find132pattern(int* nums, int numsSize) {
    if (numsSize < 3) {
        return false;
    }

    int* min = malloc(numsSize * sizeof(int));
    min[0] = nums[0];

    for (int i = 1; i < numsSize; i++) {
        min[i] = fmin(min[i - 1], nums[i]);
    }

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

    for (int j = numsSize - 1; j >= 0; j--) {
        if (nums[j] > min[j]) {
            while (top != -1 && stack[top] <= min[j]) {
                stack[top--];
```

```c
            }

            if (top != -1 && stack[top] < nums[j]) {
                free(min);
                free(stack);
                return true;
            }

            stack[++top] = nums[j];
        }
    }

    free(min);
    free(stack);
    return false;
}
```

1.27 Word Search II

```c
/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
// Trie Node
typedef struct TrieNode {
    struct TrieNode* children[26];
    char* word;
} TrieNode;

// Trie Initialization
TrieNode* createTrieNode() {
    TrieNode* node = (TrieNode*)malloc(sizeof(TrieNode));
    memset(node->children, 0, sizeof(node->children));
    node->word = NULL;
    return node;
}

// Trie Insertion
void insertWord(TrieNode* root, char* word) {
    TrieNode* node = root;
    for (int i = 0; i < strlen(word); i++) {
        int index = word[i] - 'a';
        if (!node->children[index]) {
            node->children[index] = createTrieNode();
        }
        node = node->children[index];
    }
    node->word = word;
}
```

```c
// DFS to find words on the board
void findWordsDFS(char** board, int boardSize, int* boardColSize, int row, int col,
TrieNode* node, char*** result, int* returnSize) {
    char c = board[row][col];
    TrieNode* child = node->children[c - 'a'];

    if (child == NULL) {
        return;
    }

    if (child->word != NULL) {
        (*returnSize)++;
        *result = (char**)realloc(*result, (*returnSize) * sizeof(char*));
        (*result)[(*returnSize) - 1] = child->word;
        child->word = NULL; // Avoid duplicates
    }

    board[row][col] = '#'; // Mark visited

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

    for (int i = 0; i < 4; i++) {
        int newRow = row + directions[i][0];
        int newCol = col + directions[i][1];

        if (newRow >= 0 && newRow < boardSize && newCol >= 0 && newCol < boardColSize[0]
&& board[newRow][newCol] != '#') {
            findWordsDFS(board, boardSize, boardColSize, newRow, newCol, child, result,
returnSize);
        }
    }

    board[row][col] = c; // Restore the original character
}

// Find Words on the Board
char** findWords(char** board, int boardSize, int* boardColSize, char** words, int
wordsSize, int* returnSize) {
    *returnSize = 0;
    char** result = NULL;

    TrieNode* root = createTrieNode();

    // Insert words into the Trie
    for (int i = 0; i < wordsSize; i++) {
        insertWord(root, words[i]);
    }
```

```c
    // Traverse the board and search for words
    for (int i = 0; i < boardSize; i++) {
        for (int j = 0; j < boardColSize[i]; j++) {
            findWordsDFS(board, boardSize, boardColSize, i, j, root, &result,
returnSize);
        }
    }

    // Free Trie and return result
    free(root);

    return result;
}
```

1.28 Merge Intervals

Given an array of intervals where intervals[i] = [start, end], 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;
}
```

```c
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));

    // 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;
    }
```

```c
    // Clean up
    free(intervalsStruct);

    return result;
}
```

1.29 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--];
        }
    }
```

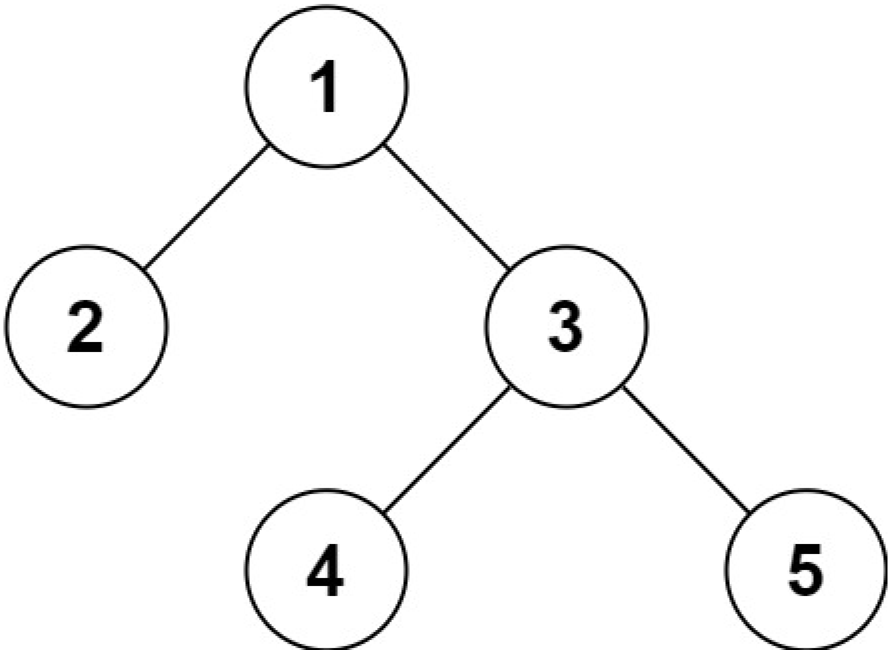
1.30 Serialize and Deserialize Binary Tree

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 {
        // 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) {
```

```c
            // 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) {
```

```c
        num /= 10;
        count++;
    }
    return count;
}
```

1.31 Bulb Switcher

```c
int bulbSwitch(int n) {
    return sqrt(n);
}
```

1.32 Sign Of The Product Of An Array

```c
int arraySign(int* nums, int numsSize) {
    int countNegatives = 0;
    int hasZero = 0;

    for (int i = 0; i < numsSize; i++) {
        if (nums[i] == 0) {
            hasZero = 1;
            break;
        } else if (nums[i] < 0) {
            countNegatives++;
        }
    }

    if (hasZero) {
        return 0;
    }

    return countNegatives % 2 == 0 ? 1 : -1;
}
```

1.33 Excel Sheet Column Title

```c
char* convertToTitle(int columnNumber) {
    // Calculate the length of the result string
    int len = 0;
    int temp = columnNumber;
    while (temp > 0) {
        len++;
        temp = (temp - 1) / 26;
    }

    // Allocate memory for the result string
    char* result = (char*)malloc((len + 1) * sizeof(char));
    result[len] = '\0';

    // Fill in the result string from right to left
```

```
    for (int i = len - 1; i >= 0; i--) {
        result[i] = (columnNumber - 1) % 26 + 'A';
        columnNumber = (columnNumber - 1) / 26;
    }

    return result;
}
```

1.34 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
Output: "ps"

C :

```
struct Stack {
    int *arr;
    int top;
    int capacity;
};

// Function to initialize a stack
```

```c
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;
}
```

1.35 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 :

```
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);

}
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;
}
```

1.36 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.

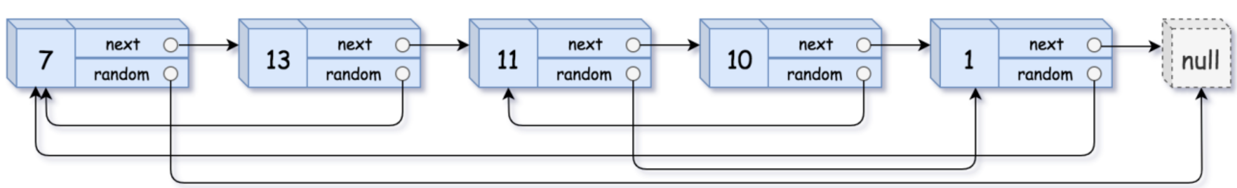
Return *the head of the copied linked list*.

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.

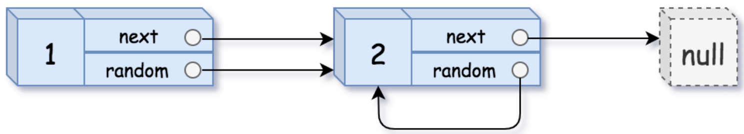
Your code will **only** be given the head of the original linked list.

Example 1:



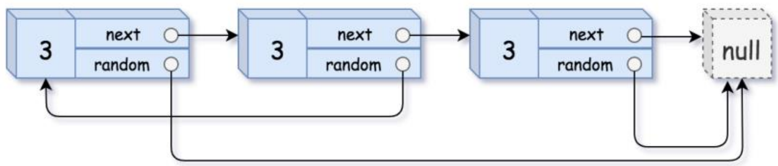
Input: head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
Output: [[7,null],[13,0],[11,4],[10,2],[1,0]]

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;
    }
    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;
}
```

1.37 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].

Example 2:

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

Example 3:

Input: nums = [3,3], target = 6
Output: [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;
}

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;
}
```

1.38 Flood Fill

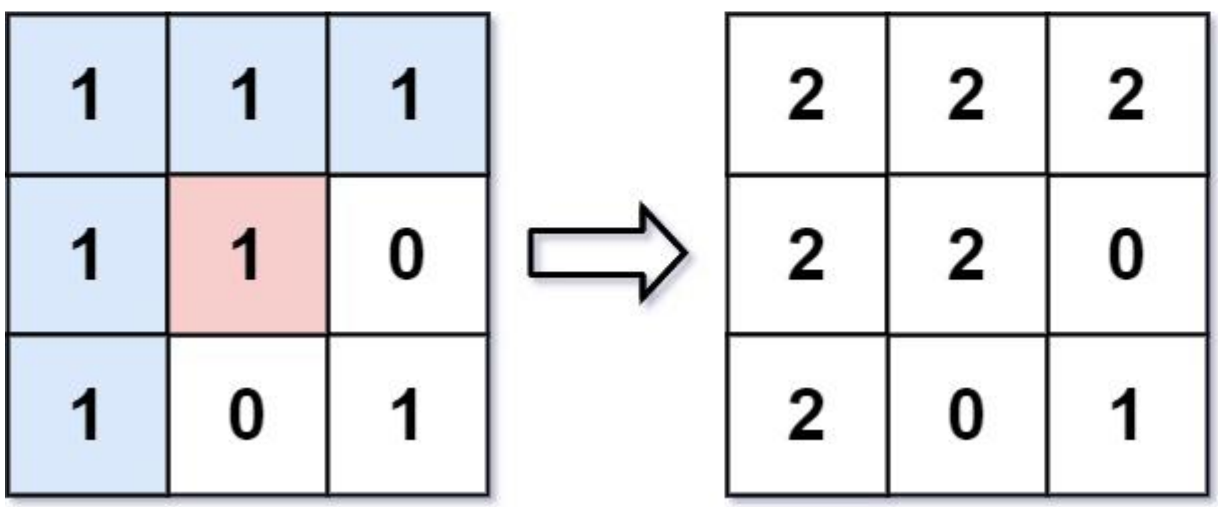
An image is represented by an m x n integer grid image where image[i][j] represents the pixel value of the image.

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
    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
```

```c
    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;
}
```

1.39 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) {
```

```c
        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;
}
```

1.40 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++;
        }
    }
}
```

1.41 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)).

Example 1:

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

Example 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 :

```
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;
```

```c
    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];
    }
}
```

1.42 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;
}
```

1.43 Meeting Rooms II – Minimum Rooms Required

Given an array of meeting time intervals intervals where intervals[i] = [start, end], 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

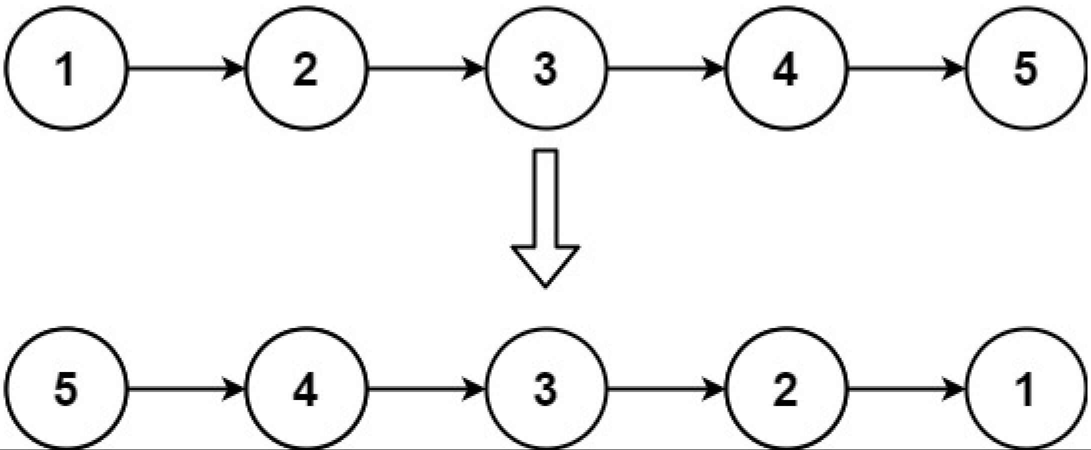
C :

```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;
}
```

1.44 Reverse Linked List

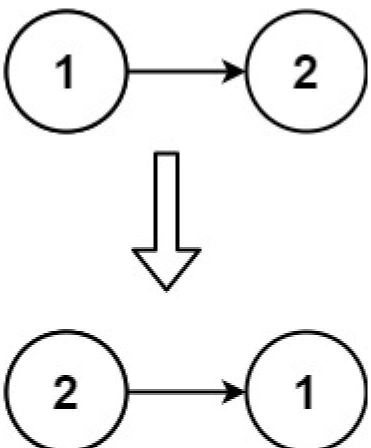
Given the head of a singly linked list, reverse the list, and return *the reversed list*.

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 :

```
/**
 * 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;
}
```

1.45 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;
}

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
    }
```

```c
    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);
}
```

1.46 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 :

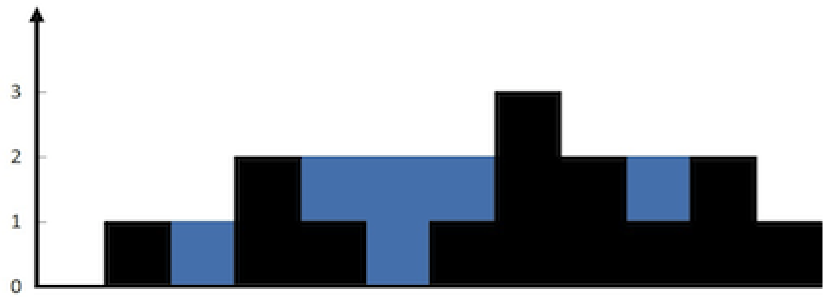
```c
typedef struct {
    int* queue;
    int front;
    int rear;
} MyStack;

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];
}

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);
*/
```

1.47 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 :

```
int trap(int* height, int heightSize) {
          int left = 0, right = heightSize - 1;
         int ans = 0;
         int leftMax = 0, rightMax = 0;

         while (left < right) {
             if (height[left] < height[right]) {
                 if (height[left] >= leftMax) {
                     leftMax = height[left];
                 } else {
                     ans += (leftMax - height[left]);
                 }
                 left++;
             } else {
                 if (height[right] >= rightMax) {
                     rightMax = height[right];
                 } else {
                     ans += (rightMax - height[right]);
                 }
                 right--;
             }
         }

         return ans;

}
```

1.48 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.

Example 2:

Input: s = "bbbbb"
Output: 1
Explanation: The answer is "b", with the length of 1.

Example 3:

Input: s = "pwwkew"
Output: 3
Explanation: The answer is "wke", with the length of 3.
Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

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>();
            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;
        }
    }
}
```

1.49 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 :

```
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;
}
```

1.50 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

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;
}
```

1.51 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 :

```
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]};
```

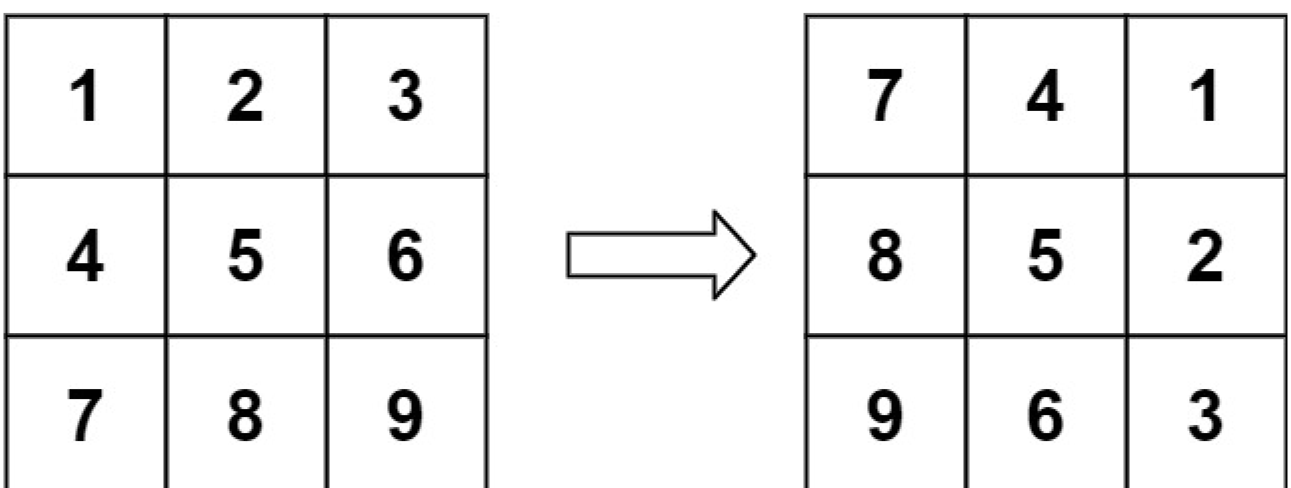
```
        }
    }

    // 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;
}
```

1.52 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.

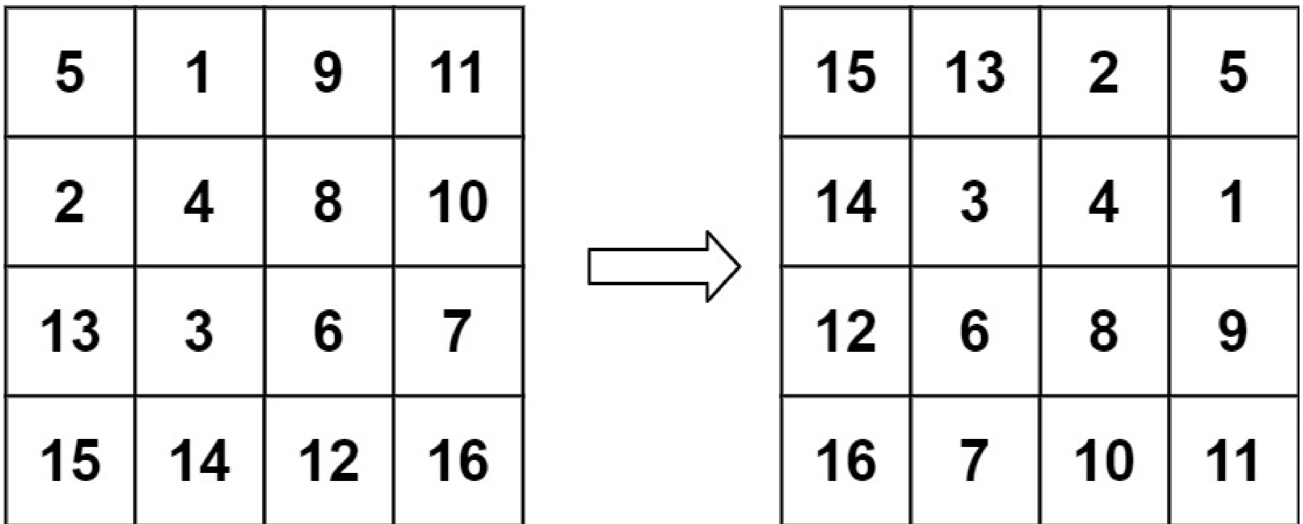
Example 1:



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 :

```
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);
}
```

1.53 Gas Station

There are n gas stations along a circular route, where the amount of gas at the i^{th} 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;
}
```

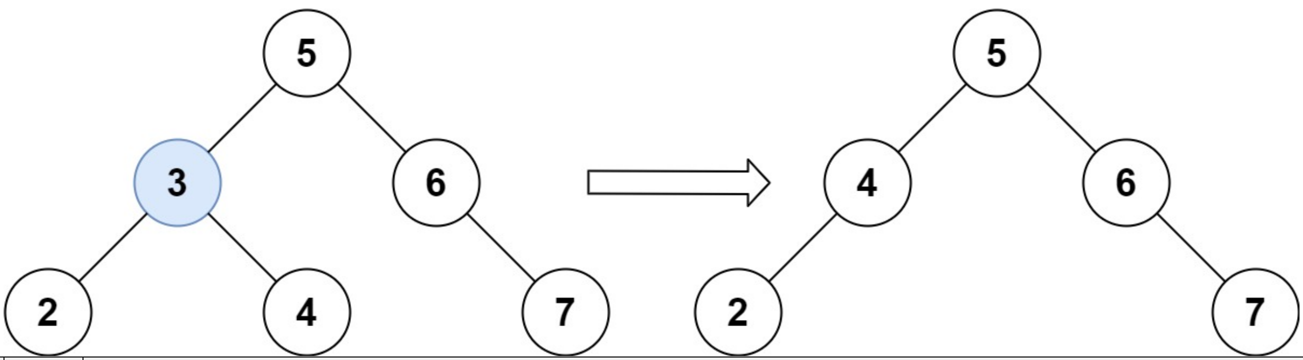
1.54 Delete Node in BST

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*.

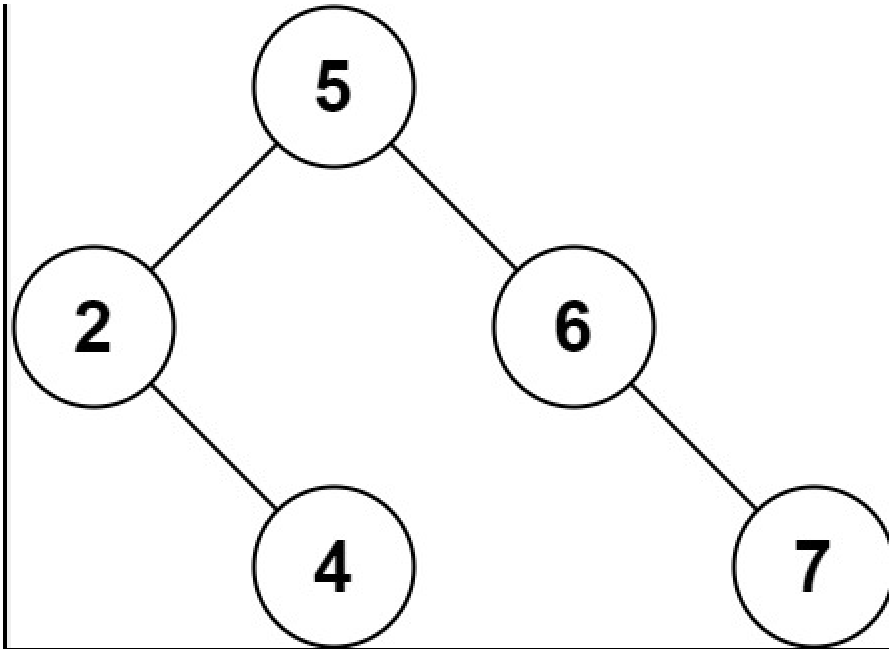
Basically, the deletion can be divided into two stages:

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.



Example 2:

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

Example 3:

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

Constraints:

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;
    }
```

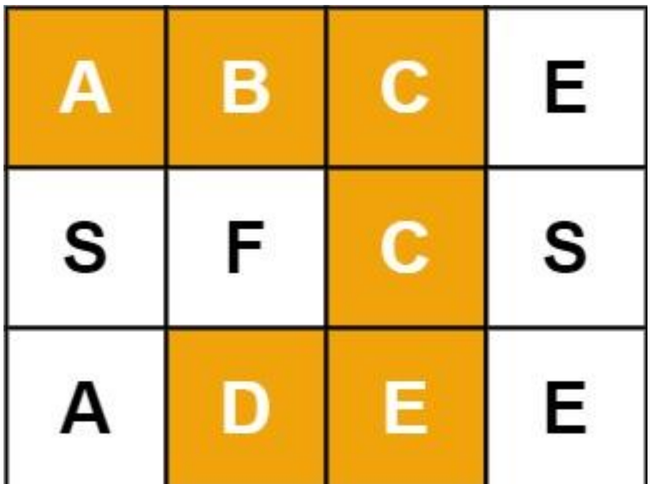
```c
    return node;
}

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);
    }
}
```

1.55 Word Search

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

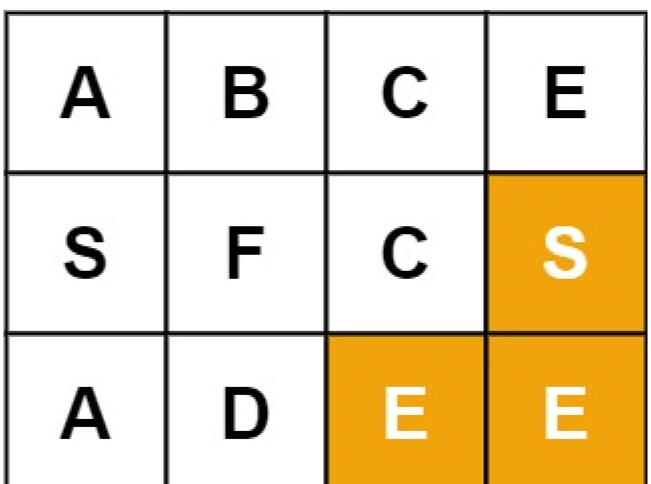
The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.

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 :

```
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;
}
```

1.56 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.

Example 2:

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 :

```
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;
}
```

1.57 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

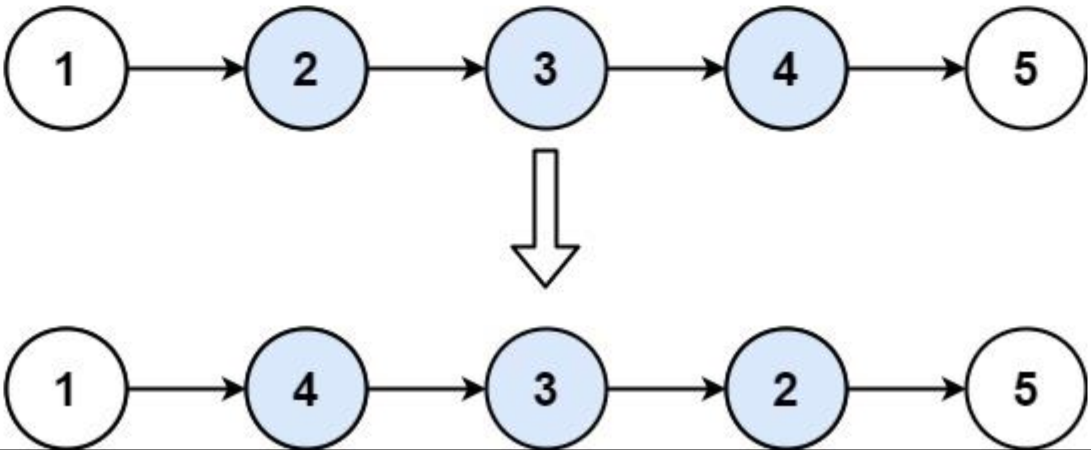
C :

```
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;
}
```

1.58 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 :

```
/**
 * 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;
}
```

1.59 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);
*/
```

1.60 Odd Even Linked List

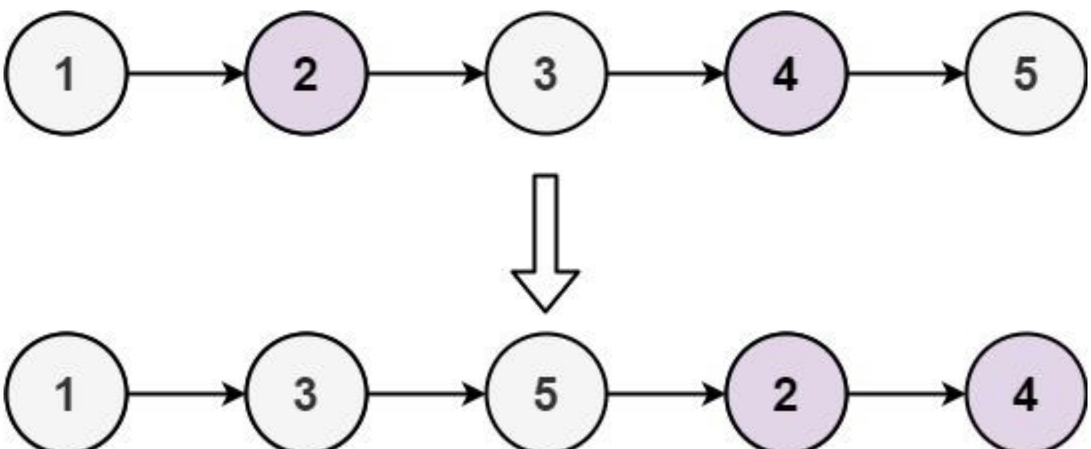
Given the head of a singly linked list, group all the nodes with odd indices together followed by the nodes with even indices, and return *the reordered list*.

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.

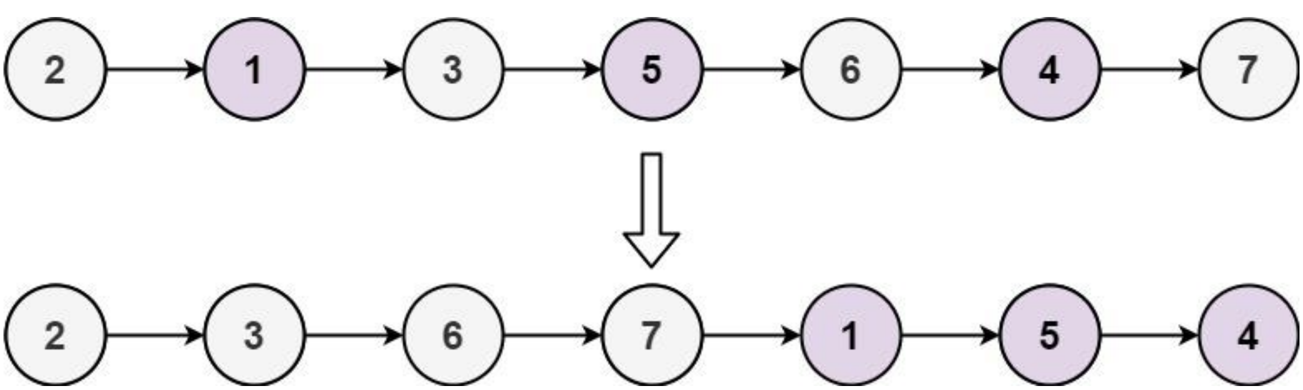
You must solve the problem in O(1) extra space complexity and O(n) time complexity.

Example 1:



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

Example 2:



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

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;
}
```

1.61 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 i^{th} day.

You want to maximize your profit by choosing a **single day** to buy one stock and choosing a **different day in the future** to sell that stock.

Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return 0.

Example 1:

Input: prices = [7,1,5,3,6,4]
Output: 5
Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.
Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

Example 2:

Input: prices = [7,6,4,3,1]
Output: 0
Explanation: In this case, no transactions are done and the max profit = 0

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;
}
```

1.62 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 :

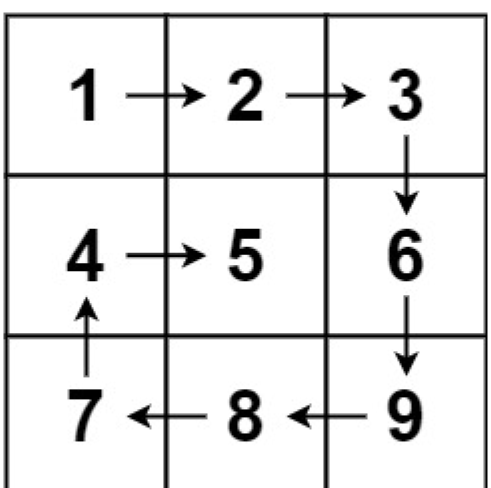
```
// HashSet structure
typedef struct {
    int* array;
    int size;
} HashSet;

// 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;
}
```

1.63 Spiral Matrix

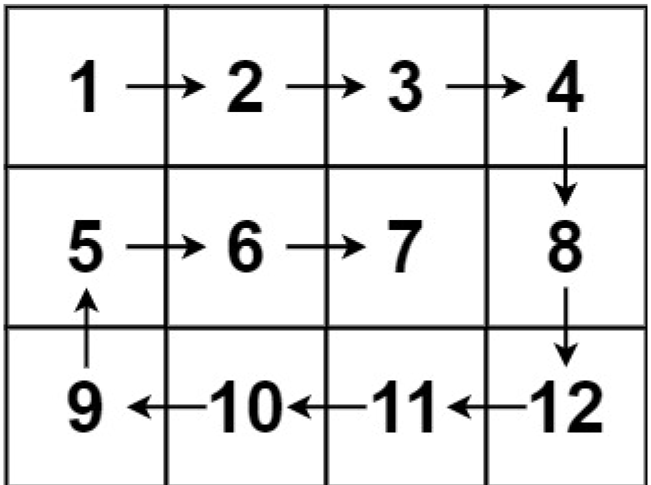
Given an m x n matrix, return *all elements of the* matrix *in spiral order*.

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 :

```
/**
 * 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;
}
```

1.64 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:

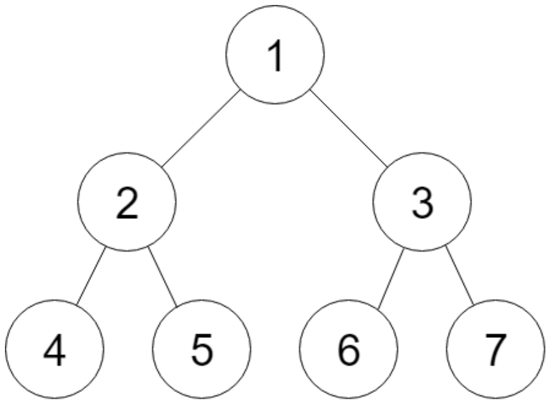


Figure A

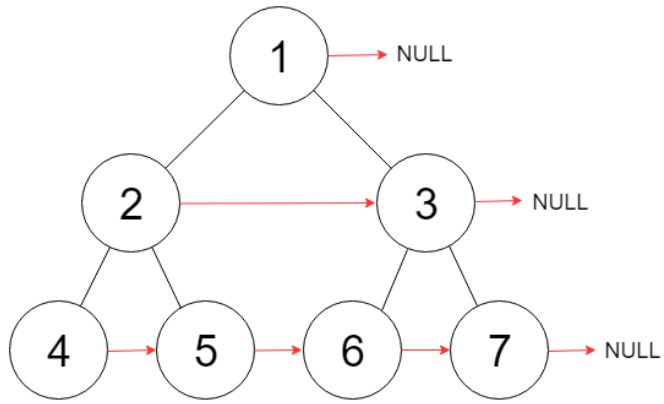


Figure B

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 :

```
/**
 * 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;
}
```

1.65 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".

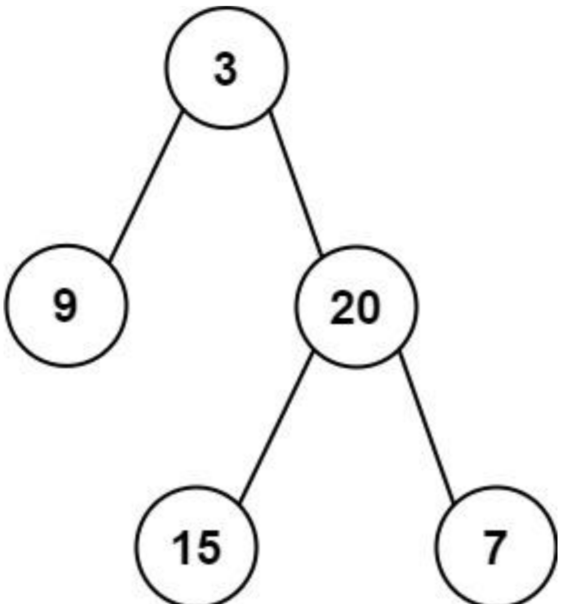
C:

```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;
}
```

1.66 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);
}
```

1.67 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:

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

Input: board =
[["5","3",".",".","7",".",".",".","."]
,["6",".",".","1","9","5",".",".","."]
,[".","9","8",".",".",".",".","6","."]
,["8",".",".",".","6",".",".",".","3"]
,["4",".",".","8",".","3",".",".","1"]
,["7",".",".",".","2",".",".",".","6"]
,[".","6",".",".",".",".","2","8","."]
,[".",".",".","4","1","9",".",".","5"]
,[".",".",".",".","8",".",".","7","9"]]
Output: true

Example 2:

Input: board =
[["8","3",".",".","7",".",".",".","."]
,["6",".",".","1","9","5",".",".","."]
,[".","9","8",".",".",".",".","6","."]
,["8",".",".",".","6",".",".",".","3"]
,["4",".",".","8",".","3",".",".","1"]
,["7",".",".",".","2",".",".",".","6"]
,[".","6",".",".",".",".","2","8","."]
,[".",".",".","4","1","9",".",".","5"]
,[".",".",".",".","8",".",".","7","9"]]
Output: false

Explanation: Same as Example 1, except with the **5** in the top left corner being modified to **8**. Since there are two 8's in the top left 3x3 sub-box, it is invalid.

C :

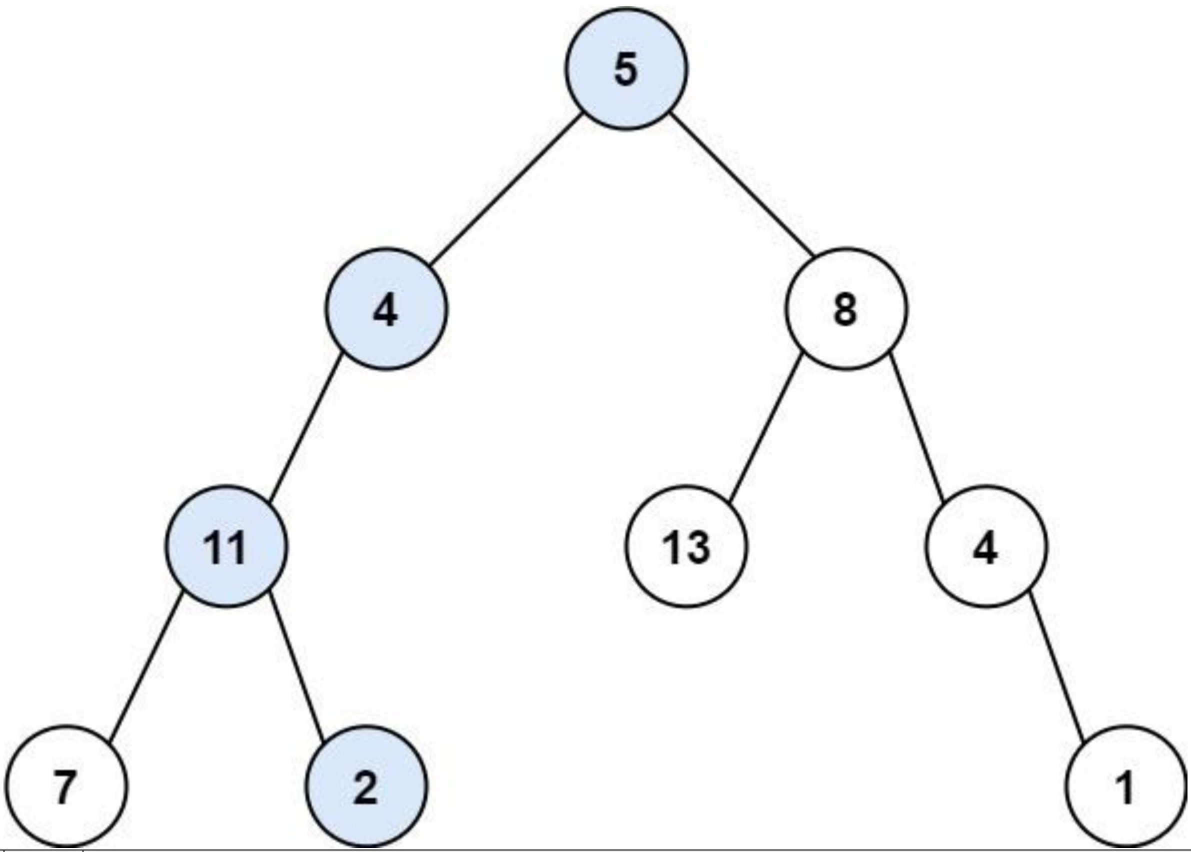
```
bool isValidSudoku(char** board, int boardSize, int* boardColSize) {
    // Arrays to check for duplicate digits in rows, columns, and sub-boxes
    bool row[9][9] = {false};
    bool col[9][9] = {false};
    bool box[9][9] = {false};

    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;
}
```

1.68 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.

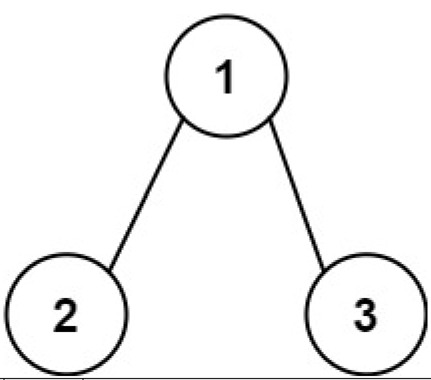
A **leaf** is a node with no children.

Example 1:



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

Example 2:



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

Example 3:

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

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;
}
```

1.69 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);
*/
```

1.70 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:

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

Input: board = [["5","3",".",".","7",".",".",".","."],["6",".",".","1","9","5",".",".","."],[".","9","8",".",".",".",".","6","."],["8",".",".",".","6",".",".",".","3"],["4",".",".","8",".","3",".",".","1"],["7",".",".",".","2",".",".",".","6"],[".","6",".",".",".",".","2","8","."],[".",".",".","4","1","9",".",".","5"],[".",".",".",".","8",".",".","7","9"]]

Output: [["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);
}
```

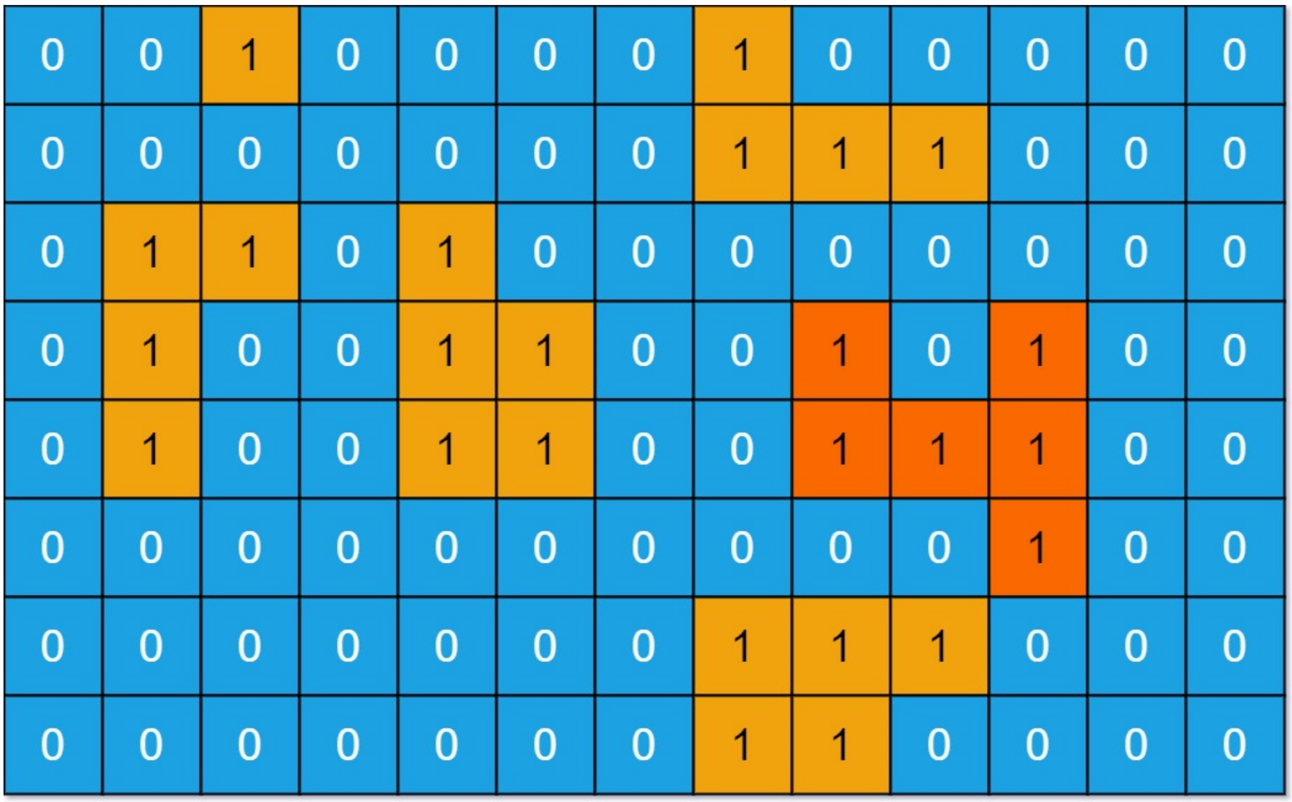
1.71 Max Area Of Island

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.

Example 1:



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 :

```
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;
}
```

1.72 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 = "(]"

Output: false

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;
}
```

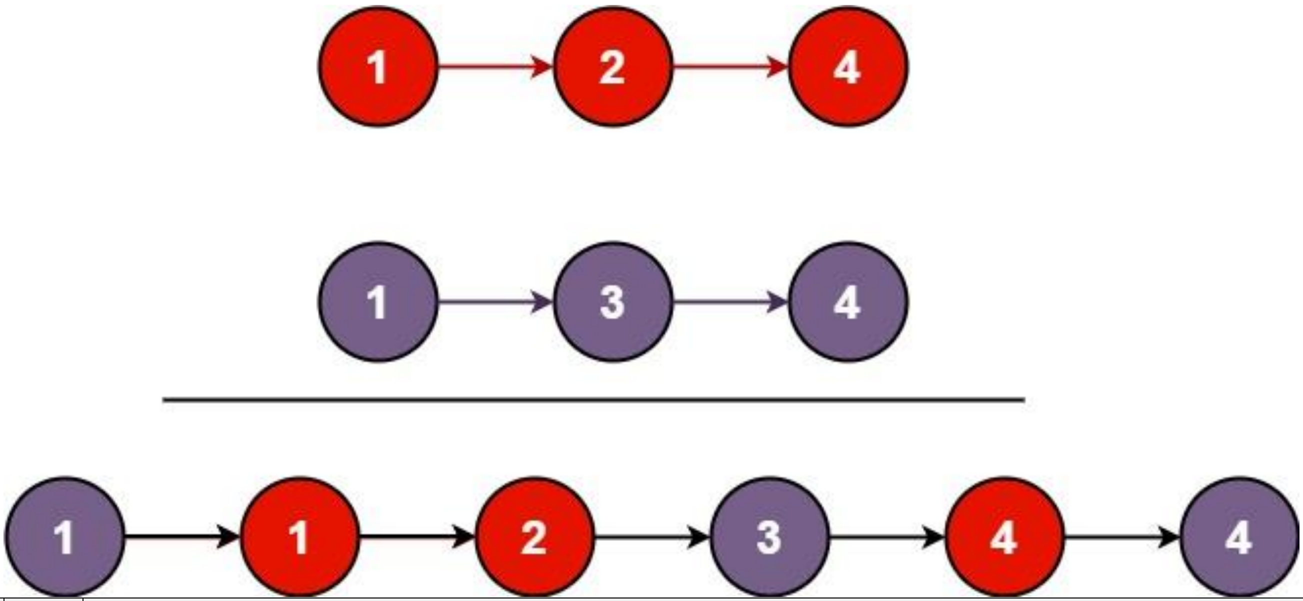
1.73 Merge Two Sorted Lists

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 :

```
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;
}
```

1.74 Container With Most Water

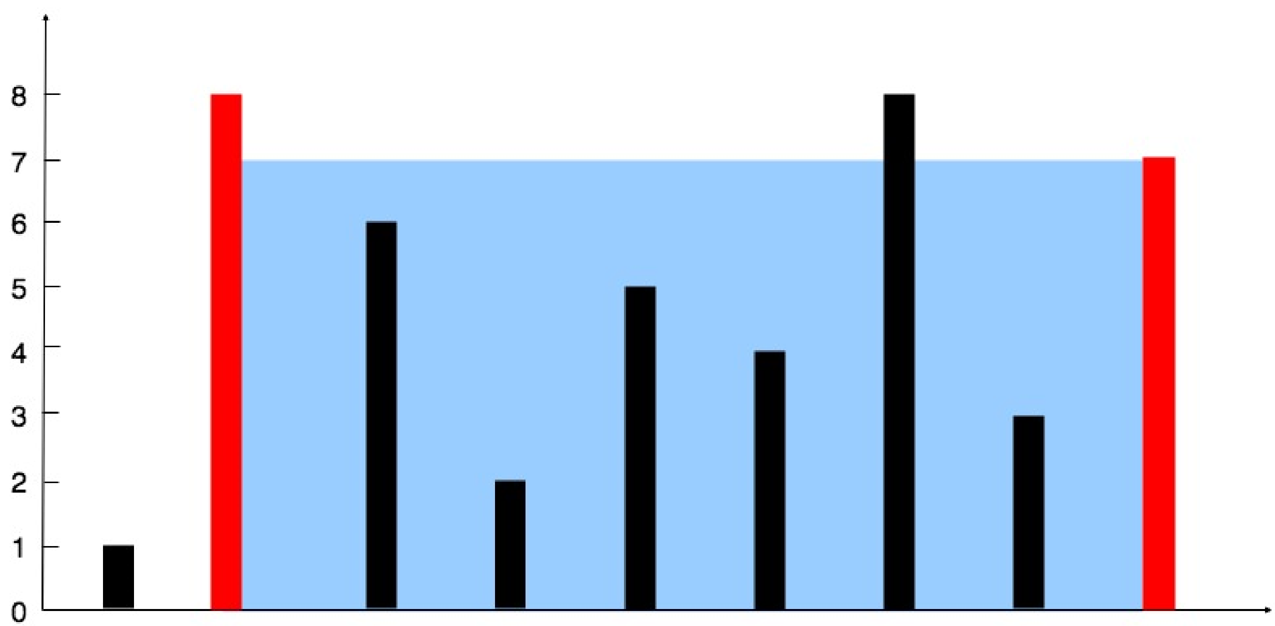
You are given an integer array height of length n. There are n vertical lines drawn such that the two endpoints of the ith line are (i, 0) and (i, height[i]).

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

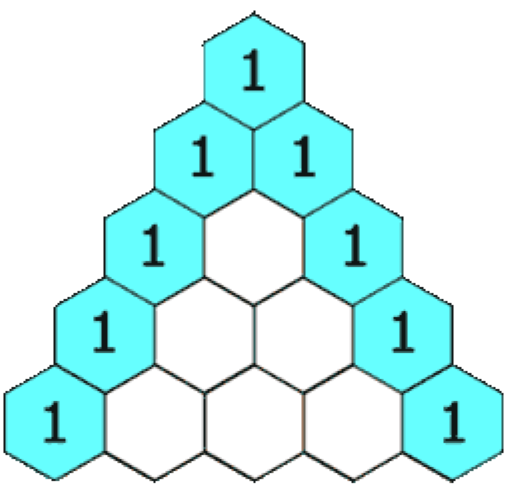
C :

```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;
}
```

1.75 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 :

```
/**
 * 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;
}
```

1.76 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 :

```
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;
}
```

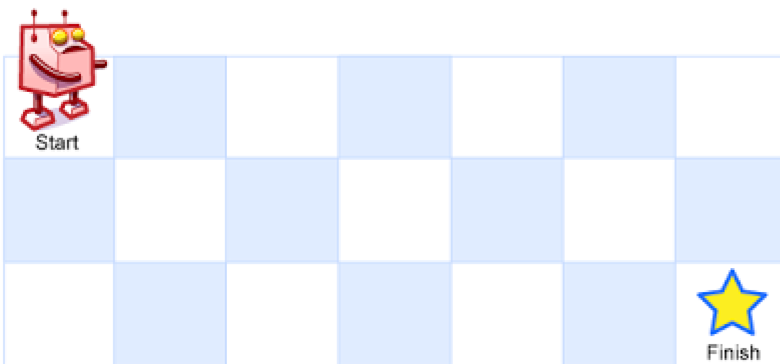
1.77 <u>Unique Paths</u>

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▯.

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

<u>C :</u>

```
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];
}
```

1.78 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;
}
```

1.79 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"]

Output: [["a"]]

C :

```
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());
    }
}
```

1.80 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 :

```
class Solution {
    public String longestPalindrome(String s) {
        int n = s.length();
        boolean[][] dp = new boolean[n][n];

        // All substrings of length 1 are palindromes
        for (int i = 0; i < n; i++) {
            dp[i][i] = true;
        }

        int start = 0; // Start index of the longest palindrome
        int maxLength = 1; // Length of the longest palindrome

        // Check substrings of length 2
        for (int i = 0; i < n - 1; i++) {
            if (s.charAt(i) == s.charAt(i + 1)) {
                dp[i][i + 1] = true;
                start = i;
                maxLength = 2;
            }
        }

        // Check substrings of length 3 and more
        for (int len = 3; len <= n; len++) {
            for (int i = 0; i <= n - len; i++) {
                int j = i + len - 1;
                if (dp[i + 1][j - 1] && s.charAt(i) == s.charAt(j)) {
                    dp[i][j] = true;
                    start = i;
                    maxLength = len;
                }
            }
        }

        return s.substring(start, start + maxLength);
    }
}

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++) {
```

```c
        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];
        }
    }

    // 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;
}
```

1.81 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 :

```
/**
 * 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().
```

```c
 */
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
*/
```

1.82 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 :

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

    return r;
}
```

1.83 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"
Output: ["a","b","c"]

C :

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

```c
 */
// 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;
    }

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

```java
    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;
    }
}
```

1.84 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.

Example 1:

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.

Example 2:

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

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;
}
```

1.85 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 ith 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 :

```
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;
}
```

1.86 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]
Output: [9,4]
Explanation: [4,9] is also accepted.

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;
}
```

1.87 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

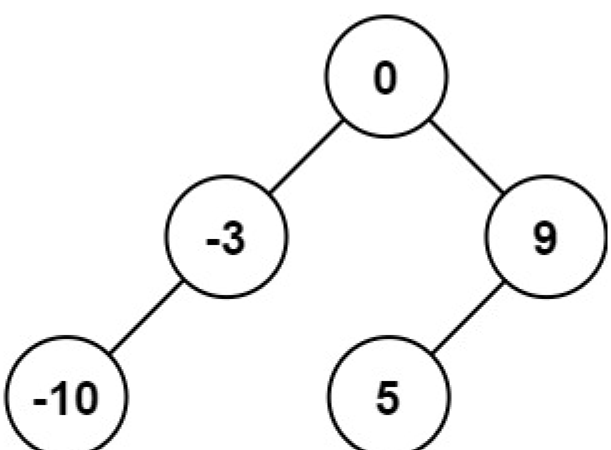
C :

```
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];

}
```

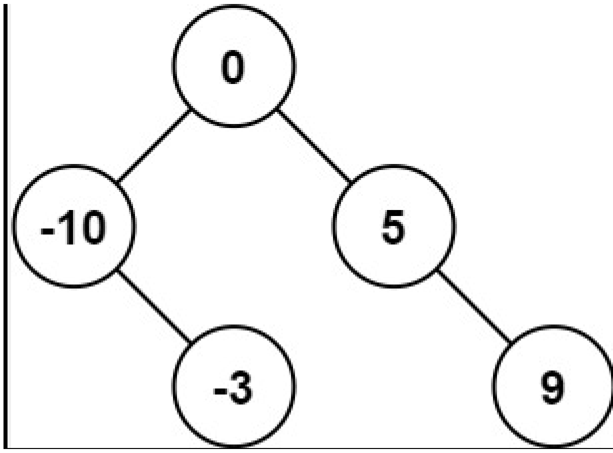
1.88 Convert Sorted Array To Binary Search Tree

binary search tree.

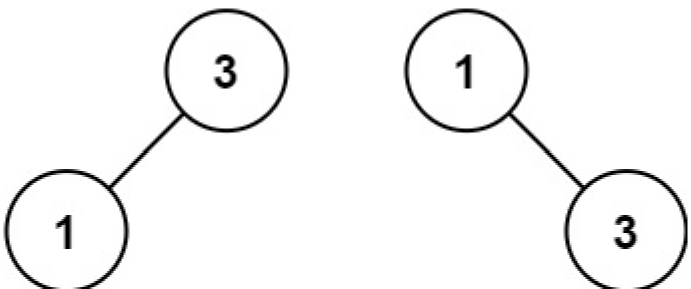
Example 1:



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:



Example 2:



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);
}
```

1.89 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

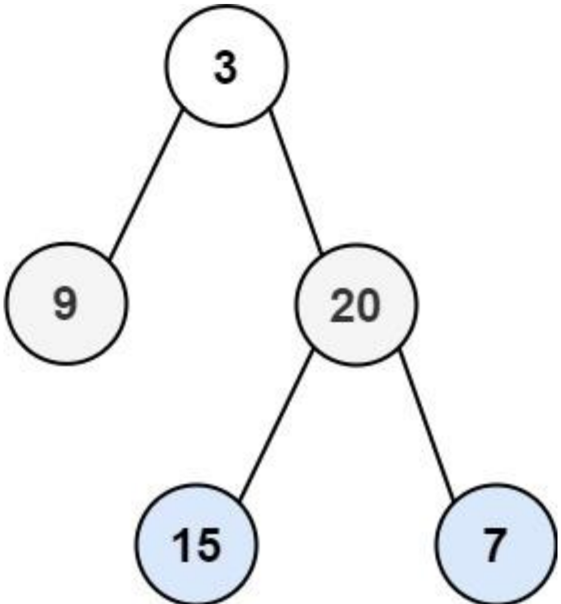
C :

```
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();
    }
}
```

1.90 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 :

```
/**
 * 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;
    }
}
```

1.91 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:

```
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;
}
```

1.92 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:

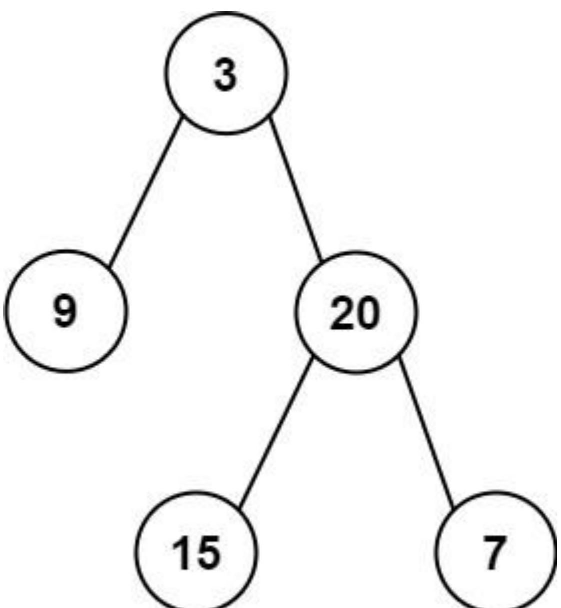
```
int search(int* nums, int numsSize, int target) {
        int left = 0;
    int right = numsSize - 1;
```

```
    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;
}
```

1.93 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]

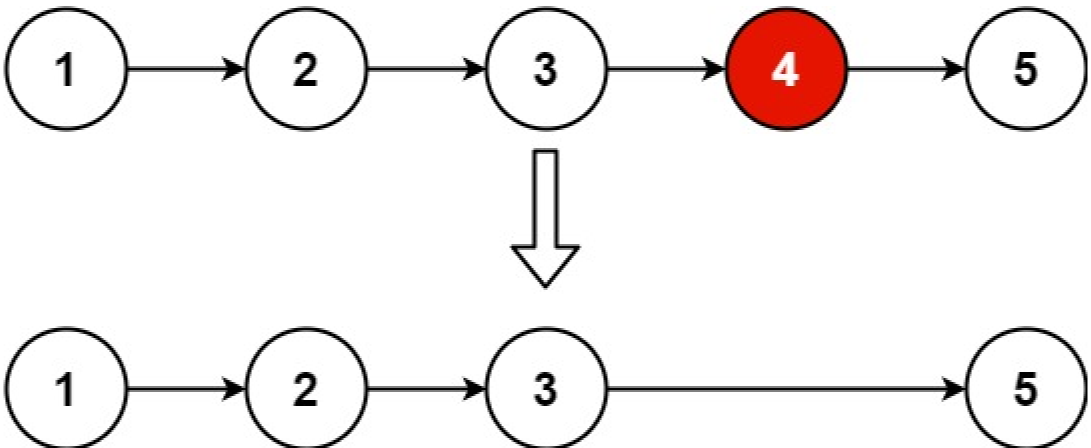
C:

```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);
}
```

1.94 Remove Nth Node From End Of List

Given the head of a linked list, remove the n^{th} node from the end of the list and return its head.

Example 1:



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

Example 2:

Input: head = [1], n = 1
Output: []

Example 3:

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

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;
```

```
    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;
    */
}
```

1.95 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.

```
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);
    }
}
bool canJump(int* nums, int numsSize) {
    int maxReach = 0;
```

```c
    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;
}
```

1.96 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].

```
randomizedSet.getRandom(); // getRandom() should return either 1 or 2 randomly.
randomizedSet.remove(1); // Removes 1 from the set, returns true. Set now contains [2].
randomizedSet.insert(2); // 2 was already in the set, so return false.
randomizedSet.getRandom(); // Since 2 is the only number in the set, getRandom() will always return 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
```

```c
    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);
*/
```

1.97 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* i^{th} *day to get a warmer temperature*. If there is no future day for which this is possible, keep answer[i] == 0 instead.

Example 1:

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

Example 2:

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

Example 3:

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

```
/**
 * 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;
}
```

1.98 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

Example 2:

Input: x = -123
Output: -321

Example 3:

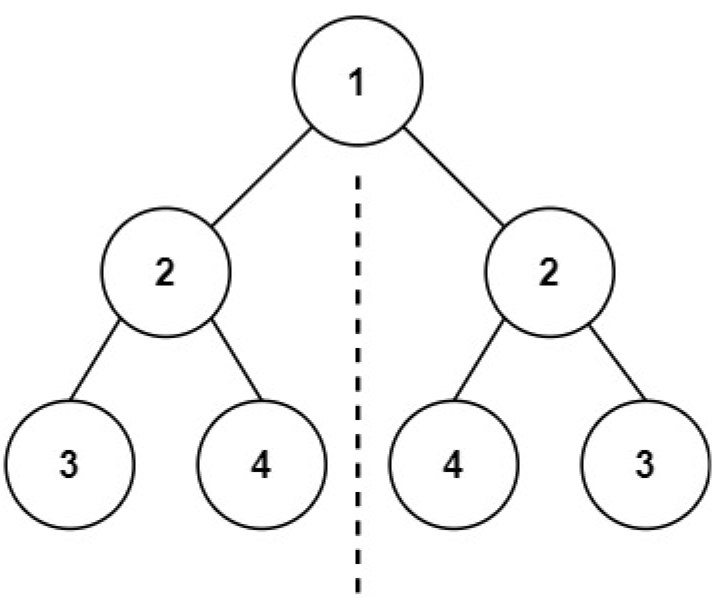
Input: x = 120
Output: 21

```
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;
    }
}
```

1.99 Symmetric Tree

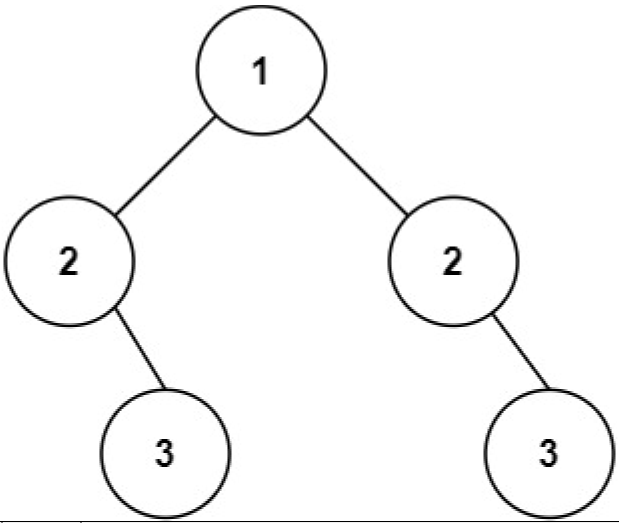
Given the root of a binary tree, *check whether it is a mirror of itself* (i.e., symmetric around its center).

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

```
/**
 * 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;
        }
```

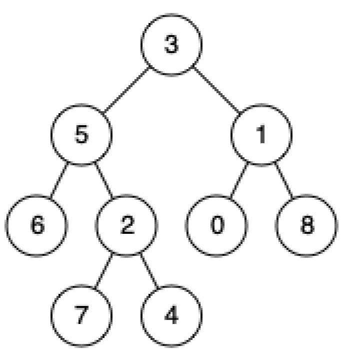
```
        if (left == NULL || right == NULL) {
            return false;
        }

        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;
}
```

1.100 Lowest Common Ancestor Of A binary tree

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

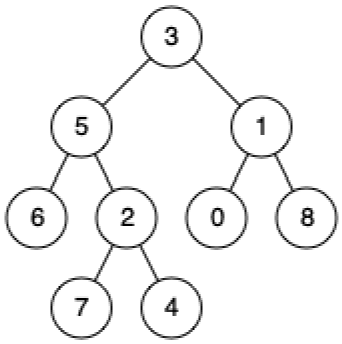
According to the definition of LCA on Wikipedia: "The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow **a node to be a descendant of itself**)."

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

```
/**
 * 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;
}
```

1.101 <u>Merge K Sorted Lists</u>

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;
}
```

1.102 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;
}
```

1.103 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;
}
```

1.104 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:

```
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;
}
```

1.105 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

```
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;
}
```

1.106 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];
    }
```

```
    // 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;
}
```

1.107 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

```
public static int coinChange(int[] S, int sum)
    {
        if (sum == 0) {
            return 0;
        }
        if (sum < 0) {
```

```
            return Integer.MAX_VALUE;
        }
        int coins = Integer.MAX_VALUE;
        for (int c: S)
        {
            int result = coinChange(S, sum - c);

            if (result != Integer.MAX_VALUE) {
                coins = Integer.min(coins, result + 1);
            }
        }
        return coins;
    }
```

```
#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];
}
```

1.108 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));

    // 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;
}
```

1.109 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

```
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;

}
```

1.110 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:

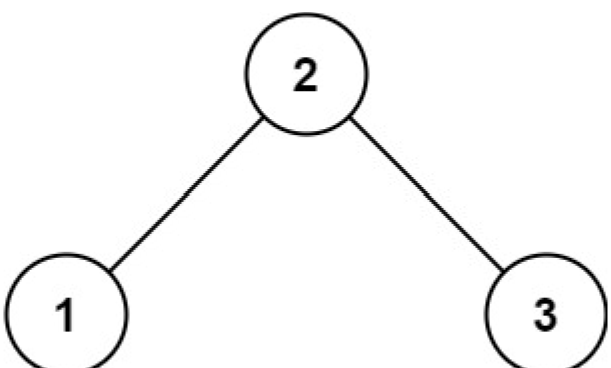
- The left

  subtree

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

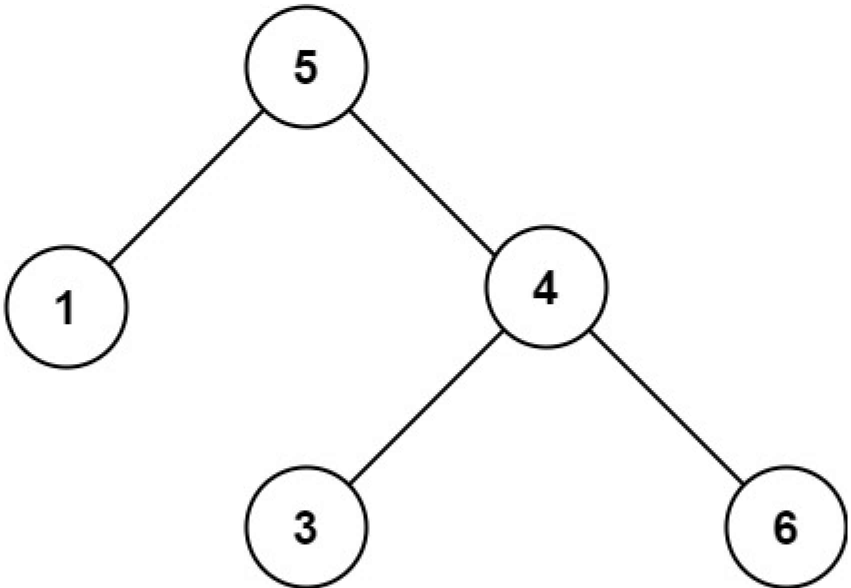
- The right subtree of a node contains only nodes with keys **greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.

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);
}
```

1.111 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;
}
```

1.112 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

```
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;
}
```

1.113 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.

Example 3:

Input: s = " "
Output: true
Explanation: s is an empty string "" after removing non-alphanumeric characters.
Since an empty string reads the same forward and backward, it is a palindrome.

```
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;
}
```

1.114 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

Explanation: Rob house 1 (money = 2), rob house 3 (money = 9) and rob house 5 (money = 1).
Total amount you can rob = 2 + 9 + 1 = 12.

```
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];
}
```

1.115 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:

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

```
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;
    }
}
```

1.116 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]], [], [], []]
```

```
Output

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



Explanation

Solution solution = new Solution([1, 2, 3]);

solution.shuffle();    // Shuffle the array [1,2,3] and return its result.

                       // Any permutation of [1,2,3] must be equally likely to be re
turned.

                       // Example: return [3, 1, 2]

solution.reset();      // Resets the array back to its original configuration [1,2,3
]. Return [1, 2, 3]

solution.shuffle();    // Returns the random shuffling of array [1,2,3]. Example: re
turn [1, 3, 2]
```

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);
}
```

1.117 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);
}
```

1.118 Sum Of Two Integers

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

Example 1:

Input: a = 1, b = 2
Output: 3

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;
}
```

1.119 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;
    }}
```

1.120 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;
}
```

1.121 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;
}
```

1.122 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.

```
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;
}
```

1.123 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;
}
```

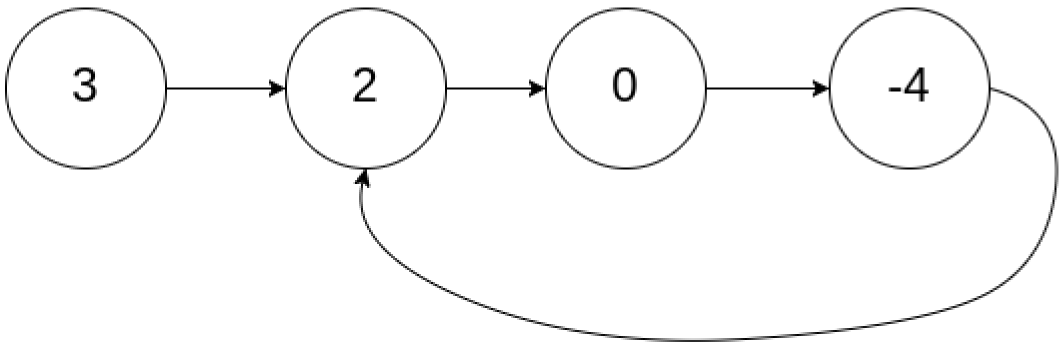
1.124 Detect Cycle in Linked List

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**.

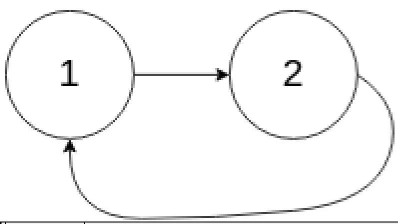
Return true *if there is a cycle in the linked list*. Otherwise, return false.

Example 1:



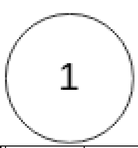
Input: head = [3,2,0,-4], pos = 1
Output: true
Explanation: There is a cycle in the linked list, where the tail connects to the 1st node (0-indexed).

Example 2:



Input: head = [1,2], pos = 0
Output: true
Explanation: There is a cycle in the linked list, where the tail connects to the 0th node.

Example 3:



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

```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;
}
```

1.125 Reorder List

You are given the head of a singly linked-list. The list can be represented as:

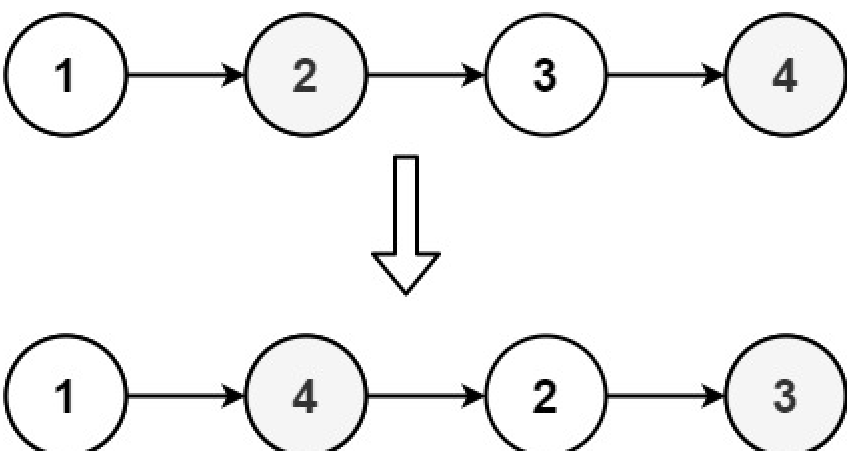
$L_0 \to L_1 \to \ldots \to L_{n-1} \to L_n$

Reorder the list to be on the following form:

$L_0 \to L_n \to L_1 \to L_{n-1} \to L_2 \to L_{n-2} \to \ldots$

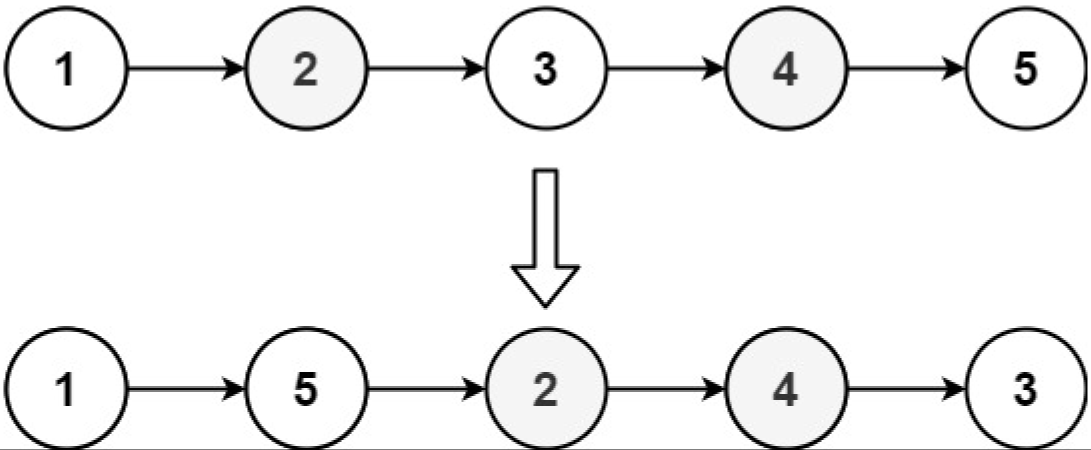
You may not modify the values in the list's nodes. Only nodes themselves may be changed.

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:

```
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;
```

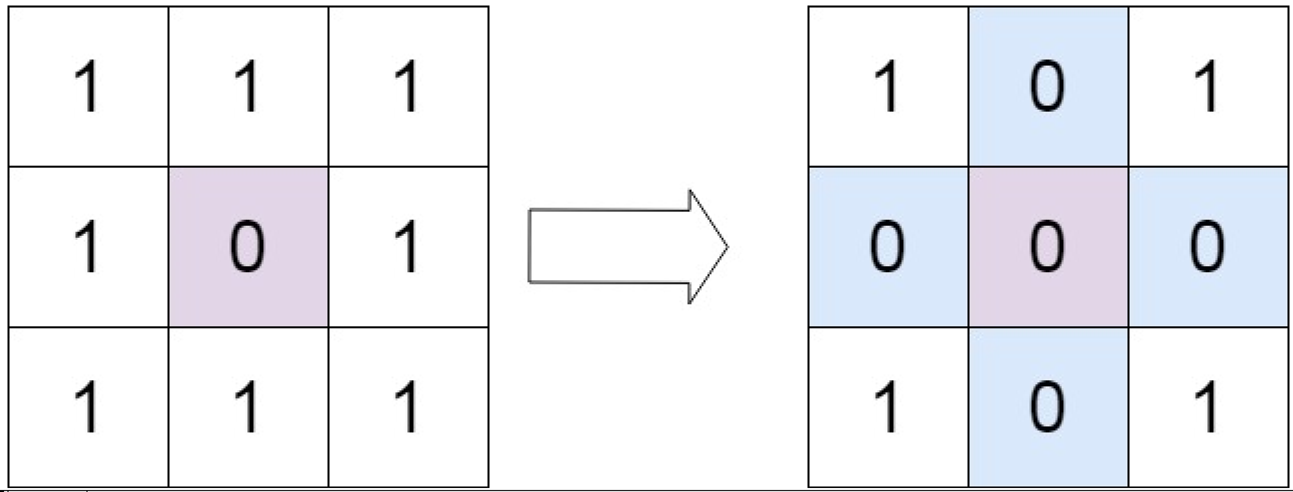
```
    while (fast != null && fast.next != null) {
      slow = slow.next;
      fast = fast.next.next;
    }

    // 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;
    }
  }
}
```

1.126 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.

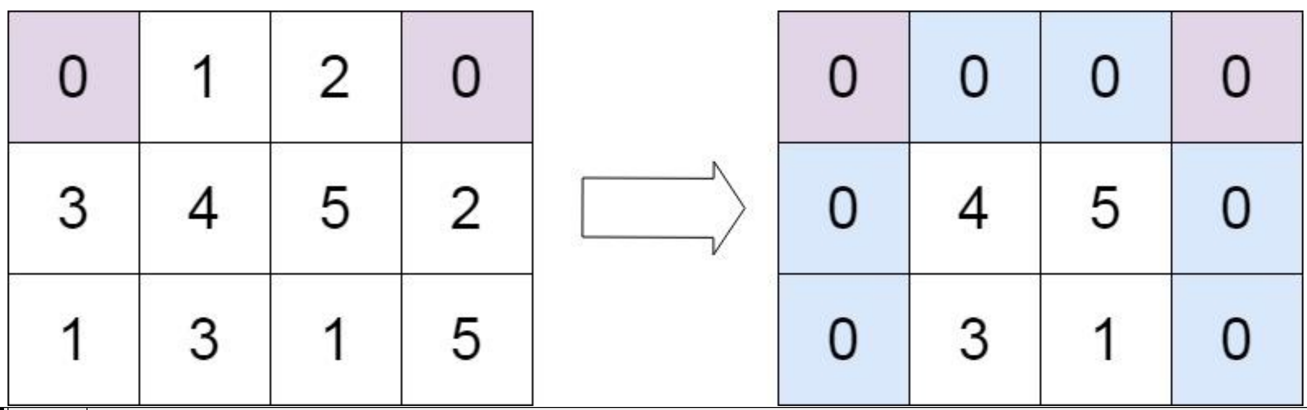
You must do it in place.

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]]

```
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;
                }
```

```
            }
        }
        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
*/
```

1.127 Insert Intervals

You are given an array of non-overlapping intervals intervals where intervals[i] = [start$_i$, end$_i$] represent the start and the end of the ith interval and intervals is sorted in ascending order by start$_i$. 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$_i$ 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];
```

```
    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;
```

```
}
```

1.128 Non Overlapping Intervals

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

Example 1:

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

Example 2:

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

```
// 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;
}
```

1.129 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.
}
```

1.130 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.

```
class Solution {
    public int characterReplacement(String s, int k) {
        // Initialising an empty array to store the count of the
        // characters in the given string s
```

```java
        int[] arr = new int[26];
        int res = 0;
        int max = 0;

        // The left pointer for the sliding window is l AND r is the
        // right pointer
        int l = 0;
        for (int r = 0; r < s.length(); r++) {
            // Counting the number of each character in the string s
            arr[s.charAt(r) - 'A']++;

            // Checking the character with max number of occurrence
            max = Math.max(max, arr[s.charAt(r) - 'A']);

            // Now we check if our current window is valid or not
            if (r - l + 1 - max > k) {
            // this means the no. of replacements is more than
            // allowed (k)
                // Decrementing the count of the character which was
                // at l because it is no longer in the window
                arr[s.charAt(l) - 'A']--;
                l++;
            }

            // The max our window can be
            res = Math.max(res, r - l + 1);
        }

        return res;
    }
}
```

```c
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;
}
```

1.131 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:

Input: s = "a", t = "aa"
Output: ""
Explanation: Both 'a's from t must be included in the window.
Since the largest window of s only has one 'a', return empty string.

```
class Solution {
    public String minWindow(String s, String t)
    {
        if(s.length() == 0 || t.length() == 0)
            return "";
        HashMap<Character,Integer> map = new HashMap<>();
        int n = s.length(),m=t.length();

        for(int i=0; i<m;i++)
            map.put(t.charAt(i) , map.getOrDefault(t.charAt(i), 0)+1);

        int count = map.size();
        int start =0,end =0,min =Integer.MAX_VALUE;
        String substring = "";
```

```
        while(end < n)
        {
            char ch = s.charAt(end);
            if(map.containsKey(ch))
            {
                map.put(ch , map.get(ch) -1);
                if(map.get(ch) == 0)
                    count--;
            }
            if(count > 0)
                end++;

            else if(count == 0)
            {
                while(count == 0)
                {
                    if(end-start +1 < min)
                    {
                        min = end-start+1;
                        substring = s.substring(start,end+1);
                    }
                    char temp = s.charAt(start);
                    if(map.containsKey(temp))
                    {
                        map.put(temp , map.get(temp)+1);
                        if(map.get(temp) == 1)
                           count++;
                    }
                    start++;
                }
                end++;
            }
        }
        return substring;
    }
}
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]]++;
    }
```

```c
    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;
}
```

1.132 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;
}
```

1.133 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().
 */
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] != '#') {
```

```c
            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);
```

1.134 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];
}
```

1.135 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

```java
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];
    }
}
```

1.136 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];
    }
}
```

1.137 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;
    }
}
```

1.138 Decode Ways

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;
```

```
  }
}
```

1.139 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<>();
    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());
```

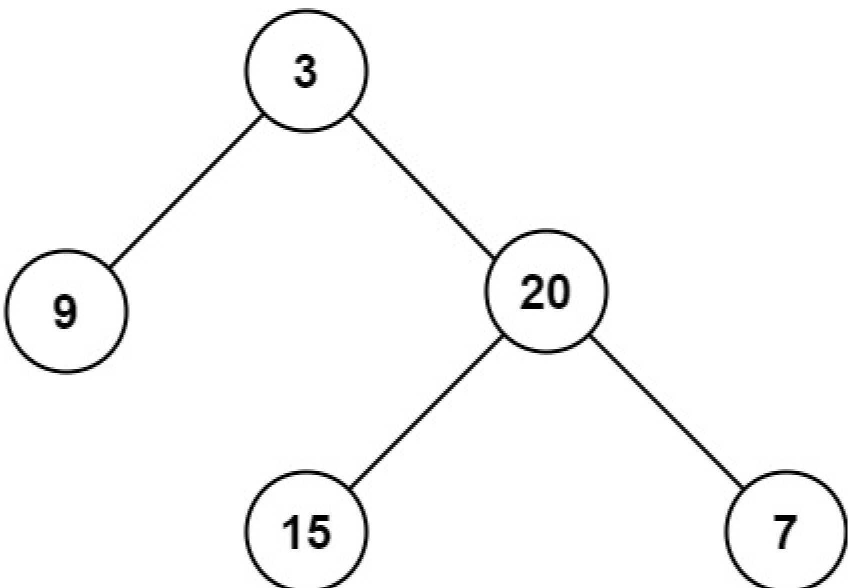
```java
    }
    even = !even;
    }

    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);
        }

    }
}
```

1.140 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

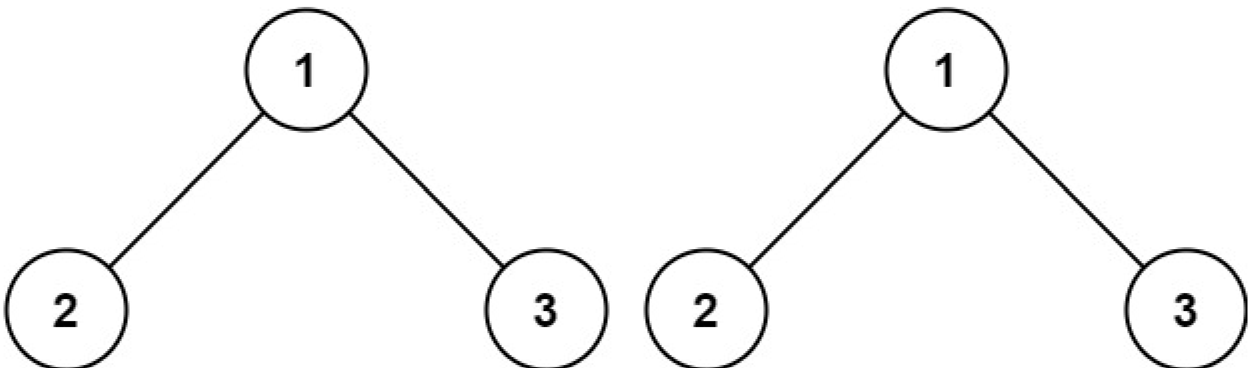
```
int max(int a, int b) {
    return (a > b) ? a : b;
}
int maxDepth(struct TreeNode* root){
    if (root == NULL) {
        return 0;
    }
    return max(1 + maxDepth(root -> left), 1 + maxDepth(root -> right));

}
```

1.141 Same Tree

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

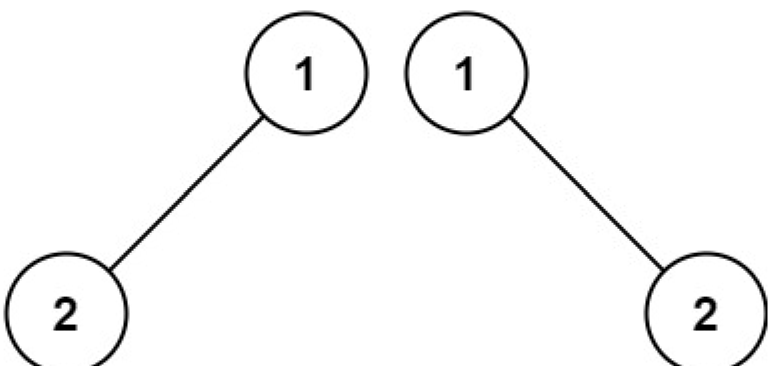
Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

Example 1:



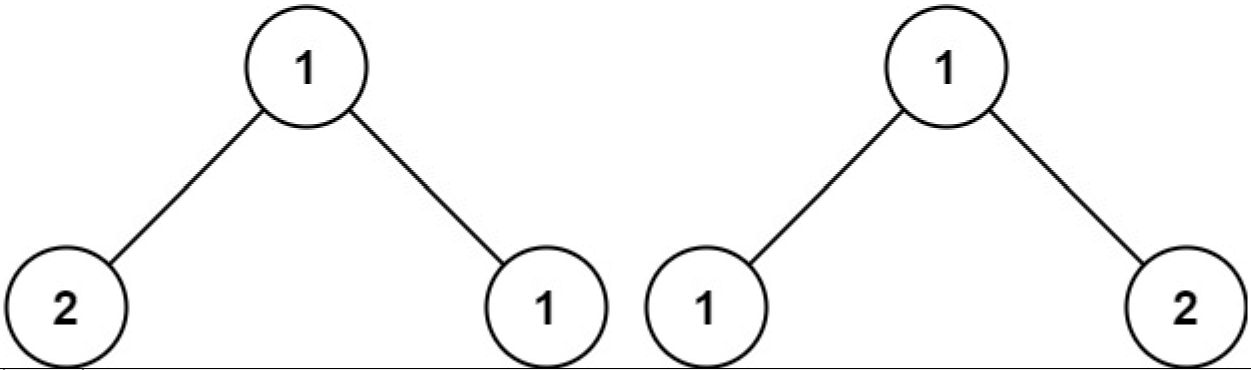
Input: p = [1,2,3], q = [1,2,3]
Output: true

Example 2:



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

Example 3:



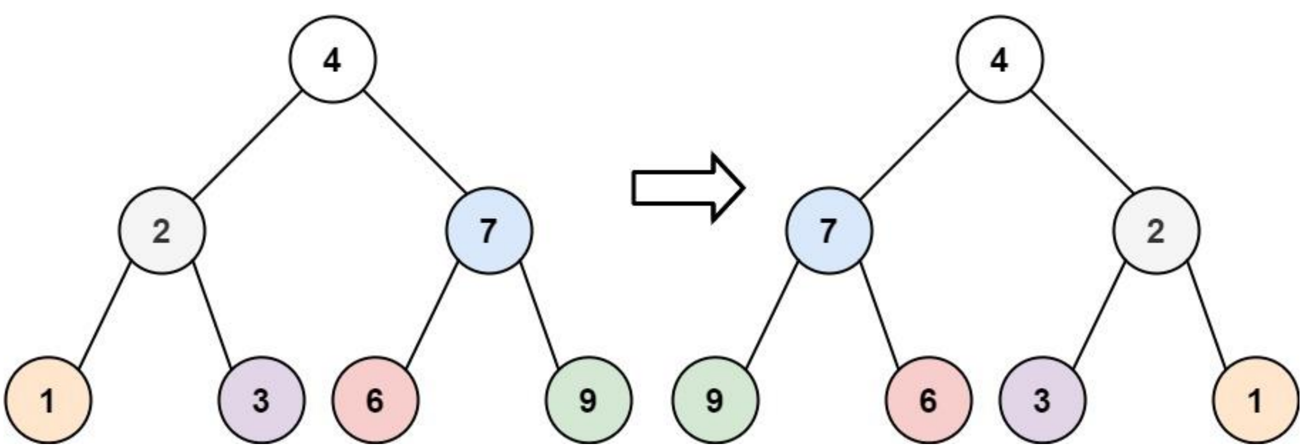
Input: p = [1,2,1], q = [1,1,2]
Output: false

```
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);

}
```

1.142 Invert Binary Tree

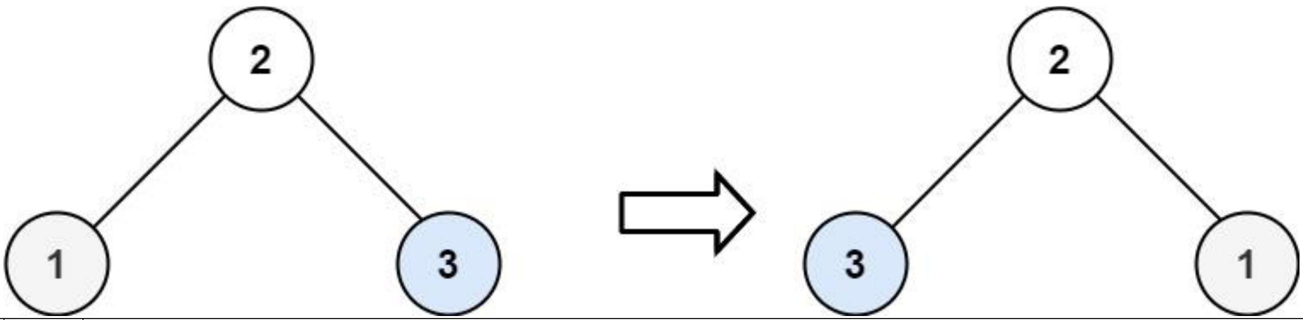
Given the root of a binary tree, invert the tree, and return *its root*.

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: []

```
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;
}
```

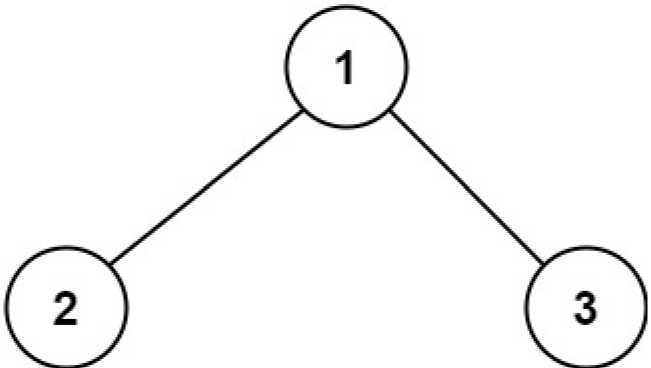
1.143 Binary Tree Maximum Path Sum

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.

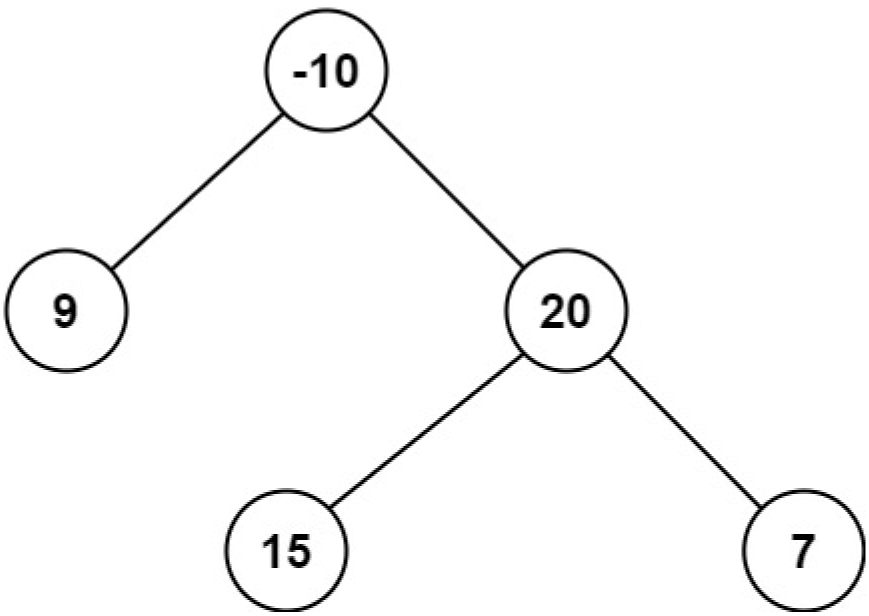
Given the root of a binary tree, return *the maximum* ***path sum*** *of any* ***non-empty*** *path*.

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:



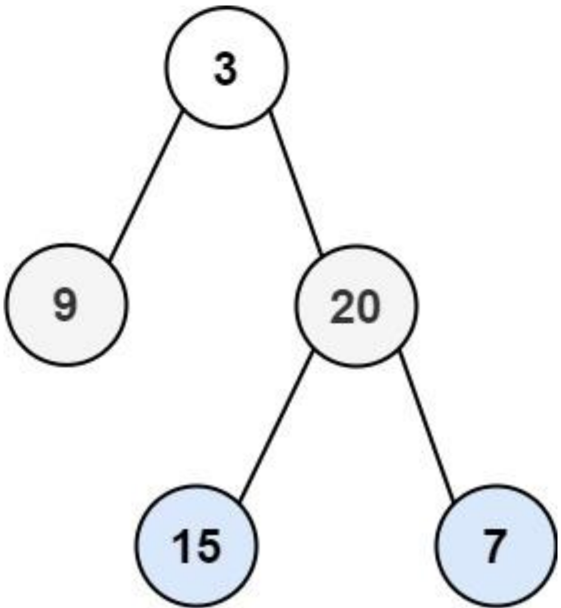

```
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode() {}
 *     TreeNode(int val) { this.val = val; }
 *     TreeNode(int val, TreeNode left, TreeNode right) {
 *         this.val = val;
 *         this.left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

1.144 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: []

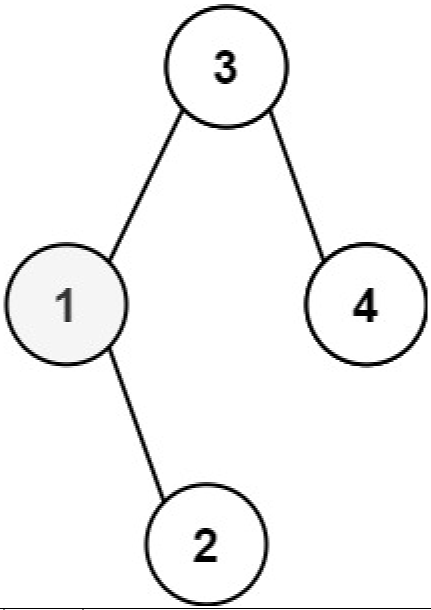
```
int** levelOrder(struct TreeNode* root, int* returnSize, int** returnColumnSizes) {
    // Check if the tree is empty
    if (root == NULL) {
        *returnSize = 0;
        *returnColumnSizes = NULL;
        return NULL;
    }

    // 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;
```

```
    *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;
}
```

1.145 Kth Smallest Element In BST

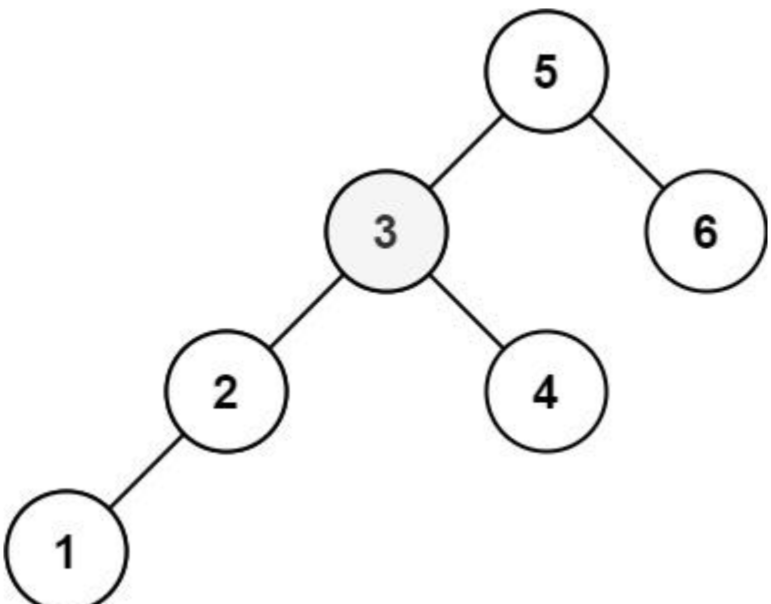
Given the root of a binary search tree, and an integer k, return *the* k[th] *smallest value (**1-indexed**) of all the values of the nodes in the tree*.

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

```
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;
    }
```

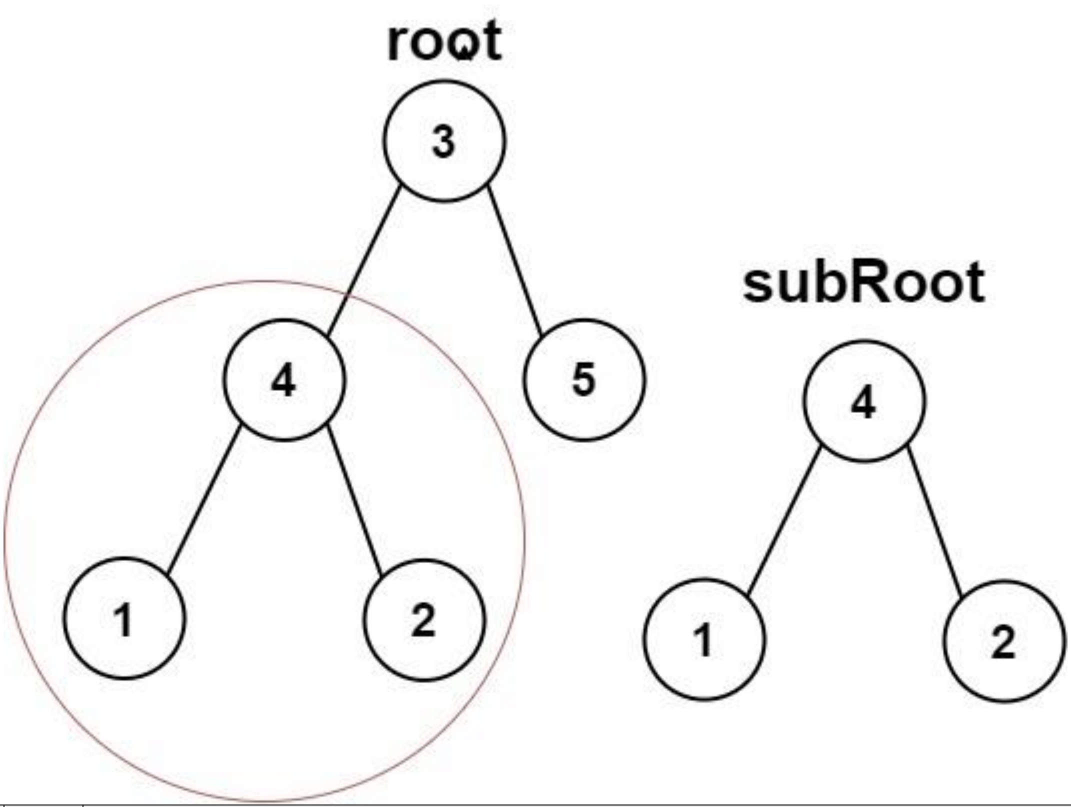
```
    public int kthSmallest(TreeNode root, int k) {
        ArrayList<Integer> nums = inorder(root, new ArrayList<Integer>());
        return nums.get(k - 1);
    }
}
```

1.146 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.

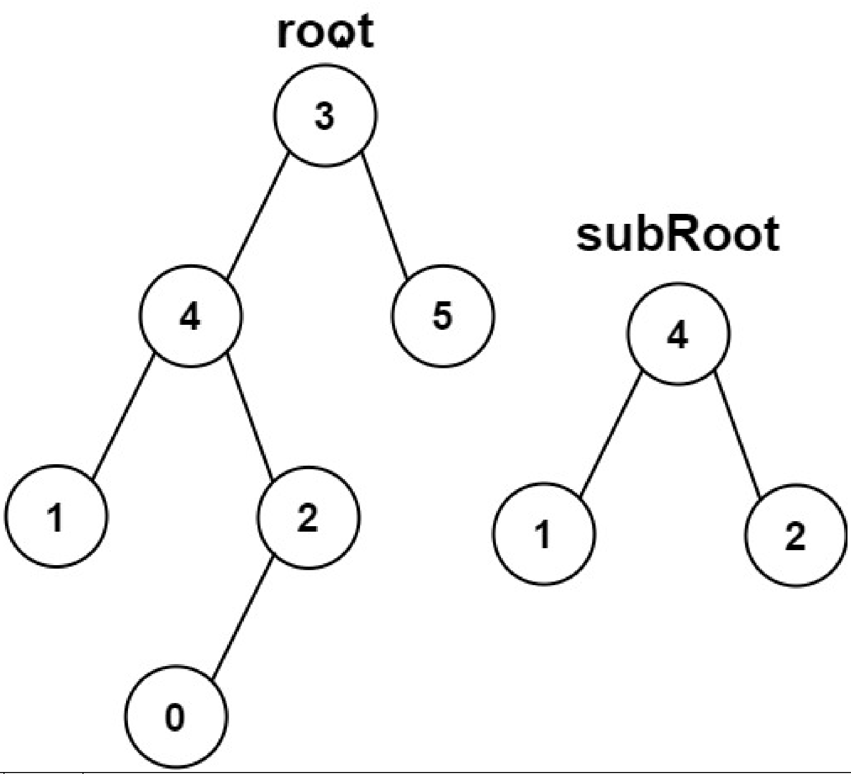
A subtree of a binary tree tree is a tree that consists of a node in tree and all of this node's descendants. The tree tree could also be considered as a subtree of itself.

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

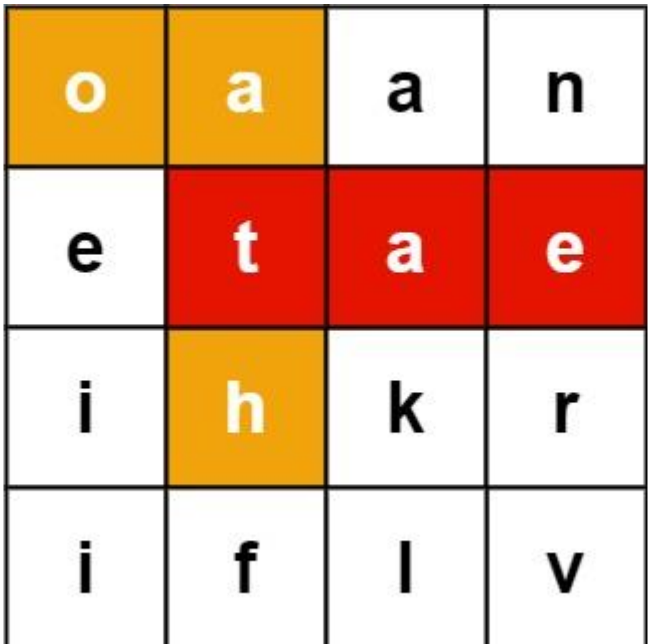
```java
public class Solution {
    public boolean isSubtree(TreeNode s, TreeNode t) {
        if (s == null) return false;
        if (isSame(s, t)) return true;
        return isSubtree(s.left, t) || isSubtree(s.right, t);
    }

    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);
    }
}
```

1.147 Word Search II

Given an m x n board of characters and a list of strings words, return *all words on the board*.

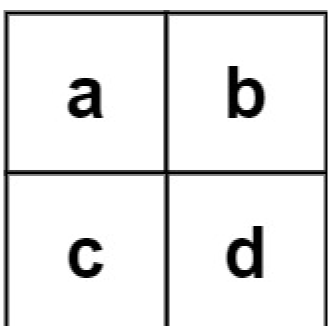
Each word must be constructed from letters of sequentially adjacent cells, where **adjacent cells** are horizontally or vertically neighboring. The same letter cell may not be used more than once in a word.

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;
```

```java
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);
    }
  }
}
```

1.148 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;
}
```

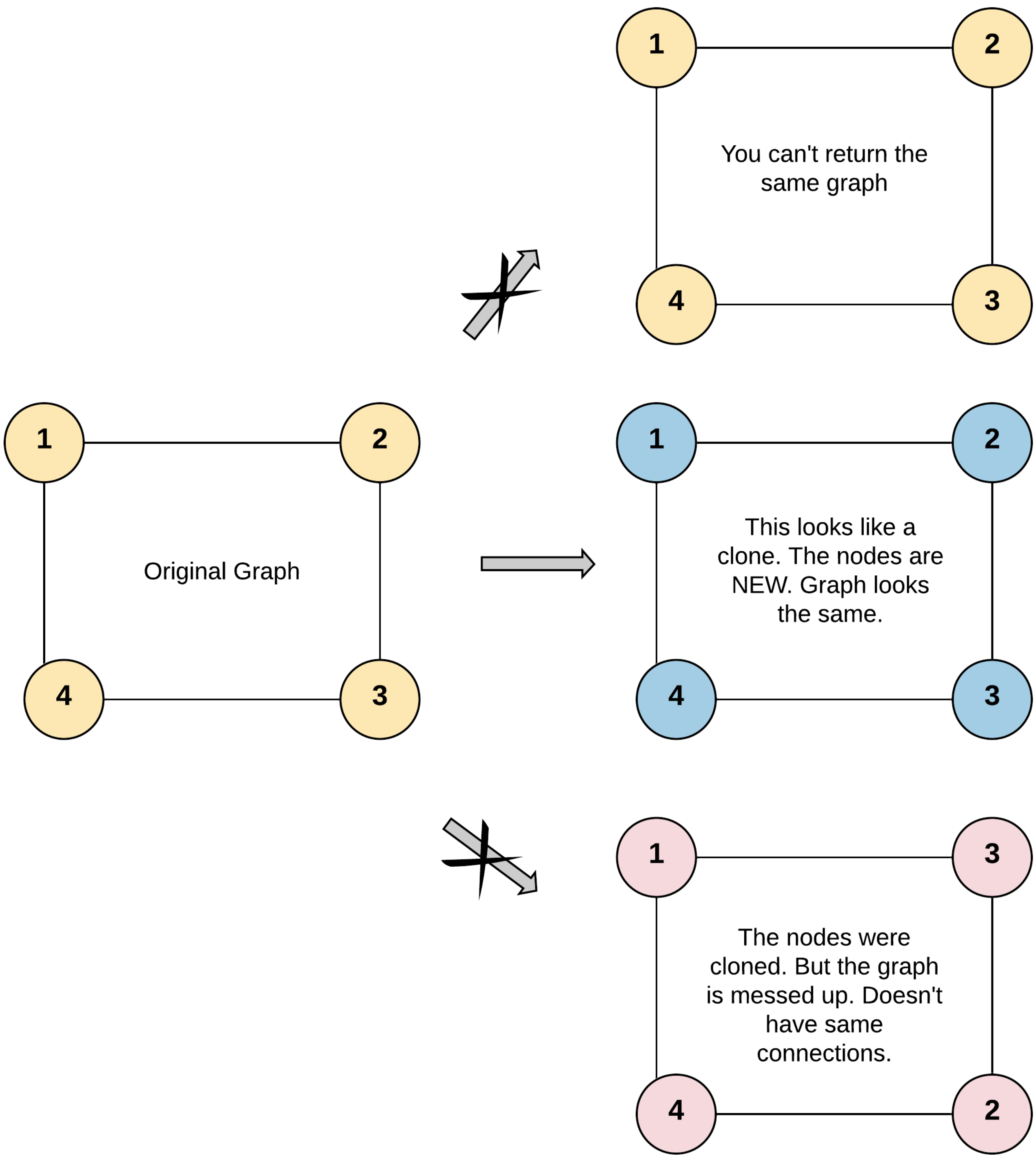
Test case format:

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.

Example 1:



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;
    }
```

}

1.149 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] = [ai, bi] indicates that you **must** take course bi first if you want to take course ai.

- 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.

```
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);
            }
        }
```

```java
        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;
}
```

1.150 Pacific Atlantic Water Flow

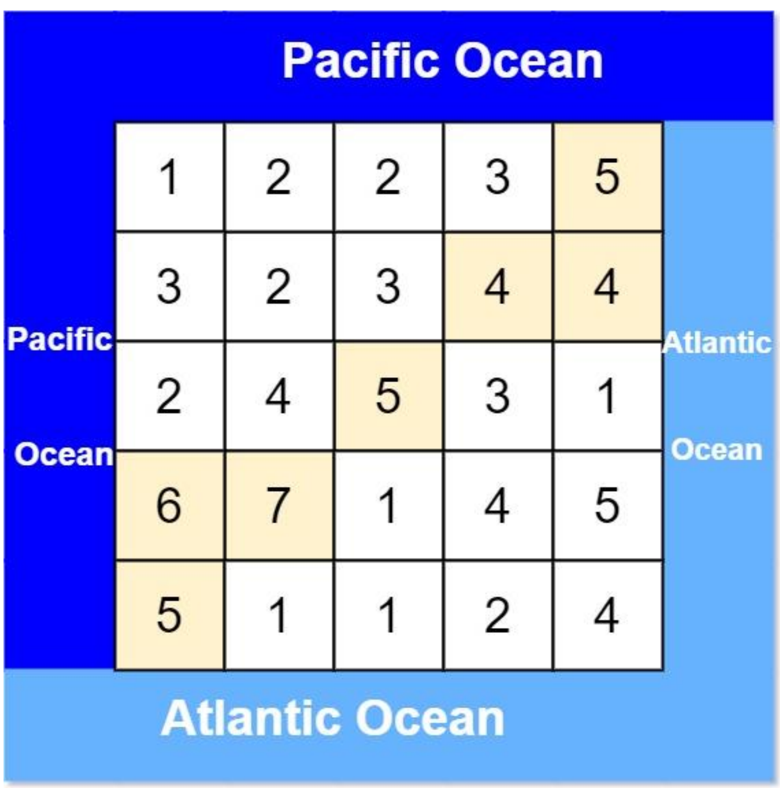
There is an m x n rectangular island that borders both the **Pacific Ocean** and **Atlantic Ocean**.
The **Pacific Ocean** touches the island's left and top edges, and the **Atlantic Ocean** touches the island's right and bottom edges.

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] = [ri, ci] *denotes that rain water can flow from cell* (ri, ci) *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.

```
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);
        }
    }
```

1.151 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 "".

```
class Solution {
    public String alienOrder(String[] words) {
        if(words.length==0)
            return "";
        Map<Character,Integer>inDegree=new HashMap<>();
        Map<Character,List<Character>>graph=new HashMap<>();
        //a. Initialise adjacency list and in degree map
        for (String word : words) {
            for (char character : word.toCharArray()) {
                inDegree.put(character, 0);
                graph.put(character, new ArrayList<Character>());
            }
        }
        for(int i=0;i<words.length-1;i++){
            String w1 = words[i], w2 = words[i + 1];
            //Checks if str1 starts wtih str2
            if (w1.length() > w2.length() && w1 .startsWith(w2)) {
                return "";
            }
            //b. Build the graph and in degree map
            for (int j = 0; j < Math.min(w1.length(), w2.length()); j++) {
                char parent = w1.charAt(j), child = w2.charAt(j);
                if (parent != child) { // if the two characters are different
                  graph.get(parent).add(child); // put the child into it's parent's list
                  inDegree.put(child, inDegree.get(child) + 1); // increment child's 
inDegree
                  break; // only the first different character between the two words 
will help us find the order
                }
            }
        }
```

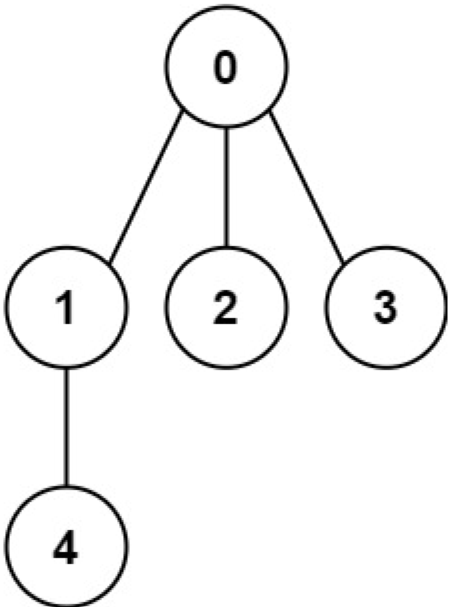
```java
            // c. Find all sources i.e., all vertices with 0 in-degrees
            Queue<Character> sources = new LinkedList<>();
            for (Map.Entry<Character, Integer> entry : inDegree.entrySet()) {
              if (entry.getValue() == 0)
                sources.add(entry.getKey());
            }
        // d. For each source, add it to the sortedOrder and subtract one from all of
its children's in-degrees
        // if a child's in-degree becomes zero, add it to the sources queue
        StringBuilder sortedOrder = new StringBuilder();
        while (!sources.isEmpty()) {
          Character vertex = sources.poll();
          sortedOrder.append(vertex);
          List<Character> children = graph.get(vertex); // get the node's children to
decrement their in-degrees
          for (Character child : children) {
            inDegree.put(child, inDegree.get(child) - 1);
            if (inDegree.get(child) == 0)
              sources.add(child);
          }
        }

        // if sortedOrder doesn't contain all characters, there is a cyclic dependency
between characters, therefore, we
        // will not be able to find the correct ordering of the characters
        if (sortedOrder.length() != inDegree.size())
          return "";

        return sortedOrder.toString();
    }
}
```

1.152 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.

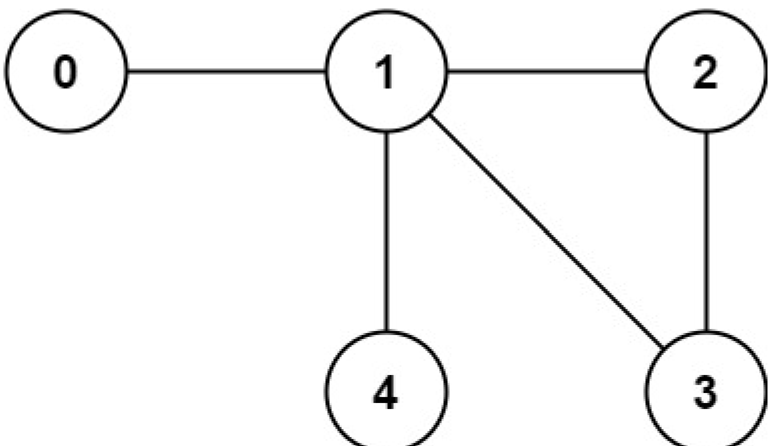
Return true *if the edges of the given graph make up a valid tree, and* false *otherwise*.

Example 1:



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

Example 2:



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<>());
        }
```

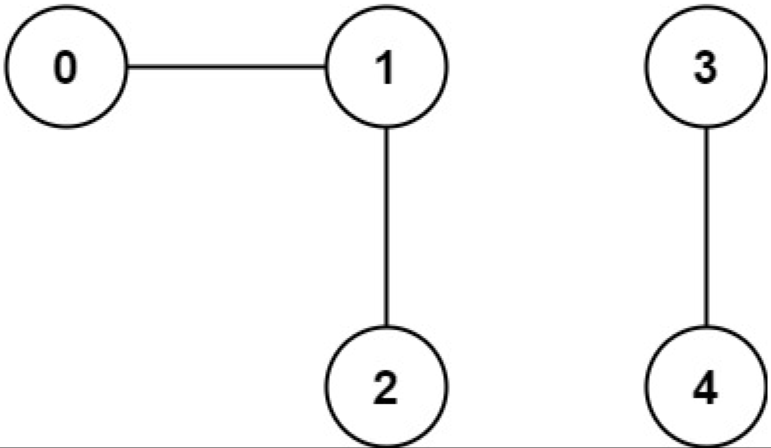
```
                //since this is an undirected graph, make each edge go both ways
        for (int[] edge : edges) {
            graph.get(edge[1]).add(edge[0]);
            graph.get(edge[0]).add(edge[1]);
        }

        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);
            }
        }
    }
}
```

1.153 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.

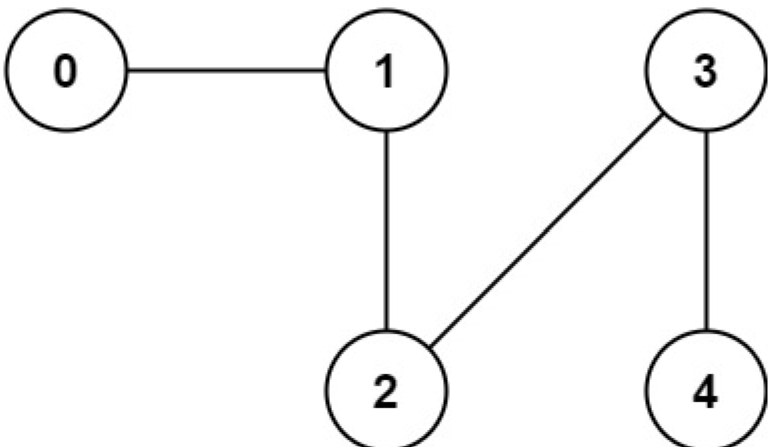
Return *the number of connected components in the graph*.

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

```
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;
}
}
```

1.154 Longest Prefix Match - Trie

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

// Define the structure for a trie node
typedef struct TrieNode {
    struct TrieNode* children[2];  // Assuming binary trie for IPv4 addresses
    int isLeaf;
} TrieNode;

// Function to create a new trie node
TrieNode* createNode() {
    TrieNode* node = (TrieNode*)malloc(sizeof(TrieNode));
    for (int i = 0; i < 2; i++) {
        node->children[i] = NULL;
    }
    node->isLeaf = 0;
    return node;
}

// Function to insert an IP address prefix into the trie
void insert(TrieNode* root, uint32_t prefix, int length) {
    TrieNode* current = root;
    for (int i = 31; i >= 32 - length; i--) {
        int bit = (prefix >> i) & 1;
        if (!current->children[bit]) {
            current->children[bit] = createNode();
        }
        current = current->children[bit];
    }
    current->isLeaf = 1;
}

// Function to perform longest prefix match and return the matched prefix
uint32_t longestPrefixMatch(TrieNode* root, uint32_t ip) {
    TrieNode* current = root;
    uint32_t match = 0;
    for (int i = 31; i >= 0; i--) {
        int bit = (ip >> i) & 1;
        if (current->children[bit]) {
            current = current->children[bit];
```

```
            match = (match << 1) | bit;  // Append the current bit to the matched prefix
        } else {
            break;
        }
    }
    return match;
}
int main() {
    // Example usage
    TrieNode* root = createNode();

    // Insert some prefixes into the trie
    insert(root, 0xC0A80100, 24);  // 192.168.1.0/24
    insert(root, 0xC0A80180, 26);  // 192.168.1.128/26
    insert(root, 0xAC100001, 16);  // 172.16.0.0/16

    // Test longest prefix match
    uint32_t ipAddress = 0xC0A80145;  // 192.168.1.69
    uint32_t matchedPrefix = longestPrefixMatch(root, ipAddress);

    if (matchedPrefix > 0) {
        printf("Longest prefix matched: %0x%x\n", matchedPrefix);
    } else {
        printf("No matching prefix found.\n");
    }

    return 0;
}
```

1.155 Longest Prefix Match - IP

```
#include <stdio.h>
#include <stdint.h>
#include <arpa/inet.h>

typedef struct {
    char network[20];
    int subnet;
} CIDR;

void findLongestPrefixMatch(CIDR cidrList[], int n, char* inputIP) {
    uint32_t input = ntohl(inet_addr(inputIP));  // Convert input IP to integer,
consider network byte order

    int maxPrefixLength = -1;
    char* matchingCIDR = NULL;

    for (int i = 0; i < n; ++i) {
        uint32_t network = ntohl(inet_addr(cidrList[i].network));  // Convert network IP
to integer
        uint32_t subnetMask = ~((1u << (32 - cidrList[i].subnet)) - 1);
```

```
        if ((input & subnetMask) == (network & subnetMask)) {
            if (cidrList[i].subnet > maxPrefixLength) {
                maxPrefixLength = cidrList[i].subnet;
                matchingCIDR = cidrList[i].network;
            }
        }
    }

    if (matchingCIDR != NULL) {
        printf("Longest Prefix Match: %s/%d\n", matchingCIDR, maxPrefixLength);
    } else {
        printf("No match found.\n");
    }
}

int main() {
    CIDR cidrList[] = {{"192.168.1.0", 24}, {"10.0.0.0", 16}, {"172.16.0.0", 20}};
    char inputIP[] = "192.168.1.15";

    findLongestPrefixMatch(cidrList, sizeof(cidrList) / sizeof(cidrList[0]), inputIP);

    return 0;
}
```

1.156 Socket Communication Client and Server

```
#include<stdio.h>
#include<arpa/inet.h>
#include<string.h>
#include<stdlib.h>
#include<pthread.h>
#define MAX 10
void *handle_client(void *arg) {
    int client_fd = *((int *)arg);
    char buffer[100];
    printf("\nClient connected!");
    send(client_fd, "Hi from Server!", strlen("Hi from Server!"), 0);
    recv(client_fd, buffer, sizeof(buffer),0);
    printf("\nClient sent : %s", buffer);
    close(client_fd);
    pthread_exit(NULL);
}
void main() {
    printf("\nSERVER INITIALIZING!");
    int server_fd = 0;
    int connected_fd = 0;
    struct sockaddr_in server_addr = {0};
    pthread_t tid[MAX];
    static int i = 0;
    int opt = 1;

    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("\nServer Socket creation failed!");
```

```
        return;
    }

    if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &opt,
sizeof(opt))) {
        perror("\nSocket option set failed!");
        return;
    }

    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(8888);
    struct sockaddr_in client_addr;
    if (bind(server_fd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
        perror("\nServer Bind failed!");
        return;
    }

    listen(server_fd, MAX);
    int addlen = sizeof(struct sockaddr_in);
    while(1) {
        if ((connected_fd = accept(server_fd, (struct sockaddr *)&client_addr, &addlen))
< 0) {
            perror("\nClient connection failed!");
            return;
        }

        pthread_create(&tid[i], NULL, handle_client, (void *)&connected_fd);
        i = (i+1)%MAX;
    }
    close(server_fd);
}
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<arpa/inet.h>
void main() {
    int socket_fd = 0;
    struct sockaddr_in server = {0};
    char buf[100];

    if ((socket_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
        perror("\nClient socket creation failed!");
        return;
    }

    server.sin_addr.s_addr = inet_addr("127.0.0.1");
    server.sin_port = htons(8888);
    server.sin_family = AF_INET;

    if (connect(socket_fd, (struct sockaddr*)&server, sizeof(server)) < 0) {
        perror("\nClient to Server connect failed!");
        return;
    }
    send(socket_fd, "Hi from Client", strlen("Hi from Client"),0);
    recv(socket_fd, &buf, sizeof(buf),0);
    printf("\nReceived from Server: %s",buf);
```

```
    close(socket_fd);
}
```

1.157 Shared Memory

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()
{
int i;
void *shared_memory;
char buff[100];
int shmid;
shmid=shmget((key_t)2345, 1024, 0666);
shared_memory=shmat(shmid,NULL,0); //process attached to shared memory segment
printf("Data read from shared memory is : %s\n",(char *)shared_memory);
}

#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()
{
int i;
void *shared_memory;
char buff[100];
int shmid;
shmid=shmget((key_t)2345, 1024, 0666|IPC_CREAT);
//creates shared memory segment with key 2345, having size 1024 bytes. IPC_CREAT is used
to create the shared segment if it does not exist. 0666 are the permissions on the
shared segment
shared_memory=shmat(shmid,NULL,0);
//process attached to shared memory segment
//this prints the address where the segment is attached with this process
printf("Enter some data to write to shared memory\n");
fgets(buff,100,stdin); //get some input from user
strcpy(shared_memory,buff); //data written to shared memory
}
```

1.158 Message Queues

```
#include<stdio.h>
#include<sys/ipc.h>
#include<sys/msg.h>
typedef struct msg_buf {
    long type;
    char msg[100];
}msg_t;
void main() {
    msg_t ms;
    key_t key = ftok("Allah",123);
    int msg_id = msgget(key, 0666|IPC_CREAT);
    msgrcv(msg_id,&ms, sizeof(ms),1,0);
```

```
    printf("%s",ms.msg);
}
#include<stdio.h>
#include<sys/ipc.h>
#include<sys/msg.h>
typedef struct msg_buffer {
    long type;
    char buffer[100];
}msg_t;
void main() {
    msg_t ms;
    key_t key = ftok("Allah",123);
    int msg_id = 0;
    msg_id = msgget(key,0666|IPC_CREAT);
    ms.type = 1;
    strncpy(ms.buffer,"HELLO DUDE!", sizeof("HELLO DUDE!"));
    msgsnd(msg_id, &ms, sizeof(ms), 0);
}
```

1.159 Dequeue

```
#include <stdio.h>
#include <stdlib.h>

// Node structure of a doubly-linked list
struct Node
{
    int data;
    struct Node* prev;
    struct Node* next;
};

// A structure for Deque
struct Deque
{
    struct Node* head;
    struct Node* tail;
    int size;
};

// Function prototypes
void initDeque(struct Deque* dq);
int isEmpty(struct Deque* dq);
int size(struct Deque* dq);
void insertFront(struct Deque* dq, int data);
void insertRear(struct Deque* dq, int data);
void deleteFront(struct Deque* dq);
void deleteRear(struct Deque* dq);
int getFront(struct Deque* dq);
int getRear(struct Deque* dq);

// Function to initialize the deque
void initDeque(struct Deque* dq)
{
    dq->head = dq->tail = NULL;
    dq->size = 0;
}
```

```c
// Function to check if the deque is empty
int isEmpty(struct Deque* dq)
{
    return (dq->head == NULL);
}

// Function to get the size of the deque
int size(struct Deque* dq)
{
    return dq->size;
}

// Function to insert an element at the front of the deque
void insertFront(struct Deque* dq, int data)
{
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL)
    {
        printf("Memory allocation failed (Overflow)\n");
        return;
    }

    newNode->data = data;
    newNode->prev = NULL;
    newNode->next = dq->head;

    if (dq->head == NULL)
    {
        dq->tail = newNode;
    }
    else
    {
        dq->head->prev = newNode;
    }

    dq->head = newNode;
    dq->size++;
}

// Function to insert an element at the rear of the deque
void insertRear(struct Deque* dq, int data)
{
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL)
    {
        printf("Memory allocation failed (Overflow)\n");
        return;
    }

    newNode->data = data;
    newNode->prev = dq->tail;
    newNode->next = NULL;

    if (dq->tail == NULL)
    {
        dq->head = newNode;
    }
```

```c
    else
    {
        dq->tail->next = newNode;
    }

    dq->tail = newNode;
    dq->size++;
}

// Function to delete an element from the front of the deque
void deleteFront(struct Deque* dq)
{
    if (isEmpty(dq))
    {
        printf("Underflow\n");
        return;
    }

    struct Node* temp = dq->head;
    dq->head = dq->head->next;

    if (dq->head == NULL)
    {
        dq->tail = NULL;
    }
    else
    {
        dq->head->prev = NULL;
    }

    free(temp);
    dq->size--;
}

// Function to delete an element from the rear of the deque
void deleteRear(struct Deque* dq)
{
    if (isEmpty(dq))
    {
        printf("Underflow\n");
        return;
    }

    struct Node* temp = dq->tail;
    dq->tail = dq->tail->prev;

    if (dq->tail == NULL)
    {
        dq->head = NULL;
    }
    else
    {
        dq->tail->next = NULL;
    }

    free(temp);
    dq->size--;
```

```
}

// Function to get the front element of the deque
int getFront(struct Deque* dq)
{
    if (isEmpty(dq))
    {
        return -1;
    }

    return dq->head->data;
}

// Function to get the rear element of the deque
int getRear(struct Deque* dq)
{
    if (isEmpty(dq))
    {
        return -1;
    }

    return dq->tail->data;
}

int main()
{
    struct Deque dq;
    initDeque(&dq);

    printf("Insert element '2' at rear end\n");
    insertRear(&dq, 2);
    printf("Insert element '0' at rear end\n");
    insertRear(&dq, 0);

    printf("Rear end element: %d\n", getRear(&dq));

    deleteRear(&dq);

    printf("After deleting rear element new rear is: %d\n", getRear(&dq));

    printf("Inserting element '27' at front end\n");
    insertFront(&dq, 27);

    printf("Front end element: %d\n", getFront(&dq));

    printf("Number of elements in Deque: %d\n", size(&dq));

    deleteFront(&dq);

    printf("After deleting front element new front is: %d\n", getFront(&dq));

    return 0;
}
```

1.160 Regex Search Target String

```
#define MAX_BUFFER_SIZE 1024

// Function to search for a target string using regex
int searchTargetString(const char *output, const char *target) {
    regex_t regex;
    int reti;
    char buffer[MAX_BUFFER_SIZE];

    // Compile the regular expression
    reti = regcomp(&regex, target, REG_EXTENDED);
    if (reti) {
        fprintf(stderr, "Could not compile regex\n");
        return -1;
    }

    // Execute the regular expression
    reti = regexec(&regex, output, 0, NULL, 0);
    if (!reti) {
        printf("Target string found in the output\n");
        return 1;
    } else if (reti == REG_NOMATCH) {
        printf("Target string not found in the output\n");
        return 0;
    } else {
        regerror(reti, &regex, buffer, sizeof(buffer));
        fprintf(stderr, "Regex match failed: %s\n", buffer);
        return -1;
    }

    // Free the compiled regular expression
    regfree(&regex);

    return 0;
}

int main() {
    FILE *fp;
    char buffer[MAX_BUFFER_SIZE];

    // Command to execute (replace with your network command)
    const char *command = "ping -c 4 google.com";

    // Open a pipe to execute the command and read its output
    fp = popen(command, "r");
    if (fp == NULL) {
        fprintf(stderr, "Failed to execute command\n");
        return 1;
    }

    // Read the command output
    while (fgets(buffer, sizeof(buffer), fp) != NULL) {
        // Search for the target string in the output
        if (searchTargetString(buffer, "time=") > 0) {
            // Do something when the target string is found
            printf("Found target string in the line: %s", buffer);
```

```c
        }
    }

    // Close the pipe
    pclose(fp);

    return 0;
}

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Function to execute a command and capture its output
char* executeCommand(const char* command) {
    FILE* fp;
    char buffer[128];
    char* result = malloc(4096); // Allocate a buffer for the command output

    result[0] = '\0'; // Initialize the result string

    // Open the command for reading
    fp = popen(command, "r");
    if (fp == NULL) {
        printf("Failed to run command\n");
        exit(1);
    }

    // Read the output line by line and concatenate it to the result string
    while (fgets(buffer, sizeof(buffer) - 1, fp) != NULL) {
        strcat(result, buffer);
    }

    // Close the pipe and return the result
    pclose(fp);
    return result;
}

// Function to extract the time value from the ping output
int extractPingTime(const char* pingOutput) {
    const char* timeString = strstr(pingOutput, "time=");
    if (timeString != NULL) {
        // Extract the time value as an integer
        int timeValue;
        sscanf(timeString, "time=%d", &timeValue);
        return timeValue;
    }
    return -1; // Return -1 if the time value is not found
}
int main() {
    const char* command = "ping -c 4 google.com"; // Your ping command
    char* pingOutput = executeCommand(command);

    int pingTime = extractPingTime(pingOutput);
    if (pingTime != -1) {
        printf("Ping time: %d ms\n", pingTime);
    } else {
```

```
        printf("Unable to extract ping time\n");
    }

    // Free the allocated memory
    free(pingOutput);

    return 0;
}
```

1.161 Quick Sort

```
void asc(int *arr, int start, int size) {
    int pivot = start;
    int i = start+1;
    int j = size;
    int temp = 0;
    if (start >= size) return;
    while (i <= j) {
        while ((j >= start) && arr[j] >= arr[pivot]) j--;
        while ((i <= size) && arr[i] <= arr[pivot]) i++;
        if (i < j) {
            temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        } else {
            temp = arr[pivot];
            arr[pivot] = arr[j];
            arr[j] = temp;
        }
    }
    asc(arr,0, j-1);
    asc(arr,j+1, size);
}
void quick_sort(int *arr, int size, void (*fn)(int*, int, int)) {
    fn(arr, 0, size-1);
    printf("\nSorted array is:");
    for (int i = 0; i < size; i++) {
        printf("%d ",arr[i]);
    }
}
```

1.162 Merge Sort

```
void merge_asc(int *arr, int start, int middle, int end) {
    int n1 = middle-start+1;
    int n2 = end-middle;
    int left[n1];
    int right[n2];
    int i = 0;
    int j = 0;
    int k = 0;

    for (i = 0; i < n1; i++) {
        left[i] = arr[start+i];
    }
    for (i = 0; i < n2; i++) {
        right[i] = arr[middle + i + 1];
```

```
    }

    i = 0;
    k = start;
    while (i < n1 && j < n2) {
        if (left[i] <= right[j]) {
            arr[k] = left[i];
            i++;
        } else {
            arr[k] = right[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        arr[k] = left[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = right[j];
        j++;
        k++;
    }
}
void asc(int *arr, int start, int end) {
    if (start < end) {
        int middle = (start + end)/2;
        asc(arr, start, middle);
        asc(arr, middle+1, end);
        merge_asc(arr,start,middle,end);
    }
}
```

1.163 Bloom Filter

```
#include <stdio.h>
#include <stdbool.h>
#include <math.h>

#define ll long long

// hash 1
int h1(int num, int arrSize)
{
    ll int hash = num % arrSize;
    return hash;
}

// hash 2
int h2(int num, int arrSize)
{
    ll int hash = 1;
    while (num > 0)
    {
        hash = (hash + num % 10);
```

```
        num /= 10;
    }
    hash = hash % arrSize;
    return hash;
}

// hash 3
int h3(int num, int arrSize)
{
    ll int hash = 7;
    while (num > 0)
    {
        hash = (hash * 31 + num % 10) % arrSize;
        num /= 10;
    }
    return hash;
}

// hash 4
int h4(int num, int arrSize)
{
    ll int hash = 3;
    int p = 7;
    while (num > 0)
    {
        hash += hash * 7 + (num % 10) * pow(p, num % 10);
        hash = hash % arrSize;
        num /= 10;
    }    return hash;
}

// lookup operation
bool lookup(int *bitarray, int arrSize, int num)
{
    int a = h1(num, arrSize);
    int b = h2(num, arrSize);
    int c = h3(num, arrSize);
    int d = h4(num, arrSize);

    if (bitarray[a] && bitarray[b] && bitarray[c] && bitarray[d])
        return true;
    else
        return false;
}

// insert operation
void insert(int *bitarray, int arrSize, int num)
{
    // check if the element is already present or not
    if (lookup(bitarray, arrSize, num))
        printf("%d is Probably already present\n", num);
    else
    {
        int a = h1(num, arrSize);
        int b = h2(num, arrSize);
        int c = h3(num, arrSize);
        int d = h4(num, arrSize);
```

```
        bitarray[a] = 1;
        bitarray[b] = 1;
        bitarray[c] = 1;
        bitarray[d] = 1;

        printf("%d inserted\n", num);
    }
}
// Driver Code
int main()
{
    int bitarray[100] = {0};
    int arrSize = 100;
    int numArray[33] = {123, 456, 789, 321, 654, 987, 111, 222, 333,
                        444, 555, 666, 777, 888, 999, 101, 202, 303,
                        404, 505, 606, 707, 808, 909, 1111, 2222, 3333,
                        4444, 5555, 6666, 7777, 8888, 9999};

    for (int i = 0; i < 33; i++)
    {
        insert(bitarray, arrSize, numArray[i]);
    }

    return 0;
}
```

1.164 Hash Table

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Define the size of the hash table
#define TABLE_SIZE 10

// Define a structure for a node in the linked list
struct Node {
    int key;
    char data[50];
    struct Node* next;
};

// Define the hash table as an array of linked lists
struct Node* hashTable[TABLE_SIZE] = {NULL};

// Hash function
int hash(int key) {
    return key % TABLE_SIZE;
}

// Insert a key-value pair into the hash table
void insert(int key, const char* data) {
    // Create a new node
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        fprintf(stderr, "Memory allocation failed.\n");
```

```
        exit(1);
    }

    // Set the key and data for the new node
    newNode->key = key;
    strncpy(newNode->data, data, sizeof(newNode->data));
    newNode->next = NULL;

    // Get the hash index
    int index = hash(key);

    // Insert the new node at the beginning of the linked list
    newNode->next = hashTable[index];
    hashTable[index] = newNode;
}
// Search for a key in the hash table
const char* search(int key) {
    // Get the hash index
    int index = hash(key);

    // Traverse the linked list at the hash index
    struct Node* current = hashTable[index];
    while (current != NULL) {
        if (current->key == key) {
            return current->data; // Key found
        }
        current = current->next;
    }

    return NULL; // Key not found
}

// Print the contents of the hash table
void printHashTable() {
    for (int i = 0; i < TABLE_SIZE; i++) {
        printf("Bucket %d: ", i);
        struct Node* current = hashTable[i];
        while (current != NULL) {
            printf("(%d, %s) ", current->key, current->data);
            current = current->next;
        }
        printf("\n");
    }
}

// Main function for testing the hash table
int main() {
    // Insert key-value pairs
    insert(1, "John");
    insert(2, "Alice");
    insert(11, "Bob");
    insert(12, "Charlie");
    insert(21, "David");
    // Print the initial hash table
    printf("Initial Hash Table:\n");
    printHashTable();
```

```
    // Search for keys
    printf("\nSearch results:\n");
    printf("Key 2: %s\n", search(2));
    printf("Key 12: %s\n", search(12));
    printf("Key 5: %s\n", search(5));

    return 0;
}
```

1.165 Endianess

```
#include<stdio.h>
//12345678 -> 0th byte 12 --> little endian, 0th byte 78 --> big endian
void main() {
    int data = 12345678;
    char *d = (char*)&data;
    printf("\n %u",*d);
    if (*d == 78)
        printf("\nLittle Endian");
    else
        printf("\nBig Endian");
}
```

1.166 Store IP address

```
#include<stdio.h>
typedef struct ip_address {
    unsigned char ip[4];
} ip_addr_t;
void store_ip(char *input_ip, ip_addr_t *ip) {
    sscanf(input_ip, "%hhu.%hhu.%hhu.%hhu", &ip->ip[0], &ip->ip[1], &ip->ip[2], &ip-
>ip[3]);
    printf("\nStored ip is : %hhu.%hhu.%hhu.%hhu",ip->ip[0], ip->ip[1], ip->ip[2], ip-
>ip[3]);
}
void retrieve_ip(char *output_ip, ip_addr_t *ip) {
    sprintf(output_ip, "%hhu.%hhu.%hhu.%hhu",ip->ip[0], ip->ip[1], ip->ip[2], ip-
>ip[3]);
}
void main() {
    char input_ip[15];
    char output_ip[15];
    ip_addr_t stored_ip;
    printf("\nEnter an ip address : ");
    scanf("%s", input_ip);
    printf("\nInput ip address is : %s",input_ip);
    store_ip(&input_ip, &stored_ip);
    retrieve_ip(&output_ip, &stored_ip);
    printf("\nRetrieved ip address is : %s",output_ip);
}
```

1.167 Store MAC address

```
#include<stdio.h>
typedef struct mac_address {
    unsigned char mac[6];
} mac_addr_t;
void store_mac(char *input_mac, mac_addr_t *mac) {
    printf("\naddress of mac %p",&mac);
```

```
    printf("\naddress of mac->mac[0] %p",&mac->mac[0]);
    sscanf(input_mac, "%hhx:%hhx:%hhx:%hhx:%hhx:%hhx", &mac->mac[0], &mac->mac[1], &mac->mac[2], &mac->mac[3], &mac->mac[4], &mac->mac[5]);
    printf("\nStored mac is : %02x:%02x:%02x:%02x:%02x:%02x",mac->mac[0], mac->mac[1], mac->mac[2], mac->mac[3], mac->mac[4], mac->mac[5]);
}
void retrieve_mac(char *output_mac, mac_addr_t *mac) {
    sprintf(output_mac, "%hhx:%hhx:%hhx:%hhx:%hhx:%hhx",mac->mac[0], mac->mac[1], mac->mac[2], mac->mac[3], mac->mac[4], mac->mac[5]);
}
void main() {
    char input_mac[18];
    char output_mac[18];
    mac_addr_t stored_mac;
    printf("\nEnter a mac address : ");
    scanf("%s", input_mac);
    printf("\nInput mac address is : %s",input_mac);
    store_mac(&input_mac, &stored_mac);
    printf("\naddress of stored mac %p",&stored_mac);
    printf("\naddress of stored mac->mac[0] %p",&stored_mac.mac[0]);
    retrieve_mac(&output_mac, &stored_mac);
    printf("\nRetrieved mac address is : %s",output_mac);
}
```

1.168 My Memory Allocation

```
#include <stdio.h>
#include <stdlib.h>

#define BUFFER_SIZE 1000  // Adjust the size as needed

// Global buffer to simulate memory allocation
char buffer[BUFFER_SIZE];
int next_free_offset = 0;

void* my_malloc(size_t size) {
    if (next_free_offset + size <= BUFFER_SIZE) {
        void* allocated_memory = buffer + next_free_offset;
        next_free_offset += size;
        return allocated_memory;
    } else {
        fprintf(stderr, "Error: Insufficient memory in the fixed buffer.\n");
        return NULL;
    }
}

void my_free(void* ptr) {
    // In a simple fixed buffer scenario, freeing memory can be a no-op
    // or could be used to reset the allocation pointer.
    // For demonstration purposes, we'll reset the allocation pointer here.
    next_free_offset = (int)((char*)ptr - buffer);
}

int main() {
    // Example usage
    int* int_ptr = (int*)my_malloc(sizeof(int));
```

```
    if (int_ptr != NULL) {
        *int_ptr = 42;
        printf("Allocated integer value: %d\n", *int_ptr);

        // Freeing the allocated memory
        my_free(int_ptr);
    }

    return 0;
}

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <pthread.h>

#define MEMORY_POOL_SIZE 1000

// Block structure to represent allocated memory blocks
typedef struct Block {
    size_t size;
    struct Block* next;
} Block;

// Memory pool
static char memory_pool[MEMORY_POOL_SIZE];

// Mutex for thread safety
static pthread_mutex_t memory_mutex = PTHREAD_MUTEX_INITIALIZER;

// Pointer to the first block in the memory pool
static Block* memory_head = NULL;

// Function to initialize the memory manager
void initialize_memory_manager() {
    memory_head = (Block*)memory_pool;
    memory_head->size = MEMORY_POOL_SIZE - sizeof(Block);
    memory_head->next = NULL;
}

// Function to allocate memory
void* my_malloc(size_t size) {
    pthread_mutex_lock(&memory_mutex);

    if (memory_head == NULL) {
        initialize_memory_manager();
    }

    size_t total_size = size + sizeof(Block);
    Block* current_block = memory_head;
    Block* previous_block = NULL;

    while (current_block != NULL) {
        if (current_block->size >= total_size) {
            // Allocate from this block
            if (current_block->size > total_size + sizeof(Block)) {
                // Split the block
```

```c
                Block* new_block = (Block*)((char*)current_block + total_size);
                new_block->size = current_block->size - total_size;
                new_block->next = current_block->next;
                current_block->size = size;
                current_block->next = new_block;
            } else {
                // Use the entire block
                total_size = current_block->size;
            }

            // Update the linked list
            if (previous_block == NULL) {
                memory_head = current_block->next;
            } else {
                previous_block->next = current_block->next;
            }

            pthread_mutex_unlock(&memory_mutex);
            return (char*)current_block + sizeof(Block);
        }

        previous_block = current_block;
        current_block = current_block->next;
    }

    pthread_mutex_unlock(&memory_mutex);
    fprintf(stderr, "Error: Insufficient memory.\n");
    return NULL;
}

// Function to free memory
void my_free(void* ptr) {
    pthread_mutex_lock(&memory_mutex);

    if (ptr != NULL) {
        Block* block = (Block*)((char*)ptr - sizeof(Block));
        block->next = memory_head;
        memory_head = block;
    }

    pthread_mutex_unlock(&memory_mutex);
}

// Function to print memory blocks for debugging
void print_memory_blocks() {
    Block* current_block = memory_head;
    while (current_block != NULL) {
        printf("Block: %p, Size: %zu\n", (void*)current_block, current_block->size);
        current_block = current_block->next;
    }
}
int main() {
    // Example usage
    initialize_memory_manager();

    int* int_ptr = (int*)my_malloc(sizeof(int));
    if (int_ptr != NULL) {
```

```
        *int_ptr = 42;
        printf("Allocated integer value: %d\n", *int_ptr);

        print_memory_blocks();

        // Freeing the allocated memory
        my_free(int_ptr);

        print_memory_blocks();
    }

    return 0;
}
```

1.169 Single Linked List

```
#include<stdio.h>
#include<stdlib.h>
struct node {
    int data;
    struct node *next;
};
struct node *head;
struct node* allocate() {
    struct node *new_node = (struct node *)malloc(sizeof(struct node));
    if (new_node == NULL) {
        printf("\nMalloc failure!");
        exit(1);
    }
    return new_node;
}
void main() {
    printf("\nSingle Linked List!");
    int choice = 0;
    int position = 0;
    int count = 0;
    int value = 0;
    struct node *new_node = NULL;
    while (1) {
        printf("\n1.Create\n2.Insert at beginning\n3.Insert at end\n4.Insert at
position\n5.Delete from first\n6.Delete at end\n7.Delete at
position\n8.Search\n9.Print\n10.End");
        printf("\nSelect a choice:");
        scanf("%d",&choice);
        switch (choice) {
            case 1:
            printf("\nCreate!");
            new_node = allocate();
            if (new_node == NULL) {
                printf("\nMalloc failure!");
                exit(1);
            } else {
                printf("\nEnter the value for node!");
                scanf("%d",&new_node->data);
                new_node->next = NULL;
                head = new_node;
            }
```

```
break;
case 2:
printf("\nInsert at beginning!");
new_node = allocate();
if (new_node == NULL) {
    printf("\nMalloc failure!");
    exit(1);
} else {
    printf("\nEnter the value for node!");
    scanf("%d",&new_node->data);
    new_node->next = head;
    head = new_node;
}
break;
case 3:
printf("\nInsert at end!");
new_node = allocate();
if (new_node == NULL) {
    printf("\nMalloc failure!");
    exit(1);
} else {
    printf("\nEnter the value for node!");
    scanf("%d",&new_node->data);
    struct node* cur = head;
    while (cur->next != NULL) {
        cur = cur->next;
    }
    cur->next = new_node;
    new_node->next = NULL;
}
break;
case 4:
printf("\nInsert at position!");
printf("\nTell me a position!");
scanf("%d",&position);
new_node = allocate();
if (new_node == NULL) {
    printf("\nMalloc failure!");
    exit(1);
} else {
    printf("\nEnter the value for node!");
    scanf("%d",&new_node->data);
    struct node* curr = head;
    while (curr != NULL) {
        count++;
        if (position-1 > count) {
            curr = curr->next;
        } else {
            break;
        }
    }
    new_node->next = curr->next;
    curr->next = new_node;
}
break;
case 5:
printf("\nDelete from first!");
```

```
if (head == NULL) {
    printf("\nList empty!");
}
struct node *ele = head;
struct node *next = head->next;
free(ele);
ele = NULL;
head = next;
break;
case 6:
printf("\nDelete at end!");
struct node* h = head;
struct node* p = head;
if (h == NULL) {
    printf("\nList is empty!");
}
if (h->next == NULL) {
    free(h);
    h = NULL;
    head = NULL;
}
else {
    while (h->next != NULL) {
        p = h;
        h = h->next;
    }
    p->next = NULL;
    free(h);
}
break;
case 7:
count = 0;
printf("\nDelete at position!");
printf("\nEnter position to delete from");
scanf("%d",&position);
struct node* cr = head;
struct node* pr = head;
while (cr != NULL) {
    count++;
    pr = cr;
    if (count == position-1) {
        pr->next = pr->next->next;
        free(cr->next);
        cr->next = NULL;
        break;
    }
    cr = cr->next;
}
break;
case 8:
printf("\nSearch!");
printf("\nEnter the value to search!");
scanf("%d",&value);
count = 0;
struct node* c = head;
while (c != NULL) {
    count++;
```

```
                if (c->data == value) {
                    printf("\nValue %d present in %d position",value,count);
                    break;
                }
                if (c->next == NULL) {
                    printf("\nValue %d not found!",value);
                }
                c = c->next;
            }
            break;
            case 9:
            printf("\nList!");
            struct node* current = head;
            printf("\n");
            while (current != NULL) {
                printf("%d->",current->data);
                current = current->next;
            }
            printf("NULL");
            break;
            case 10:
            printf("\nThank you!");
            exit(1);
            break;
            default:
            printf("\nTry again!");
            break;
        }
        choice = 0;
    }
}
```

1.170 Double Linked List

```
#include<stdio.h>
#include<stdlib.h>
struct node {
    int data;
    struct node *prev;
    struct node *next;
};
struct node *head;
struct node* allocate() {
    struct node *new_node = (struct node *)malloc(sizeof(struct node));
    if (new_node == NULL) {
        printf("\nMalloc failure!");
        exit(1);
    }
    return new_node;
}
void main() {
    printf("\nSingle Linked List!");
    int choice = 0;
    int position = 0;
    int count = 0;
    int value = 0;
    struct node *new_node = NULL;
```

```
    while (1) {
        printf("\n1.Create\n2.Insert at beginning\n3.Insert at end\n4.Insert at
position\n5.Delete from first\n6.Delete at end\n7.Delete at
position\n8.Search\n9.Print\n10.Reverse Print\n11.End");
        printf("\nSelect a choice:");
        scanf("%d",&choice);
        switch (choice) {
            case 1:
            printf("\nCreate!");
            new_node = allocate();
            if (new_node == NULL) {
                printf("\nMalloc failure!");
                exit(1);
            } else {
                printf("\nEnter the value for node!");
                scanf("%d",&new_node->data);
                new_node->prev = NULL;
                new_node->next = NULL;
                head = new_node;
            }
            break;
            case 2:
            printf("\nInsert at beginning!");
            new_node = allocate();
            if (new_node == NULL) {
                printf("\nMalloc failure!");
                exit(1);
            } else {
                printf("\nEnter the value for node!");
                scanf("%d",&new_node->data);
                new_node->prev = NULL;
                new_node->next = head;
                head->prev = new_node;
                head = new_node;
            }
            break;
            case 3:
            printf("\nInsert at end!");
            new_node = allocate();
            if (new_node == NULL) {
                printf("\nMalloc failure!");
                exit(1);
            } else {
                printf("\nEnter the value for node!");
                scanf("%d",&new_node->data);
                struct node* cur = head;
                while (cur->next != NULL) {
                    cur = cur->next;
                }
                cur->next = new_node;
                new_node->prev = cur;
                new_node->next = NULL;
            }
            break;
            case 4:
            printf("\nInsert at position!");
            printf("\nTell me a position!");
```

```
scanf("%d",&position);
new_node = allocate();
if (new_node == NULL) {
    printf("\nMalloc failure!");
    exit(1);
} else {
    printf("\nEnter the value for node!");
    scanf("%d",&new_node->data);
    struct node* curr = head;
    while (curr != NULL) {
        count++;
        if (position-1 > count) {
            curr = curr->next;
        } else {
            break;
        }
    }
    new_node->next = curr->next;
    curr->next->prev = new_node;
    new_node->prev = curr;
    curr->next = new_node;
}
break;
case 5:
printf("\nDelete from first!");
if (head == NULL) {
    printf("\nList empty!");
}
struct node *ele = head;
struct node *next = head->next;
free(ele);
ele = NULL;
next->prev = NULL;
head = next;
break;
case 6:
printf("\nDelete at end!");
struct node* h = head;
struct node* p = head;
if (h == NULL) {
    printf("\nList is empty!");
}
if (h->next == NULL) {
    free(h);
    h = NULL;
    head = NULL;
}
else {
    while (h->next != NULL) {
        p = h;
        h = h->next;
    }
    p->next = NULL;
    free(h);
}
break;
case 7:
```

```
count = 0;
printf("\nDelete at position!");
printf("\nEnter position to delete from");
scanf("%d",&position);
struct node* cr = head;
struct node* nx = NULL;
while (cr != NULL) {
    count++;
    //1 2 3 4 5
    if (count == position-1) {
        nx = cr->next;
        cr->next->next->prev = cr;
        cr->next = cr->next->next;
        free(nx);
        nx->next = NULL;
        break;
    }
    cr = cr->next;
}
break;
case 8:
printf("\nSearch!");
printf("\nEnter the value to search!");
scanf("%d",&value);
count = 0;
struct node* c = head;
while (c != NULL) {
    count++;
    if (c->data == value) {
        printf("\nValue %d present in %d position",value,count);
        break;
    }
    if (c->next == NULL) {
        printf("\nValue %d not found!",value);
    }
    c = c->next;
}
break;
case 9:
printf("\nList!");
struct node* current = head;
printf("\n");
while (current != NULL) {
    printf("%d->",current->data);
    current = current->next;
}
printf("NULL");
break;
case 10:
printf("\nList reverse print!");
struct node* currnt = head;
printf("\n");
while (currnt->next != NULL) {
    currnt = currnt->next;
}
struct node* end = currnt;
while (end != NULL) {
```

```
                printf("%d->",end->data);
                end = end->prev;
            }
            printf("NULL");
            case 11:
            printf("\nThank you!");
            exit(1);
            break;
            default:
            printf("\nTry again!");
            break;
        }
        choice = 0;
    }
}
```

1.171 Stack

```
#include<stdio.h>
#include<stdlib.h>
struct node {
    int data;
    struct node *next;
};
struct node *head;
struct node* allocate() {
    struct node *new_node = (struct node *)malloc(sizeof(struct node));
    if (new_node == NULL) {
        printf("\nMalloc failure!");
        exit(1);
    }
    return new_node;
}
void main() {
    printf("\nStack!");
    int choice = 0;
    int position = 0;
    int count = 0;
    int value = 0;
    struct node *new_node = NULL;
    while (1) {
        printf("\n1.Create\n2.Push\n3.Pop\n4.Print\n5.End");
        printf("\nSelect a choice:");
        scanf("%d",&choice);
        switch (choice) {
            case 1:
            printf("\nCreate!");
            new_node = allocate();
            if (new_node == NULL) {
                printf("\nMalloc failure!");
                exit(1);
            } else {
                printf("\nEnter the value for node!");
                scanf("%d",&new_node->data);
                new_node->next = NULL;
                head = new_node;
            }
```

```
            break;
            case 2:
            printf("\nPush!");
            new_node = allocate();
            if (new_node == NULL) {
                printf("\nMalloc failure!");
                exit(1);
            } else {
                printf("\nEnter the value for node!");
               scanf("%d",&new_node->data);
                new_node->next = head;
                head = new_node;
            }
            break;
            case 3:
            printf("\nPop!");
            if (head == NULL) {
                printf("\nList empty!");
            }
            struct node *ele = head;
            struct node *next = head->next;
            free(ele);
            ele = NULL;
            head = next;
            break;
            case 4:
            printf("\nStack Contents!");
            struct node* current = head;
            printf("\n");
            while (current != NULL) {
                printf("%d->",current->data);
                current = current->next;
            }
            printf("NULL");
            break;
            case 5:
            printf("\nThank you!");
            exit(1);
            break;
            default:
            printf("\nTry again!");
            break;
        }
        choice = 0;
    }
}
```

1.172 Queue

```
#include<stdio.h>
#include<stdlib.h>
struct node {
    int data;
    struct node *next;
};
struct node *head;
struct node* allocate() {
```

```c
    struct node *new_node = (struct node *)malloc(sizeof(struct node));
    if (new_node == NULL) {
        printf("\nMalloc failure!");
        exit(1);
    }
    return new_node;
}
void main() {
    printf("\nQueue!");
    int choice = 0;
    int position = 0;
    int count = 0;
    int value = 0;
    struct node *new_node = NULL;
    while (1) {
        printf("\n1.Create\n2.Enqueue\n3.Dequeue\n4.Print\n5.End");
        printf("\nSelect a choice:");
        scanf("%d",&choice);
        switch (choice) {
            case 1:
            printf("\nCreate!");
            new_node = allocate();
            if (new_node == NULL) {
                printf("\nMalloc failure!");
                exit(1);
            } else {
                printf("\nEnter the value for node!");
                scanf("%d",&new_node->data);
                new_node->next = NULL;
                head = new_node;
            }
            break;
            case 2:
            printf("\nEnqueue!");
            new_node = allocate();
            if (new_node == NULL) {
                printf("\nMalloc failure!");
                exit(1);
            } else {
                printf("\nEnter the value for node!");
                scanf("%d",&new_node->data);
                struct node* cur = head;
                while (cur->next != NULL) {
                    cur = cur->next;
                }
                cur->next = new_node;
                new_node->next = NULL;
            }
            break;
            case 3:
            printf("\nDequeue!");
            if (head == NULL) {
                printf("\nList empty!");
            }
            struct node *ele = head;
            struct node *next = head->next;
            free(ele);
```

```
            ele = NULL;
            head = next;
            break;
            case 4:
            printf("\nList!");
            struct node* current = head;
            printf("\n");
            while (current != NULL) {
                printf("%d->",current->data);
                current = current->next;
            }
            printf("NULL");
            break;
            case 5:
            printf("\nThank you!");
            exit(1);
            break;
            default:
            printf("\nTry again!");
            break;
        }
        choice = 0;
    }
}
```

1.173 Binary Search Tree

```
#include<stdio.h>
#include<stdlib.h>
struct node {
    int data;
    struct node *left;
    struct node *right;
};
struct node *root;
struct node* allocate() {
    struct node *new_node = (struct node *)malloc(sizeof(struct node));
    if (new_node == NULL) {
        printf("\nMalloc failure!");
        exit(1);
    }
    return new_node;
}
struct node* insert(struct node *temp, struct node *root) {
    if (root == NULL) {
        root = temp;
    } else {
        if (temp->data < root->data) {
            root->left = insert(temp,root->left);
        } else if (temp->data > root->data){
            root->right = insert(temp,root->right);
        }
    }
    return root;
}
void print_inorder(struct node *t) {
    if (t == NULL) return;
    if (t != NULL) {
```

```
        print_inorder(t->left);
        printf("%d:",t->data);
        print_inorder(t->right);
    }
}
void print_preorder(struct node *t) {
    if (t == NULL) return;
    if (t != NULL) {
        printf("%d:",t->data);
        print_preorder(t->left);
        print_preorder(t->right);
    }
}
void print_postorder(struct node *t) {
    if (t == NULL) return;
    if (t != NULL) {
        print_postorder(t->left);
        print_postorder(t->right);
        printf("%d:",t->data);
    }
}
void search(struct node *root, int value) {
    if (root == NULL) {
        printf("\nValue not found!");
        return;
    }
    else {
        if (root->data == value) {
            printf("\nValue found!");
            return;
        } else {
            if (value < root->data)
                search(root->left,value);
            else
            search(root->right,value);
        }
    }
}
struct node *delete(struct node*root, int value) {
    if (root == NULL) {
        printf("\nValue not found to be deleted!");
        return NULL;
    }
    if (value < root->data) {
        root->left = delete(root->left, value);
    } else if (value > root->data) {
        root->right = delete(root->right, value);
    } else {
        if (root->left == NULL && root->right == NULL) {
            free(root);
            return NULL;
        } else if (root->left != NULL && root->right == NULL) {
            struct node * y  = root->left;
            free(root);
            return y;
        } else if (root->right != NULL && root->left == NULL) {
            struct node *z =  root->right;
```

```
            free(root);
            return z;
        } else {
            struct node *succ_parent = root;
            struct node *succ = root->right;
            while (succ->left != NULL) {
                succ_parent = succ;
                succ = succ->left;
            }
            if (succ_parent == root) {
                root->right = succ->right;
            } else {
                root->left = succ->right;
            }
            root->data = succ->data;
            free(succ);
        }
    }
    return root;
}
void main() {
    printf("\nBinary Search Tree!");
    int choice = 0;
    int position = 0;
    int count = 0;
    int value = 0;
    struct node *new_node = NULL;
    struct node *nn = NULL;
    while (1) {
        printf("\n1.Create\n2.Insert\n3.Delete\n4.Search\n5.Print Inorder\n6.Print
PreOrder\n7.Print PostOrder\n6..End");
        printf("\nSelect a choice:");
        scanf("%d",&choice);
        switch (choice) {
            case 1:
            printf("\nCreate!");
            new_node = allocate();
            if (new_node == NULL) {
                printf("\nMalloc failure!");
                exit(1);
            } else {
                printf("\nEnter the value for node!");
                scanf("%d",&new_node->data);
                new_node->left = NULL;
                new_node->right = NULL;
                root = new_node;
            }
            break;
            case 2:
            printf("\nInsert!");
            nn = allocate();
            if (new_node == NULL) {
                printf("\nMalloc failure!");
                exit(1);
            } else {
                printf("\nEnter the value for node!");
                scanf("%d",&nn->data);
```

```
                insert(nn,root);
            }
            break;
            case 3:
            printf("\nDelete!");
            printf("\nEnter value to delete!");
            scanf("%d",&value);
            delete(root,value);
            break;
            case 4:
            printf("\nSearch!");
            printf("\nEnter the value to search!");
            scanf("%d",&value);
            search(root,value);
            break;
            case 5:
            printf("\nBST Inorder!");
            printf("\n");
            print_inorder(root);
            break;
            case 6:
            printf("\nBST Preorder!");
            printf("\n");
            print_preorder(root);
            break;
            case 7:
            printf("\nBST Postorder!");
            printf("\n");
            print_postorder(root);
            break;
            case 8:
            printf("\nThank you!");
            exit(1);
            break;
            default:
            printf("\nTry again!");
            break;
        }
        choice = 0;
    }
}
```

1.174 Trie Complete Code

```
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
#include<string.h>
#define N 26
struct node {
    char data;
    struct node *children[N];
    int is_leaf;
};
struct node *root;
struct node* allocate(char data) {
    struct node *new_node = (struct node *)malloc(sizeof(struct node));
```

```
    if (new_node == NULL) {
        printf("\nMalloc failure!");
        exit(1);
    }
    for (int i = 0; i < N; i++) {
        new_node->children[i] = NULL;
    }
    new_node->data = data;
    new_node->is_leaf = 0;

    return new_node;
}
struct node* insert(struct node *root, char *input) {
    struct node *temp = root;
    int position = 0;
    printf("\nWord input is :%s",input);
    for (int i = 0; input[i] != '\0';  i++) {
        position = (int)input[i] - 'a';
        printf("\nPosition is :%d",position);
        if (temp->children[position] == NULL) {
            temp->children[position] = allocate(input[i]);
            printf("\nCharacter : %c",temp->data);
        } else {
            //Do nothing as character is already present
        }
        temp = temp->children[position];
    }
    temp->is_leaf = 1;
    return root;
}
void print(struct node* root) {
    // Prints the nodes of the trie
    if (!root)
        return;
    struct node* temp = root;
    printf("%c -> ", temp->data);
    for (int i=0; i<N; i++) {
        print(temp->children[i]);
    }
}
void search(struct node *root, char *word) {
    struct node *temp = root;
    int position = 0;
    for (int i = 0; word[i] != '\0'; i++) {
        position = (int)word[i] - 'a';
        if (temp->children[position] == NULL) {
            printf("\nWord not found");
            return;
        } else {
            temp = temp->children[position];
        }
    }
    if (temp->is_leaf == 1) {
        printf("\nWord found!");
    } else {
        printf("\nIncomplete word found!");
    }
```

```c
}
bool is_complete_word(char *word) {
    struct node *temp = root;
    int position = 0;
    for (int i = 0; word[i] != '\0'; i++) {
        position = (int)word[i] - 'a';
        if (temp->children[position] != NULL) {
            temp = temp->children[position];
        }
    }
    return (temp->is_leaf == 1) ? true:false;
}
int check_divergence(struct node *root, char *prefix) {
    struct node *temp = root;
    int c_index = 0;
    for (int i= 0; i < strlen(prefix); i++) {
        int position = prefix[i] - 'a';
        if (temp->children[position]) {
            for (int j = 0; j < N; j++) {
                if (j != position && temp->children[j]) {
                    c_index =  i;
                    break;
                }
            }
            temp = temp->children[position];
        }
    }
    return c_index;
}
char* common_prefix (struct node *root, char *word) {
    char *prefix = (char *)malloc((1 + strlen(word))*sizeof(char));
    strcpy(prefix, word);
    int index = check_divergence(root, prefix);
    if (index >= 0) {
        prefix[index] = '\0';
        prefix = (char *)realloc(prefix, (1+index) * sizeof(char));
    }
    return prefix;
}
void delete(struct node *root, char *word) {
    printf("\nDeletion in progress!");
    if (!is_complete_word(word)) {
        printf("\nIncomplete word cant be deleted!");
        return;
    }
    struct node *temp = root;
    char* cp = common_prefix(root, word);
    int i = 0;
    for (i = 0; cp[i] != '\0'; i++) {
        int position = (int)cp[i] - 'a';
        if (temp->children[position] != NULL) {
            temp = temp->children[position];
        } else {
            return;
        }
    }
    int len = strlen(word);
```

```
    for (; i < len; i++) {
        int position = (int)word[i] - 'a';
        if (temp->children[position] != NULL) {
            struct node* rem = temp->children[position];
            temp->children[position] = NULL;
            free(rem);
        }

    }
}
void main() {
    printf("\nTrie!");
    int choice = 0;
    int position = 0;
    int count = 0;
    int value = 0;
    struct node *new_node = NULL;
    char input[100];
    while (1) {
        printf("\n1.Create\n2.Insert Word\n3.Delete Word\n4.Search Word\n5.Print
Dictionary\n6.End");
        printf("\nSelect a choice:");
        scanf("%d",&choice);
        switch (choice) {
            case 1:
            printf("\nCreate!");
            new_node = allocate('\0');
            if (new_node == NULL) {
                printf("\nMalloc failure!");
                exit(1);
            } else {
                root = new_node;
            }
            break;
            case 2:
            printf("\nInsert Word!");
            printf("\nEnter a word to add to dictionary!");
            scanf("%s",input);
            insert(root, input);
            break;
            case 3:
            printf("\nDelete Word!");
            printf("\nEnter a word to delete from dictionary!");
            scanf("%s",input);
            delete(root, input);
            break;
            case 4:
            printf("\nSearch Word!");
            printf("\nEnter a word to search in dictionary!");
            scanf("%s",input);
            search(root, input);
            break;
            case 5:
            printf("\nPrint Dictionary!");
            print(root);
            break;
            case 6:
```

```
            printf("\nThank you!");
            exit(1);
            break;
            default:
            printf("\nTry again!");
            break;
        }
        choice = 0;
    }
}
```

1.175 Graph Shortest Path

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define MAX_NODES 100

// Node structure for representing edges
typedef struct {
    int destination;
    int weight;
} Edge;

// Graph structure
typedef struct {
    Edge edges[MAX_NODES][MAX_NODES];  // Adjacency matrix
    int numNodes;                      // Number of nodes in the graph
} Graph;

// Function to create a new graph
Graph* createGraph(int numNodes) {
    Graph* graph = (Graph*)malloc(sizeof(Graph));
    graph->numNodes = numNodes;

    // Initialize edges with -1 (indicating no connection)
    for (int i = 0; i < numNodes; i++) {
        for (int j = 0; j < numNodes; j++) {
            graph->edges[i][j].destination = -1;
            graph->edges[i][j].weight = -1;
        }
    }

    return graph;
}

// Function to add an edge to the graph
void addEdge(Graph* graph, int source, int destination, int weight) {
    graph->edges[source][destination].destination = destination;
    graph->edges[source][destination].weight = weight;
}

// Function to find the shortest path using BFS
void shortestPath(Graph* graph, int source, int destination) {
    int distance[MAX_NODES];
    for (int i = 0; i < MAX_NODES; i++) {
```

```c
        distance[i] = -1; // Initialize distances to -1 (infinity)
    }

    Queue q;
    initQueue(&q);
    enqueue(&q, source);
    distance[source] = 0;

    while (!isEmpty(&q)) {
        int current = dequeue(&q);

        for (int i = 0; i < graph->numNodes; i++) {
            int neighbor = graph->edges[current][i].destination;
            int weight = graph->edges[current][i].weight;

            if (graph->edges[current][i].destination != -1 && distance[neighbor] == -1)
{
                distance[neighbor] = distance[current] + weight;
                enqueue(&q, neighbor);
            }
        }
    }

    printf("Shortest distance from %d to %d is: %d\n", source, destination,
distance[destination]);
}

// Driver program
int main() {
    int numNodes = 6;
    Graph* graph = createGraph(numNodes);

    addEdge(graph, 0, 1, 5);
    addEdge(graph, 0, 2, 3);
    addEdge(graph, 1, 3, 6);
    addEdge(graph, 1, 4, 4);
    addEdge(graph, 2, 3, 7);
    addEdge(graph, 2, 5, 8);
    addEdge(graph, 3, 4, 2);
    addEdge(graph, 4, 5, 1);

    int source = 0;
    int destination = 5;

    shortestPath(graph, source, destination);

    return 0;
}
```

1.176 Semaphore

```c
#include <stdio.h>
#include <pthread.h>

typedef struct {
    pthread_mutex_t mutex;
    pthread_cond_t condition;
```

```
    int value;
} semaphore;

void semaphore_init(semaphore* sem, int initial_value) {
    pthread_mutex_init(&(sem->mutex), NULL);
    pthread_cond_init(&(sem->condition), NULL);
    sem->value = initial_value;
}

void semaphore_wait(semaphore* sem) {
    pthread_mutex_lock(&(sem->mutex));
    while (sem->value <= 0) {
        pthread_cond_wait(&(sem->condition), &(sem->mutex));
    }
    sem->value--;
    pthread_mutex_unlock(&(sem->mutex));
}

void semaphore_signal(semaphore* sem) {
    pthread_mutex_lock(&(sem->mutex));
    sem->value++;
    pthread_cond_signal(&(sem->condition));
    pthread_mutex_unlock(&(sem->mutex));
}

void* thread_function(void* arg) {
    semaphore* sem = (semaphore*)arg;

    printf("Thread waiting\n");
    semaphore_wait(sem);

    printf("Thread acquired the semaphore\n");
    // Perform some critical section or shared resource access here

    printf("Thread releasing the semaphore\n");
    semaphore_signal(sem);

    return NULL;
}

int main() {
    semaphore sem;
    semaphore_init(&sem, 1); // Initialize the semaphore with initial value 1

    pthread_t thread;
    pthread_create(&thread, NULL, thread_function, (void*)&sem);

    // Main thread also performs some work
    printf("Main thread performing some work\n");

    // Main thread waits for the semaphore
    printf("Main thread waiting\n");
    semaphore_wait(&sem);

    printf("Main thread acquired the semaphore\n");
    // Perform some critical section or shared resource access here

    printf("Main thread releasing the semaphore\n");
```

```
    semaphore_signal(&sem);

    pthread_join(thread, NULL);

    return 0;
}
```

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t mutex;

void* thread(void* arg)
{
    //wait
    sem_wait(&mutex);
    printf("\nEntered..\n");

    //critical section
    sleep(4);

    //signal
    printf("\nJust Exiting...\n");
    sem_post(&mutex);
}


int main()
{
    sem_init(&mutex, 0, 1);
    pthread_t t1,t2;
    pthread_create(&t1,NULL,thread,NULL);
    sleep(2);
    pthread_create(&t2,NULL,thread,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    sem_destroy(&mutex);
    return 0;
}
```

1.177 Mutex

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

pthread_t tid[2];
int counter;
pthread_mutex_t lock;

void* trythis(void* arg)
{
```

```
    pthread_mutex_lock(&lock);

    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d has started\n", counter);

    for (i = 0; i < (0xFFFFFFFF); i++)
        ;

    printf("\n Job %d has finished\n", counter);

    pthread_mutex_unlock(&lock);

    return NULL;
}

int main(void)
{
    int i = 0;
    int error;

    if (pthread_mutex_init(&lock, NULL) != 0) {
        printf("\n mutex init has failed\n");
        return 1;
    }

    while (i < 2) {
        error = pthread_create(&(tid[i]),
                               NULL,
                               &trythis, NULL);
        if (error != 0)
            printf("\nThread can't be created :[%s]",
                   strerror(error));
        i++;
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);

    return 0;
}
```

1.178 Blocking Queue

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>

#define QUEUE_SIZE 5

typedef struct {
    int* buffer;
    int size;
    int front;
    int rear;
```

```
    pthread_mutex_t mutex;            // Mutex for mutual exclusion
    pthread_cond_t not_empty;        // Condition variable for not empty
    pthread_cond_t not_full;         // Condition variable for not full
} CircularQueue;

// Initialize the circular queue
void initializeQueue(CircularQueue* queue, int size) {
    queue->buffer = (int*)malloc(sizeof(int) * size);
    queue->size = size;
    queue->front = 0;
    queue->rear = 0;
    pthread_mutex_init(&queue->mutex, NULL);
    pthread_cond_init(&queue->not_empty, NULL);
    pthread_cond_init(&queue->not_full, NULL);
}

// Enqueue an item into the circular queue
void enqueue(CircularQueue* queue, int item) {
    pthread_mutex_lock(&queue->mutex);

    while ((queue->rear + 1) % queue->size == queue->front) {
        // Queue is full, wait for space
        pthread_cond_wait(&queue->not_full, &queue->mutex);
    }

    queue->buffer[queue->rear] = item;
    queue->rear = (queue->rear + 1) % queue->size;

    pthread_cond_signal(&queue->not_empty);  // Notify that the queue is not empty
    pthread_mutex_unlock(&queue->mutex);
}

// Dequeue an item from the circular queue
int dequeue(CircularQueue* queue) {
    pthread_mutex_lock(&queue->mutex);

    while (queue->front == queue->rear) {
        // Queue is empty, wait for items
        pthread_cond_wait(&queue->not_empty, &queue->mutex);
    }

    int item = queue->buffer[queue->front];
    queue->front = (queue->front + 1) % queue->size;

    pthread_cond_signal(&queue->not_full);  // Notify that the queue is not full
    pthread_mutex_unlock(&queue->mutex);

    return item;
```

```c
}

// Clean up the circular queue
void cleanupQueue(CircularQueue* queue) {
    free(queue->buffer);
    pthread_mutex_destroy(&queue->mutex);
    pthread_cond_destroy(&queue->not_empty);
    pthread_cond_destroy(&queue->not_full);
}

// Example usage
void* producer(void* arg) {
    CircularQueue* queue = (CircularQueue*)arg;

    for (int i = 1; i <= 10; ++i) {
        enqueue(queue, i);
        printf("Produced: %d\n", i);
    }

    pthread_exit(NULL);
}
void* consumer(void* arg) {
    CircularQueue* queue = (CircularQueue*)arg;

    for (int i = 0; i < 10; ++i) {
        int item = dequeue(queue);
        printf("Consumed: %d\n", item);
    }

    pthread_exit(NULL);
}

int main() {
    CircularQueue queue;
    initializeQueue(&queue, QUEUE_SIZE);

    pthread_t producerThread, consumerThread;

    pthread_create(&producerThread, NULL, producer, &queue);
    pthread_create(&consumerThread, NULL, consumer, &queue);

    pthread_join(producerThread, NULL);
    pthread_join(consumerThread, NULL);

    cleanupQueue(&queue);

    return 0;
}
```

1.179 String to Integer(atoi)

```
int myAtoi(char * s){
        int i = 0;
        int j = strlen(s)-1;
        int result = 0;
        int sign = 1;
        if (isalpha(s[i])) {
            //printf("\n1");
            return 0;
        }
        while (s[i] == ' ') {
                //printf("\n2");
                i++;
        }
        if (s[i] == '+') {
                //printf("\n3");
                sign = 1;
                i++;
        } else if (s[i] == '-'){
                //printf("\n4");
                sign = -1;
                i++;
        } else {
            sign = 1;
        }
        while (i <= j) {
            //printf("\n5");
            if (isdigit(s[i])) {
                //printf("\n6");
                if ((result > INT_MAX/10) ||
                    (result == INT_MAX/10) && (s[i]-'0') > INT_MAX%10) {
                     printf("\na");
            return (sign == -1)? INT_MIN:INT_MAX;
                }



                result = 10*result + (s[i]-'0');
                //printf("\nres : %d",result);
                i++;
            } else {
                //printf("\n7");
                i++;
                break;
            }
        }
        //printf("\n8");
        result = result*sign;
```

```
        return result;
}
```

1.180 Add Two Numbers

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

 // Function to create a new node with a given value
struct ListNode* newNode(int val) {
    struct ListNode* node = (struct ListNode*)malloc(sizeof(struct ListNode));
    node->val = val;
    node->next = NULL;
    return node;
}
struct ListNode* addTwoNumbers(struct ListNode* l1, struct ListNode* l2) {
    struct ListNode* dummyHead = (struct ListNode*)malloc(sizeof(struct ListNode));
    struct ListNode* current = dummyHead;
    int carry = 0;

    while (l1 != NULL || l2 != NULL) {
        int x = (l1 != NULL) ? l1->val : 0;
        int y = (l2 != NULL) ? l2->val : 0;
        int sum = x + y + carry;

        carry = sum / 10;

        current->next = (struct ListNode*)malloc(sizeof(struct ListNode));
        current->next->val = sum % 10;
        current->next->next = NULL;

        current = current->next;

        if (l1 != NULL) l1 = l1->next;
        if (l2 != NULL) l2 = l2->next;
    }

    if (carry > 0) {
        current->next = (struct ListNode*)malloc(sizeof(struct ListNode));
        current->next->val = carry;
        current->next->next = NULL;
    }

    struct ListNode* result = dummyHead->next;
    free(dummyHead);
```

```
    return result;
}
```

1.181 Next Permutation

```c
#include <stdio.h>

// Function to swap two elements in an array
void swap(int* nums, int i, int j) {
    int temp = nums[i];
    nums[i] = nums[j];
    nums[j] = temp;
}

// Function to reverse a portion of an array
void reverse(int* nums, int start, int end) {
    int i = start, j = end;
    while (i < j) {
        swap(nums, i, j);
        i++;
        j--;
    }
}

// Function to find the next permutation
void nextPermutation(int* nums, int numsSize) {
    //First find the lesser element in the array starting from last
    int i = numsSize - 2;
    while (i >= 0 && nums[i + 1] <= nums[i]) {
        i--;
    }
    //Then find the element which is greater than that element
    if (i >= 0) {
        int j = numsSize - 1;
        while (nums[j] <= nums[i]) {
            j--;
        }
        //Swap those numbers
        swap(nums, i, j);
    }
    //Reverse the numbers from that index to the found index (ascending order)
    reverse(nums, i + 1, numsSize - 1);
}
```

```java
public class Solution {
    public void nextPermutation(int[] nums) {
        int i = nums.length - 2;
        while (i >= 0 && nums[i + 1] <= nums[i]) {
            i--;
```

```
        }
        if (i >= 0) {
            int j = nums.length - 1;
            while (nums[j] <= nums[i]) {
                j--;
            }
            swap(nums, i, j);
        }
        reverse(nums, i + 1);
    }

    private void reverse(int[] nums, int start) {
        int i = start, j = nums.length - 1;
        while (i < j) {
            swap(nums, i, j);
            i++;
            j--;
        }
    }

    private void swap(int[] nums, int i, int j) {
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
}
```

General Permutation

```
#include <stdio.h>

// Function to swap two elements in an array
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Function to generate permutations
void generatePermutations(int arr[], int size, int index) {
    if (index == size - 1) {
        // Print the current permutation
        for (int i = 0; i < size; i++) {
            printf("%d ", arr[i]);
        }
        printf("\n");
        return;
    }

    for (int i = index; i < size; i++) {
        // Swap the current element with the element at index
```

```
        swap(&arr[index], &arr[i]);

        // Recursively generate permutations for the remaining elements
        generatePermutations(arr, size, index + 1);

        // Undo the swap to backtrack
        swap(&arr[index], &arr[i]);
    }
}

int main() {
    // Example usage
    int arr[] = {1, 2, 3};
    int size = sizeof(arr) / sizeof(arr[0]);

    printf("Permutations:\n");
    generatePermutations(arr, size, 0);

    return 0;
}
```

1.182 First Missing Positive

```
int firstMissingPositive(int* nums, int numsSize) {
    // Create a boolean array to mark the presence of positive numbers
    bool present[numsSize + 1];
    for (int i = 0; i <= numsSize; i++) {
        present[i] = false;
    }

    // Mark positive numbers in the array
    for (int i = 0; i < numsSize; i++) {
        if (nums[i] > 0 && nums[i] <= numsSize) {
            present[nums[i]] = true;
        }
    }

    // Find the first missing positive number
    for (int i = 1; i <= numsSize; i++) {
        if (!present[i]) {
            return i;
        }
    }

    // If all positive numbers are present, return the next positive number
    return numsSize + 1;
}
```

1.183 Permutations

```
/**
```

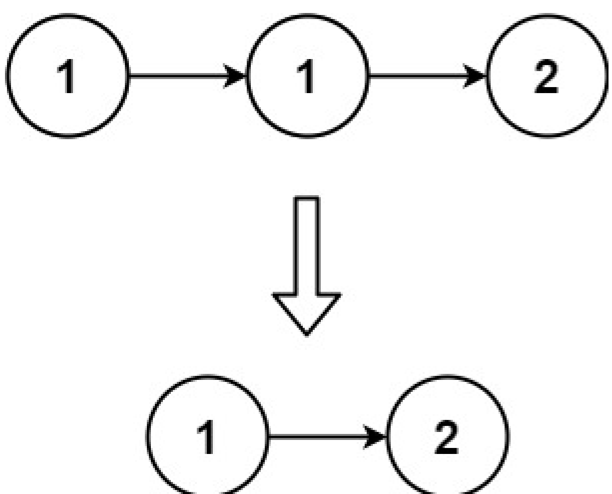
```c
 * Return an array of arrays of size *returnSize.
 * The sizes of the arrays are returned as *returnColumnSizes array.
 * Note: Both returned array and *columnSizes array must be malloced, assume caller
calls free().
 */
// Function to swap two elements in an array
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Function to generate permutations
void generatePermutations(int* nums, int numsSize, int index, int** result, int*
returnSize, int* returnColumnSizes) {
    if (index == numsSize - 1) {
        // Allocate memory for the current permutation
        result[*returnSize] = (int*)malloc(numsSize * sizeof(int));
        // Copy the current permutation to the result array
        for (int i = 0; i < numsSize; i++) {
            result[*returnSize][i] = nums[i];
        }
        // Update returnColumnSizes with the size of the current permutation
        returnColumnSizes[*returnSize] = numsSize;
        // Increment returnSize to keep track of the number of permutations
        (*returnSize)++;
        return;
    }

    for (int i = index; i < numsSize; i++) {
        // Swap the current element with the element at index
        swap(&nums[index], &nums[i]);

        // Recursively generate permutations for the remaining elements
        generatePermutations(nums, numsSize, index + 1, result, returnSize,
returnColumnSizes);

        // Undo the swap to backtrack
        swap(&nums[index], &nums[i]);
    }
}

int** permute(int* nums, int numsSize, int* returnSize, int** returnColumnSizes) {
    // Calculate the number of permutations (n!)
    int totalPermutations = 1;
    for (int i = 1; i <= numsSize; i++) {
        totalPermutations *= i;
    }
```

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

    // Allocate memory for returnColumnSizes array
    *returnColumnSizes = (int*)malloc(totalPermutations * sizeof(int));

    // Initialize returnSize
    *returnSize = 0;

    // Generate permutations
    generatePermutations(nums, numsSize, 0, result, returnSize, *returnColumnSizes);

    return result;
}
```

1.184 Remove Duplicates From Sorted List

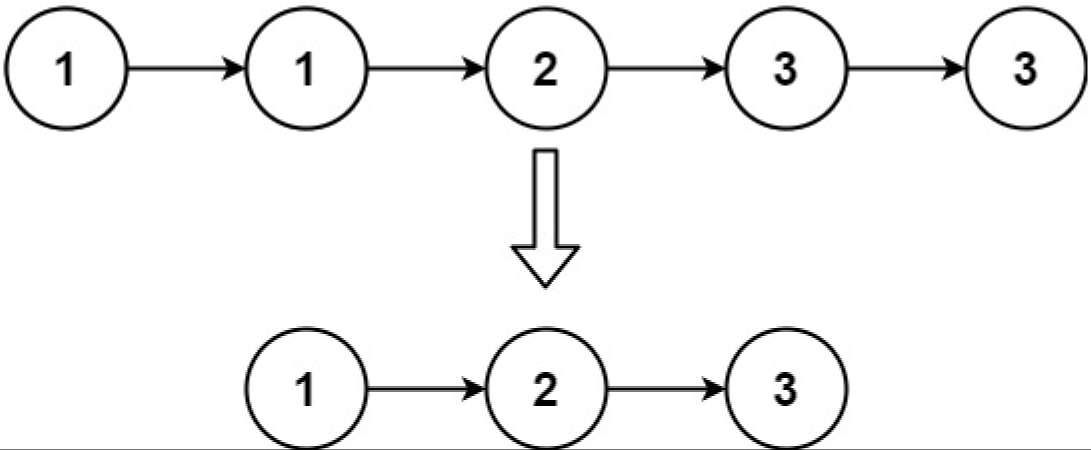
Given the head of a sorted linked list, *delete all duplicates such that each element appears only once*. Return *the linked list **sorted** as well*.

Example 1:



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

Example 2:



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

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */
// Function to delete duplicates from a sorted linked list
struct ListNode* deleteDuplicates(struct ListNode* head) {
    struct ListNode* current = head;

    // Traverse the list
    while (current != NULL && current->next != NULL) {
        // Check if the current and next nodes have the same value
        if (current->val == current->next->val) {
            // Duplicate found, remove the next node
            struct ListNode* duplicate = current->next;
            current->next = current->next->next;
            free(duplicate);
        } else {
            // Move to the next node
            current = current->next;
        }
    }

    return head;
}

// Function to print the linked list
void printList(struct ListNode* head) {
    struct ListNode* current = head;
    while (current != NULL) {
        printf("%d -> ", current->val);
        current = current->next;
    }
```

```
    printf("NULL\n");
}

// Function to create a new node with the given value
struct ListNode* newNode(int val) {
    struct ListNode* node = (struct ListNode*)malloc(sizeof(struct ListNode));
    node->val = val;
    node->next = NULL;
    return node;
}
```

1.185 Binary Tree InOrder Traversal

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */
/**
 * Note: The returned array must be malloced, assume caller calls free().
 */
// Function to perform inorder traversal of a binary tree
void inorderTraversalHelper(struct TreeNode* root, int* result, int* index) {
    if (root == NULL) {
        return;
    }

    // Traverse the left subtree
    inorderTraversalHelper(root->left, result, index);

    // Visit the current node
    result[(*index)++] = root->val;

    // Traverse the right subtree
    inorderTraversalHelper(root->right, result, index);
}

// Function to get the size of a binary tree
int getSize(struct TreeNode* root) {
    if (root == NULL) {
        return 0;
    }

    return 1 + getSize(root->left) + getSize(root->right);
}
```

```
// Function to perform inorder traversal and return the result as an array
int* inorderTraversal(struct TreeNode* root, int* returnSize) {
    // Get the size of the binary tree
    *returnSize = getSize(root);

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

    // Index to keep track of the position in the result array
    int index = 0;

    // Perform inorder traversal
    inorderTraversalHelper(root, result, &index);

    return result;
}
```

1.186 Binary Tree ZigZag Level Order Traversal

```
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++) {
```

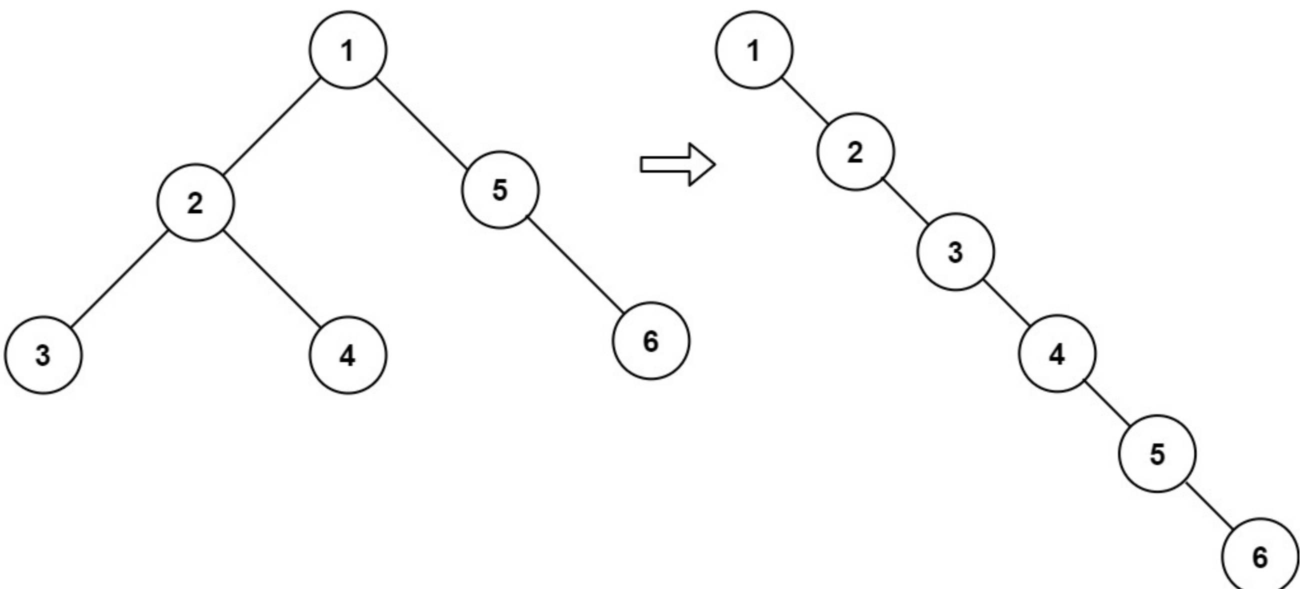
```
            if (i % 2 == 1) {
                Collections.reverse(levels.get(i));
            }
        }
        return levels;
    }
}
```

1.187 Flatten Binary Tree To Linked List

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

- The "linked list" should use the same TreeNode class where the right child pointer points to the next node in the list and the left child pointer is always null.
- The "linked list" should be in the same order as a **pre-order traversal** of the binary tree.

Example 1:



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

Example 2:

Input: root = []
Output: []

Example 3:

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

```
void flatten(struct TreeNode* root) {
```

```
    if (root == NULL) {
        return;
    }

    struct TreeNode* current = root;

    while (current != NULL) {
        // If the current node has a left child
        if (current->left != NULL) {
            // Find the rightmost node in the left subtree
            struct TreeNode* rightmost = current->left;
            while (rightmost->right != NULL) {
                rightmost = rightmost->right;
            }

            // Move the right subtree to the rightmost node's right child
            rightmost->right = current->right;

            // Move the left subtree to the right child
            current->right = current->left;
            current->left = NULL;
        }

        // Move to the next node in the flattened tree
        current = current->right;
    }
}

// Function to create a new binary tree node with the given value
struct TreeNode* newNode(int val) {
    struct TreeNode* node = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    node->val = val;
    node->left = NULL;
    node->right = NULL;
    return node;
}

// Function to print the flattened linked list
void printFlattenedList(struct TreeNode* root) {
    struct TreeNode* current = root;
    while (current != NULL) {
        printf("%d -> ", current->val);
        current = current->right;
    }
    printf("NULL\n");
}
```

1.188 Binary Tree Post Order Traversal

```
// Function to get the size of a binary tree
int getSize(struct TreeNode* root) {
```

```
    if (root == NULL) {
        return 0;
    }

    return 1 + getSize(root->left) + getSize(root->right);
}

// Function to perform postorder traversal of a binary tree
void postorderTraversalHelper(struct TreeNode* root, int* result, int* index) {
    if (root == NULL) {
        return;
    }

    // Traverse the left subtree
    postorderTraversalHelper(root->left, result, index);

    // Traverse the right subtree
    postorderTraversalHelper(root->right, result, index);

    // Visit the current node
    result[(*index)++] = root->val;
}

// Function to perform postorder traversal and return the result as an array
int* postorderTraversal(struct TreeNode* root, int* returnSize) {
    // Get the size of the binary tree
    *returnSize = getSize(root);

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

    // Index to keep track of the position in the result array
    int index = 0;

    // Perform postorder traversal
    postorderTraversalHelper(root, result, &index);

    return result;
}
```

1.189 Binary Tree Pre Order Traversal

```
int getSize(struct TreeNode* root) {
    if (root == NULL) {
        return 0;
    }

    return 1 + getSize(root->left) + getSize(root->right);
}
```

```c
// Function to perform preorder traversal of a binary tree
void preorderTraversalHelper(struct TreeNode* root, int* result, int* index) {
    if (root == NULL) {
        return;
    }

    // Visit the current node
    result[(*index)++] = root->val;

    // Traverse the left subtree
    preorderTraversalHelper(root->left, result, index);

    // Traverse the right subtree
    preorderTraversalHelper(root->right, result, index);
}

// Function to perform preorder traversal and return the result as an array
int* preorderTraversal(struct TreeNode* root, int* returnSize) {
    // Get the size of the binary tree
    *returnSize = getSize(root);

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

    // Index to keep track of the position in the result array
    int index = 0;

    // Perform preorder traversal
    preorderTraversalHelper(root, result, &index);

    return result;
}

// Function to create a new binary tree node with the given value
struct TreeNode* newNode(int val) {
    struct TreeNode* node = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    node->val = val;
    node->left = NULL;
    node->right = NULL;
    return node;
}
```

1.190 Excel Sheet Column Number

```c
int titleToNumber(char* columnTitle) {
    int result = 0;
    int length = strlen(columnTitle);

    for (int i = length - 1; i >= 0; i--) {
        result += pow(26, (length - 1 - i)) * (columnTitle[i] - 'A' + 1);
```

```
    }

    return result;
}
```

```
class Solution {
  public int titleToNumber(String s) {
    int result = 0;
    int n = s.length();
    for (int i = 0; i < n; i++) {
      result = result * 26;
      // In Java, subtracting characters is subtracting ASCII values of characters
      result += (s.charAt(i) - 'A' + 1);
    }
    return result;
  }
}
```

1.191 Ugly Number

```
bool isUgly(int n) {
       if (n <= 0) {
        return false;
    }

    while (n % 2 == 0) {
        n /= 2;
    }
    while (n % 3 == 0) {
        n /= 3;
    }
    while (n % 5 == 0) {
        n /= 5;
    }

    return n == 1;
}
```

1.192 Game Of Life

```
#define DIE 2
#define LIVE 3

int countLive(int i, int j, int** board, int rows, int cols) {
    int count = 0;
    int dirs[8][2] = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}, {1, 1}, {1, -1}, {-1, 1}, {-1, -
1}};

    for (int k = 0; k < 8; k++) {
        int x = i + dirs[k][0];
        int y = j + dirs[k][1];
```

```
        if (x >= 0 && y >= 0 && x < rows && y < cols) {
            if (board[x][y] == 1 || board[x][y] == DIE)
                count++;
        }
    }

    return count;
}

void gameOfLife(int** board, int boardSize, int* boardColSize) {
    if (!board || boardSize == 0 || !boardColSize || *boardColSize == 0)
        return;

    int rows = boardSize;
    int cols = *boardColSize;

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            int around = countLive(i, j, board, rows, cols);
            if (board[i][j] == 0 && around == 3)
                board[i][j] = LIVE;
            else if (board[i][j] == 1) {
                if (around == 2 || around == 3)
                    continue;
                if (around < 2 || around > 3)
                    board[i][j] = DIE;
            }
        }
    }

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (board[i][j] == DIE)
                board[i][j] = 0;
            if (board[i][j] == LIVE)
                board[i][j] = 1;
        }
    }
}
```

1.193 Minimum Number Of Arrows To Burst Balloons

```
// Comparison function for qsort
int compareIntervals(const void* a, const void* b) {
    int* intervalA = *(int**)a;
    int* intervalB = *(int**)b;

    if (intervalA[1] == intervalB[1]) {
        return 0;
    } else if (intervalA[1] < intervalB[1]) {
```

```
        return -1;
    } else {
        return 1;
    }
}

int findMinArrowShots(int** points, int pointsSize, int* pointsColSize) {
    if (pointsSize == 0) {
        return 0;
    }

    // Sort the intervals based on their end points
    qsort(points, pointsSize, sizeof(int*), compareIntervals);
    for (int i = 0; i < pointsSize; i++) {
        printf("\n%d %d",points[i][0], points[i][1]);
    }
    int arrows = 1;
    int xStart, xEnd, firstEnd = points[0][1];
    for (int i = 1; i < pointsSize; i++) {
        xStart = points[i][0];
        xEnd = points[i][1];

        // If the current balloon starts after the end of another one,
        // one needs one more arrow
        if (firstEnd < xStart) {
            arrows++;
            firstEnd = xEnd;
        }
    }

    return arrows;
}
```

1.194 Exclusive Time Of Functions

```
public class Solution {
    public int[] exclusiveTime(int n, List<String> logs) {
        // separate time to several intervals, add interval to their function
        int[] res = new int[n];
        //store id, not timestamp
        Stack<Integer> idStack = new Stack<>();
        //store timestamp: prevLogTime here is to maintain previous functions time
        // and if it is start function we just add it to previous if it is an end
function
        // then add plus one to mark the end of the top of stack function
        //(It's used to record last log time)
        int prevLogTime = 0;
        // pre means the start of the interval
        for (String s : logs) {
            String[] log = s.split(":");
```

```
            if (log[1].equals("start")) {
                if (!idStack.isEmpty()) res[idStack.peek()] += Integer.parseInt(log[2])
- prevLogTime;
                // log[2] is the start of next interval, doesn't belong to current
interval.
                idStack.push(Integer.parseInt(log[0]));
                prevLogTime = Integer.parseInt(log[2]);
            } else {
                res[idStack.pop()] += Integer.parseInt(log[2]) - prevLogTime + 1;
                // log[2] is end of current interval, belong to current interval. That's
why we have +1 here
                prevLogTime = Integer.parseInt(log[2]) + 1;
                // prevLogTime means the start of next interval, so we need to +1
            }
        }
        return res;
    }
}
```

1.195 Top K Frequent Words

```
class Solution {
    public List<String> topKFrequent(String[] words, int k) {
        HashMap<String,Integer> hm = new HashMap<String,Integer>();
        for (String s:words) {
            hm.put(s, hm.getOrDefault(s,0)+1);
        }

    Queue<Map.Entry<String, Integer>> pq = new PriorityQueue<>(
        (a, b) -> a.getValue().equals(b.getValue()) ? a.getKey().compareTo(b.   getKey
()) : b.getValue() - a.getValue()
);

        for(HashMap.Entry<String,Integer> entry: hm.entrySet()) {
            pq.add(entry);

        }
        System.out.println(pq.toString());
        List<String> l = new ArrayList<String>();
        int i = 0;
        while (i < k) {
            l.add(pq.poll().getKey());
            System.out.println("Now"+pq.toString());
            i++;
        }


        return l;
    }
}
```

1.196 Walking Robot Simulation

```c
#include <stdlib.h>
#include <math.h>

// Define a structure to represent a point
struct Point {
    int x;
    int y;
};

// Function to calculate the squared Euclidean distance
int calculateDistanceSquared(struct Point p) {
    return p.x * p.x + p.y * p.y;
}

int robotSim(int* commands, int commandsSize, int** obstacles, int obstaclesSize, int*
obstaclesColSize) {
    // Define the four possible directions: North, East, South, West
    int directions[4][2] = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

    // Initialize the current position and direction of the robot
    struct Point currentPosition = {0, 0};
    int currentDirection = 0; // 0 represents North

    // Create a set to store obstacle positions for quick lookup
    int** obstacleSet = (int**)malloc(obstaclesSize * sizeof(int*));
    for (int i = 0; i < obstaclesSize; i++) {
        obstacleSet[i] = (int*)malloc(2 * sizeof(int));
        obstacleSet[i][0] = obstacles[i][0];
        obstacleSet[i][1] = obstacles[i][1];
    }

    // Initialize the result (maximum squared distance)
    int result = 0;

    // Iterate through the commands
    for (int i = 0; i < commandsSize; i++) {
        if (commands[i] == -1) {
            // Turn right 90 degrees
            currentDirection = (currentDirection + 1) % 4;
        } else if (commands[i] == -2) {
            // Turn left 90 degrees
            currentDirection = (currentDirection + 3) % 4;
        } else {
            // Move forward k units
            for (int j = 0; j < commands[i]; j++) {
                struct Point nextPosition = {currentPosition.x +
directions[currentDirection][0],
```

```
                                             currentPosition.y +
directions[currentDirection][1]};

                // Check for obstacles
                int obstacleDetected = 0;
                for (int k = 0; k < obstaclesSize; k++) {
                    if (nextPosition.x == obstacleSet[k][0] && nextPosition.y ==
obstacleSet[k][1]) {
                        obstacleDetected = 1;
                        break;
                    }
                }

                // Update position if no obstacle is detected
                if (!obstacleDetected) {
                    currentPosition = nextPosition;
                    result = fmax(result, calculateDistanceSquared(currentPosition));
                } else {
                    // Break out of the loop if an obstacle is detected
                    break;
                }
            }
        }
    }

    // Free memory allocated for the obstacle set
    for (int i = 0; i < obstaclesSize; i++) {
        free(obstacleSet[i]);
    }
    free(obstacleSet);

    return result;
}
```

1.197 Cousins In Binary Tree

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

// Structure to represent a queue node for level-order traversal
struct QueueNode {
    struct TreeNode* node;
    int depth;
```

```
    int parent;
};

// Queue functions for level-order traversal
struct Queue {
    int front, rear;
    int capacity;
    struct QueueNode* array;
};

struct Queue* createQueue(int capacity) {
    struct Queue* queue = (struct Queue*)malloc(sizeof(struct Queue));
    queue->capacity = capacity;
    queue->front = queue->rear = -1;
    queue->array = (struct QueueNode*)malloc(capacity * sizeof(struct QueueNode));
    return queue;
}

void enqueue(struct Queue* queue, struct TreeNode* node, int depth, int parent) {
    struct QueueNode qNode = {node, depth, parent};
    if (queue->front == -1) {
        queue->front = queue->rear = 0;
    } else {
        queue->rear = (queue->rear + 1) % queue->capacity;
    }
    queue->array[queue->rear] = qNode;
}

struct QueueNode dequeue(struct Queue* queue) {
    struct QueueNode qNode = queue->array[queue->front];
    if (queue->front == queue->rear) {
        queue->front = queue->rear = -1;
    } else {
        queue->front = (queue->front + 1) % queue->capacity;
    }
    return qNode;
}

bool isCousins(struct TreeNode* root, int x, int y) {
    struct Queue* queue = createQueue(1000); // Assuming a maximum of 1000 nodes in the
tree
    enqueue(queue, root, 0, -1); // Root has depth 0 and no parent

    int parentX = -1, parentY = -1;
    int depthX = -1, depthY = -1;

    while (queue->front != -1) {
        int levelSize = queue->rear - queue->front + 1;
        for (int i = 0; i < levelSize; i++) {
```

```
            struct QueueNode qNode = dequeue(queue);
            struct TreeNode* node = qNode.node;
            int depth = qNode.depth;
            int parent = qNode.parent;

            if (node->val == x) {
                parentX = parent;
                depthX = depth;
            } else if (node->val == y) {
                parentY = parent;
                depthY = depth;
            }

            if (node->left) {
                enqueue(queue, node->left, depth + 1, node->val);
            }
            if (node->right) {
                enqueue(queue, node->right, depth + 1, node->val);
            }
        }

        // Check if both nodes are found at the same level
        if ((parentX != -1 && parentY != -1) && (depthX == depthY) && (parentX !=
parentY)) {
            return true;
        }

        // Reset variables for the next level
        parentX = parentY = -1;
        depthX = depthY = -1;
    }

    return false;
}
```

1.198 Maximum Equal Frequency

```
class Solution {
    public int maxEqualFreq(int[] nums) {
        int maxLen = 0;
        int n = nums.length;
        HashMap<Integer, Integer> freqMap = new HashMap<>();
        HashMap<Integer, Integer> countMap = new HashMap<>();

        for (int i = 0; i < n; i++) {
            int num = nums[i];

            // Update the frequency count for the current number
            freqMap.put(num, freqMap.getOrDefault(num, 0) + 1);
```

```
            // Update the countMap based on the frequency count
            int freq = freqMap.get(num);
            int count = countMap.getOrDefault(freq, 0);
            countMap.put(freq, count + 1);

            // Check if removing one element makes all frequencies equal
            if (count * freq == i || (count - 1) * (freq - 1) == i) {
                maxLen = i + 1;
            }
        }

        return maxLen;
    }
}
```

1.199 Packet Parser

```
#include <stdio.h>
#include <stdint.h>
#include <arpa/inet.h>

// Ethernet header structure
struct EthernetHeader {
    uint8_t destMAC[6];
    uint8_t srcMAC[6];
    uint16_t etherType;
};

// IPv4 header structure
struct IPv4Header {
    uint8_t versionIHL;
    uint8_t dscpECN;
    uint16_t totalLength;
    uint16_t identification;
    uint16_t flagsFragmentOffset;
    uint8_t timeToLive;
    uint8_t protocol;
    uint16_t headerChecksum;
    uint32_t srcIP;
    uint32_t destIP;
};

// TCP header structure
struct TCPHeader {
    uint16_t srcPort;
    uint16_t destPort;
    uint32_t sequenceNumber;
    uint32_t ackNumber;
    uint8_t dataOffsetReserved;
    uint8_t flags;
    uint16_t windowSize;
    uint16_t checksum;
```

```
    uint16_t urgentPointer;
};

void processEthernetFrame(const uint8_t* frame, uint16_t frameLength);
void processIPv4Packet(const uint8_t* packet, uint16_t packetLength);
void processTCPHeader(const uint8_t* tcpHeader, uint16_t tcpHeaderLength);
void processApplicationData(const uint8_t* data, uint16_t dataLength);

void printMACAddress(const uint8_t* mac) {
    printf("%02x:%02x:%02x:%02x:%02x:%02x", mac[0], mac[1], mac[2], mac[3], mac[4], mac[5]);
}

int main() {
    // Example Ethernet frame containing IPv4, TCP, and application data
    uint8_t frame[] = {
        // Ethernet header
        0x00, 0x11, 0x22, 0x33, 0x44, 0x55, // Source MAC
        0x66, 0x77, 0x88, 0x99, 0xAA, 0xBB, // Destination MAC
        0x08, 0x00,                         // EtherType (IPv4)
        // IPv4 header
        0x45, 0x00, 0x00, 0x3C, 0x12, 0x34, 0x40, 0x00, 0x40, 0x06, 0x00, 0x00,
        0xC0, 0xA8, 0x01, 0x01,             // Source IP: 192.168.1.1
        0xC0, 0xA8, 0x01, 0x02,             // Destination IP: 192.168.1.2
        // TCP header
        0x00, 0x50, 0x00, 0x37, 0xB8, 0xA0, 0x00, 0x00, 0x00, 0x00, 0xA0, 0x02,
        0xFA, 0xF0, 0x00, 0x00, 0x02, 0x04, 0x05, 0xB4, 0x01, 0x03, 0x03, 0x06,
        // Application data (HTTP request)
        0x47, 0x45, 0x54, 0x20, 0x2F, 0x20, 0x48, 0x54, 0x54, 0x50, 0x2F, 0x31,
        0x2E, 0x31, 0x0D, 0x0A, 0x48, 0x6F, 0x73, 0x74, 0x3A, 0x20, 0x77, 0x77,
        // ... (HTTP request continues)
    };

    processEthernetFrame(frame, sizeof(frame));

    return 0;
}

void processEthernetFrame(const uint8_t* frame, uint16_t frameLength) {
    if (frameLength < sizeof(struct EthernetHeader)) {
        printf("Invalid Ethernet frame\n");
        return;
    }

    struct EthernetHeader* ethHeader = (struct EthernetHeader*)frame;

    printf("Ethernet Frame - Source MAC: ");
    printMACAddress(ethHeader->srcMAC);
    printf(", Destination MAC: ");
    printMACAddress(ethHeader->destMAC);
    printf(", EtherType: %04x\n", ntohs(ethHeader->etherType));

    // Assuming Ethernet frame contains IPv4 packet
    if (ntohs(ethHeader->etherType) == 0x0800) {
        processIPv4Packet(frame + sizeof(struct EthernetHeader), frameLength - sizeof(struct
EthernetHeader));
    } else {
        printf("Unsupported EtherType: %04x\n", ntohs(ethHeader->etherType));
    }
```

```
}

void processIPv4Packet(const uint8_t* packet, uint16_t packetLength) {
    if (packetLength < sizeof(struct IPv4Header)) {
        printf("Invalid IPv4 packet\n");
        return;
    }

    struct IPv4Header* ipv4Header = (struct IPv4Header*)packet;
    printf("IPv4 Packet - Source IP: %s, Destination IP: %s\n",
           inet_ntoa(*(struct in_addr*)&ipv4Header->srcIP), inet_ntoa(*(struct in_addr*)&ipv4Header->destIP));

    // Assuming IPv4 packet contains TCP header
    processTCPHeader(packet + (ipv4Header->versionIHL & 0x0F) * 4, packetLength - (ipv4Header-
>versionIHL & 0x0F) * 4);
}

void processTCPHeader(const uint8_t* tcpHeader, uint16_t tcpHeaderLength) {
    if (tcpHeaderLength < sizeof(struct TCPHeader)) {
        printf("Invalid TCP header\n");
        return;
    }

    struct TCPHeader* tcpHeaderStruct = (struct TCPHeader*)tcpHeader;
    printf("TCP Header - Source Port: %u, Destination Port: %u\n",
           ntohs(tcpHeaderStruct->srcPort), ntohs(tcpHeaderStruct->destPort));

    // Assuming TCP header contains application data
    processApplicationData(tcpHeader + (tcpHeaderStruct->dataOffsetReserved >> 4) * 4, tcpHeaderLength -
(tcpHeaderStruct->dataOffsetReserved >> 4) * 4);
}

void processApplicationData(const uint8_t* data, uint16_t dataLength) {
    // Add code to process application-specific data
    printf("Application Data (%u bytes):\n", dataLength);
    for (uint16_t i = 0; i < dataLength; ++i) {
        printf("%c", data[i]);
    }
    printf("\n");
}
```

1.200 Packet Parser Multithreading

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// Define packet structures (Ethernet, IP, TCP, Application)
typedef struct {
    // Ethernet header fields
    // ...

    // IP header fields
    // ...
} EthernetPacket;

typedef struct {
    // TCP header fields
    // ...
```

```c
} TCPPacket;

typedef struct {
    // Application layer payload
    // ...
} ApplicationPacket;

// Packet structure containing all headers
typedef struct {
    EthernetPacket ethernet;
    TCPPacket tcp;
    ApplicationPacket app;
} FullPacket;

// Function to process a packet
void processPacket(FullPacket* packet) {
    // Process packet headers and payload
    // ...

    // Example: Print packet details
    printf("Processed packet\n");
}

// Function to simulate packet reception
FullPacket generateRandomPacket() {
    // Simulate packet creation with random data
    FullPacket packet;
    // ...

    return packet;
}

// Thread function for packet processing
void* processThread(void* arg) {
    for (;;) {
        FullPacket packet = generateRandomPacket();
        processPacket(&packet);
    }
    return NULL;
}

int main() {
    // Create threads for packet processing
    pthread_t threads[2];
    for (int i = 0; i < 2; ++i) {
        if (pthread_create(&threads[i], NULL, processThread, NULL) != 0) {
            fprintf(stderr, "Error creating thread\n");
            exit(EXIT_FAILURE);
        }
    }

    // Wait for threads to finish (this is a simple example, in a real scenario, threads may run indefinitely)
    for (int i = 0; i < 2; ++i) {
        pthread_join(threads[i], NULL);
    }

    return 0;
}
```

```c
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <pthread.h>

// Define Ethernet, IPv4, TCP, and application header structures as before

struct EthernetHeader {
    uint8_t destMAC[6];
    uint8_t srcMAC[6];
    uint16_t etherType;
};

struct IPv4Header {
    uint8_t versionIHL;
    uint8_t dscpECN;
    uint16_t totalLength;
    uint16_t identification;
    uint16_t flagsFragmentOffset;
    uint8_t timeToLive;
    uint8_t protocol;
    uint16_t headerChecksum;
    uint32_t srcIP;
    uint32_t destIP;
};

struct TCPHeader {
    uint16_t srcPort;
    uint16_t destPort;
    uint32_t sequenceNumber;
    uint32_t ackNumber;
    uint8_t dataOffsetReserved;
    uint8_t flags;
    uint16_t windowSize;
    uint16_t checksum;
    uint16_t urgentPointer;
};

// Define a structure to hold frame data
struct FrameData {
    uint8_t* frame;
    uint16_t frameLength;
};

void* processFrame(void* arg);

int main() {
    // Example Ethernet frames (add more frames as needed)
    uint8_t frame1[] = { /* Frame 1 data */ };
    uint8_t frame2[] = { /* Frame 2 data */ };
        uint8_t frame[] = {
        // Ethernet header
        0x00, 0x11, 0x22, 0x33, 0x44, 0x55, // Source MAC
        0x66, 0x77, 0x88, 0x99, 0xAA, 0xBB, // Destination MAC
        0x08, 0x00,                         // EtherType (IPv4)
        // IPv4 header
```

```
        0x45, 0x00, 0x00, 0x3C, 0x12, 0x34, 0x40, 0x00, 0x40, 0x06, 0x00, 0x00,
        0xC0, 0xA8, 0x01, 0x01,             // Source IP: 192.168.1.1
        0xC0, 0xA8, 0x01, 0x02,             // Destination IP: 192.168.1.2
        // TCP header
        0x00, 0x50, 0x00, 0x37, 0xB8, 0xA0, 0x00, 0x00, 0x00, 0x00, 0xA0, 0x02,
        0xFA, 0xF0, 0x00, 0x00, 0x02, 0x04, 0x05, 0xB4, 0x01, 0x03, 0x03, 0x06,
        // Application data (HTTP request)
        0x47, 0x45, 0x54, 0x20, 0x2F, 0x20, 0x48, 0x54, 0x54, 0x50, 0x2F, 0x31,
        0x2E, 0x31, 0x0D, 0x0A, 0x48, 0x6F, 0x73, 0x74, 0x3A, 0x20, 0x77, 0x77,
        // ... (HTTP request continues)
    };



    pthread_t thread1, thread2;

    // Create a FrameData structure for each frame
    struct FrameData data1 = {frame1, sizeof(frame1)};
    struct FrameData data2 = {frame2, sizeof(frame2)};
    // ...

    // Create threads to process frames concurrently
    pthread_create(&thread1, NULL, processFrame, (void*)&data1);
    pthread_create(&thread2, NULL, processFrame, (void*)&data2);
    // ...

    // Wait for threads to finish
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    // ...

    return 0;
}

void* processFrame(void* arg) {
    struct FrameData* frameData = (struct FrameData*)arg;
    processEthernetFrame(frameData->frame, frameData->frameLength);
    pthread_exit(NULL);
}

void processEthernetFrame(const uint8_t* frame, uint16_t frameLength) {
    if (frameLength < sizeof(struct EthernetHeader)) {
        printf("Invalid Ethernet frame\n");
        return;
    }

    struct EthernetHeader* ethHeader = (struct EthernetHeader*)frame;

    printf("Ethernet Frame - Source MAC: ");
    printMACAddress(ethHeader->srcMAC);
    printf(", Destination MAC: ");
    printMACAddress(ethHeader->destMAC);
    printf(", EtherType: %04x\n", ntohs(ethHeader->etherType));

    // Assuming Ethernet frame contains IPv4 packet
    if (ntohs(ethHeader->etherType) == 0x0800) {
        processIPv4Packet(frame + sizeof(struct EthernetHeader), frameLength - sizeof(struct
EthernetHeader));
```

```
    } else {
        printf("Unsupported EtherType: %04x\n", ntohs(ethHeader->etherType));
    }
}

void processIPv4Packet(const uint8_t* packet, uint16_t packetLength) {
    if (packetLength < sizeof(struct IPv4Header)) {
        printf("Invalid IPv4 packet\n");
        return;
    }

    struct IPv4Header* ipv4Header = (struct IPv4Header*)packet;
    printf("IPv4 Packet - Source IP: %s, Destination IP: %s\n",
           inet_ntoa(*(struct in_addr*)&ipv4Header->srcIP), inet_ntoa(*(struct in_addr*)&ipv4Header->destIP));

    // Assuming IPv4 packet contains TCP header
    processTCPHeader(packet + (ipv4Header->versionIHL & 0x0F) * 4, packetLength - (ipv4Header-
>versionIHL & 0x0F) * 4);
}

void processTCPHeader(const uint8_t* tcpHeader, uint16_t tcpHeaderLength) {
    if (tcpHeaderLength < sizeof(struct TCPHeader)) {
        printf("Invalid TCP header\n");
        return;
    }

    struct TCPHeader* tcpHeaderStruct = (struct TCPHeader*)tcpHeader;
    printf("TCP Header - Source Port: %u, Destination Port: %u\n",
           ntohs(tcpHeaderStruct->srcPort), ntohs(tcpHeaderStruct->destPort));

    // Assuming TCP header contains application data
    processApplicationData(tcpHeader + (tcpHeaderStruct->dataOffsetReserved >> 4) * 4, tcpHeaderLength -
(tcpHeaderStruct->dataOffsetReserved >> 4) * 4);
}

void processApplicationData(const uint8_t* data, uint16_t dataLength) {
    // Add code to process application-specific data
    printf("Application Data (%u bytes):\n", dataLength);
    for (uint16_t i = 0; i < dataLength; ++i) {
        printf("%c", data[i]);
    }
    printf("\n");
}
```

1.201 Thread Pool

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define THREAD_POOL_SIZE 4

// Structure to hold task information
typedef struct {
    void (*task)(void*); // Function pointer for the task
    void* arg;           // Argument for the task
} Task;
```

```c
// Structure to represent the thread pool
typedef struct {
    pthread_t threads[THREAD_POOL_SIZE]; // Array of threads
    Task taskQueue[THREAD_POOL_SIZE * 2]; // Task queue (adjust size as needed)
    int queueSize;                        // Current size of the task queue
    int front;                            // Front of the queue
    int rear;                             // Rear of the queue
    pthread_mutex_t mutex;                // Mutex for synchronization
    pthread_cond_t condition;             // Condition variable for synchronization
} ThreadPool;

// Function to initialize the thread pool
void initializeThreadPool(ThreadPool* pool) {
    pool->queueSize = 0;
    pool->front = 0;
    pool->rear = 0;
    pthread_mutex_init(&pool->mutex, NULL);
    pthread_cond_init(&pool->condition, NULL);

    // Create threads in the pool
    for (int i = 0; i < THREAD_POOL_SIZE; ++i) {
        pthread_create(&pool->threads[i], NULL, (void* (*)(void*))NULL, NULL);
    }
}

// Function to add a task to the thread pool
void addTask(ThreadPool* pool, void (*task)(void*), void* arg) {
    pthread_mutex_lock(&pool->mutex);

    // Wait if the task queue is full
    while (pool->queueSize == sizeof(pool->taskQueue) / sizeof(pool->taskQueue[0])) {
        pthread_cond_wait(&pool->condition, &pool->mutex);
    }

    // Add the task to the queue
    pool->taskQueue[pool->rear].task = task;
    pool->taskQueue[pool->rear].arg = arg;
    pool->rear = (pool->rear + 1) % (sizeof(pool->taskQueue) / sizeof(pool->taskQueue[0]));
    pool->queueSize++;

    // Signal a waiting thread that a new task is available
    pthread_cond_signal(&pool->condition);

    pthread_mutex_unlock(&pool->mutex);
}

// Function executed by worker threads
void* workerThread(void* arg) {
    ThreadPool* pool = (ThreadPool*)arg;

    for (;;) {
        pthread_mutex_lock(&pool->mutex);

        // Wait if the task queue is empty
        while (pool->queueSize == 0) {
            pthread_cond_wait(&pool->condition, &pool->mutex);
        }
```

```
        // Retrieve a task from the queue
        void (*task)(void*) = pool->taskQueue[pool->front].task;
        void* taskArg = pool->taskQueue[pool->front].arg;
        pool->front = (pool->front + 1) % (sizeof(pool->taskQueue) / sizeof(pool->taskQueue[0]));
        pool->queueSize--;

        // Signal that there is space in the task queue
        pthread_cond_signal(&pool->condition);

        pthread_mutex_unlock(&pool->mutex);

        // Execute the task
        task(taskArg);
    }

    return NULL;
}

// Function to destroy the thread pool
void destroyThreadPool(ThreadPool* pool) {
    // Cancel all threads in the pool
    for (int i = 0; i < THREAD_POOL_SIZE; ++i) {
        pthread_cancel(pool->threads[i]);
    }

    // Destroy the mutex and condition variable
    pthread_mutex_destroy(&pool->mutex);
    pthread_cond_destroy(&pool->condition);
}

// Sample task to be executed by the thread pool
void sampleTask(void* arg) {
    int taskNumber = *(int*)arg;
    printf("Task %d executed by thread %lu\n", taskNumber, pthread_self());
    free(arg); // Free the memory allocated for the argument
}

int main() {
    ThreadPool pool;
    initializeThreadPool(&pool);

    // Submit tasks to the thread pool
    for (int i = 0; i < 10; ++i) {
        int* taskNumber = malloc(sizeof(int)); // Allocate memory for the task argument
        *taskNumber = i;
        addTask(&pool, sampleTask, taskNumber);
    }

    // Sleep to allow the tasks to be executed
    sleep(5);

    // Destroy the thread pool
    destroyThreadPool(&pool);

    return 0;
}
```

1.202 Memory Pool

```
#include <stdio.h>
#include <stdlib.h>

#define PACKET_SIZE 1500  // Adjust the size according to your needs
#define POOL_SIZE 1000    // Adjust the number of packets in the pool

// Structure representing a packet
typedef struct {
    char data[PACKET_SIZE];
    // Add other packet-related fields as needed
} Packet;

// Structure representing a memory pool
typedef struct {
    Packet* packets[POOL_SIZE];
    int nextIndex;  // Index to track the next available packet
} MemoryPool;

// Initialize the memory pool
void initializeMemoryPool(MemoryPool* pool) {
    for (int i = 0; i < POOL_SIZE; ++i) {
        pool->packets[i] = (Packet*)malloc(sizeof(Packet));
        // Initialize other packet-related fields as needed
    }
    pool->nextIndex = 0;
}

// Get a packet from the memory pool
Packet* allocatePacket(MemoryPool* pool) {
    if (pool->nextIndex < POOL_SIZE) {
        return pool->packets[pool->nextIndex++];
    } else {
        // Handle the case when the pool is exhausted (you may want to expand the pool or return NULL)
        printf("Memory pool exhausted!\n");
        return NULL;
    }
}

// Release a packet back to the memory pool
void deallocatePacket(MemoryPool* pool, Packet* packet) {
    // You may want to perform additional cleanup or re-initialization of the packet
    pool->nextIndex--;
}

// Cleanup the memory pool
void cleanupMemoryPool(MemoryPool* pool) {
    for (int i = 0; i < POOL_SIZE; ++i) {
        free(pool->packets[i]);
    }
    pool->nextIndex = 0;
}

// Example of using the memory pool
int main() {
    MemoryPool packetPool;
    initializeMemoryPool(&packetPool);
```

```
    // Allocate and use packets
    Packet* packet1 = allocatePacket(&packetPool);
    // Process packet1...

    Packet* packet2 = allocatePacket(&packetPool);
    // Process packet2...

    // Release packets back to the pool when done
    deallocatePacket(&packetPool, packet1);
    deallocatePacket(&packetPool, packet2);

    // Cleanup the memory pool when the application exits
    cleanupMemoryPool(&packetPool);

    return 0;
}
```

1.203 DPDK

Run to Completion vs Pipeline
Dpdk setup -> compile drivers -> igb pmd - >bind driver to dpdk - >huge pages - >application run
Rte_eal_init (app to use dpdk framework)
Rte_eth_Dev_configure – rx/tx queues , RSS
Rte_Eth_tx_queue_setup – ring descriptors
Rte_eth_rx_queue_setup
Rte_eth_dev_Start – RX/TX packets
Rte_eth_rx_burst/tx_burst (port id.q no, array of structure to hold packets, number)
DPDK (Data Plane Development Kit) is a set of libraries and drivers for fast packet processing in user space. It is designed to optimize packet processing workloads and achieve high-performance data plane applications. DPDK allows developers to build efficient and scalable network applications by bypassing the traditional kernel networking stack and accessing network interfaces directly from user space.

Key features and components of DPDK include:

- Processor affinity (separate cores)
- Huge pages (no swap, TLB)
- UIO (no copying from kernel)
- Polling (no interrupts overhead)
- Lockless synchronization (avoid waiting)
- Batch packets handling
- SSE, NUMA awareness

Poll Mode Drivers (PMDs):

DPDK provides a set of Poll Mode Drivers (PMDs) for various network interface controllers (NICs). PMDs allow applications to directly interact with network interfaces, eliminating the need for kernel involvement in the data plane.

Memory Management:

DPDK includes a memory management subsystem that provides a high-performance memory allocation and deallocation framework.
It uses a custom memory allocator called the DPDK Memory Manager, which helps reduce memory fragmentation and overhead.
Poll Mode Operation:

DPDK applications operate in a poll mode, continuously polling the NICs and processing packets without relying on interrupts.
Poll mode operation reduces the overhead of interrupt processing and context switching, resulting in lower latency.
Packet Buffer Management:

DPDK includes a memory pool manager to efficiently manage packet buffers.
The memory pool manager allocates fixed-sized memory blocks, or mbufs, to store packet data. This improves cache locality and reduces memory fragmentation.
RTE (Run-Time Environment):

DPDK provides a set of APIs and abstractions collectively known as the RTE (Run-Time Environment).
RTE includes functions for memory management, I/O operations, and other utilities that facilitate the development of high-performance applications.
Multi-Queue Support:

DPDK supports multi-queue operation, allowing applications to distribute packet processing across multiple CPU cores.
This helps achieve parallelism and scalability in packet processing workloads.
Packet Classification and Filtering:

DPDK provides mechanisms for packet classification, filtering, and flow steering.
Applications can define rules for classifying and directing packets based on criteria such as MAC addresses, IP addresses, and ports.
Event Mode:

DPDK includes an Event Mode framework that allows applications to handle events asynchronously.
The Event Mode enables event-driven architectures for handling various tasks, including packet processing.
Crypto and Compression Operations:

DPDK includes libraries for hardware-accelerated cryptographic and compression operations.
Applications can leverage these libraries for secure and efficient processing of encrypted and compressed data.
Unified Packet Framework:

DPDK provides a unified packet framework that abstracts packet data structures and operations, making it easier for developers to work with packets in a consistent manner.
Dynamic Device Configuration:

DPDK allows applications to dynamically configure and manage network devices, including adding or removing devices during runtime.
DPDK EAL (Environment Abstraction Layer):

The Environment Abstraction Layer provides an abstraction over system-specific details and allows DPDK applications to run on various operating systems.
DPDK is commonly used in the development of network functions, such as routers, firewalls, load balancers, and other data plane applications requiring low-latency, high-throughput packet processing. It is widely adopted in the telecommunications and networking industry for building efficient and scalable network infrastructure.

```
#include <stdio.h>
```

```
#include <stdint.h>
#include <inttypes.h>
#include <rte_eal.h>
#include <rte_ethdev.h>
#include <rte_lcore.h>
#include <rte_mbuf.h>

#define RX_RING_SIZE 1024
#define TX_RING_SIZE 1024
#define NUM_MBUFS 8191
#define MBUF_CACHE_SIZE 250
#define BURST_SIZE 32

static const struct rte_eth_conf port_conf_default = {
    .rxmode = {
        .max_rx_pkt_len = RTE_ETHER_MAX_LEN,
    },
};

static struct rte_eth_dev_tx_buffer *tx_buffer[RTE_MAX_ETHPORTS];
static uint16_t nb_ports;

static int init_port(uint16_t port, struct rte_mempool *mbuf_pool) {
    struct rte_eth_conf port_conf = port_conf_default;
    const uint16_t rx_rings = 1, tx_rings = 1;
    int retval;
    uint16_t q;

    if (port >= rte_eth_dev_count_avail()) {
        printf("Error: Port %" PRIu16 " not available\n", port);
        return -1;
    }

    retval = rte_eth_dev_configure(port, rx_rings, tx_rings, &port_conf);
    if (retval != 0) {
        printf("Error during port configuration: %s\n", rte_strerror(retval));
        return retval;
    }

    retval = rte_eth_dev_adjust_nb_rx_tx_desc(port, &RX_RING_SIZE, &TX_RING_SIZE);
    if (retval != 0) {
        printf("Error during adjusting number of descriptors: %s\n", rte_strerror(retval));
        return retval;
    }

    for (q = 0; q < rx_rings; q++) {
        retval = rte_eth_rx_queue_setup(port, q, RX_RING_SIZE,
                                        rte_eth_dev_socket_id(port), NULL, mbuf_pool);
        if (retval < 0) {
            printf("Error during RX queue setup: %s\n", rte_strerror(-retval));
            return retval;
        }
    }

    for (q = 0; q < tx_rings; q++) {
        retval = rte_eth_tx_queue_setup(port, q, TX_RING_SIZE,
                                        rte_eth_dev_socket_id(port), NULL);
        if (retval < 0) {
```

```
            printf("Error during TX queue setup: %s\n", rte_strerror(-retval));
            return retval;
        }
        tx_buffer[port] = rte_zmalloc_socket("tx_buffer",
                                             RTE_ETH_TX_BUFFER_SIZE(BURST_SIZE),
                                             0, rte_eth_dev_socket_id(port));
        if (tx_buffer[port] == NULL) {
            printf("Error: cannot allocate buffer for tx on port %u\n", port);
            return -1;
        }
        rte_eth_tx_buffer_init(tx_buffer[port], BURST_SIZE);
    }

    retval = rte_eth_dev_start(port);
    if (retval < 0) {
        printf("Error during port start: %s\n", rte_strerror(-retval));
        return retval;
    }

    rte_eth_promiscuous_enable(port);

    return 0;
}

static void dpdk_packet_receive(uint16_t port) {
    struct rte_mbuf *bufs[BURST_SIZE];
    uint16_t nb_rx;

    printf("Core %u receiving packets. [Ctrl+C to quit]\n", rte_lcore_id());

    while (1) {
        nb_rx = rte_eth_rx_burst(port, 0, bufs, BURST_SIZE);
        if (nb_rx > 0) {
            // Process received packets (can be done on a different core)
            // For simplicity, just free the received packets
            rte_pktmbuf_free_bulk(bufs, nb_rx);
        }
    }
}

static void dpdk_packet_process(uint16_t port) {
    struct rte_mbuf *bufs[BURST_SIZE];
    uint16_t nb_rx, nb_tx;
    uint16_t dst_port = (port + 1) % nb_ports;

    printf("Core %u processing packets. [Ctrl+C to quit]\n", rte_lcore_id());

    while (1) {
        nb_rx = rte_eth_rx_burst(port, 0, bufs, BURST_SIZE);
        if (nb_rx > 0) {
            // Process received packets
            // For simplicity, just transmit the received packets to another port
            nb_tx = rte_eth_tx_buffer(dst_port, 0, tx_buffer[dst_port], bufs, nb_rx);
            if (nb_tx < nb_rx) {
                rte_pktmbuf_free_bulk(bufs + nb_tx, nb_rx - nb_tx);
            }
        }
    }
```

```
}

int main(int argc, char *argv[]) {
    struct rte_mempool *mbuf_pool;
    uint16_t portid;
    unsigned lcore_id;

    if (rte_eal_init(argc, argv) < 0) {
        rte_panic("Cannot init EAL\n");
        return -1;
    }

    argc -= rte_eal_init(argc, argv);
    argv += rte_eal_init(argc, argv);

    nb_ports = rte_eth_dev_count_avail();
    if (nb_ports == 0) {
        rte_panic("No Ethernet ports found\n");
        return -1;
    }

    mbuf_pool = rte_pktmbuf_pool_create("MBUF_POOL", NUM_MBUFS * nb_ports,
                                    MBUF_CACHE_SIZE, 0, RTE_MBUF_DEFAULT_BUF_SIZE,
                                    rte_socket_id());
    if (mbuf_pool == NULL) {
        rte_panic("Cannot create mbuf pool\n");
        return -1;
    }

    RTE_ETH_FOREACH_DEV(portid) {
        if (init_port(portid, mbuf_pool) != 0)
            return -1;
    }

    RTE_LCORE_FOREACH_SLAVE(lcore_id) {
        rte_eal_remote_launch((lcore_function_t *)dpdk_packet_process, NULL, lcore_id);
    }

    // Run packet receiving on the master core
    dpdk_packet_receive(0);

    rte_eal_mp_wait_lcore();

    return 0;
}
```