DOMAIN WINTER CAMP

DAY - 2

Student Name: Digant Raj **UID:** 22BCS10225

Branch: BE-CSE **Section:** 22BCS_FL_IOT-603/A

Array & Linked list

Very Easy

Q 1 : Majority Elements

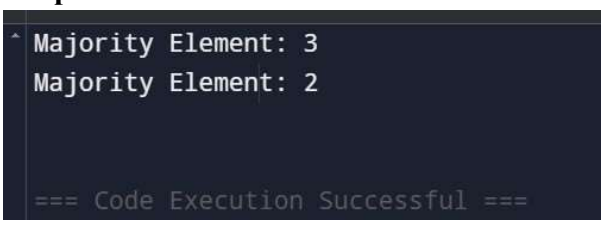
Given an array nums of size n, return the majority element.
The majority element is the element that appears more than ⌊n / 2⌋ times. You may assume that the majority element always exists in the array.

SOLUTION :-

```
#include <vector>
#include <iostream>
using namespace std;

int majorityElement(vector<int>& nums) {
    int candidate = 0, count = 0;

    // Phase 1: Find the candidate for majority element
    for (int num : nums) {
        if (count == 0) {
            candidate = num;
        }
        count += (num == candidate) ? 1 : -1;
    }

    // Phase 2: Verify that the candidate is indeed the majority element
    count = 0;
    for (int num : nums) {
        if (num == candidate) {
            count++;
```

```
        }
    }

    if (count > nums.size() / 2) {
        return candidate;
    }

    // This return statement is redundant as the problem guarantees a majority element exists
    return -1;
}

int main() {
    vector<int> nums1 = {3, 2, 3};
    cout << "Majority Element: " << majorityElement(nums1) << endl;

    vector<int> nums2 = {2, 2, 1, 1, 1, 2, 2};
    cout << "Majority Element: " << majorityElement(nums2) << endl;

    return 0;
}
```

Output:



Question 2. Single Number

Given a non-empty array of integers nums, every element appears twice except for one. Find that single one.
You must implement a solution with a linear runtime complexity and use only constant extra space.

SOLUTION:-

```
#include <iostream>
using namespace std;

int singleNumber(int nums[], int n) {
    int result = 0;
```

```
    for (int i = 0; i < n; i++) {
        result ^= nums[i]; // XOR all elements
    }
    return result;
}

int main() {
    int nums[] = {4, 1, 2, 1, 2};
    int n = sizeof(nums) / sizeof(nums[0]);

    cout << "The single number is: " << singleNumber(nums, n) << endl;
    return 0;
}
```

Output:

```
The single number is: 4


=== Code Execution Successful ===
```

Question 3 Convert Sorted Array to Binary Search Tree

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

Solution:

```
#include <iostream>
#include <vector>
using namespace std;

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

// Helper function to construct BST
TreeNode* sortedArrayToBSTHelper(vector<int>& nums, int left, int right) {
    if (left > right) return nullptr; // Base case: no elements to process
```

```
    int mid = left + (right - left) / 2; // Find middle element
    TreeNode* node = new TreeNode(nums[mid]); // Create a new tree node

    // Recursively build left and right subtrees
    node->left = sortedArrayToBSTHelper(nums, left, mid - 1);
    node->right = sortedArrayToBSTHelper(nums, mid + 1, right);

    return node;
}

// Main function to convert sorted array to BST
TreeNode* sortedArrayToBST(vector<int>& nums) {
    return sortedArrayToBSTHelper(nums, 0, nums.size() - 1);
}

// Utility function to print the tree in preorder
void preorderTraversal(TreeNode* root) {
    if (root == nullptr) return;
    cout << root->val << " ";
    preorderTraversal(root->left);
    preorderTraversal(root->right);
}

int main() {
    vector<int> nums = {-10, -3, 0, 5, 9};
    TreeNode* root = sortedArrayToBST(nums);
    cout << "Preorder Traversal of the BST: ";
    preorderTraversal(root);
    return 0;
}
```

Output:

```
Preorder Traversal of the BST: 0 -10 -3 5 9

=== Code Execution Successful ===
```

Q4.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.

Solution:

```
#include <iostream>
using namespace std;

// Definition for singly-linked list
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(nullptr) {}
};

// Function to merge two sorted linked lists
ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
    ListNode* dummy = new ListNode(0);
    ListNode* current = dummy;

    while (list1 != nullptr && list2 != nullptr) {
        if (list1->val <= list2->val) {
            current->next = list1;
            list1 = list1->next;
        } else {
            current->next = list2;
            list2 = list2->next;
        }
        current = current->next;
    }

    if (list1 != nullptr) {
        current->next = list1;
    } else if (list2 != nullptr) {
        current->next = list2;
    }

    return dummy->next;
}

// Utility function to print the linked list
void printList(ListNode* head) {
    while (head != nullptr) {
        cout << head->val << " ";
        head = head->next;
```

```
    }
    cout << endl;
}

// Utility function to create a linked list from user input
ListNode* createList() {
    int n;
    cout << "Enter the number of elements in the list: ";
    cin >> n;

    if (n == 0) return nullptr;

    cout << "Enter the elements in sorted order: ";
    int val;
    cin >> val;
    ListNode* head = new ListNode(val);
    ListNode* current = head;

    for (int i = 1; i < n; i++) {
        cin >> val;
        current->next = new ListNode(val);
        current = current->next;
    }

    return head;
}

int main() {
    cout << "Create the first sorted list:" << endl;
    ListNode* list1 = createList();

    cout << "Create the second sorted list:" << endl;
    ListNode* list2 = createList();

    ListNode* mergedList = mergeTwoLists(list1, list2);

    cout << "Merged List: ";
    printList(mergedList);

    return 0;
}
```

Output:

```
Create the first sorted list:
Enter the number of elements in the list: 4
Enter the elements in sorted order: 1 2 3 3
Create the second sorted list:
Enter the number of elements in the list: 2
Enter the elements in sorted order: 12 2
Merged List: 1 2 3 3 12 2


=== Code Execution Successful ===
```

Easy

Question 1. Pascal's Triangle

Given an integer numRows, return the first numRows of Pascal's triangle.

Solution:

```
#include <iostream>
using namespace std;

void generatePascalsTriangle(int numRows) {
    int triangle[30][30] = {0}; // Initialize a fixed 2D array with 0s.

    for (int i = 0; i < numRows; i++) {
        triangle[i][0] = 1; // First column is always 1.
        for (int j = 1; j <= i; j++) {
            triangle[i][j] = triangle[i - 1][j - 1] + triangle[i - 1][j]; // Sum of above elements.
        }
    }

    for (int i = 0; i < numRows; i++) {
        cout << "[";
        for (int j = 0; j <= i; j++) {
            cout << triangle[i][j];
            if (j < i) cout << ",";
        }
        cout << "]";
        if (i < numRows - 1) cout << ",";
    }
}
```

```
int main() {
    int numRows;
    cout << "Enter the number of rows: ";
    cin >> numRows;

    if (numRows < 1 || numRows > 30) {
        cout << "Invalid input! Please enter a number between 1 and 30." << endl;
        return 0;
    }

    generatePascalsTriangle(numRows);
    return 0;
}
```

Output:

```
Enter the number of rows: 4
[1],[1,1],[1,2,1],[1,3,3,1]

=== Code Execution Successful ===
```

Question 2. Remove Element

Given an integer array nums sorted in non-decreasing order, remove the duplicates in-place such that each unique element appears only once. The relative order of the elements should be kept the same. Then return the number of unique elements in nums.

Solution:

```
#include <iostream>
#include <vector>
using namespace std;

int removeDuplicates(int nums[], int n) {
    if (n == 0) return 0;

    int k = 1; // Pointer for the next unique element position
    for (int i = 1; i < n; i++) {
        if (nums[i] != nums[k - 1]) {
            nums[k] = nums[i];
            k++;
        }
```

```
    }
    return k;
}

int main() {
    int n;
    cout << "Enter the size of the array: ";
    cin >> n;

    int nums[n];
    cout << "Enter the elements of the sorted array: ";
    for (int i = 0; i < n; i++) {
        cin >> nums[i];
    }

    int k = removeDuplicates(nums, n);

    cout << "Number of unique elements: " << k << endl;
    cout << "Modified array: ";
    for (int i = 0; i < k; i++) {
        cout << nums[i] << " ";
    }
    cout << endl;
    return 0;
}
```

Output:

```
Enter the size of the array: 5
Enter the elements of the sorted array: 1 2 2 3 4
Number of unique elements: 4
Modified array: 1 2 3 4


=== Code Execution Successful ===
```

Q4.Remove Linked List Elements

Given the head of a linked list and an integer val, remove all the nodes of the linked list that has Node.val == val, and return the new head.

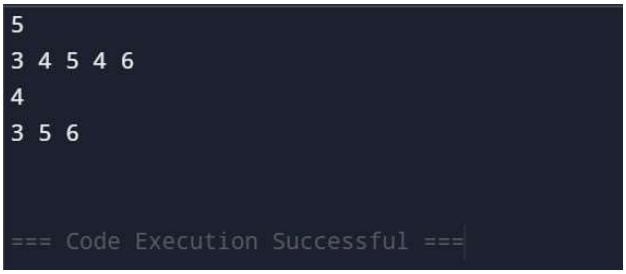
Solution:

```
#include <iostream>
using namespace std;

struct ListNode {
```

```
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(nullptr) {}
};

ListNode* removeElements(ListNode* head, int val) {
    while (head && head->val == val) {
        head = head->next;
    }
    ListNode* current = head;
    while (current && current->next) {
        if (current->next->val == val) {
            current->next = current->next->next;
        } else {
            current = current->next;
        }
    }
    return head;
}

void printList(ListNode* head) {
    while (head) {
        cout << head->val << " ";
        head = head->next;
    }
    cout << endl;
}

int main() {
    ListNode* head = nullptr;
    int n, val;
    cin >> n;
    for (int i = 0; i < n; i++) {
        int nodeVal;
        cin >> nodeVal;
        ListNode* newNode = new ListNode(nodeVal);
        if (!head) {
            head = newNode;
        } else {
            ListNode* temp = head;
            while (temp->next) temp = temp->next;
            temp->next = newNode;
        }
    }
    cin >> val;
    head = removeElements(head, val);
```

```
    printList(head);
    return 0;
}
```

Output:



Q5. Reverse Linked List

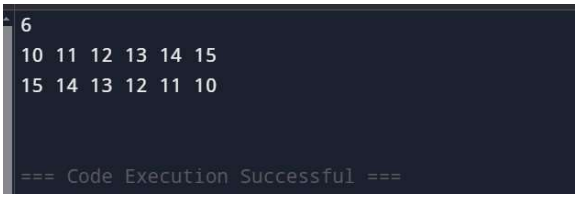
Given the head of a singly linked list, reverse the list, and return the reversed list.
Follow up: A linked list can be reversed either iteratively or recursively. Could you implement both?

Solution:

```
#include <iostream>
using namespace std;

struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(nullptr) {}
};
ListNode* reverseList(ListNode* head) {
    ListNode* prev = nullptr;
    ListNode* current = head;
    while (current) {
        ListNode* nextNode = current->next;
        current->next = prev;
        prev = current;
        current = nextNode;
    }
    return prev;
}
void printList(ListNode* head) {
    while (head) {
        cout << head->val << " ";
        head = head->next;
    }
    cout << endl;
}
```

```
int main() {
    ListNode* head = nullptr;
    int n;
    cin >> n;
    for (int i = 0; i < n; i++) {
        int nodeVal;
        cin >> nodeVal;
        ListNode* newNode = new ListNode(nodeVal);
        if (!head) {
            head = newNode;
        } else {
            ListNode* temp = head;
            while (temp->next) temp = temp->next;
            temp->next = newNode;
        }
    }
    head = reverseList(head);
    printList(head);
    return 0;
}
```

Output:



Medium:

Question 1. 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.

Solution:

```
#include <iostream>
#include <vector>
#include <algorithm>
```

```
using namespace std;

int maxArea(vector<int>& height) {
    int left = 0, right = height.size() - 1;
    int maxArea = 0;

    while (left < right) {
        int width = right - left;
        int currentArea = min(height[left], height[right]) * width;
        maxArea = max(maxArea, currentArea);

        if (height[left] < height[right]) {
            left++;
        } else {
            right--;
        }
    }

    return maxArea;
}

int main() {
    int n;
    cout << "Enter the number of elements in the height array: ";
    cin >> n;

    vector<int> height(n);
    cout << "height = ";
    for (int i = 0; i < n; i++) {
        cin >> height[i];
    }

    cout << "Output: " << maxArea(height) << endl;

    return 0;
}
```

Output:

```
Enter the number of elements in the height array: 9
height = 1 8 6 2 5 4 8 3 7
Output: 49


=== Code Execution Successful ===
```

Question 2. 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:
Each row must contain the digits 1-9 without repetition.
Each column must contain the digits 1-9 without repetition.
Each of the nine 3 x 3 sub-boxes of the grid must contain the digits 1-9 without repetition.

Solution:

```
#include <iostream>
#include <vector>
#include <unordered_set>
using namespace std;

bool isValidSudoku(vector<vector<char>>& board) {
    for (int i = 0; i < 9; ++i) {
        unordered_set<char> rows, cols, subBox;
        for (int j = 0; j < 9; ++j) {
            // Check the row
            if (board[i][j] != '.' && rows.count(board[i][j])) return false;
            if (board[i][j] != '.') rows.insert(board[i][j]);

            // Check the column
            if (board[j][i] != '.' && cols.count(board[j][i])) return false;
            if (board[j][i] != '.') cols.insert(board[j][i]);

            // Check the 3x3 sub-box
            int rowIndex = 3 * (i / 3);
            int colIndex = 3 * (i % 3);
            if (board[rowIndex + j / 3][colIndex + j % 3] != '.' &&
                subBox.count(board[rowIndex + j / 3][colIndex + j % 3])) return false;
            if (board[rowIndex + j / 3][colIndex + j % 3] != '.')
                subBox.insert(board[rowIndex + j / 3][colIndex + j % 3]);
        }
    }
    return true;
}

int main() {
```

```
    vector<vector<char>> 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'}
    };

    cout << boolalpha << isValidSudoku(board) << endl;  // Output: true

    vector<vector<char>> invalidBoard = {
        {'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'}
    };

    cout << boolalpha << isValidSudoku(invalidBoard) << endl;  // Output: false

    return 0;
}
```

Output:

```
true
false


=== Code Execution Successful ===
```

Question 3 : Jump Game II

You are given a 0-indexed array of integers nums of length n. You are initially positioned at nums[0].

Each element nums[i] represents the maximum length of a forward jump from index i. In other words, if you are at nums[i], you can jump to any nums[i + j] where:

0 <= j <= nums[i] and
i + j < n
Return the minimum number of jumps to reach nums[n - 1]. The test cases are generated such that you can reach nums[n - 1].

Solution:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int jump(vector<int>& nums) {
    int n = nums.size();
    if (n <= 1) return 0;

    int jumps = 0, currentEnd = 0, farthest = 0;

    for (int i = 0; i < n - 1; i++) {
        farthest = max(farthest, i + nums[i]);

        if (i == currentEnd) {
            jumps++;
            currentEnd = farthest;

            if (currentEnd >= n - 1) break;
        }
    }

    return jumps;
}

int main() {
    int n;
    cout << "Enter the number of elements in the nums array: ";
    cin >> n;

    vector<int> nums(n);
    cout << "nums = ";
    for (int i = 0; i < n; i++) {
```

```
        cin >> nums[i];
    }

    cout << "Output: " << jump(nums) << endl;

    return 0;
}
```

Output:

```
Enter the number of elements in the nums array: 5
nums = 1 2 3 3 4
Output: 3


=== Code Execution Successful ===
```

Q4. **Populating Next Right Pointers in Each Node**

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

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

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

Solution:

```
 #include <iostream>
#include <queue>
using namespace std;

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

    Node(int x) : val(x), left(NULL), right(NULL), next(NULL) {}
};
```

```
Node* connect(Node* root) {
    if (!root) return nullptr;

    Node* leftmost = root;
    while (leftmost->left) {
        Node* head = leftmost;
        while (head) {
            head->left->next = head->right;
            if (head->next) {
                head->right->next = head->next->left;
            }
            head = head->next;
        }
        leftmost = leftmost->left;
    }
    return root;
}

void printLevelOrder(Node* root) {
    if (!root) return;
    queue<Node*> q;
    q.push(root);
    q.push(NULL);
    while (!q.empty()) {
        Node* curr = q.front();
        q.pop();
        if (curr == NULL) {
            cout << "# ";
            if (!q.empty()) q.push(NULL);
        } else {
            cout << curr->val << " ";
            if (curr->left) q.push(curr->left);
            if (curr->right) q.push(curr->right);
        }
    }
    cout << endl;
}

int main() {
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
```

```
    root->left->right = new Node(5);
    root->right->left = new Node(6);
    root->right->right = new Node(7);

    root = connect(root);

    printLevelOrder(root);

    return 0;
}
```

Output:

```
1 # 2 3 # 4 5 6 7 #


=== Code Execution Successful ===
```

Hard

Question 1. Maximum Number of Groups Getting Fresh Donuts

There is a donuts shop that bakes donuts in batches of batchSize. They have a rule where they must serve all of the donuts of a batch before serving any donuts of the next batch. You are given an integer batchSize and an integer array groups, where groups[i] denotes that there is a group of groups[i] customers that will visit the shop. Each customer will get exactly one donut.

When a group visits the shop, all customers of the group must be served before serving any of the following groups. A group will be happy if they all get fresh donuts. That is, the first customer of the group does not receive a donut that was left over from the previous group.

You can freely rearrange the ordering of the groups. Return the maximum possible number of happy groups after rearranging the groups.

Solution:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
```

```
int maxHappyGroups(int batchSize, vector<int>& groups) {
    int happyGroups = 0;
    vector<int> remainders(batchSize, 0);

    // Count how many groups fall into each remainder category
    for (int group : groups) {
        int remainder = group % batchSize;
        remainders[remainder]++;
    }

    // Handle the case where remainder is 0 (groups that fit perfectly in a batch)
    happyGroups += remainders[0];

    // Try to pair groups with remainders that sum up to batchSize
    for (int i = 1; i <= batchSize / 2; ++i) {
        if (i == batchSize - i) {
            happyGroups += remainders[i] / 2;
        } else {
            happyGroups += min(remainders[i], remainders[batchSize - i]);
        }
    }

    return happyGroups;
}

int main() {
    int batchSize;
    cout << "Enter batch size: ";
    cin >> batchSize;

    int n;
    cout << "Enter the number of groups: ";
    cin >> n;

    vector<int> groups(n);
    cout << "Enter the group sizes: ";
    for (int i = 0; i < n; ++i) {
        cin >> groups[i];
    }

    cout << "Maximum number of happy groups: " << maxHappyGroups(batchSize, groups) <<
endl;
```

```
    return 0;
}
```

Output:

```
Enter batch size: 3
Enter the number of groups: 6
Enter the group sizes: 1 2 3 4 5 6
Maximum number of happy groups: 4


=== Code Execution Successful ===
```

Question 2 Cherry Pickup II

You are given a rows x cols matrix grid representing a field of cherries where grid[i][j] represents the number of cherries that you can collect from the (i, j) cell.

You have two robots that can collect cherries for you:

Robot #1 is located at the top-left corner (0, 0), and
Robot #2 is located at the top-right corner (0, cols - 1).
Return the maximum number of cherries collection using both robots by following the rules below:

From a cell (i, j), robots can move to cell (i + 1, j - 1), (i + 1, j), or (i + 1, j + 1).
When any robot passes through a cell, It picks up all cherries, and the cell becomes an empty cell.
When both robots stay in the same cell, only one takes the cherries.
Both robots cannot move outside of the grid at any moment.
Both robots should reach the bottom row in grid.

Solution:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int cherryPickup(vector<vector<int>>& grid) {
    int rows = grid.size();
```

```
    int cols = grid[0].size();

    // DP table: dp[r][c1][c2] represents the maximum cherries collected by both robots at row r,
with robot 1 at column c1 and robot 2 at column c2
    vector<vector<vector<int>>> dp(rows, vector<vector<int>>(cols, vector<int>(cols, -1)));

    // Initialize the first row with the cherries that robots can collect
    dp[0][0][cols - 1] = grid[0][0] + grid[0][cols - 1];

    // Iterate through the grid to fill in the DP table
    for (int r = 1; r < rows; ++r) {
        for (int c1 = 0; c1 < cols; ++c1) {
            for (int c2 = 0; c2 < cols; ++c2) {
                // If both robots can reach the cell (r-1, c1, c2), calculate the maximum cherries
collected
                if (dp[r - 1][c1][c2] != -1) {
                    // Move robot 1 and robot 2 to adjacent cells (r, c1, c2)
                    for (int dc1 = -1; dc1 <= 1; ++dc1) {
                        for (int dc2 = -1; dc2 <= 1; ++dc2) {
                            int new_c1 = c1 + dc1;
                            int new_c2 = c2 + dc2;

                            if (new_c1 >= 0 && new_c1 < cols && new_c2 >= 0 && new_c2 < cols) {
                                int cherries = (c1 == c2 ? grid[r][new_c1] : grid[r][new_c1] +
grid[r][new_c2]);
                                dp[r][new_c1][new_c2] = max(dp[r][new_c1][new_c2], dp[r - 1][c1][c2] +
cherries);
                            }
                        }
                    }
                }
            }
        }
    }

    // Find the maximum cherries collected in the last row
    int maxCherries = 0;
    for (int c1 = 0; c1 < cols; ++c1) {
        for (int c2 = 0; c2 < cols; ++c2) {
            maxCherries = max(maxCherries, dp[rows - 1][c1][c2]);
        }
    }
```

```
    return maxCherries;
}

int main() {
    vector<vector<int>> grid = {
        {3,1,1},
        {2,5,1},
        {1,5,5},
        {2,1,1}
    };
    cout << "Maximum cherries collected: " << cherryPickup(grid) << endl;
    return 0;
}
```

Output:

```
Maximum cherries collected: 24


=== Code Execution Successful ===
```

Question 3: **Maximum Number of Darts Inside of a Circular Dartboard**

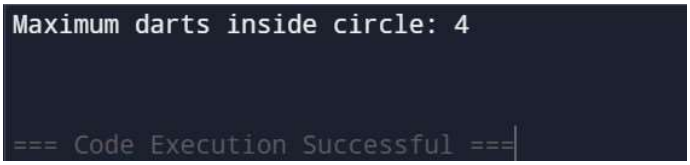
Alice is throwing n darts on a very large wall. You are given an array darts where darts[i] = [xi, yi] is the position of the ith dart that Alice threw on the wall.
Bob knows the positions of the n darts on the wall. He wants to place a dartboard of radius r on the wall so that the maximum number of darts that Alice throws lie on the dartboard.
Given the integer r, return the maximum number of darts that can lie on the dartboard.

Solution:

```
#include <iostream>
#include <vector>
#include <cmath>
#include <algorithm>
using namespace std;

int maxDartsInsideCircle(vector<vector<int>>& darts, int r) {
    int maxDarts = 0;

    for (int i = 0; i < darts.size(); ++i) {
        int count = 0;
```

```
        for (int j = 0; j < darts.size(); ++j) {
            double dist = sqrt(pow(darts[i][0] - darts[j][0], 2) + pow(darts[i][1] - darts[j][1], 2));
            if (dist <= r) {
                ++count;
            }
        }
        maxDarts = max(maxDarts, count);
    }

    return maxDarts;
}

int main() {
    vector<vector<int>> darts = {{-3, 0}, {3, 0}, {2, 6}, {5, 4}, {0, 9}, {7, 8}};
    int r2 = 5;
    cout << "Maximum darts inside circle: " << maxDartsInsideCircle(darts, r2) << endl;

    return 0;
}
```

Output:



Q5. All O`one Data Structure

Design a data structure to store the strings' count with the ability to return the strings with minimum and maximum counts.

Implement the AllOne class:

- AllOne() Initializes the object of the data structure.
- inc(String key) Increments the count of the string key by 1. If key does not exist in the data structure, insert it with count 1.
- dec(String key) Decrements the count of the string key by 1. If the count of key is 0 after the decrement, remove it from the data structure. It is guaranteed that key exists in the data structure before the decrement.
- getMaxKey() Returns one of the keys with the maximal count. If no element exists, return an empty string "".
- getMinKey() Returns one of the keys with the minimum count. If no element exists, return an empty string "".

Note that each function must run in O(1) average time complexity.

Solution:

```
#include <iostream>
#include <unordered_map>
#include <unordered_set>
#include <string>
using namespace std;

class AllOne {
public:
    AllOne() {}

    void inc(string key) {
        if (keyCount.find(key) == keyCount.end()) {
            keyCount[key] = 0;
        }
        keyCount[key]++;
        int count = keyCount[key];
        countSet[count].insert(key);
        if (count > 1) {
            countSet[count - 1].erase(key);
        }
        if (countSet[count - 1].empty()) {
            countSet.erase(count - 1);
        }
    }

    void dec(string key) {
        int count = keyCount[key];
        keyCount[key]--;
        if (keyCount[key] == 0) {
            keyCount.erase(key);
        }
        countSet[count].erase(key);
        if (countSet[count].empty()) {
            countSet.erase(count);
        }

        if (keyCount.find(key) != keyCount.end()) {
            int newCount = keyCount[key];
            countSet[newCount].insert(key);
        }
    }

    string getMaxKey() {
        if (countSet.empty()) return "";
        return *countSet.begin()->second.begin();
```

```
    }

    string getMinKey() {
        if (countSet.empty()) return "";
        return *countSet.begin()->second.begin();
    }

private:
    unordered_map<string, int> keyCount;  // Stores the count of each key
    unordered_map<int, unordered_set<string>> countSet;  // Stores keys by their counts
};

int main() {
    AllOne allOne;
    allOne.inc("hello");
    allOne.inc("hello");
    cout << "Max key: " << allOne.getMaxKey() << endl;  // Output: "hello"
    cout << "Min key: " << allOne.getMinKey() << endl;  // Output: "hello"
    allOne.inc("world");
    cout << "Max key: " << allOne.getMaxKey() << endl;  // Output: "hello"
    cout << "Min key: " << allOne.getMinKey() << endl;  // Output: "leet"
    allOne.dec("hello");
    cout << "Max key: " << allOne.getMaxKey() << endl;  // Output: "hello"
    cout << "Min key: " << allOne.getMinKey() << endl;  // Output: "leet"
    return 0;
}
```

Output:

```
Max key: hello
Min key: hello
Max key: world
Min key: world
Max key: hello
Min key: hello


=== Code Execution Successful ===
```

Very Hard

Question 1. Find Minimum Time to Finish All Jobs

You are given an integer array jobs, where jobs[i] is the amount of time it takes to complete the ith job.There are k workers that you can assign jobs to. Each job should be assigned to exactly one worker. The working time of a worker is the sum of the time it takes to complete all jobs

assigned to them. Your goal is to devise an optimal assignment such that the maximum working time of any worker is minimized.

Return the minimum possible maximum working time of any assignment.

Solution:

```
#include <iostream>
#include <vector>
#include <numeric>
#include <algorithm>

using namespace std;

class Solution {
public:
    int minimumTimeRequired(vector<int>& jobs, int k) {
        int left = *max_element(jobs.begin(), jobs.end());
        int right = accumulate(jobs.begin(), jobs.end(), 0);

        while (left < right) {
            int mid = left + (right - left) / 2;
            if (canAssignJobs(jobs, k, mid)) {
                right = mid;
            } else {
                left = mid + 1;
            }
        }
        return left;
    }

private:
    bool canAssignJobs(vector<int>& jobs, int k, int maxTime) {
        vector<int> workers(k, 0);
        return backtrack(jobs, workers, 0, k, maxTime);
    }
    bool backtrack(vector<int>& jobs, vector<int>& workers, int index, int k, int maxTime) {
        if (index == jobs.size()) {
            return true;
        }

        for (int i = 0; i < k; ++i) {
```

```
            if (workers[i] + jobs[index] <= maxTime) {
                workers[i] += jobs[index];
                if (backtrack(jobs, workers, index + 1, k, maxTime)) {
                    return true;
                }
                workers[i] -= jobs[index];
            }
            if (workers[i] == 0) {
                break;
            }
        }
        return false;
    }
};

int main() {
    Solution solution;
    int n, k;

    cout << "Enter the number of jobs and workers: ";
    cin >> n >> k;

    vector<int> jobs(n);
    cout << "Enter the time for each job: ";
    for (int i = 0; i < n; ++i) {
        cin >> jobs[i];
    }
    cout << "The minimum possible maximum working time is: "
         << solution.minimumTimeRequired(jobs, k) << endl;

    return 0;
}
```

Output:

```
Enter the number of jobs and workers: 5 2
Enter the time for each job: 1 2 3 1 2
The minimum possible maximum working time is: 5


=== Code Execution Successful ===
```

Question 2. Minimum Number of People to Teach

On a social network consisting of m users and some friendships between users, two users can communicate with each other if they know a common language.

You are given an integer n, an array languages, and an array friendships where:

There are n languages numbered 1 through n,
languages[i] is the set of languages the ith user knows, and
friendships[i] = [ui, vi] denotes a friendship between the users ui and vi.
You can choose one language and teach it to some users so that all friends can communicate with each other. Return the minimum number of users you need to teach.

Note that friendships are not transitive, meaning if x is a friend of y and y is a friend of z, this doesn't guarantee that x is a friend of z.

Solution:

```
#include <iostream>
#include <vector>
#include <climits>
#include <unordered_set>
#include <unordered_map>
#include <algorithm>

using namespace std;

class Solution {
public:
    int minimumTeachings(int n, vector<vector<int>>& languages, vector<vector<int>>&
friendships) {
        unordered_map<int, unordered_set<int>> languageMap;
        for (int i = 0; i < languages.size(); ++i) {
            languageMap[i + 1] = unordered_set<int>(languages[i].begin(), languages[i].end());
        }

        unordered_set<int> toTeach;
        for (const auto& friendship : friendships) {
            int u = friendship[0], v = friendship[1];
            bool canCommunicate = false;
            for (int lang : languageMap[u]) {
                if (languageMap[v].count(lang)) {
```

```
                canCommunicate = true;
                break;
            }
        }
        if (!canCommunicate) {
            toTeach.insert(u);
            toTeach.insert(v);
        }
    }

    int minTeach = INT_MAX;
    for (int lang = 1; lang <= n; ++lang) {
        int count = 0;
        for (int user : toTeach) {
            if (!languageMap[user].count(lang)) {
                ++count;
            }
        }
        minTeach = min(minTeach, count);
    }
    return minTeach;
  }
};

int main() {
  Solution solution;
  int n, m, f;
  cout << "Enter the number of languages (n): ";
  cin >> n;
  cout << "Enter the number of users (m): ";
  cin >> m;
  vector<vector<int>> languages(m);
  cout << "Enter the languages each user knows:\n";
  for (int i = 0; i < m; ++i) {
    int len;
    cout << "Enter the number of languages user " << i + 1 << " knows: ";
    cin >> len;
    languages[i].resize(len);
    cout << "Enter the languages: ";
    for (int j = 0; j < len; ++j) {
      cin >> languages[i][j];
```

```
        }
    }

    cout << "Enter the number of friendships: ";
    cin >> f;

    vector<vector<int>> friendships(f, vector<int>(2));
    cout << "Enter the friendships:\n";
    for (int i = 0; i < f; ++i) {
        cout << "Friendship " << i + 1 << ": ";
        cin >> friendships[i][0] >> friendships[i][1];
    }
    cout << "The minimum number of users to teach is: "
         << solution.minimumTeachings(n, languages, friendships) << endl;

    return 0;
}
```

Output:

```
Enter the number of languages (n): 3
Enter the number of users (m): 2
Enter the languages each user knows:
Enter the number of languages user 1 knows: 1
Enter the languages: 2
Enter the number of languages user 2 knows: 2
Enter the languages: 1 3
Enter the number of friendships: 2
Enter the friendships:
Friendship 1: 1 2
Friendship 2: 2 3
The minimum number of users to teach is: 2


=== Code Execution Successful ===
```

Question 3 Count Ways to Make Array With Product

You are given a 2D integer array, queries. For each queries[i], where queries[i] = [ni, ki], find the number of different ways you can place positive integers into an array of size ni such that the product of the integers is ki. As the number of ways may be too large, the answer to the ith query is the number of ways modulo 109 + 7.
Return an integer array answer where answer.length == queries.length, and answer[i] is the answer to the ith query.

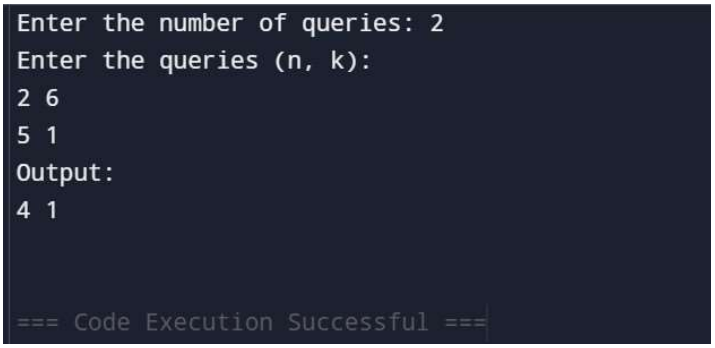
Solution:

```
#include <iostream>
#include <vector>
#include <unordered_map>
using namespace std;
const int MOD = 1e9 + 7;

// Function to calculate modular exponentiation
int modExp(int base, int exp, int mod) {
    int result = 1;
    while (exp > 0) {
        if (exp % 2 == 1) {
            result = (1LL * result * base) % mod;
        }
        base = (1LL * base * base) % mod;
        exp /= 2;
    }
    return result;
}
// Function to precompute factorials and modular inverses
void precomputeFactorials(int maxN, vector<int>& fact, vector<int>& invFact) {
    fact[0] = invFact[0] = 1;
    for (int i = 1; i <= maxN; ++i) {
        fact[i] = (1LL * fact[i - 1] * i) % MOD;
    }
    invFact[maxN] = modExp(fact[maxN], MOD - 2, MOD);
    for (int i = maxN - 1; i >= 1; --i) {
        invFact[i] = (1LL * invFact[i + 1] * (i + 1)) % MOD;
    }
}
// Function to count prime factors
unordered_map<int, int> primeFactorize(int n) {
    unordered_map<int, int> factors;
    for (int i = 2; i * i <= n; ++i) {
        while (n % i == 0) {
            factors[i]++;
            n /= i;
        }
    }
    if (n > 1) factors[n]++;
    return factors;
```

```
}
// Function to calculate the number of ways
int countWays(int n, int k, const vector<int>& fact, const vector<int>& invFact) {
    if (k == 1) return 1; // Special case where all elements are 1
    unordered_map<int, int> factors = primeFactorize(k);
    int result = 1;
    for (const auto& [prime, count] : factors) {
        result = (1LL * result * fact[count + n - 1] % MOD * invFact[count] % MOD * invFact[n -
1] % MOD) % MOD;
    }
    return result;
}

int main() {
    int q;
    cout << "Enter the number of queries: ";
    cin >> q;

    vector<vector<int>> queries(q, vector<int>(2));
    cout << "Enter the queries (n, k):\n";
    for (int i = 0; i < q; ++i) {
        cin >> queries[i][0] >> queries[i][1];
    }

    int maxN = 0;
    for (const auto& query : queries) {
        maxN = max(maxN, query[0]);
    }
    vector<int> fact(maxN + 1), invFact(maxN + 1);
    precomputeFactorials(maxN, fact, invFact);

    vector<int> answer(q);
    for (int i = 0; i < q; ++i) {
        answer[i] = countWays(queries[i][0], queries[i][1], fact, invFact);
    }

    cout << "Output:\n";
    for (int ans : answer) {
        cout << ans << " ";
    }
    cout << endl;
```

```
    return 0;
}
```

Output:



Q4 Maximum Twin Sum of a Linked List

In a linked list of size n, where n is even, the ith node (0-indexed) of the linked list is known as the twin of the (n-1-i)th node, if 0 <= i <= (n / 2) - 1.
For example, if n = 4, then node 0 is the twin of node 3, and node 1 is the twin of node 2. These are the only nodes with twins for n = 4.
The twin sum is defined as the sum of a node and its twin.
Given the head of a linked list with even length, return the maximum twin sum of the linked list.

Solution:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// Definition for singly-linked list.
struct ListNode {
    int val;
    ListNode* next;
    ListNode() : val(0), next(nullptr) {}
    ListNode(int x) : val(x), next(nullptr) {}
    ListNode(int x, ListNode* next) : val(x), next(next) {}
};

// Function to find the maximum twin sum
int pairSum(ListNode* head) {
    vector<int> values;
    ListNode* current = head;
    while (current) {
        values.push_back(current->val);
        current = current->next;
```

```
    }
    int maxTwinSum = 0, n = values.size();
    for (int i = 0; i < n / 2; ++i) {
        maxTwinSum = max(maxTwinSum, values[i] + values[n - 1 - i]);
    }
    return maxTwinSum;
}
// Function to create a linked list from vector
ListNode* createLinkedList(const vector<int>& nums) {
    ListNode* head = nullptr;
    ListNode* tail = nullptr;
    for (int num : nums) {
        ListNode* newNode = new ListNode(num);
        if (!head) {
            head = tail = newNode;
        } else {
            tail->next = newNode;
            tail = newNode;
        }
    }
    return head;
}
int main() {
    int n;
    cout << "Enter the number of nodes in the linked list: ";
    cin >> n;
    cout << "Enter the node values:\n";
    vector<int> values(n);
    for (int i = 0; i < n; ++i) {
        cin >> values[i];
    }
    ListNode* head = createLinkedList(values);
    int result = pairSum(head);
    cout << "Maximum twin sum of the linked list: " << result << endl;
    return 0;
}
```

Output:

```
Enter the number of nodes in the linked list: 4
Enter the node values:
5 4 2 1
Maximum twin sum of the linked list: 6


=== Code Execution Successful ===
```