# DOMAIN WINTER WINNING CAMP

**Student Name: OM GUPTA**                    **UID: 22BCS12174**

**Branch: BE CSE**                    **Section/Group: TPP_FL_603-A**

## *DAY 2:*

### *VERY EASY*

**QUES 1:** Majority Elements

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

The majority element is the element that appears more than $\lfloor n / 2 \rfloor$ times. You may assume that the majority element always exists in the array.

**Solution:**

```cpp
#include <iostream>

#include <vector>

using namespace std;

int majorityElement(vector<int>& nums) {

    int candidate = 0, count = 0;

    for (int num : nums) {

        if (count == 0) {

            candidate = num;

        }

        count += (num == candidate) ? 1 : -1;

    }

    return candidate;

}

int main() {

    vector<int> nums;

    int n;

    cout << "Enter the size of the array: ";
```

```cpp
    cin >> n;

    cout << "Enter the elements of the array: ";

    nums.resize(n);

    for (int i = 0; i < n; ++i) {

        cin >> nums[i];

    }

    int result = majorityElement(nums);

    cout << "The majority element is: " << result << endl;


    return 0;

}
```

```
Enter the size of the array: 3
Enter the elements of the array: 3
2
3
The majority element is: 3
```

## QUES 2: Single Number

Objective: 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:**

```cpp
#include <iostream>

#include <vector>

using namespace std;

int singleNumber(vector<int>& nums) {

    int result = 0;

    for (int num : nums) {

        result ^= num; // XOR all elements

    }
```

```cpp
        return result;
    }
int main() {
    int n;
    cout << "Enter the number of elements in the array: ";
    cin >> n;
    vector<int> nums(n);
    cout << "Enter the elements of the array: ";
    for (int i = 0; i < n; ++i) {
        cin >> nums[i];
    }
    int single = singleNumber(nums);
    cout << "The single number is: " << single << endl;
    return 0;
}
```

```
Enter the number of elements in the array: 5
Enter the elements of the array: 2 3 4 2 3
The single number is: 4
```

**QUES 3:** Convert Sorted Array to Binary Search Tree

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

**Solution:**

```cpp
#include <iostream>
#include <vector>
using namespace std;
struct TreeNode {
    int val;
    TreeNode* left;
```

```cpp
    TreeNode* right;

    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};
TreeNode* sortedArrayToBST(vector<int>& nums, int left, int right) {

    if (left > right) return nullptr;

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

    TreeNode* root = new TreeNode(nums[mid]);

    root->left = sortedArrayToBST(nums, left, mid - 1);

    root->right = sortedArrayToBST(nums, mid + 1, right);

    return root;

}
TreeNode* sortedArrayToBST(vector<int>& nums) {

    return sortedArrayToBST(nums, 0, nums.size() - 1);

}
void inorderTraversal(TreeNode* root) {

    if (!root) return;

    inorderTraversal(root->left);

    cout << root->val << " ";

    inorderTraversal(root->right);

}
int main() {

    vector<int> nums;

    int n;

    cout << "Enter the number of elements in the sorted array: ";

    cin >> n;

    cout << "Enter the elements of the sorted array: ";

    nums.resize(n);

    for (int i = 0; i < n; ++i) {

        cin >> nums[i];
```

```
    }
    TreeNode* root = sortedArrayToBST(nums);
    cout << "In-order Traversal of the Height-Balanced BST: ";
    inorderTraversal(root);
    cout << endl;
    return 0;
}
```

```
Enter the number of elements in the sorted array: 6
Enter the elements of the sorted array: 2 3 4 1 5 2
In-order Traversal of the Height-Balanced BST: 2 3 4 1 5 2
```

**QUES 4: Merge Two Sorted Lists**

Objective: 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*.

**Solution:**

```cpp
#include <iostream>
using namespace std;
struct ListNode {
    int val;
    ListNode* next;
    ListNode(int x) : val(x), next(nullptr) {}
};
ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {

    if (!list1) return list2;
    if (!list2) return list1;

    if (list1->val < list2->val) {
```

```cpp
 list1->next = mergeTwoLists(list1->next, list2);

    return list1;

  } else {

    list2->next = mergeTwoLists(list1, list2->next);

    return list2;

  }

}


void printList(ListNode* head) {

  while (head) {

    cout << head->val << " ";

    head = head->next;

  }

  cout << endl;

}


ListNode* createList(int arr[], int size) {

  if (size == 0) return nullptr;


  ListNode* head = new ListNode(arr[0]);

  ListNode* current = head;


  for (int i = 1; i < size; ++i) {

    current->next = new ListNode(arr[i]);

    current = current->next;

  }


  return head;
```

```
}
int main() {

    int arr1[] = {1, 2, 4};
    int arr2[] = {1, 3, 4};

    ListNode* list1 = createList(arr1, 3);
    ListNode* list2 = createList(arr2, 3);

    ListNode* mergedList = mergeTwoLists(list1, list2);

    cout << "Merged Linked List: ";
    printList(mergedList);
    return 0;
}
```

```
Merged Linked List: 1 1 2 3 4 4
```

**QUES 5**: **Linked List Cycle**

Objective :  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.

**Solution:**

**#include <iostream>**

**using namespace std;**

**struct ListNode {**

  **int val;**

  **ListNode\* next;**

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


bool hasCycle(ListNode* head) {
    if (!head || !head->next) return false;

    ListNode* slow = head;
    ListNode* fast = head;

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

    return false;
}

ListNode* createCyclicList(int arr[], int size, int pos) {
    if (size == 0) return nullptr;

    ListNode* head = new ListNode(arr[0]);
    ListNode* current = head;
    ListNode* cycleNode = nullptr;

    for (int i = 1; i < size; ++i) {
        current->next = new ListNode(arr[i]);
```

```cpp
        current = current->next;


      if (i == pos) {

         cycleNode = current;

      }

    }


    if (cycleNode) {

       current->next = cycleNode;

    }


    return head;

}


int main() {


   int arr[] = {3, 2, 0, -4};

   int pos = 1;

   ListNode* head = createCyclicList(arr, 4, pos);


   if (hasCycle(head)) {

      cout << "The linked list has a cycle." << endl;

   } else {

      cout << "The linked list does not have a cycle." << endl;

   }


   return 0;

}
```

```
The linked list has a cycle.
```

*EASY*

**QUES 6:** Pascal's Triangle

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

**Solution:**

```cpp
#include <iostream>
#include <vector>
using namespace std;
vector<vector<int>> generatePascalsTriangle(int numRows) {
    vector<vector<int>> triangle;
    for (int i = 0; i < numRows; ++i) {
        vector<int> row(i + 1, 1);
        for (int j = 1; j < i; ++j) {
            row[j] = triangle[i - 1][j - 1] + triangle[i - 1][j];
        }
        triangle.push_back(row);
    }
    return triangle;
}
int main() {
    int numRows;
    cout << "Enter the number of rows for Pascal's Triangle: ";
    cin >> numRows;
    vector<vector<int>> pascalsTriangle = generatePascalsTriangle(numRows);
    cout << "Pascal's Triangle:" << endl;
    for (const auto& row : pascalsTriangle) {
        for (int num : row) {
```

```
            cout << num << " ";

        }

        cout << endl;

    }

    return 0;

}
```

```
Enter the number of rows for Pascal's Triangle: 5
Pascal's Triangle:
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

**QUES 7:** Remove Element

Objective : 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.

Consider the number of unique elements of nums to be k, to get accepted, you need to do the following things:

Change the array nums such that the first k elements of nums contain the unique elements in the order they were present in nums initially. The remaining elements of nums are not important as well as the size of nums.

Return k.

**Solution:**

**#include <iostream>**

**#include <vector>**

**using namespace std;**

**int removeDuplicates(vector<int>& nums) {**

```cpp
    if (nums.empty()) return 0;

    int slow = 0;

    for (int fast = 1; fast < nums.size(); ++fast) {
        if (nums[fast] != nums[slow]) {
            ++slow;
            nums[slow] = nums[fast];
        }
    }

    return slow + 1;
}
int main() {
    vector<int> nums = {1, 1, 2, 3, 3, 4};

    int k = removeDuplicates(nums);

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

    return 0;
}
```

```
Number of unique elements: 4
Modified array: 1 2 3 4
```

**QUES 8:** Baseball Game

Objective : You are keeping the scores for a baseball game with strange rules. At the beginning of the game, you start with an empty record.

You are given a list of strings operations, where operations[i] is the ith operation you must apply to the record and is one of the following:

An integer x.

Record a new score of x.

'+'.

Record a new score that is the sum of the previous two scores.

'D'.

Record a new score that is the double of the previous score.

'C'.

Invalidate the previous score, removing it from the record.

Return the sum of all the scores on the record after applying all the operations.

The test cases are generated such that the answer and all intermediate calculations fit in a 32-bit integer and that all operations are valid.

**Solution:**

**#include <iostream>**

**#include <vector>**

**#include <stack>**

**using namespace std;**

**int calPoints(vector<string>& operations) {**

**stack<int> scores;**

```cpp
    int total = 0;

    for (const string& operation : operations) {
        if (operation == "+") {
            int last = scores.top();
            scores.pop();
            int secondLast = scores.top();
            scores.push(last);
            scores.push(last + secondLast);
        }
        else if (operation == "D") {
            scores.push(2 * scores.top());
        }
        else if (operation == "C") {
            scores.pop();
        }
        else {
            scores.push(stoi(operation));
        }
    }
    while (!scores.empty()) {
        total += scores.top();
        scores.pop();
    }

    return total;
}

int main() {
```
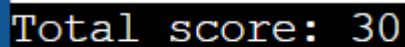
```
vector<string> operations = {"5", "2", "C", "D", "+"};

cout << "Total score: " << calPoints(operations) << endl;


return 0;

}
```

```
Total score: 30
```

**QUES 9: Remove Linked List Elements**

Objective:  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) {


    ListNode* dummy = new ListNode(0);

    dummy->next = head;

    ListNode* current = dummy;


    while (current->next != nullptr) {
```

```cpp
            if (current->next->val == val) {


                ListNode* temp = current->next;

                current->next = current->next->next;

                delete temp;

            } else {

                current = current->next;

            }

        }

        ListNode* newHead = dummy->next;

        delete dummy;

        return newHead;

    }


    void printList(ListNode* head) {

        while (head != nullptr) {

            cout << head->val << " ";

            head = head->next;

        }

        cout << endl;

    }


    ListNode* createList(const vector<int>& values) {

        ListNode* dummy = new ListNode(0);

        ListNode* current = dummy;

        for (int val : values) {

            current->next = new ListNode(val);

            current = current->next;

        }
```

```cpp
    return dummy->next;
}


int main() {

    vector<int> values = {1, 2, 6, 3, 4, 5, 6};
    ListNode* head = createList(values);


    cout << "Original list: ";
    printList(head);


    int val = 6;
    head = removeElements(head, val);


    cout << "Updated list after removing " << val << ": ";
    printList(head);


    return 0;
}
```

```
 Original list: 1 2 6 3 4 5 6
 Updated list after removing 6: 1 2 3 4 5
```

**QUES 10:** Reseved linked list

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

**Solution:**

**#include <iostream>**

**using namespace std;**

**struct ListNode {**

```cpp
    int val;

    ListNode* next;

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


ListNode* reverseList(ListNode* head) {

    ListNode* prev = nullptr;

    ListNode* curr = head;

    ListNode* nextNode = nullptr;


    while (curr != nullptr) {

        nextNode = curr->next;

        curr->next = prev;

        prev = curr;

        curr = nextNode;

    }


    return prev;

}


void printList(ListNode* head) {

    while (head != nullptr) {

        cout << head->val << " ";

        head = head->next;

    }

    cout << endl;

}


ListNode* createList(const vector<int>& values) {
```

```cpp
    ListNode* dummy = new ListNode(0);

    ListNode* current = dummy;

    for (int val : values) {

        current->next = new ListNode(val);

        current = current->next;

    }

    return dummy->next;

}


int main() {


    vector<int> values = {1, 2, 3, 4, 5};

    ListNode* head = createList(values);


    cout << "Original list: ";

    printList(head);


    head = reverseList(head);


    cout << "Reversed list: ";

    printList(head);


    return 0;

}
```

```
 Original list: 1 2 3 4 5
 Reversed list: 5 4 3 2 1
```

*MEDIUM*

**QUES 11:** **Container With Most Water**

Objective: 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.

**Solution:**

**#include <iostream>**

**#include <vector>**

**#include <algorithm>**

**using namespace std;**

**int maxArea(vector<int>& height) {**

   **int left = 0;**

   **int right = height.size() - 1;**

   **int max_area = 0;**

   **while (left < right) {**

      **// Calculate the area with the current left and right lines**

      **int width = right - left;**

      **int current_area = min(height[left], height[right]) * width;**

      **// Update the maximum area found**

      **max_area = max(max_area, current_area);**

```
    // Move the pointer pointing to the shorter line
    if (height[left] < height[right]) {
        left++;
    } else {
        right--;
    }
  }


  return max_area;
}


int main() {
    vector<int> height = {1,8,6,2,5,4,8,3,7};
    int result = maxArea(height);
    cout << "The maximum area of water the container can store is: " << result << endl;
    return 0;
}
```

```
The maximum area of water the container can store is: 49
```

QUES 12: Valid Sudoku

Objective: 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>

```cpp
#include <unordered_set>

using namespace std;

bool isValidSudoku(vector<vector<char>>& board) {
    unordered_set<string> seen;

    for (int i = 0; i < 9; ++i) {
        for (int j = 0; j < 9; ++j) {
            if (board[i][j] != '.') {
                char num = board[i][j];
                string rowKey = "row" + to_string(i) + ":" + num;
                string colKey = "col" + to_string(j) + ":" + num;
                string subgridKey = "subgrid" + to_string(i / 3) + to_string(j / 3) + ":" + num;
                if (seen.count(rowKey) || seen.count(colKey) || seen.count(subgridKey)) {
                    return false;
                }
                seen.insert(rowKey);
                seen.insert(colKey);
                seen.insert(subgridKey);
            } } }
    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'}
  };


  if (isValidSudoku(board)) {

    cout << "The Sudoku board is valid." << endl;

  } else {

    cout << "The Sudoku board is not valid." << endl;

  }

  return 0;

}
```

```
The Sudoku board is valid.
```

**QUES 13:** Jump Game II

Objective: 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>

```cpp
#include <algorithm>

using namespace std;

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

    int jumps = 0, current_end = 0, farthest = 0;

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

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

            if (current_end >= n - 1) {
                break;
            }
        }
    }

    return jumps;
}

int main() {
    vector<int> nums = {2, 3, 1, 1, 4};
    cout << "Minimum number of jumps: " << jump(nums) << endl;
```

```
    nums = {2, 3, 0, 1, 4};

    cout << "Minimum number of jumps: " << jump(nums) << endl;


    return 0;
}
```

```
Minimum number of jumps: 2
Minimum number of jumps: 2
```

**QUES 14:** Populating Next Right Pointers in Each Node.

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

**Solution:**

**#include <iostream>**

**using namespace std;**


**struct Node {**

   **int val;**

   **Node *left;**

   **Node *right;**

   **Node *next;**

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

**};**


**void connect(Node* root) {**

   **if (root == NULL) return;**

```cpp
    Node* level_start = root;


    while (level_start->left != NULL) {
        Node* current = level_start;
        while (current != NULL) {
            current->left->next = current->right;


            if (current->next != NULL) {
                current->right->next = current->next->left;
            }


            current = current->next;
        }


        level_start = level_start->left;
    }
}

void printLevels(Node* root) {
    Node* level_start = root;
    while (level_start != NULL) {
        Node* current = level_start;
        while (current != NULL) {
            cout << current->val << " -> ";
            current = current->next;
        }
        cout << "NULL" << endl;
        level_start = level_start->left;
```

```
    }
}

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);


    connect(root);


    printLevels(root);


    return 0;
}
```

```
1 -> NULL
2 -> 3 -> NULL
4 -> 5 -> 6 -> 7 -> NULL
```

### *HARD*

**QUES 15:** **Maximum Number of Groups Getting Fresh Donuts**

Objective: 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.

**Solution:**

```cpp
#include <iostream>

#include <vector>

#include <unordered_map>

#include <functional>


using namespace std;


class Solution {
public:
    int maxHappyGroups(int batchSize, vector<int>& groups) {
        vector<int> count(batchSize, 0);


        for (int group : groups) {
            count[group % batchSize]++;
        }


        int happyGroups = count[0];
        count[0] = 0;


        for (int i = 1; i <= batchSize / 2; i++) {
            if (i == batchSize - i) {
                happyGroups += count[i] / 2;
                count[i] %= 2;
            } else {

                int complement = min(count[i], count[batchSize - i]);
```

```cpp
            happyGroups += complement;

            count[i] -= complement;

            count[batchSize - i] -= complement;

        }

    }


    unordered_map<string, int> memo;


    function<int(int, vector<int>&)> dp = [&](int remainder, vector<int>& count) {

        string key = to_string(remainder) + "#";

        for (int x : count) key += to_string(x) + ",";


        if (memo.find(key) != memo.end()) return memo[key];


        bool allZero = true;

        for (int x : count) {

            if (x > 0) {

                allZero = false;

                break;

            }

        }

        if (allZero) return 0;


        int maxHappy = 0;

        for (int i = 0; i < batchSize; i++) {

            if (count[i] > 0) {

                count[i]--;

                int newRemainder = (remainder + i) % batchSize;

                int happy = (newRemainder == 0) ? 1 : 0;
```

```
                maxHappy = max(maxHappy, happy + dp(newRemainder, count));

                count[i]++;

            }

        }


        return memo[key] = maxHappy;

    };


    return happyGroups + dp(0, count);

    }

};


int main() {

    Solution sol;

    int batchSize = 3;

    vector<int> groups = {1, 2, 3, 4, 5, 6};

    cout << "Maximum Happy Groups: " << sol.maxHappyGroups(batchSize, groups) <<
endl;

    return 0;

}
```

```
Maximum Happy Groups: 4
```

**QUES 17:** **Maximum Number of Darts Inside of a Circular Dartboard**

Objective: 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:**

```cpp
#include <iostream>

#include <vector>

#include <cmath>

#include <algorithm>


using namespace std;


class Solution {
public:
    int numPoints(vector<vector<int>>& darts, int r) {
        int n = darts.size();
        int maxDarts = 1;
        double radius = r * 1.0;


        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                vector<pair<double, double>> centers = findCenters(darts[i], darts[j], radius);
                for (auto& center : centers) {
                    int count = countDarts(darts, center, radius);
                    maxDarts = max(maxDarts, count);
                }
            }
        }


        return maxDarts;
    }


private:
    vector<pair<double, double>> findCenters(vector<int>& p1, vector<int>& p2,
double radius) {
```

```cpp
        double x1 = p1[0], y1 = p1[1];
        double x2 = p2[0], y2 = p2[1];


        double d = sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1));


        if (d > 2 * radius) return {};


        double mx = (x1 + x2) / 2.0;
        double my = (y1 + y2) / 2.0;


        double h = sqrt(radius * radius - (d / 2.0) * (d / 2.0));


        double dx = -(y2 - y1) / d;
        double dy = (x2 - x1) / d;


        return {
           {mx + h * dx, my + h * dy},
           {mx - h * dx, my - h * dy}
        };
    }


    int countDarts(vector<vector<int>>& darts, pair<double, double> center, double
radius) {
        int count = 0;
        double cx = center.first, cy = center.second;


        for (auto& dart : darts) {
           double x = dart[0], y = dart[1];
           if (sqrt((x - cx) * (x - cx) + (y - cy) * (y - cy)) <= radius + 1e-7) {
              count++;
```

```
        }
      }


      return count;

   }
};


int main() {
   Solution sol;
   vector<vector<int>> darts = {{-2, 0}, {2, 0}, {0, 2}, {0, -2}};
   int r = 2;
   cout << "Maximum number of darts on the dartboard: " << sol.numPoints(darts, r) << endl;
   return 0;
}
```

```
Maximum number of darts on the dartboard: 4
```

*VERY HARD*

**QUES 18:** . Find Minimum Time to Finish All Jobs

Objective : 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.

**Solution:**

**#include <iostream>**

**#include <vector>**

```cpp
#include <algorithm>
#include <numeric>


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

        return result;
    }


private:
    bool canAssign(vector<int>& jobs, int k, int maxTime) {
        vector<int> workers(k, 0);
        return backtrack(jobs, workers, maxTime, 0);
    }
```

```cpp
    bool backtrack(vector<int>& jobs, vector<int>& workers, int maxTime, int
jobIndex) {
        if (jobIndex == jobs.size()) return true;


        int job = jobs[jobIndex];
        for (int i = 0; i < workers.size(); i++) {
            if (workers[i] + job <= maxTime) {
                workers[i] += job;
                if (backtrack(jobs, workers, maxTime, jobIndex + 1)) return true;
                workers[i] -= job;
            }


            if (workers[i] == 0) break;
        }


        return false;
    }
};


int main() {
    Solution sol;
    vector<int> jobs = {3, 2, 3};
    int k = 3;
    cout << "Minimum maximum working time: " << sol.minimumTimeRequired(jobs,
k) << endl;
    return 0;
}
```

```
Minimum maximum working time: 3
```

**QUES 19:** **Minimum Number of People to Teach**

Objective: 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.

**Solution:**

```cpp
#include <iostream>

#include <vector>

#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_set<int> problematic;
        for (const auto& friendship : friendships) {
            int u = friendship[0] - 1, v = friendship[1] - 1;
            if (!hasCommonLanguage(languages[u], languages[v])) {
                problematic.insert(u);
                problematic.insert(v);
            }
```

```cpp
        }

        unordered_map<int, int> teachCount;
        for (int user : problematic) {
            for (int lang : languages[user]) {
                teachCount[lang]++;
            }
        }

        int minTeach = problematic.size();
        for (int lang = 1; lang <= n; lang++) {
            int usersToTeach = problematic.size() - teachCount[lang];
            minTeach = min(minTeach, usersToTeach);
        }

        return minTeach;
    }

private:
    bool hasCommonLanguage(const vector<int>& user1, const vector<int>& user2) {
        unordered_set<int> langSet(user1.begin(), user1.end());
        for (int lang : user2) {
            if (langSet.count(lang)) return true;
        }
        return false;
    }
};

int main() {
```

```
    Solution sol;

    int n = 3;

    vector<vector<int>> languages = {{1, 2}, {2, 3}, {1, 3}};

    vector<vector<int>> friendships = {{1, 2}, {2, 3}, {1, 3}};

    cout << "Minimum number of users to teach: " << sol.minimumTeachings(n,
languages, friendships) << endl;

    return 0;

}
```

```
Minimum number of users to teach: 0
```