PROBLEM 1

Objective: 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.

Code:

```
#include <iostream>
#include <unordered_map>
#include <vector>
using namespace std;
int main() {
int n;
cout << "Enter the number of elements in the array: ";
cin >> n;
vector<int> arr(n);
cout << "Enter the elements of the array:" << endl;
for (int i = 0; i < n; i++) {
cin >> arr[i];
}
unordered_map<int, int> freq;
int maxFreq = 0;
for (int num : arr) {
freq[num]++;
maxFreq = max(maxFreq, freq[num]);
}
cout << "The element(s) repeated the most times: ";
for (auto& pair : freq) {
if (pair.second == maxFreq) {
cout << pair.first << " "; }}
cout << endl;
return 0;}
```

Output:

```
Enter the number of elements in the array: 6
Enter the elements of the array:
10
20
20
20
10
30
The element(s) repeated the most times: 20


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

PROBLEM 2

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.

Code:

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
int n, num, result = 0;
cout << "Enter the number of elements: ";
cin >> n;
vector<int> nums(n);
cout << "Enter the elements: ";
for (int i = 0; i < n; i++) {
cin >> nums[i];
result ^= nums[i];
}
cout << "The single number is: " << result << endl;
return 0;
}
```

Output:

```
Enter the number of elements: 5
Enter the elements: 20 20 22 22 23
The single number is: 23


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

PROBLEM 3

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

Code:

```
#include <iostream>
#include <vector>
using namespace std;
struct TreeNode {
int val;
TreeNode *left, *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;
}
void preOrder(TreeNode* root) {
if (!root) return;
cout << root->val << " ";
preOrder(root->left);
preOrder(root->right);  }
int main() {
int n;
cout << "Enter the number of elements: ";
cin >> n;
vector<int> nums(n);
cout << "Enter the sorted elements: ";
for (int i = 0; i < n; i++) cin >> nums[i];
TreeNode* root = sortedArrayToBST(nums, 0, n - 1);
cout << "Pre-order traversal of the BST: ";
preOrder(root);
cout << endl;
return 0;
}
```

Output:

```
Enter the number of elements: 5
Enter the sorted elements: 20 30 40 90 100
Pre-order traversal of the BST: 40 20 30 90 100


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

PROBLEM 4

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 lists.

Code:

```
#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) {
list1->next = mergeTwoLists(list1->next, list2);
return list1;
} else {
list2->next = mergeTwoLists(list1, list2->next);
return list2;
}
}
ListNode* createList(int n) {
ListNode *head = nullptr, *tail = nullptr;
for (int i = 0; i < n; i++) {
int val;
cin >> val;
ListNode* newNode = new ListNode(val);
if (!head) head = tail = newNode;
else {
tail->next = newNode;
tail = newNode;
}
}
return head;
}
void printList(ListNode* head) {
while (head) {
cout << head->val << " ";
head = head->next;
}
cout << endl;
}
int main() {
int n1, n2;
cout << "Enter the number of elements in list1: ";
```

```
cin >> n1;
cout << "Enter the sorted elements of list1: ";
ListNode* list1 = createList(n1);
cout << "Enter the number of elements in list2: ";
cin >> n2;
cout << "Enter the sorted elements of list2: ";
ListNode* list2 = createList(n2);
ListNode* mergedList = mergeTwoLists(list1, list2);
cout << "Merged list: ";
printList(mergedList);
return 0;
}
```

Output:

```
Enter the number of elements in list1: 5
Enter the sorted elements of list1: 20 30 60 90 91
Enter the number of elements in list2: 6
Enter the sorted elements of list2: 200 600 800 9001 10000 899999
Merged list: 20 30 60 90 91 200 600 800 9001 10000 899999


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

PROBLEM 5

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

Code:

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

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

```
bool hasCycle(ListNode* head) {
ListNode *slow = head, *fast = head;
while (fast && fast->next) {
slow = slow->next;
fast = fast->next->next;
if (slow == fast) return true;
}
return false;
}

ListNode* createList(int n, int pos) {
ListNode *head = nullptr, *tail = nullptr, *cycleNode = nullptr;
for (int i = 0; i < n; i++) {
int val;
cin >> val;
ListNode* newNode = new ListNode(val);
if (!head) head = tail = newNode;
else {
tail->next = newNode;
tail = newNode;
}
if (i == pos) cycleNode = newNode;
}
if (tail && cycleNode) tail->next = cycleNode;
return head;
}
int main() {
int n, pos;
cout << "Enter the number of nodes: ";
cin >> n;
cout << "Enter the node values: ";
cout << "Enter the position where the tail connects (-1 for no cycle): ";
cin >> pos;
ListNode* head = createList(n, pos);
cout << (hasCycle(head) ? "The linked list has a cycle." : "The linked list does not have a cycle.")
<< endl;
return 0;
}
```

Output:

```
Enter the number of nodes: 4
Enter the node values: Enter the position where the tail connects (-1 for no cycle): 1
3
2
0
-4
The linked list has a cycle.


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

PROBLEM 6

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

Code:

```
#include <iostream>
#include <vector>
#include <iomanip>
using namespace std;
void printPascalsTriangle(int rows) {
vector<vector<int>> triangle(rows);
for (int i = 0; i < rows; i++) {
triangle[i].resize(i + 1);
triangle[i][0] = triangle[i][i] = 1;
for (int j = 1; j < i; j++) {
triangle[i][j] = triangle[i - 1][j - 1] + triangle[i - 1][j];
}
cout << string((rows - i - 1) * 2, ' ');
for (int num : triangle[i]) {
cout << num << "   ";
}
cout << endl;  }}
int main() {
int rows;
cout << "Enter the number of rows for Pascal's Triangle: ";
cin >> rows;
printPascalsTriangle(rows);
return 0;
}
```

Output:

```
Enter the number of rows for Pascal's Triangle: 10
                    1
                  1   1
                1   2   1
              1   3   3   1
            1   4   6   4   1
          1   5   10   10   5   1
        1   6   15   20   15   6   1
      1   7   21   35   35   21   7   1
    1   8   28   56   70   56   28   8   1
  1   9   36   84   126   126   84   36   9   1


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

PROBLEM 7

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.

Code:

```
#include <iostream>
#include <vector>
using namespace std;
int removeDuplicates(vector<int>& nums) {
int k = 0;
for (int i = 0; i < nums.size(); i++)
if (i == 0 || nums[i] != nums[i - 1])
nums[k++] = nums[i];
 return k;
}
int main() {
int n;
cout << "Enter the number of elements: ";
cin >> n;
vector<int> nums(n);
cout << "Enter the sorted elements: ";
for (int i = 0; i < n; i++) cin >> nums[i];
int k = removeDuplicates(nums);
cout << "Number of unique elements: " << k << endl;
cout << "Array after removing duplicates: ";
for (int i = 0; i < k; i++) cout << nums[i] << " ";
cout << endl;
return 0;
}
```

Output:

```
Enter the number of elements: 6
Enter the sorted elements: 29 33 33 98 100 1019
Number of unique elements: 5
Array after removing duplicates: 29 33 98 100 1019


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

PROBLEM 8

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.

Code:

```
#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(0), *cur = &dummy;
dummy.next = head;
while (cur->next) cur->next->val == val ? cur->next = cur->next->next : cur = cur->next;
return dummy.next;
}
ListNode* reverseList(ListNode* head) {
ListNode* prev = nullptr;
while (head) {
ListNode* next = head->next;
head->next = prev;
prev = head;
head = next;
}
return prev;
}
void printList(ListNode* head) {
while (head) {
cout << head->val << " ";
head = head->next;
}
cout << endl;
}
ListNode* createList() {
int n, val;
cin >> n;
ListNode *head = nullptr, *tail = nullptr;
while (n-- && cin >> val) {
ListNode* node = new ListNode(val);
if (!head) head = tail = node;
else tail = tail->next = node;
}
return head;
}
int main() {
cout << "Enter number of nodes and values: ";
```

```
ListNode* head = createList();
cout << "Enter value to remove: ";
int val;
cin >> val;
head = removeElements(head, val);
cout << "After removal: ";
printList(head);
cout << "Reversed list: ";
printList(reverseList(head));
return 0;
}
```

Output:

```
Enter number of nodes and values: 5
1 2 3 4 5
Enter value to remove: 3
After removal: 1 2 4 5
Reversed list: 5 4 2 1


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

PROBLEM 9

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.Notice that you may not slant the container.

Code:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int maxArea(vector<int>& height) {
int left = 0, right = height.size() - 1, maxWater = 0;
while (left < right) {
maxWater = max(maxWater, min(height[left], height[right]) * (right - left));
if (height[left] < height[right]) left++;
else right--;
}
return maxWater;
}
```

```
int main() {
int n;
cout << "Enter the number of lines: ";
cin >> n;
vector<int> height(n);
cout << "Enter the heights: ";
for (int i = 0; i < n; i++) cin >> height[i];
cout << "Max water contained: " << maxArea(height) << endl;
return 0;
}
```

Output:

```
Enter the number of lines: 9
Enter the heights: 1 5 9 10 11 13 56 89 90
Max water contained: 112


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

PROBLEM 10

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.
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.

Code:

```
#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> row, col, box;
for (int j = 0; j < 9; j++) {
if (board[i][j] != '.' && !row.insert(board[i][j]).second) return false;
if (board[j][i] != '.' && !col.insert(board[j][i]).second) return false;
int boxRow = 3 * (i / 3), boxCol = 3 * (i % 3);
```

```
if (board[boxRow + j / 3][boxCol + j % 3] != '.' && !box.insert(board[boxRow + j / 3][boxCol +
j % 3]).second) return false;
}
}
return true;
}
int main() {
vector<vector<char>> board(9, vector<char>(9));
cout << "Enter the Sudoku board (use '.' for empty cells):\n";
for (int i = 0; i < 9; i++)
for (int j = 0; j < 9; j++)
cin >> board[i][j];
cout << (isValidSudoku(board) ? "true" : "false") << endl;
return 0;
}
```

Output:

```
Enter the Sudoku board (use '.' for empty cells):
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 95 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
true
```

PROBLEM 12

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].

Code:

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int jump(vector<int>& nums) {
int n = nums.size(), jumps = 0, farthest = 0, current_end = 0;
for (int i = 0; i < n - 1; i++) {
farthest = max(farthest, i + nums[i]);
if (i == current_end) {
jumps++;
current_end = farthest;
}
}
return jumps;
}
int main() {
int n;
cout << "Enter the length of 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];
cout << "Minimum number of jumps: " << jump(nums) << endl;
return 0;
}
```

Output:

```
Enter the length of the array: 5
Enter the elements of the array: 2 3 1 1 4
Minimum number of jumps: 2


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

PROBLEM 13

Objective: 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.

Code:

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

struct Node {
int val;
Node *left;
Node *right;
Node *next;
Node(int x) : val(x), left(nullptr), right(nullptr), next(nullptr) {}
};
void connect(Node* root) {
if (!root) return;
Node* level_start = root;
while (level_start->left) {
Node* current = level_start;
while (current) {
current->left->next = current->right;
if (current->next) current->right->next = current->next->left;
current = current->next;
}
level_start = level_start->left;
}
}
Node* buildTree() {
int n;
cout << "Enter number of nodes: ";
cin >> n;
if (n == 0) return nullptr;
cout << "Enter the node values level by level: ";
Node* root = new Node(0);
Node* curr;
Node* temp;
Node* parent;
int value;
cin >> value;
root->val = value;
Node* nodes[n];
nodes[0] = root;
for (int i = 1; i < n; ++i) {
cin >> value;
curr = new Node(value);
nodes[i] = curr;
}
int idx = 0;
for (int i = 0; i < n / 2; ++i) {
```

```
nodes[i]->left = nodes[2 * i + 1];
nodes[i]->right = nodes[2 * i + 2];
}
return root;
}
void printTree(Node* root) {
if (!root) return;
Node* level = root;
while (level) {
Node* curr = level;
while (curr) {
cout << curr->val << " ";
curr = curr->next;
}
cout << "# ";
level = level->left;
}
cout << endl;
}
int main() {
Node* root = buildTree();
connect(root);
printTree(root);
return 0;
}
```

Output:

```
Enter number of nodes: 7
Enter the node values level by level: 1 2 3 4 5 6 7
1 # 2 3 # 4 5 6 7 #


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

PROBLEM 14

Objective: Design your implementation of the circular queue. The circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle, and the last position is connected back to the first position to make a circle. It is also called "Ring Buffer".One of the benefits of the circular queue is that we can make use of the spaces in front of the queue. In a normal queue, once the queue becomes full, we cannot insert the next element even if there is a space in front of the queue. But using the circular queue, we can use the space to store new values.

Code:

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

class MyCircularQueue {
    int *arr;
    int front, rear, size, capacity;

public:
MyCircularQueue(int k)
{
capacity = k;
arr = new int[k];
        front = rear = -1;
        size = 0;
    }

    bool enQueue(int value) {
        if (isFull()) return false;
        if (isEmpty()) front = 0;
        rear = (rear + 1) % capacity;
        arr[rear] = value;
        size++;
        return true;
    }

    bool deQueue() {
        if (isEmpty()) return false;
        if (front == rear) front = rear = -1;
        else front = (front + 1) % capacity;
        size--;
        return true;
    }

    int Front()
    {
        return isEmpty() ? -1 : arr[front];
    }

    int Rear() {
        return isEmpty() ? -1 : arr[rear];
    }

    bool isEmpty()
    {
        return size == 0;
    }

    bool isFull() {
        return size == capacity;
```

```
    }
};

int main()
{
    int k, value;
    cout << "Enter the size of the queue: ";
    cin >> k;
    MyCircularQueue myCircularQueue(k);

    int operations;
    cout << "Enter number of operations: ";
    cin >> operations;

    for (int i = 0; i < operations; i++)
    {
        string operation;
        cout << "Enter operation: ";
        cin >> operation;

        if (operation == "enQueue")
{
            cout << "Enter value: ";
            cin >> value;
            cout << (myCircularQueue.enQueue(value) ? "true" : "false") << endl;
        }
 else if (operation == "deQueue")
{
            cout << (myCircularQueue.deQueue() ? "true" : "false") << endl;
        }
else if (operation == "Front")
{
            cout << myCircularQueue.Front() << endl;
        }
else if (operation == "Rear")
{
            cout << myCircularQueue.Rear() << endl;
        }
else if (operation == "isEmpty")
{
            cout << (myCircularQueue.isEmpty() ? "true" : "false") << endl;
        }
else if (operation == "isFull")
{
            cout << (myCircularQueue.isFull() ? "true" : "false") << endl;
        }
    }

    return 0;
}
```

Output:

```
Enter the size of the queue: 3
Enter number of operations: 10
Enter operation: enQueue
Enter value: 1
true
Enter operation: enQueue
Enter value: 5
true
Enter operation: enQueue
Enter value: 10
true
Enter operation: enQueue
Enter value: 99
false
Enter operation: Rear
10
Enter operation: isFull
true
Enter operation: isEmpty
false
Enter operation: Rear
10
```

PROBLEM 15

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

Code:

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

int maxHappyGroups(int batchSize, vector<int>& groups) {
    sort(groups.begin(), groups.end(), greater<int>());
    int happyGroups = 0, leftover = 0;

    for (int group : groups) {
        if (leftover + group <= batchSize) {
            leftover += group;
            happyGroups++;
        } else {
            leftover = group;
        }
    }
    return happyGroups;
}

int main() {
    int batchSize, n;
    cout << "Enter batch size: ";
    cin >> batchSize;
    cout << "Enter the number of groups: ";
    cin >> n;

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

    cout << "Maximum happy groups: " << maxHappyGroups(batchSize, groups) << endl;
    return 0;
}
```

Output:

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


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

PROBLEM 16

Objective: 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.

Code:

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

int cherryPickup(vector<vector<int>>& grid) {
   int rows = grid.size(), cols = grid[0].size();
   vector<vector<vector<int>>> dp(rows, vector<vector<int>>(cols, vector<int>(cols, 0)));

   for (int r = rows - 2; r >= 0; r--) {
      for (int c1 = 0; c1 < cols; c1++) {
         for (int c2 = 0; c2 < cols; c2++) {
            int maxCherries = 0;
            for (int d1 = -1; d1 <= 1; d1++) {
               for (int d2 = -1; d2 <= 1; d2++) {
                  int nc1 = c1 + d1, nc2 = c2 + d2;
                  if (nc1 >= 0 && nc1 < cols && nc2 >= 0 && nc2 < cols) {
                     maxCherries = max(maxCherries, dp[r + 1][nc1][nc2]);
                  }
               }
            }
            dp[r][c1][c2] = grid[r][c1] + (c1 == c2 ? 0 : grid[r][c2]) + maxCherries;
         }
      }
   }
   return dp[0][0][cols - 1];
}

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

```
    cin >> cols;

    vector<vector<int>> grid(rows, vector<int>(cols));
    cout << "Enter the grid values: ";
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            cin >> grid[i][j];
        }
    }

    cout << "Maximum cherries collected: " << cherryPickup(grid) << endl;
    return 0;
}
```

Output:

```
Enter the number of rows: 4
Enter the number of columns: 3
Enter the grid values: 3 1 1
2 5 1
1 5 5
2 1 1
Maximum cherries collected: 21


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

PROBLEM 17

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.

Code:

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

```
int maxDartsInBoard(vector<vector<int>>& darts, int r) {
    int n = darts.size(), maxCount = 0;
    for (int i = 0; i < n; i++) {
        int count = 0;
        for (int j = 0; j < n; j++) {
            if (pow(darts[i][0] - darts[j][0], 2) + pow(darts[i][1] - darts[j][1], 2) <= pow(r, 2)) {
                count++;
            }
        }
        maxCount = max(maxCount, count);
    }
    return maxCount;
}

int main() {
    int n, r;
    cout << "Enter the number of darts: ";
    cin >> n;
    cout << "Enter the radius of the dartboard: ";
    cin >> r;

    vector<vector<int>> darts(n, vector<int>(2));
    cout << "Enter the dart positions: ";
    for (int i = 0; i < n; i++) {
        cin >> darts[i][0] >> darts[i][1];
    }

    cout << "Maximum number of darts in the dartboard: " << maxDartsInBoard(darts, r) << endl;
    return 0;
}
```

Output:

```
Enter the number of darts: 6
Enter the radius of the dartboard: 5
Enter the dart positions: -3 0
3 0
2 6
5 4
0 9
7 8
Maximum number of darts in the dartboard: 4


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

PROBLEM 18

Objective: Design a Skiplist without using any built-in libraries.A skiplist is a data structure that takes O(log(n)) time to add, erase and search. Comparing with treap and red-black tree which has the same function and performance, the code length of Skiplist can be comparatively short and the idea behind Skiplists is just simple linked lists.

Code:

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

class Skiplist {
private:
    struct Node {
        int val;
        vector<Node*> next;
        Node(int value, int level) : val(value), next(level, nullptr) {}
    };

    Node* head;
    int maxLevel;

    int randomLevel() {
        int level = 1;
        while (rand() % 2 && level < maxLevel) {
            level++;
        }
        return level;
    }

public:
    Skiplist() {
        srand(time(0));  // Seed the random number generator
        maxLevel = 16;  // Max level for skiplist
        head = new Node(-1, maxLevel);  // Create the head node
    }

    bool search(int target) {
        Node* curr = head;
        for (int level = maxLevel - 1; level >= 0; level--) {
            while (curr->next[level] && curr->next[level]->val < target) {
                curr = curr->next[level];
            }
        }
        curr = curr->next[0];
        return curr && curr->val == target;
```

```
    }

    void add(int num) {
        vector<Node*> update(maxLevel, nullptr);
        Node* curr = head;

        for (int level = maxLevel - 1; level >= 0; level--) {
            while (curr->next[level] && curr->next[level]->val < num) {
                curr = curr->next[level];
            }
            update[level] = curr;
        }

        int level = randomLevel();
        Node* newNode = new Node(num, level);

        for (int i = 0; i < level; i++) {
            newNode->next[i] = update[i]->next[i];
            update[i]->next[i] = newNode;
        }
    }

    bool erase(int num) {
        vector<Node*> update(maxLevel, nullptr);
        Node* curr = head;

        for (int level = maxLevel - 1; level >= 0; level--) {
            while (curr->next[level] && curr->next[level]->val < num) {
                curr = curr->next[level];
            }
            update[level] = curr;
        }

        curr = curr->next[0];
        if (curr && curr->val == num) {
            for (int i = 0; i < maxLevel; i++) {
                if (update[i]->next[i] == curr) {
                    update[i]->next[i] = curr->next[i];
                }
            }
            delete curr;
            return true;
        }
        return false;
    }
};

int main() {
    Skiplist skiplist;
    skiplist.add(1);
    skiplist.add(2);
```

```
    skiplist.add(3);
    cout << skiplist.search(0) << endl;  // Output: 0 (False)
    skiplist.add(4);
    cout << skiplist.search(1) << endl;  // Output: 1 (True)
    cout << skiplist.erase(0) << endl;   // Output: 0 (False)
    cout << skiplist.erase(1) << endl;   // Output: 1 (True)
    cout << skiplist.search(1) << endl;  // Output: 0 (False)
    return 0;
}
```

Output:

```
0
1
0
1
0


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

PROBLEM 19

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

Code:

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

class AllOne {
private:
    unordered_map<string, int> keyCount;
    unordered_map<int, list<string>> freqMap;
    unordered_map<int, list<string>::iterator> freqListIt;
    int minFreq, maxFreq;

public:
    AllOne() {
        minFreq = maxFreq = 0;
```

```
}

void inc(string key) {
    int count = keyCount[key];
    keyCount[key]++;

    if (count > 0) {
        freqMap[count].erase(freqListIt[count]);
        if (freqMap[count].empty()) {
            if (minFreq == count) minFreq++;
            freqMap.erase(count);
        }
    }

    freqMap[count + 1].push_back(key);
    freqListIt[count + 1] = --freqMap[count + 1].end();

    maxFreq = max(maxFreq, count + 1);
    if (minFreq == 0 || minFreq > count + 1) {
        minFreq = count + 1;
    }
}

void dec(string key) {
    int count = keyCount[key];
    if (count == 0) return;

    freqMap[count].erase(freqListIt[count]);
    if (freqMap[count].empty()) {
        if (minFreq == count) minFreq++;
        freqMap.erase(count);
    }

    keyCount[key]--;

    if (keyCount[key] > 0) {
        freqMap[keyCount[key]].push_back(key);
        freqListIt[keyCount[key]] = --freqMap[keyCount[key]].end();
    }

    if (freqMap[minFreq].empty()) {
        minFreq++;
    }
}

string getMaxKey() {
    if (maxFreq == 0) return "";
    return *freqMap[maxFreq].begin();
}

string getMinKey() {
```

```
        if (minFreq == 0) return "";
        return *freqMap[minFreq].begin();
    }
};

int main() {
    AllOne allOne;
    allOne.inc("hello");
    allOne.inc("hello");
    cout << allOne.getMaxKey() << endl;
    cout << allOne.getMinKey() << endl;
    allOne.inc("leet");
    cout << allOne.getMaxKey() << endl;
    cout << allOne.getMinKey() << endl;
    return 0;
}
```

Output:

```
hello
hello
hello
leet


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