

DEPARTMENT OF

COMPUTER SCIENCE & ENGINEERING

Winter Domain Camp Day 2

Student Name: Karan Yadav

UID: 22BCS11136

Branch: BE-CSE

Section/Group: 22BCS_IOT_639-B

Semester: 5

Date of Performance: 20th Dec, 2024

Problem 1: Majority Element

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 :

```
#include <iostream>
```

```
#include <vector>
```

```
int majorityElement(const std::vector<int>& nums) {  
    int candidate = 0, count = 0;    // Phase 1: Find the  
    for (int num : nums) {          if (count  
        == 0) {                      candidate = num;          count = 1;  
        } else if (num == candidate) {  
            count++;          } else {          count--;  
        }  
    }
```

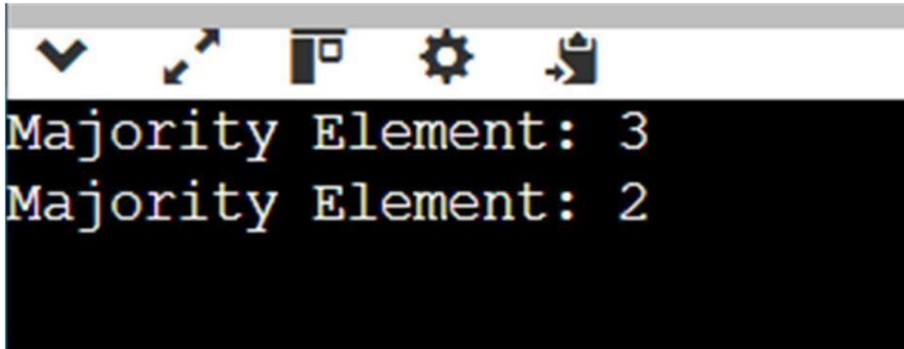
```
    // Phase 2: Verify the candidate (Optional since problem guarantees majority  
    // element)  
    count = 0;    for (int num : nums) {    if (num == candidate) {  
        count++;  
    }  
}
```

```
return candidate;
}

throw std::runtime_error("No majority element found.");
}

int main() {    std::vector<int> nums = {2, 2,
1, 1, 1, 2, 2};    try {        std::cout << "Majority Element: " <<
majorityElement(nums) << std::endl;
    } catch (const std::exception& e) {        std::cerr
<< e.what() << std::endl;
    }
return 0;
}
```

Output:



```
Majority Element: 3
Majority Element: 2
```

Problem 2: Pascal's Triangle

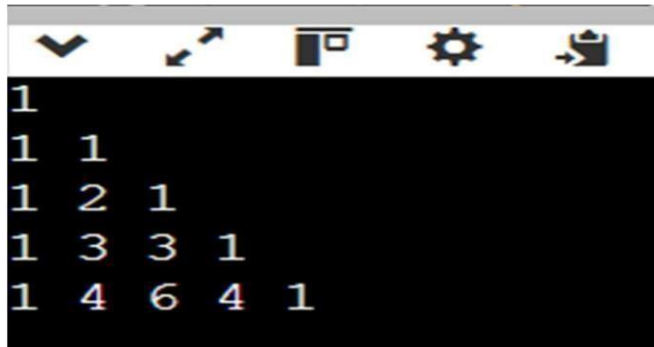
Given an integer numRows, return the first numRows of Pascal's triangle. In Pascal's triangle, each number is the sum of the two numbers directly above it

Solution:

```
#include <iostream>
#include <vector>
std::vector<std::vector<int>> generate(int numRows) {
std::vector<std::vector<int>> triangle(numRows);    for (int i = 0; i < numRows;
++i)
```

```
{    triangle[i].resize(i + 1, 1);    for (int j = 1; j < i; ++j) {        triangle[i][j]
= triangle[i - 1][j - 1] + triangle[i - 1][j];    }}    return triangle;} int main() {
int numRows = 5; // Example input    auto result = generate(numRows);    for
(const auto& row : result) {        for (int num : row) std::cout << num << " ";
std::cout << "\n";}    return 0;}
```

Output:



```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

Problem 3: 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>
```

```
#include <vector>
```

```
int singleNumber(const std::vector<int>& nums)
```

```
{    int result = 0;    for (int num : nums) {
```

```
result ^= num;
```

```
    }    return result;
```

```
}
```

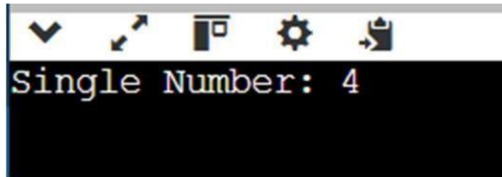
```
int main() {    std::vector<int> nums = {4, 1, 2, 1, 2}; // Example input
```

```
std::cout << "Single Number: " << singleNumber(nums) << std::endl;    return
```

```
0;
```

```
}
```

Output:



Problem 4: Merge Two Sorted Lists

You are given the heads of two sorted linked lists list1 and list2. Merge the two lists into one sorted list. The list should be made by splicing together the nodes of the first two lists. Return the head of the merged linked list.

Solution:

```
#include <iostream>
```

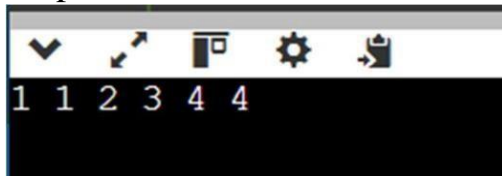
```
struct ListNode {  
    int val;  
    ListNode* next;  
    ListNode(int x) : val(x), next(nullptr) {}  
};
```

```
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;  
}
```

```
void printList(ListNode* head) {  
    while (head) {        std::cout << head-&br/>>val << " ";        head = head->next;  
    }
```

```
    }  
}  
  
int main() {  
    ListNode* list1 = new ListNode(1);    list1->next  
= new ListNode(2);    list1->next->next = new  
ListNode(4);  
  
    ListNode* list2 = new ListNode(1);    list2->next  
= new ListNode(3);    list2->next->next = new  
ListNode(4);  
  
    ListNode* mergedList = mergeTwoLists(list1, list2);    printList(mergedList);  
  
    return 0;  
}
```

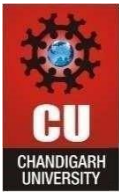
Output:



Problem 5: Linked List Cycle.

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



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

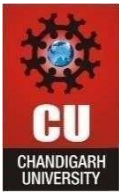
as a parameter. Return true if there is a cycle in the linked list. Otherwise, return false.

Solution:

```
#include <iostream>
```

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

```
bool hasCycle(ListNode* head) {  
    if (!head) return false;  
    ListNode *slow = head, *fast = head;
```



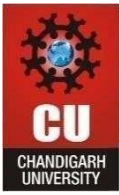
DEPARTMENT OF

Discover. Learn. Empower.

COMPUTER SCIENCE & ENGINEERING

```
while (fast && fast->next) {      slow = slow->next;
// Move slow pointer by 1 step
fast = fast->next->next;    // Move fast pointer by 2 steps

    if (slow == fast) {          // Cycle
detected      return true;}}    return false;
// No cycle} int main() {
    // Example 1: Creating a cycle in the list    ListNode* head = new
ListNode(3);    head->next = new ListNode(2);    head->next->next = new
ListNode(0);    head->next->next->next = new ListNode(-4);
head->next->next->next->next = head->next; // Cycle starts at node with
value 2
```

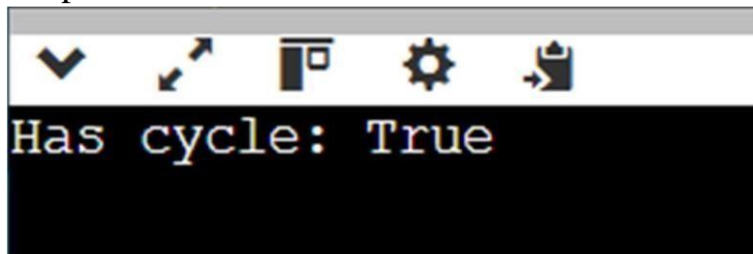


DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

COMPUTER SCIENCE & ENGINEERING

```
std::cout << "Has cycle: " << (hasCycle(head) ? "True" : "False") << std::endl;  
  
return 0;  
}
```

Output:



Problem 6: 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`. 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>
```

```
int removeDuplicates(std::vector<int>& nums) {
```

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

```
    int k = 1; // Pointer for the next unique element    for (int i = 1; i <
nums.size(); ++i) {        if (nums[i] != nums[i - 1]) {            nums[k++] =
nums[i]; // Move the unique element to the front}}    return k; //
```

```
Number of unique elements} int main() {    std::vector<int> nums =
{0,0,1,1,1,2,2,3,3,4};    int k = removeDuplicates(nums);
```

```
    std::cout << "k = " << k << std::endl;
```

```
std::cout << "Modified array: ";    for
```

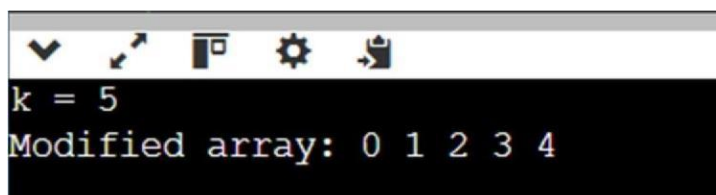
```
(int i = 0; i < k; ++i) {
```

```
std::cout << nums[i] << " ";
```

```
    }    std::cout <<
```

```
std::endl;    return 0;}
```

Output:



```
k = 5
Modified array: 0 1 2 3 4
```

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.

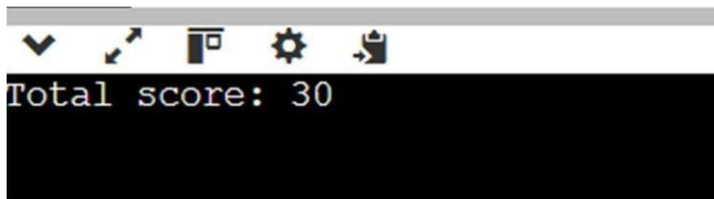
Solution:

```
#include <iostream>
#include <vector>
#include <string>

int calPoints(std::vector<std::string>& ops) {
    std::vector<int> record;    for (const auto& op
: ops) {
        if (op == "C") {        record.pop_back(); // Remove
the last score
        } else if (op == "D") {        record.push_back(2 *
record.back()); // Double the last score
        } else if (op == "+") {
            record.push_back(record[record.size() - 1] + record[record.size() - 2]); //
Sum of last two scores
        } else {
            record.push_back(std::stoi(op)); // Add the integer score
        }
    }

    int total = 0;    for (int
score : record) {        total
+= score;
```

```
    }  
  
    return total;  
}  
  
int main() {  
    std::vector<std::string> ops = {"5","2","C","D","+"};  
    std::cout << "Total score: " << calPoints(ops) << std::endl; // Output: 30  
    return 0; }Output:
```



Problem 8: 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 `i`th 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>
```

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

```
int h = std::min(height[left], height[right]);  
maxArea = std::max(maxArea, width * h);
```

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

```
int main() { std::vector<int> height = {1,8,6,2,5,4,8,3,7}; std::cout <<  
"Max area: " << maxArea(height) << std::endl; // Output: 49 return 0; }  
Output:
```



```
Max area: 49
```

Problem 9: 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: $0 \leq j \leq \text{nums}[i]$ 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]`.

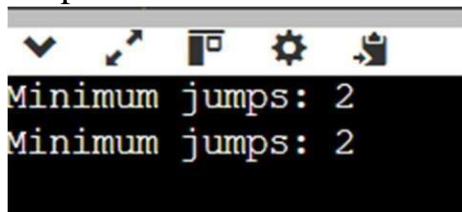
Solution:

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

```
int jump(std::vector<int>& nums) {  
    int n = nums.size(); int jumps = 0,  
    farthest = 0, currentEnd = 0;  
  
    for (int i = 0; i < n - 1; ++i) {        farthest = std::max(farthest, i + nums[i]); //  
Update the farthest point we can reach  
  
        if (i == currentEnd) {  
            jumps++;           // We make a jump when we reach the current end  
currentEnd = farthest; // Move to the farthest point we can reach  
  
            if (currentEnd >= n - 1) break; // If we can reach the end, stop  
        }  
    }  
  
    return jumps;  
}
```

```
int main() {    std::vector<int> nums1 = {2,3,1,1,4};    std::cout <<  
"Minimum jumps: " << jump(nums1) << std::endl; // Output: 2  
  
    std::vector<int> nums2 = {2,3,0,1,4};    std::cout << "Minimum jumps: " <<  
jump(nums2) << std::endl; // Output: 2  
  
    return 0;  
}
```

Output:



```
Minimum jumps: 2  
Minimum jumps: 2
```

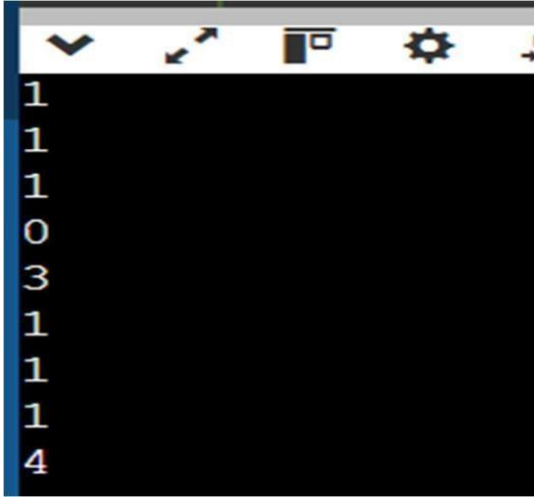
Problem 10: Design Circular Queue

Calculate the sum of the digits of a given number n. For example, for the number 12345, the sum of the digits is $1+2+3+4+5=15$. To solve this, you will need to extract each digit from the number and calculate the total sum.

Solution:

```
#include <iostream>
#include <vector>
class MyCircularQueue { private:
    std::vector<int> queue;
    int front, rear, size, capacity;
public:
    MyCircularQueue(int k) : queue(k), front(-1), rear(-1), size(0), capacity(k) {}
    bool enqueue(int value) { if (size == capacity) return false; if (size == 0) front = 0; rear = (rear + 1) % capacity; queue[rear] = value; size++; return true; }
    bool dequeue() { if (size == 0) return false; if (front == rear) front = rear = -1; else front = (front + 1) % capacity; size--; return true; }
    int Front() { return size == 0 ? -1 : queue[front]; }
    int Rear() { return size == 0 ? -1 : queue[rear]; }
    bool isEmpty() { return size == 0; }
    bool isFull() { return size == capacity; } };
int main() { MyCircularQueue q(3); std::cout << q.enqueue(1) << std::endl; // True std::cout << q.enqueue(2) << std::endl; // True std::cout << q.enqueue(3) << std::endl; // True std::cout << q.enqueue(4) << std::endl; // False (full) std::cout << q.Rear() << std::endl; // 3 std::cout << q.isFull() << std::endl; // True std::cout << q.dequeue() << std::endl; // True std::cout <<
```

```
q.enqueue(4) << std::endl; // True    std::cout <<
q.Rear() << std::endl;    // 4
return 0;} Output:
```



Problem 11: 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> using
namespace std;
```

```
class Solution { public: 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, -1))); return
dfs(0, 0, cols - 1, grid, dp);
}
```

```
private: int dfs(int row, int col1, int col2,
vector<vector<int>>& grid, vector<vector<vector<int>>>&
dp) {
    int rows = grid.size(), cols = grid[0].size(); if (col1 < 0
|| col2 < 0 || col1 >= cols || col2 >= cols) return 0;
if (dp[row][col1][col2] != -1) return dp[row][col1][col2];

    int cherries = grid[row][col1]; if (col1
!= col2) cherries += grid[row][col2]; if (row
< rows - 1) { int maxCherries = 0;
for (int d1 = -1; d1 <= 1; d1++) {
    for (int d2 = -1; d2 <= 1; d2++) { maxCherries =
max(maxCherries, dfs(row + 1, col1 + d1, col2 + d2, grid, dp));
    }
}
    cherries += maxCherries;
}
return dp[row][col1][col2] = cherries;
}
};
```

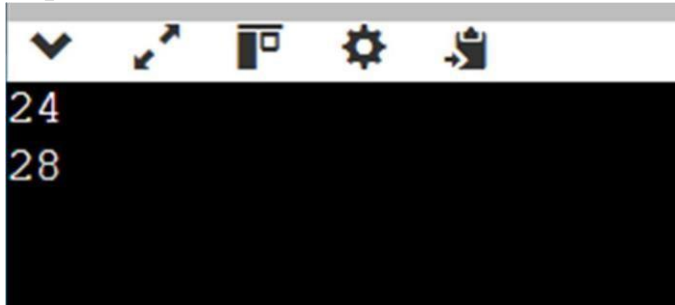
```
int main() { Solution sol; vector<vector<int>> grid1 =
{{3,1,1},{2,5,1},{1,5,5},{2,1,1}};
cout << sol.cherryPickup(grid1) << endl; // Output: 24
```

```
vector<vector<int>> grid2 =
```



```
{{1,0,0,0,0,0,1},{2,0,0,0,0,3,0},{2,0,9,0,0,0,0},{0,3,0,5,4,0,0},{1,0,2,3,0,0,6}};
cout << sol.cherryPickup(grid2) << endl; // Output: 28    return 0; }
```

Output:



Problem 12: 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 i th 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> using
```

```
namespace std;
```

```
class Solution { public:    int numPoints(vector<vector<int>>& darts,
int r) {        int maxCount = 1, n = darts.size();        for (int i = 0; i <
n; ++i) {
for (int j = i + 1; j < n; ++j) {                double dx = darts[j][0] - darts[i][0],
dy = darts[j][1] - darts[i][1];                double dist = sqrt(dx * dx + dy *
dy);
```

```

        if (dist > 2 * r) continue;
        double midX = (darts[i][0] +
darts[j][0]) / 2.0, midY = (darts[i][1] + darts[j][1]) / 2.0;
        double angle =
sqrt(r * r - (dist / 2) * (dist / 2)), norm = dist ? r / dist :
0;
        maxCount = max(maxCount, count(darts, midX - norm * dy, midY +
norm * dx, r));
    }
}
return maxCount;
}

```

```

private: int count(const vector<vector<int>>& darts, double cx, double
cy, int r) {
int c = 0;
    for (auto& dart : darts)
        if (pow(dart[0] - cx, 2) + pow(dart[1] -
cy, 2) <= r * r + 1e-7) ++c;
    return c;
}
};

```

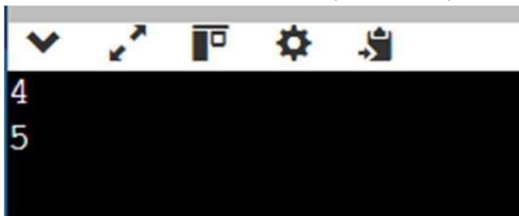
```
int main() {
```

```
Solution sol;
```

```

    vector<vector<int>> darts = {{-2, 0}, {2, 0}, {0, 2}, {0, -2}};
    cout << sol.numPoints(darts, 2) << endl; // Output: 4 }

```



Problem 13: Design Skiplist

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 redblack 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.

Solution:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <vector>
```

```
using namespace std;
```

```
// Node structure for Skiplist struct Node {    int value;    vector<Node*>
forward; // Pointers to next nodes at each level
```

```
    Node(int value, int level) : value(value), forward(level, nullptr) { } };
```

```
// Skiplist class class Skiplist { private:    int maxLevel;        // Maximum
level for the skiplist    float probability;    // Probability of promoting a
node to a higher level    Node* header;        // Header node
```

```
    // Random level generator function
int randomLevel() {        int level = 1;
    while ((rand() % 2) < probability && level < maxLevel) {
level++;
    }
    return level;
}
```

```
public:
```

```
    Skiplist(int maxLevel = 16, float probability = 0.5) :
maxLevel(maxLevel), probability(probability) {        header = new Node(-
1, maxLevel); // Header node with a dummy value
    }
```

```
// Search for a value in the Skiplist
bool search(int target) {      Node* current
= header;
    for (int i = maxLevel - 1; i >= 0; --i) {      while (current->forward[i] !=
nullptr && current->forward[i]->value < target) {      current = current-
>forward[i];
        }
    }
    current = current->forward[0];      return (current !=
nullptr && current->value == target);
}

// Insert a value into the Skiplist void
insert(int value) {      vector<Node*>
update(maxLevel, nullptr);
    Node* current = header;

    for (int i = maxLevel - 1; i >= 0; --i) {      while (current->forward[i] !=
nullptr && current->forward[i]->value < value) {      current = current-
>forward[i];
        }
        update[i] = current;
    }

    current = current->forward[0];      if (current ==
nullptr || current->value != value) {      int level =
randomLevel();
        Node* newNode = new Node(value, level);

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

```
// Erase a value from the Skiplist void
erase(int value) {
    vector<Node*>
    update(maxLevel, nullptr);
    Node* current = header;

    for (int i = maxLevel - 1; i >= 0; --i) {
        while (current->forward[i] !=
        nullptr && current->forward[i]->value < value) {
            current = current-
            >forward[i];
        }
        update[i]
        = current;
    }

    current = current->forward[0];    if (current !=
    nullptr && current->value == value) {        for (int i =
    0; i < maxLevel; ++i) {            if (update[i]>forward[i]
    != current) break;                update[i]-
    >forward[i] = current->forward[i];
        }
    delete current;
    }
}

// Print the Skiplist void
print() {
    for (int i = 0; i < maxLevel; ++i) {
        Node* current = header->forward[i];
        cout << "Level " << i << ": ";        while
        (current != nullptr) {            cout <<
        current->value << " ";            current =
        current->forward[i];
        }
        cout << endl;
    }
}
```

```
};

int main() {    srand(time(0)); // Seed for random
level generation

    Skiplist skiplist;

    // Insert values into the Skiplist    skiplist.insert(30);
    skiplist.insert(40);    skiplist.insert(50);
    skiplist.insert(60);    skiplist.insert(70);
    skiplist.insert(90);

    // Print the Skiplist
    cout << "Skiplist after insertions:" << endl;
    skiplist.print();

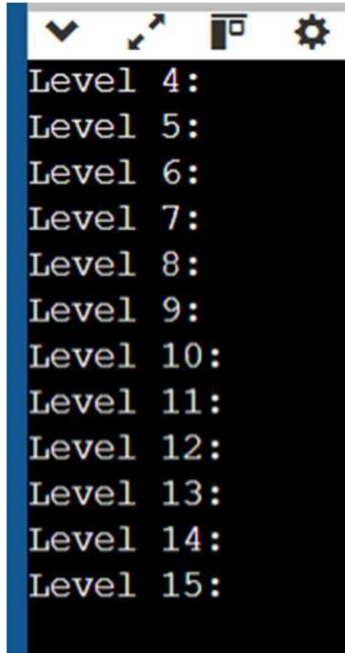
    // Insert additional values    skiplist.insert(80);
    skiplist.insert(45);

    cout << "Skiplist after adding 80 and 45:" << endl;
    skiplist.print();

    // Search for some values    cout << "Searching for 45: " <<
    (skiplist.search(45) ? "Found" : "Not Found")
    << endl;
    cout << "Searching for 100: " << (skiplist.search(100) ? "Found" : "Not
    Found") << endl;

    // Erase a value    skiplist.erase(50);    cout <<
    "Skiplist after removing 50:" << endl;
    skiplist.print();

    return 0; }Output:
```



Problem 14: 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 "".

Solution:

```
#include <iostream>
#include <unordered_map>
#include <string>
```

```
#include <list> #include
<map> using
namespace std; class
AllOne { private:
    // Hash map to store the count of each key    unordered_map<string,
int> keyCount;

    // Map to store the strings by their counts, ordered by counts    map<int,
list<string>> countKeys; public:
    AllOne() {}
    // Increment the count of a string by 1    void
inc(string key) {        int count =
keyCount[key];
    // Remove the key from its old count list
if (count > 0) {
countKeys[count].remove(key);
if (countKeys[count].empty()) {        countKeys.erase(count);} }
    // Increment the count and add the key to the new count list
keyCount[key] = count + 1;        countKeys[count +
1].push_back(key);} // Decrement the count of a string by 1
void dec(string key) {        int count = keyCount[key];        //
Remove the key from its current count list
countKeys[count].remove(key);        if
(countKeys[count].empty()) {        countKeys.erase(count);
}

    // If count becomes 0, remove the key from keyCount        if
(count == 1) {        keyCount.erase(key);
        } else {        keyCount[key] = count -
1;        countKeys[count -
1].push_back(key);} } // Get the key with the
maximum count    string getMaxKey() {
    if (countKeys.empty()) return "";        //
Get the last element (maximum count)
return countKeys.rbegin()->second.front();
}
```



```
// Get the key with the minimum count    string
getMinKey() {
    if (countKeys.empty()) return "";    //
    Get the first element (minimum count)
    return countKeys.begin()->second.front();    }
};

int main() {
    AllOne allOne;

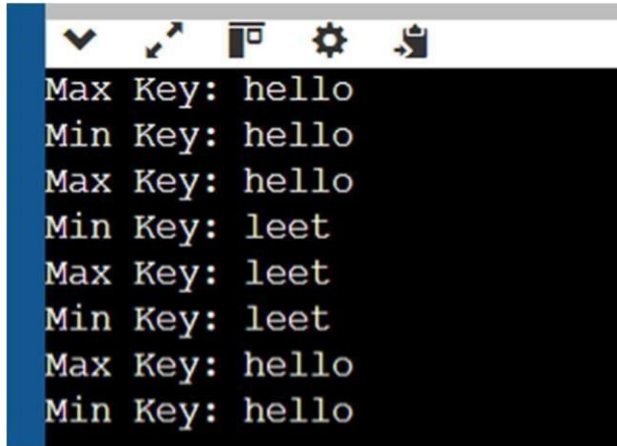
    allOne.inc("hello");    allOne.inc("hello");    cout << "Max
Key: " << allOne.getMaxKey() << endl; // "hello"    cout << "Min
Key: " << allOne.getMinKey() << endl; // "hello"

    allOne.inc("leet");
    cout << "Max Key: " << allOne.getMaxKey() << endl; // "hello"    cout
<< "Min Key: " << allOne.getMinKey() << endl; // "leet"

    allOne.dec("hello");    cout << "Max Key: " << allOne.getMaxKey()
<< endl; // "hello"    cout << "Min Key: " << allOne.getMinKey() <<
endl; // "leet"

    allOne.dec("leet");    cout << "Max Key: " <<
allOne.getMaxKey() << endl; // "hello"    cout << "Min Key: " <<
allOne.getMinKey() << endl; // "hello"

    return 0;
}
Output:
```



```
Max Key: hello
Min Key: hello
Max Key: hello
Min Key: leet
Max Key: leet
Min Key: leet
Max Key: hello
Min Key: hello
```

Problem 15: 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 `i`th 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;
```

```
// Helper function to check if we can distribute jobs with max workload <= mid
bool canAssignJobs(const vector<int>& jobs, int k, int mid) {    int currentSum
= 0;    int workersUsed = 1; // Start with one worker
```

```
    for (int job : jobs) {
        if (currentSum + job > mid) {
            // Need a new worker since current worker cannot take this job
            workersUsed++;    currentSum = job;    if (workersUsed > k)
            return false; // More workers than allowed
        } else {
            currentSum += job;
        }
    }
    return true;
}
```

```
int minimumTimeRequired(vector<int>& jobs, int k) {    int left =
*max_element(jobs.begin(), jobs.end()); // Max job time    int right =
accumulate(jobs.begin(), jobs.end(), 0); // Sum of all jobs
```

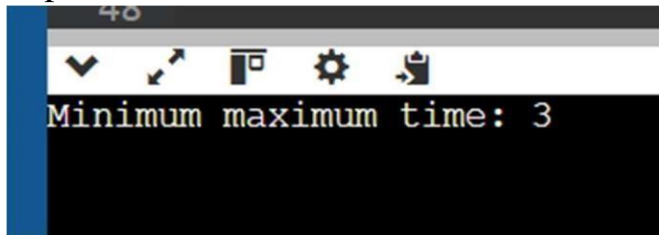
```
    while (left < right) {        int mid
= left + (right - left) / 2;
        if (canAssignJobs(jobs, k, mid)) {            right =
mid; // Try for a smaller max workload
        } else {            left = mid + 1; // Increase the allowed
max workload
        }
    }

    return left;
}
```

```
int main() {    vector<int>
jobs = {3, 2, 3};    int k = 3;
```

```
cout << "Minimum maximum time: " << minimumTimeRequired(jobs, k) <<  
endl; return 0;  
}
```

Output:



Problem 16: 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 i th 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 <unordered_set>  
#include <unordered_map>
```

```
using namespace std;
```

```
class UnionFind { public:
    UnionFind(int n) {        parent.resize(n);
    size.resize(n, 1);        for (int i = 0; i < n;
    ++i) parent[i] = i;
    }

    int find(int u) {        if (parent[u] != u) {        parent[u]
    = find(parent[u]); // Path compression
    }
    return parent[u];
    }

    void unionSets(int u, int v) {        int rootU = find(u);
    int rootV = find(v);        if (rootU != rootV) {        if
    (size[rootU] < size[rootV]) swap(rootU, rootV);
    parent[rootV] = rootU;        size[rootU] +=
    size[rootV];
    }
    }

private:
    vector<int>    parent;
    vector<int> size;
};

int minimumTeachings(int n, vector<vector<int>>& languages,
vector<vector<int>>& friendships) {    int m = languages.size();
// Number of users

    // Step 1: Union-Find initialization for users
    UnionFind uf(m);
```

```
// Step 2: Union users who share a common language
unordered_map<int, vector<int>> languageUsers;    for
(int i = 0; i < m; ++i) {        for (int
lang : languages[i]) {        languageUsers[lang].push_back(i);
    }
}
```

```
// Union users based on shared languages
for (auto& [lang, users] : languageUsers) {
    for (int i = 0; i < users.size(); ++i) {        for
        (int j = i + 1; j < users.size(); ++j) {
            uf.unionSets(users[i], users[j]);
        }
    }
}
```

```
// Step 3: Union users based on friendships    for
(auto& friendship : friendships) {
    int u = friendship[0] - 1;
    int v = friendship[1] - 1;    if (uf.find(u)
    != uf.find(v)) {        uf.unionSets(u, v);
    }
}
```

```
// Step 4: Find the connected components and check if they can communicate
unordered_map<int, unordered_set<int>> components;    for (int i = 0; i < m;
++i) {    int root = uf.find(i);    for (int lang : languages[i]) {
components[root].insert(lang);
    }
}
```

```
// Step 5: Determine how many users need to be taught a new language    int
result = 0;

for (auto& [component, langs] : components) {
    // If no language is shared among users in this component, we need to teach
    at least one user    if (langs.empty()) {        result++;
    }
}
```

```
    }

    return result;
}

int main() {    vector<vector<int>> languages1 = {{1}, {2},
{1, 2}};    vector<vector<int>> friendships1 = {{1, 2}, {1,
3}, {2, 3}};    int n1 = 2;    cout << "Minimum number of users to teach: "
<< minimumTeachings(n1,
languages1, friendships1) << endl;

    vector<vector<int>> languages2 = {{2}, {1, 3}, {1, 2}, {3}};
vector<vector<int>> friendships2 = {{1, 4}, {1, 2}, {3, 4}, {2, 3}};    int
n2 = 3;

    cout << "Minimum number of users to teach: " << minimumTeachings(n2,
languages2, friendships2) << endl;

    return 0;
}
```

Output:

Output

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

Problem 17: 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 $10^9 + 7$. 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 <cmath>
```

```
const int MOD = 1e9 + 7;
```

```
const int MAXN = 10000;
```

```
// Precompute factorials and modular inverses using Fermat's Little Theorem  
std::vector<long long> factorial(MAXN + 1), inv_factorial(MAXN + 1);
```

```
// Function to compute  $x^y \% MOD$ 
```

```
long long mod_exp(long long x, long long y, long long mod) {  
    long long result = 1;    while (y > 0) {        if (y % 2 == 1) result  
= (result * x) % mod;        x = (x * x) % mod;  
        y /= 2;  
    }  
    return result;  
}
```

```
// Function to precompute factorials and their inverses void precompute()  
{
```

```
    factorial[0] = inv_factorial[0] = 1;
```

```
    for (int i = 1; i <= MAXN; ++i) {
```

```
        factorial[i] = (factorial[i - 1] * i) % MOD;
```



```
    }  
    inv_factorial[MAXN] = mod_exp(factorial[MAXN], MOD - 2, MOD); //  
    Using Fermat's little theorem    for (int i = MAXN - 1; i >= 1; --i) {  
    inv_factorial[i] = (inv_factorial[i + 1] * (i + 1)) % MOD;  
    }  
}
```

```
// Function to compute binomial coefficient C(n, k) % MOD  
long long binomial(int n, int k) {    if (k > n || k < 0) return 0;    return  
(factorial[n] * inv_factorial[k] % MOD) * inv_factorial[n - k] % MOD; }
```

```
// Function to get the prime factorization of a number  
std::vector<std::pair<int, int>> prime_factors(int k) {  
    std::vector<std::pair<int, int>> factors;    for (int i = 2;  
    i * i <= k; ++i) {        if (k % i == 0) {            int  
    count = 0;                while (k % i == 0) {                k /= i;  
    count++;  
        }  
        factors.push_back({i, count});  
    }        }        if (k > 1) {  
    factors.push_back({k, 1});  
    }    return  
    factors;  
}
```

```
// Function to solve each query long  
long solve(int n, int k) {  
    // Prime factorize k    auto  
    factors = prime_factors(k);
```

```
    long long result = 1;    for (const  
    auto& factor : factors) {        int  
    prime = factor.first;        int exponent  
    = factor.second;
```

```
        // Calculate number of ways to split exponent of this prime into n parts
result = (result * binomial(exponent + n - 1, n - 1)) % MOD;
    }
```

```
    return result;
}
```

```
std::vector<int> waysToPlaceIntegers(std::vector<std::vector<int>>& queries) {
    precompute(); // Precompute factorials and inverses    std::vector<int> result;
```

```
    for (auto& query : queries) {
        int n = query[0];    int k =
        query[1];    result.push_back(solve(n,
        k));
    }
```

```
    return result;
}
```

```
int main() {    //
    Example usage    std::vector<std::vector<int>> queries1 = {{2, 6}, {5,
    1}, {73, 660}};
```

```
    std::vector<int> result1 = waysToPlaceIntegers(queries1);
```

```
        for (int r :
    result1) {
        std::cout << r << " ";
    }
```

```
    std::cout << std::endl;
```

```
    std::vector<std::vector<int>> queries2 = {{1, 1}, {2, 2}, {3, 3}, {4, 4}, {5,
    5}};
```

```
    std::vector<int> result2 = waysToPlaceIntegers(queries2);
```

```
        for (int r :
    result2) {
```

```
std::cout << r << " ";  
}  
std::cout << std::endl;  
  
return 0;  
}
```

Output:

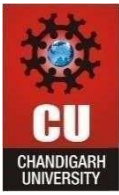
Output
4 1 50734910
1 2 3 10 5

Problem 18: Maximum Twin Sum of a Linked List

In a linked list of size n , where n is even, the i th node (0-indexed) of the linked list is known as the twin of the $(n-1-i)$ th node, if $0 \leq i \leq (n/2) - 1$. • 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.

Code:

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



DEPARTMENT OF

COMPUTER SCIENCE & ENGINEERING

```
using namespace std;
```

```
// Definition for singly-linked list.
```

```
struct ListNode {    int
```

```
val;
```

```
    ListNode* next;
```

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

```
class Solution { public: int
```

```
pairSum(ListNode* head) { //
```

```
Step 1: Store the values of the  
linked list in a vector
```

```
vector<int> values;
```

```
    ListNode* current = head;
```

```
    // Traverse the linked list and add each node's value to the vector  
while (current != nullptr) {        values.push_back(current->val);
```

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

```
}
```

```
    // Step 2: Calculate the maximum twin sum
```

```
int n = values.size();    int maxTwinSum =
```

```
0;
```

```
    // Iterate over the first half of the list and calculate twin sums
```

```
for (int i = 0; i < n / 2; ++i) {        int twinSum = values[i] +
```

```
values[n - 1 - i];        maxTwinSum = max(maxTwinSum,
```

```
twinSum);
```

```
}
```

```
    return maxTwinSum;
```

```
}
```

```
};
```

```
// Helper function to create a linked list from a vector
ListNode* createList(const vector<int>& nums) {
    ListNode* head = new ListNode(nums[0]);
    ListNode* current = head; for (int i = 1; i <
nums.size(); ++i) { current->next = new
ListNode(nums[i]);
        current = current->next;
    }
    return head;
}

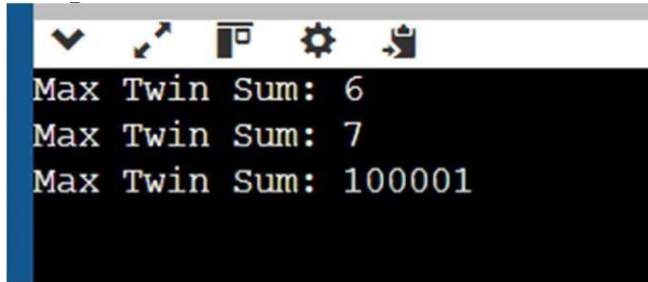
int main() { // Example 1
vector<int> input1 = {5, 4, 2, 1};
    ListNode* head1 = createList(input1);
    Solution sol;
    cout << "Max Twin Sum: " << sol.pairSum(head1) << endl; // Output: 6

    // Example 2
    vector<int> input2 = {4, 2, 2, 3}; ListNode* head2 =
createList(input2); cout << "Max Twin Sum: " << sol.pairSum(head2)
<< endl; // Output: 7

    // Example 3
    vector<int> input3 = {1, 100000};
    ListNode* head3 = createList(input3); cout << "Max Twin Sum: " <<
sol.pairSum(head3) << endl; // Output:
100001

    return 0;
}
```

Output:



```
Max Twin Sum: 6
Max Twin Sum: 7
Max Twin Sum: 100001
```

Problem 19: Insert Greatest Common Divisors in Linked List

Given the head of a linked list head, in which each node contains an integer value. Between every pair of adjacent nodes, insert a new node with a value equal to the greatest common divisor of them. Return the linked list after insertion. The greatest common divisor of two numbers is the largest positive integer that evenly divides both numbers. Solution:

```
#include <iostream>
#include <vector> // Include the vector header
#include <algorithm>
using namespace std;

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

class Solution { public:
    // Function to compute GCD of two numbers
    int gcd(int a, int b) {
        while (b != 0) {
            int temp = b;
            b = a % b;
            a = temp;
        }
        return a;
    }
};
```

```
}  
// Function to insert GCD nodes between each pair of adjacent nodes  
ListNode* insertGreatestCommonDivisors(ListNode* head) {  
    // Edge case: If the list is empty or has only one node, no insertion is needed  
    if (!head || !head->next) return head;  
  
    ListNode* current = head;  
  
    // Traverse the list  
    while (current && current->next) {  
        int gcdValue = gcd(current->val,  
current->next->val); // Calculate the  
GCD  
        ListNode* newNode = new ListNode(gcdValue); // Create a new node  
with the GCD value  
        newNode->next = current->next; // Link the new  
node to the next node  
        current->next = newNode; // Link the current  
node to the new node  
        current = newNode->next; // Move to the next pair  
of nodes  
    }  
  
    return head;  
}  
};  
  
// Helper function to create a linked list from a vector  
ListNode* createList(const vector<int>& values) {  
    if (values.empty()) return nullptr;  
    ListNode* head = new ListNode(values[0]);  
    ListNode* current = head;    for (int i = 1; i <  
values.size(); ++i) {        current->next = new  
ListNode(values[i]);  
        current = current->next;  
    }  
    return head;  
}
```

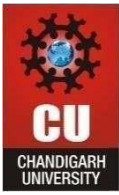
```
// Helper function to print the linked list
void printList(ListNode* head) {
    while (head) {        cout << head->val;
        if (head->next) cout << " -> ";
        head = head-
        >next;
    }
    cout << endl;
}

int main() {
    Solution sol;

    // Test case 1    vector<int> values1
    = {18, 6, 10, 3};
    ListNode* head1 = createList(values1);
    ListNode* result1 = sol.insertGreatestCommonDivisors(head1);
    printList(result1); // Expected: 18 -> 6 -> 6 -> 2 -> 10 -> 1 -> 3

    // Test case 2
    vector<int> values2 = {7};
    ListNode* head2 = createList(values2);
    ListNode* result2 = sol.insertGreatestCommonDivisors(head2);
    printList(result2); // Expected: 7

    return 0;
}
```

DEPARTMENT OF

Discover. Learn. Empower.

COMPUTER SCIENCE & ENGINEERING

}

Output:

Output

18 -> 6 -> 6 -> 2 -> 10 -> 1 -> 3
7