DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

Discover. Learn. Empower.

DOMAIN WINTER WINNING CAMP

Student Name: Ishan Jain **UID:22BCS13653**

Branch: CSE **Section: IOT-615(B)**

Semester: 5th **Date of Performance: 20/12/24**

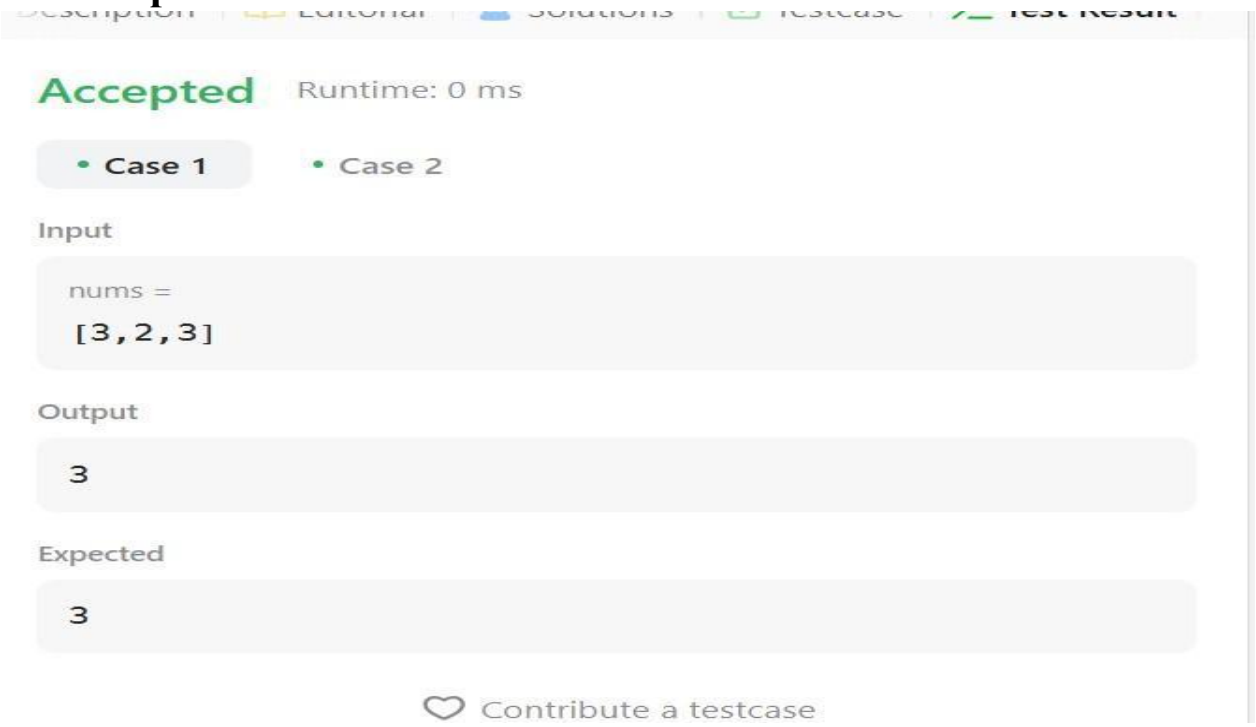
1. **Aim**: Given an array nums of size n, return the majority element. The majority element is the element that appears more than ⌊n / 2⌋ times. You may assume that the majority element always exists in the array.

Code:

```
class Solution { public:
int
majorityElement
(vector<int>& nums)
{       int count =
0;
    int candidate = 0;

    for (int num : nums) {
if (count == 0) {
        candidate = num;
      }
      count += (num == candidate) ? 1 : -1;
    }

    return candidate;
```

```
    } };
```

Output:



2. **Aim**: 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:

Output:

```
class Solution {
public:
int singleNumber(vector<int>& nums) {
unordered_map<int,int> mp; for(int i:nums){ mp[i]++;
}
```

```
        for(auto                    m:mp){
if(m.second==1)
          {          return
m.first;
          }
        }
```



```
        return -1;

    } };
```

OUTPUT

3. **Aim**: 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>
#include <queue> #include
<climits>          using
namespace std;

struct TreeNode { int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {} };

TreeNode* helper(const 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 =
    helper(nums, left, mid - 1); root-
    >right = helper(nums, mid + 1, right); return root;
}

TreeNode* sortedArrayToBST(const vector<int>& nums) { return
helper(nums, 0, nums.size() - 1); }

vector<int> levelOrder(TreeNode* root) { if
    (!root) return {};   vector<int> result;
    queue<TreeNode*> q;
```

```
    q.push(root); while
    (!q.empty()) {
       TreeNode*  node  =  q.front();  q.pop();
            if      (node)        {
       result.push_back(node->val); q.push(node->left);
          q.push(node->right);
       }  else      { result.push_back(INT_MIN);
       }
    }
    while (!result.empty() && result.back() == INT_MIN) result.pop_back();
    for (int& val : result) if (val == INT_MIN) val = -1; return result;
}

int main() { vector<int> nums1 = {-10, 3,
   0, 5, 9}; vector<int> nums2 = {1, 3};

    TreeNode* root1 = sortedArrayToBST(nums1);
    TreeNode* root2 = sortedArrayToBST(nums2);
    vector<int>   result1        =
     levelOrder(root1); vector<int> result2 =
    levelOrder(root2); for (int x : result1) cout << x
    << " "; cout << endl;
    for (int x : result2) cout << x << " ";
    cout << endl; return
    0;
}
```

Output:

```
0 -10 5 -1 -3 -1 9
1 -1 3


...Program finished with exit code 0
Press ENTER to exit console.
```

4. Aim: 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..

Code:

```
class Solution
{ public:
   ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
   ListNode*  dummy  =  new  ListNode(0);
ListNode* current = dummy;

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

       return dummy->next;
   }
};
```

5. Aim: 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.

Code:

```
class Solution { public:
bool hasCycle(ListNode *head) {
        ListNode*fastptr=head;        ListNode*slowptr=head;
while(slowptr!=nullptr  && fastptr!=nullptr && fastptr->next!=nullptr &&
slowptr->next!=nullptr)
        {          slowptr= slowptr-
>next;
fastptr=fastptr->next->next;          if(fastptr==slowptr){
            return true ;
          }
        }
        return false;

    } };
```

Output:

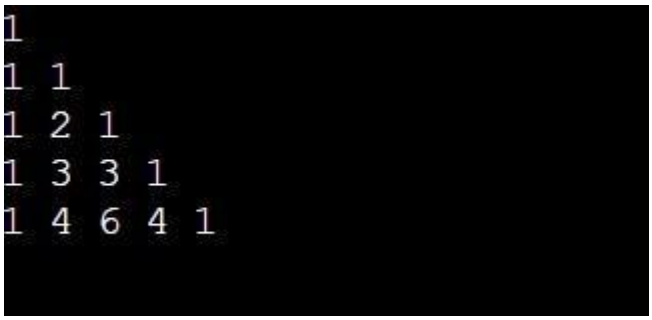

Accepted
Runtime: 2 ms
Case 1
Case 2
Case 3
Input
head =
[3,2,0,-4]
pos =
1
Output
true
Expected
true

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

Code:

```
#include <vector> #include
<iostream>  std::vector<std::vector<int>>          generate(int
    numRows)   { std::vector<std::vector<int>> triangle; for (int
i = 0; i < numRows; ++i) { std::vector<int> row(i + 1, 1); //
Initialize a row with all 1's 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 = 5; std::vector<std::vector<int>> result
   = generate(numRows);

   for (const auto& row : result) { for
      (int num : row) {
      std::cout << num << " ";
      }         std::cout     <<
      std::endl;
   } return
   0;
}
```

Output:



7. **Aim**: 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 <vector> #include
<iostream>

int removeDuplicates(std::vector<int>& nums) { if
   (nums.empty()) return 0;  int
   k = 1;
   for (int i = 1; i < nums.size(); ++i) {
      if  (nums[i]  !=  nums[i  -  1])  {
      nums[k] = nums[i];  ++k;
      }
   } return k;
} int main() { std::vector<int> nums = {0,0,1,1,1,2,2,3,3,4}; int      k
    =
removeDuplicates(nums);

   std::cout << "Number of unique elements: " << k << std::endl; std::cout
   << "Array after removal of duplicates: ";
   for (int i = 0; i < k; ++i) { std::cout <<
   nums[i] << " ";
   }     std::cout  <<  std::endl;
   return
   0;
}
```

Output:

```
Number of unique elements: 5
Array after removal of duplicates: 0 1 2 3 4
```

8. **Aim**: 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.

Code:

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

int calPoints(std::vector<std::string>& ops) { std::vector<int> record;

    for (const std::string& op : ops) { if
            (op    ==    "C")   {
       record.pop_back();
       } else if (op == "D") { record.push_back(record.back() *
          2);
       } else if (op == "+") { int size
          = record.size();
          record.push_back(record[size - 1] + record[size - 2]);
       } else { record.push_back(std::stoi(op));
       }
    } int sum =
    0;  for (int score : record) {
    sum += score;
    }  return
    sum;  }
```

```
int main() { std::vector<std::string> ops1 = {"5", "2", "C",
   "D", "+"}; std::vector<std::string> ops2 = {"5", "-2", "4", "C", "D", "9",
   "+", "+"};
   std::vector<std::string> ops3 = {"1", "C"};

   std::cout << "Example 1: " << calPoints(ops1) << std::endl;
   std::cout << "Example 2: " << calPoints(ops2) << std::endl;
   std::cout << "Example 3: " << calPoints(ops3) << std::endl;
return 0;
}
```

Output:

```
Example 1: 30
Example 2: 27
Example 3: 0


...Program finished with exit code 0
Press ENTER to exit console.
```

9. **Aim**: 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>

struct ListNode { int

val;

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

// Function to reverse the linked list iteratively
```

```
ListNode* reverseListIterative(ListNode* head) {  ListNode*
    prev = nullptr;
    ListNode* current = head;
    ListNode* nextNode = nullptr;

    while (current != nullptr) {  nextNode = current-
        >next; // Save next node
         current->next = prev;   // Reverse the current node's pointer
        prev = current;  // Move prev to current current = nextNode;  //
        Move to the next node
    }

    return prev; // Return the new head of the reversed list
}

// Function to print the linked list
void printList(ListNode* head) {
while (head != nullptr) { std::cout
<< head->val << " "; head =
head>next;      }      std::cout   <<
std::endl;
}

// Main function to demonstrate the solution int main()
{
    // Create linked list: 1 -> 2 -> 3 -> 4 -> 5
    ListNode* head = new ListNode(1); head->next =
    new ListNode(2);   head->next->next  =       new
    ListNode(3);          head->next->next->next  =
     new ListNode(4);  head->next->next->next->next
     = new ListNode(5);  std::cout << "Original List:
    "; printList(head);

    // Reverse the linked list head =
    reverseListIterative(head);
```

```
    std::cout << "Reversed List: ";
    printList(head);
return 0;
}
```

Output:

```
Original List: 1 2 6 3 4 5 6
List after removing 6: 1 2 3 4 5
Reversed List (Iterative): 5 4 3 2 1
Reversed List (Recursive): 1 2 3 4 5


...Program finished with exit code 0
Press ENTER to exit console.
```

10. Aim: Given the head of a singly linked list, reverse the list, and return the reversed list.

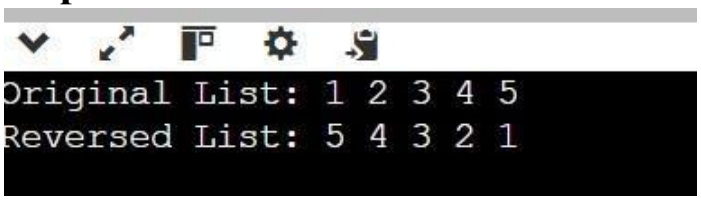
Code:

```
#include <iostream>

struct ListNode { int

val;

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

// Function to remove nodes with a given value
ListNode* removeElements(ListNode* head, int val) {
    ListNode* dummy = new ListNode(0); // Create a dummy node to handle edge
cases dummy->next = head; ListNode* current = dummy;

    while (current->next != nullptr) { if
        (current->next->val == val) {
        ListNode* temp = current-
        >next; current->next = current->next-
```

```
        >next; delete temp; // Deallocate the
        node } else {  current = current->next;
        }
    }
    ListNode* newHead = dummy->next; delete
    dummy; // Deallocate the dummy node return
    newHead;
}

// Iterative function to reverse the linked list
ListNode* reverseListIterative(ListNode* head) {
    ListNode* prev = nullptr;
    ListNode* current = head;

    while (current != nullptr) {
        ListNode* nextNode = current->next; // Save next node
    current->next = prev; // Reverse the current node's pointer prev
    = current; // Move prev to current current = nextNode;
    // Move to the next node }

    return prev; // New head of the reversed list
}

// Recursive function to reverse the linked list
ListNode*  reverseListRecursive(ListNode*  head)  {  if
    (head  ==  nullptr  ||  head->next  ==  nullptr)  {  return
    head; // Base case: empty list or single node  }

    ListNode* newHead = reverseListRecursive(head->next); // Reverse the
rest of the list head->next->next = head; // Make the next node point to
current node head->next = nullptr; // Set current node's next to null

    return newHead; // Return new head
}
```

```
// Function to print the linked list
void printList(ListNode* head) {
while (head != nullptr) { std::cout
<< head->val << " "; head
      = head->next;
   }
   std::cout << std::endl;
}
// Main function to demonstrate the solution int main()
{
   // Create linked list: 1 -> 2 -> 6 -> 3 -> 4 -> 5 -> 6
   ListNode* head = new ListNode(1); head->next = new
   ListNode(2); head->next->next = new ListNode(6); head-
   >next>next->next = new ListNode(3); head->next->next->next-
   >next = new ListNode(4); head->next->next->next->next->next
   = new ListNode(5); head->next->next->next->next->next-
   >next
   = new ListNode(6); std::cout << "Original

   List: "; printList(head);

   // Remove nodes with value 6 head =
   removeElements(head, 6); std::cout <<
   "List after removing 6: ";
   printList(head);

   // Reverse the list iteratively head = reverseListIterative(head);
   std::cout <<
   "Reversed List (Iterative): ";
   printList(head);

   // Reverse the list recursively head = reverseListRecursive(head);
   std::cout <<
   "Reversed List (Recursive): "; printList(head);
return 0;
}
```

Output:



11. Aim: 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. 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]).

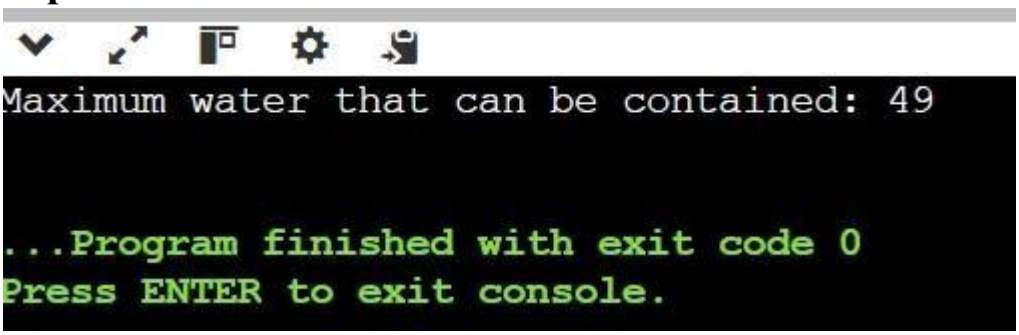
Find two lines that together with the x-axis form a container, such that the container contains the most water. Return the maximum amount of water a container can store. Notice that you may not slant the container.

Code:

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

using namespace std;

// Function to find the maximum area int maxArea(vector<int>&
height) {
    int left = 0, right = height.size() - 1;  int
    max_area = 0;
while (left < right) {
        // Calculate the area formed by the lines at the left and right pointers
        int width = right - left;
        int current_height = min(height[left], height[right]);
        int current_area = width * current_height; max_area
        = max(max_area, current_area);

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

```
    }              return

    max_area;

}

int main() { // Example Input  vector<int>
    height = {1,8,6,2,5,4,8,3,7};
    // Output the maximum area
    cout << "Maximum water that can be contained: " << maxArea(height) <<
endl;  return

    0;

}
```

Output:



12. Aim: 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.

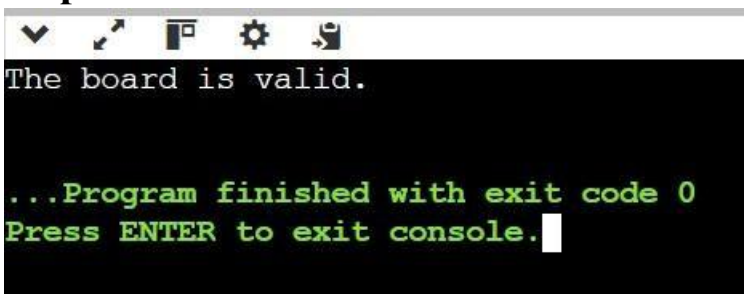
Code:

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

```
#include <unordered_set>

using namespace std;

// Function to check if the board is valid
bool isValidSudoku(vector<vector<char>>& board) {
    // Use 3 sets to track numbers in rows, columns, and sub-boxes
    unordered_set<string> rows[9], cols[9], boxes[9];
for (int i = 0; i < 9; ++i) { for (int j = 0;
    j < 9; ++j) {  if (board[i][j]
    != '.') {
            // Generate a string for the current value at [i][j] char
            num = board[i][j]; string rowKey = "row" + to_string(i) +
            num; string colKey = "col" + to_string(j) + num; string
            boxKey = "box" + to_string(i / 3 * 3 + j / 3) + num;
            // Check for duplicates in the current row, column, or sub-box if
            (rows[i].count(rowKey) || cols[j].count(colKey) || boxes[i / 3 * 3 +
        j / 3].count(boxKey)) { return false;
            }

            // Add the current value to the sets  rows[i].insert(rowKey);
            cols[j].insert(colKey);
            boxes[i / 3 * 3 + j / 3].insert(boxKey); }  }
    }
return true;  }

int main() {
    // Example Input: Sudoku board  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'}
        };

        // Output the validity of the Sudoku board if
        (isValidSudoku(board)) { cout << "The board
        is valid." << endl; } else {  cout << "The board
        is not valid." << endl; }
return 0;
    }
```

Output:



13. Aim: 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(); if (n == 1) return 0; //
   No jump needed if the array has only one element  int

   jumps = 0; int
```

```
        current_end = 0;  int
        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 we've reached or passed the last index, break if
          (current_end >= n - 1) break; }
        }         return

        jumps;

    }

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

        return 0;
    }
```

Output:

```
Minimum jumps for nums1: 2
Minimum jumps for nums2: 2


...Program finished with exit code 0
Press ENTER to exit console.
```

14. **Aim**: 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> #include
<queue>          using

namespace std;

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

class Solution { public:
    Node* connect(Node* root) { if (!root)
        return nullptr;

        Node* leftmost = root; while

              (leftmost->left)      { Node*

        current = leftmost;  while

        (current) {

              // Connect the left child to the right child current->left->next
              = current->right;

              if (current->next) { current->right->next = current->next->left;
              }     current   =   current-
              >next;
          }    leftmost  =  leftmost-
          >left;
        }      return
        root;
    }

    // Function to print the tree in level-order void
    printLevelOrder(Node* root) { if
    (!root) return;
```

```
        queue<Node*> q; q.push(root); while
        (!q.empty()) { int size = q.size();
        for (int i = 0; i < size; i++) {
        Node* node = q.front(); q.pop();
        cout << node->val; if
        (node->next) {  cout << " -> " << node-
                >next->val;
            } else { cout << " -> NULL";
            }   cout << " ";   if (node->left)
            q.push(node->left); if
            (node->right) q.push(node->right);
        }
        cout << endl;
      }
    }
};

int main() {
    Node* root = new Node(1);
    root>left = new Node(2); root-
    >right = new Node(3); root->left-
    >left = new Node(4); root->left-
    >right = new Node(5); root-
    >right->left = new Node(6); root-
    >right->right = new Node(7);

    Solution sol; sol.connect(root);
    sol.printLevelOrder(root);
    return 0;
}
```

Output:

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


...Program finished with exit code 0
Press ENTER to exit console.
```

15. Aim: 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".

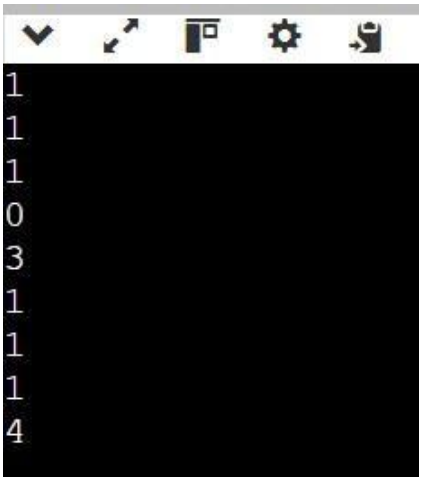
Code:

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

class MyCircularQueue { private:
    vector<int> queue; int
    front, rear, size;

public:
    // Initialize the circular queue with size k  MyCircularQueue(int
    k)
        { size = k; queue.resize(k);
        front =
        rear = -1;
    }

    // Insert an element into the circular queue
    bool enQueue(int value) { if (isFull()) return false; // If the
        queue is full, return false if (isEmpty()) { front = 0; // Set
        front to 0 when the first element is inserted  } rear =
        (rear + 1) % size; // Circular increment of rear
        queue[rear] = value; // Insert the value at the rear return
        true;
    }
```

```
// Delete an element from the circular queue
bool deQueue() { if (isEmpty()) return false; // If the queue is
   empty, return false if (front == rear) { // If there is only one
   element front = rear = -1; // Reset the queue
   } else { front = (front + 1) % size; // Circular increment
   of front  }  return true;
} int Front() { if (isEmpty()) return -1; // Return -1 if the queue
is empty return queue[front];
} int
Rear() { if
(isEmpty()
) return -1;
//   Return
1   if   the
queue is
empty return
queue[rear
];
}  bool isEmpty() {
return front == -1;
} bool isFull() { return (rear + 1)
% size == front; } };

int main() {
   MyCircularQueue queue(3); // Initialize the circular queue with size 3

   cout << queue.enQueue(1) << endl; // true cout
   << queue.enQueue(2) << endl; // true cout <<
   queue.enQueue(3) << endl; // true
   cout << queue.enQueue(4) << endl; // false, queue is full

   cout << queue.Rear() << endl; // 3 cout <<
   queue.isFull()  <<  endl;  //  true  cout  <<
   queue.deQueue() << endl; // true cout <<
   queue.enQueue(4) << endl; // true cout <<
   queue.Rear() << endl; // 4  return 0;
```

}

Output:



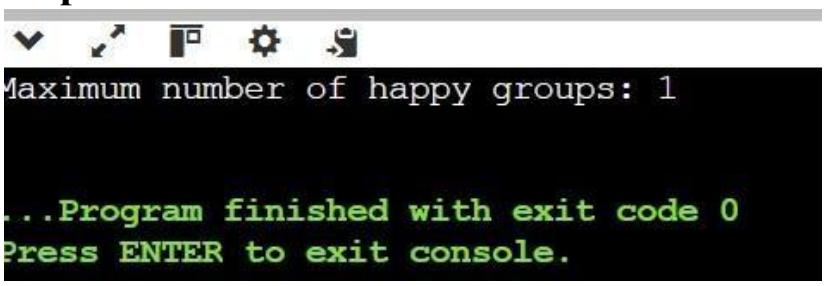
16. Aim: There is a donuts shop that bakes donuts in batches of batchSize. They have a rule where they must serve all of the donuts of a batch before serving any donuts of the next batch. You are given an integer batchSize and an integer array groups, where groups[i] denotes that there is a group of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

Code:

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

using namespace std;

int maxHappyGroups(int batchSize, vector<int>& groups) { // Sort
   the groups in descending order to prioritize larger groups
   sort(groups.begin(), groups.end(), greater<int>()); int
   happyGroups = 0; int
   leftover = 0;

   // Iterate over the groups for
   (int group : groups) {
```

```
        // Check if the group fits in the current batch
        if (leftover + group <= batchSize) { // Serve
        the group with the current batch leftover +=
        group; happyGroups++;
        } else {
            // Otherwise, start a new batch for this group leftover =
            group;
        }
    }

    return happyGroups;
}

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

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

Output:



17. Aim: 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

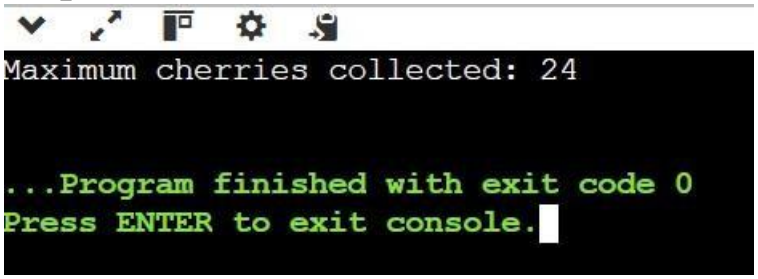
in the same cell, only one takes the cherries. Both robots cannot move outside of the grid at any moment. Both robots should reach the bottom row in grid.

Code:

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

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

   vector<vector<int>>(cols,

vector<int>(cols, -1))); dp[0][0][cols - 1] = grid[0][0] + grid[0][cols - 1]; // Only
   count cherries once
if both robots are at the same cell for (int
   i = 1; i < rows; ++i) { for (int j1 = 0;
   j1 < cols; ++j1) { for (int j2 = 0; j2 <
   cols; ++j2) {
            if (dp[i - 1][j1][j2] == -1) continue; // Skip invalid positions
            for (int move1 = -1; move1 <= 1; ++move1) { for (int move2 =
               -1; move2 <= 1; ++move2) { int nj1 = j1 + move1, nj2 = j2 +
               move2;   if (nj1 < 0 || nj1 >= cols || nj2 < 0 || nj2 >= cols)
               continue;
                  int cherries = grid[i][nj1] + grid[i][nj2];  if (nj1 == nj2) cherries
                  -= grid[i][nj1]; // If both robots are at the
same cell, count cherries once

                  dp[i][nj1][nj2] = max(dp[i][nj1][nj2], dp[i - 1][j1][j2] +
cherries);
               }
            }
         }
```

```
    } } int maxCherries = 0; for (int j1 = 0; j1 < cols; ++j1) {
    for (int j2 = 0; j2 < cols; ++j2) { maxCherries =
    max(maxCherries, dp[rows - 1][j1][j2]); }
    }                return

    maxCherries;

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

Output:



18. Aim: Maximum Number of Darts Inside of a Circular Dartboard

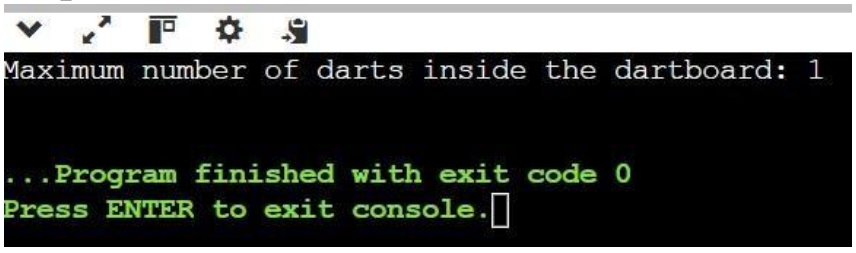
Alice is throwing n darts on a very large wall. You are given an array darts where darts[i] = [xi, yi] is the position of the ith dart that Alice threw on the wall. Bob knows the positions of the n darts on the wall. He wants to place a dartboard of radius r on the wall so that the maximum number of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

Code:

```
#include <iostream>
#include <vector>  #include
<cmath> #include
<algorithm>  using namespace std;  double distanceSquared(vector<int>&
p1, vector<int>& p2) { return (p1[0] - p2[0]) * (p1[0] - p2[0]) + (p1[1] -
p2[1]) * (p1[1] - p2[1]);
```

```
} int numPointsInsideCircle(vector<vector<int>>& darts, int r) {
int maxDarts = 0; for (int i = 0; i < darts.size(); ++i) { int count =
0;
    // Consider each dart and check if it falls within the circle centered at
darts[i] for (int j = 0; j < darts.size(); ++j) { if
(distanceSquared(darts[i], darts[j]) <= r * r) { count++; }
    }
    maxDarts = max(maxDarts, count);
  }          return
  maxDarts;
} int main() { vector<vector<int>> darts = {{-2, 0}, {2, 0}, {0,
2}, {0, -2}}; int r = 2;

  int result = numPointsInsideCircle(darts, r);
  cout << "Maximum number of darts inside the dartboard: " << result <<
endl;
return 0;
}
```

Output:



19. Aim: 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.

For example, we have a Skiplist containing [30,40,50,60,70,90] and we want to add 80 and 45 into it. The Skiplist works this way:

Code:

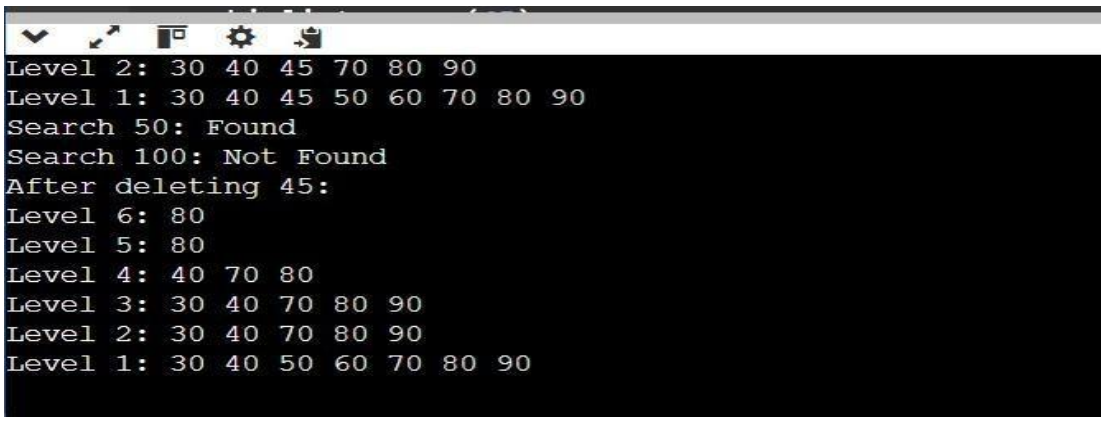
```
#include <iostream>
#include <vector>
#include  <cstdlib>  #include
<ctime>   using   namespace
std; const int MAX_LEVEL
= 16; const float P = 0.5; class SkiplistNode
{ public:
   int value;
   vector<SkiplistNode*> forward;
   SkiplistNode(int value, int level) : value(value), forward(level, nullptr) {}
}; class Skiplist
{ private:
   int level;
   SkiplistNode* header;  int
   randomLevel() { int lvl =
   1;
      while (((float)rand() / RAND_MAX) < P && lvl < MAX_LEVEL) {
      lvl++; } return lvl;
   }

public:
   Skiplist() { level =
      1;
      header = new SkiplistNode(-1, MAX_LEVEL); srand(time(nullptr));
   }  bool search(int target) {
   SkiplistNode* current = header;

      for (int i = level - 1; i >= 0; i--) { while (current->forward[i] &&
         current>forward[i]->value < target) { current = current->forward[i]; }
      }

      current = current->forward[0];
      return current && current->value == target;
   }
   void insert(int value) {
      vector<SkiplistNode*> update(MAX_LEVEL, nullptr); SkiplistNode*
      current = header;
```

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

>forward[0];

if (!current || current->value != value) { int newLevel
         =       randomLevel();     if
   (newLevel > level) { for (int i = level; i
   < newLevel; i++) { update[i] = header;
      }          level     =
      newLevel;
   }
   SkiplistNode* newNode = new SkiplistNode(value, newLevel);  for
   (int i = 0; i < newLevel; i++) {
      newNode->forward[i]     =     update[i]->forward[i];     update[i]-
      >forward[i] = newNode;
   }
} }
bool erase(int value) {
   vector<SkiplistNode*> update(MAX_LEVEL, nullptr); SkiplistNode*
   current = header;

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

   >forward[0];  if (!current || current->value !=

   value) { return false; // Value not found

   }
```

```
        for (int i = 0; i < level; i++) { if (update[i]-
          >forward[i] != current) { break; }  update[i]-
          >forward[i] = current->forward[i];
        }          delete

        current;

        while (level > 1 && !header->forward[level - 1])
        { level--; }  return

        true;

      } void print() { for (int i = level -
      1; i >= 0; i--) {
          SkiplistNode* current = header->forward[i]; cout
          << "Level " << i + 1 << ": ";
          while (current) { cout << current->value << " ";
            current
            = current->forward[i];
          }
          cout << endl;
        }
      }
    };

    int     main()     {
      Skiplist skiplist; skiplist.insert(30); skiplist.insert(40);
      skiplist.insert(50); skiplist.insert(60); skiplist.insert(70);
      skiplist.insert(90);  cout << "After initial insertions: " <<
      endl; skiplist.print();
skiplist.insert(80);
      skiplist.insert(45)
      ;

      cout << "After inserting 80 and 45: " << endl; skiplist.print();

      cout << "Search 50: " << (skiplist.search(50) ? "Found" : "Not Found") << endl;
```

```
cout << "Search 100: " << (skiplist.search(100) ? "Found" : "Not Found") << endl;

skiplist.erase(45);
cout << "After deleting 45: " << endl; skiplist.print();
return 0;
}
```

Output:



20. Aim: 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:

Code:

```
#include <iostream>
#include <unordered_map>
#include <unordered_set>  #include
<list> #include <string> using
namespace std; class AllOne {
private: struct Bucket { int count;
unordered_set<string>   keys;
Bucket(int cnt) : count(cnt) {}
   };
```

```
list<Bucket> buckets; unordered_map<string,
list<Bucket>::iterator> keyToBucket; unordered_map<int,
list<Bucket>::iterator> countToBucket;
list<Bucket>::iterator insertBucketAfter(list<Bucket>::iterator it, int count) {
  auto newBucket = buckets.insert(next(it), Bucket(count));
  countToBucket[count] = newBucket; return
  newBucket;
} void removeBucketIfEmpty(list<Bucket>::iterator
it) {
  if (it->keys.empty()) { countToBucket.erase(it>count);
  buckets.erase(it);
  }
} public: AllOne() {} void inc(string key) { if
(keyToBucket.find(key) == keyToBucket.end()) { if
(countToBucket.find(1) == countToBucket.end()) {
buckets.emplace_front(1); countToBucket[1] =
buckets.begin();
      }
      countToBucket[1]->keys.insert(key); keyToBucket[key]
      = countToBucket[1];
    } else { auto currentBucket =
      keyToBucket[key]; int newCount =
      currentBucket->count + 1; auto
      nextBucket = next(currentBucket);
      if (nextBucket == buckets.end() || nextBucket->count != newCount) {
        nextBucket = insertBucketAfter(currentBucket, newCount);
      }
      nextBucket->keys.insert(key); keyToBucket[key]
      = nextBucket; currentBucket->keys.erase(key);
      removeBucketIfEmpty(currentBucket);
    }
  }
  void dec(string key) {
    if (keyToBucket.find(key) == keyToBucket.end()) return;
    auto currentBucket = keyToBucket[key]; int newCount =
    currentBucket->count - 1; if (newCount == 0) {
    keyToBucket.erase(key);
```

```
        } else { auto prevBucket = currentBucket == buckets.begin() ?
    buckets.end() : prev(currentBucket); if (prevBucket == buckets.end() ||
    prevBucket->count != newCount) {  prevBucket =
    buckets.insert(currentBucket, Bucket(newCount));
    countToBucket[newCount] = prevBucket;
            }
            prevBucket->keys.insert(key);
            keyToBucket[key] = prevBucket;
        }
        currentBucket->keys.erase(key); removeBucketIfEmpty(currentBucket);
        }
      string        getMaxKey()        {        if
        (buckets.empty()) return ""; return
        *(buckets.back().keys.begin());
      }
      string        getMinKey()        {        if
  (buckets.empty())      return      "";     return
    *(buckets.front().keys.begin()); } };
    int main() { AllOne allOne;
      allOne.inc("hello");
      allOne.inc("world")
      ;
      allOne.inc("hello");
      cout << "Max Key: " << allOne.getMaxKey() <<
      endl; cout << "Min Key: " << allOne.getMinKey() <<
      endl; allOne.dec("hello"); cout << "Max Key: " <<
      allOne.getMaxKey() << endl; cout << "Min Key: "
      << allOne.getMinKey() << endl; return 0;
    }
```

Output:

21. Aim: Find Minimum Time to Finish All Jobs

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

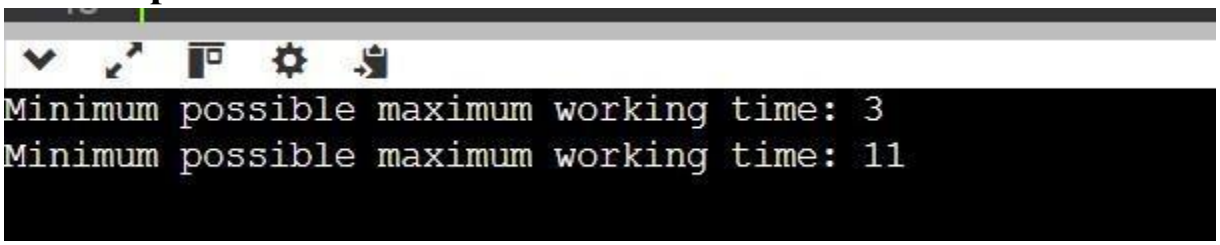
Return the minimum possible maximum working time of any assignment.

Code:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric> using namespace std; class Solution { public: bool
canDistribute(vector<int>& jobs, vector<int>& workers, int idx, int
maxTime) {  if (idx == jobs.size()) return
    true; for
    (int i = 0; i < workers.size(); i++) {  if (workers[i] + jobs[idx]
      > maxTime) continue; workers[i]
      += jobs[idx];    if (canDistribute(jobs, workers, idx + 1,
      maxTime)) return true;
      workers[i] -= jobs[idx]; if (workers[i] == 0) break;
    }
    return false;
  }

  int       minimumTimeRequired(vector<int>&       jobs,  int    k)
            { sort(jobs.rbegin(), jobs.rend());  int left = jobs[0];
    int right = accumulate(jobs.begin(), jobs.end(), 0); while
    (left < right) { int mid = left + (right - left) /
    2; vector<int> workers(k, 0);
      if (canDistribute(jobs, workers, 0, mid)) { right =
        mid;
      } else { left = mid +
        1;
      }
    }      return
    left;
```

```
    } }; int main() { Solution solution;
vector<int> jobs =
{3, 2, 3}; int
    k = 3;
    cout << "Minimum possible maximum working time: "
        << solution.minimumTimeRequired(jobs, k) << endl; jobs
    = {1, 2, 4, 7, 8}; k = 2;
    cout << "Minimum possible maximum working time: "
        << solution.minimumTimeRequired(jobs, k) << endl; return
    0;
}
```

Output:



22. Aim: 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 whee: 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.

Code:

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

```
class Solution { public:
   int minimumTeachings(int n, vector<vector<int>>& languages,
vector<vector<int>>& friendships) { vector<unordered_set<int>>
      userLanguages(languages.size()); for (int i = 0; i <
      languages.size(); i++) { userLanguages[i] =
      unordered_set<int>(languages[i].begin(),
languages[i].end());
      }
      unordered_set<int> usersToConsider; for
      (const auto& friendship : friendships) { int
      u = friendship[0] - 1;
         int v = friendship[1] - 1; bool canCommunicate = false;
         for (int lang
         :      userLanguages[u])      {      if
         (userLanguages[v].count(lang)) {
               canCommunicate = true; break;
            }
         }

         if (!canCommunicate) {
            usersToConsider.insert(u);
            usersToConsider.insert(v);
         }
      }
      vector<int> teachCount(n + 1, 0); for (int lang =
      1; lang <= n; lang++) { for (int user :
      usersToConsider)  {      if
      (!userLanguages[user].count(lang)) { teachCount[lang]++;
            }
         }
      }
      int minTeach = INT_MAX;
      for (int lang = 1; lang <= n; lang++) {
         minTeach = min(minTeach, teachCount[lang]);
      }            return
      minTeach;
```

```
} }; int main() { Solution solution; vector<vector<int>>
languages1 = {{1}, {2}, {1, 2}}; vector<vector<int>>
friendships1 = {{1, 2}, {2, 3}, {3, 1}}; cout << "Minimum
number of users to teach: "
    << solution.minimumTeachings(2, languages1, friendships1) << endl;
  vector<vector<int>> languages2 = {{1}, {2}, {3}, {4}};
  vector<vector<int>> friendships2 = {{1, 2}, {2, 3}, {3, 4}, {4, 1}}; cout
  << "Minimum number of users to teach: "
    << solution.minimumTeachings(4, languages2, friendships2) << endl; return
  0;
}
```

Output:

```
Minimum number of users to teach: 1
Minimum number of users to teach: 3
```

23. Aim: Count Ways to Make Array With Product

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

```
#include <iostream>
#include <vector>
#include <unordered_map> #include <cmath> using namespace
std; const int MOD = 1e9 + 7; const int MAX = 1e4 + 1; vector<long
long> factorial(MAX), inverseFactorial(MAX); long long
modExp(long long base, long long exp, long long mod) { long long
result = 1; while (exp > 0) { if (exp % 2 == 1) result = (result * base)
% mod; base = (base * base) % mod;  exp
    /= 2;
  }
  return result;
}
void precomputeFactorials() { factorial[0] = inverseFactorial[0] =
  1; for (int i = 1; i <
```

```
    MAX; i++) { factorial[i] = (factorial[i - 1] * i)
    % MOD;
    }
    for (int i = 0; i < MAX; i++) { inverseFactorial[i] = modExp(factorial[i],
      MOD - 2, MOD);
    } } long long comb(int n, int r) { if (r > n || n < 0 || r < 0) return 0; return
factorial[n] * inverseFactorial[r] % MOD * inverseFactorial[n - r] % MOD;
}
unordered_map<int, int> primeFactorization(int num) {
    unordered_map<int, int> factors; for
    (int p = 2; p * p <= num; p++) {
      while (num % p == 0) {
      factors[p]++; num /= p;
      }
    }
    if (num > 1) factors[num]++; return
    factors;
}

vector<int> waysToFillArray(vector<vector<int>>& queries) {
    precomputeFactorials();
    vector<int> result;

    for (const auto& query : queries) {
      int n = query[0], k = query[1]; if
      (k == 1) { result.push_back(1);
      continue;
      } auto factors =

      primeFactorization(k);

      long long ways = 1;
      for (const auto& [prime, count] : factors) {
        ways = ways * comb(count + n - 1, count) % MOD;
      } result.push_back(ways);
    }
```

```
return result;  }  int main() {  vector<vector<int>> queries = {{2, 6}, {5, 1}, {73,
        660}}; vector<int>
          result = waysToFillArray(queries);

          for (int ans : result) {  cout
            << ans << endl;
          }
return 0;
        }
```

Output:

```
4
1
50734910
```

24. **Aim**: Maximum Twin Sum of a Linked List

In a linked list of size n, where n is **even**, the ith node (**0-indexed**) of the linked list is known as the **twin** of the (n-1-i)th node, if 0 <= i <= (n / 2) – 1.

Code:

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

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

class Solution { public:
   int pairSum(ListNode* head) {
      ListNode* slow = head;
      ListNode* fast = head;

      while (fast && fast->next) { slow
        = slow->next; fast =
```

```
            fast->next->next;
        }

        ListNode* prev = nullptr;

        ListNode*  curr  =  slow;

        while (curr) {

            ListNode* nextNode = curr->next;
            curr->next = prev;
            prev = curr; curr
            = nextNode;
        }

        ListNode*  first  =  head;  ListNode*
        second = prev;
        int maxTwinSum = 0;

        while (second) { maxTwinSum = max(maxTwinSum, first->val +
            second->val); first = first->next; second = second->next;
        }

        return maxTwinSum;
    } };

int main() {
    ListNode* head = new ListNode(4); head>next =
    new ListNode(2); head->next->next
    = new ListNode(2); head->next->next->next
    = new ListNode(3);

    Solution solution; cout << "Maximum Twin Sum: " <<

    solution.pairSum(head) << endl;  return 0;

}
```

Output:

25. Aim: 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. **Code:** #include <iostream> using namespace std;

```
struct ListNode { int val;
   ListNode* next;
   ListNode(int x) : val(x), next(nullptr) {}
};  int gcd(int a, int
b {  while (b) { a
      %=       b;
      swap(a, b);
   }  return  a;  }    class
Solution { public:
   ListNode* insertGreatestCommonDivisors(ListNode* head) { ListNode*
      current = head; while (current && current-
      >next) {  int g = gcd(current->val, current-
         >next->val); ListNode* newNode = new
         ListNode(g); newNode->next = current-
         >next; current-
         >next = newNode;  current
         = newNode->next;
      }       return
      head;
   }
};
```

```
int main() {
    ListNode* head = new ListNode(2);
    head->next = new ListNode(6);
    head>next->next = new ListNode(3);

    Solution solution;
    ListNode* result = solution.insertGreatestCommonDivisors(head);
```

```
    while (result) {  cout << result->val
        << " "; result
        = result->next;
    } cout <<
    endl;  return
    0;
}
```

Output:


2 2 6 3 3