Assignment Day 2

## Majority Elements Sol-

#include <iostream> using namespace std;

int majorityElement (int nums[], int n) { int candidate = nums[0];

int count = 1;

for (int i = 1; i < n; ++i) { if (count == 0) {

candidate = nums[i]; count = 1;

} else if (nums[i] == candidate) { count++;

} else {

count--;

}

}

return candidate;

}

int main() { int n;

cout << "Enter the size of the array: "; cin >> n;

int nums[n];

cout << "Enter the elements of the array: "; for (int i = 0; i < n; ++i) {

cin >> nums[i];

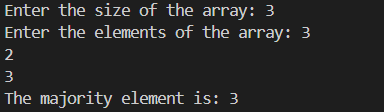
}

int result = majorityElement(nums, n);

cout << "The majority element is: " << result << endl; return 0;

}

**Output –**

****

1. **Single Number**

**Sol-**

#include <iostream> using namespace std;

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

for (int i = 0; i < n; ++i) { result ^= nums[i];

}

return result;

}

int main() { int n;

cout << "Enter the size of the array: "; cin >> n;

int nums[n];

cout << "Enter the elements of the array: "; for (int i = 0; i < n; ++i) {

cin >> nums[i];

}

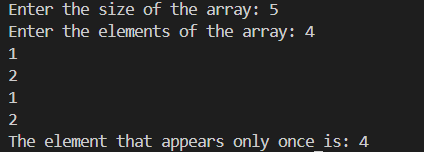
int result = singleNumber(nums, n);

cout << "The element that appears only once is: " << result << endl;

return 0;

}

**Output-**

****

1. [**Convert Sorted Array to Binary Search Tree**](https://leetcode.com/problems/convert-sorted-array-to-binary-search-tree/)

**Sol-**

#include <iostream> using namespace std; struct TreeNode {

int val; TreeNode\* left; TreeNode\* right;

TreeNode(int value) : val(value), left(nullptr), right(nullptr) {}

};

TreeNode\* sortedArrayToBST(int arr[], int start, int end) { if (start > end) {

return nullptr;

}

int mid = start + (end - start) / 2;

TreeNode\* node = new TreeNode(arr[mid]);

node->left = sortedArrayToBST(arr, start, mid - 1); node->right = sortedArrayToBST(arr, mid + 1, end); return node;

}

void inorderTraversal(TreeNode\* root) { if (root == nullptr) {

return;

}

inorderTraversal(root->left); cout << root->val << " "; inorderTraversal(root->right);

}

int main() { int n;

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

int arr[n];

cout << "Enter the sorted elements of the array: "; for (int i = 0; i < n; i++) {

cin >> arr[i];

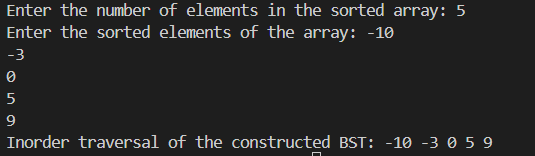
}

TreeNode\* root = sortedArrayToBST(arr, 0, n - 1); cout << "Inorder traversal of the constructed BST: "; inorderTraversal(root);

cout << endl; return 0;

}

**Output-**

****

## Merge Two Sorted Lists

**Sol-**

#include <iostream> using namespace std; struct ListNode {

int val; ListNode\* next;

ListNode(int x) : val(x), next(NULL) {}

};

ListNode\* mergeTwoLists(ListNode\* list1, ListNode\* list2) { ListNode\* dummy = new ListNode(0);

ListNode\* current = dummy;

while (list1 != NULL && list2 != NULL) { if (list1->val < list2->val) {

current->next = list1; list1 = list1->next;

} else {

current->next = list2; list2 = list2->next;

}

current = current->next;

}

if (list1 != NULL) { current->next = list1;

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

}

return dummy->next;

}

void printList(ListNode\* head) { if (head == NULL) {

cout << "Empty list" << endl; return;

}

while (head != NULL) { cout << head->val;

if (head->next != NULL) cout << " -> "; head = head->next;

}

cout << endl;

}

ListNode\* createList() {

int n;

cout << "Enter the number of elements in the list: "; cin >> n;

if (n == 0) return NULL; int val;

cout << "Enter the elements of the list (in sorted order): "; cin >> val;

ListNode\* head = new ListNode(val); ListNode\* current = head;

for (int i = 1; i < n; ++i) { cin >> val;

current->next = new ListNode(val); current = current->next;

}

return head;

}

int main() {

cout << "Enter the first linked list:" << endl; ListNode\* list1 = createList();

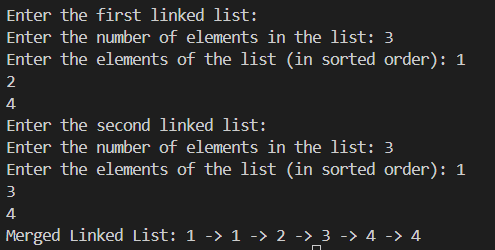
cout << "Enter the second linked list:" << endl; ListNode\* list2 = createList();

ListNode\* mergedList = mergeTwoLists(list1, list2); cout << "Merged Linked List: "; printList(mergedList);

return 0;

}

**Output-**



## Linked List Cycle

**Sol-**

#include <iostream> using namespace std; struct ListNode {

int val; ListNode\* next;

ListNode(int x) : val(x), next(NULL) {}

};

bool hasCycle(ListNode\* head) { if (head == NULL) return false; ListNode\* slow = head; ListNode\* fast = head;

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

fast = fast->next->next; if (slow == fast) {

return true;

}

}

return false;

}

ListNode\* createListWithCycle() { int n, pos;

cout << "Enter the number of nodes in the linked list: "; cin >> n;

if (n == 0) return NULL;

cout << "Enter the node values (space-separated): "; int val;

cin >> val;

ListNode\* head = new ListNode(val); ListNode\* current = head;

ListNode\* cycleNode = NULL; for (int i = 1; i < n; ++i) {

cin >> val;

current->next = new ListNode(val); current = current->next;

}

cout << "Enter the position where the tail should connect to (or -1 for no cycle): "; cin >> pos;

if (pos != -1) {

ListNode\* cyclePointer = head; for (int i = 0; i < pos; ++i) {

cyclePointer = cyclePointer->next;

}

current->next = cyclePointer;

}

return head;

}

void printList(ListNode\* head) { while (head != NULL) {

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

}

cout << "NULL" << endl;

}

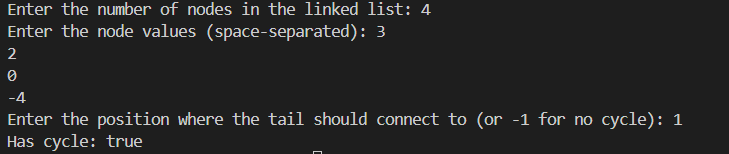
int main() {

ListNode\* head = createListWithCycle();

cout << "Has cycle: " << (hasCycle(head) ? "true" : "false") << endl; return 0;

}

**Output –**

****

## Pascal's Triangle

**Sol –**

#include <iostream> using namespace std;

void generatePascal(int numRows) { int\*\* triangle = new int\*[numRows]; for (int i = 0; i < numRows; ++i) {

triangle[i] = new int[i + 1]; triangle[i][0] = triangle[i][i] = 1; for (int j = 1; j < i; ++j) {

triangle[i][j] = triangle[i - 1][j - 1] + triangle[i - 1][j];

}

}

cout << "Pascal's Triangle: " << endl; for (int i = 0; i < numRows; ++i) {

for (int j = 0; j <= i; ++j) { cout << triangle[i][j] << " ";

}

cout << endl;

}

for (int i = 0; i < numRows; ++i) { delete[] triangle[i];

}

delete[] triangle;

}

int main() {

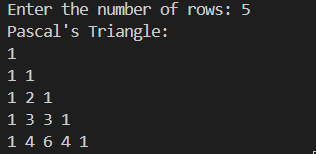
int numRows;

cout << "Enter the number of rows: "; cin >> numRows; generatePascal(numRows);

return 0;

}

**Output –**

****

## Remove Element Sol –

#include <iostream> using namespace std;

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

int i = 0;

for (int j = 1; j < numsSize; ++j) { if (nums[j] != nums[i]) {

i++;

nums[i] = nums[j];

}

}

return i + 1;

}

void printArray(int nums[], int numsSize) { for (int i = 0; i < numsSize; ++i) {

cout << nums[i] << " ";

}

cout << endl;

}

int main() {

int numsSize;

cout << "Enter the number of elements in the array: "; cin >> numsSize;

int nums[numsSize];

cout << "Enter the elements of the array: "; for (int i = 0; i < numsSize; ++i) {

cin >> nums[i];

}

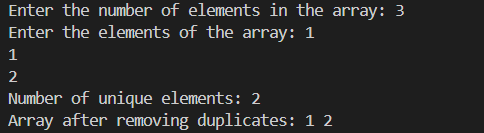
int k = removeDuplicates(nums, numsSize);

cout << "Number of unique elements: " << k << endl; cout << "Array after removing duplicates: "; printArray(nums, k);

return 0;

}

**Output –**

****

1. **Baseball GameSol –** #include <iostream> #include <string>

using namespace std;

int calculateScore(string ops[], int opsSize) { int record[opsSize];

int top = -1;

for (int i = 0; i < opsSize; ++i) { if (ops[i] == "C") {

top--;

} else if (ops[i] == "D") { record[++top] = 2 \* record[top];

} else if (ops[i] == "+") {

record[++top] = record[top - 1] + record[top];

} else {

record[++top] = stoi(ops[i]);

}

}

int totalSum = 0;

for (int i = 0; i <= top; ++i) { totalSum += record[i];

}

return totalSum;

}

int main() { int n;

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

cin.ignore(); string ops[n];

cout << "Enter the operations (space-separated): "; for (int i = 0; i < n; ++i) {

cin >> ops[i];

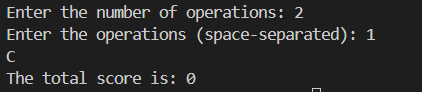
}

int result = calculateScore(ops, n);

cout << "The total score is: " << result << endl; return 0;

}

**Output –**

****

## Remove Linked List Elements Sol-

#include <iostream> using namespace std; struct ListNode {

int val; ListNode\* next;

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

};

ListNode\* removeElements(ListNode\* head, int val) { ListNode\* dummy = new ListNode(-1);

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;

} else {

current = current->next;

}

}

head = dummy->next; delete dummy;

return head;

}

void printList(ListNode\* head) { while (head != nullptr) {

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

}

cout << endl;

}

ListNode\* createList() {

int n, val;

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

if (n == 0) return nullptr; cout << "Enter the values: "; cin >> val;

ListNode\* head = new ListNode(val); ListNode\* current = head;

for (int i = 1; i < n; ++i) { cin >> val;

current->next = new ListNode(val); current = current->next;

}

return head;

}

int main() {

ListNode\* head = createList(); int valToRemove;

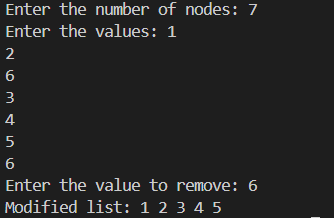
cout << "Enter the value to remove: "; cin >> valToRemove;

head = removeElements(head, valToRemove); cout << "Modified list: ";

printList(head); return 0;

}

**Output –**

****

## Reverse Linked ListSol –

#include <iostream> using namespace std; struct ListNode {

int val; ListNode\* next;

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

};

ListNode\* reverseList(ListNode\* head) { ListNode\* prev = nullptr;

ListNode\* current = head; while (current != nullptr) {

ListNode\* nextTemp = current->next; current->next = prev;

prev = current; current = nextTemp;

}

return prev;

}

void printList(ListNode\* head) { while (head != nullptr) {

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

}

cout << endl;

}

ListNode\* createList() { int n, val;

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

if (n == 0) return nullptr; cout << "Enter the values: "; cin >> val;

ListNode\* head = new ListNode(val); ListNode\* current = head;

for (int i = 1; i < n; ++i) { cin >> val;

current->next = new ListNode(val); current = current->next;

}

return head;

}

int main() {

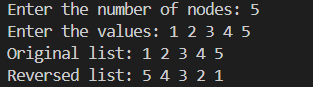
ListNode\* head = createList(); cout << "Original list: "; printList(head);

head = reverseList(head); cout << "Reversed list: "; printList(head);

return 0;

}

## Output-

****

1. **Container With Most WaterSol –**

#include <iostream> using namespace std;

int maxArea(int\* height, int n) { int left = 0, right = n - 1;

int max\_area = 0; while (left < right) {

int area = min(height[left], height[right]) \* (right - left); max\_area = max(max\_area, area);

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

} else {

right--;

}

}

return max\_area;

}

int main() { int n;

cout << "Enter the number of vertical lines: "; cin >> n;

if (n < 2) {

cout << "At least two lines are needed." << endl; return 0;

}

int\* height = new int[n];

cout << "Enter the heights of the lines: "; for (int i = 0; i < n; ++i) {

cin >> height[i];

}

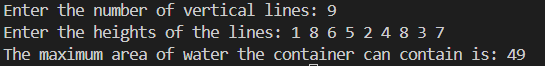
int result = maxArea(height, n);

cout << "The maximum area of water the container can contain is: " << result << endl; delete[] height;

return 0;

}

## Output -

****

1. **Valid Sudoku.**

**Sol –**

#include <iostream> #include <unordered\_set> using namespace std;

bool isValidSudoku(char board[9][9]) {

unordered\_set<string> seen; for (int i = 0; i < 9; i++) {

for (int j = 0; j < 9; j++) { char current = board[i][j]; if (current != '.') {

string row = string(1, current) + " in row " + to\_string(i); string col = string(1, current) + " in column " + to\_string(j);

string box = string(1, current) + " in box " + to\_string(i / 3) + "-" + to\_string(j / 3); if (seen.count(row) || seen.count(col) || seen.count(box)) {

return false;

}

seen.insert(row); seen.insert(col); seen.insert(box);

}

}

}

return true;

}

int main() {

char board[9][9];

cout << "Enter the Sudoku board (9x9), use '.' for empty cells:\n"; for (int i = 0; i < 9; i++) {

for (int j = 0; j < 9; j++) { cin >> board[i][j];

}

}

if (isValidSudoku(board)) {

cout << "The Sudoku board is valid.\n";

} else {

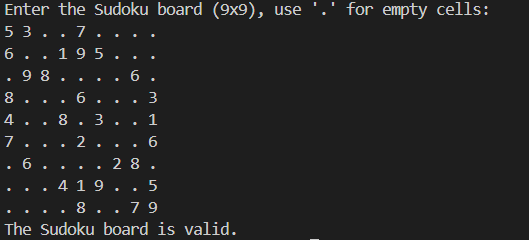
cout << "The Sudoku board is invalid.\n";

}

return 0;

}

**Output -**

****

1. **JUMP GAME II**

**Sol –**

#include <iostream> using namespace std;

int minJumps(int nums[], int n) { if (n == 1) return 0;

int jumps = 0, current\_end = 0, farthest = 0; for (int i = 0; i < n - 1; i++) {

farthest = max(farthest, i + nums[i]); if (i == current\_end) {

jumps++;

current\_end = farthest;

if (current\_end >= n - 1) break;

}

}

return jumps;

}

int main() { int n;

cout << "Enter the size of the array: "; cin >> n;

int nums[n];

cout << "Enter the elements of the array: "; for (int i = 0; i < n; i++) {

cin >> nums[i];

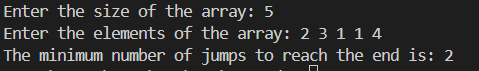
}

int result = minJumps(nums, n);

cout << "The minimum number of jumps to reach the end is: " << result << endl; return 0;

}

**Output –**

****

## Populating Next Right Pointers in Each Node Sol –

#include <iostream> using namespace std; struct Node {

int val; Node\* left; Node\* right; Node\* next;

Node(int \_val) : val(\_val), left(nullptr), right(nullptr), next(nullptr) {}

};

Node\* connect(Node\* root) { if (!root) return nullptr; Node\* leftmost = root; while (leftmost->left) {

Node\* current = leftmost; while (current) {

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

current->right->next = current->next->left;

}

current = current->next;

}

leftmost = leftmost->left;

}

return root;

}

void printLevels(Node\* root) { Node\* levelStart = root; while (levelStart) {

Node\* current = levelStart; while (current) {

cout << current->val << " "; current = current->next;

}

cout << "# ";

levelStart = levelStart->left;

}

}

int main() {

Node\* root = new Node(1); root->left = new Node(2); root->right = new Node(3);

root->left->left = new Node(4); root->left->right = new Node(5); root->right->left = new Node(6); root->right->right = new Node(7); root = connect(root);

cout << "Tree levels connected by next pointers: "; printLevels(root);

return 0;

}

**Output –**

****

## Design Circular Queue Sol –

#include <iostream>

using namespace std; class MyCircularQueue { private:

int \*queue;

int front, rear, size, count; public:

MyCircularQueue(int k) { queue = new int[k]; front = 0;

rear = -1; size = k; count = 0;

}

~MyCircularQueue() { delete[] queue;

}

bool enQueue(int value) { if (isFull()) return false; rear = (rear + 1) % size; queue[rear] = value; count++;

return true;

}

bool deQueue() {

if (isEmpty()) return false; front = (front + 1) % size; count--;

return true;

}

int Front() {

if (isEmpty()) return -1; return queue[front];

}

int Rear() {

if (isEmpty()) return -1; return queue[rear];

}

bool isEmpty() { return count == 0;

}

bool isFull() {

return count == size;

}

};

int main() {

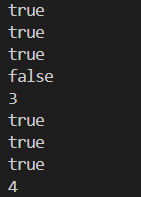
MyCircularQueue myCircularQueue(3); cout << boolalpha;

cout << myCircularQueue.enQueue(1) << endl; cout << myCircularQueue.enQueue(2) << endl; cout << myCircularQueue.enQueue(3) << endl; cout << myCircularQueue.enQueue(4) << endl; cout << myCircularQueue.Rear() << endl;

cout << myCircularQueue.isFull() << endl; cout << myCircularQueue.deQueue() << endl; cout << myCircularQueue.enQueue(4) << endl; cout << myCircularQueue.Rear() << endl; return 0;

}

**Output –**

****

1. **Maximum Number of Groups Getting Fresh Donuts**

**Sol –**

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

int maxGroups(int batchSize, int groups[], int n) { int count[batchSize];

memset(count, 0, sizeof(count)); int Groups = 0;

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

int remainder = groups[i] % batchSize; if (remainder == 0) {

Groups++;

} else if (count[batchSize - remainder] > 0) { Groups++;

count[batchSize - remainder]--;

} else {

count[remainder]++;

}

}

for (int i = 1; i < batchSize; i++) { while (count[i] > 0) {

int pair = (batchSize - i) % batchSize; if (count[pair] > 0) {

Groups++; count[i]--;

count[pair]--;

} else {

break;

}

}

}

for (int i = 1; i < batchSize; i++) { if (count[i] > 0) {

Groups++;

count[i] = 0;

}

}

return Groups;

}

int main() {

int batchSize, n;

cout << "Enter batch size: "; cin >> batchSize;

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

int groups[n];

cout << "Enter the group sizes: "; for (int i = 0; i < n; i++) {

cin >> groups[i];

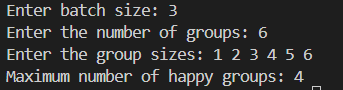
}

int result = maxGroups(batchSize, groups, n);

cout << "Maximum number of happy groups: " << result << endl; return 0;

}

**Output –**

****

## Cherry Pickup II

**Sol –**

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

int cherryPickup(int grid[][70], int rows, int cols) { int dp[70][70][70] = {0};

dp[0][0][cols - 1] = grid[0][0] + grid[0][cols - 1]; for (int r = 1; r < rows; r++) {

for (int c1 = 0; c1 < cols; c1++) { for (int c2 = 0; c2 < cols; c2++) {

if (dp[r - 1][c1][c2] == -1) continue; for (int i1 = -1; i1 <= 1; i1++) {

for (int i2 = -1; i2 <= 1; i2++) { int nc1 = c1 + i1, nc2 = c2 + i2;

if (nc1 >= 0 && nc1 < cols && nc2 >= 0 && nc2 < cols) { int cherries = grid[r][nc1] + grid[r][nc2];

if (nc1 == nc2) cherries -= grid[r][nc1];

dp[r][nc1][nc2] = max(dp[r][nc1][nc2], dp[r - 1][c1][c2] + cherries);

}

}

}

}

}

}

int maxCherries = 0;

for (int c1 = 0; c1 < cols; c1++) { for (int c2 = 0; c2 < cols; c2++) {

maxCherries = max(maxCherries, dp[rows - 1][c1][c2]);

}

}

return maxCherries;

}

int main() {

int grid[70][70] = {

{3, 1, 1},

{2, 5, 1},

{1, 5, 5},

{2, 1, 1}

};

int rows = 4, cols = 3;

cout << "Maximum cherries collected: " << cherryPickup(grid, rows, cols) << endl; return 0;

}

**Output –**

****

1. **Maximum Number of Darts Inside of a Circular Dartboard**

**Sol –**

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

double distanceSquared(int a[2], int b[2]) {

return (a[0] - b[0]) \* (a[0] - b[0]) + (a[1] - b[1]) \* (a[1] - b[1]);

}

int maxDartsInDartboard(int darts[][2], int n, int r) { int maxDarts = 1;

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

for (int j = i + 1; j < n; j++) {

double distSq = distanceSquared(darts[i], darts[j]); if (distSq > 4.0 \* r \* r) continue;

double midX = (darts[i][0] + darts[j][0]) / 2.0; double midY = (darts[i][1] + darts[j][1]) / 2.0; int count = 0;

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

double dSq = (darts[k][0] - midX) \* (darts[k][0] - midX) + (darts[k][1] - midY) \* (darts[k][1] - midY);

if (dSq <= r \* r) { count++;

}

}

maxDarts = max(maxDarts, count);

}

}

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

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

double dSq = (darts[j][0] - darts[i][0]) \* (darts[j][0] - darts[i][0]) + (darts[j][1] - darts[i][1]) \* (darts[j][1] - darts[i][1]);

if (dSq <= r \* r) { count++;

}

}

maxDarts = max(maxDarts, count);

}

return maxDarts;

}

int main() {

int darts1[][2] = {{-2, 0}, {2, 0}, {0, 2}, {0, -2}};

int r1 = 2;

cout << "Maximum darts that can lie on the dartboard: " << maxDartsInDartboard(darts1, 4, r1) << endl;

int darts2[][2] = {{-3, 0}, {3, 0}, {2, 6}, {5, 4}, {0, 9}, {7, 8}};

int r2 = 5;

cout << "Maximum darts that can lie on the dartboard: " << maxDartsInDartboard(darts2, 6, r2) << endl;

return 0;

}

**Output –**

****

## Design Skiplist

**Sol –**

#include <iostream> #include <cstdlib> #include <cmath> using namespace std; class Skiplist { private:

class Node {

public:

int val;

Node\*\* forward;

Node(int value, int level) { val = value;

forward = new Node\*[level + 1]; for (int i = 0; i <= level; ++i) {

forward[i] = nullptr;

}

}

~Node() {

delete[] forward;

}

};

Node\* head; int maxLevel;

static const int MAX\_LVL = 16; int randomLevel() {

int level = 0;

while (rand() % 2 == 0 && level < MAX\_LVL) { level++;

}

return level;

}

public:

Skiplist() {

head = new Node(-1, MAX\_LVL); maxLevel = 0;

}

bool search(int target) { Node\* current = head;

for (int i = maxLevel; i >= 0; i--) {

while (current->forward[i] != nullptr && current->forward[i]->val < target) { current = current->forward[i];

}

}

current = current->forward[0];

return current != nullptr && current->val == target;

}

void add(int num) {

Node\* update[MAX\_LVL + 1]; Node\* current = head;

for (int i = maxLevel; i >= 0; i--) {

while (current->forward[i] != nullptr && current->forward[i]->val < num) { current = current->forward[i];

}

update[i] = current;

}

current = current->forward[0];

if (current == nullptr || current->val != num) { int level = randomLevel();

if (level > maxLevel) {

for (int i = maxLevel + 1; i <= level; i++) { update[i] = head;

}

maxLevel = level;

}

Node\* newNode = new Node(num, level); for (int i = 0; i <= level; i++) {

newNode->forward[i] = update[i]->forward[i]; update[i]->forward[i] = newNode;

}

}

}

bool erase(int num) {

Node\* update[MAX\_LVL + 1]; Node\* current = head;

for (int i = maxLevel; i >= 0; i--) {

while (current->forward[i] != nullptr && current->forward[i]->val < num) { current = current->forward[i];

}

update[i] = current;

}

current = current->forward[0];

if (current != nullptr && current->val == num) { for (int i = 0; i <= maxLevel; i++) {

if (update[i]->forward[i] != current) break; update[i]->forward[i] = current->forward[i];

}

delete current;

while (maxLevel > 0 && head->forward[maxLevel] == nullptr) { maxLevel--;

}

return true;

}

return false;

}

};

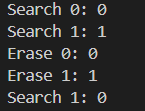
int main() { Skiplist skiplist; skiplist.add(1); skiplist.add(2); skiplist.add(3);

cout << "Search 0: " << skiplist.search(0) << endl; // Output: 0 (False) skiplist.add(4);

cout << "Search 1: " << skiplist.search(1) << endl; // Output: 1 (True) cout << "Erase 0: " << skiplist.erase(0) << endl; // Output: 0 (False) cout << "Erase 1: " << skiplist.erase(1) << endl; // Output: 1 (True) cout << "Search 1: " << skiplist.search(1) << endl; // Output: 0 (False) return 0;

}

**Output –**



## All O`one Data Structure

**Sol –**

#include <iostream> #include <unordered\_map> #include <unordered\_set> #include <string>

using namespace std; class AllOne { private:

struct Node { int count;

unordered\_set<string> keys; Node\* prev;

Node\* next;

Node(int c) : count(c), prev(nullptr), next(nullptr) {}

};

unordered\_map<string, int> keyCount; unordered\_map<int, Node\*> countList; Node\* head;

Node\* tail;

void removeNode(Node\* node) { if (node->prev) {

node->prev->next = node->next;

}

if (node->next) {

node->next->prev = node->prev;

}

delete node;

}

void insertNode(Node\* node) {

if (!head || node->count < head->count) { node->next = head;

if (head) head->prev = node; head = node;

if (!tail) tail = node;

} else {

Node\* curr = head;

while (curr->next && curr->next->count < node->count) { curr = curr->next;

}

node->next = curr->next;

if (curr->next) curr->next->prev = node; curr->next = node;

node->prev = curr;

if (!node->next) tail = node;

}

}

public:

AllOne() {

head = nullptr; tail = nullptr;

}

void inc(string key) {

int count = keyCount[key]++; if (count > 0) {

Node\* oldNode = countList[count]; oldNode->keys.erase(key);

if (oldNode->keys.empty()) { removeNode(oldNode); countList.erase(count);

}

}

int newCount = count + 1;

Node\* newNode = countList[newCount]; if (!newNode) {

newNode = new Node(newCount); countList[newCount] = newNode; insertNode(newNode);

}

newNode->keys.insert(key);

}

void dec(string key) {

int count = keyCount[key]--; Node\* oldNode = countList[count]; oldNode->keys.erase(key);

if (oldNode->keys.empty()) { removeNode(oldNode); countList.erase(count);

}

if (count > 1) {

int newCount = count - 1;

Node\* newNode = countList[newCount]; if (!newNode) {

newNode = new Node(newCount); countList[newCount] = newNode; insertNode(newNode);

}

newNode->keys.insert(key);

}

}

string getMaxKey() {

return tail ? \*tail->keys.begin() : "";

}

string getMinKey() {

return head ? \*head->keys.begin() : "";

}

};

int main() { AllOne allOne;

allOne.inc("hello"); allOne.inc("hello");

cout << allOne.getMaxKey() << endl; cout << allOne.getMinKey() << endl; allOne.inc("leet");

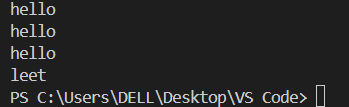
cout << allOne.getMaxKey() << endl; cout << allOne.getMinKey() << endl; allOne.dec("hello");

cout << allOne.getMaxKey() << endl; cout << allOne.getMinKey() << endl; allOne.dec("hello");

cout << allOne.getMaxKey() << endl; cout << allOne.getMinKey() << endl; return 0;

}

**Output –**

****

# Find Minimum Time to Finish All Jobs

**Sol –**

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

bool assignJob(int jobs[], int n, int k, int workers[], int index, int maxTime) { if (index == n) {

return true;

}

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

if (workers[i] + jobs[index] <= maxTime) {

workers[i] += jobs[index];

if (assignJob(jobs, n, k, workers, index + 1, maxTime)) { return true;

}

workers[i] -= jobs[index];

}

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

}

}

return false;

}

bool canAssignJobs(int jobs[], int n, int k, int maxTime) { int workers[k] = {0};

return assignJob(jobs, n, k, workers, 0, maxTime);

}

int minimumTimeRequired(int jobs[], int n, int k) { int left = \*max\_element(jobs, jobs + n);

int right = 0;

for (int i = 0; i < n; i++) { right += jobs[i];

}

while (left < right) {

int mid = (left + right) / 2;

if (canAssignJobs(jobs, n, k, mid)) { right = mid;

} else {

left = mid + 1;

}

}

return left;

}

int main() { int n, k;

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

int jobs[n];

cout << "Enter the job times:\n"; for (int i = 0; i < n; i++) {

cin >> jobs[i];

}

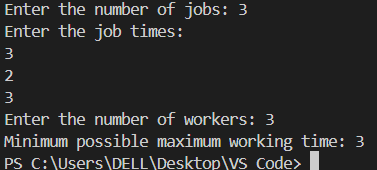
cout << "Enter the number of workers: "; cin >> k;

cout << "Minimum possible maximum working time: " << minimumTimeRequired(jobs, n, k) << endl;

return 0;

}

**Output –**

****

## Minimum Number of People to Teach Sol –

#include <iostream> #include <unordered\_set> #include <unordered\_map> #include <queue>

#include <climits> using namespace std;

const int MAX\_USERS = 500; class SocialNetwork {

public:

SocialNetwork(int n, int m, int languages[][MAX\_USERS], int friendships[][2], int f\_len) { this->n = n;

this->m = m;

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

for (int j = 0; languages[i][j] != -1; ++j) { this->languages[i].insert(languages[i][j]);

}

}

for (int i = 0; i < f\_len; ++i) { int u = friendships[i][0] - 1; int v = friendships[i][1] - 1; adj[u].insert(v);

adj[v].insert(u);

}

}

int minTeach() {

int min\_teach = 0;

bool visited[MAX\_USERS] = {false}; for (int i = 0; i < m; i++) {

if (!visited[i]) { unordered\_set<int> component; bfs(i, visited, component);

bool canCommunicate = false;

unordered\_set<int> commonLangs = languages[\*component.begin()];

for (auto it = ++component.begin(); it != component.end(); ++it) { unordered\_set<int> langs = languages[\*it];

unordered\_set<int> intersection; for (int lang : commonLangs) {

if (langs.find(lang) != langs.end()) { intersection.insert(lang);

}

}

commonLangs = intersection;

if (commonLangs.empty()) {

break;

}

}

if (!commonLangs.empty()) { continue;

} else {

int minTeachUsers = INT\_MAX; unordered\_map<int, int> languageCount;

for (auto it = component.begin(); it != component.end(); ++it) { for (int lang : languages[\*it]) {

languageCount[lang]++;

}

}

for (auto& entry : languageCount) { int count = entry.second;

minTeachUsers = min(minTeachUsers, (int)component.size() - count);

}

min\_teach += minTeachUsers;

}

}

}

return min\_teach;

}

private:

int n, m;

unordered\_set<int> languages[MAX\_USERS]; unordered\_map<int, unordered\_set<int>> adj;

void bfs(int start, bool visited[], unordered\_set<int>& component) { queue<int> q;

visited[start] = true; q.push(start); component.insert(start); while (!q.empty()) {

int user = q.front();

q.pop();

for (auto& friendUser : adj[user]) { if (!visited[friendUser]) {

visited[friendUser] = true; q.push(friendUser); component.insert(friendUser);

}

}

}

}

};

int main() { int n = 3; int m = 4;

int languages[4][MAX\_USERS] = {{2, -1}, {1, 3, -1}, {1, 2, -1}, {3, -1}};

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

SocialNetwork network(n, m, languages, friendships, 4); cout << network.minTeach() << endl;

return 0;

}

**Output –**

****

# Count Ways to Make Array With Product

**Sol –**

#include <iostream> #include <cmath> #include <map>

#define MOD 1000000007

#define MAX 10001 using namespace std;

long long fact[MAX], invFact[MAX];

long long modInverse(long long a, long long m) { long long res = 1;

long long exp = m - 2; while (exp) {

if (exp % 2) res = (res \* a) % m; a = (a \* a) % m;

exp /= 2;

}

return res;

}

void precomputeFactorials() { fact[0] = 1;

for (int i = 1; i < MAX; ++i) { fact[i] = (fact[i - 1] \* i) % MOD;

}

invFact[MAX - 1] = modInverse(fact[MAX - 1], MOD); for (int i = MAX - 2; i >= 0; --i) {

invFact[i] = (invFact[i + 1] \* (i + 1)) % MOD;

}

}

long long nCr(long long n, long long r) { if (r > n) return 0;

return (fact[n] \* invFact[r] % MOD) \* invFact[n - r] % MOD;

}

map<int, int> primeFactorization(int k) { map<int, int> factors;

for (int i = 2; i <= sqrt(k); ++i) { while (k % i == 0) {

factors[i]++; k /= i;

}

}

if (k > 1) factors[k]++; return factors;

}

long long ways(int n, int k) {

if (k == 1) return 1;

map<int, int> factors = primeFactorization(k); long long result = 1;

for (auto& factor : factors) { int p = factor.first;

int e = factor.second;

result = (result \* nCr(e + n - 1, n - 1)) % MOD;

}

return result;

}

int main() { precomputeFactorials();

int queries[3][2] = {{2, 6}, {5, 1}, {73, 660}};

for (int i = 0; i < 3; i++) { int n = queries[i][0]; int k = queries[i][1];

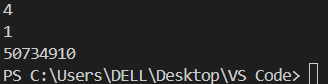
cout << ways(n, k) << endl;

}

return 0;

}

**Output –**

****

## Maximum Twin Sum of a Linked List Sol –

#include <iostream> 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;

int n = 0;

while (fast != nullptr) { n++;

fast = fast->next;

}

ListNode\* firstHalf = head; ListNode\* secondHalf = head; for (int i = 0; i < n / 2; ++i) {

secondHalf = secondHalf->next;

}

ListNode\* prev = nullptr; ListNode\* curr = secondHalf; while (curr != nullptr) {

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

prev = curr;

curr = nextNode;

}

secondHalf = prev; int maxTwinSum = 0;

ListNode\* first = firstHalf; ListNode\* second = secondHalf;

while (first != nullptr && second != nullptr) { int twinSum = first->val + second->val;

maxTwinSum = max(maxTwinSum, twinSum); first = first->next;

second = second->next;

}

return maxTwinSum;

}

};

ListNode\* createLinkedList(int arr[], int size) { ListNode\* head = nullptr;

ListNode\* tail = nullptr;

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

ListNode\* newNode = new ListNode(arr[i]); if (!head) {

head = newNode; tail = head;

} else {

tail->next = newNode; tail = newNode;

}

}

return head;

}

int main() { Solution solution; int n;

cout << "Enter the number of nodes in the linked list: "; cin >> n;

int arr[n];

cout << "Enter the values of the linked list nodes: "; for (int i = 0; i < n; ++i) {

cin >> arr[i];

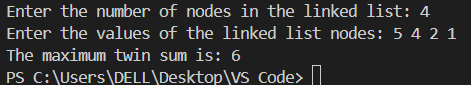
}

ListNode\* head = createLinkedList(arr, n); int result = solution.pairSum(head);

cout << "The maximum twin sum is: " << result << endl; return 0;

}

## Output –



1. **Insert Greatest Common Divisors in Linked List Sol –**

#include <iostream> using namespace std; struct ListNode {

int val; ListNode\* next;

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

};

class Solution { public:

int gcd(int a, int b) { while (b != 0) {

int temp = b; b = a % b;

a = temp;

}

return a;

}

ListNode\* insertGreatestCommonDivisors(ListNode\* head) { if (!head || !head->next) return head;

ListNode\* current = head;

while (current && current->next) {

int gcdValue = gcd(current->val, current->next->val); ListNode\* newNode = new ListNode(gcdValue); newNode->next = current->next;

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

}

return head;

}

};

ListNode\* createLinkedList(int arr[], int size) { ListNode\* head = nullptr;

ListNode\* tail = nullptr;

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

ListNode\* newNode = new ListNode(arr[i]); if (!head) {

head = newNode; tail = head;

} else {

tail->next = newNode; tail = newNode;

}

}

return head;

}

void printLinkedList(ListNode\* head) { while (head) {

cout << head->val;

if (head->next) cout << " -> "; head = head->next;

}

cout << endl;

}

int main() { Solution solution; int n;

cout << "Enter the number of elements in the linked list: "; cin >> n;

int\* arr = new int[n];

cout << "Enter the elements of the linked list: "; for (int i = 0; i < n; ++i) {

cin >> arr[i];

}

ListNode\* head = createLinkedList(arr, n); cout << "Original List: "; printLinkedList(head);

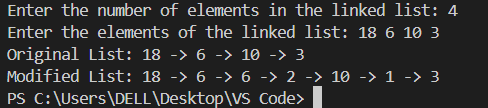
head = solution.insertGreatestCommonDivisors(head); cout << "Modified List: ";

printLinkedList(head); delete[] arr;

return 0;

}

**Output –**

****