Q-1: We are given an array consisting of n elements. At each operation you can select any one element and increase rest of n-1 elements by 1.

You have to make all elements equal performing such operation as many times you wish. Find the minimum number of operations needed for this.

Sample test case:

|  |
| --- |
| Input: arr = {1,2,3}  Output: Minimum Operation = 3 |

Solution:

#include<bits/stdc++.h>

using namespace std;

// function for finding min operation

int minOp (int arr[], int n)

{

// find array sum

int sum = accumulate(arr,arr+n,0);

// find the smallest element from array

int small = \*min\_element(arr,arr+n);

// calculate min operation required

int minOperation = sum - (n \* small);

// return result

return minOperation;

}

int main()

{

int arr[] = {1,1,1,2};

int n = sizeof(arr)/ sizeof(arr[0]);

cout << "Minimum Operation = " << minOp (arr, n);

return 0;

}

Q-2: Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to the target.

Sample test case:

|  |
| --- |
| Input: nums = {12, 8, 4, 6}  target = 14  Output: Indices of two numbers that add up to target: 1, 3 |

Solution:

#include <iostream>

#include <vector>

#include <unordered\_map>

using namespace std;

vector<int> twoSum(vector<int>& nums, int target) {

unordered\_map<int, int> num\_index\_map;

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

int complement = target - nums[i];

if (num\_index\_map.find(complement) != num\_index\_map.end()) {

return {num\_index\_map[complement], i};

}

num\_index\_map[nums[i]] = i;

}

return {};

}

int main() {

vector<int> nums = {12, 8, 4, 6};

int target = 14;

vector<int> result = twoSum(nums, target);

if (result.size() == 2) {

cout << "Indices of two numbers that add up to target: " << result[0] << ", " << result[1] << endl;

} else {

cout << "No such pair exists.\n";

}

return 0;

}

Q-3: Given an array of integers, find the number of inversions present in the array using the Merge Sort algorithm. An inversion in an array occurs when two elements at indices i and j are out of order, i.e., arr[i] > arr[j] and i < j.

Implement the Merge Sort algorithm with inversion count and return the total number of inversions in the array.

Sample test case:

|  |
| --- |
| Input: Given array: 1 20 6 4 5  Output: Number of inversions: 5 |
| Solution: |

#include <iostream>

using namespace std;

// Merge Function (same as used in Merge Sort)

// This function takes an array 'arr', a temporary array 'temp', and the indices 'left', 'mid', and 'right'.

// It merges two sorted subarrays arr[left...mid-1] and arr[mid...right] into a single sorted subarray.

// It returns the count of inversions that occurred during the merging process.

long long int merge(int arr[], int temp[], int left, int mid, int right) {

int i, j, k;

long long int inv\_count = 0;

i = left; // Index for left subarray

j = mid; // Index for right subarray

k = left; // Index for merged subarray

// Merge the subarrays while counting inversions

while ((i <= mid - 1) && (j <= right)) {

if (arr[i] <= arr[j])

temp[k++] = arr[i++];

else {

temp[k++] = arr[j++];

// If an element from the right subarray is smaller, it forms an inversion with the remaining elements in the left subarray

inv\_count += (mid - i);

}

}

// Copy the remaining elements of left subarray (if any)

while (i <= mid - 1)

temp[k++] = arr[i++];

// Copy the remaining elements of right subarray (if any)

while (j <= right)

temp[k++] = arr[j++];

// Copy the merged elements back to the original array

for (i = left; i <= right; i++)

arr[i] = temp[i];

return inv\_count;

}

// Merge Sort Algorithm (with inversion count)

// This function takes an array 'arr', a temporary array 'temp', and the indices 'left' and 'right'.

// It sorts the array using the Merge Sort algorithm and returns the count of inversions during the sorting process.

long long int mergeSort(int arr[], int temp[], int left, int right) {

long long int inv\_count = 0;

int mid;

if (right > left) {

mid = (left + right) / 2;

// Divide and conquer

inv\_count += mergeSort(arr, temp, left, mid);

inv\_count += mergeSort(arr, temp, mid + 1, right);

// Combine the results from both halves

inv\_count += merge(arr, temp, left, mid + 1, right);

}

return inv\_count;

}

int main() {

// Sample array

int arr[] = {8,4,2,1};

// Calculate the size of the array

int n = sizeof(arr) / sizeof(arr[0]);

// Temporary array to store merged subarrays during sorting

int temp[n];

// Print the given array

cout << "Given array: ";

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

cout << arr[i] << " ";

// Call the Merge Sort function with inversion count to sort the array and count inversions

long long int inversions = mergeSort(arr, temp, 0, n - 1);

// Print the number of inversions in the array

cout << "\nNumber of inversions: " << inversions;

return 0;

}

Q-4: Given a 2D matrix sorted in row-wise and column-wise order, determine if a target value is present in the matrix.

* **Constraints:**

Time complexity should be O(m+n)

Sample test case:

|  |
| --- |
| Input:  matrix=  {{1, 4, 7, 11},  {2, 5, 8, 12},  {3, 6, 9, 16},  {10, 13, 14, 17}};  target = 6  Output: Target found in the matrix. |

Solution:

#include <iostream>

using namespace std;

bool searchIn2DMatrix(int matrix[][4], int m, int n, int target) {

int row = 0, col = n - 1;

while (row < m && col >= 0) {

if (matrix[row][col] == target) {

return true;

} else if (matrix[row][col] > target) {

col--;

} else {

row++;

}

}

return false;

}

int main() {

int matrix[][4] = {{1, 4, 7, 11},

{2, 5, 8, 12},

{3, 6, 9, 16},

{10, 13, 14, 17}};

int target = 6;

bool found = searchIn2DMatrix(matrix, 4, 4, target);

if (found) {

cout << "Target found in the matrix.";

} else {

cout << "Target not found in the matrix.";

}

return 0;

}