**Assignment -1**

**Name:** Bhavin Patil

**Roll No.:** 66

**Class:** TY CS-D

**Assignment based on simple coding problems on Number, Graphs and Matrices**

1. **Number Problem**

## **Problem**:

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

You may assume that each input would have exactly one solution, and you may not use the same element twice.

You can return the answer in any order.

**Input:** integer array, integer

**Output:** integer array

**Example 1:**

**Input:** nums = [2,7,11,15], target = 9

**Output:** [0,1]

**Explanation:** Because nums[0] + nums[1] == 9, we return [0, 1].

**Example 2:**

**Input:** nums = [3,2,4], target = 6

**Output:** [1,2]

**Example 3:**

**Input:** nums = [3,3], target = 6

**Output:** [0,1]

## **Solution 1:**

To solve this problem, the first approach is brute force, where we use two for loops, and add up every pair of elements and compare it with the target. The algorithm will be:

**for(int I =0;i<arr.length-1;i++){**

**for(int j = i+1;j<arr.length;j++){**

**if(arr[i] + arr[j] == target){**

**return {I,j};**

**}**

**}**

**return null;**

The recurrence relation for this problem is

**T(n) = T(n\*(n-1))**

**T(n) = T(n^2-n)**

Considering n to be very large, it approximates upto

***O(n^2)***

And since no extra space is used, the space complexity is ***O(1)***

## **Solution 2:**

Another optimized approach to solve this problem is using a unordered\_map to store the entire array, then iterate over the entire array and look if target-arr[i] exists in the unordered\_map, if yes, return the two numbers. The code to the following approach is:

class Solution {

public:

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

vector<int>ans;

unordered\_map<int, int>mp;

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

{

if(mp.find(target-nums[i])!=mp.end())

{

ans.push\_back(mp[target-nums[i]]);

ans.push\_back(i);

return ans;

}

mp[nums[i]] = i;

}

return ans;

}

};

The recurrence relation for this particular approach will be

**T(n) = T(n) + T(n)** //first one to store the array in unordered\_map, second one to iterate

**T(n) = 2T(n)**

Which means the time complexity of the algorithm is ***O(2n) = O(n)***

And the space complexity will be ***O(n)*** for the unordered\_map.

1. **Matrix Problem**

## **Problem**:

Given a Diagonal square matrix of 5x5

[3 0 0 0 0]

[0 7 0 0 0]

[0 0 4 0 0]

[0 0 0 9 0]

[0 0 0 0 6]

## **Solution:**

This is a pattern type problem; if we represent it using 2-dimension array it will take more space and also for the 0s and while performing arithmetic operations it will also increase the time and space complexity.

So, the optimized solution for it will be taking a one-dimension array and presenting it using condition.

void diagonal::display()

{

    for (int i = 1; i <= n; i++)

    {

        for (int j = 1; j <= n; j++)

        {

            if (i == j)

                cout << A[i - 1] << " ";

            else

                cout << "0 ";

        }

        cout << endl;

    }

}

# **Graph Problem**

## **Problem**:

Perform BFS on a given adjacency matrix and Adjacency list of a graph.

void BFS(int \*\*adjMat){

    int visited[vertices];

    for(int i = 0;i<vertices;i++){

        visited[i] = 0;

    }

    int start = 0;

    printf("%d ", start);

    visited[start] = 1;

    struct queue q;

    q.size = 400;

    q.f = q.r = 0;

    q.arr = (int\*) malloc(q.size\*sizeof(int));

    enqueue(&q, start); // Enqueue i for exploration

    while (!isEmpty(&q))

    {

        int node = dequeue(&q);

        for (int j = 0; j < 7; j++)

        {

            if(adjMat[node][j] ==1 && visited[j] == 0){

                printf("%d ", j);

                visited[j] = 1;

                enqueue(&q, j);

            }

        }

    }

}

The time complexity for this particular code is **O(V^2)** because we are exploring the entire matrix which has **V^2** elements,