Day 11 / 100 :

Topic - Arrays

1 Problem statement: Longest Subarray of 1's After Deleting One element (Medium)

Given a binary array nums, you should delete one element from it.
Return the size of the longest non-empty subarray containing only 1's in the resulting array.
Return 0 if there is no such subarray.

Example 1:

Input: nums = [1,1,0,1]
Output: 3
Explanation: After deleting the number in position 2, [1,1,1] contains 3 numbers with value of 1's.
Example 2:

Input: nums = [0,1,1,1,0,1,1,0,1]
Output: 5
Explanation: After deleting the number in position 4, [0,1,1,1,1,1,0,1] longest subarray with value of 1's is [1,1,1,1,1].

Solutions :

Approach 1 - brute Force

Intuition
The intuition was an observation that the array contains only 0s and 1s. So, in order to get the maximum subArray of 1s, we need to eliminate a 0.
Approach
The approach was to get all the indexes of zeros and then calculate the subArray around it by removing the zero one by one.Additionally, handling the edge cases and returning the max length of Subarrays found

Complexity
Time complexity:
O(n)
Space complexity:
O(n)

```cpp
class Solution {
public:
    int longestSubarray(vector<int>& nums) {
        vector<int> zeroIndexes;
        int n = nums.size();
        for(int i=0; i<n; i++){
            if(nums[i]==0){
                zeroIndexes.push_back(i);
            }
        }
        if(zeroIndexes.size()==0 || zeroIndexes.size()==1){
            return nums.size()-1;
        }
        int maxAns = 0;
        for(int i=0; i<zeroIndexes.size(); i++){
            int ans = 0;
            if(i==0){
                ans += zeroIndexes[i];
                ans += (zeroIndexes[i+1]-zeroIndexes[i]-1);
            }
            else if(i==zeroIndexes.size()-1){
                ans += (nums.size()-zeroIndexes[i]-1);
                ans += (zeroIndexes[i]-zeroIndexes[i-1]-1);
            }
            else{
                ans += (zeroIndexes[i+1]-zeroIndexes[i]-1);
                ans += (zeroIndexes[i]-zeroIndexes[i-1]-1);
            }
            maxAns = max(ans, maxAns);
        }
        return maxAns;
    }
};
```

Solutions :

Approach 2 - Optimized Approach

Intuition:

The Intuition is to use a sliding window approach to find the length of the longest subarray of 1's after removing at most one element (0 or 1) from the original array. It adjusts the window to have at most one zero, calculates the subarray length, and returns the maximum length found.

Explanation:

1. The code aims to find the length of the longest subarray consisting of only 1's after deleting at most one element (0 or 1) from the original array.
2. The variable left represents the left pointer of the sliding window, which defines the subarray.
3. The variable zeros keeps track of the number of zeroes encountered in the current subarray.
4. The variable ans stores the maximum length of the subarray found so far.
5. The code iterates over the array using the right pointer right.
6. If nums[right] is 0, it means we encountered a zero in the array. We increment zeros by 1.
7. The while loop is used to adjust the window by moving the left pointer left to the right until we have at most one zero in the subarray.
8. If nums[left] is 0, it means we are excluding a zero from the subarray, so we decrement zeros by 1.
9. We update the left pointer by moving it to the right.
10. After adjusting the window, we calculate the length of the current subarray by subtracting the number of zeroes from the total length right - left + 1. We update ans if necessary.
11. Finally, we check if the entire array is the longest subarray. If it is, we subtract 1 from the maximum length to account for the one element we are allowed to delete. We return the resulting length.

```
class Solution {
public:
    int longestSubarray(vector<int>& nums) {
        int n = nums.size();

        int left = 0;
        int zeros = 0;
```

```
        int ans = 0;

        for (int right = 0; right < n; right++) {
            if (nums[right] == 0) {
                zeros++;
            }
            while (zeros > 1) {
                if (nums[left] == 0) {
                    zeros--;
                }
                left++;
            }
            ans = max(ans, right - left + 1 - zeros);
        }
        return (ans == n) ? ans - 1 : ans;
    }
};
```

2 Problem statement: Two Sum (Medium)

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.

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]

Solutions :

Approach 1 - Brute Force

Intuition:
Just a simple brute force approach - double for-loop that searches through every possible combination of sums.

Approach:
Double for loop, if i and j are not the same, test to seem if the sum of the numbers in their respective locations equal the target.

```
class Solution {
public:
vector<int> output{0,0};

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

        int location1;
        int location2;

        for(int i = 0; i < nums.size(); i++){
            for(int j = 0; j < nums.size(); j++){
                if(i!=j){
                    if(nums[i]+nums[j]==target){
                        output[0] = j;
                        output[1] = i;
                        break;
                    }
                }
            }
        }

        return output;
    }
};
```

Solutions :

Approach 2 - Two Pointers

```cpp
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        unordered_map<int, int> numMap;

        for (int i = 0; i < nums.size(); ++i) {
            int complement = target - nums[i];

            if (numMap.find(complement) != numMap.end()) {
                return { numMap[complement], i };
            }

            numMap[nums[i]] = i;
        }

        return {};
    }
```

Explanation:

1. We include the necessary headers for using vectors and unordered maps.
2. We define the Solution class.
3. Within the Solution class, we define the twoSum function that takes a vector of integers nums and an integer target as input and returns a vector of integers.
4. Inside the function, we create an unordered map numMap to store the numbers as keys and their indices as values.
5. We iterate over the vector nums using a for loop and index variable i.
6. For each number nums[i], we calculate its complement by subtracting it from the target value.
7. We use the find function of the unordered map to check if the complement exists in the map. If found, we return a vector containing the indices numMap[complement] and i.
8. If the complement is not found, we add the current number and its index to the map using numMap[nums[i]] = i.
9. If no solution is found, we return an empty vector {}.

3 Problem statement: Sub array sum equal K (Medium)

Given an array of integers nums and an integer k, return the total number of subarrays whose sum equals to k.

A subarray is a contiguous non-empty sequence of elements within an array.

Example 1:

Input: nums = [1,1,1], k = 2
Output: 2
Example 2:

Input: nums = [1,2,3], k = 3
Output: 2

Solutions :

Approach 1 - Brute Force

```
class Solution {
public:
    int subarraySum(vector<int>& nums, int k) {
        // Brute Force Solution Time O(N^3) & Auxiliary Space O(1)
        // Time Limit Exceed(TLE) 61/89 test cases passed
        int len=nums.size(),count=0;
        // Consider all possible subarrays
        for(int i=0;i<len;i++){
            for(int j=i;j<len;j++){ // Consider subarray from nums[i]
to nums[j]
                int sum=0;
                for(int s=i;s<=j;s++){ // Calculate sum of elements
from nums[i] to nums[j]
                   sum+=nums[s];
        }
                if(sum==k) // Check if sum is equal to k
                    count++;
        }
        }
```

```
        return count;
    }
};
```

Solutions :

Approach 2 - Prefix Sum

Explanation:

- we keep an accumulator variable sum with the running total of the sum of numbers; we then check if we have already met that values using our seen hashmap that acts more or less like a frequency table, storing how many times we have encountered a specific value: sum - k.
- That is why if have met sum - k before and now the value is sum, the difference between those specific points and the current iteration is, by definiton, exactly k: we are now at sum, so, the interval between the previous point(s) and now sums up to, by definition, sum - (sum - k), which equates k.
- We collect all those occurrences in count and finally we return it.

```
class Solution {
public:
    int subarraySum(vector<int>& nums, int k) {
        std::unordered_map<int, int> seen = {{0, 1}};
        int count = 0, sum = 0;
        for (auto n: nums) {
            sum += n;
            count += seen[sum - k];
            seen[sum]++;
        }
        return count;
    }
};
```