USS ACM ICPC Club

Introductory session on Two Pointers

Two Pointers

This technique is one the most basic and important techniques that has it’s application in both Programming Interviews and even Competitive Programming.

It requires us to solve a problem where we use two “Pointers” to indicate the presence of indices at different positions in an Array/Vector.

Techniques

The term two pointers is just an indicative of the fact that we are using two variables to assess our position in an Array. It can be used in any way as per the requirement of the question.

The ‘Two Pointers’ might begin at different ends of the array and ultimately converge

They might even move at different speeds to gain required information such as in middle element of Linked Lists

Problem 1

Qn 1: 2SUM

Suppose you have an Array, you need to find a pair of numbers whose sum is exactly equal to 'x'.

Brute Force

The Brute Force approach to this question would be quite simple, to iterate over all the elements of the first array and in a nested loop, iterate over all elements of the next finding their corresponding sums and checking if it is equal to ‘x’ or not.

```
for(int i = 0; i<n ; i++){
        for(int j =i+1 ; j<n ; j++){
                if(arr[i]+arr[j]==x)
                        cout<<i<<” “<<j<<endl;
        }
}
```

But this has a Time Complexity of O(n^2) which is highly undesirable. To help reduce this we will employ the two pointer approach.

Optimized

Since the array is sorted, we will store the pointer ‘s’ denoting start at Index 0 and pointer ‘e’ denoting end at index (n-1)

Now we check if sum of element at ‘s’ and ‘e’ is equal to ‘x’ or not. If it is, we have got the answer and will simply print it.

If not, we check if the sum is greater than ‘x’. If it is, we need to have smaller element than the ones we currently have and hence we will move the end pointer towards the left because the end pointer contains the greater of the two elements and shifting it to the left decrease it’s value.

Similarly when sum is smaller, we will increase the start pointer by 1 increasing the value stored at index ‘s’.

Visualization

1 2 6 19 21 : x = 8

Initially pointers are at 1 and 21…the sum is greater than 8 so, end pointer moves to left.

Now we have 1 and 19… sum is still greater so the pointer moves further left

Now we have 1 and 6… the sum is now smaller so the start pointer moves to right

Now we have 2 and 6 which is equal so we have found the solution.

Now try and code this problem and paste your code in the comments

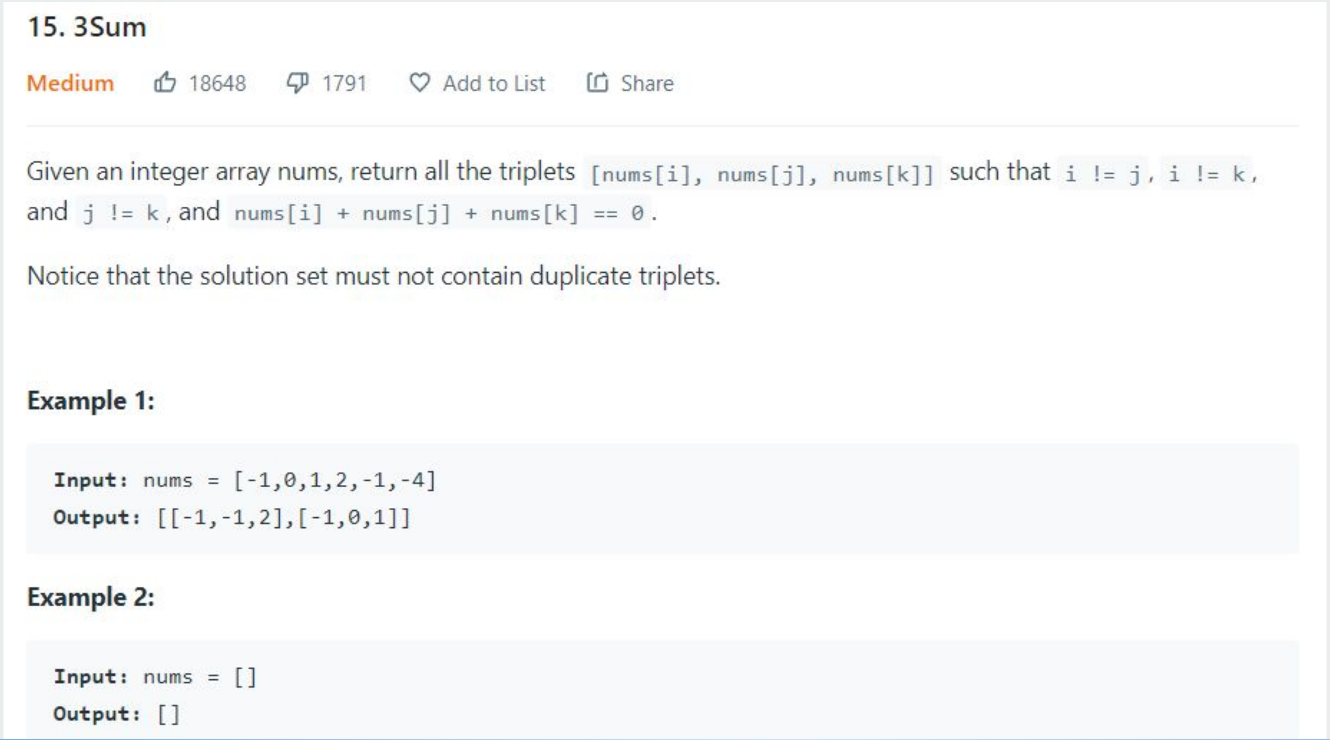
Code

```
int s =0, e = n-1;
while(s<e){
        if(arr[s] + arr[e] == x){
                cout<<s<<" "<<e<<endl;
                break;
        }

        else if(arr[s]+arr[e]<x)
                s++;
        else
                e–;
}
```

Here our time complexity is reduced to O(n) as it requires us to traverse the array only once.

Problem 2



15. 3Sum

Medium 18648 1791 Add to List Share

Given an integer array nums, return all the triplets `[nums[i], nums[j], nums[k]]` such that `i != j`, `i != k`, and `j != k`, and `nums[i] + nums[j] + nums[k] == 0`.

Notice that the solution set must not contain duplicate triplets.

Example 1:

```
Input: nums = [-1,0,1,2,-1,-4]
Output: [[-1,-1,2],[-1,0,1]]
```

Example 2:

```
Input: nums = []
Output: []
```

Brute Force

In Brute force solution, we use 3 array, one from 0 to n-1, the second starting from 1 and the third starting from 2… finding all possible combinations with a time complexity of O(n^3)

We can reduce this to O(n^2) using Two Pointers methods

Can someone take a guess at how to approach this problem using two pointers

Two Pointers

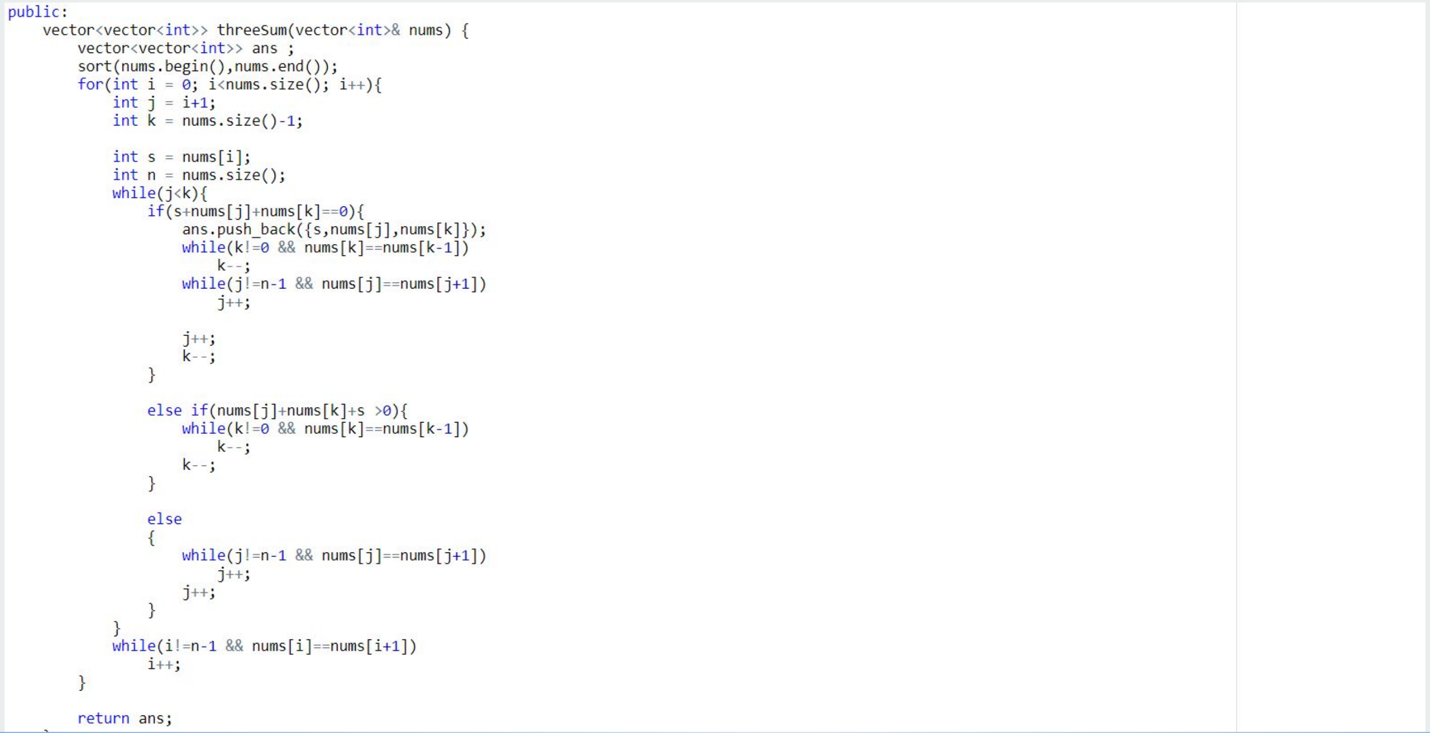
Using two pointer approach we traverse across the entire array fixing the current element and using two pointers look for the sum, x-arr[i] in the remaining portion of the array.

Eg : 1 3 5 7 9 11 13 x = 13

In first iteration we fix, 1 and then make i+1 th index as the first pointer and n-1 th index as the second pointer and employ same technique as 2Sum to find the solution

This reduced complexity to O(n^2)

Code



```
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        vector<vector<int>> ans ;
        sort(nums.begin(),nums.end());
        for(int i = 0; i<nums.size(); i++){
            int j = i+1;
            int k = nums.size()-1;

            int s = nums[i];
            int n = nums.size();
            while(j<k){
                if(s+nums[j]+nums[k]==0){
                    ans.push_back({s,nums[j],nums[k]});
                    while(k!=0 && nums[k]==nums[k-1])
                        k--;
                    while(j!=n-1 && nums[j]==nums[j+1])
                        j++;

                    j++;
                    k--;
                }

                else if(nums[j]+nums[k]+s >0){
                    while(k!=0 && nums[k]==nums[k-1])
                        k--;
                    k--;
                }

                else
                {
                    while(j!=n-1 && nums[j]==nums[j+1])
                        j++;
                    j++;
                }
            }
            while(i!=n-1 && nums[i]==nums[i+1])
                i++;
        }

        return ans;
```

Use Case 2

The other method of using Two Pointers is when both the pointers are at same position initially and their movement speed differs. This is used to find the middle element in a Linked List.

Suppose we want to find the middle element in a linked list the naive method would be to move across the entire Linked List and find its length. Following this we would traverse the list again to find the element in the middle of the list.

This can however be executed in a much faster way by using Two Pointers

Visualization

Suppose the list is : 1 -> 2 -> 3 -> 4 -> 5

Now we have two pointers both initially at element 1

I1 : 1 -> 2 -> 3 -> 4 -> 5 [fast and slow at 1]
I2 : 1 -> 2 -> 3 -> 4 -> 5 [fast at 3 and slow at 2]
I3 : 1 -> 2 -> 3 -> 4 -> 5 [fast at 5 and slow at 3]

Now fast has reached the end therefore, we stop further operations and return the element at slow pointer as the middle element of Linked List.

Code

```
class Solution {
public:
  ListNode* middleNode(ListNode* head) {
    ListNode* slow = head;
    ListNode* fast = head;
    while(fast!=NULL && fast->next!=NULL){
      fast = fast->next->next;
      slow = slow->next;
    }

    return slow;
  }
};
```

Practise Now

Let us now discuss some easy question solving them using Two Pointers techniques

1. Palindrome Check : https://leetcode.com/problems/valid-palindrome/
2. Reverse Vowels : https://leetcode.com/problems/reverse-vowels-of-a-string/