Sample Problems on Array

Problem #1 : Range Sum Queries using Prefix Sum

Description : We are given an Array of **n** integers, We are given **q** queries having indices **l and r** . We have to find out sum between the given range of indices.

```
Input
[4, 5, 3, 2, 5]
3
0 3
2 4
1 3
Output
14 (4+5+3+2)
10 (3+2+5)
10 (5+3+2)
```

Solution : The numbers of queries are large. It will be very inefficient to iterate over the array and calculate the sum for each query separately. We have to devise the solution so that we can get the answer of the query in constant time. We will be storing the sum upto a particular index in prefix sum Array. We will be using the prefix sum array to calculate the sum for the given range.

```
prefix[] = Array stores the sum (A[0]+A[1]+....A[i]) at ind
ex i.
if l == 0 :
    sum(l,r) = prefix[r]
else :
    sum(l,r) = prefix[r] - prefix[l-1]
```

Pseudo Code

```
// n : size of array
// q : Number of queries
// l, r : Finding Sum of range between index l and r
// l and r (inclusive) and 0 based indexing
```

```
void range_sum(arr, n)
{
    prefix[n] = {0}
    prefix[0] = arr[0]
    for i = 1 to n-1 :
        prefix[i] = a[i] + prefix[i-1]

    for (i = 1 to q )
    {
        if (l == 0)
        {
            ans = prefix[r]
            print(ans)
        }
        else
        {
            ans = prefix[r] - prefix[l-1]
            print(ans)
        }
    }
}
```

Time Complexity : Max(O(n),O(q))

Auxiliary Space : O(n)

Problem #2 : Equilibrium index of an array

Description - Equilibrium index of an array is an index such that the sum of elements at lower indexes is equal to the sum of elements at higher indexes. We are given an Array of integers, We have to find out the first index **i** from left such that -

```
A[0] + A[1] + ... A[i-1] = A[i+1] + A[i+2] ... A[n-1]
```

```
Input
[-7, 1, 5, 2, -4, 3, 0]
Output
```

```
3
A[0] + A[1] + A[2] = A[4] + A[5] + A[6]
```

Naive Solution : We can iterate for each index i and calculate the leftsum and rightsum and check whether they are equal.

```
for (i=0 to n-1)
{
    leftsum = 0
    for (j = 0 to i-1)
        leftsum += arr[i]
    rightsum = 0
    for (j = i+1 to n-1)
        rightsum += arr[i]

    if leftsum == rightsum :
        return i
}
```

Time Complexity : O(n^2)

Auxiliary Space : O(1)

Tricky Solution : The idea is to first get the total sum of array. Then Iterate through the array and keep updating the left sum which is initialized as zero. In the loop, we can get the right sum by subtracting the elements one by one. Then check whether the Leftsum and the Rightsum are equal.

Pseudo Code

```
// n : size of array
int eqindex(arr, n)
{
    sum = 0
    leftsum = 0
    for (i=0 to n-1)
        sum += arr[i]

    for (i=0 to n-1)
    {
        // now sum will be righsum for index i
        sum -= a[i]
        if (sum == leftsum )
            return i
        leftsum += a[i]
    }
}
```

Time Complexity : O(n)

Auxiliary Space : O(1)

Problem #3 : Largest Sum Subarray

Description : We are given an array of positive and negative integers. We have to find the subarray having maximum sum.

```
Input
[-3, 4, -1, -2, 1, 5]
Output
7
(4+(-1)+(-2)+1+5)
```

Solution :

A simple idea is to look for all the positive contiguous segments of the array (max_ending_here is used for this), and keep the track of maximum sum contiguous segment among all the positive segments (max_so_far is used for this). Each time we get a positive sum compare it with max_so_far and if it is greater than max_so_far, update max_so_far.

Pseudo Code

```
//n : size of array
int largestsum(arr, n)
{
    max_so_far = INT_MIN
    max_ending_here = 0

    for (i=0 to n-1)
    {
        max_ending_here += arr[i]
        if max_so_far < max_ending_here :
            max_so_far = max_ending_here

        if max_ending_here < 0 :
            max_ending_here = 0
    }

    return max_so_far
}
```

Time Complexity : O(n)

Auxiliary Space : O(1)

Problem #4 : Merge two sorted Arrays

Description : We are given two sorted arrays **arr1[ ]** and **arr2[ ]** of size **m** and **n** respectively. We have to merge these arrays and store the numbers in arr3[ ] of size **m+n** .

```
Input
1 3 4 6
2 5 7 8
Output
1 2 3 4 5 6 7 8
```

Solution : The idea is to traverse both the arrays simultaneously and compare the current numbers from both the Arrays. Pick the smaller element and copy it to arr3[ ] and advance the current index of the array from where the smaller element is picked. When we reach at the end of one of the arrays, copy the remaining elements of another array to arr3[ ].

Pseudo Code

```
// input arrays - arr1(size m), arr2(size n)
void merge_sorted(arr1, arr2, m, n)
{
    arr3[m+n] // merged array
    i=0,j=0,k=0
    while(i < m && j < n)
    {
        if arr1[i] < arr2[j] :
            arr3[k++] = arr1[i++]
        else :
            arr3[k++] = arr2[j++]
    }
    while(i < m)
        arr3[k++] = arr1[i++]
    while(j < n)
        arr3[k++] = arr2[j++]
}
```

Time Complexity : O(m+n)

Auxiliary Space : O(m+n)