Arrays

Array is defined as an **ordered set of similar data items**.

Common Terms:

1. **Subarray** - Continuous part of the array i.e. elements from a given range of index L to R.
2. **Subsequence -** Set of some elements of the array in the order in which they are present in the array itself.
   For example, {A[0], A[2], A[5]} is a subsequence of the array A (array.length > 5).
   [*All subsequences are also subsets of the array.*]

Prefix Sum:

1. **What** is prefix-sum?
   Given an array of size n, the prefix sum is another array (say prefixSum) of the same size such that for each index 0 <= i < n,

   prefixSum[i] = a[0] + a[1] .... + a[i]. (It represents the sum of all elements from 0 to ith index)

2. **How** to calculate the prefix-sum?

   prefixSum[0] = a[0]
   for i → 1 to (array.length - 1) {
   prefixSum[i] = prefixSum[i-1] + a[i]
   }
   Time Complexity = O(N)
   Space Complexity = O(N) [prefixSum array]

3. **Where** to use prefix-sum?
   Prefix sum is used to solve problems where we have to find the **sum of elements in the given range** from index L to R for multiple queries.

   For example:
   Index → 0, 1, 2, 3, 4, 5, 6
   A = [3, -2, 8, -5, 4, 0, 1]
   prefixSum = [3, 1, 9, 4, 8, 8, 9]

   Find the sum of elements from index 2 to 5.
   Answer = A[2] + A[3] + A[4] + A[5] = 8 - 5 + 4 + 0 = 7

Better approach,
Answer = (A[0] + A[1] + A[2] + A[3] + A[4] + A[5]) - (A[0] + A[1])
= **prefixSum[5] - prefixSum[1]** = 8 - 1 = 7

Therefore, **sum of elements in the given range** of index L to R =

- prefixSum[R] - prefixSum[L - 1] , if L > 0
- prefixSum[R] , if L == 0

BONUS: The same idea of prefix-sum can also be used to calculate **prefix-xor**, which can be used to find the **xor of values in the index range L to R**. [*So basically, we can conclude the concept of prefix can be used whenever you want to calculate anything in different subarrays of Array again again and again.*]

Practice Question: Equilibrium index of an array

Carry Forward:

1. **What** is carry forward?
   Carry forward is the concept where we **use the data which is calculated at runtime** i.e. carry forward = calculation + use (without storing the data).

2. **How** to use carry forward?
   Similar to prefix sum the calculation can be done by **traveling the array and using** the data at run-time.

For example → Find the count of index pairs (i, j) such that i < j, A[i] is odd & A[j] is even, for a given integer array 'A'.

Solution: For every even element if we know the **count of odd elements** which are on the left side of the current element then we can add them up to get the answer.

How to know the count of odd elements which are on the left side?
Calculate it by traveling the array from left to right using a single variable.

```
countOdd = 0
answer = 0
for i → 0 to (array.length - 1) {
   if (A[i] % 2 == 1) // A[i] is odd element
      countOdd += 1
   else               // A[i] is even element
      answer += countOdd
}
return answer
```

Time Complexity = O(N)
Space Complexity = O(1)

Here, **countOdd** variable is calculated at runtime and is used without storing any previous calculated value hence **saving space** in memory.

3. **Where** to use carry forward?
   It is used in any situation where **calculation of any data is required** which is to be used in another calculation that leads to the final result. So, the carry forward variable will be used to do the first calculation and without storing the result, that value will be used in the final result calculation. For example, the *countOdd* variable above.

4. **Why** use carry forward?
   Carry forward is very useful in **space optimization** as seen in the above example. For using this concept no extra space is required.

Practice Question: Max Sum Contiguous Subarray

Sliding Window:

1. **What** is a sliding window?
   Here, window is basically a **fixed size subarray** of the given array.
   It is an algorithm where we can quickly compute the things which have a fixed window for calculation and we can fetch the result in an optimized manner rather than using the nested loops(naive approach). The main goal of this algorithm is to reuse the result of one window to compute the result of the next window.

2. **How** to use a sliding window?
   For the first subarray the computation can be done by traveling and after that one **new element can be added** and another **oldest element can be removed**.

   For example → Finding the maximum sum of any **subarray of length K** in the given integer **array of length N** (>= K).

3. **Where** to use the sliding window?
   Whenever any **computation of a fixed size subarray** with the given length is required then we should use a sliding window.

Practice Question: Subarray with given sum and length (You can find this question in **Arrays** lecture)

Contribution Technique:

1. **What** is the contribution technique?
   It is a technique used for questions where the answer can be calculated by **adding the contributions** of each element of the array.

2. **How** to use the contribution technique?
   It is used by **creating a mathematical formula** to define the contribution of one array element, and using that formula to add all the elements' contribution.

   > For example → Finding the **sum of all subarray sums** of the given array can be calculated like → **Sum of (A[i] x Number of subarrays in which A[i] is present)**

3. **Where** to use the contribution technique?
   It is very useful in cases where an **array element is contributing multiple times** in the answer calculation.

4. **Why use** the contribution technique?
   Contribution technique is very useful in time complexity optimization by **reducing the recomputation** of the contribution of each element.

Practice Question: Sum of all subarrays (You can find this problem in **Arrays** lecture)

First Missing Integer

Q- Given an integer array, find the first missing positive integer. There are no duplicates.

Example - 1:
A = [10, 15, 3, 2, 8]
1 is missing in the array A.

Example - 2:
A = [10, 3, 1, 2, 5, -8, -3, 4]
6 is the first missing positive integer.

Solution:

Sol - 1: Brute force
In the brute force approach, we can simply search for each number in the array starting from 1 to N. If all numbers are present, then the answer would be N + 1.

- Time complexity: $O(N*N)$
- Space complexity: $O(1)$

Optimised Soln:
We can improve the space complexity by modifying the array A itself. Since, there are no duplicates, whenever we encounter an element which is within the array [1, N], we simply swap it with the index.

Pseudo Code:

```
public class Solution {
    public int firstMissingPositive(ArrayList<Integer> A) {
        int n = A.size();
        for (int i = 0; i < n; i++) {
            if (A.get(i) > 0 && A.get(i) <= n) {
                int pos = A.get(i) - 1;
                if (A.get(pos) != A.get(i)) {
                    Collections.swap(A, pos, i);
                    i--;
                }
            }
        }
        for (int i = 0; i < n; i++) {
            if (A.get(i) != i + 1)
                return (i + 1);
        }
        return n + 1;
    }
}
```

Revision Video - Arrays

P.s. Please watch the revision video to do a comprehensive revision of All the array lectures in 20 minutes.

Congratulations, You are one step closer to clearing your Mock Interview.