Arrays

Basics:

1. Array is defined as an **ordered set of similar data items**.
2. All the data items of an array are stored in **consecutive memory locations in RAM**.
3. The elements of an array are of the **same data type** and each item can be accessed using the same name.
4. Any element of the array can be **accessed by using the index(position)** of the element in **O(1) time complexity**. Index starts from 0 to (array.length - 1) therefore, **a[2] is the third element** of the array.

Common Terms:

1. **Subarray** - Continuous part of the array i.e. elements from a given range of index L to R.
2. **Subset of the array -** Set of some elements of the array in any order.
   For example, {A[2], A[5], A[0]} is a subset of the array A (array.length > 5).
3. **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.*]
4. **Matrix / Two-dimensional Array** - A two-dimensional array is an array of arrays i.e., it's a collection of arrays in which elements are arranged in rows and columns (tabular format). We can access the elements using both the row index and column index, i.e. **A[row_index] [column_index]**

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]**.

2. **How** to calculate the prefix-sum?
   - prefixSum[2] = a[0] + a[1] + a[2]
   - prefixSum[3] = a[0] + a[1] + a[2] + a[3] = prefixSum[2] + a[3]
   - Therefore, for all indexes i > 0, **prefixSum[i] = prefixSum[i-1] + a[i]**

```
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**. [*Try it*]

4. **Why** to use prefix-sum?
   Calculating sum of elements from index L to R by traveling will take the time complexity as linear i.e. O(N) whereas if prefixSum is used then the **time complexity will be constant** i.e.O(1) since the answer is calculated by subtracting 2 elements from prefixSum.

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)

2-D Array/Matrix

Given a matrix A of integers, find the sum of a given submatrix. There would be multiple sum queries.

Example:
Suppose the matrix is:

```
   [[1, 3, 5, 2, -1],
    [4, 8, 5, 0, 6],
    [10, 20, -1, 3, 5],
    [1, 5, -5, 10, 6]]
```

Suppose you have to find the sum of the matrix with top left coordinate (1, 0) and bottom right coordinate (3, 2).
The elements in the submatrix are and their sum are:
4 + 8 + 5 + 10 + 20 - 1 + 1 + 5 - 5 = 47

Solution:

Brute force:
In the brute force approach, we would simply answer each query by iterating the submatrix given to us and finding the sum.

Pseudo Code:

```
Function each_query(mat,x,y,u,v){
        sum=0
        for( i=x to u)
              for(j=y to v)
                   sum=sum+mat[i][j]
        }
return sum
```

- Time complexity for each query: O(N * M)
- Space complexity for each query: O(1)

Optimization:
We can use the prefix sum technique, we used in case of an array to solve each query in O(1), with precalculation in O(N * M).

Pseudo Code:

```
function calculate_prefix_matrix(mat):
    prefix_mat = mat
    N = rows in mat
    M = cols in mat

    for i from 1 to N:
        for j from 1 to M:
            prefix_mat[i][j] = prefix_mat[i - 1][j] + prefix_mat[i][j
- 1] - prefix_mat[i -1][j - 1]

    return prefix_mat

function solve(mat, queries)
    prefix_mat = calculate_prefix_matrix(mat)
    for (x, y, u, v) in queries:
        // handle negative values
        answer = prefix_mat[u][v] - prefix_mat[u][y - 1] -
prefix_mat[x - 1][v] + prefix_mat[x - 1][y - 1]
        print(answer)
```

- Time complexity: O(1) per query, with precalculation of O(N * M)
- Space complexity: O(N * M)

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 answer would be N + 1.

- Time complexity: $O(N_2)$
- Space complexity: O(1)

Sol - 2:
We will simply sort the array. Now, starting from index 0, we will find the first positive integer. Now, the numbers would be consecutive starting from 1. We will check them and report whenever we find a missing number.

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

Sol - 3:
Use extra array namely visited array to track elements from 1 to N.

- Time complexity: O(N)
- Space complexity: O(N)

Sol - 4:
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 -

https://drive.google.com/drive/folders/1Ww4p2egf541Vz8JUj0KTVBCyqYFZ1RMB?usp=share_link