Length of the longest subarray with zero Sum

Problem Statement: Given an array containing both positive and negative integers, we have to find the length of the longest subarray with the sum of all elements equal to zero.

Example 1:

```
Input Format: N = 6, array[] = {9, -3, 3, -1, 6, -5}

Result: 5

Explanation: The following subarrays sum to zero:

{-3, 3} , {-1, 6, -5}, {-3, 3, -1, 6, -5}

Since we require the length of the longest subarray, our answer is 5!
```

Example 2:

```
Input Format: N = 8, array[] = {6, -2, 2, -8, 1, 7, 4, -10}

Result: 8

Subarrays with sum 0 : {-2, 2}, {-8, 1, 7}, {-2, 2, -8, 1, 7}, {6, -2, 2, -8, 1, 7, 4, -10}

Length of longest subarray = 8
```

Example 3:

```
Input Format: N = 3, array[] = {1, 0, -5}

Result: 1

Subarray : {0}

Length of longest subarray = 1
```

Example 4:

```
Input Format: N = 5, array[] = {1, 3, -5, 6, -2}

Result: 0

Subarray: There is no subarray that sums to zero
```

Solution

***Disclaimer**: Don't jump directly to the solution, try it out yourself first.*

Solution 1 (Naive approach) :

Intuition:

We are required to find the longest subarray that sums to zero. So our initial approach could be to consider all possible subarrays of the given array and check for the subarrays that sum to zero. Among these valid subarrays (sum equal to zero) we'll return the length of the largest subarray by maintaining a variable, say max.

Approach :

1. Initialise a variable max = 0, which stores length of longest subarray with sum 0.
2. Traverse the array from start and initialise a variable sum = 0 which stores sum of subarray starting with current index
3. Traverse from next element of current_index up to end of the array, each time add the element to sum and check if it is equal to 0.
    0. If sum = 0, check if length of subarray so far is > max and if yes update max
4. Now keep adding elements and repeat step 3 a.
5. After the outer loop traverses all elements return max

Source Code:

```
static int solve(int[] a){
        int  max = 0;
        for(int i = 0; i < a.length; ++i){
                int sum = 0;
                for(int j = i; j < a.length; ++j){
                        sum += a[j];
                        if(sum == 0){
                                max = Math.max(max, j-i+1);
                        }
                }
        }
        return max;
}
```

Time Complexity: O(N^2) as we have two loops for traversal

Space Complexity : O(1) as we aren't using any extra space.

Can this be done in single traversal? Let's check 🙂

Solution 2 (Optimised Approach) :

Intuition: Now let's say we know that the sum of subarray(i, j) = S, and we also know that sum of subarray(i, x) = S where i < x < j. We can conclude that the sum of subarray(x+1, j) = 0.

Let us understand the above statement clearly

So in this problem, we'll store the prefix sum of every element, and if we observe that 2 elements have same prefix sum, we can conclude that the 2nd part of this subarray sums to zero

Now let's understand the approach

Approach:

1. First let us initialise a variable say **sum = 0** which stores the sum of elements traversed so far and another variable say **max = 0** which stores the length of longest subarray with sum zero.
2. Declare a HashMap<Integer, Integer> which stores the prefix sum of every element as key and its index as value.
3. Now traverse the array, and add the array element to our sum.

(i) If sum = 0, then we can say that the subarray until the current index has a sum = 0,   so we update max with the maximum value of (max, current_index+1)

(ii) If sum is not equal to zero then we check hashmap if we've seen a subarray with this sum before if HashMap contains sum -> this is where the above-discussed case occurs (subarray with equal sum), so we update our max

else -> Insert (sum, current_index) into hashmap to store prefix sum until current index

1. After traversing the entire array our max variable has the length of longest substring having sum equal to zero, so return max.

NOTE : we do not update the index of a sum if it's seen again because we require the length of the ***longest*** subarray

Dry Run: Let us dry run the algorithm for a better understanding

```
Input : N = 5, array[] = {1, 2, -2, 4, -4}



Output : 5
```

1. Initially sum = 0, max = 0

2. HashMap<Integer, Integer> h = [ ];

3. => **i=0** -> **sum=1**, sum!=0 so check in hashmap if we've seen a subarray with this sum before, in our case hashmap does not contain sum (1), so we add (sum, i) to hashmap.

h = [[1,0]]

=> **i=1** -> **sum=3**, sum!=0 so check in hashmap if we've seen a subarray with this sum before, in our case hashmap does not contain sum (3), so we add (sum, i) to hashmap.

h=[[1,0], [3,1]]

=> **i=2** -> **sum=1**, sum!=0 so check in hashmap if it contains sum, in our case hashmap contains sum (1)

Here we can observe that our current sum is seen before which concludes that it's a zero subarray!

So we now update our max as maximum(max, 2-0) => **max = 2**

h=[[1,0], [3,1]]

=> **i=3** -> **sum=5** , sum!=0 so check in hashmap if it contains sum, in our case hashmap does not contain sum (5), so we add (sum, i) to hashmap.

h=[[1,0], [3,1], [5,3]]

 => **i=4** -> **sum=1**, sum!=0 so check in hashmap if it contains sum, in our case hashmap contains sum (1)

Here we can again observe our above discussed case

So we now update our max as maximum(max, 4-0) => **max = maximum(2, 4) = 4**

h=[[1,0], [3,1], [5,3]]

4. Now that we have traversed our whole array, our answer is max = 4

Subarray = {2, -2, 4, -4}

Source Code:

```
int maxLen(int A[], int n)
    {
        HashMap<Integer, Integer> mpp = new HashMap<Integer, Integer>();
        int maxi = 0;
        int sum = 0;
        for(int i = 0;i<n;i++) {
            sum += A[i];
            if(sum == 0) {
                maxi = i + 1;
            }
            else {
                if(mpp.get(sum) != null) {
                    maxi = Math.max(maxi, i - mpp.get(sum));
                }
                else {
                    mpp.put(sum, i);
                }
            }
        }
        return maxi;
    }
```

Time Complexity: O(N), as we are traversing the array only once

Space Complexity: O(N), in the worst case we would insert all array elements prefix sum into our hashmap