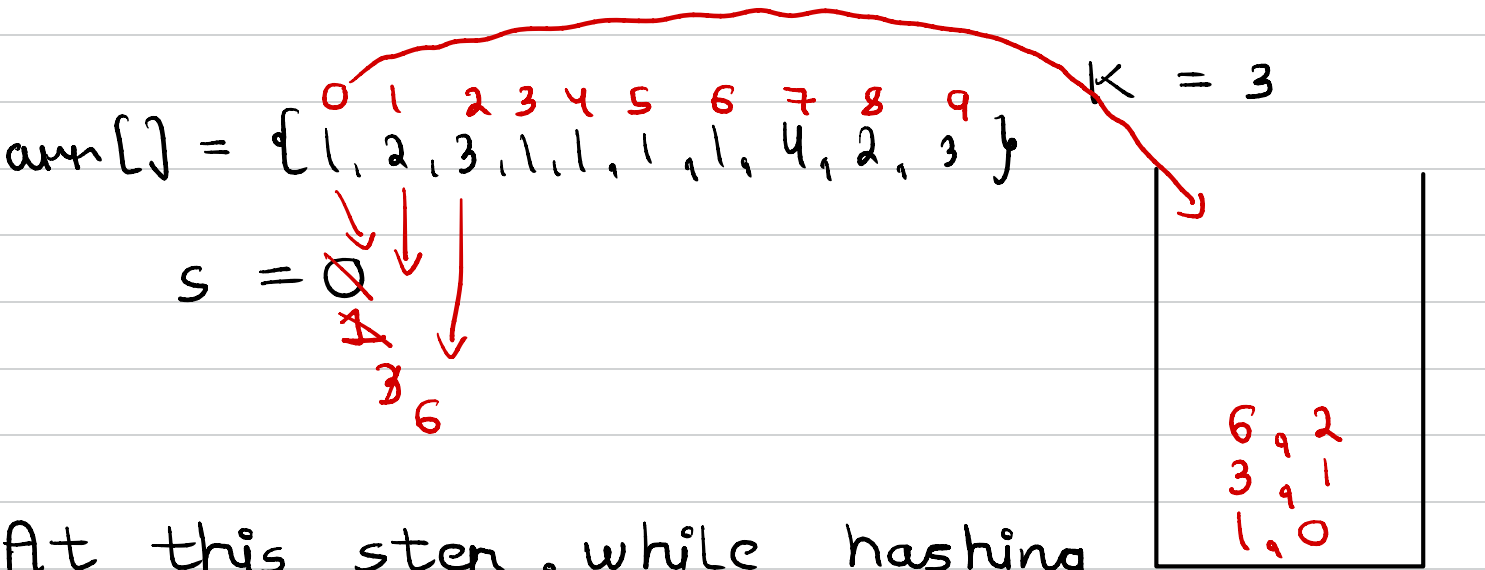
LONGEST SUBARRAY WITH A GIVEN SUM

☆ In this problem, we are supposed to return the length of the longest subarray which has a given sum.

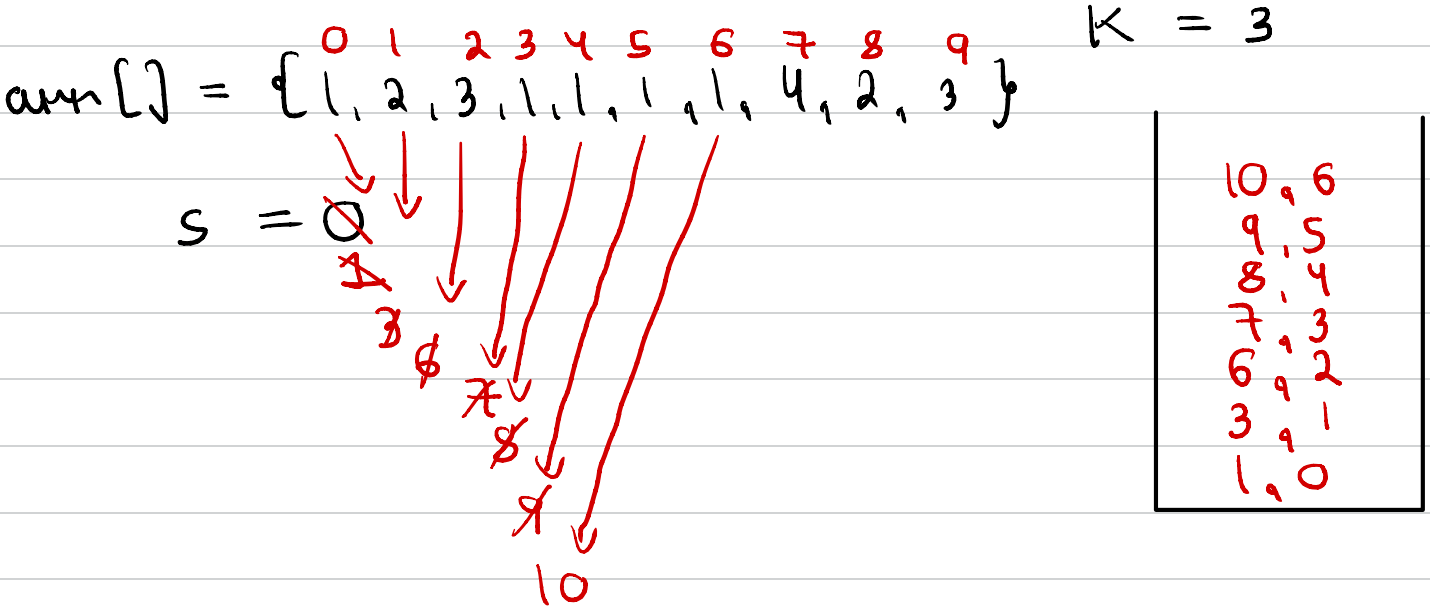
Brute force solution is to find all subarrays along with their sums and keep track of their longest one. But for array of size N, N! subarrays exist, hence this is too much time complexity.

Better approach to the brute force solution is to cache sum of smaller subarrays.
Better solution is to use hashing.
We will use each element as the last element.



At this step, while hashing the sum we realize that we have required sum as the difference

between two hashes. Hence that is one of our desired subarrays although not the longest.



Again, we stumble across a part where difference is equal to required sum. Hence we got another subarray.

Pseudocode :

```
longestSubArrayWithSumK(arr, N, K) {
    map preSumMap
    sum, len = 0, 0
    for(int i = 0 → N) {
        sum += arr[i]
        if(sum == K) {
            len = max(len, i + 1)
        }
        rem = sum - K
        if(preSumMap.find(rem)
                != preSumMap.end()){
            l = i - preSumMap(rem)
            len = max(len, l)
        }
```

```
            preSumMap[sum] = i
        }
        return len
    }
```

There is a problem if the array has zeros. To adapt, what we do is skip zeroes. This is the best solution if negative elements are also to be accounted for.

If we only talk about non negatives, the 2 pointer approach is optimal. We start with 2 pointers at position 0. If summation is lesser than required, we move j by one if it is equal then we store and if it is more we move i by one.

Pseudocode :

```
longestSubArrayWithSumK(arr, N, K) {
    i, j = 0, 0
    sum = arr[0]
    len = 0
    while (j < N) {
        while (i <= j && sum > K) {
            sum -= arr[i]
            i++
        }
        if (sum == K) {
            len = max(len, j - i + 1)
        }
```

```
        j++
        if (j < N)  sum += a[j]
    }
    return len
}
```