Technique: Prefix Sums

Level: Medium

Given an array of integers, both -ve and +ve, find a contiguous subarray that sums to 0.

For example: [2,4,-2,1,-3,5,-3] --> [4,-2,1,-3]

Questions to Clarify:
Q. How should I return the output?
A. Return the starting and ending indices of the subarray.

Q. What to return if the array is empty or null?
A. Return null.

Q. What to return if no subarray is found?
A. Return null.

Q. What to do if there are multiple subarrays?
A. Return any one.

Solution:
The brute force algorithm (going through each subarray) takes $O(n^2)$ time and O(1) space.
We can solve this problem with O(n) time and O(n) space. This is definitely better in terms of time, but make sure you mention to the interviewer that it takes more space.

We use the technique of prefix sums. For all elements, we first calculate the sum *s[i]* - which is the sum of all numbers from *a[0]* to *a[i]*. An interesting property emerges: If any *s[i]* is 0, then *a[0]* to *a[i]* sums to 0, so the subarray *[0..i]* is the answer.

If there is no *s[i]* that equals zero, we try to find two *s[i]*'s that have the same value.
For any *j* and *k*, if *s[j]* and *s[k]* have the same value, then the sum of subarray *[j + 1..k]* is 0.

Example:
If A = [2,4,-2,1,-3,5,-3],
Prefix Sums => [2,6,4,5,2,7,4]

We see that the prefix sums have two duplicates: 2 (index 0 & 4) and 4 (index 2 & 6).
This means that there are two subarrays with sum 0 -> *a[1..4]* and *a[3..6]*

To check if we've found a duplicate prefix sum, we store all previous prefix sums in a hashmap/hashtable. This way, we can quickly look up if we've seen this sum before.

Pseudocode:

```
(Note: Never write pseudocode in an actual interview. Unless you're writing a
few lines quickly to plan out your solution. Your actual solution should be in
```

```
a real language and use good syntax.)
```

We loop through the array, and at each index *i*, we update prefix sum. If we encounter sum 0 at *i*, we return *[0..i]* as answer. We also keep a map of old sums. If we find a sum again (the old sum being at index *x*), we return the subarray *[x+1..i]* as the answer.

```
sum = 0
hashmap map = {}
for i -> 0 to a.length - 1:
     sum = sum + a[i]

     if sum == 0:
         return [0..i]

     if map already has sum:
         return [map.get(sum)+1..i]

     add (sum, i) to map

if nothing found, return null
```

Test Cases:
Edge Cases: Empty Array, Null array
Base Cases: single element (-ve, +ve, 0)
Regular Cases: has sum, does not have sum, has sum at beginning/end/middle

Time Complexity: O(n)

Space Complexity: O(n)

```
public static Pair<Integer> zeroSumSubarray(int[] a) {
    if (a == null || a.length == 0)
        return null;

    int sum = 0;
    HashMap<Integer, Integer> map = new HashMap<>();

    for (int i = 0; i < a.length; i++) {
        sum += a[i];

        if (sum == 0) {
            return new Pair<Integer>(0,i);
        }

        if (map.containsKey(sum)) {
            return new Pair<Integer>(map.get(sum) + 1, i);
```

```
        }

        map.put(sum, i);
    }

    return null; // not found
}
```