Subarray Sum Equals K

Last modified @ 14 October 2020

Algorithm

In this article, we will learn to resolve the Subarray Sum Equals K problems by using Brute force, Sliding window, and Hash table algorithms

Problem 1

- Given an unsorted array of non-negative integers `a[n]` and an integer `k`
- Find a continuous sub array whose sum equals to `k`

Example 1.1

- Input: a = [4, 0, 11, 6, 1, 7] and k = 8
- Output: [1, 7]

Approach 1.1: Brute Force

- Use 2 for-loop to consider the sum of every continuous subarray
- Return the first subarray when its sum equals to `k`

```
import java.util.Arrays;

public class SubarrayGivenSumBruteforce {
    public static int[] findSubarray(int[] a, int k) {
        for(int i = 0; i < a.length; i++) {

            int currentSum = a[i];

            for(int j = i + 1; j <= a.length; j++) {

                if (currentSum == k) {
                    return Arrays.copyOfRange(a, i, j);
                } else if (currentSum > k || j == a.length) {
                    break;
                }
```

```
                currentSum += a[j];
            }
        }

        return null;
    }

    public static void main (String[] args) {
        int[] a = {4, 0, 11, 6, 1, 7};
        int k = 8;
        System.out.println(Arrays.toString(findSubarray(a, k)));
    }
}
```

- Time complexity: O(n^2) as with each element in the given array, we iterate over the remaining elements to calculate the sum of possible subarrays
- Space complexity: O(1)

Approach 1.2: Sliding Window

- Find the sliding window sum, say `ws`, in the index range `[i, j]` of the given array `a[n]`, that equals `k` by increasing `j` continuously from `i` to `n-1`
- Return `ws` as soon as it equals `k`, otherwise, reduce `ws` and increase `i`

```
import java.util.Arrays;

public class SubarrayGivenSumWindowSliding {
    public static int[] findSubarray(int[] a, int k) {
        int windowSum = 0, i = 0, j = 0;

        while (i < a.length) {
            while (j < a.length && windowSum < k) {
                windowSum += a[j++];
            }

            if (windowSum == k) {
                return Arrays.copyOfRange(a, i, j);
            }

            windowSum -= a[i++];
        }

        return null;
    }

    public static void main (String[] args) {
        int[] a = {4, 0, 11, 6, 1, 7};
```

```
        int k = 8;
        System.out.println(Arrays.toString(findSubarray(a, k)));
    }
}
```

- Time complexity: O(n)
- Space complexity: O(1)

Problem 2

- Given an unsorted array of integers `a[n]` and an integer `k`
- Find the total number of continuous subarrays whose sum equals to `k`

Example 2.1

- Input: a = [-4, 12, -11, 6, 1, 7], k = 8
- Output: 3
- Explanation: [-4, 12], [12, -11, 6, 1], [1, 7] are 3 subarrays have sum equals to 8

Example 2.2

- Input: a = [0, 0, 0], k = 0
- Output: 6
- Explanation: There are three subarrays [0] at index 0, 1, and 2, two subarrays [0, 0] at index [0, 1] and [1, 2], and one subarray [0, 0, 0] have sum equals to 0

Approach 2: Hash table

- Say c_i and c_j are the cumulative sums at index `i` and `j` in the given array `a[n]`

  Elements from `i` to `j` will form a subarray that satify the constraint if and only if $\mathbf{c_j - c_i = k}$ ($\forall$ 0 <= i < j < n)

- At index `j`, there may have multiple subarrays satisfy the above constraint. To reserve the count value of them, we can use a hash table to store c_j as key, and the count of the right subarrays until `j` (c_j-k) as value

- Use a variable `cumulativeCount` to track the cumulative count of all contiguous subarrays, every time we found a new right subarray
- Return `cumulativeCount` as the final result

```
import java.util.HashMap;
import java.util.Map;

public class SubarrayGivenSumHashtable {
    public static int countSubArraysWithHashTable(int[] a, int k) {
        int cumulativeCount = 0;
        int cumulativeSum = 0;
        Map<Integer, Integer> map = new HashMap<>();
        map.put(0, 1);

        for (int value : a) {
            cumulativeSum += value;
            if (map.containsKey(cumulativeSum - k)) {
                cumulativeCount += map.get(cumulativeSum - k);
            }

            map.put(cumulativeSum, map.getOrDefault(cumulativeSum, 0) + 1);
        }

        return cumulativeCount;
    }

    public static void main (String[] args) {
        int[] a = {-4, 12, -11, 6, 1, 7};
        int k = 8;
        System.out.println(countSubArraysWithHashTable(a, k));

        int[] b = {0, 0, 0};
        k = 0;
        System.out.println(countSubArraysWithHashTable(b, k));
    }
}
```

- Output

```
3
6
```

- Time complexity: O(n) as the given array is traversed through only once
- Space complexity: O(n) as the hash table can hold up to `n` elements

Algorithm