523. Continuous Subarray Sum

April 11, 2017 | 60.7K views

Given a list of **non-negative** numbers and a target **integer** k, write a function to check if the array has a continuous subarray of size at least 2 that sums up to a multiple of **k**, that is, sums up to n*k where n is also an **integer**.

Example 1:

```
Input: [23, 2, 4, 6, 7],  k=6
Output: True
Explanation: Because [2, 4] is a continuous subarray of size 2 and sums up to 6.
```

Example 2:

```
Input: [23, 2, 6, 4, 7],  k=6
Output: True
Explanation: Because [23, 2, 6, 4, 7] is an continuous subarray of size 5 and sums up
```

Constraints:

- The length of the array won't exceed 10,000.
- You may assume the sum of all the numbers is in the range of a signed 32-bit integer.

Solution

Approach #1 Brute Force [Time Limit Exceeded]

The brute force approach is trivial. We consider every possible subarray of size greater than or equal to 2, find out its sum by iterating over the elements of the subarray, and then we check if the sum obtained is an integer multiple of the given k.

Java

Copy

```
public class Solution {
    public boolean checkSubarraySum(int[] nums, int k) {

        for (int start = 0; start < nums.length - 1; start++) {
            for (int end = start + 1; end < nums.length; end++) {
                int sum = 0;
                for (int i = start; i <= end; i++)
                    sum += nums[i];
                if (sum == k || (k != 0 && sum % k == 0))
                    return true;
            }
        }
        return false;
    }
}
```

Complexity Analysis

- Time complexity : $O(n^2)$. Three for loops iterating over the array are used.
- Space complexity : $O(1)$. Constant extra space is used.

Approach #2 Better Brute Force [Accepted]

Algorithm

We can optimize the brute force approach to some extent, if we make use of an array sum that stores the cumulative sum of the elements of the array, such that $sum[i]$ stores the sum of the elements upto the i^{th} element of the array.

Thus, now as before, we consider every possible subarray for checking its sum. But, instead of iterating over a new subarray everytime to determine its sum, we make use of the cumulative sum array. Thus, to determine the sum of elements from the i^{th} index to the j^{th} index, including both the corners, we can use: $sum[j] - sum[i] + nums[i]$.

Java

Copy

```
public class Solution {
    public boolean checkSubarraySum(int[] nums, int k) {
        int[] sum = new int[nums.length];
        sum[0] = nums[0];
        for (int i = 1; i < nums.length; i++)
            sum[i] = sum[i - 1] + nums[i];
        for (int start = 0; start < nums.length - 1; start++) {
            for (int end = start + 1; end < nums.length; end++) {
                int sum = sum[end] - sum[start] + nums[start];
                if (sum == k || (k != 0 && sum % k == 0))
                    return true;
            }
        }
        return false;
    }
}
```

Complexity Analysis

- Time complexity : $O(n^2)$. Two for loops are used for considering every subarray possible.
- Space complexity : $O(n)$. sum array of size n is used.

Approach #3 Using HashMap [Accepted]

Algorithm

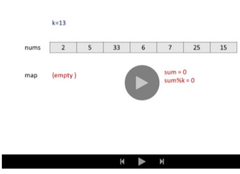
In this solution, we make use of a HashMap that is used to store the cumulative sums upto the i^{th} index after some processing along with the index i. The processing done is taking the modulus of the the sum upto the i^{th} index with the given k. The reasoning behind this will become clear soon.

We traverse over the given array, and keep on calculating the sum values upto the current index. Whenever we find a new sum value, which isn't present in the HashMap already, we make an entry in the HashMap of the form, $(sum$.

Now, assume that the given sum value at the i^{th} index be equal to rem. Now, if any subarray follows the j^{th} element, which has a sum equal to the integer multiple of k, say extending upto the j^{th} index, the sum value to be stored in the HashMap for the j^{th} index will be: $(rem + n * k)$, where n is some integer > 0. We can observe that $(rem + n * k)$, which is the same value as stored corresponding to the i^{th} index.

From this observation, we come to the conclusion that whenever the same sum value is obtained corresponding to two indices i and j, it implies that sum of elements between those indices is an integer multiple of k. Thus, if the same sum value is encountered again during the traversal, we return a True directly.

The slideshow below depicts the process for the array `nums: [2, 5, 33, 6, 7, 25, 15]` and `k=13`.



1 / 6

Java

Copy

```
public class Solution {
    public boolean checkSubarraySum(int[] nums, int k) {
        int sum = 0;
        HashMap < Integer, Integer > map = new HashMap < > ();
        map.put(0, -1);
        for (int i = 0; i < nums.length; i++) {
            sum += nums[i];
            if (k != 0)
                sum = sum % k;
            if (map.containsKey(sum)) {
                if (i - map.get(sum) > 1)
                    return true;
            } else
                map.put(sum, i);
        }
        return false;
    }
}
```

Complexity Analysis

- Time complexity : $O(n)$. Only one traversal of the array $numa$ is done.
- Space complexity : $O(min(n, k))$. The HashMap can contain upto $min(n, k)$ different pairings.

Rate this article:

Previous Next

Comments: 68

Sort By

Type comment here... (Markdown is supported)

Preview Post

MissionPrep ★ 183 December 25, 2018 12:17 AM

Description of Approach#3 is horrible....

160 Share Reply

SHOW 6 REPLIES

warm2000 ★ 123 January 7, 2019 1:29 AM

isn't the theory behind 3rd solution just as simple as this:

a%k = x

b%k = x

(a - b) %k = x - x = 0

here a - b = the sum between i and j

Read More

93 Share Reply

SHOW 5 REPLIES

jkwak ★ 82 March 4, 2019 12:33 AM

Is there any chance Leetcode could get an English native speaker to do some proof-reading of the articles?

There's no point in writing an article if readers cannot understand it

44 Share Reply

SHOW 1 REPLY

haoyangfan ★ 912 December 30, 2018 12:15 AM

I hope this explanation will help you better understand solution 3 if you are still confused

```
...
we can define a "group" relationship on the integers
```

Read More

28 Share Reply

SHOW 8 REPLIES

zyx266 ★ 35 January 18, 2019 5:13 AM

brute force can achieve O(n^2) and O(1). There is no need to have three loops

35 Share Reply

SHOW 1 REPLY

neildawg ★ 165 January 22, 2019 7:19 AM

For anybody is confused about `map.put(0,-1);`

In the case nums = [1, 5] k = 6, at i=1, `sum % k` is 0, so we need a key '0' in the map, and it must be comply with the continuous condition, `i - map.get(sum) > 1`, so we give an arbitrary value of -1.

34 Share Reply

SHOW 2 REPLIES

learnSomeCode ★ 60 September 17, 2018 9:03 PM

i really do not understand what you are trying to say for Approach #3, you got some typos/grammar that is hard to follow

21 Share Reply

Three_Thousand_world ★ 1280 March 18, 2019 2:30 AM

only jerk would ask you this problem and expect solution 3.

43 Share Reply

Haseeb92 ★ 7 September 4, 2018 8:54 AM

The Hashmap solution fails for the test case:

[15,0,0,3]

4

7 Share Reply

SHOW 3 REPLIES

lenchen1112 ★ 1007 December 23, 2018 5:13 PM

Clean Python3 of approach 3:

```
import itertools
class Solution:
    def checkSubarraySum(self,nums: List[int], k: int) -> bool:
```

Read More

3 Share Reply