Continuous Subarray Sum

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 the 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 to 42.
```

Note:

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

Solution 1

We iterate through the input array exactly once, keeping track of the running sum mod k of the elements in the process. If we find that a running sum value at index j has been previously seen before in some earlier index i in the array, then we know that the sub-array (i,j] contains a desired sum.

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

written by compton_scatter original link here

Solution 2

- `if k == 0`
  If there are two continuous zeros in `nums`, return `True`

  Time O(n).
- `if n >= 2k and k > 0`
  There will be at least three numbers in `sum` with the same remainder divided by `k`. So I can return True without any extra calculation.

  Time O(1).
- `if n < 2k and k > 0`
  If I can find two numbers in `sum` with the same remainder divided by `k` and the distance of them is greater than or equal to 2， return `True`.

  Time O(n) <= O(k).
- `k < 0`
  same as `k > 0`.

```
class Solution(object):
    def checkSubarraySum(self, nums, k):


        if k == 0:
            # if two continuous zeros in nums, return True
            # time O(n)
            for i in range(0, len(nums) - 1):
                if nums[i] == 0 and nums[i+1] == 0:
                    return True
            return False

        k = abs(k)
        if len(nums) >= k * 2:
            return True

        #if n >= 2k: return True
        #if n < 2k:  time O(n) is O(k)

        sum = [0]
        for x in nums:
            sum.append((sum[-1] + x) % k)

        Dict = {}
        for i in range(0, len(sum)):
            if Dict.has_key(sum[i]):
                if i - Dict[sum[i]] > 1:
                    return True
            else:
                Dict[sum[i]] = i

        return False
```

written by zym_ustc original link here

Solution 3

This problem contributed a lot of bugs to my contest score... Let's read the description again, pay attention to `red` sections:

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 the `multiple` of k, that is, sums up to n*k where n is also an `integer`.

Some `damn it!` test cases:

1. [0], 0 -> false;
2. [5, 2, 4], 5 -> false;
3. [0, 0], 100 -> true;
4. [1,5], -6 -> true;
   etc...

```
public class Solution {
    public boolean checkSubarraySum(int[] nums, int k) {
        // Since the size of subarray is at least 2.
        if (nums.length <= 1) return false;
        // Two continuous "0" will form a subarray which has sum = 0. 0 * k == 0
will always be true.
        for (int i = 0; i < nums.length - 1; i++) {
            if (nums[i] == 0 && nums[i + 1] == 0) return true;
        }

        // At this point, k can't be "0" any longer.
        if (k == 0) return false;
        // Let's only check positive k. Because if there is a n makes n * k = sum
, it is always true -n * -k = sum.
        if (k < 0) k = -k;

        Set<Integer> sums = new HashSet<>();
        int sum = 0;
        sums.add(0);

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

            if (i > 0) {
                // Validate from the biggest possible n * k to k
                for (int j = (sum / k) * k; j >= k; j -= k) {
                    if (sums.contains(sum - j)) return true;
                }
            }

            sums.add(sum);
        }

        return false;
    }
}
```

written by shawngao original link here