<https://leetcode.com/problems/subarray-sum-equals-k/>

Given an array of integers nums and an integer k, return *the total number of subarrays whose sum equals to* k.

A subarray is a contiguous **non-empty** sequence of elements within an array.

**Example 1:**

Input: nums = [1,1,1], k = 2

Output: 2

**Example 2:**

Input: nums = [1,2,3], k = 3

Output: 2

**Constraints:**

* 1 <= nums.length <= 2 \* 104
* -1000 <= nums[i] <= 1000
* -107 <= k <= 107

**Attempt 1: 2023-02-04**

**Solution 1:  Native for loop with 2 passes (30 min, first pass is create preSum array, second pass is calculate interval with nested for loop)**

class Solution {

public int subarraySum(int[] nums, int k) {

int[] preSum = new int[nums.length + 1];

preSum[0] = 0;

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

preSum[i] = preSum[i - 1] + nums[i - 1];

}

int count = 0;

for(int start = 0; start < preSum.length; start++) {

for(int end = start + 1; end < preSum.length; end++) {

if(preSum[end] - preSum[start] == k) {

count++;

}

}

}

return count;

}

}

Time Complexity: O(n^2)

Space Complexity: O(n)

**Solution 2:  Hash Table (30 min)**

class Solution {

public int subarraySum(int[] nums, int k) {

Map<Integer, Integer> map = new HashMap<Integer, Integer>();

//map.put(0, 0);

map.put(0, 1);

int[] preSum = new int[nums.length + 1];

int count = 0;

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

preSum[i] = preSum[i - 1] + nums[i - 1];

if(map.containsKey(preSum[i] - k)) {

count += map.get(preSum[i] - k);

}

map.put(preSum[i], map.getOrDefault(preSum[i], 0) + 1);

}

return count;

}

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to**

<https://leetcode.com/problems/subarray-sum-equals-k/solutions/803317/java-solution-with-detailed-explanation/>

## **Thinking**

1. Use an array to store the sum accumulated from the beginning to a certain position.

Example:

nums = [1, 2, 3 ]

sum = [1, 1+2, 1+2+3]

2. How to create array "sum" ?

sum[i] = sum[i - 1] + nums[i]

Q : If i == 0, the index is out of range. How to solve this problem ?

A : Set the first element of the array "sum" to 0, and initialize the array "sum" **from index 1 rather than 0**.

nums = [1, 2, 3 ]

sum = [0, 1, 1+2, 1+2+3] // Also, the length of "sum" is one more than "nums"

sum[i] = sum[i - 1] + nums[i - 1]

// Java Version

int[] sum = new int[nums.length + 1];

sum[0] = 0;

for (int i = 1; i < (nums.length + 1); i++)

sum[i] = sum[i - 1] + nums[i - 1];

3. Using array "sum" to calculate the sum of a subarray

sumOfSubarray = sum[end] - sum[start];

For example : Calculate the sum of "nums" means using the last element of "sum" minus the first element of "sum" which is 0.

nums[0] + nums[1] + nums[2] = sum[3] - sum[0] = 6 - 0

4.Using array "sum"to caculate all possibilities .

## **Code**

public class Solution {

public int subarraySum(int[] nums, int k) {

int count = 0;

int[] sum = new int[nums.length + 1];

sum[0] = 0;

for (int i = 1; i <= nums.length; i++)

sum[i] = sum[i - 1] + nums[i - 1];

for (int start = 0; start < sum.length; start++) {

for (int end = start + 1; end < sum.length; end++) {

if (sum[end] - sum[start] == k)

count++;

}

}

return count;

}

}

## **Complexity Analysis**

* Time complexity : *O(n2)*.
* Space complexity : *O(n)*.

## **Optimization by Hashmap**

### **Thinking**

1.In the previous method

Step 1. The "nums" array is traversed to calculate all the elements of the sum array

Step 2. Use the nested loop to judge.

key : Can we judge when the array is traversed(Step 1) ?

Transposition

int[] sum = new int[nums.length + 1];

sum[0] = 0;

for (int end = 1; end < (nums.length + 1); end++)

sum[end] = sum[end - 1] + nums[end - 1];

a. Put each element of "sum" array into hashmap according to this format : (*sumi*, *number of occurence*)

b. **When constructing the "sum" array, we take the currently constructed element as sum[end], then all the elements before "end" which have been calculated can be regarded as all sum[start] for this "end".**

Transform the judgment condition

Obviously, when sum[end] is calculated, all its possible sum[start] are already in the map.

sum[end] - sum[start] == k

sum[end] - k == sum[start]

c. When *sumend* is calculated, we only need to determine whether there is key == *sumend*- k in the hashmap and add the number of occurrence to the answer.

**Attention : In the previous method, we set the first element of sum to 0. Similarly, we put it in the hashmap, which is (0, 1).**

### **Code**

public class Solution {

public int subarraySum(int[] nums, int k) {

int count = 0, sum = 0;

HashMap < Integer, Integer > map = new HashMap < > ();

map.put(0, 1);

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

sum += nums[i];

if (map.containsKey(sum - k))

count += map.get(sum - k);

map.put(sum, map.getOrDefault(sum, 0) + 1);

}

return count;

}

}

### **Complexity Anaysis**

* Time complexity : *O(n)*.
* Space complexity : *O(n)*.

## **A harder problem**

<https://leetcode.com/problems/number-of-submatrices-that-sum-to-target/>

<https://leetcode.com/problems/number-of-submatrices-that-sum-to-target/discuss/803353/java-solution-with-detailed-explanation>