<https://www.lintcode.com/problem/911/>

Given an array *nums* and a target value *k*, find the maximum length of a subarray that sums to *k*. If there isn't one, return 0 instead.

**Note:** The sum of the entire *nums* array is guaranteed to fit within the 32-bit signed integer range.

**Example 1:**

Input: nums = [1, -1, 5, -2, 3], k = 3

Output: 4

Explanation: The subarray [1, -1, 5, -2] sums to 3 and is the longest.

**Example 2:**

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

Output: 2

Explanation: The subarray [-1, 2] sums to 1 and is the longest.

**Constraints:**

1 <= nums.length <= 2 \* 10^5

-10^4 <= nums[i] <= 10^4

-10^9 <= k <= 10^9

**Follow Up:**

Can you do it in O(*n*) time?

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

**Solution 1: Hash Table (360 min, refer to** [L523.Continuous Subarray Sum (Ref.L974)](note://WEBc2ea24cf3f984fc0adf073f4552f8db8)**)**

**Style 1: int[] preSum = new int[nums.length + 1], the disadvantage is we also have to specially handle the nums.length == 1 && nums[0] == k case**

public class Solution {

    /\*\*

    \* @param nums: an array

    \* @param k: a target value

    \* @return: the maximum length of a subarray that sums to k

    \*/

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

        // Test out by nums={-1}, k=-1

        if(nums.length == 1 && nums[0] == k) {

            return 1;

        }

        // {key, value} = {pre-sum, index}

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

        map.put(0, 0);

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

        int result = 0;

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

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

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

                result = Math.max(result, i - map.get(preSum[i] - k));

            }

            // Only store preSum[i] when it happened first time, since we need to find

            // maximum length between, and it supposed to get between first happening

            // index to current index, all other indexes will only have less length,

// we can also use map.putIfAbsent(preSum[i], i) to guarantee only record

// the first time happening index, refer to L523.Continuous Subarray Sum

            if(!map.containsKey(preSum[i])) {

                map.put(preSum[i], i);

            }

        }

        return result;

    }

}

==========================================================================================

OR we can use map.put(k, 0) rather than map.put(0, 0) for initial position

public class Solution {

    /\*\*

    \* @param nums: an array

    \* @param k: a target value

    \* @return: the maximum length of a subarray that sums to k

    \*/

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

        // {key, value} = {pre-sum, index}

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

        // Differ than map.put(0, 0)

        map.put(k, 0);

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

        int result = 0;

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

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

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

                result = Math.max(result, i - map.get(preSum[i]));

            }

            // Only store preSum[i] + k when it happened first time, since we need to find

            // maximum length between, and it supposed to get between first happening

            // index to current index, all other indexes will only have less length,

// we can also use map.putIfAbsent(preSum[i], i) to guarantee only record

// the first time happening index, refer to L523.Continuous Subarray Sum

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

                map.put(preSum[i] + k, i);

            }

        }

        return result;

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**The problem on i < nums.length OR i <= nums.length in for loop**

e.g nums={1, -1, 5, -2, 3}, k=3;

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

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

    ...

}

preSum={0, 1, 0, 5, 3, 6}

nums[0]= stored on preSum[1], (nums[0] + nums[1]) stored on preSum[2], (nums[0] + nums[1] + nums[2]) stored on preSum[3], (nums[0] + nums[1] + nums[2] + nums[3]) stored on preSum[4], but missing (nums[0] + nums[1]... nums[4]) which suppose to be stored on preSum[5], only set "i <= nums.length" will make it happen, the root cause is we use "nums[i - 1]" which require "i == nums.length" to cover last element in nums

--------------------------------------------------------------------

if(!map.containsKey(preSum[i])) {

    map.put(preSum[i], i);

}

map will store below:

i=0 -> {0, 0} -> for preSum initial set as 0, since we didn't store any num at index=0, only start from index=1

i=1 -> {preSum[1]=0+nums[0]=1, 1}

i=2 -> {preSum[2]=preSum[1]+nums[1]=1-1=0, 2} -> ignore since 0 alraedy have {0, 0} as key=0 stored in map, keep index=0

i=3 -> {preSum[3]=preSum[2]+nums[2]=0+5=5, 3}

i=4 -> {preSum[4]=preSum[3]+nums[3]=5-2=3, 4}

i=5 -> {preSum[5]=preSum[4]+nums[4]=3+3=6, 5}

--------------------------------------------------------------------

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

    result = Math.max(result, i - map.get(preSum[i] - k));

}

length calculate below:

i=0 -> map not contains (0-3=-3)

i=1 -> map not contains (1-3=-2)

i=2 -> map not contains (0-3=-3)

i=3 -> map not contains (5-3=2)

i=4 -> map contains (3-3=0) -> i-map.get(preSum[i]-k)=4-map.get(0)=4-0=4 -> in preSum={0, 1, 0, 5, 3, 6} the initial preSum=0 get from index=0, the current preSum=3 get from index=4, length between calculate directly as 4-0=4, NOT including compensation +1 (e.g 4-0+1=5), because we introduce preSum[0]=0 and store nums beginning only from preSum[1], now comes back to raw input nums={1, -1, 5, -2, 3}, if we based on raw input, since it shift all elements index one step to left than preSum, maximum length to get between index=0 to index=3 needs to be calculated as 3-0+1=4, the compensation +1 is required then

i=5 -> map contains (6-3=3) -> i-map.get(preSum[i]-k)=5-map.get(3)=5-4=1

**Style 2: Set map.put(0, -1) with setting int[] preSum array on accumulate preSum[i + 1] = preSum[i] + nums[i] to avoid problem when i = 0, preSum[0] is nothing, nums[i] only accumulate on preSum[i + 1], e.g nums[0] only stored on preSum[1], nums[1] only stored on preSum[2]**

public class Solution {

    /\*\*

    \* @param nums: an array

    \* @param k: a target value

    \* @return: the maximum length of a subarray that sums to k

    \*/

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

        // {key, value} = {pre-sum, index}

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

        // Subarray sum from index i to index j (e.g array[i:j]) = sum(array[0:j]) - sum(array[0:i-1])

        // So this problem is another two-sum like problem, we can calculate prefix-sum of

        // the given array and use a hashmap to look up answer.

        map.put(0, -1);

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

        int result = 0;

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

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

            if(map.containsKey(preSum[i + 1] - k)) {

                result = Math.max(result, i - map.get(preSum[i + 1] - k));

            }

            // Only store preSum[i] when it happened first time, since we need to find

            // maximum length between, and it supposed to get between first happening

            // index to current index, all other indexes will only have less length,

// we can also use map.putIfAbsent(preSum[i], i) to guarantee only record

// the first time happening index, refer to L523.Continuous Subarray Sum

            if(!map.containsKey(preSum[i + 1])) {

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

            }

        }

        return result;

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Style 3: Set map.put(0, -1) with change int[] preSum array to int preSum only to avoid problem on accumulate preSum[i] = preSum[i - 1] + nums[i] issue when i = 0, we can NOT do preSum[i] = preSum[i] + nums[i], its wrong because of NO accumulate at all**

public class Solution {

    /\*\*

    \* @param nums: an array

    \* @param k: a target value

    \* @return: the maximum length of a subarray that sums to k

    \*/

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

        // {key, value} = {pre-sum, index}

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

        // Subarray sum from index i to index j (e.g array[i:j]) = sum(array[0:j]) - sum(array[0:i-1])

        // So this problem is another two-sum like problem, we can calculate prefix-sum of

        // the given array and use a hashmap to look up answer.

        map.put(0, -1);

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

        int preSum = 0;

        int result = 0;

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

            preSum += nums[i];

            if(map.containsKey(preSum - k)) {

                result = Math.max(result, i - map.get(preSum - k));

            }

            // Only store preSum[i] when it happened first time, since we need to find

            // maximum length between, and it supposed to get between first happening

            // index to current index, all other indexes will only have less length,

// we can also use map.putIfAbsent(preSum[i], i) to guarantee only record

// the first time happening index, refer to L523.Continuous Subarray Sum

            if(!map.containsKey(preSum)) {

                map.put(preSum, i);

            }

        }

        return result;

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**Refer to**

<https://www.lintcode.com/problem/911/solution/59403>

解题思路

保存从头开始以i结尾的sum，以及i的位置，有相同的sum时，只保留最先出现的那个，key就是sum，value是对应的数组下标.

有了sum之后，令target=sum-k，如果target之前出现过，那从target出现的那个点到当前这个点的和就是k。

我们要找的就是target出现的坐标和当前的坐标之差。

**特殊的，没有元素时默认和为0，所以要put一个-1进去，比如恰好第一个数num=k，有个-1可以使我们正好得到答案。**

题解代码

public class Solution {

    /\*\*

    \* @param nums: an array

    \* @param k: a target value

    \* @return: the maximum length of a subarray that sums to k

    \*/

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

        // Write your code here

        // 保存从头开始以i结尾的sum，以及i的位置，有相同的sum时，只保留最先出现的那个

        // key就是sum，value是对应的数组下标，即i

        // 例如 [1, -1, 5, -2, 3]

        //  [0, 1, 0,  5,  3, 6]

        // 有了sum之后，令target=sum-k，如果target之前出现过，那从target出现的那个点到当前这个点的和就是k

        // 我们要找的就是target出现的坐标和当前的坐标之差

        // 特殊的，没有元素时默认和为0，所以要put一个-1进去，比如恰好第一个数num=k，有个-1可以使我们正好得到答案

        int ans = 0;

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

        map.put(0, -1);

        int sum = 0;

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

            sum = sum + nums[i];

            int target = sum-k;

            // 如果target之前出现过，则target出现的坐标和当前的坐标之差就是答案

            if(null!=map.get(target)){

                ans = Math.max(ans, (i-map.get(target)));

            }

            // sum有相同的时候只要最先出现的，只有以前没put过的才需要put

            if(null==map.get(sum)){

                map.put(sum, i);

            }

        }

        return ans;

    }

}

**Refer to chatGPT**

**Approach:**

Prefix Sum:

At each index, calculate the cumulative sum (prefix sum) of the array.

If prefixSum[j] - prefixSum[i] = k, then the subarray nums[i+1...j] sums to k.

HashMap:

Store the **first occurrence** of each prefix sum and its corresponding index.

If prefixSum - k exists in the map, it means the subarray between that stored index and the current index sums to k.

Result:

Keep track of the maximum subarray length when a valid subarray is found.

class Solution {

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

// Map to store the first occurrence of a prefix sum

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

prefixSumMap.put(0, -1); // To handle the case where the entire subarray sums to k

int prefixSum = 0; // Cumulative prefix sum

int maxLength = 0; // Result: maximum length of subarray

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

prefixSum += nums[i]; // Update prefix sum

// Check if there exists a subarray that sums to k

if (prefixSumMap.containsKey(prefixSum - k)) {

int length = i - prefixSumMap.get(prefixSum - k);

maxLength = Math.max(maxLength, length);

}

// Store the first occurrence of the prefix sum

// If the prefixSum already exists, we do NOT update it because we want the longest subarray

if (!prefixSumMap.containsKey(prefixSum)) {

prefixSumMap.put(prefixSum, i);

}

}

return maxLength;

}

}

**Explanation:**

Initialization:

prefixSumMap is a HashMap that stores the **first occurrence** of each prefix sum and its index.

Insert (0, -1) into the map to handle cases where the subarray starting from index 0 sums to k.

Iterate through the array:

Update the prefixSum at each index.

Check if prefixSum - k exists in the map:

If it exists, calculate the length of the subarray and update maxLength if the new length is greater.

If the current prefixSum does not exist in the map, store it with the current index.

Return **maxLength**:

This gives the maximum length of any subarray that sums to k.

**Example:**

**Input:**

nums = [1, -1, 5, -2, 3], k = 3

**Execution:**

Step 1: Initialize prefixSumMap = {0: -1}, prefixSum = 0, maxLength = 0

Step 2: Traverse the array:

Index 0: prefixSum = 1, store (1, 0)

Index 1: prefixSum = 0, store (0, 1)

Index 2: prefixSum = 5, check 5 - 3 = 2 (not in map), store (5, 2)

Index 3: prefixSum = 3, check 3 - 3 = 0 (found at -1), update maxLength = 4

Index 4: prefixSum = 6, check 6 - 3 = 3 (found at 3), update maxLength = 4

Step 3: Return 4.

**Output:**

4

**Edge Cases:**

No valid subarray:

Input: nums = [1, 2, 3], k = 7

Output: 0

Single Element:

Input: nums = [3], k = 3

Output: 1

Negative Numbers:

Input: nums = [1, -1, 1, -1], k = 0

Output: 4

**Refer to**

[L523.Continuous Subarray Sum (Ref.L974)](note://WEBc2ea24cf3f984fc0adf073f4552f8db8)

[L560.Subarray Sum Equals K](note://5CE29DE019904808986311AFAC74C85C)

[L713.Subarray Product Less Than K (Ref.L325,L560,L2302)](note://4B9EF6834F24414AB80E2515E98D02C4)

[L2302.Count Subarrays With Score Less Than K](note://WEB7e8bf2699e4e216a31dae44a8cebc592)