644. Maximum Average Subarray II

July 15, 2017 | 15K views

Average Rating: 4.65 (26 votes)

Given an array consisting of `n` integers, find the contiguous subarray whose **length is greater than or equal to** `k` that has the maximum average value. And you need to output the maximum average value.

Example 1:

```
Input: [1,12,-5,-6,50,3], k = 4
Output: 12.75
Explanation:
when length is 5, maximum average value is 10.8,
when length is 6, maximum average value is 9.16667.
Thus return 12.75.
```

Note:

1. 1 <= `k` <= `n` <= 10,000.
2. Elements of the given array will be in range [-10,000, 10,000].
3. The answer with the calculation error less than 10^{-5} will be accepted.

Solution

Approach #1 Iterative method [Time Limit Exceeded]

One of the simplest solutions is to consider the sum of every possible subarray with length greater than or equal to k and to determine the maximum average from out of those. But, instead of finding out this sum in a naive manner for every subarray with length greater than or equal to k separately, we can do as follows.

For every starting point, s, considered, we can iterate over the elements of $nums$ starting from $nums$, and keep a track of the sum found till the current index(i). Whenever the index reached is such that the number of elements lying between s and i is greater than or equal to k, we can check if the average of the elements between s and i is greater than the average found till now or not.

Java Copy

```java
public class Solution {
    public double findMaxAverage(int[] nums, int k) {
        double res = Integer.MIN_VALUE;
        for (int s = 0; s < nums.length - k + 1; s++) {
            long sum = 0;
            for (int i = s; i < nums.length; i++) {
                sum += nums[i];
                if (i - s + 1 >= k)
                    res = Math.max(res, sum * 1.0 / (i - s + 1));
            }
        }
        return res;
    }
}
```

Complexity Analysis

- Time complexity : $O(n^2)$. Two for loops iterating over the whole length of $nums$ with n elements.
- Space complexity : $O(1)$. Constant extra space is used.

Approach #2 Using Binary Search [Accepted]

Algorithm

To understand the idea behind this method, let's look at the following points.

Firstly, we know that the value of the average could lie between the range (min, max). Here, min and max refer to the minimum and the maximum values out of the given $nums$ array. This is because, the average can't be lesser than the minimum value and can't be larger than the maximum value.

But, in this case, we need to find the maximum average of a subarray with atleast k elements. The idea in this method is to try to approximate(guess) the solution and to try to find if this solution really exists.

If it exists, we can continue trying to approximate the solution even to a further precise value, but choosing a larger number as the next approximation. But, if the initial guess is wrong, and the initial maximum average value(guessed) isn't possible, we need to try with a smaller number as the next approximate.

Now, instead of doing the guesses randomly, we can make use of Binary Search. With min and max as the initial numbers to begin with, we can find out the mid of these two numbers given by $(min + max)/2$. Now, we need to find if a subarray with length greater than or equal to k is possible with an average sum greater than this mid value.

To determine if this is possible in a single scan, let's look at an observation. Suppose, there exist j elements, $a_1, a_2, a_3..., a_j$ in a subarray within $nums$ such that their average is greater than mid. In this case, we can say that

$(a_1 + a_2 + a_3... + a_j)/j \geq mid$ or

$(a_1 + a_2 + a_3... + a_j) \geq j * mid$ or

$(a_1 - mid) + (a_2 - mid) + (a_3 - mid)... + (a_j - mid) \geq 0$

Thus, we can see that if after subtracting the mid number from the elements of a subarray with more than $k - 1$ elements, within $nums$, if the sum of elements of this reduced subarray is greater than 0, we can achieve an average value greater than mid. Thus, in this case, we need to set the mid as the new minimum element and continue the process.

Otherwise, if this reduced sum is lesser than 0 for all subarrays with greater than or equal to k elements, we can't achieve mid as the average. Thus, we need to set mid as the new maximum element and continue the process.

In order to determine if such a subarray exists in a linear manner, we keep on adding $nums[i] - mid$ to the sum obtained till the i^{th} element while traversing over the $nums$ array. If on traversing the first k elements, the sum becomes greater than or equal to 0, we can directly determine that we can increase the average beyond mid. Otherwise, we continue making additions to sum for elements beyond the k^{th} element, making use of the following idea.

If we know the cumulative sum upto indices i and j, say sum_i and sum_j respectively, we can determine the sum of the subarray between these indices(including j) as $sum_j - sum_i$. In our case, we want this difference between the cumulative sums to be greater than or equal to 0 as discusssed above.

Further, for sum_i as the cumulative sum upto the current(i^{th}) index, all we need is $sum_j - sum_i \geq 0$ such that $j - i \geq k$.

To achive this, instead of checking with all possible values of sum_i, we can just consider the minimum cumulative sum upto the index $j - k$. This is because if the required condition can't be sastisfied with the minimum sum_i, it can never be satisfied with a larger value.

To fulfil this, we make use of a $prev$ variable which again stores the cumulative sums but, its current index(for cumulative sum) lies behind the current index for sum at an offset of k units. Thus, by finding the minimum out of $prev$ and the last minimum value, we can easily find out the required minimum sum value.

Every time after checking the possiblility with a new mid value, at the end, we need to settle at some value as the average. But, we can observe that eventually, we'll reach a point, where we'll keep moving near some same value with very small changes. In order to keep our precision in control, we limit this process to 10^-5 precision, by making use of $error$ and continuing the process till $error$ becomes lesser than 0.00001 .

Java Copy

```java
public class Solution {
    public double findMaxAverage(int[] nums, int k) {
        double max_val = Integer.MIN_VALUE;
        double min_val = Integer.MAX_VALUE;
        for (int n: nums) {
            max_val = Math.max(max_val, n);
            min_val = Math.min(min_val, n);
        }
        double prev_mid = max_val, error = Integer.MAX_VALUE;
        while (error > 0.00001) {
            double mid = (max_val + min_val) * 0.5;
            if (check(nums, mid, k))
                min_val = mid;
            else
                max_val = mid;
            error = Math.abs(prev_mid - mid);
            prev_mid = mid;
        }
        return min_val;
    }
    public boolean check(int[] nums, double mid, int k) {
        double sum = 0, prev = 0, min_sum = 0;
        for (int i = 0; i < k; i++)
            sum += nums[i] - mid;
        if (sum >= 0)
            return true;
        for (int i = k; i < nums.length; i++) {
            sum += nums[i] - mid;
```

Complexity Analysis

- Time complexity : $O(nlog(max\_val - min\_val))$. $check$ takes $O(n)$ time and it is executed $O(log(max\_val - min\_val))$ times.
- Space complexity : $O(1)$. Constant Space is used.

Analysis written by: @vinod23

Rate this article: