Previous Next

209. Minimum Size Subarray Sum

June 5, 2017 | 103.1K views

Average Rating: 4.02 (55 votes)

Given an array of **n** positive integers and a positive integer **s**, find the minimal length of a **contiguous** subarray of which the sum ≥ **s**. If there isn't one, return 0 instead.

Example:

```
Input: s = 7, nums = [2,3,1,2,4,3]
Output: 2
Explanation: the subarray [4,3]
 has the minimal length under the problem constraint.
```

Follow up:
If you have figured out the *O*(*n*) solution, try coding another solution of which the time complexity is *O*(*n* log *n*).

Solution

Approach #1 Brute force [Time Limit Exceeded]

Intuition

Do as directed in question. Find the sum for all the possible subarrays and update the ans as and when we get a better subarray that fulfill the requirements ($\text{sum} \geq \text{s}$).

Algorithm

- Initialize $\text{ans} = \text{INT\_MAX}$
- Iterate the array from left to right using i:
  - Iterate from the current element to the end of vector using j:
    - Find the sum of elements from index i to j
    - If sum is greater then s:
      - Update $\text{ans} = \min(\text{ans}, (j - i + 1))$
      - Start the next ith iteration, since, we got the smallest subarray with $\text{sum} \geq s$ starting from the current index.

C++ Copy

```cpp
int minSubArrayLen(int s, vector<int>& nums)
{
    int n = nums.size();
    int ans = INT_MAX;
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            int sum = 0;
            for (int k = i; k <= j; k++) {
                sum += nums[k];
            }
            if (sum >= s) {
                ans = min(ans, (j - i + 1));
                break; //Found the smallest subarray with sum>=s starting with index i, hence move to next index
            }
        }
    }
    return (ans != INT_MAX) ? ans : 0;
}
```

Complexity Analysis

- Time complexity: $O(n^3)$.
  - For each element of array, we find all the subarrays starting from that index which is $O(n^2)$.
  - Time complexity to find the sum of each subarray is $O(n)$.
  - Thus, the total time complexity : $O(n^2 * n) = O(n^3)$
- Space complexity: $O(1)$ extra space.

Approach #2 A better brute force [Accepted]

Intuition

In Approach #1, you may notice that the sum is calculated for every surarray in $O(n)$ time. But, we could easily find the sum in O(1) time by storing the cumulative sum from the beginning(Memoization). After we have stored the cumulative sum in sums, we could easily find the sum of any subarray from i to j.

Algorithm

- The algorithm is similar to Approach #1.
- The only difference is in the way of finding the sum of subarrays:
  - Create a vector sums of size of nums
  - Initialize $\text{sums}[0] = \text{nums}[0]$
  - Iterate over the sums vector:
    - Update $\text{sums}[i] = \text{sums}[i-1] + \text{nums}[i]$
  - Sum of subarray from i to j is calculated as: $\text{sum} = \text{sums}[j] - \text{sums}[i] + \text{nums}[i]$, , wherein $\text{sums}[j] - \text{sums}[i]$ is the sum from $(i+1)$th element to the jth element.

C++ Copy

```cpp
int minSubArrayLen(int s, vector<int>& nums)
{
    int n = nums.size();
    if (n == 0)
        return 0;
    int ans = INT_MAX;
    vector<int> sums(n);
    sums[0] = nums[0];
    for (int i = 1; i < n; i++)
        sums[i] = sums[i - 1] + nums[i];
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            int sum = sums[j] - sums[i] + nums[i];
            if (sum >= s) {
                ans = min(ans, (j - i + 1));
                break; //Found the smallest subarray with sum>=s starting with index i, hence move to next index
            }
        }
    }
    return (ans != INT_MAX) ? ans : 0;
}
```

Complexity analysis

- Time complexity: $O(n^2)$.
  - Time complexity to find all the subarrays is $O(n^2)$.
  - Sum of the subarrays is calculated in $O(1)$ time.
  - Thus, the total time complexity: $O(n^2 * 1) = O(n^2)$
- Space complexity: $O(n)$ extra space.
  - Additional $O(n)$ space for sums vector than in Approach #1.

Approach #3 Using Binary search [Accepted]

Intuition

We could further improve the Approach #2 using the binary search. Notice that we find the subarray with $\text{sum} >= \text{s}$ starting with an index i in $O(n)$ time. But, we could reduce the time to $O(\log(n))$ using binary search. Note that in Approach #2, we search for subarray starting with index i, until we find $\text{sum} = \text{sums}[j] - \text{sums}[i] + \text{nums}[i]$ that is greater than s. So, instead of iterating linearly to find the sum, we could use binary search to find the index that is not lower than $\text{s} - \text{sums}[\text{i}]$ in the sums, which can be done using lower_bound function in C++ STL or could be implemented manually.

Algorithm

- Create vector $sums$ of size $n+1$ with : $\text{sums}[0] = 0, \text{sums}[i] = \text{sums}[i-1] + \text{nums}[i-1]$
- Iterate from $i = 1$ to n:
  - Find the value to_find in sum required for minimum subarray starting from index i to have sum greater than s, that is: $\text{to\_find} = \text{s} + \text{sums}[i-1]$
  - Find the index in sums such that value at that index is not lower than the to_find value, say bound
  - If we find the to_find in sums, then:
    - Size of current subarray is given by: $\text{bound} - (\text{sums.begin}() + i - 1)$
    - Compare ans with the current subarray size and store minimum in ans

C++ Copy

```cpp
int minSubArrayLen(int s, vector<int>& nums)
{
    int n = nums.size();
    if (n == 0)
        return 0;
    int ans = INT_MAX;
    vector<int> sums(n + 1, 0); //size = n+1 for easier calculations
    //sums[0]=0 : Meaning that it is the sum of first 0 elements
    //sums[1]=A[0] : Sum of first 1 elements
    //ans so on...
    for (int i = 1; i <= n; i++)
        sums[i] = sums[i - 1] + nums[i - 1];
    for (int i = 1; i <= n; i++) {
        int to_find = s + sums[i - 1];
        auto bound = lower_bound(sums.begin(), sums.end(), to_find);
        if (bound != sums.end()) {
            ans = min(ans, static_cast<int>(bound - (sums.begin() + i - 1)));
        }
    }
    return (ans != INT_MAX) ? ans : 0;
}
```

Complexity analysis

- Time complexity: $O(n \log(n))$.
  - For each element in the vector, find the subarray starting from that index, and having sum greater than s using binary search. Hence, the time required is $O(n)$ for iteration over the vector and $O(\log(n))$ for finding the subarray for each index using binary search.
  - Therefore, total time complexity = $O(n * \log(n))$
- Space complexity: $O(n)$. Additional $O(n)$ space for sums vector

Approach #4 Using 2 pointers [Accepted]

Intuition

Until now, we have kept the starting index of subarray fixed, and found the last position. Instead, we could move the starting index of the current subarray as soon as we know that no better could be done with this index as the starting index. We could keep 2 pointer,one for the start and another for the end of the current subarray, and make optimal moves so as to keep the sum greater than s as well as maintain the lowest size possible.

Algorithm

- Initialize left pointer to 0 and sum to 0
- Iterate over the nums:
  - Add $\text{nums}[i]$ to sum
  - While sum is greater than or equal to s:
    - Update $\text{ans} = \min(\text{ans}, i + 1 - \text{left})$, where $i + 1 - \text{left}$ is the size of current subarray
    - It means that the first index can safely be incremented, since, the minimum subarray starting with this index with $\text{sum} \geq s$ has been achieved
    - Subtract $\text{nums}[\text{left}]$ from sum and increment left

C++ Copy

```cpp
int minSubArrayLen(int s, vector<int>& nums)
{
    int n = nums.size();
    int ans = INT_MAX;
    int left = 0;
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += nums[i];
        while (sum >= s) {
            ans = min(ans, i + 1 - left);
            sum -= nums[left++];
        }
    }
    return (ans != INT_MAX) ? ans : 0;
}
```

Complexity analysis

- Time complexity: $O(n)$. Single iteration of $O(n)$.
  - Each element can be visited atmost twice, once by the right pointer(i) and (atmost)once by the left pointer.
- Space complexity: $O(1)$ extra space. Only constant space required for left, sum, ans and i.

Rate this article: ★★★★★

Previous Next

Comments: 47 Sort By

tiddlyz ★ 292 August 26, 2018 3:27 PM

1,2 are terrible solutions.
Basically brute force solution is made unnecessarily complicate. The simple idea is just to calculate for each i when the sum can be larger or equal to s. No need 3 for loops, 2 loop is enough.

```
        for(int i=0; i<n; n++){
```

Read More

64 Share Reply

SHOW 6 REPLIES

v1s1on ★ 513 February 7, 2019 4:13 AM

I'm also surprised that the most obvious brute-force solution isn't there. The good old double for loop, O(n^2) time, O(1) space that actually gets accepted:

```
int minSubArrayLen(int s, vector<int>& nums) {
        int minLen = numeric_limits<int>::max();
```

Read More

32 Share Reply

SHOW 1 REPLY

Wrecked ★ 55 January 28, 2019 4:36 AM

solution 4 is buggy.

correct way to use two pointers is

public int MinSubArrayLen(int s, int[] nums) {

Read More

13 Share Reply

SHOW 8 REPLIES

bykov ★ 16 August 10, 2017 2:35 AM

input
11
[1,2,3,4,5]
expected 3

Read More

9 Share Reply

SHOW 2 REPLIES

kekko1285 ★ 8 April 8, 2018 8:12 PM

@vats.rahul if you sort the array, you are altering the position of the element of the input, hence you cannot provide a correct answer. You have to find all the possible contiguous elements in the input array, not the minimum amount of elements in the array so that this sum is >= of the required sum.

5 Share Reply

SHOW 2 REPLIES

dharpatel ★ 13 March 2, 2018 12:07 PM

input
11
[1,2,3,4,5]
expected 3
How this is possible. I guess for this out put it should return 0 because

Read More

6 Share Reply

SHOW 2 REPLIES

javak ★ 3 November 6, 2017 3:02 PM

```
int sum = 0;
        int n = nums.length;
        int res = Integer.MAX_VALUE;
        int l = 0;
```

Read More

3 Share Reply

SHOW 1 REPLY

s961206 ★ 751 July 19, 2019 5:42 AM

Still don't understand why solution 4 works? Could someone please provide formal explanation?

1 Share Reply

SHOW 3 REPLIES

fuyaoli ★ 95 July 21, 2018 10:14 PM

I think there might be problem in the second brute force approach. If there are 2147483647 "1"s in the input array, the ans will be INT_MAX at last and it should return INT_MAX since it is a available answer. However, in the given solution, we will return 0. It's wrong.

1 Share Reply

adheeth_s ★ 17 April 9, 2019 5:32 AM

The question by itself is wrong. What is the solution for
[2,7,10]
12

0 or 2?

Read More

1 Share Reply

SHOW 2 REPLIES

‹ 1 2 3 4 5 ›