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 \text{ans}ans as and when we get a better subarray that fulfill the requirements (\text{sum} \geq \text{s}sum≥s).

**Algorithm**

* Initialize ans=INT\_MAX
* Iterate the array from left to right using i*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))ans=min(ans,(*j*−*i*+1))
      * Start the next i*i*th iteration, since, we got the smallest subarray with \text{sum} \geq ssum≥*s* starting from the current index.

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)*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 \text{sums}sums, we could easily find the sum of any subarray from i*i* to j*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 \text{sums}sums of size of \text{nums}nums
  + Initialize \text{sums}[0]=\text{nums}[0]sums[0]=nums[0]
  + Iterate over the \text{sums}sums vector:
    - Update \text{sums}[i] = \text{sums}[i-1] + \text{nums}[i]sums[*i*]=sums[*i*−1]+nums[*i*]
  + Sum of subarray from i*i* to j*j* is calculated as: \text{sum}=\text{sums}[j] - \text{sums}[i] +\text{nums}[i]sum=sums[*j*]−sums[*i*]+nums[*i*], , wherein \text{sums}[j] - \text{sums}[i]sums[*j*]−sums[*i*] is the sum from (i+1*i*+1)th element to the j*j*th element.

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)*O*(*n*2).
  + Time complexity to find all the subarrays is O(n^2)*O*(*n*2).
  + Sum of the subarrays is calculated in O(1)*O*(1) time.
  + Thus, the total time complexity: O(n^2 \* 1) = O(n^2)*O*(*n*2∗1)=*O*(*n*2)
* Space complexity: O(n)*O*(*n*) extra space.
  + Additional O(n)*O*(*n*) space for \text{sums}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}sum>=s starting with an index i*i* in O(n)*O*(*n*) time. But, we could reduce the time to O(\log(n))*O*(log(*n*)) using binary search. Note that in Approach #2, we search for subarray starting with index i*i*, until we find \text{sum}=\text{sums}[j] - \text{sums}[i] +\text{nums}[i]sum=sums[*j*]−sums[*i*]+nums[*i*] that is greater than \text{s}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[i]}s−sums[i] in the \text{sums}sums, which can be done using \text{lower\\_bound}lower\_bound function in C++ STL or could be implemented manually.

**Algorithm**

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

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))*O*(*n*log(*n*)).
  + For each element in the vector, find the subarray starting from that index, and having sum greater than s*s* using binary search. Hence, the time required is O(n)*O*(*n*) for iteration over the vector and O(\log(n))*O*(log(*n*)) for finding the subarray for each index using binary search.
  + Therefore, total time complexity = O(n\*\log(n))*O*(*n*∗log(*n*))
* Space complexity: O(n)*O*(*n*). Additional O(n)*O*(*n*) space for \text{sums}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 \text{sum}sum greater than s*s* as well as maintain the lowest size possible.

**Algorithm**

* Initialize \text{left}left pointer to 0 and \text{sum}sum to 0
* Iterate over the \text{nums}nums:
  + Add \text{nums}[i]nums[*i*] to \text{sum}sum
  + While \text{sum}sum is greater than or equal to s*s*:
    - Update \text{ans}=\min(\text{ans},i+1-\text{left})ans=min(ans,*i*+1−left), where i+1-\text{left}*i*+1−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 ssum≥*s* has been achieved
    - Subtract \text{nums[left]}nums[left] from \text{sum}sum and increment \text{left}left

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