862. Shortest Subarray with Sum at Least K

Problem Description

and it won't be needed for future calculations.

The problem asks us to find the length of the shortest contiguous subarray within an integer array nums such that the sum of its elements is at least a given integer k. If such a subarray does not exist, we are to return -1. The attention is on finding the minimum-length subarray that meets or exceeds the sum condition.

Binary Search Prefix Sum Sliding Window Monotonic Queue Heap (Priority Queue)

Hard

Intuition To solve this efficiently, we utilize a monotonic <u>queue</u> and prefix sums technique. The intuition behind using prefix sums is that

they allow us to quickly calculate the sum of any subarray in constant time. This makes the task of finding subarrays with a sum

A prefix sum array s is an array that holds the sum of all elements up to the current index. So for any index i, s[i] is the sum of nums[0] + nums[1] + ... + nums[i-1].

To understand the monotonic <u>queue</u>, which is a Double-Ended Queue (deque) in this case, let's look at its properties:

1. It is used to maintain a list of indices of prefix sums in increasing order. 2. When we add a new prefix sum, we remove all the larger sums at the end of the queue because the new sum and any future sums would always

- be better choices (smaller subarray) for finding a subarray of sum k. 3. We also continuously check if the current prefix sum minus the prefix sum at the start of the queue is at least k. If it is, we found a candidate
- subarray, and update ans with the subarray's length. Then we can pop that element off the queue since we've already considered this subarray

of at least k much faster, as opposed to calculating the sum over and over again for different subarrays.

In summary, the prefix sums help us quickly compute subarray sums, and the monotonic queue lets us store and traverse candidate subarray ranges efficiently, ensuring we always have the smallest length subarray that meets the sum condition, thus arriving at the optimal solution.

Solution Approach The solution makes use of prefix sums and a monotonic queue, specifically a deque, to achieve an efficient algorithm to find the shortest subarray summing to at least k. Let's explore the steps involved:

Prefix Sum Calculation: We initiate the computation by creating a list called s that contains the cumulative sum of the nums

list, by using the accumulate function with an initial parameter set to 0. This denotes that the first element of s is 0 and

Deque Front Comparison: While there are elements in q and the current <u>prefix sum</u> v minus the prefix sum at q[0] (the front

is a requirement to consider subarrays starting at index 0. **Initialization**: A deque q is initialized to maintain the monotonic <u>queue</u> of indices. A variable <u>ans</u> is initialized to <u>inf</u> (infinity)

which will later store the smallest length of a valid subarray. Iterate Over Prefix Sums: We iterate over each value v in the prefix sums array s and its index i.

- of the deque) is greater than or equal to k, we've found a subarray that meets the requirement. We then compute its length i - q.popleft() and update ans with the minimum of ans and this length. The popleft() operation removes this index from the deque as it is no longer needed.
- **Deque Back Optimization**: Before appending the current index i to q, we pop indices from the back of the deque if their corresponding prefix sums are greater than or equal to v because these are not conducive to the smallest length requirement and will not be optimal candidates for future comparisons.

Index Appending: Append the current index i to q. This index represents the right boundary for the potential subarray sums

After the loop, two cases may arise: • If ans remains inf, it means no valid subarray summing to at least k was found, so we return -1.

• Otherwise, we return the value stored in ans as it represents the length of the shortest subarray that fulfills the condition.

those that can potentially give a smaller length subarray are considered. The key here is to understand how the prefix sum helps us quickly calculate the sum of a subarray and how the monotonically increasing property of the queue ensures we always get the shortest length possible. The use of these data structures makes the solution capable of working in O(n) time complexity.

We want to find the length of the smallest contiguous subarray with a sum greater than or equal to 7. Here's how we would apply

Overall, the algorithm smartly maintains a set of candidate indices for the start of the subarray in a deque, ensuring that only

nums = [2, 1, 5, 2, 3, 2]k = 7

And initialize ans to inf:

append 3 into q.

from collections import deque

from math import inf

class Solution:

from itertools import accumulate

min_length = inf

indices deque = deque()

indices deque.pop()

indices_deque.append(current_index)

return -1 if min_length == inf else min_length

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

int shortestSubarray(vector<int>& nums, int k) {

// Loop through all prefix sum entries

indices.pop_front();

indices.pop_back();

indices.push_back(i);

// Add current index to the deque

return minLength > n ? -1 : minLength;

// Update the minimum length

for (int i = 0; $i \le n$; ++i) {

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

// Double ended queue to store indices of the prefix sums

// Initialize the answer with maximum possible length + 1

// If the current subarray (from front of deque to i) has sum $\geq k$

minLength = min(minLength, i - indices.front());

while (!indices.empty() && prefixSum[i] - prefixSum[indices.front()] >= k) {

// Pop the front index since we found a shorter subarray ending at index i

// we can discard it. since better candidates for subarray start are available

while (!indices.empty() && prefixSum[indices.back()] >= prefixSum[i]) {

// If no valid subarrav is found, minLength remains > n. Return -1 in that case.

// While the last index in the deque has a prefix sum larger than or equal to current

import { Deque } from 'collections/deque'; // Assume a Deque implementation like "collections.js" or similar

vector<long> prefixSum(n + 1, 0);

// Calculate the prefix sums

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

// Prefix sum array with an extra slot for ease of calculations

int n = nums.size();

deque<int> indices;

int minLength = n + 1;

def shortest subarrav(self, nums: List[int], k: int) -> int:

Initialize a double-ended queue to store indices

prefix sums = list(accumulate(nums, initial=0))

Calculate the prefix sums of nums with an initial value of 0

Set the initial answer to infinity, as we are looking for the minimum

than or equal to current sum, as they are not useful anymore

Add the current index to the back of the deque

min length = min(min length, current index - indices deque.popleft())

Remove indices from the back of the deque if their prefix sums are greater

Return -1 if no such subarray exists, otherwise the length of the shortest subarray

while indices deque and prefix_sums[indices_deque[-1]] >= current_sum:

Index Appending: Our q is now [3].

anything from the queue yet.

ans = inf

Example Walkthrough

the solution approach:

evaluated in future iterations.

s = [0, 2, 3, 8, 10, 13, 15]

Notice that s[0] is 0 because we've added it artificially to account for subarrays starting at index 0.

Iterate Over Prefix Sums: We iterate over s, looking for subarrays that sum up to at least k.

q.popleft() = 6 - 3 = 3. Now the ans becomes 3, the smallest subarray [5, 2, 3] found so far.

Let's illustrate the solution approach with an example. Suppose we have the following array and target sum k:

- **Initialization**: We create a deque q to maintain indices of prefix sums: q = []
 - **Deque Front Comparison**: As we proceed, when we reach s[3] = 8 (consider nums[0..2]), we find it is greater than or equal to k. The q is empty, so we just move on.

Deque Back Optimization: Before appending index 3 to q, we don't remove anything from q because it's still empty. So we

We have a non-empty q, and s[5] - s[q[0]] = 13 - 8 = 5, which is not greater than or equal to k. So we can't pop

Once we reach s[6] = 15, we notice that 15 - 8 = 7 is exactly our k. We then calculate the subarray length 6 -

Deque Back Optimization is also done each time before we append a new index, which keeps the indices in the deque

Continuing the iteration, we eventually come to s[5] = 13, which is the sum up to nums[0..4].

Prefix Sum Calculation: We compute the prefix sum array s:

monotonically increasing in sums. After considering all elements in s, our ans is 3, as no smaller subarray summing to at least 7 is found. So we return 3 as the

length of the shortest subarray. If ans was still infinity, we would return -1 indicating no such subarray was found.

- Solution Implementation **Python**
- # Enumerate over the prefix sums to find the shortest subarray for current index, current sum in enumerate(prefix sums): # If the current sum minus the sum at the front of the deque is at least k, # update the min length and pop from the deque while indices deque and current sum - prefix sums[indices deque[0]] >= k:

// Function to find the length of the shortest subarray with a sum at least 'k' public int shortestSubarray(int[] nums, int k) { int n = nums.length; // Get the length of the input array long[] prefixSums = new long[n + 1]; // Create an array to store prefix sums

// Calculate prefix sums

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

Java

class Solution {

```
// Initialize a deque to keep track of indices
        Deque<Integer> indexDeque = new ArrayDeque<>();
        int minLength = n + 1; // Initialize the minimum length to an impossible value (larger than the array itself)
        // Iterate over the prefix sums
        for (int i = 0; i <= n; ++i) {
            // While the degue is not empty, check if the current sum minus the front value is \geq k
            while (!indexDeque.isEmpty() && prefixSums[i] - prefixSums[indexDeque.peek()] >= k) {
                minLength = Math.min(minLength, i - indexDeque.poll()); // If true, update minLength
            // While the deque is not empty, remove all indices from the back that have a prefix sum greater than or equal to the cur
            while (!indexDeque.isEmpty() && prefixSums[indexDeque.peekLast()] >= prefixSums[i]) {
                indexDeque.pollLast();
            // Add the current index to the deque
            indexDeque.offer(i);
        // If minLength is still greater than the length of the array, there is no valid subarray, return -1
        return minLength > n ? -1 : minLength;
C++
#include <vector>
#include <deque>
#include <algorithm> // For std::min
class Solution {
public:
    // Function to find the length of shortest subarray with sum at least K
```

};

TypeScript

// Importing required modules

```
// Function to find the length of shortest subarray with sum at least K
function shortestSubarray(nums: number[], k: number): number {
    let n = nums.length;
   // Prefix sum array with an extra slot for ease of calculations
    let prefixSum: number[] = new Array(n + 1).fill(0);
   // Calculate the prefix sums
    for (let i = 0; i < n; ++i) {
       prefixSum[i + 1] = prefixSum[i] + nums[i];
   // Double-ended queue to store indices of the prefix sums
   let indices: Deque<number> = new Deque<number>();
   // Initialize the answer with maximum possible length + 1
    let minLength = n + 1;
   // Loop through all prefix sum entries
    for (let i = 0; i <= n; ++i) {
       // If the current subarray (from front of deque to i) has a sum >= k
       while (!indices.isEmpty() && prefixSum[i] - prefixSum[indices.peekFront()!] >= k) {
            // Update the minimum length
           minLength = Math.min(minLength, i - indices.peekFront()!);
           // Pop the front index since we found a shorter subarray ending at index i
            indices.shift();
       // While the last index in the deque has a prefix sum larger than or equal to the current
       // we can discard it, since better candidates for subarray start are available
       while (!indices.isEmpty() && prefixSum[indices.peekBack()!] >= prefixSum[i]) {
            indices.pop();
       // Add current index to the deque
        indices.push(i);
   // If no valid subarray is found, minLength remains > n. Return -1 in that case.
   return minLength > n ? -1 : minLength;
// Example usage:
// const nums = [2, -1, 2];
// const k = 3:
// console.log(shortestSubarray(nums, k)); // Output should be 3
from collections import deque
from itertools import accumulate
from math import inf
class Solution:
   def shortest subarray(self, nums: List[int], k: int) -> int:
       # Calculate the prefix sums of nums with an initial value of 0
       prefix sums = list(accumulate(nums, initial=0))
       # Initialize a double-ended queue to store indices
       indices deque = deque()
       # Set the initial answer to infinity, as we are looking for the minimum
       min_length = inf
```

concept of prefix sums and a monotonic queue to keep track of potential candidates for the shortest subarray.

Time and Space Complexity

indices deque.pop()

indices_deque.append(current_index)

return -1 if min_length == inf else min_length

Enumerate over the prefix sums to find the shortest subarray

If the current sum minus the sum at the front of the deque is at least k,

min length = min(min length, current index - indices degue.popleft())

Remove indices from the back of the deque if their prefix sums are greater

Return -1 if no such subarray exists, otherwise the length of the shortest subarray

while indices deque and current sum - prefix sums[indices deque[0]] >= k:

while indices deque and prefix_sums[indices_deque[-1]] >= current_sum:

than or equal to current sum, as they are not useful anymore

for current index, current sum in enumerate(prefix sums):

update the min length and pop from the deque

Add the current index to the back of the deque

Time Complexity The time complexity of the code is O(N), where N is the length of the input array nums. This is because:

• The deque q is maintained by iterating through each element of the array once. Each element is added and removed from the deque at most

• The prefix sum array s is computed using itertools.accumulate, which is a single pass through the array, thus taking O(N) time.

once. Since the operations of adding to and popping from a deque are 0(1), the loop operations are also 0(N) in total.

The given Python code is for finding the length of the shortest contiguous subarray whose sum is at least k. The code uses the

• Inside the loop, q.popleft() and q.pop() are each called at most once per iteration, and as a result, each element in nums contributes at most 0(1) to the time complexity.

requiring O(N) space.

Space Complexity

- The space complexity of the code is O(N) as well: The prefix sum array s requires O(N) space.
- The deque q potentially stores indices from the entire array in the worst-case scenario. In the worst case, it could hold all indices in nums, also
- Apart from these two data structures, the code uses a constant amount of space for variables such as ans and v, which does not depend on the input size.