Day2

December 17, 2023 1:32 PM

209. Minimum Size Subarray Sum

Medium
12K
374
Companies

Given an array of positive integers nums and a positive integer target, return *the* ***minimal length*** *of a subarray whose sum is greater than or equal to* target. If there is no such subarray, return 0 instead.

Example 1:
Input: target = 7, nums = [2,3,1,2,4,3]
Output: 2
Explanation: The subarray [4,3] has the minimal length under the problem constraint.
Example 2:
Input: target = 4, nums = [1,4,4]
Output: 1
Example 3:
Input: target = 11, nums = [1,1,1,1,1,1,1,1]
Output: 0

Constraints:

- 1 <= target <= 10^9
- 1 <= nums.length <= 10^5
- 1 <= nums[i] <= 10^4

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)).

From <https://leetcode.com/problems/minimum-size-subarray-sum/>

Thought:

1. Use two for loops to go through the array and find the sub array that meet the conditions.
2. Use something like sliding window.
   In one for loop, we keep redefining the starting position once the sub array is satisfied the condition.

pseudo code:

```
Result = nums.length i=0
For (j = 0; j<nums.length; j++):
    Sum += nums[i]
    While(sum >= target):
        subL = j-i+1
        Result = min(result,subL)
        Sum = sum - nums[i]
        i++
Return result
```

35. Search Insert Position

Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.
You must write an algorithm with O(log n) runtime complexity.

Example 1:
Input: nums = [1,3,5,6], target = 5
Output: 2
Example 2:
Input: nums = [1,3,5,6], target = 2
Output: 1
Example 3:
Input: nums = [1,3,5,6], target = 7
Output: 4

Thought:
Binary search

Pseudo code:

```
Left = 0
Right = nums.length
Result = 0
While(left <= right):
    Middle = (left + right) /2
    Result = middle
    If(nums[middle] < target):
        Right = middle-1
    Else if( nums[middle] > target):
        Left = middle +1
    Else:
        Return middle
Return result
```

69. Sqrt(x)

Given a non-negative integer x, return *the square root of* x *rounded down to the nearest integer*. The returned integer should be **non-negative** as well.
You **must not use** any built-in exponent function or operator.

- For example, do not use pow(x, 0.5) in c++ or x ** 0.5 in python.

Example 1:
Input: x = 4
Output: 2
Explanation: The square root of 4 is 2, so we return 2.
Example 2:
Input: x = 8
Output: 2
Explanation: The square root of 8 is 2.82842..., and since we round it down to the nearest integer, 2 is returned.

Constraints:

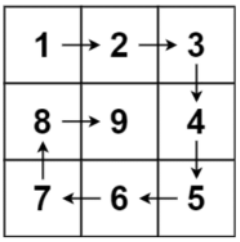
- 0 <= x <= 2^{31} - 1

From <https://leetcode.com/problems/sqrtx/description/>

Given a positive integer n, generate an n x n matrix filled with elements from 1 to n^2 in spiral order.

From <https://leetcode.com/problems/spiral-matrix-ii/>

Example 1:



1	2	3	4
12	13	14	5
11	16	15	6
10	9	8	7

```
Input: n = 3
Output: [[1,2,3],[8,9,4],[7,6,5]]
```

Example 2:

```
Input: n = 1
Output: [[1]]
```

34. Find First and Last Position of Element in Sorted Array

Given an array of integers nums sorted in non-decreasing order, find the starting and ending position of a given target value.
If target is not found in the array, return [-1, -1].
You must write an algorithm with O(log n) runtime complexity.

Example 1:
Input: nums = [5,7,7,8,8,10], target = 8
Output: [3,4]
Example 2:
Input: nums = [5,7,7,8,8,10], target = 6
Output: [-1,-1]
Example 3:
Input: nums = [], target = 0
Output: [-1,-1]

Constraints:

- 0 <= nums.length <= 10^5
- -10^9 <= nums[i] <= 10^9
- nums is a non-decreasing array.
- -10^9 <= target <= 10^9

From <https://leetcode.com/problems/find-first-and-last-position-of-element-in-sorted-array/>

Note: Using left + (right - left) / 2 instead of (left + right) / 2 to calculate the middle index in a binary search algorithm is a best practice to avoid integer overflow. Let me explain why this is important:

1. Integer Overflow Issue: When you are working with very large integers, the sum left + right could exceed the maximum value that can be stored in an integer variable (in many programming languages, this is 2^31−1for a 32-bit signed integer). If this happens, it results in integer overflow, meaning the value wraps around and becomes negative, which could cause your program to behave unexpectedly or even crash.
2. Safe Calculation: By using left + (right - left) / 2, you avoid this potential overflow. This is because (right - left) will never be larger than the maximum value of an integer (as long as left and right are within valid integer bounds), and left is added after the division, so the total will not exceed the integer limit.
   For example, consider a case where left and right are very large positive integers. Using (left + right) / 2 could result in a sum that overflows. However, left + (right - left) / 2 will not overflow since the subtraction is performed first, yielding a smaller intermediate result that is safe to divide and add.
   In summary, left + (right - left) / 2 is a more robust way to calculate the middle index, especially in scenarios where you are dealing with large indices or when working in languages with fixed-size integer types.