Minimum Size Subarray Sum

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

For example, given the array `[2,3,1,2,4,3]` and `s = 7`,
the subarray `[4,3]` has the minimal length under the problem constraint.

click to show more practice.

More practice:
If you have figured out the $O(n)$ solution, try coding another solution of which the time complexity is $O(n \log n)$.

Credits:
Special thanks to @Freezen for adding this problem and creating all test cases.

Solution 1

```
public class Solution {
    public int minSubArrayLen(int s, int[] nums) {
        return solveNLogN(s, nums);
    }

    private int solveN(int s, int[] nums) {
        int start = 0, end = 0, sum = 0, minLen = Integer.MAX_VALUE;
        while (end < nums.length) {
            while (end < nums.length && sum < s) sum += nums[end++];
            if (sum < s) break;
            while (start < end && sum >= s) sum -= nums[start++];
            if (end - start + 1 < minLen) minLen = end - start + 1;
        }
        return minLen == Integer.MAX_VALUE ? 0 : minLen;
    }

    private int solveNLogN(int s, int[] nums) {
        int[] sums = new int[nums.length + 1];
        for (int i = 1; i < sums.length; i++) sums[i] = sums[i - 1] + nums[i - 1]
;
        int minLen = Integer.MAX_VALUE;
        for (int i = 0; i < sums.length; i++) {
            int end = binarySearch(i + 1, sums.length - 1, sums[i] + s, sums);
            if (end == sums.length) break;
            if (end - i < minLen) minLen = end - i;
        }
        return minLen == Integer.MAX_VALUE ? 0 : minLen;
    }

    private int binarySearch(int lo, int hi, int key, int[] sums) {
        while (lo <= hi) {
           int mid = (lo + hi) / 2;
           if (sums[mid] >= key){
               hi = mid - 1;
           } else {
               lo = mid + 1;
           }
        }
        return lo;
    }
}
```

Since the given array contains only positive integers, the subarray sum can only increase by including more elements. Therefore, you don't have to include more elements once the current subarray already has a sum large enough. This gives the linear time complexity solution by maintaining a minimum window with a two indices.

As to NLogN solution, logN immediately reminds you of binary search. In this case, you cannot sort as the current order actually matters. How does one get an ordered array then? Since all elements are positive, the cumulative sum must be strictly increasing. Then, a subarray sum can expressed as the difference between two

cumulative sum. Hence, given a start index for the cumulative sum array, the other end index can be searched using binary search.

written by lx223 original link here

Solution 2

The problem statement has stated that there are both `O(n)` and `O(nlogn)` solutions to this problem. Let's see the `O(n)` solution first (taken from this link), which is pretty clever and short.

```
class Solution {
public:
    int minSubArrayLen(int s, vector<int>& nums) {
        int n = nums.size(), start = 0, sum = 0, minlen = INT_MAX;
        for (int i = 0; i < n; i++) {
            sum += nums[i];
            while (sum >= s) {
                minlen = min(minlen, i - start + 1);
                sum -= nums[start++];
            }
        }
        return minlen == INT_MAX ? 0 : minlen;
    }
};
```

Well, you may wonder how can it be `O(n)` since it contains an inner `while` loop. Well, the key is that the `while` loop executes at most once for each starting position `start`. Then `start` is increased by `1` and the `while` loop moves to the next element. Thus the inner `while` loop runs at most `O(n)` times during the whole `for` loop from `0` to `n - 1`. Thus both the `for` loop and `while` loop has `O(n)` time complexity in total and the overall running time is `O(n)`.

There is another `O(n)` solution in this link, which is easier to understand and prove it is `O(n)`. I have rewritten it below.

```
class Solution {
public:
    int minSubArrayLen(int s, vector<int>& nums) {
        int n = nums.size(), left = 0, right = 0, sum = 0, minlen = INT_MAX;
        while (right < n) {
            do sum += nums[right++];
            while (right < n && sum < s);
            while (left < right && sum - nums[left] >= s)
                sum -= nums[left++];
            if (sum >= s) minlen = min(minlen, right - left);
        }
        return minlen == INT_MAX ? 0 : minlen;
    }
};
```

Now let's move on to the `O(nlogn)` solution. Well, this less efficient solution is far more difficult to come up with. The idea is to first maintain an array of accumulated summations of elements in `nums`. Specifically, for `nums = [2, 3, 1, 2, 4, 3]` in the problem statement, `sums = [0, 2, 5, 6, 8, 12, 15]`. Then for each element in `sums`, if it is not less than `s`, we search for the first element that is

greater than `sums[i] - s` (in fact, this is just what the `upper_bound` function does) in `sums` using binary search.

Let's do an example. Suppose we reach `12` in `sums`, which is greater than `s = 7`. We then search for the first element in `sums` that is greater than `sums[i] - s = 12 - 7 = 5` and we find `6`. Then we know that the elements in `nums` that correspond to `6, 8, 12` sum to a number `12 - 5 = 7` which is not less than `s = 7`. Let's check for that: `6` in `sums` corresponds to `1` in `nums`, `8` in `sums` corresponds to `2` in `nums`, `12` in `sums` corresponds to `4` in `nums`. `1, 2, 4` sum to `7`, which is `12` in `sums` minus `5` in `sums`.

We add a `0` in the first position of `sums` to account for cases like `nums = [3], s = 3`.

The code is as follows.

```
class Solution {
public:
    int minSubArrayLen(int s, vector<int>& nums) {
        vector<int> sums = accumulate(nums);
        int n = nums.size(), minlen = INT_MAX;
        for (int i = 1; i <= n; i++) {
            if (sums[i] >= s) {
                int p = upper_bound(sums, 0, i, sums[i] - s);
                if (p != -1) minlen = min(minlen, i - p + 1);
            }
        }
        return minlen == INT_MAX ? 0 : minlen;
    }
private:
    vector<int> accumulate(vector<int>& nums) {
        int n = nums.size();
        vector<int> sums(n + 1, 0);
        for (int i = 1; i <= n; i++)
            sums[i] = nums[i - 1] + sums[i - 1];
        return sums;
    }
    int upper_bound(vector<int>& sums, int left, int right, int target) {
        int l = left, r = right;
        while (l < r) {
            int m = l + ((r - l) >> 1);
            if (sums[m] <= target) l = m + 1;
            else r = m;
        }
        return sums[r] > target ? r : -1;
    }
};
```

written by jianchao.li.fighter original link here

Solution 3

```
public int minSubArrayLen(int s, int[] a) {
  if (a == null || a.length == 0)
    return 0;

  int i = 0, j = 0, sum = 0, min = Integer.MAX_VALUE;

  while (j < a.length) {
    sum += a[j++];

    while (sum >= s) {
      min = Math.min(min, j - i);
      sum -= a[i++];
    }
  }

  return min == Integer.MAX_VALUE ? 0 : min;
}
```

written by jeantimex original link here