312 7

862. Shortest Subarray with Sum at Least K

Description (/problems/shortest-subarray-with-sum-at-least-k/description/) Hints (/problems/shortest-subarray-

Monotonic Queue Summary

17 VIEWS

0

Created at: December 11, 2018 6:54 PM

(/luxy622) luxy622 (/luxy622) ★ 41

Monotonic Queue

Hope it helps.

The following question can be solved by monotonic queue:

- **LC84. Largest Rectangle in Histogram**
- **LC239. Sliding Window Maximum**
- **LC739. Daily Temperatures**
- **LC862. Shortest Subarray with Sum at Least K**
- **LC901. Online Stock Span**
- **LC907. Sum of Subarray Minimums**

In general, the following "prototype" problems can be solved by monotonic queue:

Any DP problem where `A[i] = min(A[j:k]) + C` where `j < k <= i`

This is a sliding max/min window problem.

The task is to return the max/min elements in some sliding window. For example, we want a running max in the sliding windows, `amax = max(A[i:i+width])` .

Key observation: Given input array `A` , when `A[l] < A[r]` for `l < r` , then `A[l]` should never be retuned as the sliding max `amax` , once `A[r]` has entered the sliding window.

So we maintain a monotonic array with index **increasing** and value **decreasing**, because smaller elements like `A[l]` on the left are useless.

For example, with sliding window of fixed length 3,

> `A = [3, 1, 4, 3, 8] => monotonic queue is like [3], [3, 1], [4], [4, 3], [8]`
> when element `4` enters, we remove `[3, 1]` because they are on the left and smaller than `4`, no chance being chosen as the max element.

The head of the increasing queue is the running max!

The only unique thing here is that we can keep the elements in the window sorted. It brings great benefits because it takes O(1) to obtain the min/max element in the window.

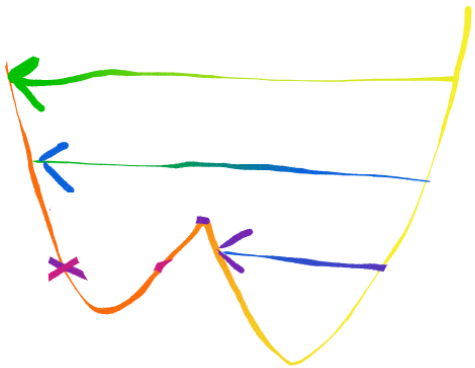
That's why any DP problem where `A[i] = min(A[j:k]) + C` for `j < k <= i` and some constant `C` can be solved by Monotonic Queue.

Find the nearest larger element on the left

Given array `A` and an element `A[i]`, the task is to find the maximum index `j < i` such that `A[j] > A[i]`. Namely, `A[j]` is the nearest larger element on the left of `A[i]`.

Key observation: given `A[k] < A[j] > A[i]` for `k < j < i`, `A[k]` never become the **nearest** element larger than `A[i]` because of `A[j]`.

So we should have a decreasing monotonic queue here. The arrow indicates that the mapping from element on the right to the nearest element on the left larger than it. The elements in the valley are ignored.



~~LC 85. Maximal Rectangle~~

Given a 2D binary matrix filled with 0's and 1's, find the largest rectangle containing only 1's and return its area.

Idea: convert 2D matrix to 1D height array. The task becomes **LC84. Largest Rectangle in Histogram** which is essentially "finding the index of the nearest previous value smaller than itself".

```
        if not matrix: return 0
        N, M = len(matrix), len(matrix[0])
        dp = [0] * (M + 1)
        area = 0

        for i in range(N):
            for j in range(M):
                # obtain the height based on each row
                if matrix[i][j] == '1':
                    dp[j] += 1
                else:
                    dp[j] = 0

            s = []
            for j in range(M + 1): # IMPORTANT: note that the last ZERO should
                if not s: s.append(j)
                else:
                    while s and dp[s[-1]] >= dp[j]:
                        x = s.pop()
                        if s: area = max(area, dp[x]*(j - s[-1] - 1))
                        else: area = max(area, dp[x]*j)
                    s.append(j)

        return area
```

LC862. Shortest Subarray with Sum at Least K

Return the length of the shortest, non-empty, contiguous subarray of A with sum at least K.

Key observation: If we accumulate array A to obtain B, then `B[l] <= B[r] - K` indicates `sum(A[l:r]) >= K`. Given `B[r]`, the problem is equivalent to finding the **nearest** previous element `B[l]` such that `B[l] <= B[r] - K`.

We maintain a **increasing queue** here because, given a new `B[i]`, the larger element on the left are inferior than `B[i]` as a candidate to make some future element `B[j] >= B[i] + K (j > i)`.

One extra optimization learnt from @lee215 (https://leetcode.com/problems/shortest-subarray-with-sum-at-least-k/discuss/143726/C%2B%2BJavaPython-O(N)-Using-Deque) is that we can also pop up the element on the left side `<= B[i] - K` of the **increasing** queue because, given current element `B[i]`, if a future element `B[j] > B[i]`, then `B[j] - K` would be within the queue after the removal of such elements `<= B[i] - K`; Otherwise, if a future element `B[j] > B[i]` then it never appears in the final results.

```
        Q = collections.deque([])

        B = [0]
        for a in A: B.append(B[-1] + a)

        res = float('inf')
        for i, b in enumerate(B):
            if not Q: Q.append(i)
            else:
                while Q and B[Q[-1]] > b: Q.pop()
                while Q and B[Q[0]] <= b - K:
                    res = min(res, i - Q[0])
                    Q.popleft()
                Q.append(i)
        return res if res < float('inf') else -1
```

More notes are kept at this github page
(https://github.com/xiaoylu/leetcode_category/tree/master/MonotonicQueue)

Comments: 0

Sort By

Type comment here... (Markdown is supported)

Preview | Post