Sliding Window Maximum

Given an array *nums*, there is a sliding window of size *k* which is moving from the very left of the array to the very right. You can only see the *k* numbers in the window. Each time the sliding window moves right by one position.

For example,
Given *nums* = `[1,3,-1,-3,5,3,6,7]`, and *k* = 3.

```
Window position                Max
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

Therefore, return the max sliding window as `[3,3,5,5,6,7]`.

Note:
You may assume *k* is always valid, ie: 1 ≤ k ≤ input array's size for non-empty array.

Follow up:
Could you solve it in linear time?

1. How about using a data structure such as deque (double-ended queue)?
2. The queue size need not be the same as the window's size.
3. Remove redundant elements and the queue should store only elements that need to be considered.

Solution 1

We scan the array from 0 to n-1, keep "promising" elements in the deque. The algorithm is amortized O(n) as each element is put and polled once.

At each i, we keep "promising" elements, which are potentially max number in window [i-(k-1),i] or any subsequent window. This means

1. If an element in the deque and it is out of i-(k-1), we discard them. We just need to poll from the head, as we are using a deque and elements are ordered as the sequence in the array
2. Now only those elements within [i-(k-1),i] are in the deque. We then discard elements smaller than a[i] from the tail. This is because if a[x] <a[i] and x<i, then a[x] has no chance to be the "max" in [i-(k-1),i], or any other subsequent window: a[i] would always be a better candidate.
3. As a result elements in the deque are ordered in both sequence in array and their value. At each step the head of the deque is the max element in [i-(k-1),i]

```
public int[] maxSlidingWindow(int[] a, int k) {
        if (a == null || k <= 0) {
            return new int[0];
        }
        int n = a.length;
        int[] r = new int[n-k+1];
        int ri = 0;
        // store index
        Deque<Integer> q = new ArrayDeque<>();
        for (int i = 0; i < a.length; i++) {
            // remove numbers out of range k
            while (!q.isEmpty() && q.peek() < i - k + 1) {
                q.poll();
            }
            // remove smaller numbers in k range as they are useless
            while (!q.isEmpty() && a[q.peekLast()] < a[i]) {
                q.pollLast();
            }
            // q contains index... r contains content
            q.offer(i);
            if (i >= k - 1) {
                r[ri++] = a[q.peek()];
            }
        }
        return r;
    }
```

written by zjm84812 original link here

Solution 2

The data structure used is know as Monotonic Queue. Clickhere for more information.

You can also view more solution on Github

```
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        deque<int> dq;
        vector<int> ans;
        for (int i=0; i<nums.size(); i++) {
            if (!dq.empty() && dq.front() == i-k) dq.pop_front();
            while (!dq.empty() && nums[dq.back()] < nums[i])
                dq.pop_back();
            dq.push_back(i);
            if (i>=k-1) ans.push_back(nums[dq.front()]);
        }
        return ans;
    }
};
```

written by zhaotianzju original link here

Solution 3

Sliding window minimum/maximum = monotonic queue. I smelled the solution just when I read the title. This is essentially same idea as others' deque solution, but this is much more standardized and modulized. If you ever need to use it in your real product, this code is definitely more preferable.

What does Monoqueue do here:

It has three basic options:

push: push an element into the queue; O (1) (amortized)

pop: pop an element out of the queue; O(1) (pop = remove, it can't report this element)

max: report the max element in queue;O(1)

It takes only O(n) time to process a N-size sliding window minimum/maximum problem.

Note: different from a priority queue (which takes O(nlogk) to solve this problem), it doesn't pop the max element: It pops the first element (in original order) in queue.

```
class Monoqueue
{
    deque<pair<int, int>> m_deque; //pair.first: the actual value,
                                   //pair.second: how many elements were deleted
between it and the one before it.
    public:
        void push(int val)
        {
            int count = 0;
            while(!m_deque.empty() && m_deque.back().first < val)
            {
                count += m_deque.back().second + 1;
                m_deque.pop_back();
            }
            m_deque.emplace_back(val, count);
        };
        int max()
        {
            return m_deque.front().first;
        }
        void pop ()
        {
            if (m_deque.front().second > 0)
            {
                m_deque.front().second --;
                return;
            }
            m_deque.pop_front();
        }
};
struct Solution {
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        vector<int> results;
        Monoqueue mq;
        k = min(k, (int)nums.size());
        int i = 0;
        for (;i < k - 1; ++i) //push first k - 1 numbers;
        {
            mq.push(nums[i]);
        }
        for (; i < nums.size(); ++i)
        {
            mq.push(nums[i]);            // push a new element to queue;
            results.push_back(mq.max()); // report the current max in queue;
            mq.pop();                    // pop first element in queue;
        }
        return results;
    }
};
```

written by fentoyal original link here