480. Sliding Window Median

Feb. 6, 2017 | 30.3K views

Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.

Examples:
`[2,3,4]` , the median is `3`

`[2,3]`, the median is `(2 + 3) / 2 = 2.5`

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. Your job is to output the median array for each window in the original array.

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

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

Therefore, return the median sliding window as `[1,-1,-1,3,5,6]`.

Note:
You may assume `k` is always valid, ie: `k` is always smaller than input array's size for non-empty array.
Answers within `10^-5` of the actual value will be accepted as correct.

A word of advice

This problem is a companion problem to 295. Find Median From Data Stream. This means that a lot of approaches to solve this problem are based on the methods to solve 295. Find Median From Data Stream. Perhaps try that problem before you approach this one.

Solution

Approach 1: Simple Sorting

Intuition

Do what the question says.

Algorithm

Store the numbers in a window container of size k. The following operations must take place:

1. **Inserting** the incoming element.
2. **Deleting** the outgoing element.
3. **Sorting** the window to find the medians.

One primitive approach is to copy k consecutive elements from the input to the window and keep sorting these every time. This constitutes duplication of effort.

We can do a bit better if we instead *insert* and *delete* one element per window shift. The challenge then is to maintain the window as **sorted**, before and after the *insert* and *delete* operations.

C++ | Python | Copy

```python
def medianSlidingWindow(self, nums, k):
    medians, window = [], []

    for i in xrange(len(nums)):

        # Find position where outgoing element should be removed from
        if i >= k:
          # window.remove(nums[i-k])        # this works too
          window.pop(bisect.bisect(window, nums[i - k]) - 1)

        # Maintain the sorted invariant while inserting incoming element
        bisect.insort(window, nums[i])

        # Find the medians
        if i >= k - 1:
          medians.append(float((window[k / 2]
            if k & 1 > 0
            else(window[k / 2 - 1] + window[k / 2]) * 0.5)))

    return medians
```

Python comes with an excellent `bisect` module to help perform efficient insert operations on lists while maintaining their sorted property.

Complexity Analysis

- Time complexity: $O(n \cdot k \log k)$ to $O(n \cdot k)$.
  - Copying elements into the container takes about $O(k)$ time each. This happens about $(n-k)$ times.
  - Sorting for each of the $(n-k)$ sliding window instances takes about $O(k \log k)$ time each.
  - Bisected insertion or deletion takes about $O(\log k)$ for searching and $O(k)$ for actual shifting of elements. This takes place about n times.
- Space complexity: $O(k)$ extra linear space for the window container.

Approach 2: Two Heaps (Lazy Removal)

Intuition

The idea is the same as Approach 3 from 295. Find Median From Data Stream. The only additional requirement is removing the outgoing elements from the window.

Since the window elements are stored in heaps, deleting elements that are not at the top of the heaps is a pain.

Some languages (like **Java**) provide implementations of the `PriorityQueue` class that allow for removing *arbitrarily* placed elements. Generally, using such features is not efficient nor is their portability assured.

Assuming that only the tops of heaps (and by extension the `PriorityQueue` class) are accessible, we need to find a way to efficiently invalidate and remove elements that are moving out of the sliding window.

At this point, an important thing to notice is the fact that if the two heaps are balanced, only the top of the heaps are actually needed to find the medians. This means that as long as we can somehow keep the heaps balanced, we could also keep some extraneous elements.

Thus, we can use hash-tables to keep track of invalidated elements. Once they reach the heap tops, we remove them from the heaps. This is the lazy removal technique.

An immediate challenge at this point is balancing the heaps while keeping extraneous elements. This is done by actually moving some elements to the heap which has extraneous elements, from the other heap. This cancels out the effect of having extraneous elements and maintains the invariant that the heaps are balanced.

NOTE: When we talk about keeping the heaps balanced, we are **not** referring to the actual heap sizes. We are only concerned with valid elements and hence when we talk about balancing heaps, we are referring to count of such elements.

Algorithm

- Two priority queues:
  1. A max-heap `lo` to store the smaller half of the numbers
  2. A min-heap `hi` to store the larger half of the numbers
- A hash-map or hash-table `hash_table` for keeping track of invalid numbers. It holds the count of the occurrences of all such numbers that have been invalidated and yet remain in the heaps.
- The max-heap `lo` is allowed to store, at worst, one more element more than the min-heap `hi`. Hence if we have processed k elements:
  - If $k = 2 \cdot n + 1 \quad (\forall n \in \mathbb{Z})$, then `lo` is allowed to hold $n+1$ elements, while `hi` can hold n elements.
  - If $k = 2 \cdot n \quad (\forall n \in \mathbb{Z})$, then both heaps are balanced and hold n elements each.

  This gives us the nice property that when the heaps are perfectly balanced, the median can be derived from the tops of both heaps. Otherwise, the top of the max-heap `lo` holds the legitimate median.

NOTE: As mentioned before, when we are talking about keeping the heaps balanced, the actual sizes of the heaps are irrelevant. Only the count of valid elements in both heaps matter.

- Keep a `balance` factor. It indicates three situations:
  - `balance` $= 0$: Both heaps are balanced or nearly balanced.
  - `balance` < 0: `lo` needs more valid elements. Elements from `hi` are moved to `lo`.
  - `balance` > 0: `hi` needs more valid elements. Elements from `lo` are moved to `hi`.
- Inserting an incoming number `in_num`:
  - If `in_num` is less than or equal to the top element of `lo`, then it can be inserted to `lo`. However this unbalances `hi` (`hi` has lesser valid elements now). Hence `balance` is incremented.
  - Otherwise, `in_num` must be added to `hi`. Obviously, now `lo` is unbalanced. Hence `balance` is decremented.
- Lazy removal of an outgoing number `out_num`:
  - If `out_num` is present in `lo`, then invalidating this occurrence will unbalance `lo` itself. Hence `balance` must be decremented.
  - If `out_num` is present in `hi`, then invalidating this occurrence will unbalance `hi` itself. Hence `balance` must be incremented.
  - We increment the count of this element in the hash_table table.
  - Once an invalid element reaches either of the heap tops, we remove them and decrement their counts in the hash_table table.

C++ | Copy

```cpp
vector<double> medianSlidingWindow(vector<int>& nums, int k)
{
    vector<double> medians;
    unordered_map<int, int> hash_table;
    priority_queue<int> lo;                                 // max heap
    priority_queue<int, vector<int>, greater<int> > hi;     // min heap

    int i = 0;      // index of current incoming element being processed

    // initialize the heaps
    while (i < k)
        lo.push(nums[i++]);
    for (int j = 0; j < k / 2; j++) {
        hi.push(lo.top());
        lo.pop();
    }

    while (true) {
        // get median of current window
        medians.push_back(k & 1 ? lo.top() : ((double)lo.top() + (double)hi.top()) * 0.5);

        if (i >= nums.size())
            break;                          // break if all elements processed

        int out_num = nums[i - k],          // outgoing element
            in_num = nums[i++],             // incoming element
            balance = 0;                    // balance factor
```

Complexity Analysis

- Time complexity: $O(2 \cdot n \log k) + O(n-k) \approx O(n \log k)$.
  - Either (or sometimes both) of the heaps gets every element inserted into it at least once. Collectively each of those takes about $O(\log k)$ time. That is n such insertions.
  - About $(n-k)$ removals from the top of the heaps take place (the number of sliding window instances). Each of those takes about $O(\log k)$ time.
  - Hash table operations are assumed to take $O(1)$ time each. This happens roughly the same number of times as removals from heaps take place.
- Space complexity: $O(k) + O(n) \approx O(n)$ extra linear space.
  - The heaps collectively require $O(k)$ space.
  - The hash table needs about $O(n-k)$ space.

Approach 3: Two Multisets

Intuition

One can see that `multiset`s are a great way to keep elements sorted while providing efficient access to the first and last elements. Inserting and deleting arbitrary elements are also fairly efficient operations in a `multiset`. (Refer to Approach 4 Intuition for 295. Find Median From Data Stream)

Thus, if the previous approach gives you too much heartburn, consider replacing the use of `priority_queue` with `multiset`.

Algorithm

Inserting or deleting an element is straight-forward. Balancing the heaps takes the same route as Approach 3 of 295. Find Median From Data Stream.

C++ | Copy

```cpp
vector<double> medianSlidingWindow(vector<int>& nums, int k)
{
    vector<double> medians;
    multiset<int> lo, hi;

    for (int i = 0; i < nums.size(); i++) {
        //remove outgoing element
        if (i >= k) {
            if (nums[i - k] <= *lo.rbegin())
                lo.erase(lo.find(nums[i - k]));
            else
                hi.erase(hi.find(nums[i - k]));
        }

        // insert incoming element
        lo.insert(nums[i]);

        // balance the sets
        hi.insert(*lo.rbegin());
        lo.erase(prev(lo.end()));

        if (lo.size() < hi.size()) {
            lo.insert(*hi.begin());
            hi.erase(hi.begin());
        }

        // get median
```

Complexity Analysis

- Time complexity: $O((n-k) \cdot 6 \cdot \log k) \approx O(n \log k)$.
  - At worst, there are three set insertions and three set deletions from the start or end. Each of these takes about $O(\log k)$ time.
  - Finding the mean takes constant $O(1)$ time since the start or ends of sets are directly accessible.
  - Each of these steps takes place about $(n-k)$ times (the number of sliding window instances).
- Space complexity: $O(k)$ extra linear space to hold contents of the window.

Approach 4: Multiset and Two Pointers

Intuition

This is same as Approach 4 for 295. Find Median From Data Stream.

> But, we don't actually need two pointers.

Median elements are derived using a single iterator position (when the window size k is odd) or two consecutive iterator positions (when k is even). Hence keeping track of *only* one pointer is sufficient. The other pointer can be *implicitly derived* when required.

Algorithm

- A single iterator `mid`, which iterates over the `window` multiset. It is analogous to `upper_median` in Approach 4 for 295. Find Median From Data Stream. `lower_median` is *implicitly derived* from `mid`. It's either equal to `mid` (when the window size k is odd) or `prev(mid)` [1].
- We start with populating our multiset `window` with the first k elements. We set `mid` to the $\lfloor k/2 \rfloor$ indexed element in `window` (`0`-based indexing; Multisets always maintain their sorted property).
- While inserting an element `num` into `window`, three cases arise:
  1. `num` is less than the value of upper median `mid`.
  2. `num` is greater than the value of upper median `mid`.
  3. `num` is equal to the value of upper median `mid`. This situation is often handled as language-dependent. Since C++ `multiset` insert elements at the end of their equal range, this situation is essentially the same as the previous case.
  4. For the first case, `num` is inserted before the upper median element `mid`. Thus `mid` now, no longer points to the $\lfloor k/2 \rfloor$ indexed element. In fact it points to the $\lfloor k/2 \rfloor + 1$ indexed element. We fix that by decrementing `mid`.
  5. For the second and third cases, `num` is inserted after the upper median element `mid` and hence does not spoil the `mid` iterator. It still points to the $\lfloor k/2 \rfloor$ indexed element.
  6. Of course, the window size just increased to $k+1$ in all three cases. That will easily be fixed by removing the element that is about to exit the window.
- While removing an element `num`, the same three cases arise as when we wanted to insert an element:
  1. `num` is less than the value of upper median `mid`.
  2. `num` is greater than the value of upper median `mid`.
  3. `num` is equal to the value of upper median `mid`. Since `mid` will point to the first occurrence of `num` in the multiset `window` and we deterministically remove the first occurrence (take note that we use `std::multiset::lower_bound()` [2] to find the correct occurrence to erase), this case is handled in the same manner as the first case.
  4. In the first and third cases, removing `num` will spoil the `mid` iterator. Thus we need to fix that by incrementing `mid` before we remove that element.
  5. For the second case, the `mid` iterator is not spoiled. No further action is required.
  6. Once this element has been removed, the window size returns to being k.
- After insertion of the incoming element and removal of the outgoing element, we are left again with some nice invariants:
  1. Window size is again k.
  2. The window is still fully sorted.
  3. `mid` still points to the $\lfloor k/2 \rfloor$ indexed element.
- Finding the median of the window is easy! It is simply the **mean** of the elements pointed to by the two pointers `lo_median` and `hi_median`. In our case those are `mid` or `prev(mid)` (as decided by whether k is odd or even) , and `mid` respectively.

[3]

C++ | Copy

```cpp
vector<double> medianSlidingWindow(vector<int>& nums, int k)
{
    vector<double> medians;
    multiset<int> window(nums.begin(), nums.begin() + k);

    auto mid = next(window.begin(), k / 2);

    for (int i = k;; i++) {

        // Push the current median
        medians.push_back(((double)(*mid) + *next(mid, k % 2 - 1)) * 0.5);

        // If all done, break
        if (i == nums.size())
            break;

        // Insert incoming element
        window.insert(nums[i]);
        if (nums[i] < *mid)
            mid--;                  // same as mid = prev(mid)

        // Remove outgoing element
        if (nums[i - k] <= *mid)
            mid++;                  // same as mid = next(mid)

        window.erase(window.lower_bound(nums[i - k]));
    }
```

Complexity Analysis

- Time complexity: $O((n-k) \log k) + O(k) \approx O(n \log k)$.
  - Initializing `mid` takes about $O(k)$ time.
  - Inserting or deleting a number takes $O(\log k)$ time for a standard `multiset` scheme. [4]
  - Finding the mean takes constant $O(1)$ time since the median elements are directly accessible from `mid` iterator.
  - The last two steps take place about $(n-k)$ times (the number of sliding window instances).
- Space complexity: $O(k)$ extra linear space to hold contents of the window.

Further Thoughts

As noted before, this problem is essentially an extension to 295. Find Median From Data Stream. That problem had a lot of ways to go about, that frankly, are not of much use in an interview. But they are interesting to follow all the same. If you are interested take a look here. Try extending those methods to this problem.

---

1. `std::prev()` is a C++ method to find the previous element to the current one being pointed to by an iterator. ↩
2. Had we used `std::multiset::find()`, there was no guarantee that the first occurrence of `num` would be found. Although the contrary did happen in all of our tests, I don't recommend using it. Your mileage may vary. ↩
3. Shout-out to @votrubac and @StefanPochmann! ↩
4. Hinting can reduce that to amortized constant $O(1)$ time. ↩