Sliding Window Median

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: 1 ≤ k ≤ input array's size for non-empty array.

Solution 1

Keep the window elements in a multiset and keep an iterator pointing to the middle value (to "index" k/2, to be precise). Thanks to @votrubac's solution and comments.

```
vector<double> medianSlidingWindow(vector<int>& nums, int k) {
    multiset<int> window(nums.begin(), nums.begin() + k);
    auto mid = next(window.begin(), k / 2);
    vector<double> medians;
    for (int i=k; ; i++) {

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

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

        // Insert nums[i].
        window.insert(nums[i]);
        if (nums[i] < *mid)
            mid--;

        // Erase nums[i-k].
        if (nums[i-k] <= *mid)
            mid++;
        window.erase(window.lower_bound(nums[i-k]));
    }
}
```

written by StefanPochmann original link here

Solution 2

There are a few solutions using BST with worst case time complexity O(n*k), but we know k can be become large. I wanted to come up with a solution that is guaranteed to run in O(n*log(n)) time. This is in my opinion the best solution so far.

The idea is inspired by solutions to Find Median from Data Stream: use two heaps to store numbers in the sliding window. However there is the issue of numbers moving out of the window, and it turns out that a hash table that records these numbers will just work (and is surprisingly neat). The recorded numbers will only be deleted when they come to the top of the heaps.

```
class Solution {
public:
    vector<double> medianSlidingWindow(vector<int>& nums, int k) {
        vector<double> medians;
        unordered_map<int, int> hash;                          // count numbers
to be deleted
        priority_queue<int, vector<int>> bheap;                // heap on the bot
tom
        priority_queue<int, vector<int>, greater<int>> theap;  // heap on the top

        int i = 0;

        // Initialize the heaps
        while (i < k)  { bheap.push(nums[i++]); }
        for (int count = k/2; count > 0; --count) {
            theap.push(bheap.top()); bheap.pop();
        }

        while (true) {
            // Get median
            if (k % 2) medians.push_back(bheap.top());
            else medians.push_back( ((double)bheap.top() + theap.top()) / 2 );

            if (i == nums.size()) break;
            int m = nums[i-k], n = nums[i++], balance = 0;

            // What happens to the number m that is moving out of the window
            if (m <= bheap.top())  { --balance;  if (m == bheap.top()) bheap.pop(
); else ++hash[m]; }
            else                   { ++balance;  if (m == theap.top()) theap.pop(
); else ++hash[m]; }

            // Insert the new number n that enters the window
            if (!bheap.empty() && n <= bheap.top())  { ++balance; bheap.push(n);
}
            else                                    { --balance; theap.push(n);
}

            // Rebalance the bottom and top heaps
            if      (balance < 0)  { bheap.push(theap.top()); theap.pop(); }
            else if (balance > 0)  { theap.push(bheap.top()); bheap.pop(); }

            // Remove numbers that should be discarded at the top of the two heap
s
            while (!bheap.empty() && hash[bheap.top()])  { --hash[bheap.top()]; b
heap.pop(); }
            while (!theap.empty() && hash[theap.top()])  { --hash[theap.top()]; t
heap.pop(); }
        }

        return medians;
    }
};
```

Since both heaps will never have a size greater than n, the time complexity is

O(n*log(n)) in the worst case.

written by ipt original link here

Solution 3

The idea is to maintain a BST of the window and just search for the k/2 largest element and k/2 smallest element then the average of these two is the median of the window.

Now if the STL's multiset BST maintained how many element were in each subtree finding each median would take O(log k) time but since it doesn't it takes O(k) time to find each median.

```
public:
    vector<double> medianSlidingWindow(vector<int>& nums, int k) {
        multiset<int> mp;
        vector<double> med;

        for(int i=0; i<k-1; ++i) mp.insert(nums[i]);

        for(int i=k-1; i< nums.size(); ++i){
            mp.insert(nums[i]); // Add the next number

            auto itb = mp.begin(); advance(itb, (k-1)/2); //Find the lower median
            auto ite = mp.end(); advance(ite, -(k+1)/2); //Find the upper median

            double avg = ((long)(*itb) + (*ite)) / 2.0;
            med.push_back(avg);

            mp.erase(mp.find(nums[i-k+1])); //Remove the oldest element
        }

        return med;
    }
};
```

written by kevin36 original link here