Sliding Window Median (hard)

We'll cover the following

Problem Statement

Given an array of numbers and a number 'k', find the median of all the 'k' sized sub-arrays (or windows) of the array.

Example 1:

Input: nums=[1, 2, -1, 3, 5], k = 2
Output: [1.5, 0.5, 1.0, 4.0]
Explanation: Lets consider all windows of size '2':

- [1, 2, -1, 3, 5] -> median is 1.5

- [1, 2, -1, 3, 5] -> median is 0.5
- [1, 2, -1, 3, 5] -> median is 1.0
- [1, 2, -1, 3, 5] -> median is 4.0

Example 2:

Input: nums=[1, 2, -1, 3, 5], k = 3
Output: [1.0, 2.0, 3.0]
Explanation: Lets consider all windows of size '3':

- [1, 2, -1, 3, 5] -> median is 1.0
- [1, 2, -1, 3, 5] -> median is 2.0
- [1, 2, -1, 3, 5] -> median is 3.0

Try it yourself

Try solving this question here:

Java	Python3	JS	C++

```
class SlidingWindowMedian {

  find_sliding_window_median(nums, k) {
    result = [];
    // TODO: Write your code here
    return result;
  }
};



var slidingWindowMedian = new SlidingWindowMedian()
result = slidingWindowMedian.find_sliding_window_median(
  [1, 2, -1, 3, 5], 2)

console.log(`Sliding window medians are: ${result}`)

slidingWindowMedian = new SlidingWindowMedian()
result = slidingWindowMedian.find_sliding_window_median(
  [1, 2, -1, 3, 5], 3)
console.log(`Sliding window medians are: ${result}`)
```

Run Save Reset

Solution

This problem follows the **Two Heaps** pattern and share similarities with Find the Median of a Number Stream (https://www.educative.io/collection/page/5668639101419520/5671464854355968/6308926461050880/). We can follow a similar approach of maintaining a max-heap and a min-heap for the list of numbers to find their median.

The only difference is that we need to keep track of a sliding window of ‘k’ numbers. This means, in each iteration, when we insert a new number in the heaps, we need to remove one number from the heaps which is going out of the sliding window. After the removal, we need to rebalance the heaps in the same way that we did while inserting.

Code

Here is what our algorithm will look like:

Java Python3 C++ JS

```
const Heap = require('./collections/heap'); //http://www.collectionsjs.com


class SlidingWindowMedian {
  constructor() {
    this.maxHeap = new Heap([], null, ((a, b) => a - b));
    this.minHeap = new Heap([], null, ((a, b) => b - a));
  }

  find_sliding_window_median(nums, k) {
    const result = Array(nums.length - k + 1).fill(0.0);
    for (let i = 0; i < nums.length; i++) {
      if (this.maxHeap.length === 0 || nums[i] <= this.maxHeap.peek()) {
        this.maxHeap.push(nums[i]);
      } else {
        this.minHeap.push(nums[i]);
      }

      this.rebalance_heaps();

      if (i - k + 1 >= 0) { // if we have at least 'k' elements in the sliding w
        // add the median to the the result array
        if (this.maxHeap.length === this.minHeap.length) {
          // we have even number of elements, take the average of middle two ele
          result[i - k + 1] = this.maxHeap.peek() / 2.0 + this.minHeap.peek() / 
        } else { // because max-heap will have one more element than the min-hea
          result[i - k + 1] = this.maxHeap.peek();
        }

        // remove the element going out of the sliding window
        const elementToBeRemoved = nums[i - k + 1];
        if (elementToBeRemoved <= this.maxHeap.peek()) {
          this.maxHeap.delete(elementToBeRemoved); // delete from heap
        } else {
          this.minHeap.delete(elementToBeRemoved); // delete from heap
        }

        this.rebalance_heaps();
      }
    }

    return result;
  }

  rebalance_heaps() {
    // either both the heaps will have equal number of elements or max-heap will
    // one more element than the min-heap
    if (this.maxHeap.length > this.minHeap.length + 1) {
      this.minHeap.push(this.maxHeap.pop());
    } else if (this.maxHeap.length < this.minHeap.length) {
      this.maxHeap.push(this.minHeap.pop());
    }
  }
}


let slidingWindowMedian = new SlidingWindowMedian();
result = slidingWindowMedian.find_sliding_window_median([1, 2, -1, 3, 5], 2);
console.log(`Sliding window medians are: ${result}`);

slidingWindowMedian = new SlidingWindowMedian();
result = slidingWindowMedian.find_sliding_window_median([1, 2, -1, 3, 5], 3);
console.log(`Sliding window medians are: ${result}`);
```

Run Save Reset

Time complexity

The time complexity of our algorithm is $O(N * K)$ where ‘N’ is the total number of elements in the input array and ‘K’ is the size of the sliding window. This is due to the fact that we are going through all the ‘N’ numbers and, while doing so, we are doing two things:

1. Inserting/removing numbers from heaps of size ‘K’. This will take $O(logK)$
2. Removing the element going out of the sliding window. This will take $O(K)$ as we will be searching this element in an array of size ‘K’ (i.e., a heap).

Space complexity

Ignoring the space needed for the output array, the space complexity will be $O(K)$ because, at any time, we will be storing all the numbers within the sliding window.

Back
Find the Median of a Number Stream (...

Next
Maximize Capital (hard)

✓ Mark as Completed

Report an Issue