

Grokking the Coding Interview: Patterns for Coding Questions

39% completed

Search Course

- Count Paths for a Sum (medium)
- Problem Challenge 1
- Solution Review: Problem Challenge 1
- Problem Challenge 2
- Solution Review: Problem Challenge 2

Pattern: Two Heaps

- Introduction
- Find the Median of a Number Stream (medium)
- Sliding Window Median (hard)**
- Maximize Capital (hard)
- Problem Challenge 1
- Solution Review: Problem Challenge 1

Pattern: Subsets

- Introduction
- Subsets (easy)
- Subsets With Duplicates (easy)
- Permutations (medium)
- String Permutations by changing case (medium)
- Balanced Parentheses (hard)
- Unique Generalized Abbreviations (hard)
- Problem Challenge 1
- Solution Review: Problem Challenge 1
- Problem Challenge 2
- Solution Review: Problem Challenge 2
- Problem Challenge 3
- Solution Review: Problem Challenge 3

Pattern: Modified

Binary Search

- Introduction
- Order-agnostic Binary Search (easy)
- Ceiling of a Number (medium)
- Next Letter (medium)
- Number Range (medium)
- Search in a Sorted Infinite Array (medium)
- Minimum Difference Element (medium)
- Bitonic Array Maximum (easy)
- Problem Challenge 1
- Solution Review: Problem Challenge 1
- Problem Challenge 2
- Solution Review: Problem Challenge 2
- Problem Challenge 3
- Solution Review: Problem Challenge 3

Sliding Window Median (hard)

We'll cover the following

- Problem Statement
- Try it yourself
- Solution
 - Code
 - Time complexity
 - Space complexity

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:

JavaPython3JS C++

```
1 class SlidingWindowMedian {
2
3   find_sliding_window_median(nums, k) {
4     result = [];
5     // TODO: Write your code here
6     return result;
7   }
8 };
9
10
11
12 var slidingWindowMedian = new SlidingWindowMedian()
13 result = slidingWindowMedian.find_sliding_window_median(
14   [1, 2, -1, 3, 5], 2)
15
16 console.log('Sliding window medians are: ${result}')
17
18 slidingWindowMedian = new SlidingWindowMedian()
19 result = slidingWindowMedian.find_sliding_window_median(
20   [1, 2, -1, 3, 5], 3)
21 console.log('Sliding window medians are: ${result}')
22
```

RUNSAVERESET

Close

Output1.811s

Sliding window medians are:
Sliding window medians are:

Solution

This problem follows the **Two Heaps** pattern and share similarities with [Find the Median of a Number Stream](#)

Pattern: Bitwise XOR

Introduction

Single Number (easy)

Two Single Numbers (medium)

Complement of Base 10 Number (medium)

Problem Challenge 1

Solution Review: Problem Challenge 1

Pattern: Top 'K' Elements

Introduction

Top 'K' Numbers (easy)

Kth Smallest Number (easy)

'K' Closest Points to the Origin (easy)

Connect Ropes (easy)

Top 'K' Frequent Numbers (medium)

Frequency Sort (medium)

Kth Largest Number in a Stream (medium)

'K' Closest Numbers (medium)

Maximum Distinct Elements (medium)

Sum of Elements (medium)

Rearrange String (hard)

Problem Challenge 1

Solution Review: Problem Challenge 1

Problem Challenge 2

Solution Review: Problem Challenge 2

Problem Challenge 3

Solution Review: Problem Challenge 3

Pattern: K-way merge

Introduction

Merge K Sorted Lists (medium)

Kth Smallest Number in M Sorted Lists (Medium)

Kth Smallest Number in a Sorted Matrix (Hard)

Smallest Number Range (Hard)

Problem Challenge 1

Solution Review: Problem Challenge 1

Pattern : 0/1 Knapsack (Dynamic Programming)

Introduction

0/1 Knapsack (medium)

Equal Subset Sum Partition (medium)

Subset Sum (medium)

Minimum Subset Sum Difference (hard)

Problem Challenge 1

Solution Review: Problem Challenge 1

Problem Challenge 2

Solution Review: Problem Challenge 2

Pattern: Topological Sort (Graph)

Introduction

MW

Explore

Tracks

My Courses

Edpresso

Refer a Friend

Create

This problem follows the [Two Heaps Pattern](#) and share similarities with [Find the Median of a Number Stream](#). 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:

JavaPython3C++JS

```
1 const Heap = require('./collections/heap'); //http://www.collectionsjs.com
2
3
4 class SlidingWindowMedian {
5   constructor() {
6     this.maxHeap = new Heap([], null, ((a, b) => a - b));
7     this.minHeap = new Heap([], null, ((a, b) => b - a));
8   }
9
10  find_sliding_window_median(nums, k) {
11    const result = Array(nums.length - k + 1).fill(0.0);
12    for (let i = 0; i < nums.length; i++) {
13      if (this.maxHeap.length === 0 || nums[i] <= this.maxHeap.peek()) {
14        this.maxHeap.push(nums[i]);
15      } else {
16        this.minHeap.push(nums[i]);
17      }
18
19      this.rebalance_heaps();
20
21      if (i - k + 1 >= 0) { // if we have at least 'k' elements in the sliding window
22        // add the median to the the result array
23        if (this.maxHeap.length === this.minHeap.length) {
24          // we have even number of elements, take the average of middle two elements
25          result[i - k + 1] = this.maxHeap.peek() / 2.0 + this.minHeap.peek() / 2.0;
26        } else { // because max-heap will have one more element than the min-heap
27          result[i - k + 1] = this.maxHeap.peek();
28        }
29      }
30    }
31  }
32}
```

RUN

SAVE

RESET

Close

Output5.984s

Sliding window medians are: 1.5,0.5,1.4
Sliding window medians are: 1.2,3

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 $O(\log K)$
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.

Interviewing soon? We've partnered with [Hired](#) so that companies apply to you, instead of the other way around. [See how](#)

← Back

Find the Median of a Number Stream [...]

MARK AS COMPLETED

Next →

Maximize Capital (hard)