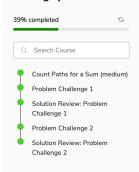


## Grokking the Coding Interview: Patterns for Coding Questions



#### Pattern: Two Heaps



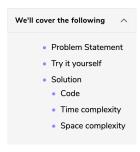
# Challenge 1 Pattern: Subsets



### Pattern: Modified Binary Search



## Sliding Window Median (hard)



#### **Problem Statement**

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

#### Example 1:

 $\label{limits} Input: nums=[1,2,-1,3,5], \ k=2$   $\label{limits} Output: [1.5,0.5,1.0,4.0]$  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:

```
Python3 Js JS
                                      (A) C++
       find_sliding_window_median(nums, k) {
        result = [];
    var slidingWindowMedian = new SlidingWindowMedian()
     result = slidingWindowMedian.find_sliding_window_median(
    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}`)
                                                                                         SAVE
                                                                                                     RESET
                                                                                                              0
                                                                                                         Close
                                                                                                         1.811s
Output
 Sliding window medians are
```

#### Solution

This problem follows the Two Heaps nattern and share similarities with Find the Median of a Number

# Pattern: Bitwise XOR 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 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 Kth Smallest Number in a Sorted 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 Pattern: Topological Sort (Graph)

**(D)** 

Explore

♦

Mv Courses

Ë

Edpresso

2+

Refer a

Create

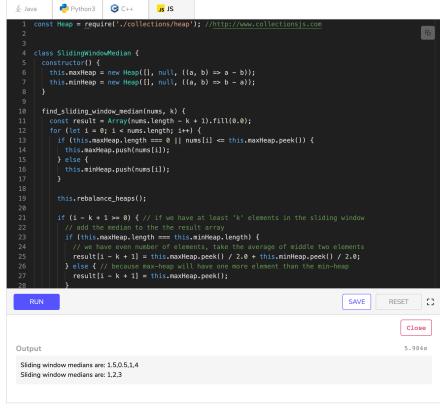
O Introduction

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:



#### 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.

Interviewing soon? We've partnered with Hired so that companies apply to you, instead of the  $\phantom{a} imes$ other way around. See how ①



