

## Grokking the Coding Interview: Patterns for Coding Questions

39% completed

### Pattern: Tree Breadth

#### First Search

- Introduction
- Binary Tree Level Order Traversal (easy)
- Reverse Level Order Traversal (easy)
- Zigzag Traversal (medium)
- Level Averages in a Binary Tree (easy)
- Minimum Depth of a Binary Tree (easy)
- Level Order Successor (easy)
- Connect Level Order Siblings (medium)
- Problem Challenge 1
- Solution Review: Problem Challenge 1
- Problem Challenge 2
- Solution Review: Problem Challenge 2

### Pattern: Tree Depth

#### First Search

- Introduction
- Binary Tree Path Sum (easy)
- All Paths for a Sum (medium)
- Sum of Path Numbers (medium)
- Path With Given Sequence (medium)
- 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

## Find the Median of a Number Stream (medium)

### We'll cover the following

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

### Problem Statement

Design a class to calculate the median of a number stream. The class should have the following two methods:

- `insertNum(int num)` : stores the number in the class
- `findMedian()` : returns the median of all numbers inserted in the class

If the count of numbers inserted in the class is even, the median will be the average of the middle two numbers.

#### Example 1:

```
1. insertNum(3)
2. insertNum(1)
3. findMedian() -> output: 2
4. insertNum(5)
5. findMedian() -> output: 3
6. insertNum(4)
7. findMedian() -> output: 3.5
```

### Try it yourself

Try solving this question here:

JavaPython3JS C++

```
1 class MedianOfAStream {
2   insert_num(num) {
3     // TODO: Write your code here
4     return -1;
5   }
6
7   find_median(self) {
8     // TODO: Write your code here
9     return 0.0
10  }
11 };
12
13
14
15 var medianOfAStream = new MedianOfAStream()
16 medianOfAStream.insert_num(3)
17 medianOfAStream.insert_num(1)
18 console.log('The median is: ${medianOfAStream.find_median()}')
19 medianOfAStream.insert_num(5)
20 console.log('The median is: ${medianOfAStream.find_median()}')
21 medianOfAStream.insert_num(4)
22 console.log('The median is: ${medianOfAStream.find_median()}')
23
```

RUNSAVERESET

### Solution

As we know, the median is the middle value in an ordered integer list. So a brute force solution could be to maintain a sorted list of all numbers inserted in the class so that we can efficiently return the median whenever required. Inserting a number in a sorted list will take  $O(N)$  time if there are 'N' numbers in the list. This insertion will be similar to the [Insertion sort](#). Can we do better than this? Can we utilize the fact that we don't need the fully sorted list - we are only interested in finding the middle element?

Assume 'x' is the median of a list. This means that half of the numbers in the list will be smaller than (or equal to) 'x' and half will be greater than (or equal to) 'x'. This leads us to an approach where we can divide the list into two halves: one half to store all the smaller numbers (let's call it `smallNumList`) and one half to store the larger numbers (let's call it `largNumList`). The median of all the numbers will either be the largest number in the `smallNumList` or the smallest number in the `largNumList`. If the total number of elements is even, the median will be the average of these two numbers.

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

## 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)

The best data structure that comes to mind to find the smallest or largest number among a list of numbers is a **Heap**. Let's see how we can use a heap to find a better algorithm.

1. We can store the first half of numbers (i.e., `smallNumList`) in a **Max Heap**. We should use a **Max Heap** as we are interested in knowing the largest number in the first half.
2. We can store the second half of numbers (i.e., `largeNumList`) in a **Min Heap**, as we are interested in knowing the smallest number in the second half.
3. Inserting a number in a heap will take  $O(\log N)$ , which is better than the brute force approach.
4. At any time, the median of the current list of numbers can be calculated from the top element of the two heaps.

Let's take the Example-1 mentioned above to go through each step of our algorithm:

1. `insertNum(3)`: We can insert a number in the **Max Heap** (i.e. first half) if the number is smaller than the top (largest) number of the heap. After every insertion, we will balance the number of elements in both heaps, so that they have an equal number of elements. If the count of numbers is odd, let's decide to have more numbers in max-heap than the **Min Heap**.

max-heap

3

min-heap

null

2. `insertNum(1)`: As '1' is smaller than '3', let's insert it into the **Max Heap**.

max-heap

3

1

min-heap

null

Now, we have two elements in the **Max Heap** and no elements in **Min Heap**. Let's take the largest element from the **Max Heap** and insert it into the **Min Heap**, to balance the number of elements in both heaps.

max-heap

1

min-heap

3

3. `findMedian()`: As we have an even number of elements, the median will be the average of the top element of both the heaps  $\rightarrow (1 + 3)/2 = 2.0$
4. `insertNum(5)`: As '5' is greater than the top element of the **Max Heap**, we can insert it into the **Min Heap**. After the insertion, the total count of elements will be odd. As we had decided to have more numbers in the **Max Heap** than the **Min Heap**, we can take the top (smallest) number from the **Min Heap** and insert it into the **Max Heap**.

max-heap

3

1

min-heap

5

5. `findMedian()`: Since we have an odd number of elements, the median will be the top element of **Max Heap**  $\rightarrow 3$ . An odd number of elements also means that the **Max Heap** will have one extra element than the **Min Heap**.
6. `insertNum(4)`: Insert '4' into **Min Heap**.

max-heap

3

1

min-heap

4

5

7. `findMedian()`: As we have an even number of elements, the median will be the average of the top element of both the heaps  $\rightarrow (3 + 4)/2 = 3.5$

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

Topological Sort (medium)

Tasks Scheduling (medium)

Tasks Scheduling Order (medium)

All Tasks Scheduling Orders (hard)

Alien Dictionary (hard)

Problem Challenge 1

Solution Review: Problem Challenge 1

Problem Challenge 2

Solution Review: Problem Challenge 2

Miscellaneous

Kth Smallest Number (hard)

Conclusions

Where to Go from Here

Mark Course as Completed

MW

Explore

Tracks

My Courses

Edpresso

Refer a Friend

Create

Code

Here is what our algorithm will look like:

JavaPython3C++JS

```
18 // more element than the min-heap
19 if (this.maxHeap.length > this.minHeap.length + 1) {
20     this.minHeap.push(this.maxHeap.pop());
21 } else if (this.maxHeap.length < this.minHeap.length) {
22     this.maxHeap.push(this.minHeap.pop());
23 }
24 }
25
26 find_median() {
27     if (this.maxHeap.length === this.minHeap.length) {
28         // we have even number of elements, take the average of middle two elements
29         return this.maxHeap.peek() / 2.0 + this.minHeap.peek() / 2.0;
30     }
31
32     // because max-heap will have one more element than the min-heap
33     return this.maxHeap.peek();
34 }
35 }
36
37
38 const medianOfAStream = new MedianOfAStream();
39 medianOfAStream.insert_num(3);
40 medianOfAStream.insert_num(1);
41 console.log(`The median is: ${medianOfAStream.find_median()}`);
42 medianOfAStream.insert_num(5);
43 console.log(`The median is: ${medianOfAStream.find_median()}`);
44 medianOfAStream.insert_num(4);
45 console.log(`The median is: ${medianOfAStream.find_median()}`);
```

RUN

SAVE

RESET

Time complexity

The time complexity of the `insertNum()` will be  $O(\log N)$  due to the insertion in the heap. The time complexity of the `findMedian()` will be  $O(1)$  as we can find the median from the top elements of the heaps.

Space complexity

The space complexity will be  $O(N)$  because, as at any time, we will be storing all the numbers.

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

← Back

Introduction

MARK AS COMPLETED

Next →

Sliding Window Median (hard)

[Report an Issue](#) [Ask a Question](#)