

Grokking the Coding Interview: Patterns for Coding Questions

25% completed

- Solution Review: Problem Challenge 2
- Problem Challenge 3
- Solution Review: Problem Challenge 3**

Pattern: Cyclic Sort

- Introduction
- Cyclic Sort (easy)
- Find the Missing Number (easy)
- Find all Missing Numbers (easy)
- Find the Duplicate Number (easy)
- Find all Duplicate Numbers (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: In-place Reversal of a LinkedList

- Introduction
- Reverse a LinkedList (easy)
- Reverse a Sub-list (medium)
- Reverse every K-element Sub-list (medium)
- Problem Challenge 1
- Solution Review: Problem Challenge 1
- Problem Challenge 2
- Solution Review: Problem Challenge 2

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

Solution Review: Problem Challenge 3

We'll cover the following

- Employee Free Time (hard)
- Solution
 - Using a Heap to Sort the Intervals
- Code
 - Time complexity
 - Space complexity

Employee Free Time (hard)

For 'K' employees, we are given a list of intervals representing the working hours of each employee. Our goal is to find out if there is a **free interval that is common to all employees**. You can assume that each list of employee working hours is sorted on the start time.

Example 1:

```
Input: Employee Working Hours=[[1,3], [5,6]], [[2,3], [6,8]]
Output: [3,5]
Explanation: Both the employees are free between [3,5].
```

Example 2:

```
Input: Employee Working Hours=[[1,3], [9,12]], [[2,4]], [[6,8]]
Output: [4,6], [8,9]
Explanation: All employees are free between [4,6] and [8,9].
```

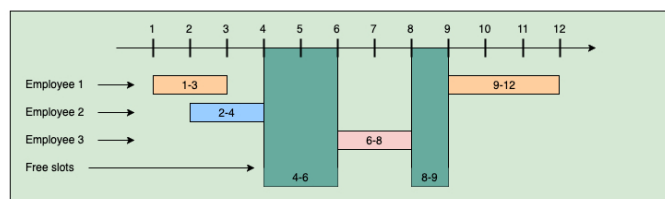
Example 3:

```
Input: Employee Working Hours=[[1,3]], [[2,4]], [[3,5], [7,9]]
Output: [5,7]
Explanation: All employees are free between [5,7].
```

Solution

This problem follows the [Merge Intervals](#) pattern. Let's take the above-mentioned example (2) and visually draw it:

```
Input: Employee Working Hours=[[1,3], [9,12]], [[2,4]], [[6,8]]
Output: [4,6], [8,9]
```



One simple solution can be to put the working hours of all employees in a list and sort them on the start time. Then we can iterate through the list to find the gaps. Let's dig deeper. Sorting the intervals of the above example will give us:

```
[1,3], [2,4], [6,8], [9,12]
```

We can now iterate through these intervals, and whenever we find non-overlapping intervals (e.g., [2,4] and [6,8]), we can calculate a free interval (e.g., [4,6]). This algorithm will take $O(N * \log N)$ time, where 'N' is the total number of intervals. This time is needed because we need to sort all the intervals. The space complexity will be $O(N)$, which is needed for sorting. Can we find a better solution?

Using a Heap to Sort the Intervals

One fact that we are not utilizing is that each employee list is individually sorted!

How about we take the first interval of each employee and insert it in a **Min Heap**. This **Min Heap** can always give us the interval with the smallest start time. Once we have the smallest start-time interval, we can then compare it with the next smallest start-time interval (again from the **Heap**) to find the gap. This interval comparison is similar to what we suggested in the previous approach.

Whenever we take an interval out of the **Min Heap**, we can insert the next interval of the same employee. This also means that we need to know which interval belongs to which employee.

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

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

MW

Explore

Tracks

My Courses

Edpresso

Refer a Friend

Create

Code

Here is what our algorithm will look like:

JavaPython3C++JS

```
1 const Heap = require('./collections/heap'); //http://www.collectionsjs.com
2
3 class Interval {
4   constructor(start, end) {
5     this.start = start;
6     this.end = end;
7   }
8
9   print_interval() {
10    process.stdout.write(`${this.start}, ${this.end}]\n`);
11  }
12 }
13
14 class EmployeeInterval {
15   constructor(interval, employeeIndex, intervalIndex) {
16     this.interval = interval; // interval representing employee's working hours
17     // index of the list containing working hours of this employee
18     this.employeeIndex = employeeIndex;
19     this.intervalIndex = intervalIndex; // index of the interval in the employee list
20   }
21 }
22
23 function find_employee_free_time(schedule) {
24   let n = schedule.length,
25       result = [];
26   if (schedule === null || n === 0) {
27     return result;
28   }
```

RUN

SAVE

RESET

Close

4.862s

Output

Free intervals: [3, 5]
Free intervals: [4, 6][8, 9]
Free intervals: [5, 7]

Time complexity

The time complexity of the above algorithm is $O(N * \log K)$, where 'N' is the total number of intervals and 'K' is the total number of employees. This is due to the fact that we are iterating through the intervals only once (which will take $O(N)$), and every time we process an interval, we remove (and can insert) one interval in the **Min Heap**, (which will take $O(\log K)$). At any time the heap will not have more than 'K' elements.

Space complexity

The space complexity of the above algorithm will be $O(K)$ as at any time the heap will not have more than 'K' elements.

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

← Back

Problem Challenge 3

MARK AS COMPLETED

Next →

Introduction

Report an Issue