Grokking the Coding Interview: Patterns for Coding Questions

25% completed

Search Course

Pattern: Cyclic Sort

Pattern: In-place Reversal of a LinkedList

Pattern: Tree Breadth First Search

Pattern: Tree Depth First Search

Solution Review: Problem Challenge 3

We'll cover the following

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 employess 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 employess are free between [4,6] and [8,9].
```

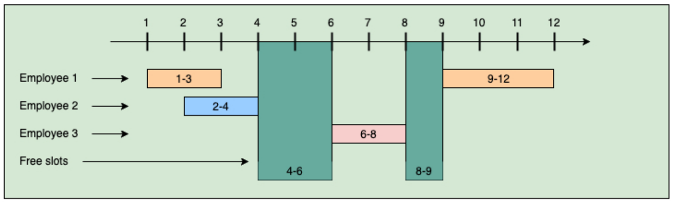
Example 3:

```
Input: Employee Working Hours=[[[1,3]], [[2,4]], [[3,5], [7,9]]]
Output: [5,7]
Explanation: All employess 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 $O(N * logN)$ 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.

MW
Explore
Tracks
My Courses
Edpresso
Refer a Friend
Create

Code

Here is what our algorithm will look like:

Java | Python3 | C++ | JS

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

class Interval {
  constructor(start, end) {
    this.start = start;
    this.end = end;
  }

  print_interval() {
    process.stdout.write(`[${this.start}, ${this.end}]`);
  }
}


class EmployeeInterval {
  constructor(interval, employeeIndex, intervalIndex) {
    this.interval = interval; // interval representing employee's working hours
    // index of the list containing working hours of this employee
    this.employeeIndex = employeeIndex;
    this.intervalIndex = intervalIndex; // index of the interval in the employee list
  }
}

function find_employee_free_time(schedule) {
  let n = schedule.length,
    result = [];
  if (schedule === null || n === 0) {
    return result;
```

RUN SAVE RESET

Close

Output 4.862s

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

Time complexity

The time complexity of the above algorithm is $O(N*logK)$, 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 $O(logK)$). 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.

☑ MARK AS COMPLETED

← Back
Problem Challenge 3

Next →
Introduction

Report an Issue