

Grokking the Coding Interview: Patterns for Coding Questions

20% completed

Intervals

- Introduction
- Merge Intervals (medium)
- Insert Interval (medium)
- Intervals Intersection (medium)
- Conflicting Appointments (medium)
- 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: 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)

Intervals Intersection (medium)

We'll cover the following

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

Problem Statement

Given two lists of intervals, find the **intersection of these two lists**. Each list consists of **disjoint intervals sorted on their start time**.

Example 1:

```
Input: arr1=[[1, 3], [5, 6], [7, 9]], arr2=[[2, 3], [5, 7]]
Output: [2, 3], [5, 6], [7, 7]
Explanation: The output list contains the common intervals between the two lists.
```

Example 2:

```
Input: arr1=[[1, 3], [5, 7], [9, 12]], arr2=[[5, 10]]
Output: [5, 7], [9, 10]
Explanation: The output list contains the common intervals between the two lists.
```

Try it yourself

Try solving this question here:

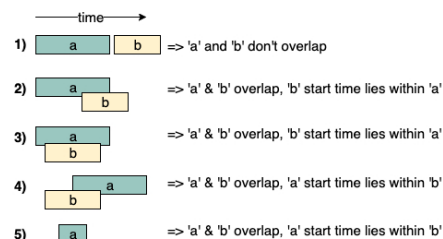
JavaPython3JS C++

```
4      this.end = end;
5    }
6
7    print_interval() {
8      process.stdout.write(`${this.start}, ${this.end}`);
9    }
10  }
11
12  const merge = function(intervals_a, intervals_b) {
13    let result = [];
14    // TODO: Write your code here
15    return result;
16  };
17
18  process.stdout.write('Intervals Intersection: ');
19  let result = merge([new Interval(1, 3), new Interval(5, 6), new Interval(7, 9)], [new Interval(2, 3), new Interval(5, 7)]);
20  for (i = 0; i < result.length; i++) {
21    result[i].print_interval();
22  }
23  console.log();
24
25  process.stdout.write('Intervals Intersection: ');
26  result = merge([new Interval(1, 3), new Interval(5, 7), new Interval(9, 12)], [new Interval(5, 10)]);
27  for (i = 0; i < result.length; i++) {
28    result[i].print_interval();
29  }
30  console.log();
```

RUNSAVERESET

Solution

This problem follows the [Merge Intervals](#) pattern. As we have discussed under [Insert Interval](#), there are five overlapping possibilities between two intervals 'a' and 'b'. A close observation will tell us that whenever the two intervals overlap, one of the interval's start time lies within the other interval. This rule can help us identify if any two intervals overlap or not.



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

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

MW

Explore

Tracks

My Courses

Edpresso

Refer a Friend

Create

b

Now, if we have found that the two intervals overlap, how can we find the overlapped part?

Again from the above diagram, the overlapping interval will be equal to:

start = max(a.start, b.start)

end = min(a.end, b.end)

That is, the highest start time and the lowest end time will be the overlapping interval.

So our algorithm will be to iterate through both the lists together to see if any two intervals overlap. If two intervals overlap, we will insert the overlapped part into a result list and move on to the next interval which is finishing early.

Code

Here is what our algorithm will look like:

JavaPython3C++JS JS

```
1  class Interval {
2  constructor(start, end) {
3      this.start = start;
4      this.end = end;
5  }
6
7  print_interval() {
8      process.stdout.write(`${this.start}, ${this.end}]\n`);
9  }
10
11
12  function merge(intervals_a, intervals_b) {
13      let result = [];
14      i = 0,
15      j = 0;
16
17      while (i < intervals_a.length && j < intervals_b.length) {
18          // check if intervals overlap and intervals_a[i]'s start time lies within the other intervals_b[j]
19          a_overlaps_b = intervals_a[i].start >= intervals_b[j].start && intervals_a[i].start <= intervals_b[j].end
20
21          // check if intervals overlap and intervals_b[j]'s start time lies within the other intervals_a[i]
22          b_overlaps_a = intervals_b[j].start >= intervals_a[i].start && intervals_b[j].start <= intervals_a[i].end
23
24          // store the the intersection part
25          if (a_overlaps_b || b_overlaps_a) {
26              result.push(new Interval(Math.max(intervals_a[i].start, intervals_b[j].start),
27                                     Math.min(intervals_a[i].end, intervals_b[j].end)));
28          }
29      }
30      return result;
31  }
```

RUNSAVERESET

Output

2.485s

Close

Intervals Intersection: [2, 3][5, 6][7, 7]

Intervals Intersection: [5, 7][9, 10]

Time complexity

As we are iterating through both the lists once, the time complexity of the above algorithm is $O(N + M)$, where 'N' and 'M' are the total number of intervals in the input arrays respectively.

Space complexity

Ignoring the space needed for the result list, the algorithm runs in constant space $O(1)$.

Interviewing soon? We've partnered with **Hired** so that companies apply to you instead of you applying to them. [See how](#)

← Back

Insert Interval (medium)

MARK AS COMPLETED

Next →

Conflicting Appointments (medium)

Report an Issue