

Grokking the Coding Interview: Patterns for Coding Questions

65% completed



- 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

Solution Review: Problem Challenge 3

We'll cover the following

- Count of Structurally Unique Binary Search Trees (hard)
- Solution
- Code
 - Time complexity
 - Space complexity
- Memoized version

Count of Structurally Unique Binary Search Trees (hard)

Given a number 'n', write a function to return the count of structurally unique Binary Search Trees (BST) that can store values 1 to 'n'.

Example 1:

```
Input: 2
Output: 2
Explanation: As we saw in the previous problem, there are 2 unique BSTs storing numbers from 1-2.
```

Example 2:

```
Input: 3
Output: 5
Explanation: There will be 5 unique BSTs that can store numbers from 1 to 5.
```

Solution

This problem is similar to [Structurally Unique Binary Search Trees](#). Following a similar approach, we can iterate from 1 to 'n' and consider each number as the root of a tree and make two recursive calls to count the number of left and right sub-trees.

Code

Here is what our algorithm will look like:

JavaPython3C++JS

```
1 function count_trees(n) {
2   if (n <= 1) {
3     return 1;
4   }
5   let count = 0;
6   for (let i = 1; i < n + 1; i++) {
7     // making 'i' the root of the tree
8     const countOfLeftSubtrees = count_trees(i - 1);
9     const countOfRightSubtrees = count_trees(n - i);
10    count += (countOfLeftSubtrees * countOfRightSubtrees);
11  }
12  return count;
13 }
14
15 console.log('Total trees: ${count_trees(2)}');
16 console.log('Total trees: ${count_trees(3)}');
```

RUNSAVERESET

Close

Output1.700s

Total trees: 2
Total trees: 5

Time complexity

The time complexity of this algorithm will be exponential and will be similar to [Balanced Parentheses](#). Estimated time complexity will be $O(n * 2^n)$ but the actual time complexity ($O(4^n / \sqrt{n})$) is bounded by the [Catalan number](#) and is beyond the scope of a coding interview. See more details [here](#).

Space complexity

The space complexity of this algorithm will be exponential too, estimated $O(2^n)$ but the actual will be ($O(4^n / \sqrt{n})$).

Memoized version

Our algorithm has overlapping subproblems as our recursive call will be evaluating the same sub-expression

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 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)

Explore

Tracks

My Courses

Edpresso

Refer a Friend

Create

Our algorithm has overlapping subproblems as our recursive call will be evaluating the same sub-expression multiple times. To resolve this, we can use memoization and store the intermediate results in a **HashMap**. In each function call, we can check our map to see if we have already evaluated this sub-expression before. Here is the memoized version of our algorithm, please see highlighted changes:

JavaPython3C++JS

```
1 class TreeNode {
2   constructor(val) {
3     this.val = val;
4     this.left = null;
5     this.right = null;
6   }
7 }
8
9
10 function count_trees(n) {
11   return count_trees_rec({}, n);
12 }
13
14
15 function count_trees_rec(map, n) {
16   if (n in map) {
17     return map[n];
18   }
19
20   if (n <= 1) {
21     return 1;
22   }
23
24   let count = 0;
25   for (let i = 1; i < n + 1; i++) {
26     // making 'i' the root of the tree
27     countOfLeftSubtrees = count_trees_rec(map, i - 1);
28     countOfRightSubtrees = count_trees_rec(map, n - i);
29     count += (countOfLeftSubtrees * countOfRightSubtrees);
30   }
31 }
```

RUNSAVERESET

Close

Output2.154s

Total trees: 2

Total trees: 5

The time complexity of the memoized algorithm will be $O(n^2)$, since we are iterating from '1' to 'n' and ensuring that each sub-problem is evaluated only once. The space complexity will be $O(n)$ for the memoization map.

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

BackNext

Problem Challenge 3Introduction

Mark as Completed

Report an IssueAsk a Question