

Grokking the Coding Interview: Patterns for Coding Questions

7% completed

Pattern: Two Pointers

- Introduction
- Pair with Target Sum (easy)
- Remove Duplicates (easy)
- Squaring a Sorted Array (easy)
- Triplet Sum to Zero (medium)
- Triplet Sum Close to Target (medium)
- Triplets with Smaller Sum (medium)
- Subarrays with Product Less than a Target (medium)
- Dutch National Flag Problem (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: Fast & Slow pointers

- Introduction
- LinkedList Cycle (easy)
- Start of LinkedList Cycle (medium)
- Happy Number (medium)
- Middle of the LinkedList (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: Merge 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

Pair with Target Sum (easy)

We'll cover the following

- Problem Statement
- Try it yourself
- Solution
 - Code
 - Time Complexity
 - Space Complexity
- An Alternate approach
 - Time Complexity
 - Space Complexity

Problem Statement

Given an array of sorted numbers and a target sum, find a **pair in the array whose sum is equal to the given target**.

Write a function to return the indices of the two numbers (i.e. the pair) such that they add up to the given target.

Example 1:

```
Input: [1, 2, 3, 4, 6], target=6
Output: [1, 3]
Explanation: The numbers at index 1 and 3 add up to 6: 2+4=6
```

Example 2:

```
Input: [2, 5, 9, 11], target=11
Output: [0, 2]
Explanation: The numbers at index 0 and 2 add up to 11: 2+9=11
```

Try it yourself

Try solving this question here:

Java Python3 JS C++

```
1 const pair_with_targetsum = function(arr, target_sum) {
2   // TODO: Write your code here
3   return [-1, -1];
4 }
5
```

TEST SAVE RESET

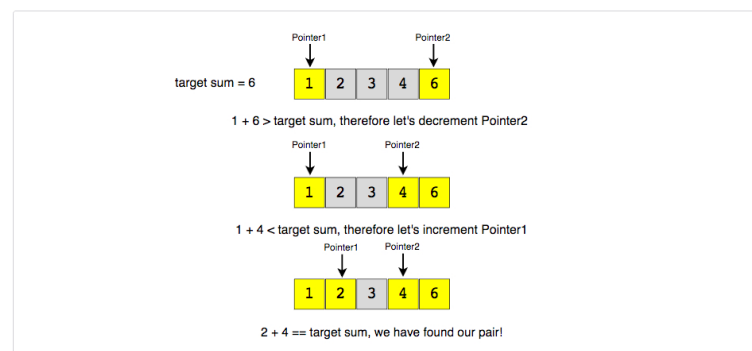
Solution

Since the given array is sorted, a brute-force solution could be to iterate through the array, taking one number at a time and searching for the second number through **Binary Search**. The time complexity of this algorithm will be $O(N * \log N)$. Can we do better than this?

We can follow the **Two Pointers** approach. We will start with one pointer pointing to the beginning of the array and another pointing at the end. At every step, we will see if the numbers pointed by the two pointers add up to the target sum. If they do, we have found our pair; otherwise, we will do one of two things:

1. If the sum of the two numbers pointed by the two pointers is greater than the target sum, this means that we need a pair with a smaller sum. So, to try more pairs, we can decrement the end-pointer.
2. If the sum of the two numbers pointed by the two pointers is smaller than the target sum, this means that we need a pair with a larger sum. So, to try more pairs, we can increment the start-pointer.

Here is the visual representation of this algorithm for Example-1:

**Code**

Here is what our algorithm will look like:

Java Python3 C++ JS

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

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)

+

Create

```
1 function pair_with_target_sum(arr, targetSum) {
2   let left = 0,
3     right = arr.length - 1;
4   while (left < right) {
5     const currentSum = arr[left] + arr[right];
6     if (currentSum === targetSum) {
7       return [left, right];
8     }
9
10    if (targetSum > currentSum) {
11      left += 1; // we need a pair with a bigger sum
12    } else {
13      right -= 1; // we need a pair with a smaller sum
14    }
15  }
16  return [-1, -1];
17 }
18
19 console.log(pair_with_target_sum([1, 2, 3, 4, 6], 6));
20 console.log(pair_with_target_sum([2, 5, 9, 11], 11));
```

Time Complexity

The time complexity of the above algorithm will be $O(N)$, where 'N' is the total number of elements in the given array.

Space Complexity

The algorithm runs in constant space $O(1)$.

An Alternate approach

Instead of using a two-pointer or a binary search approach, we can utilize a **HashTable** to search for the required pair. We can iterate through the array one number at a time. Let's say during our iteration we are at number 'X', so we need to find 'Y' such that " $X + Y == Target$ ". We will do two things here:

- Search for 'Y' (which is equivalent to " $Target - X$ ") in the **HashTable**. If it is there, we have found the required pair.
- Otherwise, insert "X" in the **HashTable**, so that we can search it for the later numbers.

Here is what our algorithm will look like:

Java Python3 C++ JS

```
1 function pair_with_target_sum(arr, targetSum) {
2   const nums = {}; // to store numbers and their indices
3   for (let i = 0; i < arr.length; i++) {
4     const num = arr[i];
5     if (targetSum - num in nums) {
6       return [nums[targetSum - num], i];
7     }
8     nums[arr[i]] = i;
9   }
10  return [-1, -1];
11 }
12
13 console.log(pair_with_target_sum([1, 2, 3, 4, 6], 6));
14 console.log(pair_with_target_sum([2, 5, 9, 11], 11));
```

Time Complexity

The time complexity of the above algorithm will be $O(N)$, where 'N' is the total number of elements in the given array.

Space Complexity

The space complexity will also be $O(N)$, as, in the worst case, we will be pushing 'N' numbers in the **HashTable**.

← Back

Introduction

✓ MARK AS COMPLETED

Next →

Remove Duplicates (easy)