Two Sum - Study Guide & Solution Analysis

Problem Overview

Given an array of integers and a target value, find two numbers that add up to the target and return their indices.

Solution Comparison

Approach 1: Brute Force (Nested Loops)

Lines 2-14 in 001-solution.ts

```
for (let i = 0; i < nums.length; i++) {
  for (let j = 0; j < nums.length; j++) {
    if (i === j) continue;
    if (nums[i] + nums[j] === target) {
      output.add(i);
      output.add(j);
    }
  }
}
```

How it Works:

- Checks every possible pair of numbers in the array
- Outer loop picks first number, inner loop picks second number
- Compares sum of each pair against target

Complexity Analysis:

- Time: $O(n^2)$ - nested loops iterate through array twice
- Space: $O(1)$ - only stores result (Set usage here is unnecessary)

Pros:

- Simple and intuitive
- Easy to understand and implement
- No extra data structures needed

Cons:

- Inefficient for large arrays
- Redundant comparisons (checks each pair twice)
- Using Set is overkill since problem guarantees single solution

---

Approach 2: Hash Map (Optimal Solution)

Lines 21-32 in 001-solution.ts

```
const numberMap = new Map();

for (let i = 0; i < nums.length; i++) {
  const number = nums[i];
  const difference = target - number;

  if (numberMap.has(difference)) return [numberMap.get(difference), i];

  numberMap.set(number, i);
}
```

How it Works:

- Single pass through the array
- For each number, calculates what value is needed to reach target (complement)
- Checks if complement exists in hash map
- If found: return both indices
- If not: store current number and index in map for future lookups

Complexity Analysis:

- Time: O(n) - single pass through array
- Space: O(n) - hash map can store up to n elements

Pros:

- Extremely efficient - linear time
- Single pass through data
- Clever use of complement concept

Cons:

- Uses extra memory for hash map

- Slightly less intuitive at first

Key Insight: The Complement Pattern

The breakthrough insight is thinking in **complements**:

```
If: a + b = target
Then: b = target - a
```

Instead of checking all pairs, we ask: "Have I already seen the complement of this number?"

Example Walkthrough: `nums = [2, 7, 11, 15]`, `target = 9`

```
i=0: number=2, diff=7, map={}, not found → map={2:0}
i=1: number=7, diff=2, map={2:0}, FOUND! → return [0,1]
```

Comparison Summary

Aspect	Brute Force	Hash Map
Time Complexity	$O(n^2)$	$O(n)$
Space Complexity	$O(1)$	$O(n)$
Passes	2 (nested)	1
Efficiency	Poor	Excellent
Readability	High	Medium
Best for	Small arrays, learning	Production, large datasets

Performance Difference:

- Array of 100 elements: Brute Force = 10,000 operations vs Hash Map = 100 operations
- Array of 10,000 elements: Brute Force = 100,000,000 vs Hash Map = 10,000

Core Concepts to Master

1. Hash Maps / Hash Tables

- **What:** Data structure for fast key-value lookups
- **Why:** O(1) average lookup time
- **In JavaScript/TypeScript:** `Map`, `Object`, or `Set`
- **Key insight:** Trade space for time

2. Time-Space Tradeoff

- Brute force: Low space, high time
- Hash map: Higher space, low time
- Understanding when each is appropriate

3. Complement/Difference Pattern

- Instead of searching for pairs, search for the "missing piece"
- Applicable to many sum-based problems
- Foundation for similar patterns (three sum, four sum)

4. Single-Pass Algorithms

- Process data in one iteration
- More efficient than nested loops
- Common optimization technique

Common Mistakes & Edge Cases

Mistakes in Your Solutions:

1. **Brute Force - Using Set Unnecessarily**

   ```
   // Unnecessary since problem guarantees one solution
   const output: Set<number> = new Set();
   ```

   Should be: `return [i, j]` directly when found

2. **Brute Force - Checking Same Pair Twice**

```
for (let j = 0; j < nums.length; j++)
```

Could optimize to: `for (let j = i + 1; j < nums.length; j++)`

Edge Cases to Consider:

- Array with duplicates: `[3, 3]` target `6` ✓ (Your solution handles this)
- Negative numbers: `[-1, -2, -3, -4]` target `-5`
- Zero in array: `[0, 4, 3, 0]` target `0`
- Minimum array size (2 elements)

Learning Path

Phase 1: Foundation (You are here!)

- ✓ Understand the problem
- ✓ Implement brute force
- ✓ Implement hash map solution
- → Analyze time/space complexity

Practice:

1. Implement both solutions from scratch without looking
2. Explain the solutions to someone else (rubber duck debugging)
3. Write test cases for edge cases

Phase 2: Pattern Recognition

Related Problems:

- **Two Sum II** (sorted array) - Learn two-pointer technique
- **Two Sum III** (data structure design) - Design patterns
- **3Sum** - Extends the concept to three numbers
- **4Sum** - Further extension
- **Valid Anagram** - Similar hash map pattern
- **Contains Duplicate** - Hash set usage

Phase 3: Advanced Variations

- Two Sum with different constraints (BST, sorted array)
- All pairs that sum to target (not just one)
- Two Sum with multiple targets
- Closest two sum to target

Resources to Learn More

Understanding Hash Maps

- **MDN Web Docs:** JavaScript Map object
- **Book:** "Cracking the Coding Interview" - Chapter 1 (Hash Tables)
- **Video:** CS50's explanation of hash tables

Algorithm Analysis

- **Book:** "Introduction to Algorithms" (CLRS) - Chapter 11
- **Course:** MIT 6.006 Introduction to Algorithms (free on YouTube)
- **Interactive:** VisuAlgo.net - Hash Table visualizations

Practice Platforms

1. **LeetCode** - 100+ problems tagged "Hash Table"
2. **NeetCode** - Curated list with video explanations
3. **AlgoExpert** - Structured learning path
4. **HackerRank** - Interview preparation kit

Similar Pattern Problems (Easy Difficulty)

- Contains Duplicate (LeetCode 217)
- Valid Anagram (LeetCode 242)
- Majority Element (LeetCode 169)
- Single Number (LeetCode 136)
- Happy Number (LeetCode 202)

Practice Exercises

Exercise 1: Variations

Modify your solution to:

1. Return all pairs (not just one) that sum to target
2. Handle the case where no solution exists
3. Use a regular object `{}` instead of `Map`

Exercise 2: Optimization

Can you improve the brute force to O(n^2) but with fewer iterations? (Hint: start second loop from `i + 1`)

Exercise 3: Different Data Structure

Solve using:

1. Sorting + two pointers (O(n log n) time, O(1) space)
2. Set instead of Map

Exercise 4: Testing

Write comprehensive tests covering:

- Normal cases
- Edge cases (duplicates, negatives, zeros)
- Boundary cases (minimum size array)

Key Takeaways

1. **Multiple solutions exist** - Start simple, then optimize
2. **Hash maps are powerful** - O(1) lookups enable O(n) solutions
3. **Think in complements** - Pattern applies to many problems
4. **Space-time tradeoff** - Sometimes using more space saves time
5. **Real-world applications:**
    - Finding matching transactions in financial systems

- Detecting duplicate entries
- Caching and memoization
- Database indexing

Next Steps

1. Implement both solutions from memory
2. Run your solutions with test cases: `ts-node 001-solution.ts`
3. Try "Two Sum II - Input Array Is Sorted" (different approach needed)
4. Learn the two-pointer technique (complements hash map approach)
5. Move on to 3Sum problem (builds on this pattern)

Remember: Understanding why the hash map solution works is more important than memorizing the code. Focus on the complement pattern - it appears in many algorithms.