Algorithm Design and Analysis

Mihail Iazinschi

Project Repository:

https://github.com/Misoding/LeetCode-Journey

Contact:

GitHub Profile · LinkedIn Profile

January 17, 2026

Contents

Introduction

Hi,

My name is **Mihail Iazinschi** and I am a second-year student at the Faculty of Automatic Control and Computer Science, within the Polytechnic University of Bucharest.

Over the years, programming has been more than just code to me: it was the fascination of "translating" a real-world problem into a digital format. I always found it incredible that, by writing the correct instructions, you can make a computer solve problems for you. This curiosity pushed me, during high school, towards Informatics Olympiads and platforms like LeetCode, CodeWars, or Codeforces. There, I learned that any complex problem, if broken down into small pieces, yields to the right algorithm.

But, I must be honest with you (and with myself). Once I arrived at university, caught up in the complexity of courses and diverse projects, I started working less and less on pure algorithmic problems. And, as we all know, engineering is like a sport: if you don't train constantly, you start forgetting the nuances.

However, I realized that algorithms are not just a subject to simply check off. The ability to analyze an algorithm is crucial for optimizing real tasks within a project. Moreover, it is about recognizing that "pattern" — knowing instinctively when a problem reduces to a graph, a stack, or dynamic programming. And yes, let's not forget the fact that: no matter how good a specialist you are, any serious interview will pass through the inevitable algorithm test.

Therefore, I decided to start this repository and this "book" for the following reason, as this is my method of self-discipline: **the objective is to upload and explain at least one problem every day.**

By writing this material, I aim for two things: to recover and polish my knowledge, but also to leave behind a public record of my progress, which I hope will help you as well.

1 1. Two Sum

1.1 Problem Statement

Given an array of integers $A = [a_0, a_1, \ldots, a_{n-1}]$ and an integer target T, find two indices i and j such that:

$$a_i + a_j = T$$

subject to the constraint $i \neq j$.

Assumptions:

- Exactly one valid solution exists.
- The same element cannot be used twice (indices must be distinct).
- The order of the returned indices does not matter.

1.2 Theoretical Approach

Naïve Approach (Brute Force)

The rudimentary method involves iterating through all unique pairs (i, j) with $0 \leq i < j \leq n-1$ and checking the condition $a_i + a_j = T$.

Implementation:

```cpp
vector<int> twoSum(vector<int>& nums, int target) {
    vector<int> solution(2);
    int n = nums.size();
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++) {
            if (nums[i] + nums[j] == target) {
                solution[0] = i;
                solution[1] = j;
            }
        }
    }
    return solution;
}
```

Listing 1: Brute Force Approach

Mathematical Derivation of Complexity: To determine the exact number of operations, we calculate how many times the inner body (the `if` statement) is executed. The outer loop runs for i from 0 to $n-2$. The inner loop runs for j from $i+1$ to $n-1$. The total number of steps $T(n)$ is the sum of these iterations:

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

For a fixed i, the inner loop runs $(n-1) - (i+1) + 1 = n - 1 - i$ times.

$$T(n) = \sum_{i=0}^{n-2} (n - 1 - i)$$

Expanding the sum for $i = 0, 1, \ldots, n-2$:

$$T(n) = (n-1) + (n-2) + \cdots + 1$$

This is the sum of the first $n-1$ integers (Arithmetic Progression). Using Gauss's formula $S_k = \frac{k(k+1)}{2}$ where $k = n-1$:

$$T(n) = \frac{(n-1)(n-1+1)}{2} = \frac{n(n-1)}{2}$$

Since the dominant term is n^2:

$$T(n) \in \Theta(n^2)$$

Optimized Approach (Hash Map)

To reduce the time complexity, we must lower the query time for the complement value. We utilize a **Hash Map** (Dictionary) to trade space for time.

Let c_i be the complement of a_i such that $c_i = T - a_i$. The problem reduces to finding if c_i exists in the array at an index $j \neq i$.

The algorithm proceeds in two logical phases (Two-pass Hash Table):

1. **Mapping Phase:** Construct a lookup table mapping each value a_k to its index k.
2. **Search Phase:** For each element a_i, calculate c_i and check the table for existence.

Correctness and Loop Invariant

We define the loop invariant for the mapping phase. Let M be the hash map. At the start of the k-th iteration ($0 \leq k < n$), the map M contains pairs (a_x, x) for all $0 \leq x < k$.

- **Initialization:** For $k = 0$, M is empty (vacuously true).
- **Maintenance:** In step k, we insert (a_k, k). Thus, at $k+1$, the property holds.
- **Termination:** When $k = n$, M contains all elements.

Since the problem guarantees a solution, the search phase is guaranteed to find the complement c_i in M if a_i is part of the solution pair.

1.3 Complexity Analysis (Optimized Solution)

Let n be the number of elements in the input vector `nums`.

Time Complexity

The algorithm executes two distinct linear traversals. We assume the Average Case for the Hash Map operations.

1. **Phase 1 (Build):** The loop runs n times. Inside the loop, the insertion into the `unordered_map` takes $O(1)$ on average.

$$T_{build}(n) = \sum_{i=0}^{n-1} O(1) = n \cdot O(1) = O(n)$$

2. **Phase 2 (Search):** The second loop also runs n times. The lookup operation `initialNumbers.conta` and arithmetic operations take $O(1)$.

$$T_{search}(n) = \sum_{i=0}^{n-1} O(1) = n \cdot O(1) = O(n)$$

The total time complexity is the sum of both phases:

$$T(n) = T_{build}(n) + T_{search}(n) \approx 2n$$

$$T(n) \in \Theta(n)$$

Space Complexity

We utilize an auxiliary hash map to store the indices of the elements. In the worst case (all elements are distinct), we store n entries.

$$S(n) \in \Theta(n)$$

1.4 Implementation

The following C++ implementation utilizes the `std::unordered_map` to achieve linear time complexity.

```cpp
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        vector<int> solution(2);
        unordered_map<int,int> initialNumbers;
        int n = nums.size();
        int mapIndex;

        // Phase 1: Build the Hash Map
        for (int i = 0; i < n; i++) {
            initialNumbers[nums[i]] = i;
        }

        // Phase 2: Search for the complement
        for (int i = 0; i < n; i++) {
            nums[i] = target-nums[i];

            if (initialNumbers.contains(nums[i])) {
                mapIndex = initialNumbers[nums[i]];
                if (i != mapIndex){
                    solution[0] = i;
                    solution[1] = mapIndex;
                }
            }
        }
        return solution;
    }
};
```

Listing 2: Two Sum Solution (Two-Pass Hash Table)