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:

```
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.cont` 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)

2 9. Palindrome Number

2.1 Problem Statement

Given an integer x, return `true` if x is a palindrome, and `false` otherwise.

Definition: An integer is a palindrome when it reads the same backward as forward. For example, 121 is a palindrome while 123 is not.

Constraints:

- $-2^{31} \leq x \leq 2^{31} - 1$
- The algorithm should ideally avoid converting the integer to a string to optimize space complexity.

2.2 Theoretical Approach

Naïve Approach (String Conversion)

The trivial solution involves converting the integer x into a string representation S and checking if S is equal to its reverse, S_{rev}. While simple, this requires allocating auxiliary memory proportional to the number of digits in x, i.e., Space Complexity $S(n) \in O(\log_{10} n)$.

Optimized Approach (Integral Reversal)

To achieve $O(1)$ space complexity, we reverse the second half of the number mathematically using modulo and division operations. However, a simpler variation (implemented below) constructs the fully reversed number R and compares it with the initial input x.

Edge Cases:

- **Negative Numbers:** Any $x < 0$ (e.g., -121) reads as $121-$ when reversed. Thus, negative numbers are never palindromes.
- **Overflow Risk:** Reversing a large integer (e.g., $2 \cdot 10^9$) might exceed the 32-bit signed integer limit. We use `long` for the reversed variable to prevent overflow.

Mathematical Model of Reversal

Let x_0 be the initial number. In each iteration k, we extract the last digit d_k and append it to the reversed number R. The recurrence relations for the state variables at step k are:

$$d_k = x_{k-1} \pmod{10}$$

$$R_k = R_{k-1} \cdot 10 + d_k$$

$$x_k = \lfloor x_{k-1}/10 \rfloor$$

The process terminates when $x_k = 0$.

2.3 Complexity Analysis

Let n be the value of the input integer x.

Time Complexity

The algorithm processes the number digit by digit. The loop continues as long as $x > 0$. The number of digits D in a positive integer n is given by the logarithmic formula:

$$D = \lfloor \log_{10}(n) \rfloor + 1$$

The total time complexity $T(n)$ is the sum of operations performed for each digit:

$$T(n) = \sum_{k=1}^{D} c_{ops}$$

Substituting D:

$$T(n) = c \cdot (\lfloor \log_{10}(n) \rfloor + 1)$$

Since logarithms in different bases are related by a constant factor ($\log_{10} n = \frac{\ln n}{\ln 10}$), we conclude:

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

Space Complexity

We utilize a fixed number of variables (`reversedInt`, `tmpNum`, `initialNumber`) regardless of the input size. No dynamic structures (arrays, strings) are allocated.

$$S(n) \in \Theta(1)$$

2.4 Implementation

The implementation uses a `long` type for the reversed integer to safely handle potential overflows during the reversal process, although input constraints suggest x fits in `int`.

```
class Solution {
public:
    bool isPalindrome(int x) {
        if (x < 0) {
            return false;
        }
        long reversedInt = 0;
        int tmpNum = 0, initialNumber = x;

        while (x) {
            reversedInt *= 10;
            tmpNum = x % 10;
            reversedInt += tmpNum;
            x /= 10;
        }

        if ((long) initialNumber == reversedInt) {
            return true;
        }
        return false;
    }
};
```

Listing 3: Palindrome Number (Mathematical Reversal)

3 3. Find the Largest Area of Square Inside Two Rectangles

3.1 Problem Statement

Given n rectangles in a 2D plane, defined by two 2D integer arrays `bottomLeft` and `topRight`, select a region formed by the intersection of exactly two rectangles. Find the largest area of a square that can fit inside this intersection region.

Input Format:

- `bottomLeft[i]` and `topRight[i]` represent the coordinates of the i-th rectangle.
- The goal is to maximize S^2, where S is the side of the inscribed square within the intersection $R_i \cap R_j$.

Problem Link: [1]

3.2 Theoretical Approach

Mathematical Abstraction (1D Projection)

The problem of finding the intersection of two rectangles in 2D can be decomposed into two independent 1D interval intersection problems. Let a rectangle R_k be defined as the Cartesian product of intervals on the X and Y axes:

$$R_k = [x^{(k)}_{start}, x^{(k)}_{end}] \times [y^{(k)}_{start}, y^{(k)}_{end}]$$

The intersection of two rectangles R_i and R_j is valid if and only if their intervals overlap on **both** axes simultaneously. The boundaries of the intersection $R_{overlap}$ are derived as:

$$x_{overlap\_start} = \max(x^{(i)}_{start}, x^{(j)}_{start})$$

$$x_{overlap\_end} = \min(x^{(i)}_{end}, x^{(j)}_{end})$$

The dimensions of the intersection rectangle are:

$$\Delta x = x_{overlap\_end} - x_{overlap\_start}$$

$$\Delta y = y_{overlap\_end} - y_{overlap\_start}$$

Optimization Function

For a valid intersection, we require $\Delta x > 0$ and $\Delta y > 0$. The side S of the largest square that fits inside a rectangle of size $\Delta x \times \Delta y$ is constrained by the smaller dimension:

$$S = \min(\Delta x, \Delta y)$$

The objective is to find:

$$\text{Result} = \max_{i<j}(S^2_{ij})$$

3.3 Complexity Analysis

Let n be the number of rectangles in the input arrays.

[1] https://leetcode.com/problems/find-the-largest-area-of-square-inside-two-rectangles/

Time Complexity

The algorithm employs a brute-force strategy to evaluate every unique pair of rectangles (i, j) with $0 \le i < j \le n-1$. To determine the exact number of operations, we sum the iterations of the inner loop:

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

Expanding the arithmetic progression:

$$T(n) = C \cdot [(n-1) + (n-2) + \cdots + 1] = C \cdot \frac{n(n-1)}{2}$$

Since the dominant term is n^2, the time complexity is quadratic:

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

Space Complexity

The solution operates using a fixed set of scalar variables (coordinates and boundaries) and does not allocate any auxiliary data structures proportional to the input size.

$$S(n) \in \Theta(1)$$

3.4 Implementation

The following C++ implementation iterates through all pairs, computes the intersection overlaps, and updates the maximum square side found.

```cpp
class Solution {
public:
    long long largestSquareArea(vector<vector<int>>& bottomLeft, vector<vector<int>>& topRight) {
        int n = bottomLeft.size();
        int i_rectangle_x_start, i_rectangle_x_finish, j_rectangle_x_start, j_rectangle_x_finish;
        int i_rectangle_y_start, i_rectangle_y_finish, j_rectangle_y_start, j_rectangle_y_finish;
        int max_overlap_x_start, min_overlap_x_finish, max_overlap_y_start, min_overlap_y_finish;
        long long finalResult = 0;

        for (int i = 0; i < n; i++) {
            i_rectangle_x_start = bottomLeft[i][0];
            i_rectangle_x_finish = topRight[i][0];
            i_rectangle_y_start = bottomLeft[i][1];
            i_rectangle_y_finish = topRight[i][1];

            for (int j = i + 1; j < n; j++) {
                j_rectangle_x_start = bottomLeft[j][0];
                j_rectangle_x_finish = topRight[j][0];
                j_rectangle_y_start = bottomLeft[j][1];
                j_rectangle_y_finish = topRight[j][1];

```

```
                max_overlap_x_start = max(i_rectangle_x_start, j_rectangle_x_start);
                min_overlap_x_finish = min(i_rectangle_x_finish, j_rectangle_x_finish);

                max_overlap_y_start = max(i_rectangle_y_start, j_rectangle_y_start);
                min_overlap_y_finish = min(i_rectangle_y_finish, j_rectangle_y_finish);

                int squareSide = min ((min_overlap_x_finish - max_overlap_x_start),
                                      (min_overlap_y_finish - max_overlap_y_start));

                if (squareSide <= 0)
                    continue;
                finalResult = max((long long) (squareSide), finalResult);
            }
        }
        return finalResult * finalResult;
    }
};
```

Listing 4: Largest Square Area Solution