

Algorithm Design and Analysis

Mihail Iazinschi

Project Repository:

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

Contact:

[GitHub Profile](#) · [LinkedIn Profile](#)

January 18, 2026

Contents

1	1. Two Sum	3
1.1	Problem Statement	3
1.2	Theoretical Approach	3
1.3	Complexity Analysis (Optimized Solution)	4
1.4	Implementation	5
2	9. Palindrome Number	5
2.1	Problem Statement	5
2.2	Theoretical Approach	6
2.3	Complexity Analysis	6
2.4	Implementation	7
3	3. Find the Largest Area of Square Inside Two Rectangles	8
3.1	Problem Statement	8
3.2	Theoretical Approach	8
3.3	Complexity Analysis	8
3.4	Implementation	9
4	4. Largest Magic Square	10
4.1	Problem Statement	10
4.2	Theoretical Approach	10
4.3	Complexity Analysis	11
4.4	Implementation	12

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. Two Sum

1.1 Problem Statement

Given an array of integers $A = [a_0, a_1, \dots, a_{n-1}]$ 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:

```
1 vector<int> twoSum(vector<int>& nums, int target) {
2     vector<int> solution(2);
3     int n = nums.size();
4     for (int i = 0; i < n; i++) {
5         for (int j = i + 1; j < n; j++) {
6             if (nums[i] + nums[j] == target) {
7                 solution[0] = i;
8                 solution[1] = j;
9             }
10        }
11    }
12    return solution;
13 }
```

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 $i = 0, 1, \dots, n-2$:

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

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.contains` 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.

```
1 class Solution {
2 public:
3     vector<int> twoSum(vector<int>& nums, int target) {
4         vector<int> solution(2);
5         unordered_map<int,int> initialNumbers;
6         int n = nums.size();
7         int mapIndex;
8
9         // Phase 1: Build the Hash Map
10        for (int i = 0; i < n; i++) {
11            initialNumbers[nums[i]] = i;
12        }
13
14        // Phase 2: Search for the complement
15        for (int i = 0; i < n; i++) {
16            nums[i] = target - nums[i];
17
18            if (initialNumbers.contains(nums[i])) {
19                mapIndex = initialNumbers[nums[i]];
20                if (i != mapIndex){
21                    solution[0] = i;
22                    solution[1] = mapIndex;
23                }
24            }
25        }
26        return solution;
27    }
28};
```

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`.

```
1 class Solution {
2 public:
3     bool isPalindrome(int x) {
4         if (x < 0) {
5             return false;
6         }
7         long reversedInt = 0;
8         int tmpNum = 0, initialNumber = x;
9
10        while (x) {
11            reversedInt *= 10;
12            tmpNum = x % 10;
13            reversedInt += tmpNum;
14            x /= 10;
15        }
16
17        if ((long) initialNumber == reversedInt) {
18            return true;
19        }
20        return false;
21    }
22};
```

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:

$$R_k = [x_{start}^{(k)}, x_{end}^{(k)}] \times [y_{start}^{(k)}, y_{end}^{(k)}]$$

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:

$$x_{overlap_start} = \max(x_{start}^{(i)}, x_{start}^{(j)})$$

$$x_{overlap_end} = \min(x_{end}^{(i)}, x_{end}^{(j)})$$

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:

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

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 $0 \leq i < j \leq n - 1$. 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.

```
1 class Solution {
2 public:
3     long long largestSquareArea(vector<vector<int>>& bottomLeft,
4     vector<vector<int>>& topRight) {
5         int n = bottomLeft.size();
6         int i_rectangle_x_start, i_rectangle_x_finish,
7         j_rectangle_x_start, j_rectangle_x_finish;
8         int i_rectangle_y_start, i_rectangle_y_finish,
9         j_rectangle_y_start, j_rectangle_y_finish;
10        int max_overlap_x_start, min_overlap_x_finish,
11        max_overlap_y_start, min_overlap_y_finish;
12        long long finalResult = 0;
13
14        for (int i = 0; i < n; i++) {
15            i_rectangle_x_start = bottomLeft[i][0];
16            i_rectangle_x_finish = topRight[i][0];
17            i_rectangle_y_start = bottomLeft[i][1];
18            i_rectangle_y_finish = topRight[i][1];
19
20            for (int j = i + 1; j < n; j++) {
21                j_rectangle_x_start = bottomLeft[j][0];
22                j_rectangle_x_finish = topRight[j][0];
23                j_rectangle_y_start = bottomLeft[j][1];
24                j_rectangle_y_finish = topRight[j][1];
```

```

22         max_overlap_x_start = max(i_rectangle_x_start ,
j_rectangle_x_start);
23         min_overlap_x_finish = min(i_rectangle_x_finish ,
j_rectangle_x_finish);
24
25         max_overlap_y_start = max(i_rectangle_y_start ,
j_rectangle_y_start);
26         min_overlap_y_finish = min(i_rectangle_y_finish ,
j_rectangle_y_finish);
27
28         int squareSide = min ((min_overlap_x_finish -
max_overlap_x_start) ,
29                               (min_overlap_y_finish -
max_overlap_y_start));
30
31         if (squareSide <= 0)
32             continue;
33         finalResult = max((long long) (squareSide),
finalResult);
34     }
35 }
36 return finalResult * finalResult;
37 }
38 };

```

Listing 4: Largest Square Area Solution

4. Largest Magic Square

4.1 Problem Statement

Given an $m \times n$ integer matrix **grid**, a **magic square** is defined as a $k \times k$ subgrid ($1 \leq k \leq \min(m, n)$) such that:

- The sum of elements in each row is equal.
- The sum of elements in each column is equal.
- The sum of elements in both the principal and secondary diagonals is equal.
- All these sums share the same common value.

The objective is to find the **largest possible value of k** (the side length) of such a subgrid.

Problem Link: ²

4.2 Theoretical Approach

Mathematical Abstraction (2D Prefix Sums)

A brute-force approach recalculating sums for every subgrid would be inefficient. To optimize range sum queries from $O(k)$ to $O(1)$, we employ the **Prefix Sum** technique extended to four directions.

Let G be the input matrix. We define four auxiliary matrices:

²<https://leetcode.com/problems/largest-magic-square/>

1. **Row Prefix Sum** (P_{row}): Stores cumulative sums along rows.

$$P_{row}[i][j] = \sum_{c=0}^j G[i][c]$$

2. **Column Prefix Sum** (P_{col}): Stores cumulative sums along columns.

$$P_{col}[i][j] = \sum_{r=0}^i G[r][j]$$

3. **Principal Diagonal** (P_{diag}): Stores sums along the main diagonal direction $(i-1, j-1)$.

$$P_{diag}[i][j] = G[i][j] + P_{diag}[i-1][j-1]$$

4. **Secondary Diagonal** (P_{anti}): Stores sums along the anti-diagonal direction $(i-1, j+1)$.

$$P_{anti}[i][j] = G[i][j] + P_{anti}[i-1][j+1]$$

With these structures precomputed, the sum of any row, column, or diagonal segment of length k can be retrieved in constant time $O(1)$ using the difference between two prefix values.

Search Strategy (Greedy)

We adopt a Greedy strategy for the dimension k . We iterate k from the maximum possible size $(\min(m, n))$ down to 1. For a fixed k , we slide a window (i, j) across the grid. The first valid magic square found guarantees that the current k is the global maximum, allowing an early exit.

4.3 Complexity Analysis

Let m be the number of rows and n be the number of columns. Let $K = \min(m, n)$.

Time Complexity

1. **Preprocessing:** Computing the four prefix sum matrices requires iterating through the grid once.

$$T_{pre} \in \Theta(m \cdot n)$$

2. **Search Phase:** For each size k , we iterate through $(m-k)(n-k)$ possible top-left positions. For each position, we verify k rows and k columns. The cost function is:

$$T_{search} \approx \sum_{k=1}^K (m-k)(n-k) \cdot 2k$$

Approximating the sum with an integral for asymptotic analysis (assuming $m \approx n \approx N$):

$$T(N) \approx \int_1^N (N-x)^2 \cdot 2x \, dx \approx O(N^4)$$

Thus, in the general case:

$$T(m, n) \in O(m \cdot n \cdot \min(m, n)^2)$$

Space Complexity

We allocate four auxiliary matrices of size $m \times n$ to store the cumulative sums.

$$S(m, n) = 4 \cdot (m \cdot n) \in \Theta(m \cdot n)$$

4.4 Implementation

```
1 class Solution {
2 public:
3     void calcRowPrefixSum(vector<vector<int>>& originalGrid,
4                           vector<vector<int>>& row_grid,
5                           int& m,
6                           int& n) {
7         for (int i = 0; i < m; i++) {
8             for (int j = 0; j < n; j++) {
9                 if (j == 0) {
10                    row_grid[i][j] = originalGrid[i][j];
11                    continue;
12                }
13                row_grid[i][j] = row_grid[i][j-1] + originalGrid[i][j]
14            };
15        }
16    }
17
18    void calcColumnPrefixSum(vector<vector<int>>& originalGrid,
19                             vector<vector<int>>& column_grid,
20                             int& m,
21                             int& n) {
22        for (int i = 0; i < n; i++) {
23            for (int j = 0; j < m; j++) {
24                if (j == 0) {
25                    column_grid[j][i] = originalGrid[j][i];
26                    continue;
27                }
28                column_grid[j][i] = column_grid[j-1][i] + originalGrid
29                [j][i];
30            }
31        }
32
33        void calcPrincipalDiagonalSum(vector<vector<int>>& originalGrid,
34                                       vector<vector<int>>&
35                                       principalDiagonal,
36                                       int& m,
37                                       int& n) {
38            for (int i = 0; i < m; i++) {
39                for (int j = 0; j < n; j++) {
40                    if (i == 0 || j == 0) {
41                        principalDiagonal[i][j] = originalGrid[i][j];
42                        continue;
43                    }
44                }
45            }
46        }
47    }
48 }
```

```

43         principalDiagonal[i][j] = principalDiagonal[i-1][j-1]
+ originalGrid[i][j];
44     }
45 }
46 }
47
48 void calcSecondaryDiagonalSum(vector<vector<int>>& originalGrid,
49                               vector<vector<int>>&
secondaryDiagonal,
50                               int& m,
51                               int& n) {
52     for (int i = 0; i < m; i++) {
53         for (int j = n-1; j >= 0; j--) {
54             if (j == n-1 || i == 0) {
55                 secondaryDiagonal[i][j] = originalGrid[i][j];
56                 continue;
57             }
58             secondaryDiagonal[i][j] = secondaryDiagonal[i-1][j+1]
+ originalGrid[i][j];
59         }
60     }
61 }
62
63 int largestMagicSquare(vector<vector<int>>& grid) {
64     int m = grid.size();
65     int n = grid[0].size();
66     int squareDimension = min(m,n);
67     int oneSum, prevRowSum, prevColSum;
68     bool validSquare;
69
70     vector<vector<int>> rowPrefixSum(m, vector<int>(n));
71     vector<vector<int>> columnPrefixSum(m, vector<int>(n));
72     vector<vector<int>> principalDiagonalSum(m, vector<int>(n));
73     vector<vector<int>> secondaryDiagonal(m, vector<int>(n));
74
75     calcRowPrefixSum(grid, rowPrefixSum, m, n);
76     calcColumnPrefixSum(grid, columnPrefixSum, m, n);
77     calcPrincipalDiagonalSum(grid, principalDiagonalSum, m, n);
78     calcSecondaryDiagonalSum(grid, secondaryDiagonal, m, n);
79
80     for (int edge = squareDimension; edge > 0; edge--) {
81         for (int i = 0; i <= m - edge; i++) {
82             for (int j = 0; j <= n - edge; j++) {
83                 validSquare = true;
84                 prevRowSum = 0;
85                 if (j > 0) {
86                     prevRowSum = rowPrefixSum[i][j-1];
87                 }
88                 oneSum = rowPrefixSum[i][j + edge-1] - prevRowSum;
89                 for (int sqRow = 0; sqRow < edge; sqRow++) {
90                     prevRowSum = 0;
91                     if (j > 0) {
92                         prevRowSum = rowPrefixSum[i + sqRow][j-1];

```

```

123         }
124         if (rowPrefixSum[i + sqRow][j + edge - 1] -
125 prevRowSum != oneSum) {
126             validSquare = false;
127             break;
128         }
129     }
130     if (!validSquare) continue;
131     for (int sqCol = 0; sqCol < edge; sqCol++) {
132         prevColSum = 0;
133         if (i > 0) {
134             prevColSum = columnPrefixSum[i-1][j +
135 sqCol];
136         }
137         if (columnPrefixSum[i + edge - 1][j + sqCol] -
138 prevColSum != oneSum) {
139             validSquare = false;
140             break;
141         }
142     }
143     if (!validSquare) continue;
144     int prevPrincipalSum = 0, prevSecondarySum = 0;
145     if (i != 0 && j != 0) {
146         prevPrincipalSum = principalDiagonalSum[i-1][j
147 -1];
148     }
149     if (i != 0 && (j + edge) < n) {
150         prevSecondarySum = secondaryDiagonal[i-1][j+
151 edge];
152     }
153     if (principalDiagonalSum[i + edge-1][j + edge-1] -
154 prevPrincipalSum != oneSum ||
155         secondaryDiagonal[i + edge-1][j] -
156 prevSecondarySum != oneSum ) {
157         continue;
158     }
159     return edge;
160 }
161 }
162 return 1;
163 }
164 };

```

Listing 5: Largest Magic Square Solution