Algorithm Design and Analysis

Mihail Iazinschi

Project Repository:

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

Contact:

GitHub Profile · LinkedIn Profile

January 21, 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.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)

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

```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

4 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

```cpp
class Solution {
public:
    void calcRowPrefixSum(vector<vector<int>>& originalGrid,
                          vector<vector<int>>& row_grid,
                          int& m,
                          int& n) {
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (j == 0) {
                    row_grid[i][j] = originalGrid[i][j];
                    continue;
                }
                row_grid[i][j] = row_grid[i][j-1] + originalGrid[i][j];
            }
        }
    }

    void calcColumnPrefixSum(vector<vector<int>>& originalGrid,
                             vector<vector<int>>& column_grid,
                             int& m,
                             int& n) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                if (j == 0) {
                    column_grid[j][i] = originalGrid[j][i];
                    continue;
                }
                column_grid[j][i] = column_grid[j-1][i] + originalGrid[j][i];
            }
        }
    }

    void calcPrincipalDiagonalSum(vector<vector<int>>& originalGrid,
                                  vector<vector<int>>& principalDiagonal,
                                  int& m,
                                  int& n) {
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (i == 0 || j == 0) {
                    principalDiagonal[i][j] = originalGrid[i][j];
                    continue;
                }
```

```cpp
                principalDiagonal[i][j] = principalDiagonal[i-1][j-1] + originalGrid[i][j];
            }
        }
    }

    void calcSecondaryDiagonalSum(vector<vector<int>>& originalGrid,
                                  vector<vector<int>>& secondaryDiagonal,
                                  int& m,
                                  int& n) {
        for (int i = 0; i < m; i++) {
            for (int j = n-1; j >= 0; j--) {
                if (j == n-1 || i == 0) {
                    secondaryDiagonal[i][j] = originalGrid[i][j];
                    continue;
                }
                secondaryDiagonal[i][j] = secondaryDiagonal[i-1][j+1] + originalGrid[i][j];
            }
        }
    }

    int largestMagicSquare(vector<vector<int>>& grid) {
        int m = grid.size();
        int n = grid[0].size();
        int squareDimension = min(m,n);
        int oneSum, prevRowSum, prevColSum;
        bool validSquare;

        vector<vector<int>> rowPrefixSum(m, vector<int>(n));
        vector<vector<int>> columnPrefixSum(m, vector<int>(n));
        vector<vector<int>> principalDiagonalSum(m, vector<int>(n));
        vector<vector<int>> secondaryDiagonal(m, vector<int>(n));

        calcRowPrefixSum(grid, rowPrefixSum, m, n);
        calcColumnPrefixSum(grid, columnPrefixSum, m, n);
        calcPrincipalDiagonalSum(grid, principalDiagonalSum, m, n);
        calcSecondaryDiagonalSum(grid, secondaryDiagonal, m, n);

        for (int edge = squareDimension; edge > 0; edge--) {
            for (int i = 0; i <= m - edge; i++) {
                for (int j = 0; j <= n - edge; j++) {
                    validSquare = true;
                    prevRowSum = 0;
                    if (j > 0) {
                        prevRowSum = rowPrefixSum[i][j-1];
                    }
                    oneSum = rowPrefixSum[i][j + edge-1] - prevRowSum;
                    for (int sqRow = 0; sqRow < edge; sqRow++) {
                        prevRowSum = 0;
                        if (j > 0) {
                            prevRowSum = rowPrefixSum[i + sqRow][j-1];
```

```
                        }
                        if (rowPrefixSum[i + sqRow][j + edge - 1] - prevRowSum != oneSum) {
                            validSquare = false;
                            break;
                        }
                    }
                    if (!validSquare) continue;
                    for (int sqCol = 0; sqCol < edge; sqCol++) {
                        prevColSum = 0;
                        if (i > 0) {
                            prevColSum = columnPrefixSum[i-1][j + sqCol];
                        }
                        if (columnPrefixSum[i + edge - 1][j + sqCol] - prevColSum != oneSum) {
                            validSquare = false;
                            break;
                        }
                    }
                    if (!validSquare) continue;
                    int prevPrincipalSum = 0, prevSecondarySum = 0;
                    if (i != 0 && j != 0) {
                      prevPrincipalSum = principalDiagonalSum[i-1][j-1];
                    }
                    if (i != 0 && (j + edge) < n) {
                        prevSecondarySum = secondaryDiagonal[i-1][j+edge];
                    }

                    if (principalDiagonalSum[i + edge-1][j + edge-1] - prevPrincipalSum != oneSum ||
                        secondaryDiagonal[i + edge-1][j] - prevSecondarySum != oneSum ){
                            continue;
                        }

                    return edge;
                }
            }
        }
        return 1;
    }
};
```

Listing 5: Largest Magic Square Solution

5 5. Maximum Side Length of a Square with Sum $\leq$ Threshold

5.1 Problem Statement

Given an $m \times n$ matrix `mat` and an integer `threshold`, return the maximum side-length of a square subgrid such that the sum of its elements is less than or equal to `threshold`. If no such square exists, return 0.

Problem Link: [3]

5.2 Theoretical Approach

2D Prefix Sums

To avoid recalculating the sum of elements for every candidate square (which would take $O(k^2)$ operations per query), we utilize the **2D Prefix Sum** technique. We transform the matrix M such that $M[i][j]$ stores the sum of the rectangle from $(0, 0)$ to (i, j).

The value of any subgrid defined by bottom-right corner (r, c) and side length k can be calculated in $O(1)$ time via the Inclusion-Exclusion principle:

$$\text{Sum} = P[r][c] - P[r-k][c] - P[r][c-k] + P[r-k][c-k]$$

Binary Search on Answer

The problem asks for the *maximum* side length k. The validity property is monotonic:

- If a square of size k exists with sum $\leq$ threshold, larger sizes might be possible.
- If no square of size k satisfies the condition (assuming non-negative elements), then no square of size $> k$ will satisfy it either.

Thus, we can perform a binary search for k in the range $[1, \min(m, n)]$.

5.3 Complexity Analysis

Let m and n be the dimensions of the matrix.

Time Complexity

The algorithm proceeds in three main steps:

1. **Preprocessing:** Building the 2D prefix sum matrix takes $\Theta(m \cdot n)$.
2. **Binary Search:** The search space for the side length is $L = \min(m, n)$. The loop runs $O(\log L)$ times.
3. **Feasibility Check:** Inside each step of the binary search, we iterate through all possible top-left corners to check for a valid square. This takes $O(m \cdot n)$ operations.

The total time complexity is:

$$T(m, n) = \Theta(m \cdot n) + O(m \cdot n \cdot \log(\min(m, n)))$$

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

[3] https://leetcode.com/problems/maximum-side-length-of-a-square-with-sum-less-than-or-equal-to-thres

Space Complexity

The implementation performs the prefix sum calculation **in-place**, modifying the input matrix directly to store cumulative sums.

$$S(m, n) \in O(1) \quad \text{(Auxiliary Space)}$$

5.4 Implementation

```
class Solution {
public:
    int maxSideLength(vector<vector<int>>& mat, int threshold) {
        int n = mat.size();
        int m = mat[0].size();
        int squareEdgeMax = min(n,m), squareEdgeMin = 1;
        int sqSum, end_sq_i, end_sq_j;
        vector<vector<int>> row_prefix_sum(n, vector<int>(m));
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                if (j == 0) {
                    continue;
                }
                mat[i][j]+=mat[i][j-1];
            }
        }
        for(int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                if (i == 0){
                    continue;
                }
                mat[i][j] += mat[i-1][j];
            }
        }
        int resEdge = 0;
        while (squareEdgeMin <= squareEdgeMax) {
            int midEdge = squareEdgeMin + (squareEdgeMax - squareEdgeMin) / 2;
            bool valid = false;
            for(int i = 0; i <= n - midEdge; i++) {
                for (int j = 0; j <= m - midEdge; j++) {
                    end_sq_i = i + midEdge - 1;
                    end_sq_j = j + midEdge - 1;
                    sqSum = mat[end_sq_i][end_sq_j];
                    if (i > 0) {
                        sqSum -= mat[i-1][end_sq_j];
                    }
                    if (j > 0) {
                        sqSum -= mat[end_sq_i][j-1];
                    }
                    if (i > 0 && j > 0) {
                        sqSum += mat[i-1][j-1];
                    }
                    if (sqSum <= threshold) {
                        valid = true;
```

```
                        break;
                    }
                }
                if (valid) break;
            }
            if(valid) {
                resEdge = midEdge;
                squareEdgeMin = midEdge + 1;
            } else {
                squareEdgeMax = midEdge - 1;
            }
        }
        return resEdge;
    }
};
```

Listing 6: Max Side Length Solution (Prefix Sum + Binary Search)

6 5. Construct the Minimum Bitwise Array I

6.1 Problem Statement

Given an array `nums` of n prime integers, construct an array `ans` such that for each index i:

$$\text{ans}[i] \text{ OR } (\text{ans}[i] + 1) = \text{nums}[i]$$

The values in `ans` must be minimized. If no solution exists, set $\text{ans}[i] = -1$.

Problem Link: [4]

6.2 Theoretical Approach

Bitwise Analysis

The operation $x \vee (x + 1)$ effectively fills the rightmost '0' bit of x with a '1' and preserves all other bits. Let $P = \text{nums}[i]$. Since P is the result of such an operation, its binary form must end with a sequence of '1's.

$$P = \ldots 1 \underbrace{1 \ldots 1}_{k \text{ ones}}$$

To minimize x, we must find the largest $x < P$ satisfying the condition. The transformation implies that x is derived from P by flipping the most significant bit of the trailing ones sequence to '0'.

Algebraic Derivation

For any odd prime P, the position of the first '0' bit can be found via:

$$\text{ZeroPos} = (P + 1) \ \& \ (\sim P)$$

Since we need to flip the bit immediately to the right of this zero (at position $k - 1$), our subtraction mask is:

$$\text{Mask} = \text{ZeroPos} \gg 1$$

The answer is simply $P - \text{Mask}$.

[4]https://leetcode.com/problems/construct-the-minimum-bitwise-array-i/

6.3 Complexity Analysis

Let n be the size of the input array.

Time Complexity

The solution avoids brute force loops by utilizing CPU-native bitwise instructions.

$$T(n) = O(n)$$

This is optimal compared to a simulation approach which might take $O(n \cdot \log(\max(nums)))$.

Space Complexity

$$S(n) \in \Theta(n) \quad \text{(Output Storage)}$$

6.4 Implementation

```cpp
class Solution {
public:
    vector<int> minBitwiseArray(vector<int>& nums) {
        int n = nums.size();
        vector<int> ans(n);
        for (int i = 0; i < n; i++) {
            if (nums[i] & 1){
                int new_bit_mask = (nums[i] + 1) & ~nums[i];
                new_bit_mask >>= 1;
                ans[i] = nums[i] - new_bit_mask;
            } else {
                ans[i] = -1;
            }
        }
        return ans;
    }
};
```

Listing 7: Minimum Bitwise Array (Optimal)

7 6. Merge Two Sorted Lists

7.1 Problem Statement

Given the heads of two sorted linked lists, `list1` and `list2`, merge them into a single sorted linked list. The merge operation should be performed by splicing together the nodes of the first two lists.

Problem Link: [5]

[5] https://leetcode.com/problems/merge-two-sorted-lists/

7.2 Theoretical Approach

Iterative Merge Strategy

The problem can be modeled as merging two sorted sequences A and B. Since the input structures are Linked Lists, we can perform the merge **in-place** by manipulating pointers, avoiding the overhead of creating new nodes.

We utilize a **Dummy Head** node (often called a sentinel) to simplify the implementation. This avoids special handling for the initialization of the result list's head pointer. We maintain a `tail` pointer that always points to the last node of the merged list constructed so far.

Mathematical Logic

At any step k, let the current heads of the unmerged portions be H_1 and H_2. The next node in the merged sequence is:

$$\text{Next} = \begin{cases} H_1 & \text{if } H_1.\text{val} < H_2.\text{val} \\ H_2 & \text{otherwise} \end{cases}$$

This greedy selection preserves the sorted order invariant: $\text{tail.val} \leq \text{Next.val}$.

7.3 Complexity Analysis

Let N and M be the lengths of the two linked lists.

Time Complexity

The algorithm performs a single pass over the lists. In each iteration of the `while` loop, one node is added to the merged list and the corresponding pointer advances. The total number of operations is proportional to the total number of nodes.

$$T(N, M) \approx N + M$$

$$T(N, M) \in \Theta(N + M)$$

Space Complexity

The algorithm uses $O(1)$ auxiliary space for the `res` and `tail` pointers. The nodes are re-linked in memory, not copied.

$$S(N, M) \in O(1)$$

7.4 Implementation

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 * int val;
 * ListNode *next;
 * ListNode() : val(0), next(nullptr) {}
 * ListNode(int x) : val(x), next(nullptr) {}
 * ListNode(int x, ListNode *next) : val(x), next(next) {}
```

```
 * };
 */
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
        ListNode res = ListNode(0);
        ListNode* tail = &res;
        while (list1 != nullptr && list2 != nullptr) {
            if (list1->val < list2->val) {
                tail->next = list1;
                list1 = list1->next;
                tail = tail->next;

            } else {
                tail->next = list2;
                list2 = list2->next;
                tail = tail->next;
            }
        }
        if (list1 != nullptr){
            tail->next = list1;
        }
        if (list2 != nullptr) {
            tail->next = list2;
        }
        return res.next;
    }
};
```

Listing 8: Merge Two Sorted Lists (Iterative)