Algorithm Design and Analysis

Mihail Iazinschi

Project Repository:

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

Contact:

GitHub Profile · LinkedIn Profile

January 24, 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.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 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 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 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;
                }
                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 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

```cpp
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 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 $x \lor (x + 1)$ 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.

$$P = \ldots 1\underbrace{1\ldots1}_{k \text{ ones}}$$

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 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) {}
```

```cpp
 * };
 */
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)

8 Minimum Pair Removal to Sort Array II

8.1 Problem Statement

Given an integer array `nums`, you can perform the following operation any number of times:

1. Select the **adjacent** pair with the **minimum** sum. If multiple such pairs exist, choose the leftmost one.

2. Replace the pair with their sum.

The goal is to return the **minimum number of operations** needed to make the array sorted in non-decreasing order.

Problem Link: [6]

[6] https://leetcode.com/problems/minimum-pair-removal-to-sort-array-ii/

8.2 Theoretical Approach

Mathematical Model

Let the sequence at step t be denoted as $S^{(t)} = \{v_1, v_2, \ldots, v_{k_t}\}$. We define the set of candidate operations (adjacent sums) at step t as:

$$\Sigma^{(t)} = \{(s_i, i) \mid s_i = v_i + v_{i+1},\ 1 \leq i < k_t\}$$

The greedy strategy dictates selecting the operation corresponding to $m^{(t)} = \min_{(s,i)\in\Sigma^{(t)}}(s)$.

To avoid the $O(N)$ cost of standard array deletions (which would lead to $O(N^2)$ total complexity), we map the sequence indices to a **Virtual Doubly Linked List** topology $\mathcal{L}$. This allows structural updates (merges) in $O(1)$ time.

Amortized Complexity Analysis (Aggregate Method)

Since the algorithm uses a **Min-Heap** with **Lazy Deletion**, the number of operations in the main loop is not fixed. We verify the efficiency using the Aggregate Method.

1. Event Accounting: The execution is driven by events stored in the heap:

- E_{init}: Initial insertion of all $N - 1$ adjacent sums.
- E_{merge}: A valid merge operation, which inserts up to 2 new sums (neighbors of the merged node).
- E_{lazy}: Extraction of a "stale" sum involving indices that were already merged.

2. Bound on Total Operations: Let N_{push} be the total number of elements pushed into the heap over the entire execution.

- Initial Load: $N - 1$ elements.
- Dynamic Load: Since each valid merge reduces the array size by 1, there are at most $N - 1$ valid merges. Each merge adds at most 2 new elements.

$$N_{push} \leq (N-1) + \sum_{k=1}^{N-1} 2 \approx 3N$$

In the worst case, we extract every element ever pushed. Thus, the total number of heap operations is bounded by $2 \cdot N_{push} \approx 6N$.

8.3 Complexity Analysis

Let N be the initial size of the input array.

Time Complexity

The total cost is the aggregate of all heap operations. The cost of a single operation is logarithmic with respect to the heap size H (where $H \leq 3N$).

$$T(N) = \sum_{j=1}^{6N} O(\log H) \approx 6N \cdot O(\log N)$$

$$T(N) \in O(N \log N)$$

Space Complexity

We utilize auxiliary structures to simulate the linked list and maintain the heap state.

- **Topology Vectors:** Two vectors of size N for `next` and `prev` pointers.
- **State Vector:** One vector of size N to track deleted nodes.
- **Heap:** Stores at most $3N$ elements.

$$S(N) \approx 2N + N + 3N = 6N \implies S(N) \in \Theta(N)$$

8.4 Implementation

```cpp
class Solution {
public:
    int minimumPairRemoval(vector<int>& nums) {
        // Min-Heap: stores {sum, left_index}
        std::priority_queue<pair<long long,int>, std::vector<pair<long long,int>>, std::greater<pair<long long,int>>> minHeap;
        int n = nums.size();
        int commitedSteps = 0;

        // Simulation of Doubly Linked List via Arrays
        vector<pair<long long,int>> linkedList(n);
        vector<pair<long long,int>> reverseLinkedList(n);
        vector<int> deleted_origin_nums(n, 0);

        int wrong_order_count = 0;

        // Initialization Phase
        for (int i = 0; i < n-1; i++) {
            minHeap.push(std::make_pair((long long)nums[i] + nums[i+1], i));
            linkedList[i] = std::make_pair((long long)nums[i], i+1);
            reverseLinkedList[i] = std::make_pair((long long)nums[i], i-1);
            if (nums[i] > nums[i+1]) wrong_order_count++;
        }
        linkedList[n-1] = std::make_pair((long long)nums[n-1], -1);

        if (wrong_order_count == 0) return 0;

        // Processing Phase
        while (wrong_order_count > 0 && !minHeap.empty()) {
            std::pair<long long,int> min_sum = minHeap.top();
            minHeap.pop();

            int min_sum_id = min_sum.second;
            long long min_sum_sum = min_sum.first;
            int sumPairSecondNumId = linkedList[min_sum_id].second;

            // Lazy Deletion Check
            if (deleted_origin_nums[min_sum_id] || sumPairSecondNumId == -1 || deleted_origin_nums[sumPairSecondNumId]) {
                continue;
            }
            if ((long long) linkedList[min_sum_id].first + linkedList[
   sumPairSecondNumId].first != min_sum_sum){
                continue;
            }

            // Inversion Check Optimization
            int prevNumId = reverseLinkedList[min_sum_id].second;
            int nextNum_afterSumId = linkedList[sumPairSecondNumId].
   second;

            if (prevNumId != -1 && linkedList[prevNumId].first >
   linkedList[min_sum_id].first) {
                wrong_order_count--;
            }
            if (linkedList[min_sum_id].first > linkedList[
   sumPairSecondNumId].first){
                wrong_order_count--;
            }
            if (nextNum_afterSumId != -1 && linkedList[
   sumPairSecondNumId].first > linkedList[nextNum_afterSumId].first) {
                wrong_order_count--;
            }

            // Commit Merge
            commitedSteps++;
            linkedList[min_sum_id].first = min_sum_sum;
            deleted_origin_nums[sumPairSecondNumId] = true;

            // Rewire Pointers
            linkedList[min_sum_id].second = nextNum_afterSumId;
            if (nextNum_afterSumId != -1) {
                reverseLinkedList[nextNum_afterSumId].second =
   min_sum_id;
            }

            // Update Inversions & Push New Candidates
            if (prevNumId != -1) {
                if (linkedList[prevNumId].first > linkedList[
   min_sum_id].first) wrong_order_count++;
                minHeap.push(std::make_pair(linkedList[prevNumId].
   first + linkedList[min_sum_id].first, prevNumId));
            }

            if (nextNum_afterSumId != -1) {
                if (linkedList[min_sum_id].first > linkedList[
   nextNum_afterSumId].first) wrong_order_count++;
                minHeap.push(std::make_pair(linkedList[
   nextNum_afterSumId].first + linkedList[min_sum_id].first,
   min_sum_id));
            }
        }
```

```
            return commitedSteps;
    }
};
```

Listing 9: Min-Heap with Lazy Deletion

9 Binary Tree Level Order Traversal

9.1 Problem Statement

Given the `root` of a binary tree, return the **level order traversal** of its nodes' values. (i.e., from left to right, level by level).

For example, given the tree:

$$[3, 9, 20, \text{null}, \text{null}, 15, 7]$$

The function should return:

$$[[3], [9, 20], [15, 7]]$$

Problem Link: [7]

9.2 Theoretical Approach

BFS with Hash Map Grouping

The problem requires traversing the tree layer by layer, which suggests a **Breadth-First Search (BFS)** approach. Unlike the standard BFS implementation that uses a nested loop to process levels, this solution utilizes a specific data structure strategy to decouple traversal from result construction.

1. State Management: We use a queue of pairs `std::queue<pair<int, TreeNode*>>`, where each element stores:

- The node pointer.
- The `depth_level` associated with that node.

2. Grouping Strategy: Instead of building the result vector immediately, we use an `unordered_map<int, vector<int>>` to group node values by their depth index.

$$\text{Map}[level] \leftarrow \text{Node.val}$$

This allows nodes to be processed in a continuous FIFO stream without needing to track the specific boundaries of each level during the queue processing.

3. Reconstruction: Finally, we iterate linearly starting from level 0 to construct the final 2D vector by extracting the accumulated lists from the map.

9.3 Complexity Analysis

Let N be the number of nodes in the binary tree.

[7] https://leetcode.com/problems/binary-tree-level-order-traversal/

Time Complexity

The algorithm consists of two phases:

1. **Traversal:** Each node is pushed into the queue and popped exactly once. The insertion into the hash map takes amortized $O(1)$.

$$T_{traversal} = \sum_{v \in V} (C_{push} + C_{pop} + C_{map}) \approx N \cdot O(1)$$

2. **Reconstruction:** We iterate through the map keys (levels) and move elements to the final vector. This visits each element one more time.

$$T_{reconstruct} = O(N)$$

Total Time Complexity:

$$T(N) \approx 2N \implies T(N) \in \Theta(N)$$

Space Complexity

We analyze the auxiliary memory usage:

- **Queue:** In the worst case (a complete binary tree), the queue holds the leaf level, which contains approximately $N/2$ nodes.
- **Map:** The hash map stores every node value exactly once, taking $O(N)$ space.
- **Output:** The result vector takes $O(N)$.

$$S(N) \approx N + \frac{N}{2} \implies S(N) \in \Theta(N)$$

9.4 Implementation

```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 * int val;
 * TreeNode *left;
 * TreeNode *right;
 * TreeNode() : val(0), left(nullptr), right(nullptr) {}
 * TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 * TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(
   left), right(right) {}
 * };
 */
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        std::queue<std::pair<int,TreeNode*>> node_queue;
        node_queue.push(std::make_pair(0, root));
        unordered_map<int, vector<int>> resultMap;
        vector<vector<int>> finalResult;

        while (!node_queue.empty()){
            std::pair<int, TreeNode*> currentPair = node_queue.front();
            node_queue.pop();

            if(currentPair.second != nullptr) {
                if (resultMap.find(currentPair.first) == resultMap.end()) {
                    vector<int> levelResult;
                    levelResult.push_back(currentPair.second->val);
                    resultMap[currentPair.first] = levelResult;
                } else {
                    resultMap[currentPair.first].push_back(currentPair.second->val);
                }

                node_queue.push(std::make_pair(currentPair.first + 1, currentPair.second->left));
                node_queue.push(std::make_pair(currentPair.first + 1, currentPair.second->right));
            }
        }

        int i = 0;
        while (resultMap.find(i) != resultMap.end()) {
            finalResult.push_back(resultMap[i++]);
        }
        return finalResult;
    }
};
```

Listing 10: BFS Level Order with Map Grouping

10 Minimize Maximum Pair Sum in Array

10.1 Problem Statement

The **pair sum** of a pair (a, b) is equal to $a + b$. given an array **nums** of even length n, pair up the elements into $n/2$ pairs such that each element is in exactly one pair, and the **maximum pair sum** is minimized.

For example, given:

$$\text{nums} = [3, 5, 2, 3]$$

The optimal pairs are $(2, 5)$ and $(3, 3)$. The max sum is $\max(7, 6) = 7$.

Problem Link: [8]

[8] https://leetcode.com/problems/minimize-maximum-pair-sum-in-array/

10.2 Theoretical Approach

Greedy Strategy

To minimize the maximum sum, we must avoid adding two large numbers together. The optimal strategy is to "dampen" the value of the largest available number by adding it to the smallest available number.

This intuition leads to a sorting-based approach:

1. Sort the array in ascending order: $x_1 \leq x_2 \leq \cdots \leq x_n$.
2. Pair the first element with the last, the second with the second-to-last, and so on.
3. The k-th pair is (x_k, x_{n-k+1}).

Correctness Proof

Let the sorted array be S. We claim that the pair (x_1, x_n) is part of an optimal solution. Suppose the optimal solution pairs x_1 with x_k (where $k < n$) and x_n with some x_j. The maximum sum is at least $x_j + x_n$. However, since $x_1 \leq x_j$, pairing (x_1, x_n) results in a sum $x_1 + x_n \leq x_j + x_n$. Thus, pairing the minimum with the maximum never yields a worse result than any other combination for the term involving x_n. By induction, this holds for all subsequent pairs.

10.3 Complexity Analysis

Let N be the number of elements in the array.

Time Complexity

The complexity is dominated by the sorting operation.

1. **Sorting:** Using an efficient comparison sort takes $O(N \log N)$.
2. **Pairing:** We iterate through the array once using two pointers, performing $N/2$ constant-time operations. This takes $O(N)$.

Total Time Complexity:

$$T(N) = O(N \log N) + O(N) \implies T(N) \in O(N \log N)$$

Space Complexity

The space complexity depends on the sorting implementation.

- C++ `std::sort` (Introsort) typically uses $O(\log N)$ stack space.
- The pairing logic uses $O(1)$ auxiliary space.

$$S(N) \in O(\log N)$$

10.4 Implementation

```cpp
class Solution {
public:
    int minPairSum(vector<int>& nums) {
        sort(nums.begin(), nums.end());
        int l = 0, r = nums.size()-1;
        int maxSum = 0;
        while (l < r){
            maxSum = max(maxSum, nums[l++] + nums[r--]);
        }
        return maxSum;
    }
};
```

Listing 11: Greedy Two-Pointer Solution