Algorithm Design and Analysis

Mihail Iazinschi

Project Repository:

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

Contact:

GitHub Profile · LinkedIn Profile

February 7, 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:

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

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.

```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 $x \vee (x+1)$ 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.

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

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 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)

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

11 Minimum Difference Between Highest and Lowest of K Scores

11.1 Problem Statement

You are given a 0-indexed integer array `nums`, where `nums[i]` represents the score of the i^{th} student. You are also given an integer k.

Pick the scores of any k students from the array so that the **difference** between the **highest** and the **lowest** of the k scores is **minimized**.

For example:

$$\text{nums} = [9, 4, 1, 7], \quad k = 2$$

Sorting the array descending gives $[9, 7, 4, 1]$. The pairs are $(9, 7), (7, 4), (4, 1)$. The minimum difference is $9 - 7 = 2$.

Problem Link: [9]

11.2 Theoretical Approach

Sorting and Sliding Window

In an unsorted array, values closest to each other can be at arbitrary positions. To select k values with the minimum range (max - min), those values must be close in magnitude. Therefore, the first step is to **sort** the array.

Let the sorted sequence (in descending order, as per the implementation) be $A = \{a_1, a_2, \dots, a_n\}$ such that $a_1 \geq a_2 \geq \cdots \geq a_n$.

Any valid selection of k scores with minimal difference forms a contiguous subarray (a window) of size k. For a window starting at index i, the elements are $\{a_i, a_{i+1}, \dots, a_{i+k-1}\}$.

- The maximum element is a_i (leftmost).
- The minimum element is a_{i+k-1} (rightmost).

[9] https://leetcode.com/problems/minimum-difference-between-highest-and-lowest-of-k-scores/

The problem reduces to finding:

$$\text{Result} = \min_{0 \le i \le n-k} (a_i - a_{i+k-1})$$

We verify this by using a **Sliding Window** of fixed size k that moves one step at a time from left to right.

11.3 Complexity Analysis

Let N be the number of elements in the array.

Time Complexity

1. **Sorting:** Sorting the array takes $O(N \log N)$.
2. **Sliding Window:** We iterate through the sorted array once, performing constant-time arithmetic operations at each step. This takes $O(N)$.

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

Space Complexity

- The sorting algorithm typically requires $O(\log N)$ stack space (for IntroSort/QuickSort).
- The sliding window uses only constant auxiliary space variables (`l`, `r`, `min_diff`).

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

11.4 Implementation

```cpp
class Solution {
public:
    int minimumDifference(vector<int>& nums, int k) {
        if (nums.size() < k) {
            return 0;
        }
        sort(nums.begin(), nums.end(), greater<int>());
        int l = 0, r = l + k - 1;
        int min_diff = INT_MAX;
        while (r < nums.size()){
            min_diff = min(nums[l++] - nums[r++], min_diff);
        }
        return min_diff;
    }
};
```

Listing 12: Sliding Window on Sorted Array

12 Minimum Absolute Difference

12.1 Problem Statement

Given an array of **distinct** integers `arr`, find all pairs of elements with the minimum absolute difference of any two elements.

Return a list of pairs in ascending order (with respect to pairs), where each pair $[a, b]$ satisfies:

1. a, b are from `arr`.
2. $a < b$.
3. $b - a$ equals the minimum absolute difference of any two elements in `arr`.

For example:

$$\text{arr} = [4, 2, 1, 3] \implies \text{Sorted: } [1, 2, 3, 4]$$

The minimum difference is 1. The pairs are $[1, 2], [2, 3], [3, 4]$.

Problem Link: [10]

12.2 Theoretical Approach

Sorting & Index Tracking Strategy

The fundamental observation is that in a set of numbers, the minimum absolute difference between two values can only occur between **adjacent elements** in the sorted sequence.

$$\min_{i \neq j} |a_i - a_j| = \min_k (a_{k+1} - a_k) \quad \text{where } a \text{ is sorted}$$

We employ a two-phase strategy using a Hash Map to decouple the search for the minimum difference from the construction of the result.

1. Sorting: First, we sort the array in ascending order to linearity place candidate pairs next to each other.

2. Single Pass with Map Tracking: We iterate through the sorted array and maintain a map:

$$\text{Map[diff]} \to \text{List of Starting Indices}$$

- If we calculate a difference smaller than our current global minimum (`min_sum`), we update `min_sum` and start a new list in the map.
- If the difference equals `min_sum`, we append the current index to the existing list.

3. Result Construction: Finally, we iterate through the indices stored at `Map[min_sum]` to build the returned vector.

12.3 Complexity Analysis

Let N be the number of elements in the array.

[10] https://leetcode.com/problems/minimum-absolute-difference/

Time Complexity

1. **Sorting:** `std::sort` takes $O(N \log N)$.
2. **Linear Scan:** We iterate through the array once. Map insertions and lookups take amortized $O(1)$.
3. **Construction:** We iterate through the valid indices (at most N) to build the result.

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

Space Complexity

We utilize an auxiliary hash map. In the worst case (e.g., an arithmetic progression like $[1, 2, 3, 4]$), all adjacent differences are equal to the minimum.

- **Map Storage:** Stores up to $O(N)$ indices.
- **Output:** The result vector takes $O(N)$.

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

12.4 Implementation

```cpp
class Solution {
public:
    vector<vector<int>> minimumAbsDifference(vector<int>& arr) {
        sort(arr.begin(), arr.end());
        vector<vector<int>> result;
        unordered_map<int, vector<int>> sum_indexes;
        int min_sum = INT_MAX;
        for(int i = 0; i < arr.size()-1; i++) {
            int sum = arr[i+1] - arr[i];
            if (sum < min_sum) {
                vector<int> current_sum_indexes;
                current_sum_indexes.push_back(i);
                sum_indexes[sum] = current_sum_indexes;
                min_sum = sum;
            } else if (sum == min_sum){
                sum_indexes[min_sum].push_back(i);
            }
        }
        for(int index : sum_indexes[min_sum]) {
            result.push_back({arr[index], arr[index+1]});
        }
        return result;
    }
};
```

Listing 13: Sorting + Hash Map Tracking

13 Minimum Cost Path with Edge Reversals

13.1 Problem Statement

You are given a directed, weighted graph with n nodes and a list of edges $[u, v, w]$ representing a directed edge from u to v with cost w.

Each node has a special **switch** mechanism. When you arrive at a node u, you can choose to traverse any **incoming** edge $v \to u$ in the reverse direction ($u \to v$).

- Traversing a standard edge $u \to v$ costs w.
- Traversing a reversed edge (using the switch) costs $2 \times w$.

Find the minimum cost to travel from node 0 to node $n - 1$.

Problem Link: [11]

13.2 Theoretical Approach

Graph Modeling

The key insight is to translate the dynamic "switch" rule into static graph edges. The rule essentially states that connectivity between two nodes u and v connected by an edge $[u, v, w]$ is bidirectional, but the costs are asymmetric.

For every input edge $(u \to v)$ with weight w, we add two directed edges to our adjacency list:

1. **Forward Edge:** $u \to v$ with weight w. This represents following the natural direction.
2. **Backward Edge:** $v \to u$ with weight $2w$. This represents activating the switch at node v to reverse the incoming edge from u.

After this transformation, the problem becomes finding the shortest path in a directed graph with non-negative weights, which is solved efficiently using **Dijkstra's Algorithm**.

13.3 Complexity Analysis

Let N be the number of nodes and M be the number of input edges.

Time Complexity

We perform Dijkstra's algorithm on a graph with N vertices and $2M$ edges. Using a Min-Heap (Priority Queue):

- Each edge is relaxed at most once: $O(M \log N)$.
- Each vertex is extracted once: $O(N \log N)$.

$$T(N, M) = O(M \log N)$$

Space Complexity

- **Adjacency List:** Stores $2M$ edges ($O(M)$).
- **Distance Array:** Stores N values ($O(N)$).

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

[11] https://leetcode.com/problems/minimum-cost-path-with-edge-reversals/

13.4 Implementation

```cpp
class Solution {
public:
    struct heap_element {
        int node;
        int weight;
        bool operator>(const heap_element& oElement) const {
            return weight > oElement.weight;
        }
    };

    int minCost(int n, vector<vector<int>>& edges) {
        vector<vector<pair<int,int>>> graph(n);
        for (auto edge : edges) {
            int from = edge[0];
            int to = edge[1];
            int weight = edge[2];
            graph[from].push_back(std::make_pair(to, weight));
            graph[to].push_back(std::make_pair(from, 2 * weight));
        }

        vector<int> distanceTo(n, INT_MAX);
        vector<bool> visited(n, false);
        priority_queue<heap_element, vector<heap_element>, greater<heap_element>> minHeap;

        minHeap.push({0, 0});
        distanceTo[0] = 0;

        while (!minHeap.empty()) {
            int node = minHeap.top().node;
            minHeap.pop();

            if (visited[node]) continue;
            visited[node] = true;

            for (auto toNode : graph[node]) {
                int toId = toNode.first;
                int toWeight = toNode.second;
                if (distanceTo[node] + toWeight < distanceTo[toId]) {
                    distanceTo[toId] = distanceTo[node] + toWeight;
                    minHeap.push({toId, distanceTo[toId]});
                }
            }
        }
        if (distanceTo[n-1] == INT_MAX) return -1;
        return distanceTo[n-1];
    }
};
```

Listing 14: Dijkstra with Asymmetric Edge Costs

14 Palindrome Partitioning

14.1 Problem Statement

Given a string s, partition s such that **every substring** of the partition is a **palindrome**. Return all possible palindrome partitioning.

For example, given the input:

$$s = \text{"aab"}$$

The valid partitions are:

$$[[\text{"a"}, \text{"a"}, \text{"b"}], [\text{"aa"}, \text{"b"}]]$$

The partition $[\text{"a"}, \text{"ab"}]$ is invalid because "ab" is not a palindrome.

Problem Link: [12]

14.2 Theoretical Approach

Backtracking (State Space Search)

The problem asks for an enumeration of all valid decompositions, which implies an exhaustive search strategy. We can model the string S of length N as having $N - 1$ potential "cut" positions between characters. Each cut can either be active or inactive, leading to a theoretical search space of 2^{N-1} partitions.

We employ a **Depth-First Search (DFS)** strategy:

1. Start from index 0.
2. Iterate through all possible end indices i (from current start to $N - 1$).
3. **Validation:** Check if the prefix substring $s[\text{start} \dots i]$ is a palindrome.
4. **Recursion:** If valid, add the substring to the current partition path and recursively solve for the remaining suffix starting at $i + 1$.
5. **Backtrack:** After the recursive call returns, remove the substring and try the next potential cut.

This approach effectively **prunes** the recursion tree: if a prefix is not a palindrome, we do not waste computation time exploring any partitions that would follow it.

14.3 Complexity Analysis

Let N be the length of the string s.

Time Complexity

The worst-case scenario occurs when every substring is a palindrome (e.g., $s = \text{"aaaa"}$). In this case, the algorithm generates all 2^{N-1} possible partitions. For each valid partition (leaf node in the recursion tree), we perform linear work $O(N)$ to copy the substrings into the result vector.

$$T(N) \approx \sum_{k=0}^{N-1} \binom{N}{k} \cdot N$$

$$T(N) \in O(N \cdot 2^N)$$

[12] https://leetcode.com/problems/palindrome-partitioning/

Space Complexity

- **Recursion Stack:** The maximum depth of the recursion tree is N (single character partitions).
- **Buffer Space:** The temporary vector stores at most N substrings.

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

14.4 Implementation

```cpp
class Solution {
public:
    bool palindromeCheck(string s){
        int i = 0;
        int n = s.length();
        while(i < n && s[i] == s[n-i-1]){
            i++;
        }
        if (i < n) return false;
        return true;
    }

    void solvePalindromes(int begin, string s, vector<string> &
   currentBuffer, vector<vector<string>> &res) {
        // Base Case: Reached end of string
        if (begin == s.length()){
            res.push_back(currentBuffer);
            return;
        }
        for(int i = begin; i < s.length(); i++) {
            string currentPalindrome = s.substr(begin, i-begin+1);
                if (palindromeCheck(currentPalindrome)) {
                currentBuffer.push_back(currentPalindrome);
                solvePalindromes(i+1, s, currentBuffer, res);
                currentBuffer.pop_back();
            }
        }
    }

    vector<vector<string>> partition(string s) {
        vector<vector<string>> res;
        vector<string> tempBuffer;
        solvePalindromes(0, s, tempBuffer, res);
        return res;
    }
};
```

Listing 15: Backtracking Solution

15 Minimum Cost to Convert String I

15.1 Problem Statement

You are given two strings `source` and `target` of length n, and arrays `original`, `changed`, and `cost` defining directed transformation rules. Each rule allows changing the character `original[i]` to `changed[i]` with a cost of `cost[i]`.

The goal is to convert `source` into `target` character by character with the **minimum total cost**. If a character conversion is impossible, return -1.

For example:

$$\text{source} = "abcd", \quad \text{target} = "acbe"$$

Transformations: $a \to b(2), b \to c(5), c \to b(5), c \to e(1), e \to b(2), d \to e(20)$.

- 'a' $\to$ 'a': Cost 0.
- 'b' $\to$ 'c': Cost 5.
- 'c' $\to$ 'b': Path $c \to e \to b$ costs $1 + 2 = 3$ (optimal).
- 'd' $\to$ 'e': Cost 20.

Total Cost: $0 + 5 + 3 + 20 = 28$.

Problem Link: [13]

15.2 Theoretical Approach

Graph Modeling

We interpret the 26 lowercase English letters as nodes in a weighted directed graph. Each transformation rule represents a directed edge $u \to v$ with weight w. The problem reduces to finding the sum of shortest path distances for pairs $(source[i], target[i])$.

On-Demand Dijkstra with Caching

Since the graph is small ($|V| = 26$), we can compute shortest paths efficiently. While Floyd-Warshall is an option ($O(V^3)$), the implemented solution uses an **On-Demand Dijkstra**:

1. Maintain a cache matrix `distTo[26][26]`.
2. Iterate through the `source` string.
3. If we encounter a source character u for the first time, run Dijkstra starting from u to compute distances to *all* other characters and populate the cache row.
4. If we encounter u again, simply look up the cached result.

This strategy is efficient when the input string contains repeated characters or uses a subset of the alphabet.

15.3 Complexity Analysis

Let N be the string length, M be the number of rules, and $\Sigma = 26$.

[13] https://leetcode.com/problems/minimum-cost-to-convert-string-i/

Time Complexity

1. **Graph Construction:** $O(M)$.
2. **Shortest Paths:** In the worst case, we run Dijkstra Σ times. Each run takes $O(M \log \Sigma)$. Total: $O(\Sigma \cdot M \log \Sigma)$.
3. **Cost Calculation:** Iterating through the string takes $O(N)$.

Since Σ is constant:

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

Space Complexity

We store the graph edges and a fixed-size distance matrix.

$$S(N, M) \in O(M + \Sigma^2) \approx O(M)$$

15.4 Implementation

```cpp
#define ll long long
class Solution {
public:
    struct heap_element {
        ll node;
        ll weight;
        bool operator>(const heap_element& oElement) const {
            return weight > oElement.weight;
        }
    };

    vector<vector<pair<ll, ll>>> fullGraph;
    const long long cmpMax = LONG_LONG_MAX - 1000000000;
    ll distTo[26][26];
    vector<bool> visited;

    void dijkstra(ll startNode) {
        priority_queue<heap_element, vector<heap_element>, greater<
  heap_element>> minHeap;
        minHeap.push({startNode, 0});
        while (!minHeap.empty()) {
            auto node = minHeap.top();
            minHeap.pop();
            ll distCost = node.weight;
            ll destId = node.node;

            if (distCost > distTo[startNode][destId]) continue;

            for (auto neighNode : fullGraph[destId]) {
                ll to = neighNode.first;
                ll price = neighNode.second;
                if (distTo[startNode][destId] + price < distTo[
  startNode][to]) {
```

```
                    distTo[startNode][to] = distTo[startNode][destId] + price;
                    minHeap.push({to, distTo[startNode][to]});
                }
            }
        }
    }

    long long minimumCost(string source, string target, vector<char>& original, vector<char>& changed, vector<int>& cost) {
        vector<vector<pair<ll, ll>>> graph(26);
        visited = vector<bool>(26, false);
        for(int i = 0; i < 26; i++) {
            for(int j = 0; j < 26; j ++) distTo[i][j] = (i==j) ? 0 : cmpMax;
        }

        for(int i = 0; i < original.size(); i++) {
            graph[original[i]-'a'].push_back({changed[i]-'a', (ll)cost[i]});
        }
        fullGraph = graph;

        ll resCost = 0;
        for (int i = 0; i < source.size(); i++) {
            ll from = source[i] - 'a';
            ll to = target[i] - 'a';

            if (from == to) continue;

            if (!visited[from]) {
                dijkstra(from);
                visited[from] = true;
            }

            if (distTo[from][to] == cmpMax) return -1;
            resCost += distTo[from][to];
        }
        return resCost;
    }
};
```

Listing 16: On-Demand Dijkstra Solution

16 Non-overlapping Intervals

16.1 Problem Statement

Given an array of intervals where $intervals[i] = [start_i, end_i]$, return the **minimum number of intervals** you need to remove to make the rest of the intervals non-overlapping.

Example:

$$Input : [[1, 2], [2, 3], [3, 4], [1, 3]]$$

$$Output : 1$$

Explanation: removing $[1, 3]$ leaves $[1, 2], [2, 3], [3, 4]$ which are non-overlapping.

Problem Link: [14]

16.2 Theoretical Approach

Greedy Strategy (Interval Scheduling)

This is equivalent to finding the maximum number of non-overlapping intervals (the Activity Selection Problem). The optimal greedy strategy is to select intervals based on their **End Times**.

Logic:

1. Sort all intervals by their end time in ascending order.
2. Select the first interval (which ends earliest). This leaves the maximum possible free time for subsequent intervals.
3. Iterate through the sorted list. If an interval starts *after* the previous one ends, select it. Otherwise, it overlaps, so we discard it (increment the removal counter).

Sorting by start time or length does not guarantee an optimal solution. Sorting by end time ensures we always minimize the "resource consumption" (time) of the selected interval.

16.3 Complexity Analysis

Let N be the number of intervals.

Time Complexity

1. **Sorting:** We use `std::sort` with a custom comparator.
$$T_{sort} = O(N \log N)$$
2. **Scanning:** A single pass through the sorted array.
$$T_{scan} = O(N)$$

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

Space Complexity

The sorting algorithm typically uses $O(\log N)$ stack space. The linear scan uses constant extra space.

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

[14] https://leetcode.com/problems/non-overlapping-intervals/

16.4 Implementation

```cpp
class Solution {
public:
    static bool customCompare(vector<int> &a, vector<int> &b) {
        return a[1] < b[1];
    }
    int eraseOverlapIntervals(vector<vector<int>>& intervals) {
        int res = -1;
        sort(intervals.begin(), intervals.end(), customCompare);

        vector<int> lastInterval = intervals[0];

        for(auto interval: intervals) {
            if (lastInterval[1] > interval[0]){
                res++;
            } else {
                lastInterval = interval;
            }
        }
        return res;
    }
};
```

Listing 17: Greedy by End Time

17 Find Smallest Letter Greater Than Target

17.1 Problem Statement

Given a characters array `letters` that is sorted in **non-decreasing order** and a character `target`, return the **smallest character** in the array that is strictly greater than `target`.

If such a character does not exist, return the first character in `letters` (wrap-around behavior).

Example:

$$\text{letters} = ['c', 'f', 'j'], \quad \text{target} = 'a'$$

$$\text{Output} : 'c'$$

Explanation: The smallest character strictly greater than 'a' is 'c'.

Problem Link: [15]

17.2 Theoretical Approach

Binary Search (Upper Bound)

Since the input array is sorted, the problem is a classic application of **Binary Search** to find the **Upper Bound** of a value.

We seek the index of the first element satisfying $L[i] > target$.

1. Initialize result `res` to $L[0]$ to handle the wrap-around case by default.

[15] https://leetcode.com/problems/find-smallest-letter-greater-than-target/

2. Maintain a search range $[l, r]$. Calculate *mid*.
3. If $L[mid] > target$: This is a valid candidate. We store it and attempt to find a smaller valid candidate in the left half ($r = mid - 1$).
4. If $L[mid] \leq target$: The current element and everything to its left are too small. We search in the right half ($l = mid + 1$).

17.3 Complexity Analysis

Let N be the length of the `letters` array.

Time Complexity

The algorithm uses Binary Search, which divides the search space in half at each iteration.

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

Space Complexity

The algorithm operates in constant extra space.

$$S(N) = O(1)$$

17.4 Implementation

```cpp
class Solution {
public:
    char nextGreatestLetter(vector<char>& letters, char target) {
        int l = 0, r = letters.size() - 1;
        char res = letters[0];
        while (l <= r) {
            int mid = l + (r - l) / 2;

            if (letters[mid] > target) {
                res = letters[mid];
                r = mid - 1;
            } else {
                l = mid + 1;
            }
        }
        return res;
    }
};
```

Listing 18: Binary Search for Upper Bound

18 Divide an Array Into Subarrays With Minimum Cost I

18.1 Problem Statement

You are given an array of integers `nums`. The **cost** of an array is defined as the value of its **first element**. You must divide `nums` into 3 disjoint contiguous subarrays. Return the **minimum**

possible sum of the costs of these subarrays.

Example:

$$\text{Input: } [1, 2, 3, 12]$$

$$\text{Optimal Split: } [1], [2], [3, 12]$$

$$\text{Cost: } 1 + 2 + 3 = 6$$

Problem Link: [16]

18.2 Theoretical Approach

Greedy Strategy

The division into 3 subarrays implies selecting 3 starting positions (indices).

1. The first subarray **must** start at index 0. Thus, $nums[0]$ is an unavoidable component of the total cost.
2. We must select two additional indices i and j ($1 \leq i < j < n$) to start the second and third subarrays.

To minimize the total sum $S = nums[0]+nums[i]+nums[j]$, we must minimize the variable components $nums[i] + nums[j]$. This is equivalent to finding the **two smallest integers** in the subarray $nums[1 \dots n-1]$.

18.3 Complexity Analysis

Let N be the size of the array.

Time Complexity

We iterate through the array once (from index 1 to $N-1$) to identify the two minimum values.

$$T(N) = O(N)$$

Space Complexity

The algorithm uses constant extra space for variables.

$$S(N) = O(1)$$

18.4 Implementation

```
class Solution {
public:
    int minimumCost(vector<int>& nums) {
        int n = nums.size();
        int minL = INT_MAX, minR = INT_MAX;

        for(int i = 1; i < n; i++) {
            if (nums[i] < minL) {
                minR = minL;
```

[16] https://leetcode.com/problems/divide-an-array-into-subarrays-with-minimum-cost-i/

```
                minL = nums[i];
            } else if (nums[i] < minR) {
                minR = nums[i];
            }
        }
        return nums[0] + minL + minR;
    }
};
```

Listing 19: Linear Scan for Two Minimums

19 Longest Consecutive Sequence

19.1 Problem Statement

Given an unsorted array of integers `nums`, return the length of the longest consecutive elements sequence. The algorithm must run in $O(N)$ time complexity.

Example:

$$\text{Input: } [100, 4, 200, 1, 3, 2]$$

$$\text{Sorted (Conceptual): } [1, 2, 3, 4], [100], [200]$$

$$\text{Longest Sequence: } [1, 2, 3, 4] \implies \text{Length } 4$$

Problem Link: [17]

19.2 Theoretical Approach

Hash Set & Bi-directional Expansion

Sorting the array would take $O(N \log N)$, which violates the requirement. To achieve $O(N)$, we utilize a **Hash Set** (`unordered_set`) for $O(1)$ lookups.

The strategy is "Expand and Consume":

1. Insert all numbers into the set.
2. Iterate through the array. If a number x exists in the set:
   - Remove x to mark it as visited.
   - Repeatedly check for $x - 1, x - 2, \ldots$ (Left Expansion) and remove them.
   - Repeatedly check for $x + 1, x + 2, \ldots$ (Right Expansion) and remove them.
   - The total count of removed elements forms one consecutive component.

By removing elements as soon as they are visited, we ensure that each "cluster" of consecutive numbers is processed exactly once, regardless of which element in the cluster we encounter first.

19.3 Complexity Analysis

Let N be the number of elements.

[17] https://leetcode.com/problems/longest-consecutive-sequence/

Time Complexity

We perform an aggregate analysis. Although there are nested loops, the total number of operations is bounded linearly:

- Each element is inserted into the set once: $O(N)$.
- Each element is erased from the set exactly once: $O(N)$.
- The lookups that stop the `while` loops occur at most $2 \times (\text{number of clusters}) \leq 2N$.

Thus, the amortized time complexity is linear.

$$T(N) = O(N)$$

Space Complexity

We store all unique elements in the hash set.

$$S(N) = O(N)$$

19.4 Implementation

```cpp
class Solution {
public:
    int longestConsecutive(vector<int>& nums) {
        int maxSeq = 0;
        unordered_set<int> allNums;
        for(int x : nums) allNums.insert(x);

        for(int i = 0; i < nums.size(); i++) {
            int curSeq = 0;
            auto it = allNums.find(nums[i]);

            if (it != allNums.end()) {
                allNums.erase(*it);
                curSeq++;
                int l = 1;
                while (true) {
                    auto lower = allNums.find(nums[i] - l);
                    if (lower != allNums.end()) {
                        curSeq++; allNums.erase(*lower); l++;
                    } else break;
                }
                int r = 1;
                while (true) {
                    auto higher = allNums.find(nums[i] + r);
                    if (higher != allNums.end()) {
                        curSeq++; allNums.erase(*higher); r++;
                    } else break;
                }
                maxSeq = max(maxSeq, curSeq);
            }
        }
        return maxSeq;
    }
};
```

Listing 20: O(N) Set Expansion

20 Car Fleet

20.1 Problem Statement

There are N cars going to the same destination `target`. You are given arrays `position` and `speed`. A car cannot pass another car ahead of it. If a faster car catches up to a slower one, they form a **car fleet** and travel at the speed of the slower car.

Return the number of car fleets that will arrive at the destination.

Example:

$$\text{Target} = 12, \quad \text{Pos} = [10, 8, 0, 5, 3], \quad \text{Spd} = [2, 4, 1, 1, 3]$$

$$\text{Output: } 3$$

Problem Link: [18]

20.2 Theoretical Approach

Time-Based Monotonic Analysis

The key insight is that cars only interact with those ahead of them. Therefore, processing cars from the closest to the target (largest position) to the farthest simplifies the logic.

For each car, calculate the time to reach the target:

$$t = \frac{\text{target} - \text{position}}{\text{speed}}$$

We iterate through the cars sorted by position in descending order. We maintain the arrival time of the current **fleet leader** (the slowest car in the current pack).

- If a car behind has $t_{current} \leq t_{leader}$, it catches up and merges into the fleet.
- If a car behind has $t_{current} > t_{leader}$, it arrives later than the fleet ahead. It becomes the leader of a new fleet.

This logic essentially mimics a **Monotonic Stack** (strictly increasing arrival times of fleets).

20.3 Complexity Analysis

Let N be the number of cars.

[18] https://leetcode.com/problems/car-fleet/

Time Complexity

1. **Sorting:** We sort the cars by position.

$$T_{sort} = O(N \log N)$$

2. **Iteration:** A single pass to compute times and manage the stack.

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

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

Space Complexity

We require $O(N)$ space to store the pair vector and the stack.

$$S(N) = O(N)$$

20.4 Implementation

```cpp
class Solution {
public:
    int carFleet(int target, vector<int>& position, vector<int>& speed) {
        vector<pair<int,int>> cars(position.size());
        stack<double> timeLeft;
        for(int i = 0; i < position.size(); i++) {
            cars[i] = std::make_pair(position[i], speed[i]);
        }
        sort(cars.rbegin(), cars.rend());
        timeLeft.push((double) (target - cars[0].first) / cars[0].second);
        for(int i = 1; i < cars.size();i++){
            double time = (double)(target - cars[i].first) / cars[i].second;
            if (timeLeft.size() >= 1){
                double lastFleetCar = timeLeft.top();
                if (time > lastFleetCar) {
                    timeLeft.push(time);
                }
            }
        }
        return timeLeft.size();
    }
};
```

Listing 21: Sorting and Stack approach

21 Trionic Array I

21.1 Problem Statement

You are given an integer array `nums`. An array is defined as **Trionic** if it follows a specific pattern of three strictly monotonic segments:

1. **Increasing** from index 0 to p.
2. **Decreasing** from index p to q.
3. **Increasing** from index q to $n - 1$.

The indices must satisfy $0 < p < q < n - 1$. Return `true` if the array is trionic.

Example:

$$\text{Input: } [1, 3, 5, 4, 2, 6]$$

$$\text{Segments: } [1, 3, 5] \nearrow, [5, 4, 2] \searrow, [2, 6] \nearrow$$

$$\text{Output: true}$$

Problem Link: [19]

21.2 Theoretical Approach

Sequential Linear Scan

The problem requires validating the shape of the data sequence. This is effectively a state machine with three states:

1. **Climbing to First Peak (p):** Iterate while $nums[i] < nums[i + 1]$. If we stop at $i = 0$ or $i = n - 1$, the shape is invalid (no peak or monotonic array).
2. **Descending to Valley (q):** From p, iterate while $nums[i] > nums[i + 1]$. If we don't move ($p = q$) or reach the end immediately, the shape is invalid.
3. **Climbing to End:** From q, iterate while $nums[i] < nums[i + 1]$. We must successfully reach the index $n - 1$.

This approach ensures we verify both the strict monotonicity and the existence of the turning points (peak and valley).

21.3 Complexity Analysis

Let N be the length of the array.

Time Complexity

We perform a single pass through the array. The index i is incremented at most N times across the three loops.

$$T(N) = O(N)$$

[19] https://leetcode.com/problems/trionic-array-i/

Space Complexity

The algorithm uses $O(1)$ auxiliary space.

$$S(N) = O(1)$$

21.4 Implementation

```cpp
class Solution {
public:
    bool isTrionic(vector<int>& nums) {
        int i = 0; int n = nums.size();
        if (n <= 3) return false;
        bool validArray = true;
        int s = -2000, q = -2000, p = -2000;
        while (i < n-1){
            if (!(nums[i+1] > nums[i])) {
                break;
            }
            i++;
        }
        if (i == 0) return 0;
        p = nums[i];
        while (i < n-1){
            if (!(nums[i+1] < nums[i])) {
                break;
            }
            i++;
        }
        q = nums[i];
        while (i < n-1){
            if (!(nums[i+1] > nums[i])) {
                break;
            }
            i++;
        }
        if (i == n-1 && nums[n-1] > q  && q < p && p > s) {
            return validArray;
        }
        return false;
    }
};
```

Listing 22: Three-Pass Simulation

22 Largest Rectangle in Histogram

22.1 Problem Statement

Given an array of integers `heights` representing the histogram's bar height where the width of each bar is 1, return the **area of the largest rectangle** in the histogram.

Example:

$$\text{Input: } [2, 1, 5, 6, 2, 3]$$

Output: 10

The largest rectangle is formed by bars 5 and 6, limited by height 5. Width = 2. Area = $5 \times 2 = 10$.

Problem Link: [20]

22.2 Theoretical Approach

Monotonic Stack with Width Accumulation

The core idea is to maintain a **Monotonic Increasing Stack**. This means every element in the stack is taller than or equal to the element below it.

When we encounter a bar `current` that is shorter than the bar at the top of the stack (`top`), we identify a potential maximum rectangle:

- The rectangle with height `top.height` cannot extend further to the right (blocked by `current`).
- We pop `top` and calculate the area.

Width Calculation Strategy: Instead of storing indices, your solution stores a structure {`height, width`}.

1. When popping elements (taller bars), we sum their widths into a variable `accumulatedWidth`.
2. We calculate the area for each popped bar using this accumulated width.
3. Crucially, before pushing the `current` (shorter) bar, we add `accumulatedWidth` to its own width. This signifies that the current bar can technically extend to the left over the space previously occupied by the taller bars we just removed.

Sentinel Value

We append a height of `0` to the end of the array. This acts as a universal "stopper," guaranteeing that any bars remaining in the stack (which represent an increasing sequence extending to the end) are popped and processed.

22.3 Complexity Analysis

Let N be the number of bars.

Time Complexity

Each bar is pushed onto the stack exactly once and popped exactly once. The processing per bar is $O(1)$ amortized.

$$T(N) = O(N)$$

Space Complexity

The stack can store up to N elements in the worst case (increasing heights).

$$S(N) = O(N)$$

[20] https://leetcode.com/problems/largest-rectangle-in-histogram/

22.4 Implementation

```
class Solution {
public:
    struct stackBar {
        int barHeight;
        int barWidth;
    };
    int largestRectangleArea(vector<int>& heights) {
        // area = max(min(cur_bar_min_height, stack_last_el_height) * (stack_last_el_width + 1), cur_bar_min_height)
        // fail on [5 9 9]
        stack<stackBar> bars;
        int maxRectangle = 0, n = heights.size();
        heights.push_back(0);
        for(int i = 0; i < n+1; i++) {
           stackBar currentBar = {heights[i], 1};
           int maxWidth = 0;
           while (!bars.empty() && bars.top().barHeight >= currentBar.barHeight) {
                stackBar lastBar = bars.top();
                bars.pop();
                maxWidth += lastBar.barWidth;
                maxRectangle = max(maxRectangle, maxWidth * lastBar.barHeight);
           }
           currentBar.barWidth +=maxWidth;
           bars.push(currentBar);
        }
        return maxRectangle;
    }
};
```

Listing 23: Monotonic Stack with Struct

23 Find Minimum in Rotated Sorted Array

23.1 Problem Statement

Suppose an array sorted in ascending order is **rotated** at some pivot unknown to you beforehand. (i.e., $[0, 1, 2, 4, 5, 6, 7]$ might become $[4, 5, 6, 7, 0, 1, 2]$).

Find the minimum element. You may assume no duplicate exists in the array. The algorithm must run in $O(\log N)$ time.

Example:

Input: $[3, 4, 5, 1, 2]$

Output: 1

Problem Link: [21]

[21] https://leetcode.com/problems/find-minimum-in-rotated-sorted-array/

23.2 Theoretical Approach

Modified Binary Search

Since the array is partially sorted, we can utilize **Binary Search**. The key challenge is determining which half of the array contains the minimum element (the "pivot" point where the order resets).

We compare the middle element `nums[mid]` with the rightmost element of the current search space `nums[r]`:

1. **Case 1:** $nums[mid] < nums[r]$
   - This implies the sub-array from mid to r is sorted.
   - The minimum value cannot lie in the range $(mid, r]$.
   - However, mid itself could be the minimum.
   - **Action:** Set $r = mid$.
2. **Case 2:** $nums[mid] > nums[r]$
   - This implies the sequence "breaks" somewhere to the right of mid. The values from l to mid are part of the larger rotated segment.
   - The minimum must lie strictly to the right of mid.
   - **Action:** Set $l = mid + 1$.

The loop terminates when $l == r$, at which point we have found the minimum.

23.3 Complexity Analysis

Let N be the size of the array.

Time Complexity

The search space is divided by 2 at every iteration.

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

Space Complexity

The iterative approach requires constant extra space.

$$S(N) = O(1)$$

23.4 Implementation

```cpp
class Solution {
public:
    int findMin(vector<int>& nums) {
        int l = 0, r = nums.size() - 1, mid;
        while(l < r) {
            mid = l + (r-l)/2;
            if (nums[mid] < nums[r]){
                r = mid;
            } else if (nums[mid] > nums[r]) {
                l = mid + 1;
            }
        }
        return nums[l];
    }
};
```

Listing 24: Binary Search for Pivot

24 Transformed Array

24.1 Problem Statement

Given an integer array `nums` representing a circular array, create a new array `result` of the same size. For each index i, perform the following shift based on `nums[i]`:

- If `nums[i]` > 0: Move `nums[i]` steps to the **right**.
- If `nums[i]` < 0: Move |`nums[i]`| steps to the **left**.
- If `nums[i]` $= 0$: Stay at index i.

The movement wraps around the array boundaries.

Example:

$$\text{Input: } [3, -2, 1, 1]$$

$$\text{Output: } [1, 1, 1, 3]$$

Problem Link: [22]

24.2 Theoretical Approach

Circular Indexing Modular Arithmetic

The problem asks for a simulation of movement on a ring. The destination index T after moving k steps from index i in a circular array of size N is generally given by:

$$T = (i + k) \pmod N$$

The Negative Modulo Challenge: In C++, the modulo operator `%` retains the sign of the dividend (e.g., $-2\%5 = -2$). However, array indices must be in the range $[0, N-1]$. To handle this correctly, we use the following formula which works for both positive and negative k:

$$T = ((i + k)\%N + N)\%N$$

This ensures that if $(i + k)\%N$ results in a negative value (e.g., -2), adding N brings it into the positive range, and the final modulo ensures it stays within bounds.

24.3 Complexity Analysis

Let N be the length of the array.

[22] https://leetcode.com/problems/transformed-array/

Time Complexity

We perform a single pass through the array. For each element, we calculate the target index using constant-time arithmetic operations.

$$T(N) = O(N)$$

Space Complexity

We allocate a new vector of size N to store the result.

$$S(N) = O(N)$$

24.4 Implementation

```cpp
class Solution {
public:
    vector<int> constructTransformedArray(vector<int>& nums) {
        vector<int> res(nums.begin(), nums.end());
        int n = nums.size();
        for(int i = 0; i < n; i++) {
            int finalIdx;
            int toCommit = nums[i];
            if (toCommit < 0) {
                finalIdx = i + (toCommit % n);
                if (finalIdx < 0) {
                    res[i] = nums[n+finalIdx];
                } else {
                    res[i] = nums[finalIdx];
                }
            } else {
                finalIdx = i + toCommit;
                    res[i] = nums[finalIdx %(n)];
            }
        }
        return res;
    }
};
```

Listing 25: Circular Array Simulation

25 Minimum Removals to Balance Array

25.1 Problem Statement

You are given an integer array `nums` and an integer k. An array is considered **balanced** if the maximum element is at most k times the minimum element:

$$\max(A) \leq \min(A) \cdot k$$

Return the **minimum number of elements** to remove from `nums` so that the remaining array is balanced.

Example:

$$\text{Input: } [1, 6, 2, 9],\ k = 3$$

$$\text{Output: } 2$$

Explanation: Removing 1 and 9 leaves $[2, 6]$. Here $\max(6) \leq \min(2) \cdot 3$.

Problem Link: [23]

25.2 Theoretical Approach

Sorting Strategy

The "balanced" property relies solely on the values of the elements, not their indices. By **sorting** the array first, any valid balanced subset forms a contiguous subarray. This simplifies the problem from finding an arbitrary subset to finding a range $[i, j]$ such that:

$$nums[j] \leq nums[i] \cdot k$$

Binary Search (Upper Bound)

We iterate through the sorted array, treating each index i as the start of the valid subarray (the minimum element).

1. Fix minimum: $min\_val = nums[i]$.
2. Calculate allowed maximum: $limit = min\_val \cdot k$.
3. Find the end of the range: Use `std::upper_bound` to find the first element strictly greater than $limit$.
4. Calculate cost: The number of removals corresponds to the elements outside this range (indices $< i$ and indices $\geq found$).

25.3 Complexity Analysis

Let N be the number of elements.

Time Complexity

1. **Sorting:** Takes $O(N \log N)$.
2. **Search:** We iterate N times. Each iteration performs a binary search taking $O(\log N)$.

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

Space Complexity

The space complexity is determined by the sorting implementation (stack depth).

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

[23] https://leetcode.com/problems/minimum-removals-to-balance-array/

25.4 Implementation

```cpp
class Solution {
public:
    int minRemoval(vector<int>& nums, int k) {
        sort(nums.begin(), nums.end());
        int n = nums.size();
        int minRemoves = INT_MAX;

        for(int i = 0; i < n; i++) {
            long long limit = (long long) nums[i] * k;

            // Find boundary where condition breaks
            auto it = upper_bound(nums.begin() + i, nums.end(), limit)
;

            // Removals = elements to the left (i) + elements to the
right
            int rightRemoves = distance(it, nums.end());
            minRemoves = min(minRemoves, i + rightRemoves);
        }
        return minRemoves;
    }
};
```

Listing 26: Sort + Binary Search

26 Search in Rotated Sorted Array

26.1 Problem Statement

There is an integer array `nums` sorted in ascending order (with distinct values) that has been rotated at an unknown pivot index. Given the array `nums` and an integer `target`, return the index of `target` if it is in `nums`, or -1 if it is not. The algorithm must run in $O(\log N)$ time.

Example:

$$\text{Input: } [4,5,6,7,0,1,2],\ \text{target} = 0$$

$$\text{Output: } 4$$

Problem Link: [24]

26.2 Theoretical Approach

Binary Search with Half-Sorted Property

In a rotated sorted array, picking any middle element divides the array into two halves. The key property is that **one of these two halves must be sorted**.

We compare `nums[mid]` with boundaries to determine which side is monotonic. Your specific implementation handles three explicit cases:

1. **Case 1: Left Side Sorted** ($nums[mid] > nums[r]$)

[24] https://leetcode.com/problems/search-in-rotated-sorted-array/

- The pivot is in the right half. The left half $[l, mid]$ is strictly increasing.
- We check if the target lies within this sorted range: $nums[l] \leq target < nums[mid]$.

2. **Case 2: Right Side Sorted** $(nums[mid] < nums[l])$
   - The pivot is in the left half. The right half $[mid, r]$ is strictly increasing.
   - We check if the target lies within this sorted range: $nums[mid] < target \leq nums[r]$.
3. **Case 3: Fully Sorted** $(nums[mid] \geq nums[l] \wedge nums[mid] \leq nums[r])$
   - The current subarray $[l, r]$ has no rotation. Standard Binary Search applies.

26.3 Complexity Analysis

Let N be the size of the array.

Time Complexity

Like standard binary search, we halve the search space at every step.

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

Space Complexity

The iterative approach requires constant extra space.

$$S(N) = O(1)$$

26.4 Implementation

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int l = 0, r = nums.size()-1;
        while (l <= r) {
            int mid = l + (r - l) / 2;
            if (nums[mid] == target) {
                return mid;
            }
            if (nums[mid] > nums[r]) {
                // right half sorted (pivot in right)
                // left half sorted
                if (target < nums[mid] && target >= nums[l]) {
                    r = mid - 1;
                } else {
                    l = mid + 1;
                }
            } else if(nums[mid] < nums[l]) {
                // pivot in left, right half sorted
                if (target > nums[mid] && target <= nums[r]) {
                    l = mid + 1;
                } else {
                    r = mid - 1;
                }
            } else if (nums[mid] >= nums[l] && nums[mid] <= nums[r]) {
                // all array sorted
                if (nums[mid] < target) {
                    l = mid + 1;
                } else{
                    r = mid - 1;
                }
            }
        }
        return -1;
    }
};
```

Listing 27: Binary Search on Rotated Array