Code – 347. Top K Frequent Elements

Medium

LeetCode leetcode.com/problems/top-k-frequent-elements

Code (2)

Time: **O(n log k)** Space: **O(n)**

```
vector<int> topKFrequent(vector<int>& nums, int k) {
    // 1. Create the number's frequency map
    // O(n)
    unordered_map<int, int> freq;
    for (const auto& num : nums) {
        freq[num] += 1;
    }
    // 2. Create the min heap with priority queue
    // O(n log k)
    priority_queue<pair<int, int>, vector<pair<int,int>>, greater<>> minHeap;
    for (const auto& [num, count] : freq) {
        minHeap.push({count, num});
        if (minHeap.size() > k) minHeap.pop();
    }
    // 3. build the result
    vector<int> result;
    while (!minHeap.empty()) {
        auto num = minHeap.top().second;
        minHeap.pop();
        result.push_back(num);
    }
    return result;
}
```

Solution – 347. Top K Frequent Elements

Medium

LeetCode leetcode.com/problems/top-k-frequent-elements

Solution (3) - hashmap + bucket sort

- Go over the array, count the numbers and store them in an *unordered_map*

  Example:

  `nums = [1,1,1,2,2,3], k = 2`

  `freq[1] = 3`

  `freq[2] = 2`

  ...

- Create buckets for each frequency and add the corresponding numbers:

  `bucket[1] = [3]` → *3 only appears once in nums*

  `bucket[2] = [2]` → *2 appears twice*

  `bucket[3] = [1]` → *1 appears three times*

- Go over each bucket, add to the result and return it

Code – 347. Top K Frequent Elements

Medium

LeetCode leetcode.com/problems/top-k-frequent-elements

Code (3)

Time: **O(n)** Space: **O(n)**

```
vector<int> topKFrequent(vector<int>& nums, int k) {
    // Create the number's frequency map
    unordered_map<int, int> freq;
    for (const auto& num : nums) {
        freq[num]++;
    }
    // create the buckets
    // e.g. [[1,2,3],[4,5,6]] ...
    vector<vector<int>> buckets(nums.size() + 1);
    for (const auto& [num, count] : freq) {
        buckets[count].push_back(num);
    }
    // go over each bucket to build the result
    vector<int> result;
    for (int i = buckets.size() - 1; i >= 0; --i) {
        for (const auto& num : buckets[i]) {
            result.push_back(num);
            if (result.size() == k) return result;
        }
    }
    return result;
}
```

Problem – 347. Top K Frequent Elements

Medium

LeetCode leetcode.com/problems/top-k-frequent-elements

Some considerations

- Theoretically, bucket sort should be the fastest solution **O(n) < O(n log k)**
- In practice, min heap end up being faster:

  - fewer allocations: priority_queue stores flat pairs rather than inner vectors

  - better cache locality: heap is built over a single array (binary heap)

  - if **k** is small, heap touches fewer elements

MATRIX

Problem – 73. Set Matrix Zeroes

Medium

LeetCode leetcode.com/problems/set-matrix-zeroes

Problem

- You are given a **matrix m x n**
- When an element in the matrix is **0**, set the whole column and row to **zero**
- **You must do it in place**

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

Solution – 73. Set Matrix Zeroes

Medium

LeetCode leetcode.com/problems/set-matrix-zeroes

Solution

- Iterate through the matrix top-down
- For each column (from col = 1 to n – 1), when you **find 0**, set:

  matrix[row][0] = 0 → marks that the row should be zeroed

  matrix[0][col] = 0 → marks that the col should be zeroed
- If **matrix[row][0] == 0**, set a variable **col0 = true** to remember if column 0 must be zeroed later
- Iterate bottom-top, right-left
- For each cell, if **matrix[row][0] == 0** or **matrix[0][col] == 0** set **matrix[row][col] = 0**
- It must be bottom-top, right-left to not set the first row to zero first
- Also check if col0 is true. If true, set the first column to zero:

  matrix[row][0] = 0

Code – 73. Set Matrix Zeroes

Medium

LeetCode leetcode.com/problems/set-matrix-zeroes

Code

Time: **O(m × n)** Space: **O(1)** where m is the number of rows and n the number of columns. As this is an in-place implementation, there is no additional space being allocated, therefore space complexity is O(1)

```
void setZeroes(vector<vector<int>>& matrix) {
    int m = matrix.size();
    int n = matrix[0].size();
    bool col0 = false;

    for (int row = 0; row < m; ++row) {
        if (matrix[row][0] == 0) col0 = true;
        for (int col = 1; col < n; ++col) {
            if (matrix[row][col] == 0) {
                matrix[row][0] = 0;
                matrix[0][col] = 0;
            }
        }
    }

    for (int row = m - 1; row >= 0; --row) {
        for (int col = n - 1; col >= 1; --col) {
            if (matrix[row][0] == 0 || matrix[0][col] == 0) {
                matrix[row][col] = 0;
            }
        }
        if (col0) {
            matrix[row][0] = 0;
        }
    }
}
```

Problem – 54. Spiral Matrix

Medium

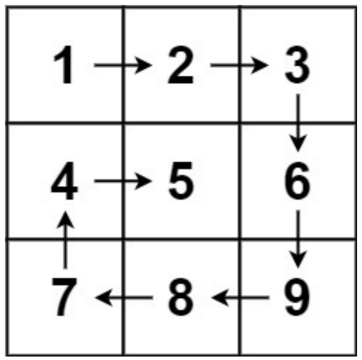
LeetCode leetcode.com/problems/spiral-matrix

Problem

- You are given a matrix **m x n**
- Return the elements in a flat array, the same order as the image
- **Example**

  Input: matrix = `[[1,2,3],[4,5,6],[7,8,9]]`

  Output: matrix = `[1,2,3,6,9,8,7,4,5]`



Solution – 54. Spiral Matrix

Medium

LeetCode leetcode.com/problems/spiral-matrix

Solution

- **Traverse the matrix in spiral order following four directions**: right across top row, down the right column, left across bottom row, and up the left column, then repeat inward
- **Maintain four boundary variables that shrink after each direction**: rowStart, rowEnd, colStart, and colEnd get updated after completing each side of the spiral to move the boundaries inward
- **Boundary handling is the main challenge**: It's easy to introduce bugs when determining when to stop traversal or when to skip certain directions, requiring careful condition checks
- **Non-square matrices require special boundary checks**: The conditional statements if (`rowStart <= rowEnd`) and if (`colStart <= colEnd`) prevent adding duplicate elements when dealing with rectangular matrices or edge cases like single rows/columns

Code – 54. Spiral Matrix

Medium

LeetCode leetcode.com/problems/spiral-matrix

Code

Time: **O(m × n)** Space: **O(1)**

```
vector<int> spiralOrder(vector<vector<int>>& matrix) {
    vector<int> result;
    int rowStart = 0;
    int colStart = 0;
    int colEnd = matrix[0].size() - 1;
    int rowEnd = matrix.size() - 1;

    while(rowStart <= rowEnd && colStart <= colEnd) {
        for (int col = colStart; col <= colEnd; ++col) {
            result.push_back(matrix[rowStart][col]);
        }
        ++rowStart;

        for (int row = rowStart; row <= rowEnd; ++row) {
            result.push_back(matrix[row][colEnd]);
        }
        --colEnd;

        // if matrix is not square condition
        if (rowStart <= rowEnd) {
            for (int col = colEnd; col >= colStart; --col) {
                result.push_back(matrix[rowEnd][col]);
            }
            --rowEnd;
        }


        if (colStart <= colEnd) {
            for (int row = rowEnd; row >= rowStart; --row) {
                result.push_back(matrix[row][colStart]);
            }
            colStart++;
        }
    }
    return result;
}
```

DYNAMIC PROGRAMMING

Dynamic Programming

Dynamic Programming (DP) is an algorithm technique used to solve problems that can be broken down into **simpler, overlapping subproblems.**

Key Concepts of Dynamic Programming

- **Overlapping subproblems**: a problem has overlapping subproblems if it can be broken down into subproblems.
- **Memoization (Top-Down Approach)**: store the results in a cache (typically a dictionary or array) to avoid recalculation – recursion and caching approach.
- **Tabulation (Bottom-Up Approach)**: first solve all possible subproblems iteratively, and store them in a table.

Common Patterns in Dynamic Programming

- **Toy example (Fibonacci):** Climbing Stairs, N-th Tribonacci Number, Perfect Squares
- **Constant Transition:** Min Cost Climbing Stairs, House Robber, Decode Ways, Minimum Cost For Tickets, Solving Questions With Brainpower
- **Grid:** Unique Paths, Unique Paths II, Minimum Path Sum, Count Square Submatrices with All Ones, Maximal Square, Dungeon Game
- **Dual-Sequence:** Longest Common Subsequence, Uncrossed Lines, Minimum ASCII Delete Sum for Two Strings, Edit Distance, Distinct Subsequences, Shortest Common Supersequence
- **Interval:** Longest Palindromic Subsequence, Stone Game VII, Palindromic Substrings, Minimum Cost Tree From Leaf Values, Burst Balloons, Strange Printer
- **Longest Increasing Subsequence:** Count Number of Teams, Longest Increasing Subsequence, Partition Array for Maximum Sum, Largest Sum of Averages, Filling Bookcase Shelves
- **Knapsack:** Partition Equal Subset Sum, Number of Dice Rolls With Target Sum, Combination Sum IV, Ones and Zeroes, Coin Change, Coin Change II, Target Sum, Last Stone Weight II, Profitable Schemes
- **Topological Sort on Graphs:** Longest Increasing Path in a Matrix, Longest String Chain, Course Schedule III
- **DP on Trees:** House Robber III, Binary Tree Cameras
- **Other problems:** 2 Keys Keyboard, Word Break, Minimum Number of Removals to Make Mountain Array, Out of Boundary Paths

Credits
https://www.youtube.com/watch?v=9k31KcQmS_U
https://algo.monster/problems/dp-list

Dynamic Programming – Example – Fibonacci Sequence

Naive Recursive Approach $O(2^n)$

```
int fib(int n) {
    if (n <= 1) {
        return n;
    }
    return fib(n - 1) + fib(n - 2);
}
```

Memoization (Top-Down DP) $O(n)$

```
std::unordered_map<int, int> memo;

int fib(int n) {
    if (n <= 1) {
        return n;
    }
    if (memo.find(n) != memo.end()) {
        return memo[n];
    }
    memo[n] = fib(n - 1) + fib(n - 2);
    return memo[n];
}
```

Tabulation (Bottom-up DP) $O(n)$

```
int fib(int n) {
    if (n <= 1) {
        return n;
    }
    int dp[n + 1];
    dp[0] = 0;
    dp[1] = 1;
    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i - 1] + dp[i - 2];
    }
    return dp[n];
}
```