WEEK3

Contents

1. Binary Tree Paths - Easy

2. Maximum Average Subarray I - Easy

3. Product of Array Except Self - Medium

4. Find the Duplicate Number - Medium

5. Find All Duplicates in an Array - Medium

6. Set Matrix Zeroes - Medium

7. Spiral Matrix - Medium

8. Rotate Image - Medium

9. Word Search - Medium

10. Longest Consecutive Sequence - Medium

11. Letter Case Permutation - Medium

12. Subsets - Medium

13. Subsets II - Medium

14. Permutations - Medium

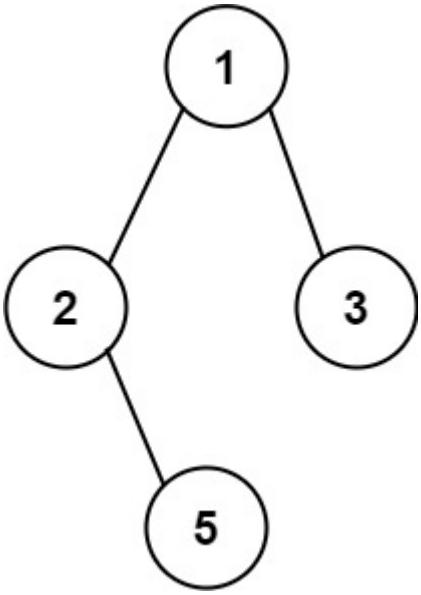
15. Permutations II - Medium

Binary Tree Paths

Given the `root` of a binary tree, return *all root-to-leaf paths in **any order***.

A **leaf** is a node with no children.

Example 1:



Input: `root = [1,2,3,null,5]` **Output:** `["1->2->5","1->3"]`

Example 2:

Input: `root = [1]` **Output:** `["1"]`

Constraints:

- The number of nodes in the tree is in the range `[1, 100]`.
- `-100 <= Node.val <= 100`

/** * 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 binaryTreePaths(TreeNode* root) { } };

Maximum Average Subarray I

You are given an integer array `nums` consisting of `n` elements, and an integer `k`.

Find a contiguous subarray whose **length is equal to** `k` that has the maximum average value and return *this value*. Any answer with a calculation error less than 10^{-5} will be accepted.

Example 1:

Input: `nums = [1,12,-5,-6,50,3], k = 4` **Output:** `12.75000` **Explanation:** `Maximum average is (12 - 5 - 6 + 50) / 4 = 51 / 4 = 12.75`

Example 2:

Input: `nums = [5], k = 1` **Output:** `5.00000`

Constraints:

- `n == nums.length`
- `1 <= k <= n <= 10^5`
- `-10^4 <= nums[i] <= 10^4`

class Solution { public: double findMaxAverage(vector& nums, int k) { } };

Product of Array Except Self

Given an integer array `nums`, return *an array* `answer` *such that* `answer[i]` *is equal to the product of all the elements of* `nums` *except* `nums[i]`.

The product of any prefix or suffix of `nums` is **guaranteed** to fit in a **32-bit** integer.

You must write an algorithm that runs in `O(n)` time and without using the division operation.

Example 1:

Input: `nums = [1,2,3,4]` **Output:** `[24,12,8,6]`

Example 2:

Input: `nums = [-1,1,0,-3,3]` **Output:** `[0,0,9,0,0]`

Constraints:

- `2 <= nums.length <= 10^5`
- `-30 <= nums[i] <= 30`
- The product of any prefix or suffix of `nums` is **guaranteed** to fit in a **32-bit** integer.

Follow up: Can you solve the problem in `O(1)` extra space complexity? (The output array **does not** count as extra space for space complexity analysis.)

class Solution { public: vector productExceptSelf(vector& nums) { } };

Find the Duplicate Number

Given an array of integers `nums` containing `n + 1` integers where each integer is in the range `[1, n]` inclusive.

There is only **one repeated number** in `nums`, return *this repeated number*.

You must solve the problem **without** modifying the array `nums` and uses only constant extra space.

Example 1:

Input: `nums = [1,3,4,2,2]` **Output:** `2`

Example 2:

Input: `nums = [3,1,3,4,2]` **Output:** `3`

Example 3:

Input: `nums = [3,3,3,3,3]` **Output:** `3`

Constraints:

- `1 <= n <= 10^5`
- `nums.length == n + 1`
- `1 <= nums[i] <= n`
- All the integers in `nums` appear only **once** except for **precisely one integer** which appears **two or more** times.

Follow up:

- How can we prove that at least one duplicate number must exist in `nums`?
- Can you solve the problem in linear runtime complexity?

class Solution { public: int findDuplicate(vector& nums) { } };

Find All Duplicates in an Array

Given an integer array `nums` of length `n` where all the integers of `nums` are in the range `[1, n]` and each integer appears **once** or **twice**, return *an array of all the integers that appears **twice***.

You must write an algorithm that runs in `O(n)` time and uses only *constant* auxiliary space, excluding the space needed to store the output

Example 1:

Input: `nums = [4,3,2,7,8,2,3,1]` **Output:** `[2,3]`

Example 2:

Input: `nums = [1,1,2]` **Output:** `[1]`

Example 3:

Input: `nums = [1]` **Output:** `[]`

Constraints:

- `n == nums.length`
- `1 <= n <= 10^5`
- `1 <= nums[i] <= n`
- Each element in `nums` appears **once** or **twice**.

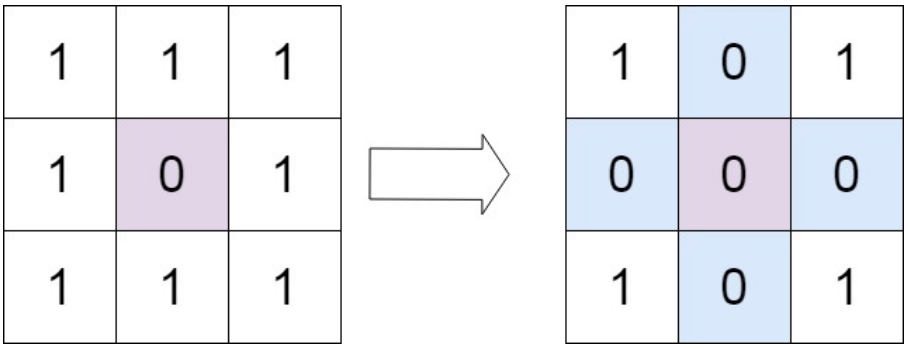
class Solution { public: vector findDuplicates(vector& nums) { } };

Set Matrix Zeroes

Given an `m x n` integer matrix `matrix`, if an element is `0`, set its entire row and column to `0`'s.

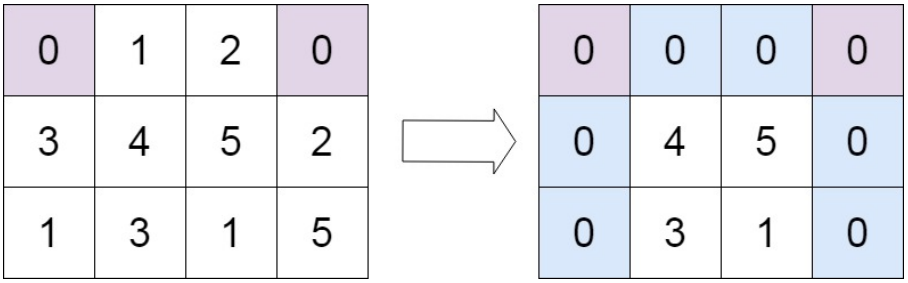
You must do it in place.

Example 1:



Input: `matrix = [[1,1,1],[1,0,1],[1,1,1]]` **Output:** `[[1,0,1],[0,0,0],[1,0,1]]`

Example 2:



Input: `matrix = [[0,1,2,0],[3,4,5,2],[1,3,1,5]]` **Output:** `[[0,0,0,0],[0,4,5,0],[0,3,1,0]]`

Constraints:

- `m == matrix.length`
- `n == matrix[0].length`
- `1 <= m, n <= 200`
- -2^{31} `<= matrix[i][j] <=` $2^{31} - 1$

Follow up:

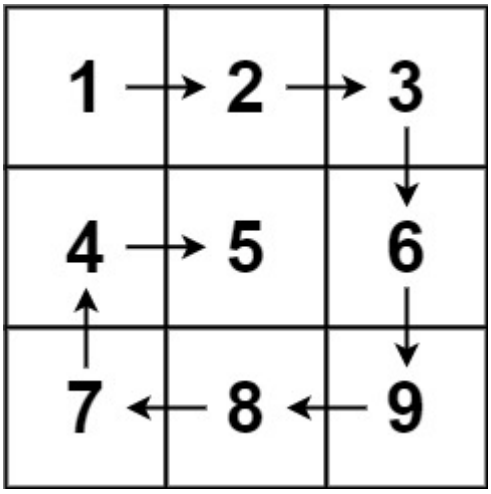
- A straightforward solution using `O(mn)` space is probably a bad idea.
- A simple improvement uses `O(m + n)` space, but still not the best solution.
- Could you devise a constant space solution?

class Solution { public: void setZeroes(vector>& matrix) { } };

Spiral Matrix

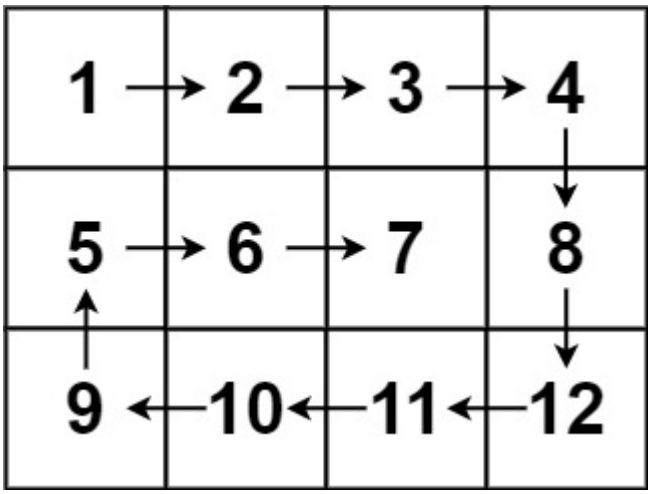
Given an `m x n matrix`, return *all elements of the* `matrix` *in spiral order*.

Example 1:



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

Example 2:



Input: `matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]` **Output:** `[1,2,3,4,8,12,11,10,9,5,6,7]`

Constraints:

- `m == matrix.length`
- `n == matrix[i].length`
- `1 <= m, n <= 10`
- `-100 <= matrix[i][j] <= 100`

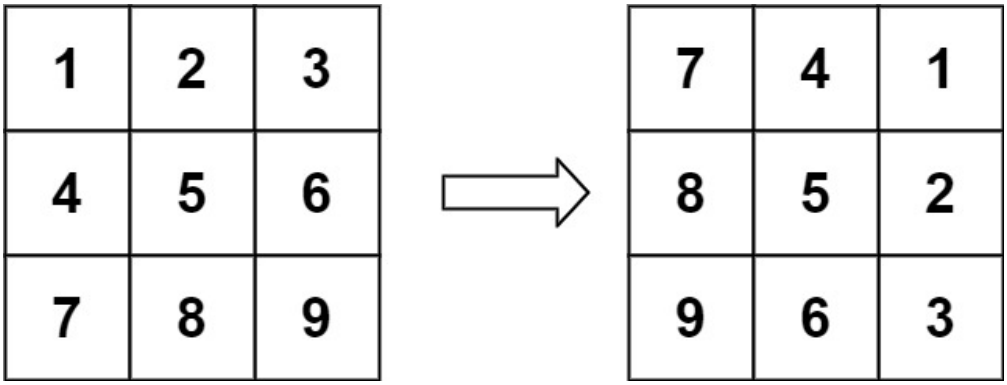
class Solution { public: vector spiralOrder(vector>& matrix) { } };

Rotate Image

You are given an `n x n` 2D `matrix` representing an image, rotate the image by **90** degrees (clockwise).

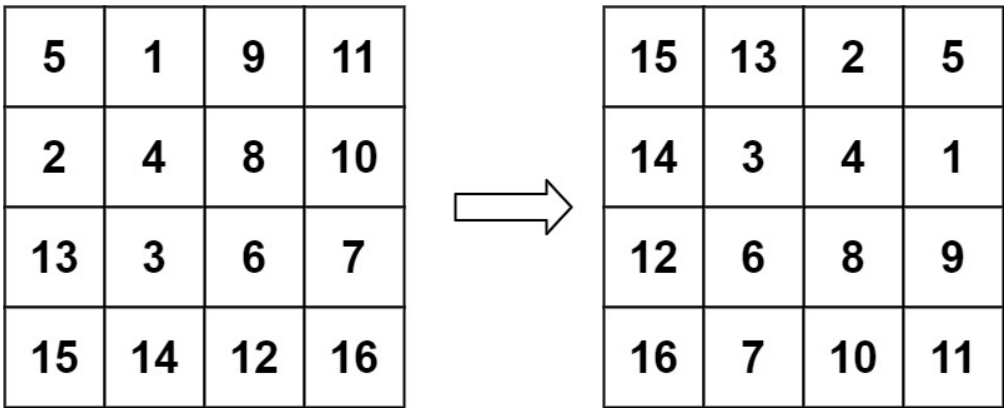
You have to rotate the image **in-place**, which means you have to modify the input 2D matrix directly. **DO NOT** allocate another 2D matrix and do the rotation.

Example 1:



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

Example 2:



Input: `matrix = [[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,12,16]]` **Output:** `[[15,13,2,5],[14,3,4,1],[12,6,8,9],[16,7,10,11]]`

Constraints:

- `n == matrix.length == matrix[i].length`
- `1 <= n <= 20`
- `-1000 <= matrix[i][j] <= 1000`

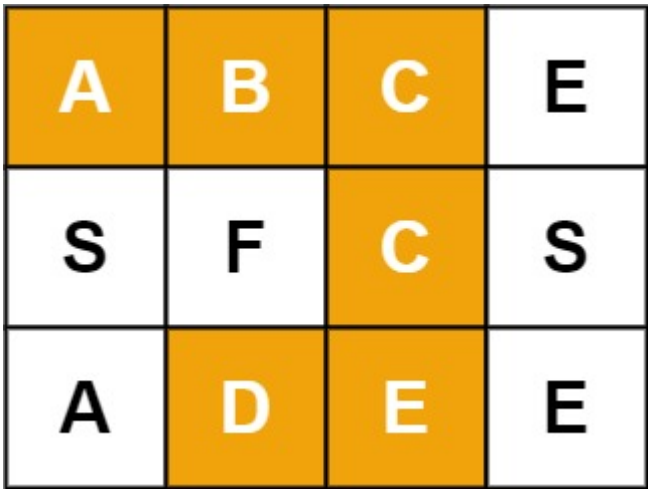
class Solution { public: void rotate(vector>& matrix) { } };

Word Search

Given an `m x n` grid of characters `board` and a string `word`, return `true` *if* `word` *exists in the grid*.

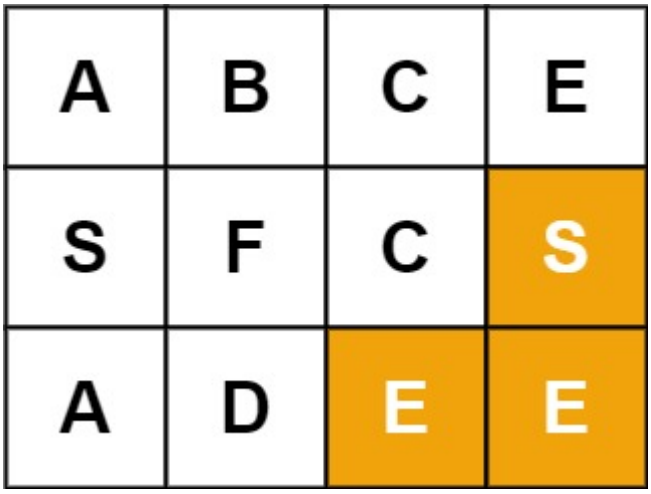
The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.

Example 1:



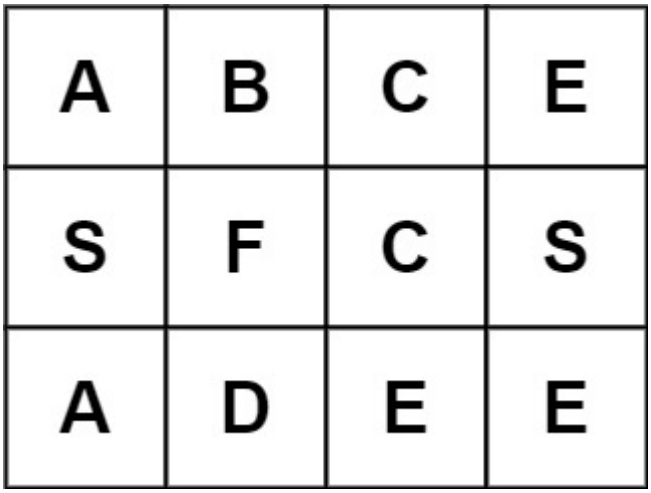
Input: `board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCCED"` **Output:** `true`

Example 2:



Input: `board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "SEE"` **Output:** `true`

Example 3:



Input: `board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCB"` **Output:** `false`

Constraints:

- `m == board.length`
- `n = board[i].length`
- `1 <= m, n <= 6`
- `1 <= word.length <= 15`
- `board` and `word` consists of only lowercase and uppercase English letters.

Follow up: Could you use search pruning to make your solution faster with a larger `board`?

class Solution { public: bool exist(vector>& board, string word) { } };

Longest Consecutive Sequence

Given an unsorted array of integers `nums`, return *the length of the longest consecutive elements sequence.*

You must write an algorithm that runs in `O(n)` time.

Example 1:

Input: `nums = [100,4,200,1,3,2]` **Output:** `4` **Explanation:** `The longest consecutive elements sequence is [1, 2, 3, 4]. Therefore its length is 4.`

Example 2:

Input: `nums = [0,3,7,2,5,8,4,6,0,1]` **Output:** `9`

Constraints:

- `0 <= nums.length <= 10^5`
- `-10^9 <= nums[i] <= 10^9`

class Solution { public: int longestConsecutive(vector& nums) { } };

Letter Case Permutation

Given a string s, you can transform every letter individually to be lowercase or uppercase to create another string.

Return *a list of all possible strings we could create*. Return the output in **any order**.

Example 1:

Input: `s = "a1b2"` **Output:** `["a1b2","a1B2","A1b2","A1B2"]`

Example 2:

Input: `s = "3z4"` **Output:** `["3z4","3Z4"]`

Constraints:

- `1 <= s.length <= 12`
- s consists of lowercase English letters, uppercase English letters, and digits.

class Solution { public: vector letterCasePermutation(string s) { } };

Subsets

Given an integer array `nums` of **unique** elements, return *all possible subsets (the power set)*.

The solution set **must not** contain duplicate subsets. Return the solution in **any order**.

Example 1:

Input: `nums = [1,2,3]` **Output:** `[[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]`

Example 2:

Input: `nums = [0]` **Output:** `[[],[0]]`

Constraints:

- `1 <= nums.length <= 10`
- `-10 <= nums[i] <= 10`
- All the numbers of `nums` are **unique**.

class Solution { public: vector> subsets(vector& nums) { } };

Subsets II

Given an integer array `nums` that may contain duplicates, return *all possible subsets (the power set)*.

The solution set **must not** contain duplicate subsets. Return the solution in **any order**.

Example 1:

Input: `nums = [1,2,2]` **Output:** `[[],[1],[1,2],[1,2,2],[2],[2,2]]`

Example 2:

Input: `nums = [0]` **Output:** `[[],[0]]`

Constraints:

- `1 <= nums.length <= 10`
- `-10 <= nums[i] <= 10`

class Solution { public: vector> subsetsWithDup(vector& nums) { } };

Permutations

Given an array `nums` of distinct integers, return *all the possible permutations*. You can return the answer in **any order**.

Example 1:

Input: `nums = [1,2,3]` **Output:** `[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]`

Example 2:

Input: `nums = [0,1]` **Output:** `[[0,1],[1,0]]`

Example 3:

Input: `nums = [1]` **Output:** `[[1]]`

Constraints:

- `1 <= nums.length <= 6`
- `-10 <= nums[i] <= 10`
- All the integers of `nums` are **unique**.

class Solution { public: vector> permute(vector& nums) { } };

Permutations II

Given a collection of numbers, `nums`, that might contain duplicates, return *all possible unique permutations* ***in any order***.

Example 1:

Input: `nums = [1,1,2]` **Output:** `[[1,1,2], [1,2,1], [2,1,1]]`

Example 2:

Input: `nums = [1,2,3]` **Output:** `[[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]`

Constraints:

- `1 <= nums.length <= 8`
- `-10 <= nums[i] <= 10`

class Solution { public: vector> permuteUnique(vector& nums) { } };