1. 01_Arrays_&_Hashing/01_Contains_Duplicate/0217-contains-duplicate.rs

```
/*
Problem: LeetCode 217 - Contains Duplicate

Key Idea:
The key idea is to use a hashset to keep track of the numbers we have seen so far. If we encounter a number that is already in the hashset, we know there is a duplicate.

Approach:
1. Create an empty hashset to store the numbers and their occurrences.
2. Iterate through the input array:
    - For each number `num`, check if it exists in the hashset. If yes, return `true` as a duplicate is found.
   - Otherwise, insert `num` into the hashset.
3. If no duplicate is found during the iteration, return `false`.

Time Complexity:
O(n), where n is the number of elements in the input array. We iterate through the array once and each hashset lookup takes O(1) time on average.

Space Complexity:
O(n), as the worst case is when all elements in the array are distinct and we store all of them in the hashset.
*/

use std::collections::HashSet;

impl Solution {
    pub fn contains_duplicate(nums: Vec<i32>) -> bool {
        let mut seen_nums = HashSet::new();

        for num in nums {
            if seen_nums.contains(&num) {
                return true;
            }
            seen_nums.insert(num);
        }

        false
    }
}
```

2. 01_Arrays_&_Hashing/02_Valid_Anagram/0242-valid-anagram.rs

```
/*
Problem: LeetCode 242 - Valid Anagram

Key Idea:
The key idea is to compare the frequencies of characters in two strings. An anagram will have the same frequency of characters.

Approach:
1. Create two arrays, `freq_s` and `freq_t`, of size 26 to represent the frequencies of characters in the input strings `s` and `t`.
2. Iterate through the characters of `s` and `t`:
   - Increment the corresponding frequency in `freq_s` for characters in `s`.
   - Decrement the corresponding frequency in `freq_t` for characters in `t`.
3. After the iteration, if `freq_s` and `freq_t` are equal, then `s` and `t` are valid anagrams.

Time Complexity:
O(n), where n is the length of the input strings `s` and `t`. We iterate through the strings once.

Space Complexity:
O(1), as the arrays `freq_s` and `freq_t` have constant size (26) regardless of the input size.
*/

impl Solution {
    pub fn is_anagram(s: String, t: String) -> bool {
        if s.len() != t.len() {
            return false;
        }

        let mut freq_s = [0; 26];
        let mut freq_t = [0; 26];

        for ch in s.chars() {
            freq_s[(ch as u8 - b'a') as usize] += 1;
        }

        for ch in t.chars() {
            freq_t[(ch as u8 - b'a') as usize] += 1;
        }

        freq_s == freq_t
    }
}
```

3. 01_Arrays_&_Hashing/03_Two_Sum/0001-two-sum.rs

```
/*
Problem: LeetCode 1 - Two Sum

Key Idea:
The key idea is to use a hashmap to store the elements as we traverse through the input array. For each element, we check if the complement (target - current element) is already in the hashmap. If it is, we've found a solution; if not, we add the current element to the hashmap for future reference.

Approach:
1. Create an empty hashmap to store the elements.
2. Iterate through the input array:
   - For each element `num` at index `i`, calculate the complement `complement = target - num`.
   - Check if `complement` exists in the hashmap. If yes, return `[index_of_complement, i]`.
   - Otherwise, add `num` to the hashmap with its index as the value.
3. If no solution is found, return an empty vector.

Time Complexity:
O(n), where n is the number of elements in the input array. We iterate through the array once, and each lookup in the hashmap takes O(1) time on average.

Space Complexity:
O(n), as the hashmap can contain up to n elements in the worst case.
*/

use std::collections::HashMap;

impl Solution {
    pub fn two_sum(nums: Vec<i32>, target: i32) -> Vec<i32> {
        let mut num_indices: HashMap<i32, i32> = HashMap::new();

        for (i, &num) in nums.iter().enumerate() {
            let complement = target - num;
            if let Some(&index) = num_indices.get(&complement) {
                return vec![index, i as i32];
            }
            num_indices.insert(num, i as i32);
        }

        vec![]
    }
}
```

4. 01_Arrays_&_Hashing/04_Group_Anagrams/0049-group-anagrams.rs

```
/*
Problem: LeetCode 49 - Group Anagrams

Key Idea:
The key idea is to recognize that anagrams will have the same sorted characters. We can use a hashmap to group words that share the same sorted characters.

Approach:
1. Create an empty hashmap where the keys are the sorted characters of words, and the values are vectors of words.
2. Iterate through the input list of words:
   - Sort the characters of the word to obtain a key.
    - Check if the key is present in the hashmap. If it is, add the word to the corresponding vector. If not, create a new vector with the word and insert it into the hashmap with the key.
3. After iterating through all words, collect the values of the hashmap into a result vector.

Time Complexity:
O(n * k * log k), where n is the number of words and k is the maximum length of a word. Sorting each word takes O(k * log k) time, and we perform this operation for each of the n words.

Space Complexity:
O(n * k), where n is the number of words and k is the maximum length of a word. In the worst case, each word has distinct sorted characters, so we have n keys in the hashmap.
*/

use std::collections::HashMap;

impl Solution {
    pub fn group_anagrams(strs: Vec<String>) -> Vec<Vec<String>> {
        // Keys are sorted character vectors and the values are vectors of strings.
        let mut anagram_groups: HashMap<Vec<u8>, Vec<String>> = HashMap::new();

        for s in strs {
            // Convert the string's characters to bytes and collect them into a vector.
            let mut sorted_chars = s.bytes().collect::<Vec<u8>>();
            // Sort the vector of bytes to create a unique key for anagrams.
            sorted_chars.sort();

            // Use the sorted key as an entry in the HashMap and push the original string into the group.
              // If the key doesn't exist in the HashMap, the entry will be created with an empty vector.
            anagram_groups.entry(sorted_chars).or_default().push(s);
        }

        // Transform the HashMap into an iterator of (sorted_key, group) pairs.
        // Then, extract only the group from each pair using the map() function.
        // Finally, collect the groups into a vector of vectors of strings.
        anagram_groups.into_iter().map(|(_, group)| group).collect()
    }
}

/*
// Alternative solution

use std::collections::HashMap;

impl Solution {
    pub fn group_anagrams(strs: Vec<String>) -> Vec<Vec<String>> {
        // Create a HashMap to store anagrams grouped by sorted strings
```

```
        let mut anagrams: HashMap<String, Vec<String>> = HashMap::new();

        // Iterate through each input string
        for s in strs {
            // Convert the string to a sorted string
            let mut sorted_str: Vec<char> = s.chars().collect();
            sorted_str.sort();
            let sorted_str: String = sorted_str.into_iter().collect();

            // Insert the sorted string as the key in the HashMap,
            // and insert the input string into the corresponding value list
            anagrams.entry(sorted_str).or_insert(vec![]).push(s);
        }

        // Collect and return the grouped anagrams as a Vec<Vec<String>>
        anagrams.values().cloned().collect()
    }
}
*/
```

5. 01_Arrays_&_Hashing/05_Top_K_Frequent_Elements/0347-top-k-frequent-element

```
/*
Problem: LeetCode 347 - Top K Frequent Elements

Key Idea:
The key idea is to use a combination of a hashmap to count the frequency of each element and a min heap (priority queue) to efficiently maintain the top k frequent elements.

Approach:
1. Create a hashmap to count the frequency of each element in the input array.
2. Create a min heap (priority queue) of size k to store pairs of elements and their frequencies.
3. Iterate through the hashmap:
   - For each element, if the heap size is less than k, insert it into the heap.
   - If the heap size is already k, compare the element's frequency with the smallest frequency in the heap. If the element's frequency is greater, pop the smallest element from the heap and insert the current element.
4. Extract the elements from the heap to obtain the top k frequent elements.

Time Complexity:
O(n log k), where n is the number of elements in the input array and k is the given value. Inserting an element into the heap takes O(log k) time, and we perform this operation for each of the n elements.

Space Complexity:
O(n), as we use a hashmap to store the frequency of elements, which can have up to n distinct keys.
*/

use std::cmp::Reverse;
use std::collections::{BinaryHeap, HashMap};

impl Solution {
    pub fn top_k_frequent(nums: Vec<i32>, k: i32) -> Vec<i32> {
        let mut freq_map: HashMap<i32, i32> = HashMap::new();

        // Count the frequency of each element
        for num in &nums {
            *freq_map.entry(*num).or_insert(0) += 1;
        }

        // Create a min heap of size k to store the top k frequent elements
        let mut min_heap: BinaryHeap<Reverse<(i32, i32)>> = BinaryHeap::new();

        // Insert elements into the heap
        for (num, freq) in freq_map.iter() {
            if min_heap.len() < k as usize {
                min_heap.push(Reverse((*freq, *num)));
            } else if freq > &min_heap.peek().unwrap().0 .0 {
                min_heap.pop();
                min_heap.push(Reverse((*freq, *num)));
            }
        }

        // Extract the elements from the heap
        let result: Vec<i32> = min_heap.into_iter().map(|rev| rev.0 .1).collect();

        result
    }
}
```

6. 01_Arrays_&_Hashing/06_Product_of_Array_Except_Self/0238-product-of-array-ex

```
/*
Problem: LeetCode 238 - Product of Array Except Self

Key Idea:
The key idea is to compute the product of all elements to the left of each element and the product of all elements to the right of each element. The final product for each element is the product of these left and right products.

Approach:
1. Create two arrays, `prefix` and `postfix`, of the same size as the input array to store the products of elements to the left and right of each index, respectively.
2. Initialize `prefix[0] = 1` and `postfix[n-1] = 1`, where n is the length of the input array.
3. Populate the `prefix` array by iterating through the input array from left to right. `prefix[i]` will be the product of `prefix[i-1]` and `nums[i-1]`.
4. Populate the `postfix` array by iterating through the input array from right to left. `postfix[i]` will be the product of `postfix[i+1]` and `nums[i+1]`.
5. For each index i, the result for that index is `prefix[i] * postfix[i]`.

Time Complexity:
O(n), where n is the length of the input array. We iterate through the input array three times.

Space Complexity:
O(n), as we use two additional arrays, `prefix` and `postfix`, each of size n.
*/

impl Solution {
    pub fn product_except_self(nums: Vec<i32>) -> Vec<i32> {
        let n = nums.len();
        let mut prefix = vec![1; n];
        let mut postfix = vec![1; n];

        for i in 1..n {
            prefix[i] = prefix[i - 1] * nums[i - 1];
        }

        for i in (0..n - 1).rev() {
            postfix[i] = postfix[i + 1] * nums[i + 1];
        }

        let mut result = vec![1; n];
        for i in 0..n {
            result[i] = prefix[i] * postfix[i];
        }

        result
    }
}
```

7. 01_Arrays_&_Hashing/07_Valid_Sudoku/0036-valid-sudoku.rs

```
/*
Problem: LeetCode 36 - Valid Sudoku

Key Idea:
The key idea is to check whether the rules of a Sudoku are satisfied: no repeated digits in each row, each column, and each of the 9 sub-boxes.

Approach:
1. Create three arrays, `row_sets`, `col_sets`, and `box_sets`, of size 9x9 to store sets of digits encountered in each row, column, and sub-box.
2. Iterate through the Sudoku board:
   - For each cell `(i, j)` with digit `num`:
      - Check if `num` is already in `row_sets[i]`, `col_sets[j]`, or `box_sets[box_index]`. If yes, return `false`.
     - Otherwise, add `num` to `row_sets[i]`, `col_sets[j]`, and `box_sets[box_index]`.
3. If the iteration completes without any violations, return `true`.

Time Complexity:
O(1), as we iterate through the Sudoku board with a constant size of 9x9 cells.

Space Complexity:
O(1), as the sizes of `row_sets`, `col_sets`, and `box_sets` are all fixed and independent of the input size.
*/

impl Solution {
    pub fn is_valid_sudoku(board: Vec<Vec<char>>) -> bool {
        let mut row_sets = vec![vec![false; 9]; 9];
        let mut col_sets = vec![vec![false; 9]; 9];
        let mut box_sets = vec![vec![false; 9]; 9];

        for i in 0..9 {
            for j in 0..9 {
                if board[i][j] != '.' {
                    let num = (board[i][j] as u8 - b'1') as usize;
                    let box_index = (i / 3) * 3 + j / 3;

                    if row_sets[i][num] || col_sets[j][num] || box_sets[box_index][num] {
                        return false;
                    }

                    row_sets[i][num] = true;
                    col_sets[j][num] = true;
                    box_sets[box_index][num] = true;
                }
            }
        }

        true
    }
}
```

8. 01_Arrays_&_Hashing/09_Longest_Consecutive_Sequence/0128-longest-consecut

```
/*
Problem: LeetCode 128 - Longest Consecutive Sequence

Key Idea:
The key idea is to use a hash set to efficiently keep track of the consecutive sequence lengths starting from each number.

Approach:
1. Create a hash set to store the numbers from the input array.
2. Initialize a variable `longest_streak` to track the maximum consecutive sequence length found so far.
3. Iterate through the numbers in the input array:
   - For each number `num`, check if `num - 1` exists in the hash set. If not, it's a potential starting point of a consecutive sequence.
   - While `num + 1` exists in the hash set, increment `num` to extend the sequence.
   - Update `longest_streak` with the maximum of its current value and the length of the current sequence.
4. Return `longest_streak` as the result.

Time Complexity:
O(n), where n is the number of elements in the input array. We iterate through the array once and perform O(1) operations on average for each number.

Space Complexity:
O(n), as the hash set can contain all the elements from the input array.
*/

use std::collections::HashSet;

impl Solution {
    pub fn longest_consecutive(nums: Vec<i32>) -> i32 {
        let num_set: HashSet<i32> = nums.iter().cloned().collect();
        let mut longest_streak: i32 = 0;

        for num in num_set.iter() {
            if !num_set.contains(&(num - 1)) {
                let mut streak = 1;
                let mut current = *num;

                while (num_set.contains(&(current + 1))) {
                    current += 1;
                    streak += 1;
                }

                longest_streak = std::cmp::max(streak, longest_streak);
            }
        }

        longest_streak
    }
}
```

9. 02_Two_Pointers/01_Valid_Palindrome/0125-valid-palindrome.rs

```
/*
Problem: LeetCode 125 - Valid Palindrome

Key Idea:
The key idea is to compare the characters in the string while ignoring non-alphanumeric characters
and considering case-insensitivity.

Approach:
1. Convert the input string to lowercase to handle case-insensitivity.
2. Use two pointers, `left` and `right`, initialized to the start and end of the string.
3. Iterate while `left` is less than `right`:
   - Skip non-alphanumeric characters and increment `left` or decrement `right`.
   - Compare the characters at `left` and `right`. If they are not equal, return `false`.
   - Increment `left` and decrement `right` to continue comparing.
4. If the iteration completes without returning `false`, the string is a valid palindrome.

Time Complexity:
O(n), where n is the length of the input string. We perform a single pass through the string while
comparing characters.

Space Complexity:
O(1), as we use only a constant amount of extra space for pointers and variables.
*/

impl Solution {
    pub fn is_palindrome(s: String) -> bool {
        let chars: Vec<char> = s.chars().collect();
        let mut left = 0;
        let mut right = chars.len() - 1;

        while left < right {
            while left < right && !chars[left].is_ascii_alphanumeric() {
                left += 1;
            }
            while left < right && !chars[right].is_ascii_alphanumeric() {
                right -= 1;
            }

            if chars[left].to_ascii_lowercase() != chars[right].to_ascii_lowercase() {
                return false;
            }

            left += 1;
            right -= 1;
        }

        true
    }
}

/*
// Alternative Solution

pub fn is_palindrome(s: String) -> bool {
    // Step 1: Filter out non-alphanumeric characters and convert to lowercase
    let cleaned: String = s.chars()                                 // Iterate over characters in the
input string
           .filter(|c| c.is_ascii_alphanumeric())                        // Keep only alphanumeric
characters
        .map(|c| c.to_ascii_lowercase())                        // Convert characters to lowercase
```

```
        .collect();                                             // Collect the filtered characters
into a new String

    // Step 2: Create a reversed version of the cleaned string
    let reversed: String = cleaned.chars().rev().collect();  // Reverse the characters and collect
into a String

    // Step 3: Check if the cleaned string is equal to its reversed version
    cleaned == reversed                                         // Compare the cleaned and reversed
strings

    // The result of the comparison is the final result of the function
}
*/
```

10. 02_Two_Pointers/02_Two_Sum_II_-_Input_Array_Is_Sorted/0167-two-sum-ii-input

```
/*
Problem: LeetCode 167 - Two Sum II - Input array is sorted

Key Idea:
The key idea is to use a two-pointer approach to find the two numbers that add up to the target sum.

Approach:
1. Initialize two pointers, `left` at the start of the array and `right` at the end.
2. While `left` is less than `right`:
   - Calculate the sum of the elements at `nums[left]` and `nums[right]`.
    - If the sum is equal to the target, return a vector containing `left + 1` and `right + 1` (1-based indices).
   - If the sum is less than the target, increment `left`.
   - If the sum is greater than the target, decrement `right`.
3. If no solution is found, return an empty vector.

Time Complexity:
O(n), where n is the number of elements in the input array. We perform a single pass through the array using the two-pointer approach.

Space Complexity:
O(1), as we use only a constant amount of extra space for pointers and variables.
*/

impl Solution {
    pub fn two_sum(numbers: Vec<i32>, target: i32) -> Vec<i32> {
        let mut left = 0;
        let mut right = numbers.len() - 1;

        while left < right {
            let sum = numbers[left] + numbers[right];

            if sum == target {
                return vec![(left + 1) as i32, (right + 1) as i32];
            } else if sum < target {
                left += 1;
            } else {
                right -= 1;
            }
        }

        vec![]
    }
}
```

11. 02_Two_Pointers/03_3Sum/0015-3sum.rs

```
/*
Problem: LeetCode 15 - 3Sum

Key Idea:
The key idea is to use a combination of sorting the array and a two-pointer approach to find all unique triplets that sum up to zero.

Approach:
1. Sort the input array in ascending order.
2. Iterate through the array up to the second-to-last element:
    - Initialize two pointers, `left` and `right`, one pointing to the element after the current element and the other pointing to the end of the array.
   - While `left` is less than `right`:
     - Calculate the sum of the current element, `nums[left]`, and `nums[right]`.
      - If the sum is equal to zero, add the triplet `[nums[i], nums[left], nums[right]]` to the result.
     - If the sum is less than zero, increment `left`.
     - If the sum is greater than zero, decrement `right`.
     - Move both `left` and `right` until they are no longer pointing to duplicate elements.
3. Return the result containing all unique triplets.

Time Complexity:
O(n^2), where n is the number of elements in the input array. The sorting step takes O(n log n), and the two-pointer traversal takes O(n^2) in the worst case.

Space Complexity:
O(log n), as the sorting step requires additional space for the call stack due to the recursive nature of the sorting algorithm. The result array also contributes to the space complexity.
*/

impl Solution {
    pub fn three_sum(nums: Vec<i32>) -> Vec<Vec<i32>> {
        let mut result: Vec<Vec<i32>> = Vec::new();
        let n = nums.len();
        let mut nums = nums;

        nums.sort();

        for i in 0..n - 2 {
            if i > 0 && nums[i] == nums[i - 1] {
                continue; // Skip duplicate elements
            }

            let mut left = i + 1;
            let mut right = n - 1;

            while left < right {
                let sum = nums[i] + nums[left] + nums[right];

                if sum == 0 {
                    result.push(vec![nums[i], nums[left], nums[right]]);
                    while left < right && nums[left] == nums[left + 1] {
                        left += 1;
                    }
                    while left < right && nums[right] == nums[right - 1] {
                        right -= 1;
                    }
                    left += 1;
                    right -= 1;
                } else if sum < 0 {
```

```
                    left += 1;
                } else {
                    right -= 1;
                }
            }
        }

        result
    }
}
```

12. 02_Two_Pointers/04_Container_With_Most_Water/0011-container-with-most-wate

```
/*
Problem: LeetCode 11 - Container With Most Water

Key Idea:
The key idea is to use a two-pointer approach to find the container with the most water. The water height of the container is determined by the shorter of the two container walls, and the width is determined by the distance between the two walls.

Approach:
1. Initialize two pointers, `left` at the start of the array and `right` at the end of the array.
2. Initialize a variable `max_area` to track the maximum area found so far.
3. While `left` is less than `right`:
     - Calculate the current container's area using the formula: `area = min(height[left], height[right]) * (right - left)`.
   - Update `max_area` with the maximum of its current value and the current area.
   - Move the pointer pointing to the shorter wall (either `left` or `right`) inward.
4. Return `max_area` as the result.

Time Complexity:
O(n), where n is the number of elements in the input array. We perform a single pass through the array using the two-pointer approach.

Space Complexity:
O(1), as we use only a constant amount of extra space for pointers and variables.
*/

impl Solution {
    pub fn max_area(height: Vec<i32>) -> i32 {
        let mut max_area = 0;
        let mut left = 0;
        let mut right = height.len() - 1;

        while left < right {
            let current_area = std::cmp::min(height[left], height[right]) * (right - left) as i32;
            max_area = std::cmp::max(max_area, current_area);

            if height[left] < height[right] {
                left += 1;
            } else {
                right -= 1;
            }
        }

        max_area
    }
}
```

13. 02_Two_Pointers/05_Trapping_Rain_Water/0042-trapping-rain-water.rs

```
/*
Problem: LeetCode 42 - Trapping Rain Water

Key Idea:
The key idea is to calculate the amount of trapped rainwater at each position by finding the maximum height on the left and right sides.

Approach:
1. Create two arrays, `max_left` and `max_right`, to store the maximum height on the left and right sides of each position.
2. Iterate through the input array and fill the `max_left` array:
   - For each position `i`, the maximum height on the left is the maximum of `max_left[i-1]` and the height at position `i-1`.
3. Iterate through the input array in reverse and fill the `max_right` array:
   - For each position `i`, the maximum height on the right is the maximum of `max_right[i+1]` and the height at position `i+1`.
4. Iterate through the input array and calculate the trapped rainwater at each position:
    - The trapped rainwater at position `i` is the minimum of `max_left[i]` and `max_right[i]` minus the height at position `i`.
5. Sum up the trapped rainwater from step 4 to obtain the total trapped rainwater.

Time Complexity:
O(n), where n is the number of elements in the input array. We perform three passes through the array to fill the `max_left`, `max_right`, and calculate the trapped rainwater.

Space Complexity:
O(n), as we use two additional arrays of size n to store the `max_left` and `max_right` values.
*/

impl Solution {
    pub fn trap(height: Vec<i32>) -> i32 {
        let n = height.len();
        if n == 0 {
            return 0;
        }

        let mut max_left = vec![0; n];
        let mut max_right = vec![0; n];

        max_left[0] = height[0];
        for i in 1..n {
            max_left[i] = max_left[i - 1].max(height[i]);
        }

        max_right[n - 1] = height[n - 1];
        for i in (0..n - 1).rev() {
            max_right[i] = max_right[i + 1].max(height[i]);
        }

        let mut trapped_water = 0;
        for i in 0..n {
            trapped_water += (max_left[i].min(max_right[i]) - height[i]) as i32;
        }

        trapped_water
    }
}
```

14. 03_Sliding_Window/01_Best_Time_to_Buy_and_Sell_Stock/0121-best-time-to-buy

```
/*
Problem: LeetCode 121 - Best Time to Buy and Sell Stock

Key Idea:
The key idea is to iterate through the prices while keeping track of the minimum price seen so far. Calculate the potential profit at each step by subtracting the minimum price from the current price, and update the maximum profit if a higher profit is found.

Approach:
1. Initialize variables `min_price` to represent the minimum price encountered so far and `max_profit` to represent the maximum profit.
2. Iterate through the prices:
   - For each price `p`, update `min_price` if `p` is smaller.
   - Calculate the potential profit by subtracting `min_price` from `p`.
   - Update `max_profit` with the maximum of its current value and the potential profit.
3. Return `max_profit` as the result.

Time Complexity:
O(n), where n is the number of elements in the input array. We perform a single pass through the array to find the maximum profit.

Space Complexity:
O(1), as we use only a constant amount of extra space for variables.
*/

impl Solution {
    pub fn max_profit(prices: Vec<i32>) -> i32 {
        let mut min_price = i32::max_value();
        let mut max_profit = 0;

        for price in prices {
            min_price = min_price.min(price);
            max_profit = max_profit.max(price - min_price);
        }

        max_profit
    }
}
```

15. 03_Sliding_Window/02_Longest_Substring_Without_Repeating_Characters/0003-

```
/*
Problem: LeetCode 3 - Longest Substring Without Repeating Characters

Key Idea:
The key idea is to use a sliding window approach with two pointers. Maintain a window (substring) using two pointers, and a set to track characters within the current window. Move the right pointer to expand the window and add characters to the set, while checking for repeats. If a repeat is found, move the left pointer to shrink the window and remove characters from the set. Keep updating the maximum length as you iterate through the string.

Approach:
1. Initialize two pointers, `left` and `right`, representing the start and end of the current window.
2. Initialize a hashmap to store the last seen index of each character.
3. Initialize a variable `max_length` to track the maximum length of the substring.
4. Iterate through the string using the `right` pointer:
   - If the character at `right` is already in the hashmap and its index is greater than or equal to `left`, update `left` to the next position after the last occurrence of the character.
   - Calculate the current length of the substring as `right - left + 1`.
   - Update `max_length` with the maximum of its current value and the current length.
   - Update the index of the character in the hashmap.
5. Return `max_length` as the result.

Time Complexity:
O(n), where n is the length of the input string. We perform a single pass through the string using the sliding window approach.

Space Complexity:
O(min(n, m)), where n is the length of the input string and m is the size of the character set. In the worst case, the hashmap can store all characters in the string.
*/

impl Solution {
    pub fn length_of_longest_substring(s: String) -> i32 {
        let mut char_indices = std::collections::HashMap::new();
        let mut max_length = 0;
        let mut left = 0;

        for (right, ch) in s.chars().enumerate() {
            if let Some(&prev_index) = char_indices.get(&ch) {
                left = left.max(prev_index + 1);
            }

            max_length = max_length.max(right - left + 1);
            char_indices.insert(ch, right);
        }

        max_length as i32
    }
}
```

16. 03_Sliding_Window/03_Longest_Repeating_Character_Replacement/0424-longes

```
/*
Problem: LeetCode 424 - Longest Repeating Character Replacement

Key Idea:
The key idea is to use a sliding window approach and keep track of the frequency of characters within the window to find the longest substring with the same character, while allowing for replacements.

Approach:
1. Initialize variables `max_count` to track the maximum frequency of a character in the current window and `max_length` to track the maximum length of a valid substring.
2. Initialize a hashmap to store the frequency of characters within the window.
3. Initialize two pointers, `left` and `right`, representing the start and end of the current window.
4. Iterate through the string using the `right` pointer:
   - Update the frequency of the character at position `right` in the hashmap.
   - Update `max_count` with the maximum of its current value and the frequency of the character.
   - Calculate the current window size as `right - left + 1`.
    - If the current window size minus `max_count` is greater than `k`, it means there are more characters to replace than allowed. Move the `left` pointer to shrink the window.
   - Update `max_length` with the maximum of its current value and the current window size.
5. Return `max_length` as the result.

Time Complexity:
O(n), where n is the length of the input string. We perform a single pass through the string using the sliding window approach.

Space Complexity:
O(1), as the hashmap stores only a constant number of characters.
*/

impl Solution {
    pub fn character_replacement(s: String, k: i32) -> i32 {
        let s = s.as_bytes();
        let mut char_freq = [0; 26];
        let mut max_length = 0;
        let mut max_count = 0;
        let mut left = 0;

        for right in 0..s.len() {
            char_freq[(s[right] - b'A') as usize] += 1;
            max_count = max_count.max(char_freq[(s[right] - b'A') as usize]);

            if right - left + 1 - max_count > k as usize {
                char_freq[(s[left] - b'A') as usize] -= 1;
                left += 1;
            }

            max_length = max_length.max(right - left + 1);
        }

        max_length as i32
    }
}
```

17. 03_Sliding_Window/04_Permutation_in_String/0567-permutation-in-string.rs

```
/*
Problem: LeetCode 567 - Permutation in String

Key Idea:
The key idea is to use a sliding window approach with two frequency maps. Create a frequency map of characters in the pattern string, then slide a window of the same length over the main string and maintain a frequency map of characters within the window. Compare the two frequency maps at each step to check for a permutation.

Approach:
1. Initialize two arrays, `target_freq` and `window_freq`, each of size 26 to represent the frequency of characters in the target string and the current window, respectively.
2. Convert the input strings to byte arrays to work with ASCII values.
3. Initialize two pointers, `left` and `right`, representing the start and end of the current window.
4. Iterate through the `s2` string using the `right` pointer:
   - Update the frequency of the character at position `right` in the `window_freq` array.
   - If the window size is greater than or equal to the length of `s1`, decrement the frequency of the character at position `left` in the `window_freq` array and move the `left` pointer to the right.
   - If the frequencies of characters in `window_freq` match the frequencies in `target_freq`, it means a permutation of `s1` is found.
5. Return `true` if a permutation is found, otherwise return `false`.

Time Complexity:
O(n), where n is the length of the `s2` string. We perform a single pass through the `s2` string using the sliding window approach.

Space Complexity:
O(1), as the arrays `target_freq` and `window_freq` are of constant size 26.
*/

impl Solution {
    pub fn check_inclusion(s1: String, s2: String) -> bool {
        let s1 = s1.as_bytes();
        let s2 = s2.as_bytes();
        let mut target_freq = [0; 26];
        let mut window_freq = [0; 26];

        for i in 0..s1.len() {
            target_freq[(s1[i] - b'a') as usize] += 1;
            window_freq[(s2[i] - b'a') as usize] += 1;
        }

        let mut left = 0;

        for right in s1.len()..s2.len() {
            if window_freq == target_freq {
                return true;
            }

            window_freq[(s2[left] - b'a') as usize] -= 1;
            left += 1;
            window_freq[(s2[right] - b'a') as usize] += 1;
        }

        window_freq == target_freq
    }
}
```

18. 03_Sliding_Window/05_Minimum_Window_Substring/0076-minimum-window-sub

```
/*
Problem: LeetCode 76 - Minimum Window Substring

Key Idea:
The key idea is to use a sliding window approach with two pointers. Maintain a window that covers a substring in the main string and contains all required characters. Move the right pointer to expand the window, and when all characters are included, move the left pointer to minimize the window size while still satisfying the conditions. Keep track of the minimum window size as you iterate through the string.

Approach:
1. Initialize two arrays, `target_freq` and `window_freq`, each of size 128 to represent the frequency of characters in the target string and the current window, respectively. Use ASCII values for indexing.
2. Convert the input strings to byte arrays to work with ASCII values.
3. Initialize variables `left` and `right` representing the start and end of the current window, and a variable `min_length` to track the minimum window length found so far.
4. Initialize a variable `required_chars` to track the count of characters needed to form the target substring.
5. Iterate through the `t` string and update the `target_freq` array with the frequency of each character.
6. Iterate through the `s` string using the `right` pointer:
   - Update the frequency of the character at position `right` in the `window_freq` array.
   - If the frequency of the character at position `right` in the `window_freq` array is less than or equal to the frequency in the `target_freq` array, increment `required_chars`.
    - If `required_chars` is equal to the length of the target string `t`, it means all required characters are found in the current window. Move the `left` pointer to the right to try to minimize the window size.
   - Update `min_length` with the minimum of its current value and the current window size.
7. Return the minimum window substring using the `left` pointer and `min_length`.

Time Complexity:
O(n), where n is the length of the `s` string. We perform a single pass through the `s` string using the sliding window approach.

Space Complexity:
O(1), as the arrays `target_freq` and `window_freq` are of constant size 128.
*/

impl Solution {
    pub fn min_window(s: String, t: String) -> String {
        let s = s.as_bytes();
        let t = t.as_bytes();
        let mut target_freq = [0; 128];
        let mut window_freq = [0; 128];
        let mut left = 0;
        let mut min_length = usize::MAX;
        let mut start_idx = 0;
        let mut required_chars = 0;

        for &ch in t {
            target_freq[ch as usize] += 1;
            required_chars += 1;
        }

        for (right, &ch) in s.iter().enumerate() {
            window_freq[ch as usize] += 1;
            if window_freq[ch as usize] <= target_freq[ch as usize] {
                required_chars -= 1;
            }
```

```
            while required_chars == 0 {
                if right - left + 1 < min_length {
                    min_length = right - left + 1;
                    start_idx = left;
                }

                window_freq[s[left] as usize] -= 1;
                if window_freq[s[left] as usize] < target_freq[s[left] as usize] {
                    required_chars += 1;
                }

                left += 1;
            }
        }

        if min_length == usize::MAX {
            "".to_string()
        } else {
            String::from_utf8(s[start_idx..start_idx + min_length].to_vec()).unwrap()
        }
    }
}
```

19. 03_Sliding_Window/06_Sliding_Window_Maximum/0239-sliding-window-maximum

```
/*
Problem: LeetCode 239 - Sliding Window Maximum

Key Idea:
The key idea is to use a double-ended queue (deque) to store indices of array elements. As you slide the window, maintain the deque to ensure it only contains relevant indices for the current window. This helps in efficiently finding the maximum value within each window by comparing array elements with the front of the deque.

Approach:
1. Initialize an empty double-ended queue `deque` to store indices of elements.
2. Iterate through the input array using the `i` pointer:
    - While the `deque` is not empty and the front element of the `deque` is out of the current window (i.e., its index is less than or equal to `i - k`), remove it from the front of the `deque`.
    - While the `deque` is not empty and the element at the back of the `deque` is less than the current element, remove it from the back of the `deque`.
   - Push the index of the current element to the back of the `deque`.
    - If the current index is greater than or equal to `k - 1`, it means the first sliding window is complete. Add the front element of the `deque` (the maximum element) to the result array.
3. Return the result array.

Time Complexity:
O(n), where n is the number of elements in the input array. We perform a single pass through the array using the sliding window approach.

Space Complexity:
O(k), where k is the size of the sliding window. The deque can store at most `k` elements.
*/

use std::collections::VecDeque;

impl Solution {
    pub fn max_sliding_window(nums: Vec<i32>, k: i32) -> Vec<i32> {
        let k = k as usize;
        let mut result = Vec::new();
        let mut deque = VecDeque::new();

        for i in 0..nums.len() {
            while !deque.is_empty() && deque.front().unwrap() <= &(i as i32 - k as i32) {
                deque.pop_front();
            }

            while !deque.is_empty() && nums[*deque.back().unwrap() as usize] < nums[i] {
                deque.pop_back();
            }

            deque.push_back(i as i32);

            if i >= k - 1 {
                result.push(nums[*deque.front().unwrap() as usize]);
            }
        }

        result
    }
}
```

20. 04_Stack/01_Valid_Parentheses/0020-valid-parentheses.rs

```
/*
Problem: LeetCode 20 - Valid Parentheses

Key Idea:
The key idea is to use a stack to keep track of open parentheses and ensure that they are closed in the correct order.

Approach:
1. Initialize an empty stack to store open parentheses.
2. Iterate through each character in the input string:
   - If the character is an open parenthesis ('(', '{', or '['), push it onto the stack.
   - If the character is a closing parenthesis (')', '}', or ']'):
       - Check if the stack is empty. If it is, return false as there is no corresponding open parenthesis.
     - Pop the top element from the stack.
      - Check if the popped element matches the current closing parenthesis. If they don't match, return false.
3. After iterating through the entire string, check if the stack is empty. If it is, return true; otherwise, return false.

Time Complexity:
O(n), where n is the length of the input string. We perform a single pass through the string.

Space Complexity:
O(n), where n is the length of the input string in the worst case. This is the space required to store the stack.
*/

impl Solution {
    pub fn is_valid(s: String) -> bool {
        let mut stack: Vec<char> = Vec::new();

        for c in s.chars() {
            match c {
                '(' | '{' | '[' => stack.push(c),
                ')' => {
                    if stack.pop() != Some('(') {
                        return false;
                    }
                }
                '}' => {
                    if stack.pop() != Some('{') {
                        return false;
                    }
                }
                ']' => {
                    if stack.pop() != Some('[') {
                        return false;
                    }
                }
                _ => return false, // Invalid character
            }
        }

        stack.is_empty()
    }
}
```

21. 04_Stack/02_Min_Stack/0155-min-stack.rs

```
/*
Problem: LeetCode 155 - Min Stack

Key Idea:
The key idea is to maintain a stack that not only stores elements but also keeps track of the minimum element in the stack at any point.

Approach:
1. Create a stack to store elements.
2. Create a second stack to store the minimum elements.
3. For each push operation, push the element onto the main stack.
   - If the second stack is empty or the element is less than or equal to the top element of the second stack, push the element onto the second stack as the new minimum.
4. For each pop operation, pop the top element from the main stack.
    - If the popped element is equal to the top element of the second stack (i.e., the current minimum), pop the top element from the second stack as well to update the minimum.
5. For each top operation, return the top element of the main stack.
6. For each get_min operation, return the top element of the second stack, which represents the current minimum element.

Time Complexity:
- Push, pop, top, and get_min operations all have O(1) time complexity as we perform constant-time stack operations.

Space Complexity:
- O(n), where n is the number of elements in the stack. We maintain two separate stacks to store elements and minimums.
*/

struct MinStack {
    stack: Vec<i32>,
    min_stack: Vec<i32>,
}


/**
 * `&self` means the method takes an immutable reference.
 * If you need a mutable reference, change it to `&mut self` instead.
 */
impl MinStack {
    fn new() -> Self {
        MinStack {
            stack: Vec::new(),
            min_stack: Vec::new(),
        }
    }

    fn push(&mut self, val: i32) {
        self.stack.push(val);
        if self.min_stack.is_empty() || val <= *self.min_stack.last().unwrap() {
            self.min_stack.push(val);
        }
    }

    fn pop(&mut self) {
        if let Some(top) = self.stack.pop() {
            if top == *self.min_stack.last().unwrap() {
                self.min_stack.pop();
            }
        }
```

```
    }

    fn top(&self) -> i32 {
        *self.stack.last().unwrap()
    }

    fn get_min(&self) -> i32 {
        *self.min_stack.last().unwrap()
    }
}

/**
 * Your MinStack object will be instantiated and called as such:
 * let obj = MinStack::new();
 * obj.push(val);
 * obj.pop();
 * let ret_3: i32 = obj.top();
 * let ret_4: i32 = obj.get_min();
 */
```

22. 04_Stack/03_Evaluate_Reverse_Polish_Notation/0150-evaluate-reverse-polish-no

```
/*
Problem: LeetCode 150 - Evaluate Reverse Polish Notation

Key Idea:
The key idea is to use a stack to evaluate the given reverse Polish notation expression.

Approach:
1. Initialize an empty stack to store operands.
2. Iterate through each element in the input tokens:
   - If the current element is an operand (a number), push it onto the stack.
   - If the current element is an operator ('+', '-', '*', '/'):
     - Pop the top two elements from the stack as the operands.
       - Perform the operation based on the current operator and push the result back onto the stack.
3. After processing all elements, the result should be the only element left on the stack. Pop and return this result.

Time Complexity:
O(n), where n is the number of elements in the input tokens. We perform a single pass through the tokens.

Space Complexity:
O(n), where n is the number of elements in the input tokens. The stack can have at most n/2 elements in the worst case.
*/

impl Solution {
    pub fn eval_rpn(tokens: Vec<String>) -> i32 {
        let mut stack: Vec<i32> = Vec::new();

        for token in tokens {
            if let Ok(operand) = token.parse::<i32>() {
                stack.push(operand);
            } else {
                let b = stack.pop().unwrap();
                let a = stack.pop().unwrap();
                let result = match token.as_str() {
                    "+" => a + b,
                    "-" => a - b,
                    "*" => a * b,
                    "/" => a / b,
                    _ => 0, // Invalid operator, can be handled differently if needed
                };
                stack.push(result);
            }
        }

        stack.pop().unwrap()
    }
}
```

23. 04_Stack/04_Generate_Parentheses/0022-generate-parentheses.rs

```
/*
Problem: LeetCode 22 - Generate Parentheses

Key Idea:
The key idea is to use recursion to generate all valid combinations of parentheses.

Approach:
1. Initialize an empty vector `result` to store the generated combinations.
2. Create a recursive function `generate` that takes four arguments: `current` (the current combination), `open` (the number of open parentheses used), `close` (the number of closing parentheses used), and `n` (the total number of pairs).
3. In the `generate` function:
   - If `current` is of length `2 * n`, it means a valid combination is generated, so add it to the `result` vector.
    - If `open` is less than `n`, it means we can add an open parenthesis. Recursively call `generate` with `current` + '(', `open + 1`, `close`, and `n`.
   - If `close` is less than `open`, it means we can add a closing parenthesis. Recursively call `generate` with `current` + ')', `open`, `close + 1`, and `n`.
4. Start the recursion with an empty `current`, `open` and `close` as 0, and `n` as the given input.
5. Return the `result` vector containing all valid combinations.

Time Complexity:
O(2^n), where n is the number of pairs. Each valid combination has a length of 2 * n, and there are 2^(2n) possible combinations.

Space Complexity:
O(2^n), as there can be a total of 2^(2n) valid combinations in the result vector.
*/

impl Solution {
    pub fn generate_parenthesis(n: i32) -> Vec<String> {
        let mut result: Vec<String> = Vec::new();

        fn generate(current: String, open: i32, close: i32, n: i32, result: &mut Vec<String>) {
            if current.len() == (2 * n) as usize {
                result.push(current);
                return;
            }

            if open < n {
                generate(current.clone() + "(", open + 1, close, n, result);
            }

            if close < open {
                generate(current.clone() + ")", open, close + 1, n, result);
            }
        }

        generate("".to_string(), 0, 0, n, &mut result);
        result
    }
}
```

24. 04_Stack/05_Daily_Temperatures/0739-daily-temperatures.rs

```
/*
Problem: LeetCode 739 - Daily Temperatures

Key Idea:
The key idea is to use a stack to keep track of the indices of the temperatures while iterating through the array.

Approach:
1. Initialize an empty stack to store indices of temperatures.
2. Initialize a result vector `result` of the same length as the input `temperatures` with all values set to 0.
3. Iterate through the `temperatures` array with an index `i`:
   - While the stack is not empty and the temperature at the current index `i` is greater than the temperature at the index at the top of the stack, pop elements from the stack and calculate the difference between their indices and `i`. Set this difference as the value at the corresponding index in the `result` vector.
   - Push the current index `i` onto the stack.
4. After the loop, the stack may still contain indices of temperatures for which there were no warmer days. Set the values at these indices in the `result` vector to 0.
5. Return the `result` vector.

Time Complexity:
O(n), where n is the length of the input `temperatures`. We perform a single pass through the array.

Space Complexity:
O(n), as the stack can store at most n elements.
*/

impl Solution {
    pub fn daily_temperatures(temperatures: Vec<i32>) -> Vec<i32> {
        let mut result: Vec<i32> = vec![0; temperatures.len()];
        let mut stack: Vec<usize> = Vec::new();

        for (i, &temp) in temperatures.iter().enumerate() {
            while let Some(&prev_index) = stack.last() {
                if temp > temperatures[prev_index] {
                    stack.pop();
                    result[prev_index] = (i - prev_index) as i32;
                } else {
                    break;
                }
            }
            stack.push(i);
        }

        result
    }
}
```

25. 04_Stack/06_Car_Fleet/0853-car-fleet.rs

```
/*
Problem: LeetCode 853 - Car Fleet

Key Idea:
The key idea is to simulate the movement of cars and calculate the time it takes for each car to reach the target.

Approach:
1. Create a vector `cars` of pairs, where each pair represents a car with its position and speed.
2. Sort the `cars` vector by the position of the cars, with the car closest to the target at the end.
3. Initialize a variable `time` to 0, which represents the current time.
4. Initialize a variable `count` to 0, which represents the number of car fleets.
5. Iterate through the `cars` vector from the end to the beginning (from the car closest to the target to the farthest):
   - Calculate the time it takes for the current car to reach the target using the formula: `time = (target - position) as f64 / speed as f64`.
    - If the calculated time is greater than the current `time`, it means a new car fleet is formed. Increment `count` and update `time` to the calculated time.
6. Return the value of `count`, which represents the number of car fleets.

Time Complexity:
O(n log n), where n is the number of cars. Sorting the `cars` vector takes O(n log n) time.

Space Complexity:
O(n), as we store the cars in a vector.
*/

impl Solution {
    pub fn car_fleet(target: i32, position: Vec<i32>, speed: Vec<i32>) -> i32 {
        let mut cars: Vec<(i32, i32)> = position.into_iter().zip(speed.into_iter()).collect();
        cars.sort_by_key(|&(pos, _)| -pos); // Sort by position in descending order.

        let mut time = 0.0;
        let mut count = 0;

        for (pos, spd) in cars {
            let curr_time = (target - pos) as f64 / spd as f64;
            if curr_time > time {
                count += 1;
                time = curr_time;
            }
        }

        count
    }
}
```

26. 04_Stack/07_Largest_Rectangle_In_Histogram/0084-largest-rectangle-in-histogra

```
/*
Problem: LeetCode 84 - Largest Rectangle in Histogram

Key Idea:
The key idea is to use a stack to keep track of the indices of the histogram bars while iterating through the heights.

Approach:
1. Initialize an empty stack to store indices of histogram bars.
2. Initialize variables `max_area` and `i` to store the maximum area and the current index.
3. Iterate through the heights array:
   - While the stack is not empty and the current height is less than the height at the index at the top of the stack, pop elements from the stack and calculate the area using the height at the popped index and the difference between the current index and the index at the top of the stack. Update `max_area` if the calculated area is greater.
   - Push the current index onto the stack.
4. After the loop, there may be remaining elements in the stack. For each remaining index in the stack, pop it and calculate the area using the height at the popped index and the difference between the current index and the popped index. Update `max_area` if the calculated area is greater.
5. Return `max_area` as the largest rectangle area.

Time Complexity:
O(n), where n is the number of heights in the input array. We perform a single pass through the heights.

Space Complexity:
O(n), as the stack can store at most n indices.
*/

impl Solution {
    pub fn largest_rectangle_area(heights: Vec<i32>) -> i32 {
        let mut stack: Vec<usize> = Vec::new();
        let mut max_area = 0;
        let mut i = 0;

        while i < heights.len() {
            if stack.is_empty() || heights[i] >= heights[*stack.last().unwrap()] {
                stack.push(i);
                i += 1;
            } else {
                let top = stack.pop().unwrap();
                let width = if stack.is_empty() {
                    i
                } else {
                    i - stack.last().unwrap() - 1
                };
                max_area = max_area.max(heights[top] * width as i32);
            }
        }

        while !stack.is_empty() {
            let top = stack.pop().unwrap();
            let width = if stack.is_empty() {
                i
            } else {
                i - stack.last().unwrap() - 1
            };
            max_area = max_area.max(heights[top] * width as i32);
        }
```

```
        max_area
    }
}
```

27. 05_Binary_Search/01_Binary_Search/0704-binary-search.rs

```
/*
Problem: LeetCode 704 - Binary Search

Key Idea:
The key idea is to use binary search. Compare the target with the middle element of the array and eliminate half of the search space based on this comparison. Repeat this process until the target is found or the search space is empty.

Approach:
1. Initialize two pointers, `left` and `right`, representing the leftmost and rightmost indices of the search interval.
2. While `left <= right`, perform binary search:
   a. Calculate the middle index as `mid = left + (right - left) / 2`.
   b. If the element at the `mid` index is equal to the target, return `mid`.
    c. If the element at the `mid` index is less than the target, update `left` to `mid + 1` (search in the right half).
   d. If the element at the `mid` index is greater than the target, update `right` to `mid - 1` (search in the left half).
3. If the target is not found in the array, return -1 to indicate that it doesn't exist in the array.

Time Complexity:
The time complexity of binary search is O(log N) since the search interval is halved in each step.

Space Complexity:
The space complexity is O(1) as we are using a constant amount of extra space.
*/

impl Solution {
    pub fn search(nums: Vec<i32>, target: i32) -> i32 {
        let mut left = 0;
        let mut right = nums.len() as i32 - 1;

        while left <= right {
            let mid = left + (right - left) / 2;

            if nums[mid as usize] == target {
                return mid;
            } else if nums[mid as usize] < target {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }

        -1 // Target not found
    }
}
```

28. 05_Binary_Search/02_Search_a_2D_Matrix/0074-search-a-2d-matrix.rs

```
/*
Problem: LeetCode 74 - Search a 2D Matrix

Key Idea:
The given matrix is sorted both row-wise and column-wise. We can treat this matrix as a 1D sorted array and use binary search to efficiently find the target element.

Approach:
1. Initialize two pointers, `start` and `end`, representing the leftmost and rightmost indices of the 1D array.
2. While `start <= end`, perform binary search:
   a. Calculate the middle index as `mid = start + (end - start) / 2`.
    b. Convert `mid` into a 2D index as `(row, col)` where `row = mid / n` and `col = mid % n` (assuming the matrix has 'n' columns).
   c. If `matrix[row][col]` is equal to the target, return `true`.
   d. If `matrix[row][col]` is less than the target, update `start` to `mid + 1`.
   e. If `matrix[row][col]` is greater than the target, update `end` to `mid - 1`.
3. If the target is not found after the binary search, return `false`.

Time Complexity:
The time complexity of binary search is O(log(N*M)), where N is the number of rows, and M is the number of columns in the matrix.

Space Complexity:
The space complexity is O(1) as we are using a constant amount of extra space.
*/

impl Solution {
    pub fn search_matrix(matrix: Vec<Vec<i32>>, target: i32) -> bool {
        let rows = matrix.len() as i32;
        let cols = matrix[0].len() as i32;

        let mut start = 0;
        let mut end = rows * cols - 1;

        while start <= end {
            let mid = start + (end - start) / 2;
            let mid_value = matrix[(mid / cols) as usize][(mid % cols) as usize];

            if mid_value == target {
                return true;
            } else if mid_value < target {
                start = mid + 1;
            } else {
                end = mid - 1;
            }
        }

        false
    }
}
```

29. 05_Binary_Search/03_Koko_Eating_Bananas/0875-koko-eating-bananas.rs

```
/*
Problem: LeetCode 875 - Koko Eating Bananas

Key Idea:
The key idea is to use binary search on the possible days. Calculate the mid-point of the search range as the number of days, then check if it's possible for the workers to finish the bananas in those days. Adjust the search range based on the result until you find the minimum possible number of days.

Approach:
1. Define a binary search range for the possible speeds.
   - Initialize `left` to 1 (minimum possible speed) and `right` to the maximum pile size.
2. Perform a binary search on the speed range:
   a. Calculate the middle speed as `mid` = (left + right) / 2.
   b. For each pile of bananas, calculate the time required to eat it completely at speed `mid`.
      - Time = (pile[i] + mid - 1) / mid
   c. Sum up the times for all piles to get the total time required.
   d. If the total time is less than or equal to `H` (maximum allowed time), reduce the speed by setting `right` = `mid`.
   e. If the total time exceeds `H`, increase the speed by setting `left` = `mid + 1`.
3. Return the minimum speed `left` after the binary search.

Time Complexity:
- Binary search takes O(log(max pile size)) time.
- Calculating the total time for all piles takes O(n) time.
- Overall time complexity is O(n * log(max pile size)).

Space Complexity:
- The space complexity is O(1) as we are using a constant amount of extra space.
*/

impl Solution {
    pub fn min_eating_speed(piles: Vec<i32>, h: i32) -> i32 {
        let mut left = 1;
        let mut right = *piles.iter().max().unwrap();

        while left < right {
            let mid = left + (right - left) / 2;
            let total_time = piles
                .iter()
                .map(|&pile| (pile + mid - 1) / mid)
                .sum::<i32>();

            if total_time <= h {
                right = mid;
            } else {
                left = mid + 1;
            }
        }

        left
    }
}
```

30. 05_Binary_Search/04_Find_Minimum_In_Rotated_Sorted_Array/0153-find-minimu

```
/*
Problem: LeetCode 153 - Find Minimum in Rotated Sorted Array

Key Idea:
The key idea is to use binary search. Compare the middle element with the rightmost element to determine whether the minimum element is in the left or right half. Adjust the search range accordingly until you find the minimum element.

Approach:
1. Use binary search to find the minimum element.
2. Initialize two pointers, `left` and `right`, to the start and end of the array, respectively.
3. While `left` is less than `right`:
   a. Calculate the middle index as `mid` = (left + right) / 2.
   b. Check if the element at `mid` is greater than the element at `right`.
       - If true, this means the minimum element is in the right half, so update `left` = `mid + 1`.
      - If false, the minimum element is in the left half, so update `right` = `mid`.
4. After the binary search, `left` will point to the minimum element, and we return the element at `left`.

Time Complexity:
- Binary search takes O(log n) time.

Space Complexity:
- The space complexity is O(1) as we use only a constant amount of extra space.
*/

impl Solution {
    pub fn find_min(nums: Vec<i32>) -> i32 {
        let mut left = 0;
        let mut right = nums.len() - 1;

        while left < right {
            let mid = left + (right - left) / 2;

            if nums[mid] > nums[right] {
                left = mid + 1;
            } else {
                right = mid;
            }
        }

        nums[left]
    }
}
```

31. 05_Binary_Search/05_Search_In_Rotated_Sorted_Array/0033-search-in-rotated-so

```
/*
Problem: LeetCode 33 - Search in Rotated Sorted Array

Key Idea:
The key idea is to use modified binary search. Compare the middle element with the left and right endpoints to determine if the left or right half is sorted. Adjust the search range accordingly until you find the target or exhaust the search space.

Approach:
1. Initialize two pointers, `left` and `right`, to the start and end of the array.
2. In each step of the binary search:
   a. Calculate the middle index as `mid` = (left + right) / 2.
   b. Check if `nums[mid]` is equal to the target. If true, return `mid` as the result.
   c. Determine if the left half or right half of the array is sorted by comparing `nums[left]` and `nums[mid]`.
      - If the left half is sorted:
        i. Check if the target falls within the range of `nums[left]` and `nums[mid]`. If true, update `right = mid - 1`.
        ii. Otherwise, update `left = mid + 1`.
      - If the right half is sorted:
        i. Check if the target falls within the range of `nums[mid]` and `nums[right]`. If true, update `left = mid + 1`.
        ii. Otherwise, update `right = mid - 1`.
3. If the target is not found after the binary search, return -1 to indicate that it's not in the array.

Time Complexity:
- The binary search takes O(log n) time, where n is the length of the input array.

Space Complexity:
- The space complexity is O(1) as we are using only a constant amount of extra space.
*/

impl Solution {
    pub fn search(nums: Vec<i32>, target: i32) -> i32 {
        let mut left = 0;
        let mut right = nums.len() - 1;

        while left <= right {
            let mid = left + (right - left) / 2;

            if nums[mid] == target {
                return mid as i32;
            }

            if nums[left] <= nums[mid] {
                if nums[left] <= target && target < nums[mid] {
                    right = mid - 1;
                } else {
                    left = mid + 1;
                }
            } else {
                if nums[mid] < target && target <= nums[right] {
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            }
        }
```

```
        -1
    }
}
```

32. 05_Binary_Search/06_Time_Based_Key_Value_Store/0981-time-based-key-value-

```
/*
Problem: LeetCode 981 - Time Based Key-Value Store

Key Idea:
We can use a HashMap to store the key-value pairs and their timestamps. To efficiently retrieve
values based on timestamps, we can use binary search.

Approach:
1. Create a HashMap where the keys are strings (the data keys) and the values are vectors of pairs
(timestamp, value).
   - HashMap<String, Vec<(i32, String)>> for storing key-value pairs.
2. Implement two methods, `set` and `get`.
   - `set` method:
     - Insert the key-value pair with the given timestamp into the HashMap.
      - Since the input timestamps are monotonically increasing, we can insert new pairs directly
into the vector.
   - `get` method:
     - Retrieve the vector associated with the given key.
      - Perform a binary search within the vector to find the largest timestamp that is less than
or equal to the input timestamp.
     - Return the corresponding value if found, or an empty string otherwise.
3. The binary search for each `get` operation takes O(log n) time, where n is the number of
timestamps associated with the key.
   - The space complexity is O(N) for storing all key-value pairs and their timestamps.

Time Complexity:
- O(log n) for each `get` operation.
- O(1) for each `set` operation.

Space Complexity:
- O(N) where N is the total number of key-value pairs.
*/

use std::collections::HashMap;

struct TimeMap {
    store: HashMap<String, Vec<(i32, String)>>,
}

impl TimeMap {
    fn new() -> Self {
        TimeMap {
            store: HashMap::new(),
        }
    }

    fn set(&mut self, key: String, value: String, timestamp: i32) {
        self.store
            .entry(key)
            .or_insert_with(Vec::new)
            .push((timestamp, value));
    }

    fn get(&self, key: String, timestamp: i32) -> String {
        if let Some(values) = self.store.get(&key) {
            let mut left = 0;
            let mut right = values.len() as i32 - 1;

            while left <= right {
                let mid = left + (right - left) / 2;
```

```
                let (ts, val) = &values[mid as usize];

                if *ts == timestamp {
                    return val.clone();
                } else if *ts < timestamp {
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            }

            if right >= 0 {
                return values[right as usize].1.clone();
            }
        }

        String::new()
    }
}
```

33. 05_Binary_Search/07_Median_of_Two_Sorted_Arrays/0004-median-of-two-sorted

```
/*
Problem: LeetCode 4 - Median of Two Sorted Arrays

Key Idea:
The key idea is to perform a binary search on the smaller of the two arrays while ensuring a balanced partition between the arrays. This involves comparing elements near the middle to find the correct partition point and calculate the median based on the partitioned elements. Continue adjusting the partition until you find the median.

Approach:
1. Let's denote the two input arrays as nums1 and nums2.
2. Find the smaller of the two array lengths: n and m, where n = nums1.len() and m = nums2.len().
3. Perform a binary search on the smaller array (nums1) to partition it into two parts: left_part and right_part, such that:
   - The length of left_part is (n + m + 1) / 2, rounded down.
   - The maximum element in left_part is less than or equal to the minimum element in right_part.
4. Calculate the median based on left_part and right_part:
   - If n + m is odd, the median is the maximum element in left_part.
    - If n + m is even, the median is the average of the maximum element in left_part and the minimum element in right_part.
5. Implement the binary search:
   - Initialize two pointers, left1 and right1, to perform binary search on nums1.
   - Compute the corresponding positions left2 and right2 in nums2.
   - Adjust the pointers based on the comparison of elements at these positions.
   - Continue the binary search until the correct partition is found.

Time Complexity:
- The binary search has a time complexity of O(log(min(n, m))).

Space Complexity:
- The space complexity is O(1) as we are using a constant amount of extra space.
*/

use std::cmp::max;
use std::cmp::min;

impl Solution {
    pub fn find_median_sorted_arrays(nums1: Vec<i32>, nums2: Vec<i32>) -> f64 {
        let (nums1, nums2) = if nums1.len() <= nums2.len() {
            (nums1, nums2)
        } else {
            (nums2, nums1)
        };

        let (n, m) = (nums1.len(), nums2.len());
        let half_len = (n + m + 1) / 2;
        let (mut left1, mut right1) = (0, n);

        while left1 <= right1 {
            let partition1 = (left1 + right1) / 2;
            let partition2 = half_len - partition1;

            let max_left1 = if partition1 == 0 {
                i32::MIN
            } else {
                nums1[partition1 - 1]
            };
            let min_right1 = if partition1 == n {
                i32::MAX
            } else {
```

```
                nums1[partition1]
            };

            let max_left2 = if partition2 == 0 {
                i32::MIN
            } else {
                nums2[partition2 - 1]
            };
            let min_right2 = if partition2 == m {
                i32::MAX
            } else {
                nums2[partition2]
            };

            if max_left1 <= min_right2 && max_left2 <= min_right1 {
                if (n + m) % 2 == 0 {
                    return (max_left1.max(max_left2) as f64 + min_right1.min(min_right2) as f64)
                        / 2.0;
                } else {
                    return max_left1.max(max_left2) as f64;
                }
            } else if max_left1 > min_right2 {
                right1 = partition1 - 1;
            } else {
                left1 = partition1 + 1;
            }
        }

        0.0 // This should not be reached
    }
}

/*
// Merging arrays

impl Solution {
    pub fn find_median_sorted_arrays(nums1: Vec<i32>, nums2: Vec<i32>) -> f64 {
        let mut merged = Vec::with_capacity(nums1.len() + nums2.len());
        let (mut i, mut j) = (0, 0);

        while i < nums1.len() && j < nums2.len() {
            if nums1[i] <= nums2[j] {
                merged.push(nums1[i]);
                i += 1;
            } else {
                merged.push(nums2[j]);
                j += 1;
            }
        }

        while i < nums1.len() {
            merged.push(nums1[i]);
            i += 1;
        }

        while j < nums2.len() {
            merged.push(nums2[j]);
            j += 1;
        }

        let len = merged.len();
        if len % 2 == 0 {
            let mid = len / 2;
            (merged[mid - 1] as f64 + merged[mid] as f64) / 2.0
```

```
        } else {
            merged[len / 2] as f64
        }
    }
}
*/

/*
// Short solution

impl Solution {
    pub fn find_median_sorted_arrays(nums1: Vec<i32>, nums2: Vec<i32>) -> f64 {
        let mut merged = nums1.iter().chain(&nums2).copied().collect::<Vec<_>>();
        merged.sort();

        let mid = merged.len() / 2;
        if merged.len() % 2 == 0 {
            (merged[mid - 1] as f64 + merged[mid] as f64) / 2.0
        } else {
            merged[mid] as f64
        }
    }
}
*/
```

34. 06_Linked_List/01_Reverse_Linked_List/0206-reverse-linked-list.rs

```
/*
Problem: LeetCode 206 - Reverse Linked List

Key Idea:
The key idea is to reverse the linked list by iteratively changing the direction of the pointers.

Approach:
1. Initialize three pointers: `prev` as `None`, `curr` as the head of the linked list, and `next` as `None`.
2. Iterate through the linked list:
   - Save the next node of `curr` in `next`.
   - Update the `next` pointer of `curr` to point to the `prev` node (reverse the direction).
   - Move `prev` to `curr` and `curr` to `next`.
3. After the loop, `prev` will be the new head of the reversed linked list.
4. Return `prev` as the new head.

Time Complexity:
O(n), where n is the number of nodes in the linked list. We perform a single pass through the linked list.

Space Complexity:
O(1), as we use a constant amount of extra space for the three pointers.
*/

// Definition for singly-linked list

// #[derive(PartialEq, Eq, Clone, Debug)]
// pub struct ListNode {
//     pub val: i32,
//     pub next: Option<Box<ListNode>>,
// }

// impl ListNode {
//     #[inline]
//     fn new(val: i32) -> Self {
//         ListNode { next: None, val }
//     }
// }

impl Solution {
    pub fn reverse_list(head: Option<Box<ListNode>>) -> Option<Box<ListNode>> {
        let mut prev = None;
        let mut curr = head;

        while let Some(mut node) = curr.take() {
            let next = node.next.take();
            node.next = prev.take();
            prev = Some(node);
            curr = next;
        }

        prev
    }
}
```

35. 06_Linked_List/02_Merge_Two_Sorted_Lists/0021-merge-two-sorted-lists.rs

```
/*
Problem: LeetCode 21 - Merge Two Sorted Lists

Key Idea:
The key idea is to create a new linked list while comparing and merging nodes from both input lists in a sorted manner.

Approach:
1. Start by handling the base cases:
   - If both `l1` and `l2` are `None`, there are no elements to merge, so return `None`.
    - If either `l1` or `l2` is `None`, return the other list because a single linked list is always sorted.
2. Initialize a pattern match on `(l1, l2)` to handle the different cases:
   - If `l1` is not `None` and `l2` is `None`, return `l1`.
   - If `l1` is `None` and `l2` is not `None`, return `l2`.
   - If both `l1` and `l2` are not `None`, proceed with the merging process.
3. Initialize mutable references `node1` and `node2` to the values in `l1` and `l2`.
4. Compare the values of `node1` and `node2`. Take the smaller of the two and set it as the `next` node for the current result node.
5. Recursively call `Solution::merge_two_lists` to continue merging the remaining elements. Update the references `node1` and `node2` accordingly based on which value was selected in the previous step.
6. Repeat steps 4 and 5 until one of the input lists becomes empty.
7. After the loop, one of the input lists might have remaining elements. Append the remaining elements to the merged list.
8. Finally, return the merged list starting from the `next` node of the dummy node.

Time Complexity:
O(n), where n is the total number of nodes in the two input linked lists. We perform a single pass through both lists.

Space Complexity:
O(1), as we use a constant amount of extra space for pointers and nodes.
*/

// Definition for singly-linked list

// #[derive(PartialEq, Eq, Clone, Debug)]
// pub struct ListNode {
//     pub val: i32,
//     pub next: Option<Box<ListNode>>,
// }

// impl ListNode {
//     #[inline]
//     fn new(val: i32) -> Self {
//         ListNode { next: None, val }
//     }
// }

impl Solution {
    pub fn merge_two_lists(
        l1: Option<Box<ListNode>>,
        l2: Option<Box<ListNode>>,
    ) -> Option<Box<ListNode>> {
        match (l1, l2) {
            (None, None) => None,
            (Some(node1), None) => Some(node1),
            (None, Some(node2)) => Some(node2),
            (Some(mut node1), Some(mut node2)) => {
```

```
                if node1.val < node2.val {
                    node1.next = Solution::merge_two_lists(node1.next, Some(node2));
                    Some(node1)
                } else {
                    node2.next = Solution::merge_two_lists(Some(node1), node2.next);
                    Some(node2)
                }
            }
        }
    }
}
```

36. 06_Linked_List/03_Reorder_List/0143-reorder-list.rs

```
/*
Problem: LeetCode 143 - Reorder List

Key Idea:
The key idea is to reorder the linked list by dividing it into two halves, reversing the second half, and merging the two halves.

Approach:
1. Find the middle of the linked list using the slow and fast pointer technique.
2. Split the linked list into two halves at the middle node.
3. Reverse the second half of the linked list.
4. Merge the two halves of the linked list in an interleaving manner.
    - Start with two pointers, `h1` at the head of the first half and `h2` at the head of the reversed second half.
   - While `h2` is not None, insert the node pointed to by `h2` after the node pointed to by `h1`.
   - Move `h1` and `h2` accordingly.
5. Set the next of the last node in the merged list to None.

Time Complexity:
O(n), where n is the number of nodes in the linked list. We perform a single pass through the list to find the middle and reverse the second half.

Space Complexity:
O(1), as we use a constant amount of extra space for pointers and variables.
*/

use std::cmp::Ordering;

impl Solution {
    pub fn reorder_list(head: &mut Option<Box<ListNode>>) {
        let mut second_half = Self::split_list(head);
        second_half = Self::reverse_list(second_half);
        Self::merge_lists(head, second_half);
    }

    // Function to find the middle of the list and split it
    fn split_list(head: &mut Option<Box<ListNode>>) -> Option<Box<ListNode>> {
        let (mut fast, mut slow) = (&head.clone(), head);
        while fast.is_some() {
            fast = &(fast.as_ref().unwrap().next);
            if fast.is_some() {
                fast = &fast.as_ref().unwrap().next;
                slow = &mut slow.as_mut().unwrap().next;
            }
        }
        slow.as_mut().unwrap().next.take()
    }

    // Function to reverse a linked list
    fn reverse_list(mut head: Option<Box<ListNode>>) -> Option<Box<ListNode>> {
        let mut prev = None;
        while let Some(mut curr) = head {
            head = curr.next;
            curr.next = prev;
            prev = Some(curr);
        }
        prev
    }

    // Function to merge two lists alternately
```

```
    fn merge_lists(mut head1: &mut Option<Box<ListNode>>, mut head2: Option<Box<ListNode>>) {
        let mut h1 = head1;
        let mut h2 = head2;
        while h1.is_some() && h2.is_some() {
            let mut h1next = h1.as_mut().unwrap().next.take();
            let mut h2next = h2.as_mut().unwrap().next.take();
            h1.as_mut().unwrap().next = h2;
            h1.as_mut().unwrap().next.as_mut().unwrap().next = h1next;
            h1 = &mut h1.as_mut().unwrap().next.as_mut().unwrap().next;
            h2 = h2next;
        }
        if h2.is_some() {
            h1 = &mut h2;
        }
    }
}
```

37. 06_Linked_List/04_Remove_Nth_Node_From_End_of_List/0019-remove-nth-node

```
/*
Problem: LeetCode 19 - Remove Nth Node From End of List

Key Idea:
The key idea is to use two pointers, `fast` and `slow`, to find the Nth node from the end.

Approach:
1. Initialize two pointers, `fast` and `slow`, to the head of the linked list.
2. Move the `fast` pointer N nodes ahead, effectively creating a gap of N nodes between `fast` and
`slow`.
3. Move both `fast` and `slow` one node at a time until `fast` reaches the end of the list.
4. Now, `slow` is at the Nth node from the end.
5. To remove the Nth node, update the `next` pointer of the node before `slow` to skip `slow`.
6. If N is equal to the length of the list, remove the head node.
7. Return the modified linked list.

Time Complexity:
O(N), where N is the number of nodes in the linked list. We perform a single pass through the
list.

Space Complexity:
O(1), as we use a constant amount of extra space for pointers and variables.
*/

impl Solution {
    pub fn remove_nth_from_end(head: Option<Box<ListNode>>, n: i32) -> Option<Box<ListNode>> {
        let mut dummy = Box::new(ListNode::new(0));
        dummy.next = head;

        let mut fast = dummy.clone();
        let mut slow = dummy.as_mut();

        // Move the fast pointer N nodes ahead
        for _ in 0..n {
            fast = fast.next.unwrap();
        }

        // Move both pointers until fast reaches the end
        while fast.next.is_some() {
            fast = fast.next.unwrap();
            slow = slow.next.as_mut().unwrap();
        }

        // Remove the Nth node
        let next = slow.next.as_mut().unwrap();
        slow.next = next.next.clone();

        dummy.next
    }
}
```

38. 06_Linked_List/05_Copy_List_With_Random_Pointer/0138-copy-list-with-random-

39. 06_Linked_List/06_Add_Two_Numbers/0002-add-two-numbers.rs

```
/*
Problem: LeetCode 2 - Add Two Numbers

Key Idea:
The key idea is to simulate the addition of two numbers, taking care of carry values.

Approach:
1. Initialize a dummy node as the head of the result linked list.
2. Initialize two pointers, `p` and `q`, to the heads of the input linked lists `l1` and `l2`.
3. Initialize a carry variable as 0.
4. Iterate through both `l1` and `l2` simultaneously:
   - Calculate the sum of the current nodes' values and the carry.
   - Update the carry for the next iteration.
   - Create a new node with the value of (sum % 10) and append it to the result linked list.
   - Move `p`, `q`, and the result pointer accordingly.
5. After the loop, if there is a carry, create a new node with the carry value and append it to the result linked list.
6. Return the next of the dummy node as the head of the result linked list.

Time Complexity:
O(max(N, M)), where N and M are the lengths of the input linked lists `l1` and `l2`. We perform a single pass through the longer list.

Space Complexity:
O(max(N, M)), as the result linked list can have a maximum length of max(N, M) + 1.
*/

impl Solution {
    pub fn add_two_numbers(
        l1: Option<Box<ListNode>>,
        l2: Option<Box<ListNode>>,
    ) -> Option<Box<ListNode>> {
        let mut dummy = ListNode::new(0);
        let (mut p, mut q, mut carry) = (l1, l2, 0);
        let mut current = &mut dummy;

        while p.is_some() || q.is_some() {
            let x = p.as_ref().map_or(0, |node| node.val);
            let y = q.as_ref().map_or(0, |node| node.val);
            let sum = x + y + carry;

            carry = sum / 10;
            current.next = Some(Box::new(ListNode::new(sum % 10)));

            if let Some(ref mut node) = current.next {
                current = &mut *node;
            }

            if let Some(ref mut node) = p {
                p = node.next.take();
            }

            if let Some(ref mut node) = q {
                q = node.next.take();
            }
        }

        if carry > 0 {
            current.next = Some(Box::new(ListNode::new(carry)));
        }
```

```
        dummy.next
    }
}
```

40. 06_Linked_List/07_Linked_List_Cycle/0141-linked-list-cycle.rs

41. 06_Linked_List/08_Find_The_Duplicate_Number/0287-find-the-duplicate-number.r

```
/*
Problem: LeetCode 287 - Find the Duplicate Number

Key Idea:
The key idea is to treat the array as a linked list with a cycle and find the entrance point of the cycle using the Floyd's Tortoise and Hare algorithm.

Approach:
1. Initialize two pointers, `slow` and `fast`, to the first element of the array.
2. Move `slow` one step at a time and `fast` two steps at a time until they meet inside the cycle.
3. Once they meet, reset one of the pointers, say `slow`, to the first element and move both `slow` and `fast` one step at a time until they meet again.
4. The point where they meet is the entrance point of the cycle, which corresponds to the duplicate number.
5. Return the duplicate number.

Time Complexity:
O(n), where n is the length of the array. We perform two iterations through the array, each taking O(n) time.

Space Complexity:
O(1), as we use a constant amount of extra space for pointers and variables.
*/

impl Solution {
    pub fn find_duplicate(nums: Vec<i32>) -> i32 {
        let mut slow = nums[0];
        let mut fast = nums[0];

        // Phase 1: Detect cycle
        loop {
            slow = nums[slow as usize];
            fast = nums[nums[fast as usize] as usize];

            if slow == fast {
                break;
            }
        }

        // Phase 2: Find entrance to the cycle
        slow = nums[0];
        while slow != fast {
            slow = nums[slow as usize];
            fast = nums[fast as usize];
        }

        slow
    }
}
```

42. 06_Linked_List/09_LRU_Cache/0146-lru-cache.rs

```
/*
Problem: LeetCode 146 - LRU Cache

Key Idea:
The key idea is to implement an LRU (Least Recently Used) cache, which efficiently stores and manages a fixed number of items while removing the least recently used item when the capacity is exceeded.

Approach:
1. Use a combination of a HashMap and a doubly linked list to implement the LRU cache.
2. The HashMap stores the key-value pairs, where the key is used for quick access to the corresponding node in the linked list.
3. The doubly linked list represents the order of usage, with the most recently used item at the front (head) and the least recently used item at the end (tail).
4. When an item is accessed (get or put operation), move it to the front of the linked list to mark it as the most recently used.
5. When the cache capacity is exceeded during a put operation, remove the least recently used item from both the HashMap and the linked list.
6. The doubly linked list is used to efficiently manage the order of usage by adjusting the pointers when items are accessed or removed.

Time Complexity:
- Get Operation: O(1) - Accessing a value in the HashMap is an O(1) operation.
- Put Operation: O(1) - Inserting or updating a value in the HashMap is an O(1) operation.
Overall, the operations are O(1) because the linked list operations (removing and adding nodes) are constant time.

Space Complexity:
O(capacity), where the space complexity is determined by the capacity of the LRU cache.
*/

use std::collections::{HashMap, HashSet, VecDeque};

struct LRUCache {
    deque: VecDeque<i32>,
    cache: HashMap<i32, i32>,
    capacity: usize,
    keys: HashSet<i32>,
}

impl LRUCache {
    fn new(capacity: i32) -> Self {
        Self {
            deque: VecDeque::new(),
            cache: HashMap::new(),
            capacity: capacity as usize,
            keys: HashSet::new(),
        }
    }

    fn get(&mut self, key: i32) -> i32 {
        if self.keys.contains(&key) {
            let index = self.deque.iter().position(|&x| x == key).unwrap();
            self.deque.remove(index);
            self.deque.push_front(key);
            *self.cache.get(&key).unwrap()
        } else {
            -1
        }
    }
```

```
    fn put(&mut self, key: i32, value: i32) {
        if self.keys.contains(&key) {
            let index = self.deque.iter().position(|&x| x == key).unwrap();
            self.deque.remove(index);
            self.deque.push_front(key);
            self.cache.insert(key, value);
        } else {
            if self.cache.len() == self.capacity {
                if let Some(last) = self.deque.pop_back() {
                    self.cache.remove(&last);
                    self.keys.remove(&last);
                }
            }
            self.cache.insert(key, value);
            self.deque.push_front(key);
            self.keys.insert(key);
        }
    }
}

/*
// Faster solution

use std::collections::{HashMap, VecDeque};

struct LRUCache {
    cache: HashMap<i32, (i32, u32)>,
    queue: VecDeque<(i32, u32)>,
    time: u32,
    capacity: usize,
}

impl LRUCache {
    fn new(capacity: i32) -> Self {
        let capacity = capacity as usize;
        Self {
            cache: HashMap::with_capacity(capacity),
            queue: VecDeque::new(),
            time: 0,
            capacity,
        }
    }

    fn get(&mut self, key: i32) -> i32 {
        match self.cache.get(&key) {
            None => -1,
            Some(&(value, _)) => {
                self.update(key, value);
                value
            }
        }
    }

    fn put(&mut self, key: i32, value: i32) {
        if !self.cache.contains_key(&key) && self.cache.len() == self.capacity {
            let (key, _) = self.queue.pop_front().unwrap();
            self.cache.remove(&key);
        }
        self.update(key, value);
    }

    fn update(&mut self, key: i32, value: i32) {
        self.time = self.time.wrapping_add(1);
```

```
        self.cache.insert(key, (value, self.time));
        self.queue.push_back((key, self.time));
        loop {
            if let Some(&(key, time1)) = self.queue.front() {
                if let Some(&(_, time2)) = self.cache.get(&key) {
                    if time1 == time2 {
                        break;
                    }
                }
            }
            self.queue.pop_front();
        }
    }
}
*/
```

43. 06_Linked_List/10_Merge_K_Sorted_Lists/0023-merge-k-sorted-lists.rs

```
/*
Problem: LeetCode 23 - Merge k Sorted Lists

Key Idea:
The key idea is to use a min-heap (priority queue) to efficiently merge the k sorted lists.

Approach:
1. Initialize an empty min-heap (priority queue).
2. Push the head of each sorted list (if it's not None) into the min-heap along with its list index.
3. Initialize a dummy node as the head of the merged list.
4. While the min-heap is not empty:
   - Pop the smallest element (with the list index) from the min-heap.
   - Add this element to the merged list.
    - Move the pointer in the corresponding list to its next element (if it exists) and push it into the min-heap.
5. Return the merged list.

Time Complexity:
O(N log k), where N is the total number of elements across all lists, and k is the number of lists.
   - The min-heap can have at most k elements, and extracting the minimum element takes O(log k) time.
   - We perform this extraction for all N elements.

Space Complexity:
O(k), as the min-heap can have at most k elements.
*/

//                                                                                                  Solution: https://leetcode.com/problems/merge-k-sorted-lists/solutions/1427861/Rust-Two-solutions-using-priority-queue-and-merging-the-lists-one-by-one/

use std::cmp::{Ordering, Reverse};
use std::collections::BinaryHeap;

impl Solution {
    pub fn merge_k_lists(lists: Vec<Option<Box<ListNode>>>) -> Option<Box<ListNode>> {
        let mut merged_head = ListNode::new(0);
        let mut tail = &mut merged_head.next;

        // Create a min-heap to keep track of the smallest nodes
        let mut min_heap = BinaryHeap::with_capacity(lists.len());

        // Initialize the min-heap with the heads of all lists
        for list in lists {
            if let Some(node) = list {
                min_heap.push(Reverse(NodeWrapper(node)));
            }
        }

        // Merge nodes from the min-heap
        while let Some(Reverse(mut node)) = min_heap.pop() {
            // Push the next node from the same list into the min-heap if it exists
            if let Some(next) = node.0.next.take() {
                min_heap.push(Reverse(NodeWrapper(next)));
            }

            // Add the current smallest node to the merged list
            tail.replace(node.0);
```

```
            tail = &mut tail.as_mut().unwrap().next;
        }

        // Return the merged list starting from the real head (skip the dummy head)
        merged_head.next.take()
    }
}

struct NodeWrapper(Box<ListNode>);
impl PartialOrd<NodeWrapper> for NodeWrapper {
    fn partial_cmp(&self, other: &NodeWrapper) -> Option<Ordering> {
        self.0.val.partial_cmp(&other.0.val)
    }
}

impl Ord for NodeWrapper {
    fn cmp(&self, other: &Self) -> Ordering {
        self.0.val.cmp(&other.0.val)
    }
}

impl PartialEq<NodeWrapper> for NodeWrapper {
    fn eq(&self, other: &NodeWrapper) -> bool {
        self.0.val.eq(&other.0.val)
    }
}

impl Eq for NodeWrapper {}
```

44. 06_Linked_List/11_Reverse_Nodes_In_K_Group/0025-reverse-nodes-in-k-group.r

```
/*
Problem: LeetCode 25 - Reverse Nodes in k-Group

Key Idea:
The key idea is to reverse k-node groups in a linked list using recursion. We need to reverse each group, connect them together, and stop when there are fewer than k nodes left.

Approach:
1. Initialize `original_head` as a mutable reference to the head of the original linked list. This reference will be used to keep track of the current group of nodes.
2. Advance the `current_group` pointer by k nodes using a loop. This loop serves two purposes:
    - It ensures that `current_group` points to the kth node if there are at least k nodes remaining.
    - If there are fewer than k nodes left, return the `original_head` as there is no need to reverse this group.
3. Recursively reverse the next group of nodes (i.e., the portion of the list after the current k nodes). This is done by calling `Self::reverse_k_group` with `current_group` as the new head of the list. The result is stored in `reversed_next_group`.
4. Reverse the current group of k nodes:
   - Initialize a new variable `new_head` as `None` to store the new head of the reversed group.
   - Use a loop to reverse the nodes one by one:
     - Take a node from `original_head`.
     - Update `original_head` to point to the next node.
     - Set the `next` pointer of the taken node to point to the current `new_head`.
     - Update `new_head` to point to the taken node.
   - After the loop, `new_head` will contain the reversed k nodes.
5. Connect the reversed current group to the reversed next group:
   - Initialize a mutable reference `current` to the end of `new_head`.
   - Traverse `current` until it reaches the end of the reversed current group (where `next` is `None`).
   - Set `current`'s `next` to point to `reversed_next_group`.
6. Return the `new_head`, which is the head of the fully reversed linked list.

Time Complexity:
O(N), where N is the number of nodes in the linked list. We traverse each node exactly once.

Space Complexity:
O(1), as we use a constant amount of extra space for pointers and variables.
*/

impl Solution {
    pub fn reverse_k_group(head: Option<Box<ListNode>>, k: i32) -> Option<Box<ListNode>> {
        let mut original_head = head;
        let mut current_group = &mut original_head;

        // Advance the current_group pointer by k nodes
        for _ in 0..k {
            if let Some(node) = current_group {
                current_group = &mut node.next;
            } else {
                // If there are fewer than k nodes left, return the original head
                return original_head;
            }
        }

        // Recursively reverse the next group and get its head
        let mut reversed_next_group = Self::reverse_k_group(current_group.take(), k);

        // Reverse the current group and attach it to the reversed_next_group
        let mut new_head = None;
```

```
        while let Some(mut node) = original_head.take() {
            original_head = node.next.take();
            node.next = new_head;
            new_head = Some(node);
        }

        // Connect the reversed current group to the reversed next group
        let mut current = &mut new_head;
        while current.as_ref().is_some() {
            current = &mut current.as_mut().unwrap().next;
        }
        *current = reversed_next_group;

        new_head
    }
}
```

45. 07_Trees/01_Invert_Binary_Tree/0226-invert-binary-tree.rs

```
/*
Problem: LeetCode 226 - Invert Binary Tree

Key Idea:
The key idea is to swap the left and right subtrees of each node.

Approach:
1. Create a recursive function 'invert_tree' that takes a reference to the root of the binary tree.
2. If the root is None, return None (base case).
3. Swap the left and right subtrees of the root node by recursively calling 'invert_tree' on both the left and right subtrees.
4. Update the root node's left child with the result of inverting the right subtree and the right child with the result of inverting the left subtree.
5. Return the root node after the inversion.
6. In the main function, return the result of 'invert_tree' on the input root.

Time Complexity:
O(n), where 'n' is the number of nodes in the binary tree. We visit each node once.

Space Complexity:
O(h), where 'h' is the height of the binary tree. This space is used for the recursive call stack.
*/

use std::cell::RefCell;
use std::rc::Rc;

impl Solution {
    pub fn invert_tree(root: Option<Rc<RefCell<TreeNode>>>) -> Option<Rc<RefCell<TreeNode>>> {
        match root {
            None => None,
            Some(node) => {
                let mut node = node.borrow_mut();
                let left = node.left.take();
                let right = node.right.take();
                node.left = Solution::invert_tree(right);
                node.right = Solution::invert_tree(left);
                Some(Rc::new(RefCell::new(TreeNode {
                    val: node.val,
                    left: node.left.clone(),
                    right: node.right.clone(),
                })))
            }
        }
    }
}
```

46. 07_Trees/02_Maximum_Depth_of_Binary_Tree/0104-maximum-depth-of-binary-tre

```
/*
Problem: LeetCode 104 - Maximum Depth of Binary Tree

Key Idea:
The key idea is to use recursive depth-first traversal. For each node, calculate the depth of its left and right subtrees recursively, and return the maximum of these depths plus one as the depth of the current node.

Approach:
1. Create a recursive function 'max_depth' that takes a reference to the root of the binary tree.
2. If the root is None, return 0 (base case).
3. Recursively calculate the maximum depth of the left subtree by calling 'max_depth' on the left child.
4. Recursively calculate the maximum depth of the right subtree by calling 'max_depth' on the right child.
5. Return the maximum depth among the left and right subtrees, incremented by 1 to account for the current level.
6. In the main function, return the result of 'max_depth' on the input root.

Time Complexity:
O(n), where 'n' is the number of nodes in the binary tree. We visit each node once.

Space Complexity:
O(h), where 'h' is the height of the binary tree. This space is used for the recursive call stack.
*/

use std::cell::RefCell;
use std::rc::Rc;

impl Solution {
    pub fn max_depth(root: Option<Rc<RefCell<TreeNode>>>) -> i32 {
        match root {
            None => 0,
            Some(node) => {
                let node = node.borrow();
                let left_depth = Solution::max_depth(node.left.clone());
                let right_depth = Solution::max_depth(node.right.clone());
                1 + left_depth.max(right_depth)
            }
        }
    }
}
```

47. 07_Trees/03_Diameter_of_Binary_Tree/0543-diameter-of-binary-tree.rs

```
/*
Problem: LeetCode 543 - Diameter of Binary Tree

Key Idea:
The key idea is to use depth-first traversal. For each node, calculate the sum of the depths of its left and right subtrees. The diameter will be the maximum of these sums or the maximum diameter found in any subtree.

Approach:
1. Create a recursive function 'diameter_of_binary_tree' that takes a reference to the root of the binary tree and returns a tuple (depth, diameter).
2. If the root is None, return (0, 0) (base case).
3. Recursively calculate the depth and diameter of the left subtree by calling 'diameter_of_binary_tree' on the left child.
4. Recursively calculate the depth and diameter of the right subtree by calling 'diameter_of_binary_tree' on the right child.
5. Calculate the diameter as the maximum of three values:
   - The diameter of the left subtree.
   - The diameter of the right subtree.
    - The sum of the depths of the left and right subtrees plus 1 (to account for the current node).
6. Calculate the depth as the maximum of the depths of the left and right subtrees plus 1 (to account for the current node).
7. Return the calculated depth and diameter as a tuple.
8. In the main function, return the diameter component of the result of 'diameter_of_binary_tree' on the input root.

Time Complexity:
O(n), where 'n' is the number of nodes in the binary tree. We visit each node once.

Space Complexity:
O(h), where 'h' is the height of the binary tree. This space is used for the recursive call stack.
*/

use std::cell::RefCell;
use std::rc::Rc;

impl Solution {
    pub fn diameter_of_binary_tree(root: Option<Rc<RefCell<TreeNode>>>) -> i32 {
        fn diameter_of_binary_tree(
            root: &Option<Rc<RefCell<TreeNode>>>,
            max_diameter: &mut i32,
        ) -> (i32, i32) {
            match root {
                None => (0, 0),
                Some(node) => {
                    let node = node.borrow();
                    let (left_depth, left_diameter) =
                        diameter_of_binary_tree(&node.left, max_diameter);
                    let (right_depth, right_diameter) =
                        diameter_of_binary_tree(&node.right, max_diameter);

                    let depth = left_depth.max(right_depth) + 1;
                    let diameter = left_diameter
                        .max(right_diameter)
                        .max(left_depth + right_depth);

                    *max_diameter = (*max_diameter).max(diameter);

                    (depth, diameter)
```

```
                }
            }
        }

        let mut max_diameter = 0;
        let (_, diameter) = diameter_of_binary_tree(&root, &mut max_diameter);
        diameter
    }
}
```

48. 07_Trees/04_Balanced_Binary_Tree/0110-balanced-binary-tree.rs

```
/*
Problem: LeetCode 110 - Balanced Binary Tree

Key Idea:
The key idea is to use recursive depth-first traversal. For each node, calculate the heights of its left and right subtrees recursively. If both subtrees are balanced and the height difference is at most one, the tree is balanced.

Approach:
1. Create a recursive function 'is_balanced' that takes a reference to the root of the binary tree and returns a tuple (is_balanced, depth).
2. If the root is None, return (true, 0) (base case).
3. Recursively check if the left subtree is balanced and get its depth.
4. Recursively check if the right subtree is balanced and get its depth.
5. Calculate the depth of the current node as the maximum of the depths of the left and right subtrees plus 1.
6. Check if the left subtree is balanced, the right subtree is balanced, and the absolute difference of their depths is at most 1.
7. Return whether the tree rooted at the current node is balanced and its depth as a tuple.
8. In the main function, return the 'is_balanced' component of the result of 'is_balanced' on the input root.

Time Complexity:
O(n), where 'n' is the number of nodes in the binary tree. We visit each node once.

Space Complexity:
O(h), where 'h' is the height of the binary tree. This space is used for the recursive call stack.
*/

use std::cell::RefCell;
use std::rc::Rc;

impl Solution {
    pub fn is_balanced(root: Option<Rc<RefCell<TreeNode>>>) -> bool {
        fn is_balanced(root: &Option<Rc<RefCell<TreeNode>>>) -> (bool, i32) {
            match root {
                None => (true, 0),
                Some(node) => {
                    let node = node.borrow();
                    let (left_balanced, left_depth) = is_balanced(&node.left);
                    let (right_balanced, right_depth) = is_balanced(&node.right);

                    let depth = left_depth.max(right_depth) + 1;
                    let is_current_balanced =
                        left_balanced && right_balanced && (left_depth - right_depth).abs() <= 1;

                    (is_current_balanced, depth)
                }
            }
        }

        let (is_balanced, _) = is_balanced(&root);
        is_balanced
    }
}
```

49. 07_Trees/05_Same_Tree/0100-same-tree.rs

```
/*
Problem: LeetCode 100 - Same Tree

Key Idea:
The key idea is to use recursive depth-first traversal. For each pair of corresponding nodes in the trees, check if their values are equal and recursively check the left and right subtrees. If all corresponding nodes are equal, the trees are identical.

Approach:
1. Create a recursive function 'is_same_tree' that takes references to the roots of two binary trees as input and returns a boolean.
2. If both roots are None, return true (base case).
3. If one of the roots is None while the other is not, return false (as they cannot be identical).
4. Check if the values of the current nodes are equal.
5. Recursively call 'is_same_tree' for the left subtrees and right subtrees.
6. Return the logical AND of the results from steps 4 and 5.
7. In the main function, return the result of 'is_same_tree' on the input roots.

Time Complexity:
O(n), where 'n' is the number of nodes in the binary trees. We visit each node once.

Space Complexity:
O(h), where 'h' is the height of the binary trees. This space is used for the recursive call stack.
*/

use std::cell::RefCell;
use std::rc::Rc;

impl Solution {
    pub fn is_same_tree(
        p: Option<Rc<RefCell<TreeNode>>>,
        q: Option<Rc<RefCell<TreeNode>>>,
    ) -> bool {
        fn is_same_tree(
            p: &Option<Rc<RefCell<TreeNode>>>,
            q: &Option<Rc<RefCell<TreeNode>>>,
        ) -> bool {
            match (p, q) {
                (None, None) => true,
                (Some(p_node), Some(q_node)) => {
                    let p_node = p_node.borrow();
                    let q_node = q_node.borrow();
                    p_node.val == q_node.val
                        && is_same_tree(&p_node.left, &q_node.left)
                        && is_same_tree(&p_node.right, &q_node.right)
                }
                _ => false,
            }
        }

        is_same_tree(&p, &q)
    }
}
```

50. 07_Trees/06_Subtree_of_Another_Tree/0572-subtree-of-another-tree.rs

```
/*
Problem: LeetCode 572 - Subtree of Another Tree

Key Idea:
The key idea is to use a recursive approach. Start by checking if the current nodes in both trees are equal. If they are, recursively check if the left and right subtrees are also identical. Repeat this process until you find a matching subtree or exhaust all possibilities in the main tree.

Approach:
1. Create a recursive function 'is_subtree' that takes references to the roots of two binary trees as input and returns a boolean.
2. If the first tree's root is None, return false (base case).
3. Check if the current nodes of both trees are equal.
4. If the nodes are equal, recursively check if the left and right subtrees of both trees are also subtrees.
5. If any of the conditions in step 3 or 4 is true, return true.
6. In the main function, return the result of 'is_subtree' on the input trees.

Time Complexity:
O(m * n), where 'm' is the number of nodes in the first tree, and 'n' is the number of nodes in the second tree. In the worst case, we may have to check all nodes in the first tree against the second tree.

Space Complexity:
O(h), where 'h' is the height of the first tree. This space is used for the recursive call stack.
*/

use std::cell::RefCell;
use std::rc::Rc;

impl Solution {
    pub fn is_subtree(s: Option<Rc<RefCell<TreeNode>>>, t: Option<Rc<RefCell<TreeNode>>>) -> bool {
        fn is_subtree(
            s: &Option<Rc<RefCell<TreeNode>>>,
            t: &Option<Rc<RefCell<TreeNode>>>,
        ) -> bool {
            match (s, t) {
                (None, None) => true,
                (Some(s_node), Some(t_node)) => {
                    let s_node = s_node.borrow();
                    let t_node = t_node.borrow();
                    s_node.val == t_node.val
                        && is_subtree(&s_node.left, &t_node.left)
                        && is_subtree(&s_node.right, &t_node.right)
                }
                _ => false,
            }
        }

        fn is_subtree_helper(
            s: &Option<Rc<RefCell<TreeNode>>>,
            t: &Option<Rc<RefCell<TreeNode>>>,
        ) -> bool {
            match s {
                Some(s_node) => {
                    let s_node = s_node.borrow();
                    is_subtree(s, t)
                        || is_subtree_helper(&s_node.left, t)
```

```
                        || is_subtree_helper(&s_node.right, t)
                }
                None => false,
            }
        }

        is_subtree_helper(&s, &t)
    }
}
```

51. 07_Trees/07_Lowest_Common_Ancestor_of_a_Binary_Search_Tree/0235-lowest-

```
/*
Problem: LeetCode 235 - Lowest Common Ancestor of a Binary Search Tree

Key Idea:
The key idea is to utilize the property of a Binary Search Tree (BST), where the nodes in the left
subtree of a node are smaller than the node, and nodes in the right subtree are larger than the
node. This property allows us to efficiently find the lowest common ancestor.

Approach:
1. Start at the root of the BST.
2. Traverse down the tree.
3. If both p and q are smaller than the current node's value, move to the left subtree.
4. If both p and q are larger than the current node's value, move to the right subtree.
5. Otherwise, the current node is the lowest common ancestor.

Time Complexity:
O(log N) on average, where 'N' is the number of nodes in the BST. In the worst case, where the
tree is completely unbalanced, it can be O(N).

Space Complexity:
O(1), as we are using a constant amount of space for traversal.
*/

use std::cell::RefCell;
use std::rc::Rc;

type TreeNodeRef = Option<Rc<RefCell<TreeNode>>>;

impl Solution {
    pub fn lowest_common_ancestor(
        root: TreeNodeRef,
        p: TreeNodeRef,
        q: TreeNodeRef,
    ) -> TreeNodeRef {
        let p_value = p.as_ref().unwrap().borrow().val;
        let q_value = q.as_ref().unwrap().borrow().val;
        let mut current_node = root;

        while let Some(node) = current_node {
            let mut node_ref = node.borrow_mut();

            if p_value < node_ref.val && q_value < node_ref.val {
                current_node = node_ref.left.take();
            } else if p_value > node_ref.val && q_value > node_ref.val {
                current_node = node_ref.right.take();
            } else {
                return Some(node.clone());
            }
        }

        None
    }
}
```

52. 07_Trees/08_Binary_Tree_Level_Order_Traversal/0102-binary-tree-level-order-tra

```
/*
Problem: LeetCode 102 - Binary Tree Level Order Traversal

Key Idea:
The key idea is to use a breadth-first traversal (commonly implemented using a queue). Start with
the root node, enqueue it, and process nodes level by level. For each level, enqueue the children
of the nodes at the current level. Keep doing this until the queue is empty, forming a list of
lists representing each level's nodes as you go.

Approach:
1. Start with an empty result vector to store the level order traversal.
2. Initialize a queue for a breadth-first search (BFS).
3. Push the root node into the queue.
4. While the queue is not empty:
   - Initialize a temporary vector to store values at the current level.
   - Calculate the number of nodes at the current level by taking the length of the queue.
   - Iterate for each node in the current level (based on the count):
     - Pop a node from the queue.
     - Push its value into the temporary vector.
     - Push its left and right children into the queue if they exist.
   - Push the temporary vector into the result vector.
5. Return the result vector.

Time Complexity:
The time complexity is O(n), where 'n' is the number of nodes in the binary tree. We visit each
node once during the BFS traversal.

Space Complexity:
The space complexity is O(m), where 'm' is the maximum number of nodes at any level. In the worst
case, 'm' can be the number of leaf nodes, which is O(n/2) in a balanced tree, making the space
complexity O(n).
*/

use std::cell::RefCell;
use std::collections::VecDeque;
use std::rc::Rc;

impl Solution {
    pub fn level_order(root: Option<Rc<RefCell<TreeNode>>>) -> Vec<Vec<i32>> {
        let mut result: Vec<Vec<i32>> = Vec::new();

        if let Some(node) = root {
            let mut queue = VecDeque::new();
            queue.push_back(node);

            while !queue.is_empty() {
                let mut current_level: Vec<i32> = Vec::new();
                let level_size = queue.len();

                for _ in 0..level_size {
                    if let Some(node) = queue.pop_front() {
                        let node_ref = node.borrow();
                        current_level.push(node_ref.val);

                        if let Some(left) = &node_ref.left {
                            queue.push_back(left.clone());
                        }

                        if let Some(right) = &node_ref.right {
                            queue.push_back(right.clone());
```

```
                        }
                    }
                }

                result.push(current_level);
            }
        }

        result
    }
}
```

53. 07_Trees/09_Binary_Tree_Right_Side_View/0199-binary-tree-right-side-view.rs

```
/*
Problem: LeetCode 199 - Binary Tree Right Side View

Key Idea:
The key idea is to perform a level order traversal of the binary tree and only add the rightmost
node at each level to the result.

Approach:
1. Start with an empty result vector to store the right side view of the binary tree.
2. Initialize a queue for a breadth-first search (BFS).
3. Push the root node into the queue.
4. While the queue is not empty:
   - Initialize a variable to store the rightmost node's value at the current level.
   - Calculate the number of nodes at the current level by taking the length of the queue.
   - Iterate for each node in the current level (based on the count):
     - Pop a node from the queue.
     - Update the rightmost value to the current node's value.
     - Push its left and right children into the queue if they exist.
   - Push the rightmost value into the result vector.
5. Return the result vector.

Time Complexity:
The time complexity is O(n), where 'n' is the number of nodes in the binary tree. We visit each
node once during the BFS traversal.

Space Complexity:
The space complexity is O(m), where 'm' is the maximum number of nodes at any level. In the worst
case, 'm' can be the number of leaf nodes, which is O(n/2) in a balanced tree, making the space
complexity O(n).
*/

use std::cell::RefCell;
use std::collections::VecDeque;
use std::rc::Rc;

impl Solution {
    pub fn right_side_view(root: Option<Rc<RefCell<TreeNode>>>) -> Vec<i32> {
        let mut result: Vec<i32> = Vec::new();

        if let Some(node) = root {
            let mut queue = VecDeque::new();
            queue.push_back(node);

            while !queue.is_empty() {
                let mut rightmost_val = 0;
                let level_size = queue.len();

                for i in 0..level_size {
                    if let Some(node) = queue.pop_front() {
                        let node_ref = node.borrow();

                        if i == level_size - 1 {
                            // Rightmost node at the current level
                            rightmost_val = node_ref.val;
                        }

                        if let Some(left) = &node_ref.left {
                            queue.push_back(left.clone());
                        }
```

```
                    if let Some(right) = &node_ref.right {
                        queue.push_back(right.clone());
                    }
                }
            }

            result.push(rightmost_val);
        }
    }

    result
  }
}
```

54. 07_Trees/10_Count_Good_Nodes_In_Binary_Tree/1448-count-good-nodes-in-bina

```
/*
Problem: LeetCode 1448 - Count Good Nodes in Binary Tree

Key Idea:
The key idea is to perform a depth-first search (DFS) traversal of the binary tree while keeping track of the maximum value encountered in the path from the root to the current node. If the current node's value is greater than or equal to the maximum value encountered so far, it is considered a "good" node, and we increment the count.

Approach:
1. Start with a count variable initialized to 0 to keep track of good nodes.
2. Define a recursive DFS function that takes the current node, the maximum value encountered so far, and updates the count.
3. In the DFS function:
   - If the current node is None, return.
    - Calculate the new maximum value as the maximum of the current node's value and the maximum value encountered so far.
   - If the current node's value is greater than or equal to the new maximum value, increment the count.
    - Recursively call the DFS function for the left and right children, passing the new maximum value.
4. Call the DFS function starting from the root node with an initial maximum value of negative infinity.
5. Return the count, which represents the number of good nodes.

Time Complexity:
The time complexity is O(n), where 'n' is the number of nodes in the binary tree. We visit each node once during the DFS traversal.

Space Complexity:
The space complexity is O(h), where 'h' is the height of the binary tree. In the worst case, 'h' can be equal to 'n' in a skewed tree, making the space complexity O(n). In a balanced tree, 'h' is O(log n).
*/

use std::cell::RefCell;
use std::rc::Rc;

impl Solution {
    pub fn good_nodes(root: Option<Rc<RefCell<TreeNode>>>) -> i32 {
        fn dfs(node: Option<Rc<RefCell<TreeNode>>>, max_val: i32, count: &mut i32) {
            if let Some(n) = node {
                let node_ref = n.borrow();
                let new_max = i32::max(max_val, node_ref.val);

                if node_ref.val >= max_val {
                    *count += 1;
                }

                dfs(node_ref.left.clone(), new_max, count);
                dfs(node_ref.right.clone(), new_max, count);
            }
        }

        let mut count = 0;
        dfs(root, i32::min_value(), &mut count);
        count
    }
}
```

55. 07_Trees/11_Validate_Binary_Search_Tree/0098-validate-binary-search-tree.rs

```
/*
Problem: LeetCode 98 - Validate Binary Search Tree

Key Idea:
The key idea is to perform an in-order traversal of the binary tree while keeping track of the previous node's value. In a valid binary search tree (BST), an in-order traversal should yield values in ascending order. Therefore, we can compare each node's value with the previous node's value to check if the tree is a valid BST.

Approach:
1. Define a recursive function to perform an in-order traversal of the binary tree.
2. In the function:
   - If the current node is None, return true (empty subtree is valid).
   - Recursively call the function for the left child.
   - Check if the current node's value is greater than the previous node's value. If not, return false.
   - Update the previous node's value to the current node's value.
   - Recursively call the function for the right child.
3. Start the traversal from the root node with the previous value initialized to negative infinity.
4. If the traversal completes without any violations, return true (valid BST); otherwise, return false.

Time Complexity:
The time complexity is O(n), where 'n' is the number of nodes in the binary tree. We visit each node once during the in-order traversal.

Space Complexity:
The space complexity is O(h), where 'h' is the height of the binary tree. In the worst case, 'h' can be equal to 'n' in a skewed tree, making the space complexity O(n). In a balanced tree, 'h' is O(log n).
*/

use std::cell::RefCell;
use std::rc::Rc;

impl Solution {
    pub fn is_valid_bst(root: Option<Rc<RefCell<TreeNode>>>) -> bool {
        fn is_valid(node: Option<Rc<RefCell<TreeNode>>>, prev: &mut i64) -> bool {
            if let Some(n) = node {
                let node_ref = n.borrow();
                if !is_valid(node_ref.left.clone(), prev) {
                    return false;
                }
                if node_ref.val as i64 <= *prev {
                    return false;
                }
                *prev = node_ref.val as i64;
                is_valid(node_ref.right.clone(), prev)
            } else {
                true
            }
        }

        let mut prev = i64::min_value();
        is_valid(root, &mut prev)
    }
}
```

56. 07_Trees/12_Kth_Smallest_Element_In_a_BST/0230-kth-smallest-element-in-a-bst

```
/*
Problem: LeetCode 230 - Kth Smallest Element in a BST

Key Idea:
The key idea is to perform an in-order traversal of the binary search tree (BST) while maintaining a count of visited nodes. When the count reaches 'k', we have found the kth smallest element in the BST.

Approach:
1. Define a recursive function to perform an in-order traversal of the BST.
2. In the function:
   - If the current node is None, return None.
   - Recursively call the function for the left child.
   - Increment the count of visited nodes.
   - If the count equals 'k', return the value of the current node.
   - Recursively call the function for the right child.
3. Start the traversal from the root node with the count initialized to 0.
4. If the traversal completes, return None (k is out of bounds).

Time Complexity:
The time complexity is O(h + k), where 'h' is the height of the BST. In the worst case, the traversal goes down to the leftmost leaf node (height 'h') and then back up to find the kth smallest element.

Space Complexity:
The space complexity is O(h) due to the recursion stack. In the worst case, 'h' can be equal to 'n' in a skewed tree, making the space complexity O(n). In a balanced tree, 'h' is O(log n).
*/

use std::cell::RefCell;
use std::rc::Rc;

impl Solution {
    pub fn kth_smallest(root: Option<Rc<RefCell<TreeNode>>>, k: i32) -> i32 {
        fn in_order_traversal(
            node: Option<Rc<RefCell<TreeNode>>>,
            k: i32,
            count: &mut i32,
        ) -> Option<i32> {
            if let Some(n) = node {
                let node_ref = n.borrow();
                let left_result = in_order_traversal(node_ref.left.clone(), k, count);
                if let Some(val) = left_result {
                    return Some(val);
                }
                *count += 1;
                if *count == k {
                    return Some(node_ref.val);
                }
                in_order_traversal(node_ref.right.clone(), k, count)
            } else {
                None
            }
        }

        let mut count = 0;
        in_order_traversal(root, k, &mut count).unwrap()
    }
}
```

57. 07_Trees/13_Construct_Binary_Tree_from_Preorder_and_Inorder_Traversal/0105

```
/*
Problem: LeetCode 105 - Construct Binary Tree from Preorder and Inorder Traversal

Key Idea:
The key idea is to use recursion to construct the binary tree. We can determine the root node from the preorder traversal and then find its position in the inorder traversal to divide the inorder traversal into left and right subtrees. We repeat this process for each subtree recursively.

Approach:
1. Create a hashmap that maps values from the inorder traversal to their indices. This will allow us to quickly find the position of the root node in the inorder traversal.
2. Define a recursive function that takes the following parameters:
   - `pre_start` and `pre_end`: The start and end indices in the preorder traversal.
   - `in_start` and `in_end`: The start and end indices in the inorder traversal.
3. In the function:
   - If `pre_start > pre_end`, return None (no more nodes to construct).
   - Get the value of the root node from the current `pre_start` index.
   - Find the position of the root value in the inorder traversal using the hashmap.
   - Calculate the number of nodes in the left subtree (the difference between the root's position and `in_start`).
   - Create the root node.
   - Recursively call the function for the left subtree and right subtree, adjusting the indices accordingly.
   - Return the root node.
4. Start the recursion with `pre_start = 0`, `pre_end = n - 1`, `in_start = 0`, and `in_end = n - 1`, where `n` is the number of nodes.
5. The result is the root node of the constructed binary tree.

Time Complexity:
The time complexity is O(n), where 'n' is the number of nodes, as we visit each node once during the construction.

Space Complexity:
The space complexity is O(n) due to the recursion stack.
*/

use std::cell::RefCell;
use std::collections::HashMap;
use std::rc::Rc;

impl Solution {
    pub fn build_tree(preorder: Vec<i32>, inorder: Vec<i32>) -> Option<Rc<RefCell<TreeNode>>> {
        fn build(
            preorder: &[i32],
            inorder: &[i32],
            pre_start: usize,
            pre_end: usize,
            in_start: usize,
            in_end: usize,
            inorder_map: &HashMap<i32, usize>,
        ) -> Option<Rc<RefCell<TreeNode>>> {
            if pre_start > pre_end {
                return None;
            }

            let root_val = preorder[pre_start];
            let root_inorder_index = *inorder_map.get(&root_val).unwrap();
            let left_size = root_inorder_index - in_start;

            let mut root = TreeNode::new(root_val);
```

```
            root.left = build(
                preorder,
                inorder,
                pre_start + 1,
                pre_start + left_size,
                in_start,
                root_inorder_index - 1,
                inorder_map,
            );
            root.right = build(
                preorder,
                inorder,
                pre_start + left_size + 1,
                pre_end,
                root_inorder_index + 1,
                in_end,
                inorder_map,
            );

            Some(Rc::new(RefCell::new(root)))
        }

        let mut inorder_map = HashMap::new();
        for (i, &val) in inorder.iter().enumerate() {
            inorder_map.insert(val, i);
        }

        build(
            &preorder,
            &inorder,
            0,
            preorder.len() - 1,
            0,
            inorder.len() - 1,
            &inorder_map,
        )
    }
}
```

58. 07_Trees/14_Binary_Tree_Maximum_Path_Sum/0124-binary-tree-maximum-path-s

```
/*
Problem: LeetCode 124 - Binary Tree Maximum Path Sum

Key Idea:
The key idea is to use a recursive approach to calculate the maximum path sum for each subtree while keeping track of the maximum path sum across all subtrees. At each node, we have three choices:
1. Include only the current node.
2. Include the current node and the maximum path sum from the left subtree.
3. Include the current node and the maximum path sum from the right subtree.

Approach:
1. Define a recursive function that takes a reference to the current node and returns the maximum path sum that starts from this node and goes downwards (either left or right).
2. In the function:
   - If the current node is None, return 0.
   - Recursively calculate the maximum path sum for the left subtree and the right subtree using the same function.
   - Calculate the maximum path sum that includes the current node and goes either left or right, or just the current node itself.
   - Update the global maximum path sum if the new path sum is greater.
   - Return the maximum path sum that starts from the current node and goes either left or right.
3. Initialize a global variable to store the maximum path sum with a value of negative infinity.
4. Start the recursion from the root node.
5. The result is the global maximum path sum.

Time Complexity:
The time complexity is O(n), where 'n' is the number of nodes in the binary tree, as we visit each node once.

Space Complexity:
The space complexity is O(h), where 'h' is the height of the binary tree. In the worst case (unbalanced tree), 'h' can be equal to 'n', but in the average case (balanced tree), 'h' is O(log n).
*/

use std::cell::RefCell;
use std::cmp::max;
use std::rc::Rc;

impl Solution {
    pub fn max_path_sum(root: Option<Rc<RefCell<TreeNode>>>) -> i32 {
        fn max_path_sum_helper(node: Option<Rc<RefCell<TreeNode>>>, max_sum: &mut i32) -> i32 {
            if let Some(inner) = node {
                let val = inner.borrow().val;
                let left_sum = max_path_sum_helper(inner.borrow_mut().left.take(), max_sum);
                let right_sum = max_path_sum_helper(inner.borrow_mut().right.take(), max_sum);

                // Calculate the maximum path sum including the current node.
                let node_sum = max(val, max(val + left_sum, val + right_sum));

                // Update the global maximum path sum.
                *max_sum = max(*max_sum, max(node_sum, val + left_sum + right_sum));

                // Return the maximum path sum starting from the current node.
                return node_sum;
            }

            // If the node is None, return 0.
            0
```

```
        }

        let mut max_sum = i32::min_value();
        max_path_sum_helper(root, &mut max_sum);
        max_sum
    }
}
```

59. 07_Trees/15_Serialize_and_Deserialize_Binary_Tree/0297-serialize-and-deserializ

```
/*
Problem: LeetCode 297 - Serialize and Deserialize Binary Tree

Key Idea:
The key idea is to perform a depth-first traversal (preorder) of the binary tree to serialize it
into a string representation, and then parse the string to deserialize it back into a binary tree.

Approach:
1. Serialization:
   - Implement a recursive function that serializes the binary tree into a string.
   - In the serialization function:
     - If the current node is None, append "null" to the serialized string.
      - Otherwise, append the node's value to the serialized string and recursively serialize its
left and right subtrees.
2. Deserialization:
   - Implement a deserialization function that parses the string to reconstruct the binary tree.
   - In the deserialization function:
     - Split the serialized string into a list of tokens.
     - Initialize an index to keep track of the current token.
     - Recursively deserialize the tree:
       - If the current token is "null", return None.
       - Otherwise, parse the current token as the node's value and create a new TreeNode.
       - Recursively call the deserialization function for the left and right subtrees.
3. Initialize the index to 0 and call the deserialization function to reconstruct the binary tree.
4. The result is the root of the deserialized binary tree.

Time Complexity:
- Serialization: O(n) - We visit each node exactly once during serialization.
- Deserialization: O(n) - We parse each token exactly once during deserialization.

Space Complexity:
- Serialization: O(n) - The serialized string can be of length 'n' in the worst case.
- Deserialization: O(n) - We use the serialized tokens to reconstruct the tree, which can have a
length of 'n' in the worst case.
*/

use std::rc::Rc;
use std::cell::RefCell;

struct Codec;

/**
 * `&self` means the method takes an immutable reference.
 * If you need a mutable reference, change it to `&mut self` instead.
 */
impl Codec {
    fn new() -> Self {
        Codec
    }

    fn serialize(&self, root: Option<Rc<RefCell<TreeNode>>>) -> String {
        fn serialize_helper(node: Option<Rc<RefCell<TreeNode>>>, serialized: &mut String) {
            match node {
                None => serialized.push_str("null,"),
                Some(inner) => {
                    let val = inner.borrow().val.to_string();
                    serialized.push_str(&val);
                    serialized.push(',');
                    serialize_helper(inner.borrow().left.clone(), serialized);
                    serialize_helper(inner.borrow().right.clone(), serialized);
```

```
                }
            }
        }

        let mut serialized = String::new();
        serialize_helper(root, &mut serialized);
        serialized
    }

    fn deserialize(&self, data: String) -> Option<Rc<RefCell<TreeNode>>> {
        let tokens: Vec<&str> = data.split(',').collect();
        let mut index = 0;

                        fn deserialize_helper(tokens: &Vec<&str>, index: &mut usize) ->
Option<Rc<RefCell<TreeNode>>> {
            if *index >= tokens.len() {
                return None;
            }

            let val = tokens[*index];
            *index += 1;

            if val == "null" {
                return None;
            }

            let val = val.parse::<i32>().unwrap();
            let node = Rc::new(RefCell::new(TreeNode::new(val)));

            node.borrow_mut().left = deserialize_helper(tokens, index);
            node.borrow_mut().right = deserialize_helper(tokens, index);

            Some(node)
        }

        deserialize_helper(&tokens, &mut index)
    }
}

/**
 * Your Codec object will be instantiated and called as such:
 * let obj = Codec::new();
 * let data: String = obj.serialize(strs);
 * let ans: Option<Rc<RefCell<TreeNode>>> = obj.deserialize(data);
**/
```

60. 08_Tries/01_Implement_Trie_Prefix_Tree/0208-implement-trie-prefix-tree.rs

```
/*
Problem: LeetCode 208 - Implement Trie (Prefix Tree)

Key Idea:
The key idea is to use a Trie data structure to efficiently store and search for words with common prefixes.

Approach:
1. Create a struct `TrieNode` to represent a node in the Trie. Each node contains:
   - A boolean flag `is_end` to indicate whether the current node marks the end of a word.
   - An array of 26 elements (one for each lowercase letter) to store child nodes.
2. Create a struct `Trie` that represents the Trie data structure. It contains a single field `root`, which is the root node of the Trie.
3. Implement the following methods for the `Trie`:
   - `new()`: Initialize an empty Trie with a root node.
   - `insert(word: String)`: Insert a word into the Trie by iterating through its characters and creating nodes as necessary.
    - `search(word: String) -> bool`: Search for a word in the Trie by traversing nodes and checking the `is_end` flag.
   - `starts_with(prefix: String) -> bool`: Check if any word in the Trie starts with the given prefix.

Time Complexity:
- Insertion, search, and starts_with operations all have a time complexity of O(L), where L is the length of the word or prefix being processed.

Space Complexity:
- The space complexity is O(M * L), where M is the number of words inserted, and L is the average length of words. Each character in a word occupies space in the Trie.
*/

struct TrieNode {
    is_end: bool,
    children: [Option<Box<TrieNode>>; 26],
}

impl TrieNode {
    fn new() -> Self {
        TrieNode {
            is_end: false,
            children: Default::default(),
        }
    }
}

pub struct Trie {
    root: TrieNode,
}


/**
 * `&self` means the method takes an immutable reference.
 * If you need a mutable reference, change it to `&mut self` instead.
 */
impl Trie {
    pub fn new() -> Self {
        Trie { root: TrieNode::new() }
    }

    pub fn insert(&mut self, word: String) {
```

```
        let mut node = &mut self.root;
        for c in word.chars() {
            let index = (c as u8 - b'a') as usize;
            node = node.children[index].get_or_insert_with(|| Box::new(TrieNode::new()));
        }
        node.is_end = true;
    }

    pub fn search(&self, word: String) -> bool {
        let mut node = &self.root;
        for c in word.chars() {
            let index = (c as u8 - b'a') as usize;
            if let Some(next) = &node.children[index] {
                node = next;
            } else {
                return false;
            }
        }
        node.is_end
    }

    pub fn starts_with(&self, prefix: String) -> bool {
        let mut node = &self.root;
        for c in prefix.chars() {
            let index = (c as u8 - b'a') as usize;
            if let Some(next) = &node.children[index] {
                node = next;
            } else {
                return false;
            }
        }
        true
    }
}

/**
 * Your Trie object will be instantiated and called as such:
 * let obj = Trie::new();
 * obj.insert(word);
 * let ret_2: bool = obj.search(word);
 * let ret_3: bool = obj.starts_with(prefix);
**/

/**
struct Trie {
    children: HashMap<char, Trie>,
    is_end: bool,
}

impl Trie {
    fn new() -> Self {
        Trie {
            children: HashMap::new(),
            is_end: false,
        }
    }

    fn insert(&mut self, word: String) {
        let mut node = self;
        for c in word.chars() {
            node = node.children.entry(c).or_insert(Trie::new());
        }
        node.is_end = true;
    }
```

```
    fn search(&self, word: String) -> bool {
        let mut node = self;
        for c in word.chars() {
            if let Some(next_node) = node.children.get(&c) {
                node = next_node;
            } else {
                return false;
            }
        }
        node.is_end
    }

    fn starts_with(&self, prefix: String) -> bool {
        let mut node = self;
        for c in prefix.chars() {
            if let Some(next_node) = node.children.get(&c) {
                node = next_node;
            } else {
                return false;
            }
        }
        true
    }
}
**/
```

61. 08_Tries/02_Design_Add_and_Search_Words_Data_Structure/0211-design-add-a

```
/*
Problem: LeetCode 211 - Add and Search Word - Data structure

Key Idea:
The key idea is to use a Trie data structure to efficiently store and search for words with wildcard characters.

Approach:
1. Create a struct `TrieNode` to represent a node in the Trie. Each node contains:
   - A boolean flag `is_end` to indicate whether the current node marks the end of a word.
   - An array of 26 elements (one for each lowercase letter) to store child nodes.
2. Create a struct `WordDictionary` that represents the word dictionary. It contains a single field `root`, which is the root node of the Trie.
3. Implement the following methods for the `WordDictionary`:
   - `new()`: Initialize an empty dictionary with a root node.
   - `add_word(word: String)`: Insert a word into the Trie by iterating through its characters and creating nodes as necessary.
   - `search(word: String) -> bool`: Search for a word in the Trie by traversing nodes. For each character:
     - If it is a regular letter, move to the corresponding child node.
     - If it is a dot ('.'), recursively search all child nodes for the remaining word.

Time Complexity:
- Adding a word has a time complexity of O(L), where L is the length of the word.
- Searching for a word has a time complexity of O(26^M), where M is the number of dots ('.') in the word. In the worst case, all child nodes need to be explored.

Space Complexity:
- The space complexity is O(M * L), where M is the number of words added, and L is the average length of words. Each character in a word occupies space in the Trie.
*/

struct TrieNode {
    is_end: bool,
    children: [Option<Box<TrieNode>>; 26],
}

impl TrieNode {
    fn new() -> Self {
        TrieNode {
            is_end: false,
            children: Default::default(),
        }
    }
}

pub struct WordDictionary {
    root: TrieNode,
}

impl WordDictionary {
    pub fn new() -> Self {
        WordDictionary {
            root: TrieNode::new(),
        }
    }

    pub fn add_word(&mut self, word: String) {
        let mut node = &mut self.root;
        for c in word.chars() {
```

```
            let index = (c as u8 - b'a') as usize;
            node = node.children[index].get_or_insert_with(|| Box::new(TrieNode::new()));
        }
        node.is_end = true;
    }

    pub fn search(&self, word: String) -> bool {
        self.search_recursive(&self.root, word.chars().collect())
    }

    fn search_recursive(&self, node: &TrieNode, word: Vec<char>) -> bool {
        if word.is_empty() {
            return node.is_end;
        }

        let c = word[0];
        if c == '.' {
            for child in &node.children {
                if let Some(child_node) = child {
                    if self.search_recursive(child_node, word[1..].to_vec()) {
                        return true;
                    }
                }
            }
        } else {
            let index = (c as u8 - b'a') as usize;
            if let Some(child_node) = &node.children[index] {
                return self.search_recursive(child_node, word[1..].to_vec());
            }
        }

        false
    }
}
```

62. 08_Tries/03_Word_Search_II/0212-word-search-ii.rs

```
/*
Problem: LeetCode 212 - Word Search II

Key Idea:
The key idea is to use a Trie data structure to efficiently store the given words and then perform a depth-first search (DFS) on the board to find words that match the Trie structure.

Approach:
1. Create a Trie structure with methods for inserting words and tracking word solutions. Each Trie node has 26 children (one for each lowercase letter) and a flag to mark the end of a word.
2. Insert all the words from the given list into the Trie.
3. Initialize an empty result vector `ans` to store the found words.
4. Iterate through each cell of the board and call the DFS function:
    - In the DFS function, mark the current cell as visited by replacing its character with a space.
   - Check if there's a Trie node corresponding to the character at the current cell.
   - If a word solution is found in the Trie, add it to the result vector `ans`.
   - Recursively explore the neighboring cells (up, down, left, right) that have valid characters and Trie nodes associated with them.
   - Restore the original character at the current cell.
5. After processing all cells on the board, return the `ans` vector containing the found words.

Time Complexity:
- Building the Trie from the given words takes O(W*L), where W is the total number of characters in all words, and L is the average word length.
- The DFS operation on the board has a worst-case time complexity of O(N*M*4^L), where N and M are the dimensions of the board, and L is the maximum word length. This is because, in the worst case, the algorithm explores four neighboring cells at each step for each cell on the board.

Space Complexity:
- The space complexity of the Trie structure is O(W*L), where W is the total number of characters in all words, and L is the average word length.
- The space complexity of the DFS stack is O(L), where L is the maximum word length.
*/

struct Trie {
    children: Vec<Option<Trie>>,
    is_solution: Option<String>,
}

impl Trie {
    fn new() -> Self {
        Trie {
            children: (0..26).map(|_| None).collect(),
            is_solution: None,
        }
    }

    fn insert(&mut self, word: String) {
        let mut curr = self;
        for byte in word.bytes() {
            let index = (byte - b'a') as usize;
            curr = curr.children[index].get_or_insert_with(Trie::new);
        }
        curr.is_solution = Some(word);
    }
}

impl Solution {
    pub fn find_words(mut board: Vec<Vec<char>>, words: Vec<String>) -> Vec<String> {
```

```
        let mut trie = Trie::new();
        for word in words {
            trie.insert(word);
        }
        let (n, m) = (board.len(), board[0].len());
        let mut ans = Vec::new();
        for i in 0..n {
            for j in 0..m {
                Self::dfs(&mut board, &mut trie, &mut ans, i, j);
            }
        }
        ans
    }

    fn dfs(board: &mut Vec<Vec<char>>, trie: &mut Trie, ans: &mut Vec<String>, i: usize, j: usize)
{
        let c = board[i][j];
        board[i][j] = ' ';
        if let Some(next) = trie.children[(c as u8 - b'a') as usize].as_mut() {
            if let Some(word) = next.is_solution.take() {
                ans.push(word);
            }
            for &(x, y) in &[
                (i + 1, j),
                (i.wrapping_sub(1), j),
                (i, j + 1),
                (i, j.wrapping_sub(1)),
            ] {
                if x < board.len()
                    && y < board[0].len()
                    && board[x][y] != ' '
                    && next.children[(board[x][y] as u8 - b'a') as usize].is_some()
                {
                    Self::dfs(board, next, ans, x, y);
                }
            }
        }
        board[i][j] = c;
    }
}

/*
// Another Solution

use std::collections::HashMap;

type TrieNode = HashMap<char, Trie>;

#[derive(Default)]
struct Trie {
    children: TrieNode,
    word: Option<String>,
}

impl Trie {
    fn new() -> Self {
        Trie {
            children: TrieNode::new(),
            word: None,
        }
    }
}

impl Solution {
```

```
pub fn find_words(mut board: Vec<Vec<char>>, words: Vec<String>) -> Vec<String> {
    let letter_freq = board.iter().flatten().fold(HashMap::new(), |mut acc, &ch| {
        *acc.entry(ch).or_insert(0) += 1;
        acc
    });

    let mut root = Trie::default();
    for word in words {
        let should_reverse = match (word.chars().next(), word.chars().last()) {
            (Some(first), Some(last)) => {
                letter_freq.get(&first).unwrap_or(&0) > letter_freq.get(&last).unwrap_or(&0)
            }
            _ => false,
        };

        let processed_word: String = if should_reverse {
            word.chars().rev().collect()
        } else {
            word.clone()
        };

        let mut node = &mut root;
        for letter in processed_word.chars() {
            node = node.children.entry(letter).or_insert_with(Trie::default);
        }
        node.word = Some(word);
    }

    let mut matched_words = Vec::new();
    let (row_num, col_num) = (board.len(), board[0].len());

    fn backtracking(
        row: usize,
        col: usize,
        parent: &mut Trie,
        board: &mut Vec<Vec<char>>,
        matched_words: &mut Vec<String>,
        row_num: usize,
        col_num: usize,
    ) {
        let letter = board[row][col];
        let curr_node = match parent.children.get_mut(&letter) {
            Some(node) => node,
            None => return,
        };

        if let Some(word_match) = curr_node.word.take() {
            matched_words.push(word_match);
        }

        board[row][col] = '#';

        const ROW_ADJUST: [isize; 4] = [-1, 0, 1, 0];
        const COL_ADJUST: [isize; 4] = [0, 1, 0, -1];

        for i in 0..4 {
            let new_row = row as isize + ROW_ADJUST[i];
            let new_col = col as isize + COL_ADJUST[i];

            if new_row >= 0
                && new_row < row_num as isize
                && new_col >= 0
                && new_col < col_num as isize
            {
```

```
                    backtracking(
                        new_row as usize,
                        new_col as usize,
                        curr_node,
                        board,
                        matched_words,
                        row_num,
                        col_num,
                    );
                }
            }

            board[row][col] = letter;

            if curr_node.children.is_empty() && curr_node.word.is_none() {
                parent.children.remove(&letter);
            }
        }

        for (row, col) in (0..row_num).flat_map(|r| (0..col_num).map(move |c| (r, c))) {
            if root.children.contains_key(&board[row][col]) {
                backtracking(
                    row,
                    col,
                    &mut root,
                    &mut board,
                    &mut matched_words,
                    row_num,
                    col_num,
                );
            }
        }

        matched_words
    }
}
*/
```

63. 09_Heap_Priority_Queues/01_Kth_Largest_Element_in_a_Stream/0703-kth-larges

```
/*
Problem: LeetCode 703 - Kth Largest Element in a Stream

Key Idea:
The key idea is to use a min-heap (priority queue) to efficiently maintain the k largest elements
from the stream.

Approach:
1. Initialize a min-heap with a capacity of k elements.
2. As new elements are added to the stream:
   - If the min-heap is not at capacity (i.e., less than k elements), simply push the new element
into the heap.
   - If the min-heap is at capacity, compare the new element with the smallest element in the heap
(the root).
     - If the new element is larger, remove the smallest element (pop from the heap) and push the
new element.
     - Otherwise, ignore the new element.
3. The top element of the min-heap (the root) will be the kth largest element.

Time Complexity:
- Add Operation: O(log k), where k is the capacity of the min-heap. Inserting into a heap takes
O(log k) time.
- Get Kth Largest Operation: O(1), as the top element of the min-heap is the kth largest element.

Space Complexity:
O(k), where k is the capacity of the min-heap.
*/

use std::cmp::Reverse;
use std::collections::BinaryHeap;

struct KthLargest {
    k: usize,
    max_heap: BinaryHeap<Reverse<i32>>,
}

impl KthLargest {
    fn new(k: i32, nums: Vec<i32>) -> Self {
        let mut obj = KthLargest {
            k: k as usize,
            max_heap: BinaryHeap::new(),
        };
        for num in nums {
            obj.max_heap.push(Reverse(num));
        }
        obj
    }

    fn add(&mut self, val: i32) -> i32 {
        self.max_heap.push(Reverse(val));
        while self.max_heap.len() > self.k {
            self.max_heap.pop();
        }

        if let Some(&Reverse(kth_largest)) = self.max_heap.peek() {
            kth_largest
        } else {
            0
        }
    }
```

```
}
```

64. 09_Heap_Priority_Queues/02_Last_Stone_Weight/1046-last-stone-weight.rs

```
/*
Problem: LeetCode 1046 - Last Stone Weight

Key Idea:
The key idea is to use a max-heap (priority queue) to efficiently simulate the process of smashing stones until only one stone remains or no stones are left.

Approach:
1. Create a max-heap (priority queue) to store the weights of the stones.
2. Insert all stone weights into the max-heap.
3. While there are at least two stones in the max-heap:
   - Pop the two largest stones from the max-heap.
   - Calculate their difference (smash them together).
   - If the difference is not zero, insert it back into the max-heap.
4. If there is exactly one stone left in the max-heap, return its weight; otherwise, return 0.

Time Complexity:
- Heap Insertion and Extraction: O(log n) per operation, where n is the number of stones.
- In the worst case, we perform n-1 iterations of the while loop, resulting in O(n log n) time complexity for the entire process.

Space Complexity:
O(n), where n is the number of stones, as we use a max-heap to store the stone weights.
*/

use std::collections::BinaryHeap;

impl Solution {
    pub fn last_stone_weight(stones: Vec<i32>) -> i32 {
        let mut max_heap = BinaryHeap::new();

        // Insert all stone weights into the max-heap
        for stone in stones {
            max_heap.push(stone);
        }

        // Simulate stone smashing until only one stone remains or none
        while max_heap.len() >= 2 {
            let stone1 = max_heap.pop().unwrap();
            let stone2 = max_heap.pop().unwrap();
            let diff = stone1 - stone2;

            if diff > 0 {
                max_heap.push(diff);
            }
        }

        // Return the remaining stone weight or 0
        max_heap.pop().unwrap_or(0)
    }
}
```

65. 09_Heap_Priority_Queues/03_K_Closest_Points_to_Origin/0973-k-closest-points-

```
/*
Problem: LeetCode 973 - K Closest Points to Origin

Key Idea:
The key idea is to use a min-heap (priority queue) to efficiently find the k closest points to the
origin (0, 0) based on their Euclidean distances.

Approach:
1. Create a min-heap (priority queue) to store points based on their squared Euclidean distances.
2. Iterate through the points:
   - Calculate the squared Euclidean distance of each point to the origin: `x^2 + y^2`.
   - Push the point and its squared distance as a tuple into the min-heap.
3. If the min-heap size exceeds `k`, pop the point with the largest squared distance. This step
ensures that the min-heap always contains the `k` closest points.
4. After processing all points, the min-heap will contain the `k` closest points to the origin.
5. Extract the points from the min-heap in the order they appear. Since it's a min-heap, they will
come out in ascending order of distance.
6. Return the `k` closest points.

Time Complexity:
- Heap Insertion and Extraction: O(log k) per operation, where k is the value of 'K'.
- We perform this operation for each of the 'N' points.
- Overall time complexity is O(N log K).

Space Complexity:
O(K), as the max-heap can contain at most K points.
*/

use std::cmp::Ordering;
use std::collections::BinaryHeap;

impl Solution {
    pub fn k_closest(points: Vec<Vec<i32>>, k: i32) -> Vec<Vec<i32>> {
        let k = k as usize;
        let mut min_heap = BinaryHeap::with_capacity(k + 1);

        for point in &points {
            let p = Point::from_vec(point);
            min_heap.push(p);

            if min_heap.len() > k {
                min_heap.pop();
            }
        }

        min_heap
            .into_sorted_vec()
            .into_iter()
            .map(Point::into_vec)
            .collect()
    }
}

#[derive(Eq, PartialEq)]
struct Point {
    x: i32,
    y: i32,
}

impl Point {
```

```
    fn distance_squared(&self) -> i32 {
        self.x * self.x + self.y * self.y
    }

    fn from_vec(v: &Vec<i32>) -> Self {
        Point { x: v[0], y: v[1] }
    }

    fn into_vec(self) -> Vec<i32> {
        vec![self.x, self.y]
    }
}

impl Ord for Point {
    fn cmp(&self, other: &Self) -> Ordering {
        self.distance_squared().cmp(&other.distance_squared())
    }
}

impl PartialOrd for Point {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        Some(self.cmp(other))
    }
}

/*
// Quickselect method

impl Solution {
    pub fn k_closest(mut points: Vec<Vec<i32>>, k: i32) -> Vec<Vec<i32>> {
        let k = k as usize;
        let (mut left, mut right) = (0, points.len() - 1);

        while left <= right {
            let pivot_index = partition(&mut points, left, right);

            if pivot_index < k {
                left = pivot_index + 1;
            } else if pivot_index > k {
                right = pivot_index - 1;
            } else {
                break;
            }
        }

        points[..k].to_vec()
    }
}

const X: usize = 0;
const Y: usize = 1;

fn squared_distance(point: &Vec<i32>) -> i32 {
    point[X].pow(2) + point[Y].pow(2)
}

fn partition(points: &mut Vec<Vec<i32>>, mut left: usize, mut right: usize) -> usize {
    // Use Hoare's partition scheme.
    let pivot_index = left;
    while left < right {
                        while   left   <   right   &&   squared_distance(&points[pivot_index])   <=
squared_distance(&points[right]) {
            right -= 1;
        }
```

```
                            while left < right && squared_distance(&points[left]) <=
squared_distance(&points[pivot_index]) {
                left += 1;
            }
            points.swap(left, right);
        }
        points.swap(left, pivot_index);
        left
}
*/

/*
// Sorting method

impl Solution {
    fn squared_distance_to_zero(point: &Vec<i32>) -> i32 {
        point[0].pow(2) + point[1].pow(2)
    }

    pub fn k_closest(mut points: Vec<Vec<i32>>, k: i32) -> Vec<Vec<i32>> {
        points.sort_by_key(|point| Solution::squared_distance_to_zero(point));
        points.into_iter().take(k as usize).collect()
    }
}
*/
```

66. 09_Heap_Priority_Queues/04_Kth_Largest_Element_in_an_Array/0215-kth-larges

```
/*
Problem: LeetCode 215 - Kth Largest Element in an Array

Key Idea:
The key idea is to use a min-heap (priority queue) to efficiently find the kth largest element in
the array.

Approach:
1. Create a min-heap (priority queue) with a capacity of k elements.
2. Iterate through the array:
   - Insert each element into the min-heap.
   - If the min-heap size exceeds k, remove the smallest element (pop from the heap).
3. The top element of the min-heap (the root) will be the kth largest element.

Time Complexity:
- Heap Insertion and Extraction: O(log k) per operation, where k is the value of 'k'.
- We perform this operation for each of the 'n' elements in the array.
- Overall time complexity is O(n log k).

Space Complexity:
O(k), as the min-heap can contain at most k elements.
*/

use std::cmp::Reverse;
use std::collections::BinaryHeap;

impl Solution {
    pub fn find_kth_largest(nums: Vec<i32>, k: i32) -> i32 {
        // Create a min-heap with a capacity of k
        let mut min_heap: BinaryHeap<Reverse<i32>> = nums
            .iter()
            .take(k as usize)
            .map(|&num| Reverse(num))
            .collect();

        // Iterate through the remaining elements
        for &num in nums.iter().skip(k as usize) {
             // If the current number is larger than the smallest number in the min-heap, replace
it
            if num > min_heap.peek().unwrap().0 {
                min_heap.pop();
                min_heap.push(Reverse(num));
            }
        }

        // The top of the min-heap contains the kth largest element
        min_heap.peek().unwrap().0
    }
}
```

67. 09_Heap_Priority_Queues/05_Task_Scheduler/0621-task-scheduler.rs

```
/*
Problem: LeetCode 621 - Task Scheduler

Key Idea:
The key idea is to arrange tasks by their frequency and then insert the most frequent tasks into the schedule, leaving gaps equal to the cooldown period. Fill these gaps with less frequent tasks if available.

Approach:
1. Calculate the frequency of each task and store it in a character frequency array.
2. Sort the frequency array in descending order to prioritize scheduling tasks with higher frequencies.
3. Initialize a variable 'idle_slots' to represent the number of idle CPU cooling intervals.
4. While there are tasks left to schedule:
   - Start scheduling tasks from the task with the highest frequency.
   - Place the task in an available slot in the CPU cooling intervals.
   - Decrease its frequency by 1.
   - If there are no more tasks with a higher frequency, use an idle slot.
   - If all slots are used, increase 'idle_slots' by the number of available slots.
   - Repeat this process until all tasks are scheduled.
5. The minimum time required to execute all tasks will be the sum of the tasks and idle slots.

Time Complexity:
O(n), where n is the number of tasks. We iterate through the tasks once to calculate frequencies and once to schedule them.

Space Complexity:
O(26) = O(1), as we use an array of size 26 to store the task frequencies.
*/

impl Solution {
    pub fn least_interval(tasks: Vec<char>, n: i32) -> i32 {
        if n == 0 {
            return tasks.len() as i32;
        }

        let mut task_count = vec![0; 26];
        for &task in &tasks {
            task_count[(task as u8 - b'A') as usize] += 1;
        }

        task_count.sort_unstable();
        task_count.reverse();

        let max_task = task_count[0] as i32;
        let mut idle_slots = (max_task - 1) * n;

        for &count in task_count.iter().skip(1) {
            idle_slots -= std::cmp::min(count as i32, max_task - 1);
        }

        idle_slots = idle_slots.max(0);

        tasks.len() as i32 + idle_slots
    }
}
```

68. 09_Heap_Priority_Queues/06_Design_Twitter/0355-design-twitter.rs

```
/*
Problem: LeetCode 355 - Design Twitter

Key Idea:
The key idea is to use a combination of data structures: a hash map to store users and their tweets, a priority queue to retrieve the most recent tweets, and a timestamp to track tweet posting times.

Approach:
1. Create two data structures: one for user-to-tweets mapping and another for user-to-following mapping.
2. Maintain a global timestamp to order tweets chronologically.
3. For each tweet, store its content and timestamp along with the user who posted it.
4. Implement a 'postTweet' function to add a tweet to a user's tweet list.
5. Implement a 'getNewsFeed' function to retrieve the most recent tweets from the user and the users they follow, sort them by timestamp, and return the top 'n' tweets.
6. Implement a 'follow' function to allow one user to follow another user.
7. Implement an 'unfollow' function to allow one user to unfollow another user.
8. Use a min-heap (priority queue) to efficiently retrieve the top 'n' tweets in 'getNewsFeed'.

Time Complexity:
- Posting a tweet: O(1)
- Retrieving the news feed: O(n * log(k)), where n is the number of tweets in the user's feed, and k is the number of users the user follows.
- Following/unfollowing a user: O(1)

Space Complexity:
O(m + n), where m is the total number of tweets, and n is the total number of users.
*/

use std::cmp::Reverse;
use std::collections::HashSet;
use std::collections::{BinaryHeap, HashMap};

struct Twitter {
    user_tweets: HashMap<i32, Vec<(i32, i32)>>, // User ID -> List of (Tweet ID, Timestamp)
    user_follows: HashMap<i32, Vec<i32>>,       // User ID -> List of Followed User IDs
    timestamp: i32,
}

impl Twitter {
    fn new() -> Self {
        Twitter {
            user_tweets: HashMap::new(),
            user_follows: HashMap::new(),
            timestamp: 0,
        }
    }

    fn post_tweet(&mut self, user_id: i32, tweet_id: i32) {
        self.timestamp += 1;
        self.user_tweets
            .entry(user_id)
            .or_insert(vec![])
            .push((tweet_id, self.timestamp));
    }

    fn get_news_feed(&self, user_id: i32) -> Vec<i32> {
        let mut tweet_set = HashSet::new();
        let mut min_heap = BinaryHeap::new();
```

```
        // Include user's own tweets
        if let Some(tweets) = self.user_tweets.get(&user_id) {
            for &(tweet_id, timestamp) in tweets.iter() {
                min_heap.push(Reverse((timestamp, tweet_id)));
                if min_heap.len() > 10 {
                    min_heap.pop();
                }
            }
        }

        // Include tweets from followed users
        if let Some(followed) = self.user_follows.get(&user_id) {
            for &followed_user_id in followed.iter() {
                if let Some(tweets) = self.user_tweets.get(&followed_user_id) {
                    for &(tweet_id, timestamp) in tweets.iter() {
                        min_heap.push(Reverse((timestamp, tweet_id)));
                        if min_heap.len() > 10 {
                            min_heap.pop();
                        }
                    }
                }
            }
        }

        // Extract the top tweets from the min-heap
        let mut result = vec![];
        while let Some(Reverse((_, tweet_id))) = min_heap.pop() {
            if tweet_set.insert(tweet_id) {
                result.push(tweet_id);
            }
        }

        result.reverse(); // Reverse to get tweets in chronological order
        result
    }

    fn follow(&mut self, follower_id: i32, followee_id: i32) {
        if follower_id != followee_id {
            self.user_follows
                .entry(follower_id)
                .or_insert(vec![])
                .push(followee_id);
        }
    }

    fn unfollow(&mut self, follower_id: i32, followee_id: i32) {
        if let Some(followees) = self.user_follows.get_mut(&follower_id) {
            if let Some(pos) = followees.iter().position(|&id| id == followee_id) {
                followees.remove(pos);
            }
        }
    }
}
```

69. 09_Heap_Priority_Queues/07_Find_Median_from_Data_Stream/0295-find-median-

```
/*
Problem: LeetCode 295 - Find Median from Data Stream

Key Idea:
The key idea is to use two heaps (priority queues) to find the median of a data stream.

Approach:
1. Maintain two heaps: a max-heap (left_heap) to store the lower half of the elements and a min-heap (right_heap) to store the upper half.
2. Ensure that the size of the left_heap is either equal to or one greater than the size of the right_heap.
3. When inserting an element into the data stream:
    - If the element is less than or equal to the median (top of left_heap), insert it into the left_heap.
   - Otherwise, insert it into the right_heap.
   - Balance the heaps to ensure the size property mentioned in step 2.
4. To find the median:
    - If the size of the left_heap is greater than the size of the right_heap, return the top element of the left_heap as the median.
   - If the sizes are equal, return the average of the top elements of both heaps as the median.

Time Complexity:
- Insertion: O(log n), where n is the number of elements in the data stream.
- Finding the median: O(1).

Space Complexity:
O(n), where n is the number of elements in the data stream (space is used to store the elements in heaps).
*/

use std::cmp::Reverse;
use std::collections::BinaryHeap;

struct MedianFinder {
    left_heap: BinaryHeap<i32>,           // Max-heap for the lower half
    right_heap: BinaryHeap<Reverse<i32>>, // Min-heap for the upper half
}

impl MedianFinder {
    fn new() -> Self {
        MedianFinder {
            left_heap: BinaryHeap::new(),
            right_heap: BinaryHeap::new(),
        }
    }

    fn add_num(&mut self, num: i32) {
        self.left_heap.push(num);
        self.right_heap.push(Reverse(self.left_heap.pop().unwrap()));

        if self.left_heap.len() < self.right_heap.len() {
            self.left_heap.push(self.right_heap.pop().unwrap().0);
        }
    }

    fn find_median(&self) -> f64 {
        let left_len = self.left_heap.len();
        let right_len = self.right_heap.len();

        if left_len > right_len {
```

```
            self.left_heap.peek().map(|&x| x as f64).unwrap_or(0.0)
        } else if left_len < right_len {
            self.right_heap.peek().map(|&x| x.0 as f64).unwrap_or(0.0)
        } else {
            let left_top = self.left_heap.peek().map(|&x| x).unwrap_or(0);
            let right_top = self.right_heap.peek().map(|&x| x.0).unwrap_or(0);
            (left_top + right_top) as f64 / 2.0
        }
    }
}
```

70. 10_Backtracking/01_Subsets/0078-subsets.rs

```
/*
Problem: LeetCode 78 - Subsets

Key Idea:
To generate all possible subsets of a given set of numbers, we can use a recursive approach. Starting with an empty subset, we gradually build subsets by including or excluding each number in the set.

Approach:
1. Initialize an empty vector to store subsets.
2. Create a helper function, backtrack, that takes three parameters: the current index, the current subset being built, and the input vector of numbers.
3. In the helper function:
   a. If the current index is equal to the length of the input vector, push the current subset into the result vector.
   b. Recursively call the helper function for the next index while including the current number in the subset.
   c. Recursively call the helper function for the next index while excluding the current number from the subset.
4. Start the recursion with index 0 and an empty subset.
5. Return the result vector containing all generated subsets.

Time Complexity:
O(2^n), where n is the number of elements in the input vector. This is because there are 2^n possible subsets for n elements.

Space Complexity:
O(2^n) in the worst case, as there can be up to 2^n subsets.
*/

impl Solution {
    pub fn subsets(nums: Vec<i32>) -> Vec<Vec<i32>> {
        let mut result: Vec<Vec<i32>> = Vec::new();

        let mut current_subset: Vec<i32> = Vec::new();
        Self::backtrack(0, &mut current_subset, &nums, &mut result);

        result
    }

    // Helper function to recursively generate subsets
    fn backtrack(
        index: usize,
        current_subset: &mut Vec<i32>,
        nums: &Vec<i32>,
        result: &mut Vec<Vec<i32>>,
    ) {
        if index == nums.len() {
            // Base case: All elements processed, add current subset to the result
            result.push(current_subset.clone());
            return;
        }

        // Include the current element in the subset and recurse
        current_subset.push(nums[index]);
        Self::backtrack(index + 1, current_subset, nums, result);

        // Exclude the current element from the subset and recurse
        current_subset.pop();
        Self::backtrack(index + 1, current_subset, nums, result);
```

```
    }
}
```

71. 10_Backtracking/02_Combination_Sum/0039-combination-sum.rs

```
/*
Problem: LeetCode 39 - Combination Sum

Key Idea:
To find all unique combinations of numbers in a given array that sum up to a target value, we can use a recursive backtracking approach. Starting with an empty combination, we gradually build combinations by considering each number in the array and exploring all possible combinations.

Approach:
1. Initialize an empty vector to store the result combinations.
2. Create a helper function, backtrack, that takes four parameters: the current index, the current combination being built, the remaining target value, and the input vector of numbers.
3. In the helper function:
    a. If the remaining target value becomes zero, push the current combination into the result vector as a valid combination.
    b. If the remaining target value becomes negative or the current index exceeds the array bounds, return without making any changes.
   c. Recursively call the helper function with the current index to include the current number in the combination and subtract the number from the remaining target value.
   d. Recursively call the helper function with the next index to exclude the current number from the combination and keep the target value unchanged.
4. Start the recursion with index 0, an empty combination, and the target value equal to the input target.
5. Return the result vector containing all valid combinations.

Time Complexity:
The time complexity of this solution is exponential, as there can be many valid combinations. In the worst case, it can be O(N^target), where N is the number of elements in the input vector.

Space Complexity:
The space complexity depends on the number of valid combinations. In the worst case, it can be O(target) for the recursion stack.
*/

impl Solution {
    pub fn combination_sum(candidates: Vec<i32>, target: i32) -> Vec<Vec<i32>> {
        let mut result: Vec<Vec<i32>> = Vec::new();

        let mut current_combination: Vec<i32> = Vec::new();
        Self::backtrack(
            0,
            &mut current_combination,
            target,
            &candidates,
            &mut result,
        );

        result
    }

    // Helper function to recursively generate combinations
    fn backtrack(
        index: usize,
        current_combination: &mut Vec<i32>,
        remaining_target: i32,
        candidates: &Vec<i32>,
        result: &mut Vec<Vec<i32>>,
    ) {
        if remaining_target == 0 {
            // Base case: Target sum reached, add current combination to result
```

```
            result.push(current_combination.clone());
            return;
        }

        if remaining_target < 0 || index >= candidates.len() {
            // Base case: Invalid combination, return without changes
            return;
        }

        // Include the current number in the combination and recurse
        current_combination.push(candidates[index]);
        Self::backtrack(
            index,
            current_combination,
            remaining_target - candidates[index],
            candidates,
            result,
        );

        // Exclude the current number from the combination and recurse with the next number
        current_combination.pop();
        Self::backtrack(
            index + 1,
            current_combination,
            remaining_target,
            candidates,
            result,
        );
    }
}
```

72. 10_Backtracking/03_Permutations/0046-permutations.rs

```
/*
Problem: LeetCode 46 - Permutations

Key Idea:
To find all permutations of a given array of distinct integers, we can use a backtracking approach. Starting with an empty permutation, we gradually build permutations by considering each unused element in the input array and exploring all possible choices.

Approach:
1. Initialize an empty vector to store the result permutations.
2. Create a helper function, backtrack, that takes three parameters: the current permutation being built, a boolean array to track used elements, and the result vector.
3. In the helper function:
    a. If the current permutation size equals the input array size, push the current permutation into the result vector as a valid permutation.
   b. Iterate through the input array:
       i. If the current element is not used, mark it as used, add it to the current permutation, and recursively call the helper function.
         ii. After the recursive call, backtrack by marking the current element as unused and removing it from the current permutation.
4. Start the recursion with an empty permutation, a boolean array of all unused elements, and the result vector.
5. Return the result vector containing all valid permutations.

Time Complexity:
The time complexity of this solution is O(N!), where N is the number of elements in the input array. This is because there can be N! possible permutations.

Space Complexity:
The space complexity is O(N) for the recursion stack and the boolean array to track used elements.
*/

impl Solution {
    pub fn permute(nums: Vec<i32>) -> Vec<Vec<i32>> {
        let mut result: Vec<Vec<i32>> = Vec::new();

        let mut current_permutation: Vec<i32> = Vec::new();
        let mut used: Vec<bool> = vec![false; nums.len()];

        Self::backtrack(&mut current_permutation, &mut used, &mut result, &nums);

        result
    }

    // Helper function to recursively generate permutations
    fn backtrack(
        current_permutation: &mut Vec<i32>,
        used: &mut Vec<bool>,
        result: &mut Vec<Vec<i32>>,
        nums: &Vec<i32>,
    ) {
        if current_permutation.len() == nums.len() {
            // Base case: Permutation size equals array size, add to result
            result.push(current_permutation.clone());
            return;
        }

        for (i, &num) in nums.iter().enumerate() {
            if !used[i] {
                // Mark element as used, add to current permutation, and recurse
```

```
                used[i] = true;
                current_permutation.push(num);
                Self::backtrack(current_permutation, used, result, nums);

                // Backtrack: Mark element as unused and remove from current permutation
                used[i] = false;
                current_permutation.pop();
            }
        }
    }
}
```

73. 10_Backtracking/04_Subsets_II/0090-subsets-ii.rs

```
/*
Problem: LeetCode 90 - Subsets II

Key Idea:
To find all unique subsets of an array that may contain duplicate elements, we can use a backtracking approach with a twist. The twist is to skip duplicates to avoid generating duplicate subsets.

Approach:
1. Sort the input array to bring duplicates together.
2. Initialize an empty vector to store the result subsets.
3. Create a helper function, backtrack, that takes three parameters: the current subset being built, the starting index for considering elements, and the result vector.
4. In the helper function:
   a. Add the current subset to the result vector.
   b. Iterate through the input array starting from the given index:
        i. If the current element is not a duplicate (i.e., it's different from the previous element), add it to the current subset and recursively call the helper function with the next index.
        ii. After the recursive call, backtrack by removing the last element from the current subset.
5. Start the recursion with an empty subset, 0 as the starting index, and the result vector.
6. Return the result vector containing all unique subsets.

Time Complexity:
The time complexity of this solution is O(2^N), where N is the number of elements in the input array. This is because there can be 2^N possible subsets.

Space Complexity:
The space complexity is O(N) for the recursion stack.
*/

impl Solution {
    pub fn subsets_with_dup(nums: Vec<i32>) -> Vec<Vec<i32>> {
        let mut result: Vec<Vec<i32>> = Vec::new();

        let mut current_subset: Vec<i32> = Vec::new();
        let mut sorted_nums = nums.clone();
        sorted_nums.sort();

        Self::backtrack(&mut current_subset, 0, &mut result, &sorted_nums);

        result
    }

    // Helper function to recursively generate subsets
    fn backtrack(
        current_subset: &mut Vec<i32>,
        start_index: usize,
        result: &mut Vec<Vec<i32>>,
        nums: &Vec<i32>,
    ) {
        result.push(current_subset.clone());

        for i in start_index..nums.len() {
            if i == start_index || nums[i] != nums[i - 1] {
                current_subset.push(nums[i]);
                Self::backtrack(current_subset, i + 1, result, nums);
                current_subset.pop();
            }
```

```
            }
        }
    }
```

74. 10_Backtracking/05_Combination_Sum_II/0040-combination-sum-ii.rs

```
/*
Problem: LeetCode 40 - Combination Sum II

Key Idea:
To find all unique combinations of candidates that sum to a target value, we can use a backtracking approach. However, to avoid duplicate combinations, we need to make sure that we don't consider the same candidate multiple times in a single combination and skip duplicates.

Approach:
1. Sort the input candidates to bring duplicates together.
2. Initialize an empty vector to store the result combinations.
3. Create a helper function, backtrack, that takes four parameters: the current combination being built, the current index in the candidates, the remaining target value, and the result vector.
4. In the helper function:
   a. If the remaining target is 0, add the current combination to the result vector.
   b. Otherwise, iterate through the candidates starting from the given index:
        i. If the current candidate is greater than the remaining target, stop the iteration (candidates are sorted).
       ii. If the current index is greater than the start index and the current candidate is the same as the previous candidate, skip it to avoid duplicates.
       iii. Add the current candidate to the current combination and recursively call the helper function with the updated index and remaining target reduced by the current candidate.
        iv. After the recursive call, backtrack by removing the last candidate from the current combination.
5. Start the recursion with an empty combination, 0 as the starting index, the target value, and the result vector.
6. Return the result vector containing all unique combinations.

Time Complexity:
The time complexity of this solution is O(2^N), where N is the number of elements in the input candidates. In the worst case, all possible combinations are considered.

Space Complexity:
The space complexity is O(N) for the recursion stack.
*/

impl Solution {
    pub fn combination_sum2(candidates: Vec<i32>, target: i32) -> Vec<Vec<i32>> {
        let mut result: Vec<Vec<i32>> = Vec::new();

        let mut current_combination: Vec<i32> = Vec::new();
        let mut sorted_candidates = candidates.clone();
        sorted_candidates.sort();

        Self::backtrack(
            &mut current_combination,
            0,
            target,
            &mut result,
            &sorted_candidates,
        );

        result
    }

    // Helper function to recursively generate combinations
    fn backtrack(
        current_combination: &mut Vec<i32>,
        start_index: usize,
        remaining_target: i32,
```

```
        result: &mut Vec<Vec<i32>>,
        candidates: &Vec<i32>,
    ) {
        if remaining_target == 0 {
            result.push(current_combination.clone());
            return;
        }

        for i in start_index..candidates.len() {
            if candidates[i] > remaining_target {
                break;
            }

            if i > start_index && candidates[i] == candidates[i - 1] {
                continue; // Skip duplicates
            }

            current_combination.push(candidates[i]);
            Self::backtrack(
                current_combination,
                i + 1,
                remaining_target - candidates[i],
                result,
                candidates,
            );
            current_combination.pop();
        }
    }
}
```

75. 10_Backtracking/06_Word_Search/0079-word-search.rs

```
/*
Problem: LeetCode 79 - Word Search

Key Idea:
To determine if a word exists in a 2D board, we can perform a depth-first search (DFS) starting from each cell in the board. We'll explore all possible paths to find the target word.

Approach:
1. Create a helper function, dfs, that takes four parameters: the current row and column indices, the current position in the word, and the board.
2. In the helper function:
    a. Check if the current position in the word matches the character at the current row and column in the board.
   b. If not, return false, indicating that the word cannot be formed from this position.
    c. If the current position in the word matches the character at the current row and column, mark the cell as visited (e.g., change the character to a special character like '#') to avoid revisiting it during the search.
   d. Recursively call the helper function for the adjacent cells (up, down, left, and right) with the updated position in the word.
    e. After the recursive calls, backtrack by restoring the original character at the current cell.
3. Iterate through all cells in the board, and for each cell, start the search with the first character of the word.
4. If any search from any cell returns true, it means the word exists on the board, so return true.
5. If no search from any cell returns true, return false, indicating that the word does not exist on the board.
6. The main function initializes the search for each cell and returns the result.

Time Complexity:
The time complexity of this solution is O(M * N * 4^L), where M and N are the dimensions of the board, L is the length of the word, and 4^L represents the maximum number of recursive calls (exploring all possible paths).

Space Complexity:
The space complexity is O(L), where L is the length of the word, due to the recursive call stack.
*/

impl Solution {
    pub fn exist(board: Vec<Vec<char>>, word: String) -> bool {
        let word_chars: Vec<char> = word.chars().collect();
        let rows = board.len();
        let cols = board[0].len();

        for row in 0..rows {
            for col in 0..cols {
                if Self::dfs(&mut board.clone(), row, col, &word_chars, 0) {
                    return true; // Word found
                }
            }
        }

        false
    }

    // Helper function for DFS search
    fn dfs(
        board: &mut Vec<Vec<char>>,
        row: usize,
        col: usize,
```

```
        word_chars: &Vec<char>,
        word_idx: usize,
    ) -> bool {
        if word_idx == word_chars.len() {
            return true; // Word found
        }

        if row < 0 || col < 0 || row >= board.len() || col >= board[0].len() {
            return false; // Out of bounds
        }

        if board[row][col] != word_chars[word_idx] {
            return false; // Mismatched character
        }

        // Mark the cell as visited
        let original_char = board[row][col];
        board[row][col] = '#';

        // Explore adjacent cells
        let found = Self::dfs(board, row - 1, col, word_chars, word_idx + 1)
            || Self::dfs(board, row + 1, col, word_chars, word_idx + 1)
            || Self::dfs(board, row, col - 1, word_chars, word_idx + 1)
            || Self::dfs(board, row, col + 1, word_chars, word_idx + 1);

        // Restore the original character
        board[row][col] = original_char;

        found
    }
}
```

76. 10_Backtracking/07_Palindrome_Partitioning/0131-palindrome-partitioning.rs

```
/*
Problem: LeetCode 131 - Palindrome Partitioning

Key Idea:
To generate all possible palindrome partitions of a given string, we can use a backtracking approach. We start from the beginning of the string and recursively try to partition it into palindromes.

Approach:
1. Create a helper function, backtrack, that takes three parameters: the current position in the string, the current partition, and the result vector to store all valid partitions.
2. In the helper function:
    a. Check if the current position has reached the end of the string. If yes, add the current partition to the result vector.
   b. Otherwise, iterate through the string from the current position to the end.
    c. For each substring from the current position to the current index, check if it is a palindrome.
    d. If it is a palindrome, add it to the current partition and recursively call the helper function for the next position.
    e. After the recursive call, remove the last added substring to backtrack and explore other possibilities.
3. Initialize an empty result vector to store all valid partitions.
4. Start the recursive partitioning from the beginning of the string with an empty current partition.
5. Return the result vector containing all valid partitions.

Time Complexity:
The time complexity of this solution is O(N * 2^N), where N is the length of the input string. In the worst case, there can be 2^N possible partitions, and for each partition, we spend O(N) time to check if it is a palindrome.

Space Complexity:
The space complexity is O(N) for the recursive call stack and O(N) for the result vector, so it is O(N) in total.
*/

impl Solution {
    pub fn partition(s: String) -> Vec<Vec<String>> {
        let mut result: Vec<Vec<String>> = Vec::new();
        let mut current: Vec<String> = Vec::new();

        Self::backtrack(&s, 0, &mut current, &mut result);

        result
    }

    fn is_palindrome(s: &str) -> bool {
        let chars: Vec<char> = s.chars().collect();
        let mut left = 0;
        let mut right = chars.len() - 1;
        while left < right {
            if chars[left] != chars[right] {
                return false;
            }
            left += 1;
            right -= 1;
        }
        true
    }
```

```
    fn backtrack(s: &str, start: usize, current: &mut Vec<String>, result: &mut Vec<Vec<String>>)
{
        if start == s.len() {
            // Reached the end of the string, add the current partition to the result.
            result.push(current.clone());
            return;
        }

        for end in start + 1..=s.len() {
            let substr = &s[start..end];
            if Self::is_palindrome(substr) {
                current.push(substr.to_string());
                Self::backtrack(s, end, current, result);
                current.pop(); // Backtrack
            }
        }
    }
}
```

77. 10_Backtracking/08_Letter_Combinations_of_a_Phone_Number/0017-letter-comb

```
/*
Problem: LeetCode 17 - Letter Combinations of a Phone Number

Key Idea:
To generate all possible letter combinations of a phone number, we can use a backtracking approach. We start with an empty combination and explore all possible letters for each digit of the phone number.

Approach:
1. Create a helper function, backtrack, that takes three parameters: the current digit index, the current combination, and the result vector to store all valid combinations.
2. In the helper function:
   a. Check if the current digit index has reached the end of the phone number. If yes, add the current combination to the result vector.
   b. Otherwise, get the letters corresponding to the current digit.
   c. Iterate through the letters and append each letter to the current combination.
   d. Recursively call the helper function for the next digit index.
   e. After the recursive call, remove the last added letter to backtrack and explore other possibilities.
3. Initialize an empty result vector to store all valid combinations.
4. Start the recursive combination from the first digit with an empty current combination.
5. Return the result vector containing all valid combinations.

Time Complexity:
The time complexity of this solution is O(4^N * N), where N is the number of digits in the input phone number. For each digit, there can be up to 4 letters, and we explore all possible combinations.

Space Complexity:
The space complexity is O(N) for the recursive call stack and O(N) for the result vector, so it is O(N) in total.
*/

impl Solution {
    pub fn letter_combinations(digits: String) -> Vec<String> {
        let mapping: Vec<Vec<char>> = vec![
            vec![' '],
            vec![],
            vec!['a', 'b', 'c'],
            vec!['d', 'e', 'f'],
            vec!['g', 'h', 'i'],
            vec!['j', 'k', 'l'],
            vec!['m', 'n', 'o'],
            vec!['p', 'q', 'r', 's'],
            vec!['t', 'u', 'v'],
            vec!['w', 'x', 'y', 'z'],
        ];

        let mut result: Vec<String> = Vec::new();
        let mut current: String = String::new();

        if !digits.is_empty() {
            Self::backtrack(&digits, 0, &mut current, &mut result, &mapping);
        }

        result
    }

    fn backtrack(
        digits: &str,
```

```
        index: usize,
        current: &mut String,
        result: &mut Vec<String>,
        mapping: &Vec<Vec<char>>,
    ) {
        if index == digits.len() {
            // Reached the end of the digits, add the current combination to the result.
            result.push(current.clone());
            return;
        }

        let digit = digits.chars().nth(index).unwrap() as usize - '0' as usize;
        for &letter in &mapping[digit] {
            current.push(letter);
            Self::backtrack(digits, index + 1, current, result, mapping);
            current.pop(); // Backtrack
        }
    }
}

/*
// Faster Solution

impl Solution {
    pub fn letter_combinations(digits: String) -> Vec<String> {
        if digits.is_empty() {
            return Vec::new();
        }

        const PHONE_MAP: [&str; 8] = ["abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"];
        let mut combinations = Vec::new();

        fn backtrack(
            current_combination: String,
            remaining_digits: &str,
            phone_map: &[&str; 8],
            combinations: &mut Vec<String>,
        ) {
            if remaining_digits.is_empty() {
                combinations.push(current_combination);
            } else {
                let digit = remaining_digits.chars().next().unwrap();
                let digit_idx = digit as usize - '2' as usize;
                let letters = phone_map[digit_idx];
                for letter in letters.chars() {
                    let new_combination = format!("{}{}", current_combination, letter);
                    backtrack(new_combination, &remaining_digits[1..], phone_map, combinations);
                }
            }
        }

        backtrack(String::new(), &digits, &PHONE_MAP, &mut combinations);
        combinations
    }
}
*/
```

78. 10_Backtracking/09_N_Queens/0051-n-queens.rs

```
/*
Problem: LeetCode 51 - N-Queens

Key Idea:
The N-Queens problem is a classic backtracking problem. The key idea is to place queens on a chessboard of size N×N in such a way that no two queens threaten each other.

Approach:
1. Create a helper function, `backtrack`, which uses backtracking to find all valid solutions.
2. Initialize an empty board of size N×N represented as a 2D vector of characters. Initially, all cells are filled with '.' to represent empty cells.
3. In the `backtrack` function:
   a. Start by placing queens row by row.
   b. For each row, try placing a queen in each column and check if it's a valid placement (no other queens threaten it).
   c. If a valid placement is found, mark that cell as 'Q', and recursively move on to the next row.
   d. After placing queens and recursively checking all rows, if we reach the N-th row, it means we have found a valid solution. Add this solution to the result.
   e. Regardless of whether a solution is found or not, backtrack by marking the current cell as '.' again and exploring other possibilities.
4. Initialize an empty result vector to store all valid solutions.
5. Start the backtracking process from the first row.
6. Return the result vector containing all valid solutions.
7. Convert the result into the required format where each solution is represented as a vector of strings (each string representing a row of the chessboard).

Time Complexity:
The time complexity is O(N!), where N is the size of the chessboard. This is because there are N possibilities for the queen placement in the first row, (N-1) possibilities for the second row, (N-2) for the third, and so on. In the worst case, we explore all possible permutations.

Space Complexity:
The space complexity is O(N^2) for the chessboard.
*/

impl Solution {
    pub fn solve_n_queens(n: i32) -> Vec<Vec<String>> {
        let mut board: Vec<Vec<char>> = vec![vec!['.'; n as usize]; n as usize];
        let mut result: Vec<Vec<String>> = Vec::new();

        Self::backtrack(n, 0, &mut board, &mut result);

        result
    }

    fn backtrack(n: i32, row: i32, board: &mut Vec<Vec<char>>, result: &mut Vec<Vec<String>>) {
        if row == n {
            // We've successfully placed queens in all rows, so add this solution.
            let solution: Vec<String> = board.iter().map(|row| row.iter().collect()).collect();
            result.push(solution);
            return;
        }

        for col in 0..n {
            if Self::is_safe(row as usize, col as usize, n as usize, board) {
                // Place the queen in the current cell.
                board[row as usize][col as usize] = 'Q';
                // Recursively try to place queens in the next row.
                Self::backtrack(n, row + 1, board, result);
```

```
                // Backtrack by removing the queen.
                board[row as usize][col as usize] = '.';
            }
        }
    }

    fn is_safe(row: usize, col: usize, n: usize, board: &Vec<Vec<char>>) -> bool {
        // Check the column above this row.
        for i in 0..row {
            if board[i][col] == 'Q' {
                return false;
            }
        }

        // Check the upper-left diagonal.
        let mut i = row as i32 - 1;
        let mut j = col as i32 - 1;
        while i >= 0 && j >= 0 {
            if board[i as usize][j as usize] == 'Q' {
                return false;
            }
            i -= 1;
            j -= 1;
        }

        // Check the upper-right diagonal.
        let mut i = row as i32 - 1;
        let mut j = col as i32 + 1;
        while i >= 0 && j < n as i32 {
            if board[i as usize][j as usize] == 'Q' {
                return false;
            }
            i -= 1;
            j += 1;
        }

        true
    }
}

/*
// Bitmask solution

impl Solution {
    pub fn solve_n_queens(n: i32) -> Vec<Vec<String>> {
        let mut result = Vec::new();
        let mut path = Vec::new();
        let n = n as usize;
        let bitmask = (1 << n) - 1;
        let mut diagonal_135 = 0;
        let mut diagonal_45 = 0;
        let mut col_mask = 0;

        fn dfs(
            n: usize,
            bitmask: i32,
            diagonal_135: i32,
            diagonal_45: i32,
            col_mask: i32,
            result: &mut Vec<Vec<String>>,
            path: &mut Vec<usize>,
        ) {
            if bitmask == col_mask {
                result.push(decode(&path, n));
```

```
            return;
        }

        let available = bitmask & (!(diagonal_135 | diagonal_45 | col_mask));

        for i in 0..n {
            let bit_info = 1 << i;
            if available & bit_info == 0 {
                continue;
            }
            path.push(i);
            dfs(
                n,
                bitmask,
                (diagonal_135 | bit_info) >> 1,
                (diagonal_45 | bit_info) << 1,
                col_mask | bit_info,
                result,
                path,
            );
            path.pop();
        }
    }

    fn decode(path: &[usize], n: usize) -> Vec<String> {
        path.iter()
            .enumerate()
            .fold(vec![vec!['.'; n]; n], |mut acc, (i, &j)| {
                acc[i][j] = 'Q';
                acc
            })
            .into_iter()
            .map(|x| x.into_iter().collect())
            .collect()
    }

    dfs(n, bitmask, diagonal_135, diagonal_45, col_mask, &mut result, &mut path);
    result
  }
}
*/
```

79. 11_Graphs/01_Number_of_Islands/0200-number-of-islands.rs

```
/*
Problem: LeetCode 200 - Number of Islands

Key Idea:
The key idea is to use Depth-First Search (DFS) to count the number of connected islands in a grid.

Approach:
1. We can approach this problem by iterating through the grid and using DFS to explore each island.
2. Initialize a variable `count` to keep track of the number of islands.
3. Iterate through each cell in the grid:
    - If the cell is '1' (land), increment `count` and start a DFS from that cell to mark all connected land cells as visited.
     - During DFS, mark visited cells as '0' to avoid counting them again.
4. Continue this process for all cells in the grid.
5. Return the `count` as the result.

Time Complexity:
O(m * n), where `m` is the number of rows and `n` is the number of columns in the grid. We visit each cell once.

Space Complexity:
O(m * n), as in the worst case, the entire grid can be composed of '1's and require a DFS stack.
*/

impl Solution {
    pub fn num_islands(grid: Vec<Vec<char>>) -> i32 {
        if grid.is_empty() || grid[0].is_empty() {
            return 0;
        }

        let mut grid = grid; // Make grid mutable

        let rows = grid.len();
        let cols = grid[0].len();
        let mut count = 0;

        for i in 0..rows {
            for j in 0..cols {
                if grid[i][j] == '1' {
                    count += 1;
                    Solution::dfs(&mut grid, i, j);
                }
            }
        }

        count
    }

    fn dfs(grid: &mut Vec<Vec<char>>, i: usize, j: usize) {
        let rows = grid.len();
        let cols = grid[0].len();

        if i >= rows || j >= cols || grid[i][j] == '0' {
            return;
        }

        grid[i][j] = '0'; // Mark as visited
```

```
        // Explore neighbors in all four directions
        let directions = [(0, 1), (0, -1), (1, 0), (-1, 0)];
        for (dx, dy) in directions.iter() {
            let x = (i as isize + dx) as usize;
            let y = (j as isize + dy) as usize;
            if x < rows && y < cols && grid[x][y] == '1' {
                Solution::dfs(grid, x, y);
            }
        }
    }
}
```

80. 11_Graphs/02_Clone_Graph/0133-clone-graph.rs

```
/*
Problem: LeetCode 133 - Clone Graph

Key Idea:
The key idea is to perform a depth-first traversal of the original graph while creating a new node for each node in the original graph. We'll use a hashmap to keep track of the mapping between the original nodes and their corresponding new nodes.

Approach:
1. We can approach this problem using depth-first traversal (DFS) with the help of recursion.
2. Create a HashMap to store the mapping between the original nodes and their corresponding new nodes.
3. Initialize a DFS function that takes the current node from the original graph as an argument.
4. In the DFS function:
   - Create a new node corresponding to the current node from the original graph.
   - Iterate through the neighbors of the current node.
   - For each neighbor, recursively call the DFS function to create its corresponding new node and add it to the neighbors of the new node.
   - Add the new node to the mapping in the HashMap.
5. Start the DFS from the initial node of the original graph.
6. Return the new node corresponding to the initial node as the result.

Time Complexity:
O(N + E), where N is the number of nodes in the graph, and E is the number of edges. We visit each node once and each edge once during the DFS traversal.

Space Complexity:
O(N), where N is the number of nodes in the graph. This is the space required for the HashMap and the recursion stack.
*/

// There is no option to solve this problem in Rust at the time of writing this
```

81. 11_Graphs/03_Max_Area_of_Island/0695-max-area-of-island.rs

```
/*
Problem: LeetCode 695 - Max Area of Island

Key Idea:
The key idea is to use depth-first search (DFS) to traverse the grid and identify connected islands. We maintain a count of the area of each island and keep track of the maximum area encountered.

Approach:
1. We can approach this problem using depth-first search (DFS).
2. Initialize a variable `max_area` to 0 to keep track of the maximum area of an island.
3. Iterate through the grid cells:
   - For each unvisited '1' cell, start a DFS to explore the island.
   - During DFS, mark visited cells as '0' to avoid counting them multiple times.
   - Increment the area count for the current island.
   - Update `max_area` with the maximum of the current area and the previous maximum.
4. Continue this process for all cells in the grid.
5. Return `max_area` as the result.

Time Complexity:
O(m * n), where `m` is the number of rows and `n` is the number of columns in the grid. We visit each cell once.

Space Complexity:
O(m * n) for the recursion stack, as in the worst case, we might have to explore all cells in the grid.
*/

impl Solution {
    pub fn max_area_of_island(grid: Vec<Vec<i32>>) -> i32 {
        let mut grid = grid; // Make grid mutable

        let rows = grid.len();
        let cols = grid[0].len();
        let mut max_area = 0;

        for i in 0..rows {
            for j in 0..cols {
                if grid[i][j] == 1 {
                    let area = Solution::dfs(&mut grid, i, j);
                    max_area = max_area.max(area);
                }
            }
        }

        max_area
    }

    fn dfs(grid: &mut Vec<Vec<i32>>, i: usize, j: usize) -> i32 {
        let rows = grid.len();
        let cols = grid[0].len();

        if i >= rows || j >= cols || grid[i][j] == 0 {
            return 0;
        }

        grid[i][j] = 0; // Mark as visited

        // Explore neighbors in all four directions
        let directions = [(0, 1), (0, -1), (1, 0), (-1, 0)];
```

```
        let mut area = 1;

        for (dx, dy) in directions.iter() {
            let x = (i as isize + dx) as usize;
            let y = (j as isize + dy) as usize;
            area += Solution::dfs(grid, x, y);
        }

        area
    }
}
```

82. 11_Graphs/04_Pacific_Atlantic_Water_Flow/0417-pacific-atlantic-water-flow.rs

```
/*
Problem: LeetCode 417 - Pacific Atlantic Water Flow

Key Idea:
The key idea is to perform depth-first search (DFS) from the ocean boundaries (Pacific and Atlantic) to identify cells that can flow water to both oceans. We'll use two boolean grids to mark cells reachable by each ocean, and the intersection of these grids will give us the answer.

Approach:
1. Create two boolean grids, one for the Pacific Ocean and one for the Atlantic Ocean, both initialized to false.
2. Start DFS from the boundaries of the grid (ocean coasts) and mark cells as reachable for each ocean separately.
   - For the Pacific Ocean, start DFS from the top row and left column.
   - For the Atlantic Ocean, start DFS from the bottom row and right column.
3. After the DFS is complete for both oceans, iterate through all cells in the grid.
   - If a cell is marked as reachable by both oceans, it can flow water to both oceans. Add it to the result.
4. Return the result, which is a list of cells that can flow water to both oceans.

Time Complexity:
O(m * n), where `m` is the number of rows and `n` is the number of columns in the grid. We visit each cell once.

Space Complexity:
O(m * n) for the two boolean grids, as we need to mark each cell in the grid.
*/

impl Solution {
    pub fn pacific_atlantic(matrix: Vec<Vec<i32>>) -> Vec<Vec<i32>> {
        let rows = matrix.len();
        if rows == 0 {
            return Vec::new();
        }
        let cols = matrix[0].len();

        let mut pacific_reachable = vec![vec![false; cols]; rows];
        let mut atlantic_reachable = vec![vec![false; cols]; rows];

        for i in 0..rows {
            Solution::dfs(&matrix, &mut pacific_reachable, i, 0, i, 0);
            Solution::dfs(&matrix, &mut atlantic_reachable, i, cols - 1, i, cols - 1);
        }

        for j in 0..cols {
            Solution::dfs(&matrix, &mut pacific_reachable, 0, j, 0, j);
            Solution::dfs(&matrix, &mut atlantic_reachable, rows - 1, j, rows - 1, j);
        }

        let mut result = Vec::new();
        for i in 0..rows {
            for j in 0..cols {
                if pacific_reachable[i][j] && atlantic_reachable[i][j] {
                    result.push(vec![i as i32, j as i32]);
                }
            }
        }

        result
    }
```

```
    fn dfs(
        matrix: &Vec<Vec<i32>>,
        reachable: &mut Vec<Vec<bool>>,
        i: usize,
        j: usize,
        prev_i: usize,
        prev_j: usize,
    ) {
        let rows = matrix.len();
        let cols = matrix[0].len();

        if i < 0
            || i >= rows
            || j < 0
            || j >= cols
            || reachable[i][j]
            || matrix[i][j] < matrix[prev_i][prev_j]
        {
            return;
        }

        reachable[i][j] = true;

        let directions = [(0, 1), (0, -1), (1, 0), (-1, 0)];
        for (dx, dy) in directions.iter() {
            let x = (i as isize + dx) as usize;
            let y = (j as isize + dy) as usize;
            Solution::dfs(matrix, reachable, x, y, i, j);
        }
    }
}
```

83. 11_Graphs/05_Surrounded_Regions/0130-surrounded-regions.rs

```
/*
Problem: LeetCode 130 - Surrounded Regions

Key Idea:
The key idea is to use a depth-first search (DFS) or breadth-first search (BFS) algorithm to mark and traverse the regions connected to the boundaries (ocean edges) of the board. Any 'O' that is not marked as part of these connected regions must be surrounded by 'X'.

Approach:
1. Iterate through the boundaries (ocean edges) of the board.
   - For each 'O' cell encountered, perform a DFS or BFS to mark all connected 'O' cells as 'B'.
2. After marking the connected regions to the boundaries, iterate through the entire board.
   - Replace 'O' cells with 'X' and 'B' cells with 'O'.
3. The board is now updated with 'X' for cells surrounded by 'X' and 'O' for cells connected to the boundaries.

Time Complexity:
O(m * n), where `m` is the number of rows and `n` is the number of columns in the board. We visit each cell once.

Space Complexity:
O(m * n) for the recursive stack (DFS) or queue (BFS) space.
*/

impl Solution {
    pub fn solve(board: &mut Vec<Vec<char>>) {
        let rows = board.len();
        if rows == 0 {
            return;
        }
        let cols = board[0].len();

        // Mark and traverse the connected regions to the top and bottom boundaries
        for i in 0..rows {
            Solution::dfs(board, i, 0);
            Solution::dfs(board, i, cols - 1);
        }

        // Mark and traverse the connected regions to the left and right boundaries
        for j in 0..cols {
            Solution::dfs(board, 0, j);
            Solution::dfs(board, rows - 1, j);
        }

        // Update the board based on the marked cells
        for i in 0..rows {
            for j in 0..cols {
                if board[i][j] == 'O' {
                    board[i][j] = 'X';
                } else if board[i][j] == 'B' {
                    board[i][j] = 'O';
                }
            }
        }
    }

    fn dfs(board: &mut Vec<Vec<char>>, i: usize, j: usize) {
        let rows = board.len();
        let cols = board[0].len();
```

```
        if i < 0 || i >= rows || j < 0 || j >= cols || board[i][j] != 'O' {
            return;
        }

        board[i][j] = 'B'; // Mark as part of connected region

        let directions = [(0, 1), (0, -1), (1, 0), (-1, 0)];
        for (dx, dy) in directions.iter() {
            let x = (i as isize + dx) as usize;
            let y = (j as isize + dy) as usize;
            Solution::dfs(board, x, y);
        }
    }
}
```

84. 11_Graphs/06_Rotting_Oranges/0994-rotting-oranges.rs

```
/*
Problem: LeetCode 994 - Rotting Oranges

Key Idea:
The key idea is to perform a breadth-first search (BFS) traversal of the grid, where the initial rotten oranges are treated as the starting points. We keep track of the time taken for each orange to rot and find the maximum time.

Approach:
1. Initialize a queue for BFS and a variable `time` to 0.
2. Iterate through the grid and enqueue the positions of initially rotten oranges (with value 2) into the queue. Also, count the number of fresh oranges.
3. While the queue is not empty:
    - For each level of BFS (representing a minute), process all oranges in the queue at that level:
     - Dequeue an orange and mark its neighbors as rotten if they are fresh.
     - Decrement the count of fresh oranges.
     - Increment the `time` for the current level.
   - After processing a level, increment `time` to indicate a minute has passed.
4. If there are no fresh oranges remaining, return `time`. Otherwise, return -1 to indicate that some oranges cannot rot.

Time Complexity:
O(m * n), where `m` is the number of rows and `n` is the number of columns in the grid. In the worst case, we may visit all cells once.

Space Complexity:
O(m * n) for the queue and the grid.
*/

use std::collections::VecDeque;

impl Solution {
    pub fn oranges_rotting(mut grid: Vec<Vec<i32>>) -> i32 {
        let rows = grid.len();
        if rows == 0 {
            return 0;
        }
        let cols = grid[0].len();

        let mut queue = VecDeque::new();
        let mut fresh_count = 0;

        for i in 0..rows {
            for j in 0..cols {
                if grid[i][j] == 2 {
                    queue.push_back((i, j, 0)); // (row, col, time)
                } else if grid[i][j] == 1 {
                    fresh_count += 1;
                }
            }
        }

        let directions = [(0, 1), (0, -1), (1, 0), (-1, 0)];
        let mut time = 0;

        while !queue.is_empty() {
            let (x, y, t) = queue.pop_front().unwrap();

            for (dx, dy) in directions.iter() {
```

```
                let new_x = x as isize + dx;
                let new_y = y as isize + dy;

                if new_x >= 0 && new_x < rows as isize && new_y >= 0 && new_y < cols as isize {
                    let new_x = new_x as usize;
                    let new_y = new_y as usize;

                    if grid[new_x][new_y] == 1 {
                        grid[new_x][new_y] = 2;
                        fresh_count -= 1;
                        queue.push_back((new_x, new_y, t + 1));
                    }
                }
            }

            if fresh_count == 0 {
                time = t; // All oranges have rotted
            }
        }

        if fresh_count == 0 {
            time
        } else {
            -1 // Some oranges cannot rot
        }
    }
}
```

85. 11_Graphs/07_Walls_and_Gates/0286-walls-and-gates.rs

```
/*
Problem: LeetCode 286 - Walls and Gates

Key Idea:
The key idea is to perform a breadth-first search (BFS) from each gate to find the distance from that gate to each empty room (represented by INF). We start BFS from all gates simultaneously to propagate distances efficiently.

Approach:
1. Create a queue and enqueue all gate positions, initializing their distance as 0.
2. Perform BFS:
   - Dequeue a position from the queue.
   - Explore its neighbors (up, down, left, right).
   - If a neighbor is an empty room (INF), update its distance and enqueue it.
   - Repeat until the queue is empty.
3. After BFS from all gates, the grid contains the shortest distance from each gate to each empty room.
4. Return the updated grid.

Time Complexity:
O(m * n), where `m` is the number of rows and `n` is the number of columns in the grid. In the worst case, we might visit all cells once.

Space Complexity:
O(m * n) for the queue and other variables.
*/

impl Solution {
    pub fn walls_and_gates(rooms: &mut Vec<Vec<i32>>) {
        let rows = rooms.len();
        if rows == 0 {
            return;
        }
        let cols = rooms[0].len();

        let directions = [(0, 1), (0, -1), (1, 0), (-1, 0)];
        let mut queue = std::collections::VecDeque::new();

        // Enqueue all gate positions
        for i in 0..rows {
            for j in 0..cols {
                if rooms[i][j] == 0 {
                    queue.push_back((i, j, 0));
                }
            }
        }

        while !queue.is_empty() {
            if let Some((x, y, distance)) = queue.pop_front() {
                for (dx, dy) in directions.iter() {
                    let new_x = (x as i32 + dx) as usize;
                    let new_y = (y as i32 + dy) as usize;

                    if new_x < rows && new_y < cols && rooms[new_x][new_y] == std::i32::MAX {
                        rooms[new_x][new_y] = distance + 1;
                        queue.push_back((new_x, new_y, distance + 1));
                    }
                }
            }
        }
```

```
    }
}
```

86. 11_Graphs/08_Course_Schedule/0207-course-schedule.rs

```
/*
Problem: LeetCode 207 - Course Schedule

Key Idea:
The key idea is to use a directed graph to represent the prerequisites as edges between courses.
If there is a cycle in the graph, it means it's impossible to complete all courses.

Approach:
1. Create an adjacency list representation of the directed graph.
2. Initialize an array to keep track of the in-degrees (number of prerequisites) for each course.
3. Initialize a queue and enqueue all courses with no prerequisites (in-degree = 0).
4. While the queue is not empty:
   - Dequeue a course.
   - Decrement the in-degrees of its neighbors (courses that depend on it).
   - If a neighbor's in-degree becomes 0, enqueue it.
   - Increment a counter to track the number of courses taken.
5. If the counter equals the total number of courses, it means all courses can be taken, and we return true.
6. Otherwise, return false (there is a cycle in the graph).

Time Complexity:
O(V + E), where V is the number of courses (vertices) and E is the number of prerequisites (edges). We visit each vertex and each edge once.

Space Complexity:
O(V + E) for the adjacency list and in-degrees.
*/

impl Solution {
    pub fn can_finish(num_courses: i32, prerequisites: Vec<Vec<i32>>) -> bool {
        let mut graph: Vec<Vec<i32>> = vec![vec![]; num_courses as usize];
        let mut in_degrees: Vec<i32> = vec![0; num_courses as usize];

        // Build the directed graph and in-degrees
        for prerequisite in prerequisites.iter() {
            let course = prerequisite[0] as usize;
            let prereq = prerequisite[1] as usize;
            graph[prereq].push(course as i32);
            in_degrees[course as usize] += 1;
        }

        let mut queue: std::collections::VecDeque<i32> = std::collections::VecDeque::new();
        let mut courses_taken = 0;

        // Enqueue courses with no prerequisites
        for (course, &in_degree) in in_degrees.iter().enumerate() {
            if in_degree == 0 {
                queue.push_back(course as i32);
                courses_taken += 1;
            }
        }

        while let Some(course) = queue.pop_front() {
            for &next_course in graph[course as usize].iter() {
                in_degrees[next_course as usize] -= 1;
                if in_degrees[next_course as usize] == 0 {
                    queue.push_back(next_course);
                    courses_taken += 1;
                }
            }
```

```
        }

        courses_taken == num_courses
    }
}
```

87. 11_Graphs/09_Course_Schedule_II/0210-course-schedule-ii.rs

```
/*
Problem: LeetCode 210 - Course Schedule II

Key Idea:
The key idea is to use a directed graph to represent the prerequisites as edges between courses.
We need to find an order in which all courses can be taken, or determine that it's impossible.

Approach:
1. Create an adjacency list representation of the directed graph.
2. Initialize an array to keep track of the in-degrees (number of prerequisites) for each course.
3. Initialize a queue and enqueue all courses with no prerequisites (in-degree = 0).
4. Initialize a result vector to store the order of courses taken.
5. While the queue is not empty:
   - Dequeue a course.
   - Append it to the result vector.
   - Decrement the in-degrees of its neighbors (courses that depend on it).
   - If a neighbor's in-degree becomes 0, enqueue it.
6. If the size of the result vector equals the total number of courses, return it (all courses can
be taken).
7. Otherwise, return an empty vector (impossible to complete all courses).

Time Complexity:
O(V + E), where V is the number of courses (vertices) and E is the number of prerequisites
(edges). We visit each vertex and each edge once.

Space Complexity:
O(V + E) for the adjacency list, in-degrees, queue, and result vector.
*/

impl Solution {
    pub fn find_order(num_courses: i32, prerequisites: Vec<Vec<i32>>) -> Vec<i32> {
        let num_courses = num_courses as usize;
        let mut graph: Vec<Vec<i32>> = vec![vec![]; num_courses];
        let mut in_degrees: Vec<i32> = vec![0; num_courses];

        // Build the directed graph and in-degrees
        for prerequisite in prerequisites.iter() {
            let course = prerequisite[0] as usize;
            let prereq = prerequisite[1] as usize;
            graph[prereq].push(course as i32);
            in_degrees[course as usize] += 1;
        }

        let mut queue: std::collections::VecDeque<i32> = std::collections::VecDeque::new();
        let mut result: Vec<i32> = Vec::new();

        // Enqueue courses with no prerequisites
        for (course, &in_degree) in in_degrees.iter().enumerate() {
            if in_degree == 0 {
                queue.push_back(course as i32);
                result.push(course as i32);
            }
        }

        while let Some(course) = queue.pop_front() {
            for &next_course in graph[course as usize].iter() {
                in_degrees[next_course as usize] -= 1;
                if in_degrees[next_course as usize] == 0 {
                    queue.push_back(next_course);
                    result.push(next_course);
```

```
                }
            }
        }

        if result.len() == num_courses {
            result
        } else {
            Vec::new()
        }
    }
}
```

88. 11_Graphs/10_Redundant_Connection/0684-redundant-connection.rs

```
/*
Problem: LeetCode 684 - Redundant Connection

Key Idea:
The key idea is to use a Union-Find (Disjoint Set Union) data structure to detect the redundant connection that forms a cycle in the given graph.

Approach:
1. Initialize a parent array where each node initially points to itself.
2. Iterate through the given edges:
   - For each edge (u, v), find the root nodes of u and v (representing their sets).
   - If the root nodes are the same, it means there is already a path from u to v, and adding this edge would create a cycle. This edge is the redundant connection.
   - Otherwise, union the two sets by updating the parent of one of the root nodes to point to the other.
3. Return the redundant connection edge.

Time Complexity:
O(n * ?(n)), where n is the number of nodes, and ?(n) is the inverse Ackermann function, which is very slow-growing. In practice, ?(n) is close to a constant, so the time complexity is nearly linear.

Space Complexity:
O(n) for the parent array.
*/

impl Solution {
    pub fn find_redundant_connection(edges: Vec<Vec<i32>>) -> Vec<i32> {
        let n = edges.len();
        let mut parent: Vec<usize> = (0..n + 1).collect();

        fn find(mut x: usize, parent: &mut Vec<usize>) -> usize {
            while parent[x] != x {
                x = parent[x];
            }
            x
        }

        for edge in edges.iter() {
            let u = edge[0] as usize;
            let v = edge[1] as usize;

            let root_u = find(u, &mut parent);
            let root_v = find(v, &mut parent);

            if root_u == root_v {
                return edge.clone();
            } else {
                parent[root_u] = root_v;
            }
        }

        vec![]
    }
}
```

89. 11_Graphs/11_Number_of_Connected_Components_in_an_Undirected_Graph/03

```
/*
Problem: LeetCode 323 - Number of Connected Components in an Undirected Graph

Key Idea:
The key idea is to use Depth-First Search (DFS) to find connected components in the undirected
graph. We count the number of times we perform DFS as the number of connected components.

Approach:
1. Build an adjacency list representation of the undirected graph.
2. Initialize a set to keep track of visited nodes.
3. Initialize a variable `num_components` to count the number of connected components.
4. Iterate through all nodes (0 to n-1):
   - If the node is not in the visited set:
     - Perform DFS starting from this node.
     - Increment `num_components` by 1 after each DFS call.
5. Return `num_components` as the result.

Time Complexity:
O(V + E), where V is the number of nodes (vertices) and E is the number of edges in the graph.

Space Complexity:
O(V + E) for the adjacency list and the visited set.
*/

impl Solution {
    pub fn count_components(n: i32, edges: Vec<Vec<i32>>) -> i32 {
        let n = n as usize;
        let mut graph: Vec<Vec<usize>> = vec![vec![]; n];

        // Build the adjacency list
        for edge in edges.iter() {
            let u = edge[0] as usize;
            let v = edge[1] as usize;
            graph[u].push(v);
            graph[v].push(u);
        }

        let mut visited: Vec<bool> = vec![false; n];
        let mut num_components = 0;

        // Perform DFS to find connected components
        for node in 0..n {
            if !visited[node] {
                Self::dfs(node, &mut visited, &graph);
                num_components += 1;
            }
        }

        num_components
    }

    fn dfs(node: usize, visited: &mut Vec<bool>, graph: &Vec<Vec<usize>>) {
        visited[node] = true;

        for &neighbor in graph[node].iter() {
            if !visited[neighbor] {
                Self::dfs(neighbor, visited, graph);
            }
        }
    }
```

```
}
```

90. 11_Graphs/12_Graph_Valid_Tree/0261-graph-valid-tree.rs

```
/*
Problem: LeetCode 261 - Graph Valid Tree

Key Idea:
The key idea is to check whether the given undirected graph forms a valid tree. A valid tree has two properties:
1. It has n - 1 edges, where n is the number of nodes.
2. It is a connected graph, meaning all nodes are reachable from any node.

Approach:
1. Build an adjacency list representation of the undirected graph.
2. Initialize a set to keep track of visited nodes.
3. Perform a Depth-First Search (DFS) starting from node 0.
4. During DFS, check if we encounter any visited node that is not the parent of the current node (to avoid cycles).
5. After DFS, check if all nodes are visited. If not, return false (not connected).
6. Finally, check if the number of edges is n - 1. If true, return true; otherwise, return false.

Time Complexity:
O(V + E), where V is the number of nodes (vertices) and E is the number of edges in the graph.

Space Complexity:
O(V + E) for the adjacency list and visited set.
*/

impl Solution {
    pub fn valid_tree(n: i32, edges: Vec<Vec<i32>>) -> bool {
        let n = n as usize;
        let mut graph: Vec<Vec<usize>> = vec![vec![]; n];

        // Build the adjacency list
        for edge in edges.iter() {
            let u = edge[0] as usize;
            let v = edge[1] as usize;
            graph[u].push(v);
            graph[v].push(u);
        }

        let mut visited: Vec<bool> = vec![false; n];

        // Perform DFS to check connectivity
        if !Self::dfs(0, &mut visited, &graph, n, -1) {
            return false;
        }

        // Check if all nodes are visited
        if visited.iter().all(|&v| v) {
            return edges.len() == n - 1;
        }

        false
    }

    fn dfs(
        node: usize,
        visited: &mut Vec<bool>,
        graph: &Vec<Vec<usize>>,
        n: usize,
        parent: i32,
    ) -> bool {
```

```
        visited[node] = true;

        for &neighbor in graph[node].iter() {
            if !visited[neighbor] {
                if !Self::dfs(neighbor, visited, graph, n, node as i32) {
                    return false;
                }
            } else if neighbor as i32 != parent {
                return false;
            }
        }

        true
    }
}
```

91. 11_Graphs/13_Word_Ladder/0127-word-ladder.rs

```
/*
Problem: LeetCode 127 - Word Ladder

Key Idea:
The key idea is to treat the given word list as a graph, where each word is a node, and two words are connected if they differ by only one character. We can then perform a breadth-first search (BFS) to find the shortest transformation sequence from the beginWord to the endWord.

Approach:
1. Build a set for fast word lookup in the word list.
2. Initialize a queue for BFS, starting with the beginWord.
3. Initialize a set to keep track of visited words.
4. Initialize a level variable to track the depth of the BFS.
5. Perform BFS:
   - For each word in the current level of the queue:
     - Generate all possible one-character transformations of the word.
     - Check if each transformation is in the word list and not visited.
     - If a transformation is the endWord, return the current level + 1 (depth).
     - Otherwise, add the transformation to the queue for the next level.
     - Mark the transformation as visited.
   - Increment the level.
6. If the BFS completes without finding the endWord, return 0 (no transformation sequence found).

Time Complexity:
O(M^2 * N), where M is the length of each word, and N is the total number of words in the word list. Generating all possible transformations for a word takes O(M^2) time, and we do this for each word in the word list.

Space Complexity:
O(N) for the word set and visited set, and O(M) for the queue and transformations.
*/

use std::collections::{HashSet, VecDeque};

impl Solution {
    pub fn ladder_length(begin_word: String, end_word: String, word_list: Vec<String>) -> i32 {
        let word_set: HashSet<String> = word_list.into_iter().collect();
        let mut visited: HashSet<String> = HashSet::new();
        let mut queue: VecDeque<String> = VecDeque::new();
        queue.push_back(begin_word.clone());
        visited.insert(begin_word.clone());
        let mut level = 1;

        while !queue.is_empty() {
            let level_size = queue.len();

            for _ in 0..level_size {
                if let Some(word) = queue.pop_front() {
                    let chars: Vec<char> = word.chars().collect();

                    for i in 0..chars.len() {
                        let original_char = chars[i];

                        for c in b'a'..=b'z' {
                            chars[i] = c as char;
                            let new_word: String = chars.iter().collect();

                            if word_set.contains(&new_word) {
                                if new_word == end_word {
                                    return level + 1;
```

```
                            }

                            if !visited.contains(&new_word) {
                                visited.insert(new_word.clone());
                                queue.push_back(new_word.clone());
                            }
                        }
                    }

                    chars[i] = original_char;
                }
            }
        }

        level += 1;
    }

    0
  }
}
```

92. 12_Advanced_Graphs/01_Reconstruct_Itinerary/0332-reconstruct-itinerary.rs

```
/*
Problem: LeetCode 332 - Reconstruct Itinerary

Key Idea:
The key idea is to treat the given list of tickets as a directed graph, where each airport is a node, and the tickets represent directed edges. We need to find an itinerary that starts from "JFK" and visits all airports once while minimizing the lexicographical order.

Approach:
1. Build a graph using a HashMap with airport codes as keys and a BinaryHeap (min heap) containing Reverse references to destination airports.
2. Create a vector to store the final itinerary.
3. Start with "JFK" as the initial airport and use a stack to perform a Depth-First Search (DFS).
4. While the stack is not empty:
   - If there are destinations available for the current airport:
     - Pop the smallest destination airport from the BinaryHeap.
     - Push the destination airport onto the stack.
   - If there are no destinations available:
     - Pop the airport from the stack and add it to the final itinerary.
5. Reverse the final itinerary vector to obtain the correct order.

Time Complexity:
- O(E * log E), where E is the number of edges (tickets).
- Building the adjacency list takes O(E) time.
- For each airport, we perform a log E operation when popping from the BinaryHeap.

Space Complexity:
- O(E) for the adjacency list and the final itinerary.
*/

use std::cmp::Reverse;
use std::collections::{BinaryHeap, HashMap};

impl Solution {
    pub fn find_itinerary(tickets: Vec<Vec<String>>) -> Vec<String> {
        let mut graph: HashMap<&str, BinaryHeap<Reverse<&str>>> = HashMap::new();

        // Build the graph.
        for ticket in &tickets {
            graph
                .entry(&ticket[0])
                .or_insert(BinaryHeap::new())
                .push(Reverse(&ticket[1]));
        }

        let mut itinerary: Vec<String> = Vec::with_capacity(tickets.len() + 1);
        let mut stack: Vec<&str> = vec!["JFK"];

        while let Some(src) = stack.last().cloned() {
            if let Some(dsts) = graph.get_mut(src) {
                if let Some(dst) = dsts.pop() {
                    stack.push(dst.0);
                    continue;
                }
            }

            if let Some(last) = stack.pop() {
                itinerary.push(last.to_string());
            }
        }
```

```
        itinerary.reverse();
        itinerary
    }
}
```

93. 12_Advanced_Graphs/02_Min_Cost_to_Connect_All_Points/1584-min-cost-to-con

```
/*
Problem: LeetCode 1584 - Minimum Cost to Connect All Points

Key Idea:
The key idea is to treat the given points as nodes in a graph, where the edges represent the distance between points. To minimize the cost of connecting all points, we can use Kruskal's algorithm, a greedy algorithm for finding the minimum spanning tree (MST) of a graph.

Approach:
1. Create a vector to store all edges, where each edge is a tuple of (distance, point1, point2).
2. Sort the edges in ascending order based on distance.
3. Initialize a vector "parent" to keep track of the parent node for each point in the graph. Initially, each point is its own parent.
4. Initialize variables "min_cost" to keep track of the minimum cost and "num_connected" to keep track of the number of connected components (initially equal to the number of points).
5. Iterate through the sorted edges:
   - For each edge (distance, point1, point2):
     - Check if point1 and point2 belong to different connected components. If they do, merge the components by updating their parent pointers.
     - Add the distance to "min_cost."
     - Decrement "num_connected" by 1.
     - If "num_connected" becomes 1, all points are connected, and we can break from the loop.
6. Return "min_cost" as the minimum cost to connect all points.

Time Complexity:
O(E * log E), where E is the number of edges (combinations of points). Sorting the edges takes O(E * log E) time, and the iteration through edges also takes O(E * log E) time in the worst case.

Space Complexity:
O(N), where N is the number of points. This is used for the "parent" vector.
*/

impl Solution {
    pub fn min_cost_connect_points(points: Vec<Vec<i32>>) -> i32 {
        fn find_parent(point: usize, parent: &mut Vec<usize>) -> usize {
            if parent[point] != point {
                parent[point] = find_parent(parent[point], parent);
            }
            parent[point]
        }

        let n = points.len();
        let mut edges: Vec<(i32, usize, usize)> = Vec::new();

        for i in 0..n {
            for j in (i + 1)..n {
                let distance =
                    (points[i][0] - points[j][0]).abs() + (points[i][1] - points[j][1]).abs();
                edges.push((distance, i, j));
            }
        }

        edges.sort_by_key(|edge| edge.0);
        let mut parent: Vec<usize> = (0..n).collect();
        let mut min_cost = 0;
        let mut num_connected = n;

        for edge in edges {
            let parent1 = find_parent(edge.1, &mut parent);
            let parent2 = find_parent(edge.2, &mut parent);
```

```
            if parent1 != parent2 {
                parent[parent1] = parent2;
                min_cost += edge.0;
                num_connected -= 1;

                if num_connected == 1 {
                    break;
                }
            }
        }

        min_cost
    }
}

/*
impl Solution {
    pub fn min_cost_connect_points(points: Vec<Vec<i32>>) -> i32 {
        let points = points.into_iter().map(|p| (p[0], p[1])).collect::<Vec<_>>();
        let n = points.len();

        let mut connected = vec![false; n];
        let mut distances = vec![i32::MAX; n];

        let mut min_index = 0;
        let mut min_cost = 0;

        let mut total_cost = 0;

        for _ in 0..n {
            connected[min_index] = true;
            total_cost += min_cost;

            let current_point = points[min_index];
            min_cost = i32::MAX;

            for (idx, point) in points.iter().enumerate() {
                if !connected[idx] {
                            let distance = (current_point.0 - point.0).abs() + (current_point.1 -
point.1).abs();
                    distances[idx] = distances[idx].min(distance);

                    if distances[idx] < min_cost {
                        min_cost = distances[idx];
                        min_index = idx;
                    }
                }
            }
        }

        total_cost
    }
}
*/
```

94. 12_Advanced_Graphs/03_Network_Delay_Time/0743-network-delay-time.rs

```
/*
Problem: LeetCode 743 - Network Delay Time

Key Idea:
The key idea is to find the minimum time it takes for a signal to reach all nodes in a weighted directed graph. This problem can be solved using Dijkstra's algorithm, a greedy algorithm for finding the shortest path in a weighted graph.

Approach:
1. Create a vector "dist" to store the minimum time to reach each node from the source node (initialized with infinity for all nodes except the source).
2. Create a priority queue (min heap) to keep track of the next node to visit based on their estimated distances.
3. Start with the source node, set its distance to 0, and push it into the priority queue.
4. While the priority queue is not empty:
   - Pop the node with the smallest estimated distance from the priority queue.
   - If this node has been visited before, skip it.
   - For each neighbor of the current node:
     - Calculate the tentative distance to the neighbor through the current node.
     - If the tentative distance is smaller than the current distance stored in "dist," update the distance and push the neighbor into the priority queue.
5. After processing all nodes, check if any node has infinite distance. If so, it's not reachable, so return -1. Otherwise, return the maximum distance in "dist" as the answer.

Time Complexity:
O(N^2 + E * log E), where N is the number of nodes and E is the number of edges. In the worst case, Dijkstra's algorithm can have O(N^2) complexity if using a naive priority queue. Using a binary heap-based priority queue can optimize it to O(E * log E).

Space Complexity:
O(N + E), where N is the number of nodes for the "dist" vector, and E is the number of edges for the adjacency list.
*/

use std::collections::{BinaryHeap, HashMap};

impl Solution {
    pub fn network_delay_time(times: Vec<Vec<i32>>, n: i32, k: i32) -> i32 {
        // Create an adjacency list to represent the graph.
        let mut graph: HashMap<i32, Vec<(i32, i32)>> = HashMap::new();
        for time in &times {
            let (u, v, w) = (time[0], time[1], time[2]);
            graph.entry(u).or_insert(Vec::new()).push((v, w));
        }

        // Initialize distances with infinity, except for the source.
        let mut dist: Vec<i32> = vec![std::i32::MAX; n as usize];
        dist[k as usize - 1] = 0;

        // Create a min heap to store (distance, node) pairs.
        let mut min_heap: BinaryHeap<(i32, i32)> = BinaryHeap::new();
        min_heap.push((0, k));

        while let Some((d, u)) = min_heap.pop() {
            if d > dist[u as usize - 1] {
                continue;
            }
            if let Some(neighbors) = graph.get(&u) {
                for (v, w) in neighbors {
                    let new_dist = dist[u as usize - 1] + w;
```

```
                if new_dist < dist[*v as usize - 1] {
                    dist[*v as usize - 1] = new_dist;
                    min_heap.push((new_dist, *v));
                }
            }
        }
    }

    // Check if any node is unreachable.
    if dist.iter().any(|&d| d == std::i32::MAX) {
        return -1;
    }

    // Return the maximum distance in "dist."
    *dist.iter().max().unwrap()
  }
}
```

95. 12_Advanced_Graphs/04_Swim_in_Rising_Water/0778-swim-in-rising-water.rs

```
/*
Problem: LeetCode 778 - Swim in Rising Water

Key Idea:
The key idea is to perform a binary search on the time it takes to reach a point (i, j) in the grid from the starting point (0, 0). Then, we can use depth-first search (DFS) to check if it's possible to reach the destination point (n-1, n-1) within that time limit.

Approach:
1. Initialize the lower bound as the minimum value in the grid and the upper bound as the maximum value in the grid.
2. Perform a binary search on the time, searching for the minimum time required to reach the destination point (n-1, n-1) from the starting point (0, 0).
3. In each binary search iteration:
   - Calculate the mid-time (current guess) as the average of the lower and upper bounds.
   - Create a visited array of size n x n to keep track of visited cells.
   - Start DFS from the starting point (0, 0) and explore cells while considering two conditions:
     - The cell is within bounds (0 <= i < n and 0 <= j < n).
     - The cell value is less than or equal to the current mid-time.
   - If DFS reaches the destination point (n-1, n-1), update the upper bound with the mid-time.
   - Otherwise, update the lower bound.
4. Repeat the binary search until the lower bound is less than the upper bound.
5. Return the final lower bound as the minimum time required to reach the destination.

Time Complexity:
O(n^2 * log(maximum value in the grid)), where n is the size of the grid.

Space Complexity:
O(n^2) for the visited array.
*/

impl Solution {
    pub fn swim_in_water(grid: Vec<Vec<i32>>) -> i32 {
        let n = grid.len();
        let mut low = grid[0][0];
        let mut high = (n * n) as i32 - 1;

        while low < high {
            let mid = low + (high - low) / 2;
            let mut visited = vec![vec![false; n]; n];

            if Self::dfs(&grid, &mut visited, 0, 0, mid) {
                high = mid;
            } else {
                low = mid + 1;
            }
        }

        low
    }

    fn dfs(
        grid: &Vec<Vec<i32>>,
        visited: &mut Vec<Vec<bool>>,
        i: usize,
        j: usize,
        time: i32,
    ) -> bool {
        if i == grid.len() - 1 && j == grid[0].len() - 1 {
            return true;
```

```
        }

        visited[i][j] = true;
        let directions = [(0, 1), (0, -1), (1, 0), (-1, 0)];
        let n = grid.len();

        for (dx, dy) in &directions {
            let ni = i as i32 + dx;
            let nj = j as i32 + dy;

            if ni >= 0
                && ni < n as i32
                && nj >= 0
                && nj < n as i32
                && !visited[ni as usize][nj as usize]
                && grid[ni as usize][nj as usize] <= time
            {
                if Self::dfs(grid, visited, ni as usize, nj as usize, time) {
                    return true;
                }
            }
        }

        false
    }
}
```

96. 12_Advanced_Graphs/05_Alien_Dictionary/0269-alien-dictionary.rs

97. 12_Advanced_Graphs/06_Cheapest_Flights_Within_K_Stops/0787-cheapest-fligh

```
/*
Problem: LeetCode 787 - Cheapest Flights Within K Stops

Key Idea:
The key idea is to use a modified form of Dijkstra's algorithm to find the cheapest price from the source to the destination while considering the maximum number of stops.

Approach:
1. Create an adjacency list representation of the graph where each node is an airport, and each edge is a flight with its cost. You can use a vector of vectors for this purpose. The index of the outer vector represents the airport, and each inner vector contains pairs (to, price) representing flights from that airport.
2. Initialize a vector stops to keep track of the minimum number of stops required to reach each airport from the source
3. Initialize all elements to i32::MAX initially, except for the source airport, which should be set to 0.
4. Initialize a priority queue (min-heap) min_heap to store tuples (total_price, airport, stops) in reverse order of total_price (i.e., lowest total_price comes first).
5. Push a tuple (0, src, 0) onto the min_heap, where 0 is the total price, src is the source airport, and 0 is the number of stops.
6. Explore Nodes in Priority Queue:
  - While the min_heap is not empty:
      - Pop the tuple (total_price, from, steps) from the min_heap in reverse order.
          - If steps exceeds k or if from is equal to dst, return total_price as the answer.
          - Otherwise, update stops[from] with steps if steps is smaller than the current value.
          - For each flight (to, price) from from to other airports:
              - Calculate the new_total_price as total_price + price.
              - Push the tuple (new_total_price, to, steps + 1) onto the min_heap.
7. If no valid path is found after exploring all nodes, return -1.

Time Complexity:
O(E + V * log(V)), where E is the number of flights (edges) and V is the number of airports (nodes). The use of a priority queue ensures that we explore nodes in order of increasing total_price.

Space Complexity:
O(V), for stops vector and the min_heap.
*/

use std::cmp::Reverse;
use std::collections::BinaryHeap;

impl Solution {
    pub fn find_cheapest_price(n: i32, flights: Vec<Vec<i32>>, src: i32, dst: i32, k: i32) -> i32 {
        let mut graph = vec![vec![]; n as usize];
        for flight in &flights {
            let (from, to, price) = (flight[0] as usize, flight[1] as usize, flight[2]);
            graph[from].push((to, price));
        }

        let mut stops = vec![i32::MAX; n as usize];

        let mut min_heap = BinaryHeap::new();
        min_heap.push(Reverse((0, src as usize, 0)));

        while !min_heap.is_empty() {
            let Reverse((total_price, from, steps)) = min_heap.pop().unwrap();

            if steps > stops[from] || steps > k + 1 {
```

```
                continue;
            }
            if from as i32 == dst {
                return total_price;
            }

            stops[from] = steps;
            for &(to, price) in &graph[from] {
                min_heap.push(Reverse((total_price + price, to, steps + 1)));
            }
        }

        -1
    }
}
```

98. 13_1-D_Dynamic_Programming/01_Climbing_Stairs/0070-climbing-stairs.rs

```
/*
Problem: LeetCode 70 - Climbing Stairs

Key Idea:
The key idea is to realize that to reach the i-th step, you can either:
1. Take one step from the (i-1)-th step if you had already climbed (i-1) steps.
2. Take two steps from the (i-2)-th step if you had already climbed (i-2) steps.

Approach:
1. Initialize two variables, `prev` and `curr`, to represent the number of ways to reach the (i-1)-th and i-th step, respectively.
2. Start iterating from step 3 to n (inclusive), where n is the total number of steps:
    - Calculate the number of ways to reach the i-th step as `curr = prev + curr`, where `prev` represents the number of ways to reach the (i-1)-th step, and `curr` represents the number of ways to reach the i-th step.
   - Update `prev` to the previous value of `curr`.
3. Return the value of `curr` as the number of ways to reach the n-th step.

Time Complexity:
O(n), where n is the total number of steps. We perform a single pass through the steps.

Space Complexity:
O(1), as we use only two variables (`prev` and `curr`) to store intermediate results.
*/

impl Solution {
    pub fn climb_stairs(n: i32) -> i32 {
        if n <= 2 {
            return n;
        }

        let (mut prev, mut curr) = (1, 2);

        for _ in 3..=n {
            let temp = curr;
            curr = prev + curr;
            prev = temp;
        }

        curr
    }
}
```

99. 13_1-D_Dynamic_Programming/02_Min_Cost_Climbing_Stairs/0746-min-cost-clin

```
/*
Problem: LeetCode 746 - Min Cost Climbing Stairs

Key Idea:
The key idea is to use dynamic programming to find the minimum cost to reach each step.

Approach:
1. Initialize an array `dp` of size n+1 to store the minimum cost to reach each step, where n is the length of the `cost` array.
2. Initialize `dp[0]` and `dp[1]` to be 0, as there is no cost to reach the first two steps.
3. Iterate from step 2 to n:
    - Calculate `dp[i]` as the minimum of `dp[i-1] + cost[i-1]` and `dp[i-2] + cost[i-2]`. This represents the minimum cost to reach the i-th step.
4. Return `dp[n]`, which represents the minimum cost to reach the top floor.

Time Complexity:
O(n), where n is the length of the `cost` array. We perform a single pass through the steps.

Space Complexity:
O(n), as we use an array `dp` of size n+1 to store intermediate results.
*/

impl Solution {
    pub fn min_cost_climbing_stairs(cost: Vec<i32>) -> i32 {
        let n = cost.len();
        if n <= 2 {
            return cost.iter().min().unwrap_or(&0).to_owned();
        }

        let mut dp = vec![0; n + 1];

        for i in 2..=n {
            dp[i] = std::cmp::min(dp[i - 1] + cost[i - 1], dp[i - 2] + cost[i - 2]);
        }

        dp[n]
    }
}
```

100. 13_1-D_Dynamic_Programming/03_House_Robber/0198-house-robber.rs

```
/*
Problem: LeetCode 198 - House Robber

Key Idea:
The key idea is to use dynamic programming to find the maximum amount of money that can be robbed without alerting the police.

Approach:
1. Initialize two variables, `prev` and `curr`, to represent the maximum amount of money that can be robbed from the previous two houses and the current house, respectively.
2. Start iterating from the first house to the last house:
    - Calculate the maximum amount that can be robbed from the current house as `max(prev + current_house_value, curr)`.
   - Update `prev` and `curr` accordingly.
3. Return `curr` as the maximum amount of money that can be robbed without alerting the police.

Time Complexity:
O(n), where n is the number of houses. We perform a single pass through the houses.

Space Complexity:
O(1), as we use only two variables (`prev` and `curr`) to store intermediate results.
*/

impl Solution {
    pub fn rob(nums: Vec<i32>) -> i32 {
        let (mut prev, mut curr) = (0, 0);

        for &num in nums.iter() {
            let temp = curr;
            curr = curr.max(prev + num);
            prev = temp;
        }

        curr
    }
}
```

101. 13_1-D_Dynamic_Programming/04_House_Robber_II/0213-house-robber-ii.rs

```
/*
Problem: LeetCode 213 - House Robber II

Key Idea:
The key idea is to break the problem into two cases:
1. Rob the first house and don't rob the last house.
2. Don't rob the first house and rob the last house.

Approach:
1. If the input `nums` is empty or has only one house, return the maximum value in `nums`.
2. Create a helper function `rob_range(nums: &[i32])` to calculate the maximum amount that can be robbed within a given range.
    - Initialize two variables, `prev` and `curr`, to keep track of the maximum amounts for the previous house and the current house, respectively.
    - Iterate through the houses in the given range and calculate the maximum amount for the current house as `max(prev, curr)`.
   - Update `prev` and `curr` accordingly.
   - Return `curr` as the maximum amount that can be robbed within the given range.
3. Calculate the maximum amount for the two cases:
      - Case 1: Rob the first house and don't rob the last house. Calculate this as `rob_range(&nums[1..])`, including all houses except the last one.
      - Case 2: Don't rob the first house and rob the last house. Calculate this as `rob_range(&nums[..nums.len() - 1])`, including all houses except the first one.
4. Return the maximum value between the two cases as the final result.

Time Complexity:
O(n), where n is the number of houses. We perform two passes through the `nums` array.

Space Complexity:
O(1), as we use a constant amount of extra space to store `prev` and `curr`.
*/

impl Solution {
    pub fn rob(nums: Vec<i32>) -> i32 {
        let n = nums.len();

        if n == 0 {
            return 0;
        } else if n == 1 {
            return nums[0];
        }

        let max_case1 = Self::rob_range(&nums[1..]);
        let max_case2 = Self::rob_range(&nums[..n - 1]);

        max_case1.max(max_case2)
    }

    fn rob_range(nums: &[i32]) -> i32 {
        let (mut prev, mut curr) = (0, 0);

        for &num in nums {
            let temp = curr;
            curr = curr.max(prev + num);
            prev = temp;
        }

        curr
    }
}
```

102. 13_1-D_Dynamic_Programming/05_Longest_Palindromic_Substring/0005-longes

```
/*
Problem: LeetCode 5 - Longest Palindromic Substring

Key Idea:
The key idea is to use dynamic programming to find the longest palindromic substring.

Approach:
1. Initialize a 2D boolean array `dp` where `dp[i][j]` represents whether the substring from index `i` to `j` is a palindrome.
2. Initialize a variable `start` to keep track of the starting index of the longest palindromic substring found so far and `max_len` to keep track of its length.
3. Iterate through the string characters:
   - For each character at index `i`, iterate from `i` to the beginning of the string:
      - If `s[i]` is equal to `s[j]` and the substring between `i` and `j` is either empty or a palindrome (`i+1 > j-1` or `dp[i+1][j-1]` is true), set `dp[i][j]` to true.
      - If `dp[i][j]` is true and the length of the substring is greater than `max_len`, update `max_len` and `start` to the current values of `i` and `j`.
4. Return the longest palindromic substring by slicing the string from index `start` to `start + max_len`.

Time Complexity:
O(n^2), where n is the length of the input string `s`. We fill in the `dp` array for all possible substrings.

Space Complexity:
O(n^2), as we use a 2D array `dp` of size n x n.
*/

impl Solution {
    pub fn longest_palindrome(s: String) -> String {
        let s_chars: Vec<char> = s.chars().collect();
        let n = s_chars.len();
        if n <= 1 {
            return s;
        }

        let mut dp = vec![vec![false; n]; n];
        let (mut start, mut max_len) = (0, 1);

        // All substrings of length 1 are palindromes.
        for i in 0..n {
            dp[i][i] = true;
        }

        // Check for palindromes of length 2.
        for i in 0..n - 1 {
            if s_chars[i] == s_chars[i + 1] {
                dp[i][i + 1] = true;
                start = i;
                max_len = 2;
            }
        }

        // Check for palindromes of length >= 3.
        for len in 3..=n {
            for i in 0..=n - len {
                let j = i + len - 1;

                if s_chars[i] == s_chars[j] && dp[i + 1][j - 1] {
                    dp[i][j] = true;
```

```
                    if len > max_len {
                        start = i;
                        max_len = len;
                    }
                }
            }
        }

        s_chars[start..start + max_len].iter().collect()
    }
}

/*
// Expanding around the center approach

impl Solution {
    pub fn longest_palindrome(s: String) -> String {
        let bytes = s.as_bytes();
        let n = bytes.len();
        let (mut max_start, mut max_end) = (0, 0);

        for i in 0..n {
            let (mut left, mut right) = (i, i);

            // Expand the palindrome around the current character.
            while right + 1 < n && bytes[left] == bytes[right + 1] {
                right += 1;
            }

            // Expand the palindrome while possible.
            while left > 0 && right + 1 < n && bytes[left - 1] == bytes[right + 1] {
                left -= 1;
                right += 1;
            }

            // Check if the current palindrome is longer.
            if right - left > max_end - max_start {
                max_start = left;
                max_end = right;
            }
        }

        String::from(&s[max_start..=max_end])
    }
}
*/
```

103. 13_1-D_Dynamic_Programming/06_Palindromic_Substrings/0647-palindromic-su

```
/*
Problem: LeetCode 647 - Palindromic Substrings

Key Idea:
The key idea is to use dynamic programming to count palindromic substrings.

Approach:
1. Initialize a variable `count` to keep track of the total number of palindromic substrings found. Initially, set it to 0.
2. Create a 2D boolean array `is_palindrome` of size n x n, where n is the length of the input string.
   - `is_palindrome[i][j]` will be true if the substring from index i to j is a palindrome.
3. Iterate through the input string:
   - For each character at index `i`, set `is_palindrome[i][i]` to true, as single characters are palindromes.
   - For each pair of adjacent characters at indices `i` and `i+1`, check if they are the same. If they are, set `is_palindrome[i][i+1]` to true, indicating a palindrome of length 2.
4. Iterate through the string from length 3 to n:
    - For each substring of length `len`, starting from index `i`, check if the characters at indices `i` and `i+len-1` are the same.
   - If they are the same and the substring between them (`i+1` to `i+len-2`) is also a palindrome (`is_palindrome[i+1][i+len-2]` is true), then set `is_palindrome[i][i+len-1]` to true, indicating a palindrome of length `len`.
5. During the iteration in step 4, whenever `is_palindrome[i][i+len-1]` is set to true, increment `count` by 1.
6. After all iterations, `count` will represent the total number of palindromic substrings.
7. Return `count` as the result.

Time Complexity:
O(n^2), where n is the length of the input string. We fill in the `is_palindrome` table in a nested loop.

Space Complexity:
O(n^2), as we use a 2D boolean array of size n x n to store the palindrome information.
*/

impl Solution {
    pub fn count_substrings(s: String) -> i32 {
        let n = s.len();
        let s = s.chars().collect::<Vec<_>>();
        let mut is_palindrome = vec![vec![false; n]; n];
        let mut count = 0;

        // Single characters are palindromes
        for i in 0..n {
            is_palindrome[i][i] = true;
            count += 1;
        }

        // Check for palindromes of length 2
        for i in 0..n - 1 {
            if s[i] == s[i + 1] {
                is_palindrome[i][i + 1] = true;
                count += 1;
            }
        }

        // Check for palindromes of length >= 3
        for len in 3..=n {
            for i in 0..=n - len {
```

```
                let j = i + len - 1;
                if s[i] == s[j] && is_palindrome[i + 1][j - 1] {
                    is_palindrome[i][j] = true;
                    count += 1;
                }
            }
        }

        count
    }
}

/*
// Alternate solution

impl Solution {
    pub fn count_substrings(s: String) -> i32 {
        let s = s.chars().collect::<Vec<char>>();
        let length = s.len() as i32;
        let mut count = 0;

        for i in 0..length {
            // Odd-length palindromes
            let (mut left, mut right) = (i, i);
            while left >= 0 && right < length && s[left as usize] == s[right as usize] {
                count += 1;
                left -= 1;
                right += 1;
            }

            // Even-length palindromes
            let (mut left, mut right) = (i, i + 1);
            while left >= 0 && right < length && s[left as usize] == s[right as usize] {
                count += 1;
                left -= 1;
                right += 1;
            }
        }

        count
    }
}
*/
```

104. 13_1-D_Dynamic_Programming/07_Decode_Ways/0091-decode-ways.rs

```
/*
Problem: LeetCode 91 - Decode Ways

Key Idea:
The key idea is to use dynamic programming to count the number of ways to decode a given string.

Approach:
1. Initialize a vector `dp` of size n+1, where n is the length of the input string.
   - `dp[i]` will represent the number of ways to decode the substring from index i to n-1.
2. Set `dp[n]` to 1, as there is one way to decode an empty string.
3. Iterate through the characters of the input string in reverse order:
    - For each character at index `i`, calculate the number of ways to decode the substring starting from index `i`:
     - If the current character is '0', set `dp[i]` to 0 because '0' cannot be decoded as a single character.
      - Otherwise, set `dp[i]` to `dp[i+1]` because the current character can be decoded as a single character.
     - If the current character and the next character form a valid two-digit number (between '10' and '26'), add `dp[i+2]` to `dp[i]` because the two characters can be decoded as a single unit.
4. Finally, `dp[0]` will contain the number of ways to decode the entire string.
5. Return `dp[0]` as the result.

Time Complexity:
O(n), where n is the length of the input string. We perform a single pass through the string.

Space Complexity:
O(n), as we use a vector of size n+1 to store the dynamic programming values.
*/

impl Solution {
    pub fn num_decodings(s: String) -> i32 {
        let s = s.as_bytes();
        let n = s.len();
        let mut dp = vec![0; n + 1];
        dp[n] = 1;

        for i in (0..n).rev() {
            if s[i] == b'0' {
                dp[i] = 0;
            } else {
                dp[i] = dp[i + 1];
                if i + 1 < n && (s[i] == b'1' || (s[i] == b'2' && s[i + 1] <= b'6')) {
                    dp[i] += dp[i + 2];
                }
            }
        }

        dp[0]
    }
}
```

105. 13_1-D_Dynamic_Programming/08_Coin_Change/0322-coin-change.rs

```
/*
Problem: LeetCode 322 - Coin Change

Key Idea:
The key idea is to use dynamic programming to find the minimum number of coins required to make a certain amount of money.

Approach:
1. Initialize a vector `dp` of size `amount + 1` to store the minimum number of coins required to make each amount from 0 to `amount`.
   - `dp[i]` will represent the minimum number of coins required to make the amount `i`.
2. Set `dp[0]` to 0, as it takes 0 coins to make an amount of 0.
3. Iterate through the amounts from 1 to `amount`:
   - For each amount `i`, initialize `dp[i]` to a value greater than `amount` (e.g., `amount + 1`) to represent an impossible scenario initially.
   - For each coin value `coin` in the list of coins, check if `i - coin` is a valid amount (not negative) and if `dp[i - coin] + 1` is smaller than the current `dp[i]`. If so, update `dp[i]` to `dp[i - coin] + 1`.
4. After the iteration, `dp[amount]` will contain the minimum number of coins required to make the given amount.
5. If `dp[amount]` is still greater than `amount` (the amount is not possible to make), return -1; otherwise, return `dp[amount]` as the result.

Time Complexity:
O(amount * n), where `amount` is the target amount and `n` is the number of coins. We fill in the `dp` table with a nested loop.

Space Complexity:
O(amount), as we use a vector of size `amount + 1` to store the dynamic programming values.
*/

impl Solution {
    pub fn coin_change(coins: Vec<i32>, amount: i32) -> i32 {
        let amount = amount as usize;
        let mut dp = vec![amount + 1; amount + 1];
        dp[0] = 0;

        for i in 1..=amount {
            for &coin in &coins {
                if i >= coin as usize && dp[i - coin as usize] + 1 < dp[i] {
                    dp[i] = dp[i - coin as usize] + 1;
                }
            }
        }

        if dp[amount] > amount {
            -1
        } else {
            dp[amount] as i32
        }
    }
}
```

106. 13_1-D_Dynamic_Programming/09_Maximum_Product_Subarray/0152-maximum

```
/*
Problem: LeetCode 152 - Maximum Product Subarray

Key Idea:
The key idea is to maintain two variables: `max_product` and `min_product`. These variables keep track of the maximum and minimum product subarrays ending at the current index. We need to consider the minimum product because multiplying a large negative number by a negative number can result in a large positive product.

Approach:
1. Initialize three variables: `max_product`, `min_product`, and `result`, all initially set to the first element of the input array `nums[0]`.
2. Iterate through the array `nums` from the second element to the end.
3. For each element `nums[i]`, update `max_product` and `min_product` as follows:
   - Calculate the product of `max_product` with `nums[i]` and `min_product` with `nums[i]`.
   - Update `max_product` by taking the maximum of these two products and `nums[i]` itself.
   - Update `min_product` by taking the minimum of these two products and `nums[i]` itself.
   - Update `result` by taking the maximum of `result` and `max_product`.
4. After the iteration, `result` will contain the maximum product subarray.
5. Return `result` as the result.

Time Complexity:
O(n), where n is the length of the input array `nums`. We perform a single pass through the array.

Space Complexity:
O(1), as we use a constant amount of extra space to store variables.
*/

impl Solution {
    pub fn max_product(nums: Vec<i32>) -> i32 {
        let n = nums.len();
        let mut max_product = nums[0];
        let mut min_product = nums[0];
        let mut result = nums[0];

        for i in 1..n {
            let temp_max = max_product;
            max_product = nums[i]
                .max(max_product * nums[i])
                .max(min_product * nums[i]);
            min_product = nums[i].min(temp_max * nums[i]).min(min_product * nums[i]);
            result = result.max(max_product);
        }

        result
    }
}
```

107. 13_1-D_Dynamic_Programming/10_Word_Break/0139-word-break.rs

```
/*
Problem: LeetCode 139 - Word Break

Key Idea:
The key idea is to use dynamic programming to determine if a given string can be segmented into words from a dictionary.

Approach:
1. Initialize a boolean vector `dp` of size `n + 1`, where `n` is the length of the input string `s`.
    - `dp[i]` will represent whether the substring from index 0 to `i-1` can be segmented into words from the dictionary.
2. Set `dp[0]` to `true` because an empty string can always be segmented.
3. Iterate through the input string `s` from the first character to the last character:
    - For each index `i`, iterate through the dictionary words and check if a word `word` can be found such that:
     - `dp[i - len]` is `true`, where `len` is the length of `word`.
     - The substring from index `i - len` to `i-1` (inclusive) is equal to `word`.
   - If such a word is found, set `dp[i]` to `true`.
4. After the iteration, `dp[n]` will be `true` if and only if the entire string `s` can be segmented into words from the dictionary.
5. Return `dp[n]` as the result.

Time Complexity:
O(n^2), where `n` is the length of the input string `s`. In the worst case, we have nested loops over `i` and `j` such that `i * j <= n`.

Space Complexity:
O(n), as we use a boolean vector of size `n + 1` to store the dynamic programming values.
*/

impl Solution {
    pub fn word_break(s: String, word_dict: Vec<String>) -> bool {
        let n = s.len();
        let mut dp = vec![false; n + 1];
        dp[0] = true;

        let word_set: std::collections::HashSet<String> = word_dict.into_iter().collect();

        for i in 1..=n {
            for j in 0..i {
                if dp[j] && word_set.contains(&s[j..i]) {
                    dp[i] = true;
                    break;
                }
            }
        }

        dp[n]
    }
}
```

108. 13_1-D_Dynamic_Programming/11_Longest_Increasing_Subsequence/0300-long

```
/*
Problem: LeetCode 300 - Longest Increasing Subsequence

Key Idea:
The key idea is to use dynamic programming to find the length of the longest increasing subsequence in the given array.

Approach:
1. Initialize a vector `dp` of the same length as the input vector `nums`, all initially set to 1.
   - `dp[i]` will represent the length of the longest increasing subsequence ending at index `i`.
   - Initialize it to 1 because a single element is always a valid subsequence.
2. Iterate through the input vector `nums` from the second element to the end.
3. For each element `nums[i]`, iterate through the elements from the start to index `i - 1`:
    - For each element `nums[j]`, if `nums[i]` is greater than `nums[j]`, update `dp[i]` to the maximum of `dp[i]` and `dp[j] + 1`.
     - This means that if we can extend an increasing subsequence ending at index `j` with `nums[i]`, we update the length of the increasing subsequence ending at index `i`.
4. After the iteration, the maximum value in the `dp` vector will be the length of the longest increasing subsequence.
5. Return this maximum value as the result.

Time Complexity:
O(n^2), where `n` is the length of the input vector `nums`. We have nested loops for each element.

Space Complexity:
O(n), as we use a vector of size `n` to store the dynamic programming values.
*/

impl Solution {
    pub fn length_of_lis(nums: Vec<i32>) -> i32 {
        let n = nums.len();
        let mut dp = vec![1; n];

        for i in 1..n {
            for j in 0..i {
                if nums[i] > nums[j] {
                    dp[i] = dp[i].max(dp[j] + 1);
                }
            }
        }

        dp.iter().cloned().max().unwrap_or(0)
    }
}
```

109. 13_1-D_Dynamic_Programming/12_Partition_Equal_Subset_Sum/0416-partition-

```
/*
Problem: LeetCode 416 - Partition Equal Subset Sum

Key Idea:
The key idea is to use dynamic programming to determine if it's possible to partition the input array into two subsets with equal sums.

Approach:
1. Calculate the sum of all elements in the input array `nums` and store it in a variable `total_sum`.
2. If the total sum is odd, it's impossible to partition the array into two subsets with equal sums. Return `false` in this case.
3. Otherwise, initialize a boolean vector `dp` of size `total_sum / 2 + 1`, where `dp[i]` represents whether it's possible to form a subset with a sum of `i`.
   - Initialize `dp[0]` to `true` because an empty subset has a sum of 0.
4. Iterate through the elements of the input array `nums`:
   - For each element `num`, iterate through the `dp` vector in reverse order from `total_sum / 2` down to `num`.
   - Update `dp[i]` to `true` if either `dp[i]` is already `true` or `dp[i - num]` is `true`. This means we can form a subset with a sum of `i` by either not including the current `num` or including it.
5. After the iteration, if `dp[total_sum / 2]` is `true`, it's possible to partition the array into two subsets with equal sums; otherwise, it's not possible.
6. Return `dp[total_sum / 2]` as the result.

Time Complexity:
O(n * sum), where `n` is the number of elements in the input array `nums`, and `sum` is the sum of all elements in the array. We fill in the `dp` table with a nested loop.

Space Complexity:
O(sum / 2), as we use a boolean vector of size `total_sum / 2 + 1` to store the dynamic programming values.
*/

impl Solution {
    pub fn can_partition(nums: Vec<i32>) -> bool {
        let total_sum: i32 = nums.iter().sum();

        if total_sum % 2 != 0 {
            return false;
        }

        let target_sum = (total_sum / 2) as usize;
        let mut dp = vec![false; target_sum + 1];
        dp[0] = true;

        for &num in nums.iter() {
            for i in (num as usize..=target_sum).rev() {
                dp[i] = dp[i] || dp[i - num as usize];
            }
        }

        dp[target_sum]
    }
}
```

110. 14_2-D_Dynamic_Programming/01_Unique_Paths/0062-unique-paths.rs

```
/*
Problem: LeetCode 62 - Unique Paths

Key Idea:
The key idea is to use dynamic programming to find the total number of unique paths from the top-left corner to the bottom-right corner of a grid. The intuition is that at any given cell, the number of unique paths to reach it is the sum of the paths from the cell above and the cell to the left, as you can only move down or right in the grid. By applying this principle iteratively and filling a 2D array, you can efficiently compute the final count of unique paths.

Approach:
1. Initialize a 2D vector `dp` of size `m x n`, where `dp[i][j]` represents the number of unique paths to reach cell `(i, j)` from the top-left corner `(0, 0)`.
2. Initialize the first row and the first column of `dp` to 1 because there is only one way to reach any cell in the first row or the first column (by moving right or down).
3. Iterate through the grid from row 1 to `m-1` and column 1 to `n-1`.
4. For each cell `(i, j)`, update `dp[i][j]` as the sum of the number of unique paths from the cell above `(i-1, j)` and the cell to the left `(i, j-1)`.
5. After the iteration, `dp[m-1][n-1]` will contain the number of unique paths to reach the bottom-right corner.
6. Return `dp[m-1][n-1]` as the result.

Time Complexity:
O(m * n), where `m` is the number of rows and `n` is the number of columns in the grid. We fill in the `dp` table with a nested loop.

Space Complexity:
O(m * n), as we use a 2D vector of size `m x n` to store the dynamic programming values.
*/

impl Solution {
    pub fn unique_paths(m: i32, n: i32) -> i32 {
        let m = m as usize;
        let n = n as usize;

        let mut dp = vec![vec![1; n]; m];

        for i in 1..m {
            for j in 1..n {
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
            }
        }

        dp[m - 1][n - 1]
    }
}
```

111. 14_2-D_Dynamic_Programming/02_Longest_Common_Subsequence/1143-longe

```
/*
Problem: LeetCode 1143 - Longest Common Subsequence

Key Idea:
The key idea is to use dynamic programming to find the longest common subsequence (LCS) between
two strings.

Approach:
1. Initialize a 2D vector `dp` of size `(text1.len() + 1) x (text2.len() + 1)`, where `dp[i][j]`
represents the length of the LCS of `text1[0..i]` and `text2[0..j]`.
2. Initialize the first row and the first column of `dp` to 0 because the LCS of any string with
an empty string is 0.
3. Iterate through the characters of `text1` and `text2` using nested loops from 1 to
`text1.len()` and 1 to `text2.len()`.
4. For each pair of characters `text1[i-1]` and `text2[j-1]`:
   - If they are equal, set `dp[i][j]` to `dp[i-1][j-1] + 1` because we can extend the LCS.
   - If they are not equal, set `dp[i][j]` to the maximum of `dp[i-1][j]` and `dp[i][j-1]` because
we take the maximum length LCS without the current characters.
5. After the iteration, `dp[text1.len()][text2.len()]` will contain the length of the LCS.
6. Return `dp[text1.len()][text2.len()]` as the result.

Time Complexity:
O(m * n), where `m` is the length of `text1` and `n` is the length of `text2`. We fill in the `dp`
table with a nested loop.

Space Complexity:
O(m * n), as we use a 2D vector of size `(text1.len() + 1) x (text2.len() + 1)` to store the
dynamic programming values.
*/

impl Solution {
    pub fn longest_common_subsequence(text1: String, text2: String) -> i32 {
        let m = text1.len();
        let n = text2.len();

        let mut dp = vec![vec![0; n + 1]; m + 1];

        for i in 1..=m {
            for j in 1..=n {
                let char1 = text1.chars().nth(i - 1).unwrap();
                let char2 = text2.chars().nth(j - 1).unwrap();

                if char1 == char2 {
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                } else {
                    dp[i][j] = dp[i - 1][j].max(dp[i][j - 1]);
                }
            }
        }

        dp[m][n]
    }
}

/*
// Faster solution

impl Solution {
    pub fn longest_common_subsequence(text1: String, text2: String) -> i32 {
        let text1_bytes = text1.as_bytes();
```

```
        let text2_bytes = text2.as_bytes();
        let (m, n) = (text1_bytes.len(), text2_bytes.len());

        let mut dp = vec![vec![0; n + 1]; m + 1];

        for i in 1..=m {
            for j in 1..=n {
                dp[i][j] = if text1_bytes[i - 1] == text2_bytes[j - 1] {
                    dp[i - 1][j - 1] + 1
                } else {
                    std::cmp::max(dp[i - 1][j], dp[i][j - 1])
                };
            }
        }

        dp[m][n]
    }
}
*/
```

112. 14_2-D_Dynamic_Programming/03_Best_Time_to_Buy_And_Sell_Stock_With_C

```
/*
Problem: LeetCode 309 - Best Time to Buy and Sell Stock with Cooldown

Key Idea:
The key idea is to use dynamic programming to find the maximum profit that can be obtained by buying and selling a stock with a cooldown period.

Approach:
1. Initialize three vectors: `buy`, `sell`, and `cooldown`, all of size `n`, where `n` is the length of the input prices vector.
   - `buy[i]` represents the maximum profit that can be obtained by buying a stock on day `i`.
   - `sell[i]` represents the maximum profit that can be obtained by selling a stock on day `i`.
   - `cooldown[i]` represents the maximum profit on day `i` with a cooldown (no stock held).
2. Initialize `buy[0]` as the negative of the first price, indicating buying the stock on the first day.
   - Initialize `sell[0]` and `cooldown[0]` as 0 because there are no transactions yet.
3. Iterate through the prices vector from day 1 to `n-1`.
     - Update `buy[i]` as the maximum of `buy[i-1]` (not buying a stock on day `i`) and `cooldown[i-1] - prices[i]` (buying a stock on day `i` after a cooldown).
   - Update `sell[i]` as the maximum of `sell[i-1]` (not selling a stock on day `i`) and `buy[i-1] + prices[i]` (selling a stock on day `i`).
     - Update `cooldown[i]` as the maximum of `cooldown[i-1]` (no transaction on day `i`) and `sell[i-1]` (selling a stock on day `i-1` and having a cooldown on day `i`).
4. The maximum profit can be obtained by either holding no stock (`cooldown[n-1]`) or selling the stock on the last day (`sell[n-1]`).
5. Return the maximum of `cooldown[n-1]` and `sell[n-1]` as the result.

Time Complexity:
O(n), where `n` is the length of the input prices vector. We iterate through the prices vector once.

Space Complexity:
O(n), as we use three vectors of size `n` to store the dynamic programming values.
*/

impl Solution {
    pub fn max_profit(prices: Vec<i32>) -> i32 {
        let n = prices.len();
        if n <= 1 {
            return 0;
        }

        let mut buy = vec![0; n];
        let mut sell = vec![0; n];
        let mut cooldown = vec![0; n];

        buy[0] = -prices[0];

        for i in 1..n {
            cooldown[i] = cooldown[i - 1].max(sell[i - 1]);
            buy[i] = buy[i - 1].max(cooldown[i - 1] - prices[i]);
            sell[i] = sell[i - 1].max(buy[i - 1] + prices[i]);
        }

        cooldown[n - 1].max(sell[n - 1])
    }
}
```

113. 14_2-D_Dynamic_Programming/04_Coin_Change_II/0518-coin-change-ii.rs

```
/*
Problem: LeetCode 518 - Coin Change II

Key Idea:
The key idea is to use dynamic programming to find the number of combinations to make a certain amount of money using a given set of coin denominations.

Approach:
1. Initialize a vector `dp` of size `amount + 1`, where `dp[i]` represents the number of combinations to make amount `i`.
2. Initialize `dp[0]` to 1 because there is one way to make amount 0 (by using no coins).
3. Iterate through the coin denominations:
   - For each coin denomination `coin`, iterate through the `dp` vector from `coin` to `amount`.
    - For each index `i`, update `dp[i]` by adding `dp[i - coin]`. This represents the number of combinations to make amount `i` by including the current coin denomination.
4. After the iteration, `dp[amount]` will contain the number of combinations to make the given amount using the given coins.
5. Return `dp[amount]` as the result.

Time Complexity:
O(amount * n), where `amount` is the target amount and `n` is the number of coin denominations. We fill in the `dp` table with a nested loop.

Space Complexity:
O(amount), as we use a vector of size `amount + 1` to store the dynamic programming values.
*/

impl Solution {
    pub fn change(amount: i32, coins: Vec<i32>) -> i32 {
        let amount = amount as usize;
        let mut dp = vec![0; amount + 1];
        dp[0] = 1;

        for coin in coins.iter() {
            for i in *coin as usize..=amount {
                dp[i] += dp[i - *coin as usize];
            }
        }

        dp[amount]
    }
}
```

114. 14_2-D_Dynamic_Programming/05_Target_Sum/0494-target-sum.rs

```
/*
Problem: LeetCode 494 - Target Sum

Key Idea:
The key idea is to use dynamic programming to count the number of ways to achieve a target sum using the given positive and negative numbers.

Approach:
1. Initialize a variable `sum` to store the sum of all elements in the input array `nums`.
2. Calculate the target sum `target` as the difference between the `sum` and the given `S`.
3. Check if `target` is odd or negative. If it is, there are no ways to achieve the target sum, so return 0.
4. Initialize a vector `dp` of size `2 * target + 1` to store the number of ways to achieve each possible sum from `-target` to `target`.
5. Initialize `dp[0]` as 1 because there is one way to achieve a sum of 0 (by not choosing any number).
6. Iterate through the elements of `nums`:
     - For each element `num`, iterate through the `dp` vector from `target` down to `num` (inclusive).
    - For each index `i`, update `dp[i]` by adding `dp[i - num]`. This represents the number of ways to achieve the sum `i` by including the current element.
7. After the iteration, `dp[target]` will contain the number of ways to achieve the target sum.
8. Return `dp[target]` as the result.

Time Complexity:
O(n * target), where `n` is the number of elements in the input array `nums` and `target` is the target sum. We fill in the `dp` table with a nested loop.

Space Complexity:
O(target), as we use a vector of size `2 * target + 1` to store the dynamic programming values.
*/

impl Solution {
    pub fn find_target_sum_ways(nums: Vec<i32>, s: i32) -> i32 {
        let sum: i32 = nums.iter().sum();
        let target = sum - s;

        if target < 0 || target % 2 != 0 {
            return 0;
        }

        let target = (target / 2) as usize;
        let mut dp = vec![0; target + 1];
        dp[0] = 1;

        for num in nums {
            for i in (num as usize..=target).rev() {
                dp[i] += dp[i - num as usize];
            }
        }

        dp[target]
    }
}
```

115. 14_2-D_Dynamic_Programming/06_Interleaving_String/0097-interleaving-string.r

```
/*
Problem: LeetCode 97 - Interleaving String

Key Idea:
The key idea is to use dynamic programming to determine whether a third string is formed by interleaving two given strings.

Approach:
1. Initialize a 2D vector `dp` of size `(s1.len() + 1) x (s2.len() + 1)`, where `dp[i][j]` represents whether the first `i` characters of `s1` and the first `j` characters of `s2` can interleave to form the first `i+j` characters of the result string.
2. Initialize `dp[0][0]` to `true` because two empty strings can interleave to form an empty string.
3. Initialize the first row and the first column of `dp`:
   - For `dp[i][0]`, it is `true` if `s1[i-1] == s3[i-1]` and `dp[i-1][0]` is `true` (meaning the previous characters interleaved correctly).
   - For `dp[0][j]`, it is `true` if `s2[j-1] == s3[j-1]` and `dp[0][j-1]` is `true` (meaning the previous characters interleaved correctly).
4. Iterate through the characters of `s1` and `s2` using nested loops from 1 to `s1.len()` and 1 to `s2.len()`.
5. For each pair of indices `i` and `j`, calculate `k = i + j` (the current index in `s3`).
6. Update `dp[i][j]` as `true` if any of the following conditions is met:
   - `s1[i-1] == s3[k-1]` and `dp[i-1][j]` is `true` (meaning the previous characters interleaved correctly).
   - `s2[j-1] == s3[k-1]` and `dp[i][j-1]` is `true` (meaning the previous characters interleaved correctly).
7. After the iteration, `dp[s1.len()][s2.len()]` will indicate whether `s1` and `s2` can interleave to form `s3`.
8. Return `dp[s1.len()][s2.len()]` as the result.

Time Complexity:
O(s1.len() * s2.len()), where `s1.len()` is the length of string `s1` and `s2.len()` is the length of string `s2`. We fill in the `dp` table with a nested loop.

Space Complexity:
O(s1.len() * s2.len()), as we use a 2D vector of size `(s1.len() + 1) x (s2.len() + 1)` to store the dynamic programming values.
*/

impl Solution {
    pub fn is_interleave(s1: String, s2: String, s3: String) -> bool {
        let s1 = s1.as_bytes();
        let s2 = s2.as_bytes();
        let s3 = s3.as_bytes();
        let len1 = s1.len();
        let len2 = s2.len();
        let len3 = s3.len();

        if len1 + len2 != len3 {
            return false;
        }

        let mut dp = vec![vec![false; len2 + 1]; len1 + 1];
        dp[0][0] = true;

        for i in 1..=len1 {
            dp[i][0] = dp[i - 1][0] && s1[i - 1] == s3[i - 1];
        }

        for j in 1..=len2 {
```

```
            dp[0][j] = dp[0][j - 1] && s2[j - 1] == s3[j - 1];
        }

        for i in 1..=len1 {
            for j in 1..=len2 {
                let k = i + j;
                dp[i][j] = (dp[i - 1][j] && s1[i - 1] == s3[k - 1])
                    || (dp[i][j - 1] && s2[j - 1] == s3[k - 1]);
            }
        }

        dp[len1][len2]
    }
}
```

116. 14_2-D_Dynamic_Programming/07_Longest_Increasing_Path_In_a_Matrix/0329-

```
/*
Problem: LeetCode 329 - Longest Increasing Path in a Matrix

Key Idea:
The key idea is to use dynamic programming to find the length of the longest increasing path in a matrix.

Approach:
1. Initialize a 2D vector `dp` of the same size as the matrix, where `dp[i][j]` represents the length of the longest increasing path starting from cell `(i, j)`.
2. Initialize a variable `max_length` to store the maximum path length found so far, and set it to 0.
3. Iterate through each cell in the matrix:
    - For each cell `(i, j)`, recursively explore its neighboring cells (up, down, left, and right).
   - For each neighboring cell `(x, y)`, if `(x, y)` has a greater value than `(i, j)`, calculate `length` as `dp[x][y] + 1`.
   - Update `dp[i][j]` as the maximum `length` among all valid neighboring cells.
   - Update `max_length` with the maximum value of `dp[i][j]` and `max_length`.
4. After iterating through all cells, `max_length` will contain the length of the longest increasing path in the matrix.
5. Return `max_length` as the result.

Time Complexity:
O(m * n), where `m` is the number of rows and `n` is the number of columns in the matrix. We visit each cell once in the matrix.

Space Complexity:
O(m * n), as we use a 2D vector of the same size as the matrix to store the dynamic programming values.
*/

impl Solution {
    pub fn longest_increasing_path(matrix: Vec<Vec<i32>>) -> i32 {
        if matrix.is_empty() || matrix[0].is_empty() {
            return 0;
        }

        let m = matrix.len();
        let n = matrix[0].len();
        let mut dp = vec![vec![0; n]; m];
        let mut max_length = 0;

        fn dfs(
            matrix: &Vec<Vec<i32>>,
            dp: &mut Vec<Vec<i32>>,
            m: usize,
            n: usize,
            i: usize,
            j: usize,
        ) -> i32 {
            if dp[i][j] != 0 {
                return dp[i][j];
            }

            let directions = [(0, 1), (0, -1), (1, 0), (-1, 0)];
            let mut max_len = 1;

            for &(dx, dy) in directions.iter() {
                let x = i as i32 + dx;
```

```
                let y = j as i32 + dy;

                if x >= 0
                    && x < m as i32
                    && y >= 0
                    && y < n as i32
                    && matrix[x as usize][y as usize] > matrix[i][j]
                {
                    max_len = max_len.max(1 + dfs(matrix, dp, m, n, x as usize, y as usize));
                }
            }

            dp[i][j] = max_len;
            max_len
        }

        for i in 0..m {
            for j in 0..n {
                max_length = max_length.max(dfs(&matrix, &mut dp, m, n, i, j));
            }
        }

        max_length
    }
}
```

117. 14_2-D_Dynamic_Programming/08_Distinct_Subsequences/0115-distinct-subsec

```
/*
Problem: LeetCode 115 - Distinct Subsequences

Key Idea:
The key idea is to use dynamic programming to count the number of distinct subsequences of string
`s` that are equal to string `t`.

Approach:
1. Initialize a 2D vector `dp` of size `(m+1) x (n+1)`, where `dp[i][j]` represents the number of
distinct subsequences of the first `i` characters of string `s` that are equal to the first `j`
characters of string `t`.
2. Initialize the first row of `dp` with all zeros because there is no way to form a non-empty
string `t` from an empty string `s`.
3. Initialize `dp[0][0]` to 1 because an empty string `s` can form an empty string `t` in one way
(by not selecting any characters).
4. Iterate through the characters of string `s` and string `t` using nested loops, from 1 to `m`
and 1 to `n`.
5. For each pair of indices `i` and `j`, compare `s[i-1]` and `t[j-1]`:
   - If `s[i-1]` is equal to `t[j-1]`, we have two choices:
       - We can include the current character `s[i-1]` in the subsequence. In this case, add
`dp[i-1][j-1]` to `dp[i][j]`, which represents selecting the current character.
       - We can exclude the current character `s[i-1]` from the subsequence. In this case, add
`dp[i-1][j]` to `dp[i][j]`, which represents not selecting the current character.
   - If `s[i-1]` is not equal to `t[j-1]`, we can only exclude the current character `s[i-1]`, so
add `dp[i-1][j]` to `dp[i][j]`.
6. After the iteration, `dp[m][n]` will contain the number of distinct subsequences of string `s`
equal to string `t`.
7. Return `dp[m][n]` as the result.

Time Complexity:
O(m * n), where `m` is the length of string `s` and `n` is the length of string `t`. We fill in
the `dp` table with a nested loop.

Space Complexity:
O(m * n), as we use a 2D vector of size `(m+1) x (n+1)` to store the dynamic programming values.
*/

impl Solution {
    pub fn num_distinct(s: String, t: String) -> i32 {
        let s_chars: Vec<char> = s.chars().collect();
        let t_chars: Vec<char> = t.chars().collect();
        let m = s_chars.len();
        let n = t_chars.len();

        if m < n {
            return 0;
        }

        let mut dp = vec![vec![0; n + 1]; m + 1];

        for i in 0..=m {
            dp[i][0] = 1; // Any string can form an empty string in one way.
        }

        for i in 1..=m {
            for j in 1..=n {
                if s_chars[i - 1] == t_chars[j - 1] {
                    dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j];
                } else {
                    dp[i][j] = dp[i - 1][j];
```

```
                }
            }
        }

        dp[m][n]
    }
}

/*
impl Solution {
    pub fn num_distinct(s: String, t: String) -> i32 {
        let s_bytes = s.as_bytes();
        let t_bytes = t.as_bytes();

        let mut dp = vec![0; s.len()];
        let mut result = 1;

        for &char_t in t_bytes {
            let mut temp = result;
            result = 0;

            for i in (0..s.len()).rev() {
                temp -= dp[i];
                dp[i] = (s_bytes[i] == char_t) as i32 * temp;
                result += dp[i];
            }
        }

        result
    }
}
*/
```

118. 14_2-D_Dynamic_Programming/09_Edit_Distance/0072-edit-distance.rs

```
/*
Problem: LeetCode 72 - Edit Distance

Key Idea:
The key idea is to use dynamic programming to find the minimum number of operations (insert, delete, or replace) required to convert one string into another.

Approach:
1. Initialize a 2D vector `dp` of size `(m+1) x (n+1)`, where `dp[i][j]` represents the minimum edit distance between the first `i` characters of string `word1` and the first `j` characters of string `word2`.
2. Initialize the first row and the first column of `dp` to represent the case where one of the strings is empty. For example, `dp[i][0]` represents the edit distance between the first `i` characters of `word1` and an empty string, which is simply `i` (deletions).
    Similarly, `dp[0][j]` represents the edit distance between an empty string and the first `j` characters of `word2`, which is simply `j` (insertions).
3. Iterate through the characters of `word1` and `word2` using nested loops from 1 to `m` and 1 to `n`.
4. For each pair of indices `i` and `j`, compare `word1[i-1]` and `word2[j-1]`:
   - If they are equal, no operation is needed, so `dp[i][j] = dp[i-1][j-1]`.
   - If they are not equal, calculate `dp[i][j]` as the minimum of three options:
     - Deletion: `dp[i-1][j] + 1` (delete a character from `word1`).
     - Insertion: `dp[i][j-1] + 1` (insert a character into `word1`).
     - Replacement: `dp[i-1][j-1] + 1` (replace `word1[i-1]` with `word2[j-1]`).
5. After the iteration, `dp[m][n]` will contain the minimum edit distance between `word1` and `word2`.
6. Return `dp[m][n]` as the result.

Time Complexity:
O(m * n), where `m` is the length of `word1` and `n` is the length of `word2`. We fill in the `dp` table with a nested loop.

Space Complexity:
O(m * n), as we use a 2D vector of size `(m+1) x (n+1)` to store the dynamic programming values.
*/

impl Solution {
    pub fn min_distance(word1: String, word2: String) -> i32 {
        let word1_chars: Vec<char> = word1.chars().collect();
        let word2_chars: Vec<char> = word2.chars().collect();
        let m = word1_chars.len();
        let n = word2_chars.len();

        let mut dp = vec![vec![0; n + 1]; m + 1];

        // Initialize the first row and first column
        for i in 0..=m {
            dp[i][0] = i;
        }
        for j in 0..=n {
            dp[0][j] = j;
        }

        for i in 1..=m {
            for j in 1..=n {
                if word1_chars[i - 1] == word2_chars[j - 1] {
                    dp[i][j] = dp[i - 1][j - 1];
                } else {
                    dp[i][j] = 1 + dp[i - 1][j - 1].min(dp[i][j - 1].min(dp[i - 1][j]));
                }
```

```
            }
        }

        dp[m][n] as i32
    }
}
```

119. 14_2-D_Dynamic_Programming/10_Burst_Balloons/0312-burst-balloons.rs

```
/*
Problem: LeetCode 312 - Burst Balloons

Key Idea:
The key idea is to use dynamic programming to find the maximum coins that can be obtained by bursting balloons in the given order.

Approach:
1. We can approach this problem using dynamic programming. We will create a 2D array `dp` where `dp[i][j]` represents the maximum coins obtained by bursting balloons from index `i` to `j` (inclusive).
2. We will iterate over different balloon ranges by varying the length of the range from 1 to `n`, where `n` is the number of balloons.
3. For each range, we will iterate over all possible starting points for the range.
4. Within each range, we will consider different options for the last balloon to burst. We will choose the last balloon such that it maximizes the total coins.
5. To calculate the coins for bursting the last balloon, we will use the formula:
   `coins[i][j] = max(coins[i][j], nums[i-1]*nums[k]*nums[j+1] + dp[i][k-1] + dp[k+1][j])`
   where `k` is the index of the last balloon in the range `[i, j]`.
6. After iterating through all possible ranges and all possible last balloons to burst, `dp[0][n-1]` will contain the maximum coins that can be obtained.
7. Return `dp[0][n-1]` as the result.

Time Complexity:
O(n^3), where `n` is the number of balloons. We have three nested loops to calculate the `dp` array.

Space Complexity:
O(n^2), as we use a 2D array of size `n x n` to store the dynamic programming values.
*/

impl Solution {
    pub fn max_coins(nums: Vec<i32>) -> i32 {
        let n = nums.len();
        let mut dp = vec![vec![0; n]; n];

        for len in 1..=n {
            for i in 0..=n - len {
                let j = i + len - 1;
                for k in i..=j {
                    let left = if i == 0 { 1 } else { nums[i - 1] };
                    let right = if j == n - 1 { 1 } else { nums[j + 1] };
                    let burst = nums[k] * left * right;
                    let before = if k == i { 0 } else { dp[i][k - 1] };
                    let after = if k == j { 0 } else { dp[k + 1][j] };
                    dp[i][j] = dp[i][j].max(burst + before + after);
                }
            }
        }

        dp[0][n - 1]
    }
}
```

120. 14_2-D_Dynamic_Programming/11_Regular_Expression_Matching/0010-regular-

```
/*
Problem: LeetCode 10 - Regular Expression Matching

Key Idea:
The key idea is to use dynamic programming to determine if a given string matches a regular expression pattern.

Approach:
1. We can approach this problem using dynamic programming. We will create a 2D array `dp` where `dp[i][j]` represents whether the first `i` characters of the string match the first `j` characters of the pattern.
2. Initialize `dp[0][0]` to true, as an empty string matches an empty pattern.
3. Iterate through the pattern characters using a nested loop:
   - If the current pattern character is not a wildcard '*', set `dp[0][j]` to false for all `j > 0` because no non-empty string can match an empty pattern.
   - If the current pattern character is a wildcard '*', set `dp[0][j]` to `dp[0][j-2]` to handle the case where the '*' acts as zero occurrences of the preceding character.
4. Iterate through the string and pattern characters using nested loops, from 1 to the length of the string and 1 to the length of the pattern.
5. For each pair of indices `i` and `j`, compare the current string character `s[i-1]` with the current pattern character `p[j-1]`:
   - If they match (or the pattern character is a '.'):
     - Set `dp[i][j]` to `dp[i-1][j-1]`.
   - If the pattern character is a '*':
     - Consider two cases:
       - Zero occurrences of the preceding character: Set `dp[i][j]` to `dp[i][j-2]`.
       - One or more occurrences of the preceding character:
           - Check if the preceding character in the pattern `p[j-2]` matches the current string character `s[i-1]` or if `p[j-2]` is a '.'.
         - If there's a match, set `dp[i][j]` to `dp[i-1][j]`.
6. After the iteration, `dp[m][n]` will contain whether the entire string matches the entire pattern.
7. Return `dp[m][n]` as the result, where `m` is the length of the string and `n` is the length of the pattern.

Time Complexity:
O(m * n), where `m` is the length of the string and `n` is the length of the pattern. We fill in the `dp` table with a nested loop.

Space Complexity:
O(m * n), as we use a 2D array of size `(m+1) x (n+1)` to store the dynamic programming values.
*/

impl Solution {
    pub fn is_match(s: String, p: String) -> bool {
        let s_chars: Vec<char> = s.chars().collect();
        let p_chars: Vec<char> = p.chars().collect();
        let m = s_chars.len();
        let n = p_chars.len();

        let mut dp = vec![vec![false; n + 1]; m + 1];
        dp[0][0] = true; // Empty string matches empty pattern

        // Initialize dp[0][j] for wildcards '*'
        for j in 2..=n {
            if p_chars[j - 1] == '*' {
                dp[0][j] = dp[0][j - 2];
            }
        }
```

```
        for i in 1..=m {
            for j in 1..=n {
                if p_chars[j - 1] == s_chars[i - 1] || p_chars[j - 1] == '.' {
                    dp[i][j] = dp[i - 1][j - 1];
                } else if p_chars[j - 1] == '*' {
                    dp[i][j] = dp[i][j - 2]
                        || (dp[i - 1][j]
                            && (p_chars[j - 2] == s_chars[i - 1] || p_chars[j - 2] == '.'));
                }
            }
        }

        dp[m][n]
    }
}
```

121. 15_Greedy/01_Maximum_Subarray/0053-maximum-subarray.rs

```
/*
Problem: LeetCode 53 - Maximum Subarray

Key Idea:
The key idea is to use Kadane's algorithm to efficiently find the maximum sum of a subarray.

Approach:
1. Initialize two variables: 'max_sum' to store the maximum sum found so far and 'current_sum' to store the current sum.
2. Iterate through the array:
    - For each element, update 'current_sum' by taking the maximum of the current element and the sum of the current element and 'current_sum'.
   - Update 'max_sum' by taking the maximum of 'max_sum' and 'current_sum'.
    - This algorithm effectively keeps track of the maximum subarray sum ending at the current element.
   - By the end of the iteration, 'max_sum' will contain the maximum subarray sum.
3. Return 'max_sum' as the result.

Time Complexity:
O(n), where n is the number of elements in the array. We perform a single pass through the array.

Space Complexity:
O(1), as we only use a constant amount of extra space to store 'max_sum' and 'current_sum'.
*/

impl Solution {
    pub fn max_sub_array(nums: Vec<i32>) -> i32 {
        let mut max_sum = nums[0];
        let mut current_sum = nums[0];

        for i in 1..nums.len() {
            current_sum = nums[i].max(nums[i] + current_sum);
            max_sum = max_sum.max(current_sum);
        }

        max_sum
    }
}
```

122. 15_Greedy/02_Jump_Game/0055-jump-game.rs

```
/*
Problem: LeetCode 55 - Jump Game

Key Idea:
The key idea is to determine if it's possible to reach the last index of the array by jumping from one element to another.

Approach:
1. Initialize a variable 'max_reach' to store the maximum index that can be reached from the current position.
2. Iterate through the array:
   - For each element, update 'max_reach' by taking the maximum of its current value and the sum of its value and the current index.
   - If 'max_reach' is greater than or equal to the last index, it means it's possible to reach the end of the array.
   - Keep updating 'max_reach' as long as it's greater than the current index.
3. If 'max_reach' is greater than or equal to the last index after the iteration, return true; otherwise, return false.

Time Complexity:
O(n), where n is the number of elements in the array. We perform a single pass through the array.

Space Complexity:
O(1), as we only use a constant amount of extra space to store 'max_reach'.
*/

impl Solution {
    pub fn can_jump(nums: Vec<i32>) -> bool {
        let mut max_reach = 0;

        for i in 0..nums.len() {
            if max_reach < i {
                return false; // Cannot reach the current index
            }

            max_reach = max_reach.max(i + nums[i] as usize);

            if max_reach >= nums.len() - 1 {
                return true; // Can reach or surpass the last index
            }
        }

        false
    }
}
```

123. 15_Greedy/03_Jump_Game_II/0045-jump-game-ii.rs

```
/*
Problem: LeetCode 45 - Jump Game II

Key Idea:
The key idea is to use a greedy approach to find the minimum number of jumps required to reach the end of the array.

Approach:
1. Initialize two variables: 'max_reach' to store the maximum index that can be reached with the current number of jumps and 'end' to store the index of the last element.
2. Initialize a variable 'jumps' to store the number of jumps needed to reach the current 'max_reach'.
3. Iterate through the array:
   - For each element, update 'max_reach' by taking the maximum of its current value and the sum of its value and the current index.
   - If the current index reaches the 'end', it means we have reached the end of the current jump.
   - In this case, update 'end' to 'max_reach' and increment 'jumps'.
4. Return 'jumps' as the result.

Time Complexity:
O(n), where n is the number of elements in the array. We perform a single pass through the array.

Space Complexity:
O(1), as we only use a constant amount of extra space to store variables.
*/

impl Solution {
    pub fn jump(nums: Vec<i32>) -> i32 {
        let mut max_reach = 0;
        let mut end = 0;
        let mut jumps = 0;

        for i in 0..nums.len() - 1 {
            max_reach = max_reach.max(i + nums[i] as usize);

            if i == end {
                end = max_reach;
                jumps += 1;
            }
        }

        jumps
    }
}
```

124. 15_Greedy/04_Gas_Station/0134-gas-station.rs

```
/*
Problem: LeetCode 134 - Gas Station

Key Idea:
The key idea is to determine if there exists a starting gas station such that you can complete a circular tour of all gas stations.

Approach:
1. Initialize two variables: 'total_gas' to track the total gas available and 'current_gas' to track the gas available at the current station.
2. Initialize a variable 'start_station' to store the potential starting station's index.
3. Iterate through the gas stations in a circular manner (starting from 0 and looping back to 0):
   - For each station, calculate the difference between gas and cost (gas[i] - cost[i]) and add it to 'current_gas'.
    - If 'current_gas' becomes negative at any station, reset 'current_gas' to 0 and update 'start_station' to the next station.
   - Keep track of 'total_gas' by accumulating the gas-cost differences.
4. If 'total_gas' is non-negative, return 'start_station' as the answer; otherwise, return -1, indicating that it's impossible to complete the tour.

Time Complexity:
O(n), where n is the number of gas stations. We perform a single pass through the stations.

Space Complexity:
O(1), as we only use a constant amount of extra space to store variables.
*/

impl Solution {
    pub fn can_complete_circuit(gas: Vec<i32>, cost: Vec<i32>) -> i32 {
        let mut total_gas = 0;
        let mut current_gas = 0;
        let mut start_station = 0;

        for i in 0..gas.len() {
            let diff = gas[i] - cost[i];
            current_gas += diff;
            total_gas += diff;

            if current_gas < 0 {
                current_gas = 0;
                start_station = i + 1;
            }
        }

        if total_gas >= 0 {
            start_station as i32
        } else {
            -1
        }
    }
}
```

125. 15_Greedy/05_Hand_of_Straights/0846-hand-of-straights.rs

```
/*
Problem: LeetCode 846 - Hand of Straights

Key Idea:
The key idea is to group the cards into consecutive hands of 'W' cards each, where 'W' is the given group size, such that each group is consecutive and all groups are of the same size.

Approach:
1. Create a HashMap to count the frequency of each card in the input hand.
2. Sort the unique cards from the input hand.
3. Iterate through the sorted unique cards:
   - For each card, check if its frequency is greater than zero.
   - If so, try to form a group of 'W' consecutive cards starting from the current card.
     - To do this, iterate from the current card to the next 'W - 1' cards, reducing their frequencies.
   - If a group of 'W' consecutive cards cannot be formed, return false.
4. If all groups can be formed, return true.

Time Complexity:
O(N * log(N)), where N is the number of unique cards in the input hand. This is because we need to sort the unique cards.

Space Complexity:
O(N), where N is the number of unique cards in the input hand. This is for the HashMap to store the frequencies of the cards.
*/

use std::collections::HashMap;

impl Solution {
    pub fn is_n_straight_hand(hand: Vec<i32>, w: i32) -> bool {
        if hand.len() % w as usize != 0 {
            return false;
        }

        let mut freq_map: HashMap<i32, i32> = HashMap::new();

        // Count the frequency of each card in the hand
        for card in &hand {
            *freq_map.entry(*card).or_insert(0) += 1;
        }

        let mut unique_cards: Vec<i32> = freq_map.keys().cloned().collect();
        unique_cards.sort();

        for card in unique_cards {
            let freq = *freq_map.get(&card).unwrap();

            if freq > 0 {
                for i in 0..w {
                    let next_card = card + i;

                    if let Some(&next_freq) = freq_map.get(&next_card) {
                        if next_freq < freq {
                            return false; // Cannot form a group
                        }
                        *freq_map.get_mut(&next_card).unwrap() -= freq;
                    } else {
                        return false; // Cannot form a group
                    }
```

```
                }
            }
        }

        true
    }
}

/*
// Using heap

use std::collections::{BinaryHeap, HashMap};
use std::cmp::Reverse;

impl Solution {
    pub fn is_n_straight_hand(hand: Vec<i32>, group_size: i32) -> bool {
        if hand.len() % group_size as usize != 0 {
            return false;
        }

        let mut card_counts: HashMap<i32, i32> = HashMap::new();
        for card in hand {
            *card_counts.entry(card).or_insert(0) += 1;
        }

        let mut min_heap: BinaryHeap<Reverse<i32>> = BinaryHeap::new();
        for &card in card_counts.keys() {
            min_heap.push(Reverse(card));
        }

        while let Some(Reverse(first_card)) = min_heap.peek() {
            for card in *first_card..(*first_card + group_size) {
                if !card_counts.contains_key(&card) {
                    return false;
                }

                if let Some(&count) = card_counts.get(&card) {
                    if count == 1 {
                        if let Some(Reverse(peeked_card)) = min_heap.peek() {
                            if card != *peeked_card {
                                return false;
                            }
                            min_heap.pop();
                        }
                    } else {
                        *card_counts.get_mut(&card).unwrap() -= 1;
                    }
                }
            }
        }

        true
    }
}
*/
```

126. 15_Greedy/06_Merge_Triplets_to_Form_Target_Triplet/1899-merge-triplets-to-fo

```
/*
Problem: LeetCode 1899 - Merge Triplets to Form Target Triplet

Key Idea:
The key idea is to iterate through the triplets and check if they can be combined to form the target triplet.

Approach:
1. Initialize three boolean variables 'exists_x', 'exists_y', and 'exists_z' to track whether each component (x, y, and z) exists in any valid triplet.
2. Initialize an empty vector 'valid_triplets' to store valid triplets that satisfy the conditions.
3. Iterate through the triplets:
    - For each triplet (a, b, c), if a <= target_x, b <= target_y, and c <= target_z, add the triplet to 'valid_triplets'.
   - While iterating, if a == target_x, set 'exists_x' to true. If b == target_y, set 'exists_y' to true. If c == target_z, set 'exists_z' to true.
4. After iterating through all triplets, check if 'exists_x', 'exists_y', and 'exists_z' are all true, indicating that each component exists in at least one valid triplet.
5. Also, check if 'valid_triplets' is not empty, as this confirms that there are valid triplets satisfying the conditions.
6. If both conditions are met, return true; otherwise, return false.

Time Complexity:
O(n), where n is the number of triplets. We perform a single pass through the triplets.

Space Complexity:
O(n), as the 'valid_triplets' vector can potentially store all valid triplets.
*/

impl Solution {
    pub fn merge_triplets(triplets: Vec<Vec<i32>>, target: Vec<i32>) -> bool {
        let (target_x, target_y, target_z) = (target[0], target[1], target[2]);
        let mut exists = [false; 3];
        let mut valid_triplets = vec![];

        for triplet in triplets {
            let (x, y, z) = (triplet[0], triplet[1], triplet[2]);

            if x <= target_x && y <= target_y && z <= target_z {
                if x == target_x {
                    exists[0] = true;
                }
                if y == target_y {
                    exists[1] = true;
                }
                if z == target_z {
                    exists[2] = true;
                }
                valid_triplets.push(triplet);
            }
        }

        exists.iter().all(|&e| e) && !valid_triplets.is_empty()
    }
}
```

127. 15_Greedy/07_Partition_Labels/0763-partition-labels.rs

```
/*
Problem: LeetCode 763 - Partition Labels

Key Idea:
The key idea is to find the partitions of the given string such that each letter appears in only one partition.

Approach:
1. Initialize a HashMap to store the last occurrence index of each character in the string.
2. Iterate through the string to populate the HashMap with the last occurrence index of each character.
3. Initialize two variables 'start' and 'end' to keep track of the current partition's start and end.
4. Iterate through the string again:
     - For each character, update 'end' with the maximum of its current value and the last occurrence index of that character from the HashMap.
    - If the current index is equal to 'end', it means we have reached the end of the current partition.
   - Add the length of this partition (end - start + 1) to the result list.
   - Update 'start' to the next index.
5. Return the result list.

Time Complexity:
O(n), where n is the length of the input string. We perform two passes through the string.

Space Complexity:
O(1), as the HashMap has a constant number of characters (26 for lowercase English letters) and the result list is not counted towards space complexity.
*/

use std::collections::HashMap;

impl Solution {
    pub fn partition_labels(s: String) -> Vec<i32> {
        let mut last_occurrence: HashMap<char, usize> = HashMap::new();
        let mut result: Vec<i32> = Vec::new();

        // Populate the last_occurrence HashMap
        for (i, c) in s.chars().enumerate() {
            last_occurrence.insert(c, i);
        }

        let mut start = 0;
        let mut end = 0;

        // Iterate through the string to find partitions
        for (i, c) in s.chars().enumerate() {
            end = end.max(*last_occurrence.get(&c).unwrap());

            if i == end {
                result.push((end - start + 1) as i32);
                start = i + 1;
            }
        }

        result
    }
}
```

128. 15_Greedy/08_Valid_Parenthesis_String/0678-valid-parenthesis-string.rs

```
/*
Problem: LeetCode 678 - Valid Parenthesis String

Key Idea:
The key idea is to maintain two counters, 'balance_low' and 'balance_high,' to keep track of the possible minimum and maximum open parentheses.

Approach:
1. Initialize two variables 'balance_low' and 'balance_high' to 0. These variables will track the minimum and maximum open parentheses count.
2. Iterate through each character 'ch' in the input string 's':
   - If 'ch' is '(', increment both 'balance_low' and 'balance_high' because it can be used as an open parenthesis.
    - If 'ch' is '*', decrement 'balance_low' (considering it as an empty string) and increment 'balance_high' (considering it as an open parenthesis or empty string).
   - If 'ch' is ')', decrement both 'balance_low' and 'balance_high' because it can be used as a close parenthesis.
     - If 'ch' is any other character, it should not exist in a valid string, so we use 'unreachable!' to indicate an unreachable code path.
3. After processing each character, check if 'balance_high' is negative. If it is, return false because there are more closing parentheses than open parentheses, and it's impossible to balance them.
4. To ensure 'balance_low' is non-negative, use 'balance_low.max(0)' to set it to zero if it becomes negative.
5. After processing the entire string, check if 'balance_low' is zero. If it is, return true because it means all open parentheses have corresponding close parentheses.
6. If 'balance_low' is not zero, return false because there are unmatched open parentheses.

Time Complexity:
O(n), where n is the length of the input string 's.' We perform a single pass through the string.

Space Complexity:
O(1), as we use only a constant amount of extra space for variables.
*/

impl Solution {
    pub fn check_valid_string(s: String) -> bool {
        let (mut balance_low, mut balance_high) = (0, 0);

        for ch in s.chars() {
            match ch {
                '(' => {
                    balance_low += 1;
                    balance_high += 1;
                }
                '*' => {
                    balance_low -= 1;
                    balance_high += 1;
                }
                ')' => {
                    balance_low -= 1;
                    balance_high -= 1;
                }
                _ => unreachable!(),
            }

            if balance_high < 0 {
                return false;
            }
```

```
            balance_low = balance_low.max(0);
        }

        balance_low == 0
    }
}
```

129. 16_Intervals/01_Insert_Interval/0057-insert-intervals.rs

```
/*
Problem: LeetCode 57 - Insert Interval

Key Idea:
The key idea is to iterate through the existing intervals and merge any overlapping intervals with the new interval to be inserted. This ensures that the resulting list remains sorted and non-overlapping.

Approach:
1. Initialize an empty vector 'result' to store the merged intervals.
2. Iterate through the intervals in 'intervals':
   - If the current interval's end is less than the new interval's start, add it to 'result'.
   - If the current interval's start is greater than the new interval's end, add the new interval to 'result' and update the new interval to the current interval.
    - If there is an overlap between the current interval and the new interval, update the new interval's start to the minimum of the two intervals' starts and its end to the maximum of the two intervals' ends.
3. After the loop, add the remaining new interval to 'result'.
4. Return 'result' as the merged intervals.

Time Complexity:
O(n), where n is the number of intervals. We perform a single pass through the intervals.

Space Complexity:
O(n), where n is the number of intervals. In the worst case, 'result' will contain all intervals.
*/

impl Solution {
    pub fn insert(intervals: Vec<Vec<i32>>, new_interval: Vec<i32>) -> Vec<Vec<i32>> {
        let mut result: Vec<Vec<i32>> = Vec::new();
        let mut new_interval = new_interval;
        let mut i = 0;
        let n = intervals.len();

        // Add intervals that come before the new_interval
        while i < n && intervals[i][1] < new_interval[0] {
            result.push(intervals[i].clone());
            i += 1;
        }

        // Merge intervals with overlap
        while i < n && intervals[i][0] <= new_interval[1] {
            new_interval[0] = new_interval[0].min(intervals[i][0]);
            new_interval[1] = new_interval[1].max(intervals[i][1]);
            i += 1;
        }

        // Add the merged new_interval
        result.push(new_interval);

        // Add intervals that come after the new_interval
        while i < n {
            result.push(intervals[i].clone());
            i += 1;
        }

        result
    }
}
```

130. 16_Intervals/02_Merge_Intervals/0056-merge-intervals.rs

```
/*
Problem: LeetCode 56 - Merge Intervals

Key Idea:
The key idea is to first sort the intervals by their start times. Then, iterate through the sorted intervals and merge overlapping ones by updating the end time of the merged interval. Keep track of non-overlapping intervals and return the merged result.

Approach:
1. Sort the intervals based on their start values.
2. Initialize an empty vector 'result' to store the merged intervals.
3. Iterate through the sorted intervals:
   - If 'result' is empty or the current interval's start is greater than the last interval's end in 'result', add the current interval to 'result'.
   - If there is an overlap between the current interval and the last interval in 'result', merge them by updating the last interval's end to the maximum of the two intervals' ends.
4. Return 'result' as the merged intervals.

Time Complexity:
O(n log n), where n is the number of intervals. The sorting step takes O(n log n) time.

Space Complexity:
O(n), where n is the number of intervals. 'result' can have a maximum of n merged intervals.
*/

impl Solution {
    pub fn merge(intervals: Vec<Vec<i32>>) -> Vec<Vec<i32>> {
        if intervals.is_empty() {
            return vec![];
        }

        let mut intervals = intervals;
        intervals.sort_by_key(|x| x[0]);

        let mut result: Vec<Vec<i32>> = Vec::new();
        result.push(intervals[0].clone());

        for i in 1..intervals.len() {
            let current = intervals[i].clone();
            let last = result.last_mut().unwrap();

            if current[0] > last[1] {
                result.push(current);
            } else {
                last[1] = last[1].max(current[1]);
            }
        }

        result
    }
}
```

131. 16_Intervals/03_Non_Overlapping_Intervals/0435-non-overlapping-intervals.rs

```
/*
Problem: LeetCode 435 - Non-overlapping Intervals

Key Idea:
The key idea is to sort the intervals by their end times and then use greedy selection. Choose intervals that end earliest and remove overlapping intervals. This ensures that you retain as many non-overlapping intervals as possible.

Approach:
1. Sort the intervals based on their end values in ascending order.
2. Initialize a variable 'count' to 0 to keep track of non-overlapping intervals.
3. Initialize a variable 'prev_end' to -inf to store the end value of the last added interval.
4. Iterate through the sorted intervals:
   - If the current interval's start is greater than or equal to 'prev_end', it means it does not overlap with the previous interval.
     - Increment 'count' by 1.
     - Update 'prev_end' to the end value of the current interval.
   - If the current interval overlaps with the previous interval, skip it.
5. Return the value of 'count' as the maximum number of non-overlapping intervals.

Time Complexity:
O(n log n), where n is the number of intervals. The sorting step takes O(n log n) time.

Space Complexity:
O(1), as we use only a constant amount of extra space.
*/

impl Solution {
    pub fn erase_overlap_intervals(intervals: Vec<Vec<i32>>) -> i32 {
        if intervals.is_empty() {
            return 0;
        }

        let mut intervals = intervals;
        intervals.sort_by_key(|x| x[1]);

        let mut count = 1;
        let mut prev_end = intervals[0][1];

        for i in 1..intervals.len() {
            if intervals[i][0] >= prev_end {
                count += 1;
                prev_end = intervals[i][1];
            }
        }

        intervals.len() as i32 - count
    }
}
```

132. 16_Intervals/04_Meeting_Rooms/0252-meeting-rooms.rs

```
/*
Problem: LeetCode 252 - Meeting Rooms

Key Idea:
The key idea is to sort the meetings by their start times and then check if there are any overlaps by comparing the end time of one meeting with the start time of the next. If there are no overlaps, you can schedule all the meetings.

Approach:
1. Sort the intervals based on their start times in ascending order.
2. Iterate through the sorted intervals:
    - If the end time of the current interval is greater than or equal to the start time of the next interval, it indicates an overlap.
   - Return false, as overlapping intervals mean that a room is double-booked.
3. If there are no overlaps after the loop, return true, indicating that all meetings can be scheduled without conflicts.

Time Complexity:
O(n log n), where n is the number of intervals. The sorting step takes O(n log n) time.

Space Complexity:
O(1), as we use only a constant amount of extra space.
*/

impl Solution {
    pub fn can_attend_meetings(intervals: Vec<Vec<i32>>) -> bool {
        if intervals.is_empty() {
            return true;
        }

        let mut intervals = intervals;
        intervals.sort_by_key(|x| x[0]);

        for i in 0..intervals.len() - 1 {
            if intervals[i][1] > intervals[i + 1][0] {
                return false; // Overlapping intervals
            }
        }

        true
    }
}
```

133. 16_Intervals/05_Meeting_Rooms_II/0253-meeting-rooms-ii.rs

```
/*
Problem: LeetCode 253 - Meeting Rooms II

Key Idea:
The key idea is to sort the meetings by their start times and use a priority queue (min heap) to track the end times of ongoing meetings. For each meeting, check if there's an available room (i.e., the earliest ending meeting) in the priority queue. If not, allocate a new room. Keep updating the priority queue as meetings start and end. The size of the priority queue represents the minimum number of rooms needed.

Approach:
1. Create two vectors, 'starts' and 'ends', to store the start times and end times of the meetings, respectively.
2. Sort both 'starts' and 'ends' in ascending order.
3. Initialize variables 'rooms' to 0 and 'end_ptr' to 0.
4. Iterate through 'starts':
    - If the current start time is less than the current end time in 'ends', it means a meeting overlaps with an ongoing meeting.
     - Increment 'rooms' as a new room is required.
   - Otherwise, increment 'end_ptr' to indicate that the ongoing meeting has ended.
5. Return 'rooms' as the minimum number of meeting rooms required.

Time Complexity:
O(n log n), where n is the number of meetings. Sorting the 'starts' and 'ends' vectors takes O(n log n) time.

Space Complexity:
O(n), as we use two vectors 'starts' and 'ends' to store the start and end times of meetings.
*/

impl Solution {
    pub fn min_meeting_rooms(intervals: Vec<Vec<i32>>) -> i32 {
        if intervals.is_empty() {
            return 0;
        }

        let mut starts: Vec<i32> = Vec::new();
        let mut ends: Vec<i32> = Vec::new();

        for interval in &intervals {
            starts.push(interval[0]);
            ends.push(interval[1]);
        }

        starts.sort();
        ends.sort();

        let mut rooms = 0;
        let mut end_ptr = 0;

        for start in starts {
            if start < ends[end_ptr] {
                rooms += 1;
            } else {
                end_ptr += 1;
            }
        }

        rooms
    }
```

```
}
```

134. 16_Intervals/06_Minimum_Interval_to_Include_Each_Query/1851-minimum-interv

```
/*
Problem: LeetCode 1851 - Minimum Interval to Include Each Query

Key Idea:
The main idea is to find, for each query, the smallest interval from a sorted list of intervals
that contains that query point. This is achieved by maintaining a BinaryHeap (priority queue) of
intervals that can potentially contain each query point.

Approach:
1. Convert input intervals into a vector of Interval structs and sort them by start points.
2. Initialize a HashMap to store query results.
3. Initialize a BinaryHeap to hold intervals overlapping the current query.
4. Iterate through sorted queries:
   - Add intervals with start points less than or equal to the query to the heap.
   - Find the minimum interval containing the query point:
     - If the top interval in the heap contains the query, store its length in the result map.
     - If the heap is empty, mark the query as not containing any interval.
     - Otherwise, remove the top interval from the heap.
5. Collect results in the original order and return them.

Time Complexity:
O(N * log(N)), due to sorting.

Space Complexity:
O(N + Q), for intervals and sorted_queries.
*/

use std::cmp::Ordering;
use std::collections::{BinaryHeap, HashMap};

#[derive(Debug, PartialEq, Eq)]
struct Interval {
    start: i32,
    end: i32,
}

impl Solution {
    pub fn min_interval(intervals: Vec<Vec<i32>>, queries: Vec<i32>) -> Vec<i32> {
        let mut intervals = intervals
            .into_iter()
            .map(|v| Interval {
                start: v[0],
                end: v[1],
            })
            .collect::<Vec<Interval>>();

        intervals.sort_by(|a, b| a.start.cmp(&b.start));

        let mut result_map = HashMap::new();
        let mut heap = BinaryHeap::new();
        let mut i = 0;

        let mut sorted_queries = queries.clone();
        sorted_queries.sort();

        for query in sorted_queries {
            while i < intervals.len() && intervals[i].start <= query {
                heap.push(&intervals[i]);
                i += 1;
            }
```

```
            loop {
                if let Some(top_interval) = heap.peek() {
                    if top_interval.start <= query && query <= top_interval.end {
                        result_map.insert(query, top_interval.length());
                        break;
                    } else {
                        heap.pop();
                    }
                } else {
                    result_map.insert(query, -1);
                    break;
                }
            }
        }

        queries.into_iter().map(|x| result_map[&x]).collect()
    }
}

impl Ord for Interval {
    fn cmp(&self, other: &Self) -> Ordering {
        if self.length() == other.length() {
            return self.end.cmp(&other.end).reverse();
        }
        return self.length().cmp(&other.length()).reverse();
    }
}

impl PartialOrd for Interval {
    fn partial_cmp(&self, other: &Self) -> Option<Ordering> {
        Some(self.cmp(other))
    }
}

impl Interval {
    fn length(&self) -> i32 {
        self.end - self.start + 1
    }
}
```

135. 17_Math_&_Geometry/01_Rotate_Image/0048-rotate-image.rs

```
/*
Problem: LeetCode 48 - Rotate Image

Key Idea:
The key idea is to perform a series of swaps and rotations to rotate the given square matrix in-place.

Approach:
1. Initialize variables 'n' to the size of the matrix (number of rows or columns) and 'layer' to 0.
2. Use a loop to iterate through each layer from the outermost layer to the innermost layer (if any).
3. Within each layer, use a nested loop to iterate through the elements in the layer:
   a. Swap the four elements in a cycle (top, left, bottom, right).
   b. Repeat this process for all elements in the layer except for the corners.
4. Increment 'layer' and repeat the process until all layers have been rotated.
5. The matrix is now rotated 90 degrees clockwise in-place.

Time Complexity:
O(n^2), where 'n' is the size of the matrix. We need to touch each element once to perform the rotations.

Space Complexity:
O(1), as we perform the rotations in-place without using any extra space.
*/

impl Solution {
    pub fn rotate(matrix: &mut Vec<Vec<i32>>) {
        let n = matrix.len();
        let mut layer = 0;

        while layer < n / 2 {
            for i in layer..n - layer - 1 {
                let top = matrix[layer][i];
                let left = matrix[n - i - 1][layer];
                let bottom = matrix[n - layer - 1][n - i - 1];
                let right = matrix[i][n - layer - 1];

                matrix[layer][i] = left;
                matrix[n - i - 1][layer] = bottom;
                matrix[n - layer - 1][n - i - 1] = right;
                matrix[i][n - layer - 1] = top;
            }

            layer += 1;
        }
    }
}

/*
impl Solution {
    pub fn rotate(matrix: &mut Vec<Vec<i32>>) {
        let n = matrix.len();

        for i in 0..n {
            for j in i+1..n {
                matrix.swap(i, j, j, i);
            }
        }
```

```
        for row in matrix.iter_mut() {
            row.reverse();
        }
    }
}

trait MatrixSwap {
    fn swap(&mut self, i1: usize, j1: usize, i2: usize, j2: usize);
}

impl MatrixSwap for Vec<Vec<i32>> {
    fn swap(&mut self, i1: usize, j1: usize, i2: usize, j2: usize) {
        let tmp = self[i1][j1];
        self[i1][j1] = self[i2][j2];
        self[i2][j2] = tmp;
    }
}
*/
```

136. 17_Math_&_Geometry/02_Spiral_Matrix/0054-spiral-matrix.rs

```
/*
Problem: LeetCode 54 - Spiral Matrix

Key Idea:
The key idea is to simulate the process of traversing the matrix in a spiral order by defining four boundaries: top, bottom, left, and right.

Approach:
1. Initialize variables 'top' to 0, 'bottom' to the last row index, 'left' to 0, and 'right' to the last column index.
2. Initialize an empty vector 'result' to store the spiral order elements.
3. Use a loop to continue until 'top' is less than or equal to 'bottom' and 'left' is less than or equal to 'right'.
4. Inside the loop, iterate from 'left' to 'right' and add the elements in the top row to 'result'.
5. Increment 'top' to exclude the top row from further consideration.
6. Iterate from 'top' to 'bottom' and add the elements in the rightmost column to 'result'.
7. Decrement 'right' to exclude the rightmost column from further consideration.
8. Check if 'top' is still less than or equal to 'bottom' to avoid duplicate traversal.
9. Iterate from 'right' to 'left' and add the elements in the bottom row to 'result'.
10. Decrement 'bottom' to exclude the bottom row from further consideration.
11. Check if 'left' is still less than or equal to 'right' to avoid duplicate traversal.
12. Iterate from 'bottom' to 'top' and add the elements in the leftmost column to 'result'.
13. Increment 'left' to exclude the leftmost column from further consideration.
14. Repeat the loop until 'top' is no longer less than or equal to 'bottom' or 'left' is no longer less than or equal to 'right'.
15. Return the 'result' vector containing the elements in spiral order.

Time Complexity:
O(m * n), where 'm' is the number of rows and 'n' is the number of columns in the matrix. We visit each element once.

Space Complexity:
O(1), as we do not use any additional data structures that scale with the size of the input.
*/

impl Solution {
    pub fn spiral_order(matrix: Vec<Vec<i32>>) -> Vec<i32> {
        if matrix.is_empty() {
            return Vec::new();
        }

        let mut top = 0;
        let mut bottom = matrix.len() as i32 - 1;
        let mut left = 0;
        let mut right = matrix[0].len() as i32 - 1;
        let mut result = Vec::new();

        while top <= bottom && left <= right {
            for i in left..=right {
                result.push(matrix[top as usize][i as usize]);
            }
            top = top.checked_add(1).unwrap();

            for i in top..=bottom {
                result.push(matrix[i as usize][right as usize]);
            }
            right = right.checked_sub(1).unwrap();

            if top <= bottom {
```

```
                for i in (left..=right).rev() {
                    result.push(matrix[bottom as usize][i as usize]);
                }
                bottom = bottom.checked_sub(1).unwrap();
            }

            if left <= right {
                for i in (top..=bottom).rev() {
                    result.push(matrix[i as usize][left as usize]);
                }
                left = left.checked_add(1).unwrap();
            }
        }

        result
    }
}

/*
// Pattern Matching and Enums

enum Direction {
    Up,
    Down,
    Right,
    Left,
}

impl Solution {
    pub fn spiral_order(matrix: Vec<Vec<i32>>) -> Vec<i32> {
        let mut current_direction = Direction::Right;
        let mut current_round = 0;
        let mut current_row = 0;
        let mut current_col = 0;
        let m = matrix.len();
        let n = matrix[0].len();
        let mut result = vec![];

        for _ in 0..m * n {
            result.push(matrix[current_row][current_col]);

            match current_direction {
                Direction::Up => {
                    // Check if we've reached the upper limit
                    if current_row == current_round + 1 {
                        current_col += 1;
                        current_direction = Direction::Right;
                        current_round += 1;
                    } else {
                        current_row -= 1;
                    }
                }
                Direction::Down => {
                    // Check if we've reached the lower limit
                    if current_row + 1 == m - current_round {
                        current_col -= 1;
                        current_direction = Direction::Left;
                    } else {
                        current_row += 1;
                    }
                }
                Direction::Right => {
                    // Check if we've reached the right limit
                    if current_col + 1 == n - current_round {
```

```
                    current_row += 1;
                    current_direction = Direction::Down;
                } else {
                    current_col += 1;
                }
            }
            Direction::Left => {
                // Check if we've reached the left limit
                if current_col == current_round {
                    current_row -= 1;
                    current_direction = Direction::Up;
                } else {
                    current_col -= 1;
                }
            }
        }
    }

    result
  }
}
*/
```

137. 17_Math_&_Geometry/03_Set_Matrix_Zeroes/0073-set-matrix-zeroes.rs

```
/*
Problem: LeetCode 73 - Set Matrix Zeroes

Key Idea:
The key idea is to use the first row and the first column to store information about whether the corresponding row or column should be zeroed.

Approach:
1. Initialize two boolean variables, 'first_row_zero' and 'first_col_zero', to keep track of whether the first row and the first column should be zeroed.
2. Iterate through the matrix starting from the second row and the second column.
3. If a cell in the matrix is zero, set the corresponding cell in the first row and the first column to zero to mark that the row or column should be zeroed.
4. Iterate through the matrix again and set the cells to zero based on the information in the first row and the first column.
5. If 'first_row_zero' is true, zero out the entire first row.
6. If 'first_col_zero' is true, zero out the entire first column.
7. Return the modified matrix.

Time Complexity:
O(m * n), where 'm' is the number of rows and 'n' is the number of columns in the matrix. We visit each cell in the matrix twice.

Space Complexity:
O(1), as we use the first row and the first column of the matrix to store information and do not use any additional data structures.
*/

impl Solution {
    pub fn set_zeroes(matrix: &mut Vec<Vec<i32>>) {
        let mut first_row_zero = false;
        let mut first_col_zero = false;
        let m = matrix.len();
        let n = matrix[0].len();

        // Check if the first row should be zeroed.
        for i in 0..m {
            if matrix[i][0] == 0 {
                first_col_zero = true;
                break;
            }
        }

        // Check if the first column should be zeroed.
        for j in 0..n {
            if matrix[0][j] == 0 {
                first_row_zero = true;
                break;
            }
        }

        // Mark rows and columns to be zeroed using the first row and first column.
        for i in 1..m {
            for j in 1..n {
                if matrix[i][j] == 0 {
                    matrix[i][0] = 0;
                    matrix[0][j] = 0;
                }
            }
        }
```

```
        // Zero out rows based on the information in the first column.
        for i in 1..m {
            if matrix[i][0] == 0 {
                for j in 1..n {
                    matrix[i][j] = 0;
                }
            }
        }

        // Zero out columns based on the information in the first row.
        for j in 1..n {
            if matrix[0][j] == 0 {
                for i in 1..m {
                    matrix[i][j] = 0;
                }
            }
        }

        // Zero out the first row if necessary.
        if first_row_zero {
            for j in 0..n {
                matrix[0][j] = 0;
            }
        }

        // Zero out the first column if necessary.
        if first_col_zero {
            for i in 0..m {
                matrix[i][0] = 0;
            }
        }
    }
}

/*
// Using HashSet

use std::collections::HashSet;

impl Solution {
    pub fn set_zeroes(matrix: &mut Vec<Vec<i32>>) {
        let (rows, cols) = (matrix.len(), matrix[0].len());
        let mut rows_with_zero = HashSet::new();
        let mut cols_with_zero = HashSet::new();

        // Find rows and columns with zeros
        for (r, row) in matrix.iter().enumerate() {
            for (c, &val) in row.iter().enumerate() {
                if val == 0 {
                    rows_with_zero.insert(r);
                    cols_with_zero.insert(c);
                }
            }
        }

        // Set rows with zeros to all zeros
        for &r in &rows_with_zero {
            matrix[r].iter_mut().map(|val| *val = 0).count();
        }

        // Set columns with zeros to all zeros
        for &c in &cols_with_zero {
            for row in matrix.iter_mut() {
```

```
                row[c] = 0;
            }
        }
    }
}
*/
```

138. 17_Math_&_Geometry/04_Happy_Number/0202-happy-number.rs

```
/*
Problem: LeetCode 202 - Happy Number

Key Idea:
The key idea is to use two pointers, one slow and one fast, to determine if the sequence of numbers ends in a cycle. If the sequence ends in a cycle, it means the number is not a happy number.

Approach:
1. Define a function 'get_next' that calculates the sum of the squares of the digits of a number.
2. Initialize two variables, 'slow' and 'fast', to the given number.
3. Use a loop to repeatedly calculate the next number using 'get_next' for both 'slow' and 'fast'.
4. In each iteration, move 'slow' one step and 'fast' two steps.
5. If 'slow' and 'fast' become equal at any point, break the loop as this indicates a cycle.
6. Check if 'slow' is equal to 1. If it is, return true as the number is a happy number.
7. If the loop completes without finding a cycle and 'slow' is not equal to 1, return false as the number is not a happy number.

Time Complexity:
O(log n), where 'n' is the given number. The time complexity is determined by the number of digits in the input number.

Space Complexity:
O(1), as we are using only a constant amount of extra space.
*/

impl Solution {
    pub fn is_happy(n: i32) -> bool {
        fn get_next(n: i32) -> i32 {
            let mut sum = 0;
            let mut n = n;
            while n > 0 {
                let digit = n % 10;
                sum += digit * digit;
                n /= 10;
            }
            sum
        }

        let mut slow = n;
        let mut fast = n;

        loop {
            slow = get_next(slow);
            fast = get_next(get_next(fast));

            if slow == fast {
                break;
            }
        }

        slow == 1
    }
}

/*
impl Solution {
    pub fn is_happy(n: i32) -> bool {
        let mut n = n;
        while n != 1 && n != 4 {
```

```
            let mut sum = 0;
            while n > 0 {
                let d = n % 10;
                sum += d * d;
                n /= 10;
            }
            n = sum;
        }
        n == 1
    }
}
*/
```

139. 17_Math_&_Geometry/05_Plus_One/0066-plus-one.rs

```
/*
Problem: LeetCode 66 - Plus One

Key Idea:
The key idea is to simulate the addition of one to the given number as if it were an integer. This is essentially adding 1 to the last digit and handling any carryover that might occur.

Approach:
1. Start iterating from the least significant digit (the rightmost digit) to the most significant digit (the leftmost digit).
2. Add 1 to the current digit.
3. Check if there is a carryover from the addition. If there is a carryover, set the current digit to 0 and continue to the next digit.
4. If there is no carryover, the addition is complete, and we can return the updated array.
5. If the loop completes and there is still a carryover, it means we need to insert a new digit at the beginning of the array with a value of 1.

Time Complexity:
O(n), where 'n' is the number of digits in the input array. In the worst case, we may need to traverse the entire array.

Space Complexity:
O(1), as we modify the input array in-place and do not use any additional data structures.
*/

impl Solution {
    pub fn plus_one(digits: Vec<i32>) -> Vec<i32> {
        let mut digits = digits;
        let mut carry = 1;

        for i in (0..digits.len()).rev() {
            let sum = digits[i] + carry;
            digits[i] = sum % 10;
            carry = sum / 10;
        }

        if carry > 0 {
            digits.insert(0, 1);
        }

        digits
    }
}
```

140. 17_Math_&_Geometry/06_Pow(x,n)/0050-powx-n.rs

```
/*
Problem: LeetCode 50 - Pow(x, n)

Key Idea:
The key idea is to use the concept of exponentiation by squaring to calculate the power efficiently. Instead of multiplying 'x' by itself 'n' times, we can recursively calculate 'x' raised to 'n/2' and use that result to calculate 'x' raised to 'n'.

Approach:
1. Handle the base cases:
   - If 'n' is 0, return 1 because any number raised to the power of 0 is 1.
   - If 'n' is negative, calculate the reciprocal of 'x' and negate 'n' to make it positive. This allows us to handle negative exponents.
2. Calculate 'x' raised to 'n' by recursively calculating 'x' raised to 'n/2'.
3. Use the result from step 2 to calculate 'x' raised to 'n':
   - If 'n' is even, return the square of the result from step 2.
   - If 'n' is odd, return the square of the result from step 2 multiplied by 'x'.
4. The recursion terminates when 'n' becomes 0.

Time Complexity:
O(log n), where 'n' is the exponent. We reduce the problem size by half in each recursive step.

Space Complexity:
O(log n), as the maximum depth of the recursion stack is proportional to the logarithm of 'n'.
*/

impl Solution {
    pub fn my_pow(x: f64, n: i32) -> f64 {
        if n == 0 {
            return 1.0;
        }

        let mut x = x;
        let mut n = n as i64;

        if n < 0 {
            x = 1.0 / x;
            n = -n;
        }

        let mut result = 1.0;
        let mut current_x = x;

        while n > 0 {
            if n % 2 == 1 {
                result *= current_x;
            }
            current_x *= current_x;
            n /= 2;
        }

        result
    }
}
```

141. 17_Math_&_Geometry/07_Multiply_Strings/0043-multiply-strings.rs

```
/*
Problem: LeetCode 43 - Multiply Strings

Key Idea:
The key idea is to simulate the process of manual multiplication, where we multiply each digit of one number with each digit of the other number and add the results at the appropriate positions. This can be efficiently done using two nested loops.

Approach:
1. Convert the input strings 'num1' and 'num2' into vectors of integers, where each digit is represented as an integer.
2. Initialize a vector 'result' with zeros, with a length equal to the sum of the lengths of 'num1' and 'num2' to store the intermediate results.
3. Use two nested loops to multiply each digit of 'num1' with each digit of 'num2' and add the results at the correct positions in 'result'.
4. After the multiplication, we may have carry values at various positions in 'result', so we need to perform a carry operation to ensure that each digit is within the range [0, 9].
5. Convert the 'result' vector back to a string representation of the multiplied number by removing leading zeros.
6. Return the final result.

Time Complexity:
O(M * N), where M and N are the lengths of the input strings 'num1' and 'num2'. The nested loops perform multiplications for each digit pair.

Space Complexity:
O(M + N), as we use vectors to store the digits of 'num1', 'num2', and 'result', where M and N are the lengths of 'num1' and 'num2'.
*/

impl Solution {
    pub fn multiply(num1: String, num2: String) -> String {
        let num1: Vec<u8> = num1.chars().map(|c| (c as u8) - b'0').collect();
        let num2: Vec<u8> = num2.chars().map(|c| (c as u8) - b'0').collect();
        let len1 = num1.len();
        let len2 = num2.len();

        let mut result: Vec<u8> = vec![0; len1 + len2];

        for i in (0..len1).rev() {
            for j in (0..len2).rev() {
                let product = num1[i] * num2[j];
                let sum = result[i + j + 1] + product;
                result[i + j + 1] = sum % 10;
                result[i + j] += sum / 10;
            }
        }

        // Convert result to string, removing leading zeros
        let mut result_str = String::new();
        let mut leading_zeros = true;

        for digit in result {
            if digit != 0 {
                leading_zeros = false;
            }
            if !leading_zeros {
                result_str.push((digit + b'0') as char);
            }
        }
```

```
        if result_str.is_empty() {
            return "0".to_string();
        }

        result_str
    }
}

/*
impl Solution {
    pub fn multiply(num1: String, num2: String) -> String {
        let len1 = num1.len();
        let len2 = num2.len();
        let mut result_digits = vec![0; len1 + len2];

        for (i, ch1) in num1.chars().rev().enumerate() {
            for (j, ch2) in num2.chars().rev().enumerate() {
                let digit1 = ch1.to_digit(10).unwrap() as i32;
                let digit2 = ch2.to_digit(10).unwrap() as i32;
                let product = digit1 * digit2 + result_digits[i + j];
                result_digits[i + j] = product % 10;
                result_digits[i + j + 1] += product / 10;
            }
        }

        while result_digits.len() > 1 && result_digits.last() == Some(&0) {
            result_digits.pop();
        }

        result_digits.into_iter().rev().map(|d| d.to_string()).collect::<String>()
    }
}
*/
```

142. 17_Math_&_Geometry/08_Detect_Squares/2013-detect-squares.rs

```
/*
Problem: LeetCode 2013 - Detect Squares

Key Idea:
A rectangle requires two pairs of points to share one common point and have equal distances along both axes.

Approach:
1. Create a `DetectSquares` struct to maintain a `points_count` HashMap, which counts the occurrences of each point.
2. In the `add` method, update the count for the given point in the `points_count` HashMap.
3. In the `count` method, iterate through the points in the `points_count` HashMap.
4. For each point `key`, check if it can form a rectangle with the input point `point`. If so, calculate the count of two additional points to complete the rectangle.
5. Multiply these three counts (including `val`, which represents the count of `key`) to get the total count of rectangles formed with `point` as one corner.
6. Add this count to the result and continue iterating.
7. Finally, return the result, which represents the total count of rectangles formed by the given points.

Time Complexity:
O(n), due to count() method.

Space Complexity:
O(n), for using a hashmap.
*/

use std::collections::HashMap;

struct DetectSquares {
    points_count: HashMap<Vec<i32>, i32>,
}

impl DetectSquares {
    fn new() -> Self {
        DetectSquares {
            points_count: HashMap::new(),
        }
    }

    fn add(&mut self, point: Vec<i32>) {
        self.points_count
            .entry(point)
            .and_modify(|count| *count += 1)
            .or_insert(1);
    }

    fn count(&self, point: Vec<i32>) -> i32 {
        let mut result: i32 = 0;

        for (key, val) in self.points_count.iter() {
            if key[0] != point[0]
                && key[1] != point[1]
                && (point[0] - key[0]).abs() == (point[1] - key[1]).abs()
            {
                let count_a = match self.points_count.get(&*vec![key[0], point[1]]) {
                    Some(&count) => count,
                    None => 0,
                };
```

```
                let count_b = match self.points_count.get(&*vec![point[0], key[1]]) {
                    Some(&count) => count,
                    None => 0,
                };

                result += val * count_a * count_b;
            }
        }

        result
    }
}
```

143. 18_Bit_Manipulation/01_Single_Number/0136-single-number.rs

```
/*
Problem: LeetCode 136 - Single Number

Key Idea:
The key idea is to use the bitwise XOR operation to find the single number in the array. XOR has the property that it returns 1 for each bit position where the two input bits are different and 0 for each bit position where they are the same. Therefore, when we XOR all the elements in the array, all the elements that appear twice will cancel each other out, leaving only the single element.

Approach:
1. Initialize a variable 'result' to 0. This variable will store the single number.
2. Iterate through each element 'num' in the array.
3. Update 'result' by XOR-ing it with 'num'.
4. After the loop, 'result' will contain the single number that appears only once in the array.
5. Return 'result'.

Time Complexity:
O(n), where 'n' is the number of elements in the array. We iterate through the array once.

Space Complexity:
O(1), as we use a constant amount of extra space.
*/

impl Solution {
    pub fn single_number(nums: Vec<i32>) -> i32 {
        let mut result = 0;
        for num in nums {
            result ^= num;
        }
        result
    }
}

/*
// One liner

impl Solution {
    pub fn single_number(nums: Vec<i32>) -> i32 {
        nums.iter().fold(0, |acc, x| acc ^ x)
    }
}
*/
```

144. 18_Bit_Manipulation/02_Number_of_1_Bits/0191-number-of-1-bits.rs

```
/*
Problem: LeetCode 191 - Number of 1 Bits

Key Idea:
The key idea is to use bit manipulation. Iterate through the bits of the integer and use bitwise AND operation to check if each bit is 1. Increment a count variable for each 1 bit encountered. Repeat this process until all bits have been checked to count the number of 1 bits in the integer.

Approach:
1. Initialize a variable 'count' to 0. This variable will store the number of set bits.
2. Iterate through each bit of the input integer 'n' from the least significant bit to the most significant bit.
3. To check if a bit is set (1), perform a bitwise AND operation between 'n' and 1 (binary 0001).
4. If the result of the bitwise AND operation is not zero, increment the 'count' variable.
5. Right-shift 'n' by one position to check the next bit.
6. Repeat steps 3-5 until 'n' becomes zero.
7. Return the 'count' as the number of set bits.

Time Complexity:
O(log n), where 'n' is the input integer. We iterate through each bit in the binary representation of 'n'.

Space Complexity:
O(1), as we use a constant amount of extra space.
*/

impl Solution {
    pub fn hammingWeight(n: u32) -> i32 {
        let mut count = 0;
        let mut n = n;

        while n != 0 {
            if n & 1 != 0 {
                count += 1;
            }
            n >>= 1;
        }

        count
    }
}

/*
// short solution

impl Solution {
    pub fn hammingWeight(n: u32) -> i32 {
        n.count_ones() as i32
    }
}
*/
```

145. 18_Bit_Manipulation/03_Counting_Bits/0338-counting-bits.rs

```
/*
Problem: LeetCode 338 - Counting Bits

Key Idea:
The key idea is to use dynamic programming to count the number of set bits (1s) in the binary representation of each number from 0 to the given number 'num'. We can observe certain patterns to optimize this process.

Approach:
1. Initialize a vector 'count' of size 'num + 1' to store the count of set bits for each number from 0 to 'num'.
2. For the base case, set 'count[0] = 0' since the count of set bits for 0 is 0.
3. Iterate through the numbers from 1 to 'num'.
4. To find the count of set bits for a number 'i', we can utilize the result for 'i / 2' (right-shifting by one position) and add 1 if 'i' is odd (i.e., 'i & 1 == 1').
5. Store the result in 'count[i]'.
6. After the loop, 'count' will contain the count of set bits for all numbers from 0 to 'num'.
7. Return 'count' as the result.

Time Complexity:
O(num), where 'num' is the given input number. We iterate through all numbers from 1 to 'num'.

Space Complexity:
O(num), as we use a vector of size 'num + 1' to store the count of set bits for each number.
*/

impl Solution {
    pub fn count_bits(num: i32) -> Vec<i32> {
        let mut count = vec![0; (num + 1) as usize];

        for i in 1..=num {
            count[i as usize] = count[(i / 2) as usize] + (i & 1);
        }

        count
    }
}

/*
impl Solution {
    pub fn count_bits(n: i32) -> Vec<i32> {
        let mut result:Vec<i32> = Vec::new();
        for i in 0..n+1 {
            result.push(i.count_ones() as i32);
        }
        result
    }
}
*/
```

146. 18_Bit_Manipulation/04_Reverse_Bits/0190-reverse-bits.rs

```
/*
Problem: LeetCode 190 - Reverse Bits

Key Idea:
We can iterate through the bits of 'n' from the least significant bit to the most significant bit
and construct the reversed number bit by bit.

Approach:
1. Initialize a variable 'result' to 0. This variable will store the reversed bits.
2. Iterate through 'n' while it's not zero:
   a. Right-shift 'result' by one position to make space for the next bit.
   b. Check if the least significant bit of 'n' is 1 using 'n & 1'.
   c. If it's 1, set the least significant bit of 'result' to 1 using 'result |= 1'.
   d. Right-shift 'n' by one position to process the next bit.
3. Repeat steps 2a-2d until 'n' becomes zero.
4. After the loop, 'result' will contain the reversed bits of 'n'.
5. Return 'result' as the result.

Time Complexity:
O(32) or O(1), since we iterate through a constant number of bits (32 bits in an unsigned
integer).

Space Complexity:
O(1), as we use a constant amount of extra space.
*/

impl Solution {
    pub fn reverse_bits(n: u32) -> u32 {
        let mut result = 0;
        let mut n = n;

        for _ in 0..32 {
            result <<= 1;
            if n & 1 == 1 {
                result |= 1;
            }
            n >>= 1;
        }

        result
    }
}

/*
impl Solution {
    pub fn reverse_bits(mut x: u32) -> u32 {
        x.reverse_bits()
    }
}
*/
```

147. 18_Bit_Manipulation/05_Missing_Number/0268-missing-number.rs

```
/*
Problem: LeetCode 268 - Missing Number

Key Idea:
We can solve this by calculating the sum of all numbers from 0 to 'n' using the formula 'n * (n + 1) / 2' and subtracting the sum of the elements in the array from it.

Approach:
1. Initialize a variable 'expected_sum' to 0. This variable will store the sum of all numbers from 0 to 'n'.
2. Initialize a variable 'actual_sum' to 0. This variable will store the sum of the elements in the input array.
3. Iterate through the input array and add each element to 'actual_sum'.
4. Calculate 'expected_sum' using the formula 'n * (n + 1) / 2', where 'n' is the length of the input array.
5. The missing number is 'expected_sum - actual_sum'.
6. Return the missing number.

Time Complexity:
O(n), where 'n' is the length of the input array. We iterate through the array once to calculate 'actual_sum' and 'expected_sum'.

Space Complexity:
O(1), as we use only a constant amount of extra space.
*/

impl Solution {
    pub fn missing_number(nums: Vec<i32>) -> i32 {
        let n = nums.len() as i32;
        let expected_sum = n * (n + 1) / 2;
        let actual_sum: i32 = nums.iter().sum();

        expected_sum - actual_sum
    }
}

/*
impl Solution {
    pub fn missing_number(nums: Vec<i32>) -> i32 {
        (nums.len() * (nums.len() + 1) / 2) as i32 - nums.iter().sum::<i32>()
    }
}
*/
```

148. 18_Bit_Manipulation/06_Sum_of_Two_Integers/0371-sum-of-two-integers.rs

```
/*
Problem: LeetCode 371 - Sum of Two Integers

Key Idea:
The key idea is to use bitwise operations to perform addition without using the `+` operator. We can use bitwise XOR to perform the addition without carrying, and bitwise AND to calculate the carry.

Approach:
1. Initialize two variables, 'a' and 'b', to represent the two integers to be added.
2. Loop while 'b' is not zero:
   a. Calculate the sum without carrying by performing 'a XOR b' and store it in 'a'.
   b. Calculate the carry by performing '(a AND b) << 1' and store it in 'b'.
3. After the loop, 'a' will contain the result, which is the sum of the two integers.

Time Complexity:
O(1), as the number of iterations in the loop is constant (32 bits for integers).

Space Complexity:
O(1), as we use only a constant amount of extra space.
*/

impl Solution {
    pub fn get_sum(a: i32, b: i32) -> i32 {
        let mut x = a;
        let mut y = b;

        while y != 0 {
            let carry = x & y;
            x = x ^ y;
            y = carry << 1;
        }

        x
    }
}

/*
impl Solution {
    pub fn get_sum(a: i32, b: i32) -> i32 {
        return [a, b].iter().sum();
    }
}
*/
```

149. 18_Bit_Manipulation/07_Reverse_Integer/0007-reverse-integer.rs

```
/*
Problem: LeetCode 7 - Reverse Integer

Key Idea:
We can repeatedly extract the last digit of 'x' and append it to the result while updating 'x' by
removing the last digit.

Approach:
1. Initialize a variable 'result' to 0. This variable will store the reversed integer.
2. Iterate while 'x' is not zero:
   a. Calculate the last digit of 'x' by taking 'x % 10'.
   b. Update 'result' by multiplying it by 10 and adding the last digit.
   c. Update 'x' by removing the last digit using 'x / 10'.
3. After the loop, 'result' will contain the reversed integer.
4. Check for overflow by comparing 'result' with the maximum and minimum values of a 32-bit
integer.
5. Return 0 if 'result' overflows, otherwise return 'result'.

Time Complexity:
O(log(x)), where 'x' is the input integer. We iterate through the digits of 'x'.

Space Complexity:
O(1), as we use only a constant amount of extra space.
*/

impl Solution {
    pub fn reverse(x: i32) -> i32 {
        let mut result = 0;
        let mut x = x;

        while x != 0 {
            let digit = x % 10;
            // Check for overflow
            if result > i32::MAX / 10 || (result == i32::MAX / 10 && digit > 7) {
                return 0;
            }
            if result < i32::MIN / 10 || (result == i32::MIN / 10 && digit < -8) {
                return 0;
            }
            result = result * 10 + digit;
            x /= 10;
        }

        result
    }
}

/*
// One liner

impl Solution {
    pub fn reverse(x: i32) -> i32 {
        x.abs()
            .to_string()
            .chars()
            .rev()
            .collect::<String>()
            .parse::<i32>()
            .map(|r| r * x.signum())
            .unwrap_or(0)
```

```
    }
}
*/
```