Implement a function that checks whether a given string is a palindrome or not.

```
fn is_palindrome(s: &str) -> bool {
    let reversed = s.chars().rev().collect::<String>();
    s == reversed
}

fn main() {
    let test_str = "racecar";
    if is_palindrome(test_str) {
        println!("{} is a palindrome.", test_str);
    } else {
        println!("{} is not a palindrome.", test_str);
    }
}
```

Given a sorted array of integers, implement a function that returns the index of the first occurrence of a given number.

```
fn first_occurrence(arr: &[i32], target: i32) -> Option<usize> {
    let mut low = 0;
    let mut high = arr.len() - 1;
    let mut result = None;

    while low <= high {
        let mid = low + (high - low) / 2;
        if arr[mid] == target {
            result = Some(mid);
            high = mid - 1; // Look for the first occurrence on the left side
        } else if arr[mid] < target {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    result
}

fn main() {
    let arr = [1, 2, 3, 4, 5, 5, 5, 6, 7, 8];
```

```
    let target = 5;

    match first_occurrence(&arr, target) {

        Some(index) => println!("First occurrence of {} is at index {}", target,
index),

        None => println!("{} not found in the array", target),

    }

}
```

Given a string of words, implement a function that returns the shortest word in the string.

```
fn shortest_word(sentence: &str) -> Option<&str> {

    sentence.split_whitespace().min_by_key(|word| word.len())

}


fn main() {

    let sentence = "The quick brown fox jumps over the lazy dog";

    match shortest_word(sentence) {

        Some(shortest) => println!("The shortest word is: {}", shortest),

        None => println!("No words found in the sentence"),

    }

}
```

Implement a function that checks whether a given number is prime or not.

```
fn is_prime(n: u64) -> bool {
    if n <= 1 {
        return false;
    }
    // Iterate from 2 to the square root of n
    for i in 2..=(n as f64).sqrt() as u64 {
        if n % i == 0 {
            return false; // n is divisible by i, hence not prime
        }
    }
    true // If no divisor found, n is prime
}

fn main() {
    let num = 17; // Change this to test different numbers
    if is_prime(num) {
        println!("{} is prime", num);
    } else {
        println!("{} is not prime", num);
    }
}
```

Given a sorted array of integers, implement a function that returns the median of the array.

```
fn find_median(arr: &[i32]) -> f64 {
    let len = arr.len();
    if len % 2 == 0 {
        // If the length of the array is even
        let mid = len / 2;
        // Calculate the average of the two middle elements
        (arr[mid - 1] + arr[mid]) as f64 / 2.0
    } else {
        // If the length of the array is odd
        arr[len / 2] as f64 // Return the middle element
    }
}

fn main() {
    let arr = vec![1, 2, 3, 4, 5]; // Example sorted array
    let median = find_median(&arr);
    println!("Median of the array is: {}", median);
}
```

Implement a function that finds the longest common prefix of a given set of strings.

```
fn longest_common_prefix(strs: Vec<String>) -> String {
    if strs.is_empty() {
        return String::new(); // If the input vector is empty, return an empty string
    }

    let mut prefix = strs[0].clone(); // Initialize the prefix with the first string

    for s in strs.iter().skip(1) {
        // Iterate through the remaining strings
        while !s.starts_with(&prefix) {
            // Remove characters from the prefix until it's a prefix of the current string
            prefix.pop();
        }
    }

    prefix // Return the longest common prefix
}

fn main() {
    let strings = vec![
```

```
        "flower".to_string(),
        "flow".to_string(),
        "flight".to_string(),
    ]; // Example set of strings
    let result = longest_common_prefix(strings);
    println!("Longest common prefix: {}", result);
}
```

Implement a function that returns the kth smallest element in a given array.

```
fn kth_smallest(arr: &mut [i32], k: usize) -> Option<i32> {
    // Sort the array in ascending order
    arr.sort();

    // Check if k is within the bounds of the array
    if k > 0 && k <= arr.len() {
        Some(arr[k - 1]) // Return the kth smallest element
    } else {
        None // Return None if k is out of bounds
    }
}

fn main() {
```

```
    let mut array = [3, 1, 4, 1, 5, 9, 2, 6, 5];
    let k = 3; // Example value of k

    match kth_smallest(&mut array, k) {
        Some(result) => println!("The {}th smallest element is: {}", k, result),
        None => println!("Invalid value of k or empty array"),
    }
}
```

Given a binary tree, implement a function that returns the maximum depth of the tree.

```
// Definition for a binary tree node.
#[derive(Debug, PartialEq, Eq)]
pub struct TreeNode {
    pub val: i32,
    pub left: Option<Box<TreeNode>>,
    pub right: Option<Box<TreeNode>>,
}

impl TreeNode {
    #[inline]
    pub fn new(val: i32) -> Self {
        TreeNode { val, left: None, right: None }
```

```
    }
}

fn max_depth(root: Option<Box<TreeNode>>) -> i32 {
    match root {
        None => 0, // Base case: empty tree has depth 0
        Some(node) => {
            // Recursively find the maximum depth of the left and right subtrees
            let left_depth = max_depth(node.left);
            let right_depth = max_depth(node.right);

            // Return the maximum depth of the left or right subtree, plus 1 for the current node
            1 + left_depth.max(right_depth)
        }
    }
}

fn main() {
    // Example binary tree:
    //     3
    //    / \
    //   9  20
```

```
    //     / \
    //    15   7
    let root = Some(Box::new(TreeNode {
        val: 3,
        left: Some(Box::new(TreeNode::new(9))),
        right: Some(Box::new(TreeNode {
            val: 20,
            left: Some(Box::new(TreeNode::new(15))),
            right: Some(Box::new(TreeNode::new(7))),
        })),
    }));

    println!("Maximum depth of the binary tree: {}", max_depth(root));
}
```

Reverse a string in Rust

```
fn reverse_string(s: &str) -> String {
    // Convert the input string into a sequence of characters
    let mut chars: Vec<char> = s.chars().collect();

    // Use Rust's reverse method to reverse the characters in-place
    chars.reverse();
```

```
    // Convert the reversed characters back into a string
    let reversed_string: String = chars.iter().collect();

    reversed_string
}

fn main() {
    let input_string = "Hello, world!";
    let reversed_string = reverse_string(input_string);
    println!("Original string: {}", input_string);
    println!("Reversed string: {}", reversed_string);
}
```

Check if a number is prime in Rust

```
fn is_prime(n: u64) -> bool {
    // Numbers less than 2 are not prime
    if n < 2 {
        return false;
    }

    // Check if n is divisible by any integer from 2 to sqrt(n)
    let sqrt_n = (n as f64).sqrt() as u64;
    for i in 2..=sqrt_n {
```

```
        if n % i == 0 {
            return false; // n is divisible by i, hence not prime
        }
    }

    true // n is not divisible by any integer from 2 to sqrt(n), hence prime
}

fn main() {
    // Test cases
    let numbers = vec![2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31];
    for &num in &numbers {
        if is_prime(num) {
            println!("{} is prime", num);
        } else {
            println!("{} is not prime", num);
        }
    }
}
```

Merge two sorted arrays in Rust

```
fn merge_sorted_arrays(arr1: &[i32], arr2: &[i32]) -> Vec<i32> {
    let mut merged = Vec::with_capacity(arr1.len() + arr2.len());
```

```
let (mut i, mut j) = (0, 0);


while i < arr1.len() && j < arr2.len() {
    if arr1[i] <= arr2[j] {
        merged.push(arr1[i]);
        i += 1;
    } else {
        merged.push(arr2[j]);
        j += 1;
    }
}

// Add remaining elements from arr1, if any
while i < arr1.len() {
    merged.push(arr1[i]);
    i += 1;
}

// Add remaining elements from arr2, if any
while j < arr2.len() {
    merged.push(arr2[j]);
    j += 1;
```

```
    }

    merged
}

fn main() {
    let arr1 = vec![1, 3, 5, 7, 9];
    let arr2 = vec![2, 4, 6, 8, 10];
    let merged = merge_sorted_arrays(&arr1, &arr2);
    println!("Merged array: {:?}", merged);
}
```

Find the maximum subarray sum in Rust

```
fn max_subarray_sum(arr: &[i32]) -> i32 {
    let mut max_sum = arr[0];
    let mut current_sum = arr[0];

    for &num in arr.iter().skip(1) {
        current_sum = num.max(current_sum + num);
        max_sum = max_sum.max(current_sum);
    }

    max_sum
}

fn main() {
```

```
    let arr = vec![-2, 1, -3, 4, -1, 2, 1, -5, 4];
    let max_sum = max_subarray_sum(&arr);
    println!("Maximum subarray sum: {}", max_sum);
}
```