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

```
fn is_palindrome(s: &str) -> bool {
    let normalized: String = s.chars()
        .filter(|c| c.is_alphanumeric())
        .map(|c| c.to_lowercase().next().unwrap())
        .collect();

    let reversed: String = normalized.chars().rev().collect();

    normalized == reversed
}

fn main() {
    let test_str = "A man, a plan, a canal, Panama";
    println!("Is the string \"{}\" a palindrome? {}", test_str, is_palindrome(test_str));
}
```

0. 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() as isize - 1;
    let mut result = None;

    while low <= high {
        let mid = (low + high) / 2;
        if arr[mid as usize] == target {
            result = Some(mid as usize);
```

```
            high = mid - 1;  // continue searching in the left half
        } else if arr[mid as usize] < target {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }

    result
}

fn main() {
    let sorted_array = [1, 2, 2, 2, 3, 4, 5];
    let target = 2;
    match first_occurrence(&sorted_array, target) {
        Some(index) => println!("The first occurrence of {} is at index {}", target, index),
        None => println!("{} is not in the array", target),
    }
}
```

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

```
fn shortest_word(s: &str) -> Option<&str> {
    s.split_whitespace()
        .min_by_key(|word| word.len())
}

fn main() {
    let test_str = "This is a sample string with some short and some reallylongwords";
    match shortest_word(test_str) {
        Some(word) => println!("The shortest word is: {}", word),
```

```
        None => println!("The input string is empty"),
    }
}
```

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

```
fn is_prime(n: u32) -> bool {
    if n <= 1 {
        return false;
    }
    if n == 2 {
        return true;
    }
    if n % 2 == 0 {
        return false;
    }
    let sqrt_n = (n as f64).sqrt() as u32;
    for i in (3..=sqrt_n).step_by(2) {
        if n % i == 0 {
            return false;
        }
    }
    true
}

fn main() {
    let number = 29;
    println!("Is the number {} prime? {}", number, is_prime(number));
}
```

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

```
fn median(arr: &[i32]) -> f64 {
    let len = arr.len();
    if len == 0 {
        panic!("Array must not be empty");
    }

    if len % 2 == 1 {
        arr[len / 2] as f64
    } else {
        let mid1 = arr[len / 2 - 1];
        let mid2 = arr[len / 2];
        (mid1 + mid2) as f64 / 2.0
    }
}

fn main() {
    let sorted_array_odd = [1, 2, 3, 4, 5];
    let sorted_array_even = [1, 2, 3, 4, 5, 6];

    println!("The median of {:?} is {}", sorted_array_odd, median(&sorted_array_odd));
    println!("The median of {:?} is {}", sorted_array_even, median(&sorted_array_even));
}
```

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

```
fn longest_common_prefix(strs: &[String]) -> String {
    if strs.is_empty() {
        return String::new();
    }
```

```
    let mut prefix = strs[0].clone();

    for s in strs.iter().skip(1) {
        while !s.starts_with(&prefix) {
            prefix.pop(); // Remove the last character
            if prefix.is_empty() {
                return String::new();
            }
        }
    }

    prefix
}

fn main() {
    let strs = vec![
        "flower".to_string(),
        "flow".to_string(),
        "flight".to_string(),
    ];

    let lcp = longest_common_prefix(&strs);
    println!("The longest common prefix is: {}", lcp);
}
```

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

```
fn partition(arr: &mut [i32], left: usize, right: usize) -> usize {
    let pivot = arr[right];
    let mut i = left;
```

```
    for j in left..right {
        if arr[j] <= pivot {
            arr.swap(i, j);
            i += 1;
        }
    }
    arr.swap(i, right);
    i
}

fn quickselect(arr: &mut [i32], left: usize, right: usize, k: usize) -> i32 {
    if left == right {
        return arr[left];
    }

    let pivot_index = partition(arr, left, right);

    if k == pivot_index {
        arr[k]
    } else if k < pivot_index {
        quickselect(arr, left, pivot_index - 1, k)
    } else {
        quickselect(arr, pivot_index + 1, right, k)
    }
}

fn kth_smallest(arr: &mut [i32], k: usize) -> i32 {
    if k >= arr.len() {
        panic!("k is out of bounds");
    }
```

```
    quickselect(arr, 0, arr.len() - 1, k)
}

fn main() {
    let mut arr = [7, 10, 4, 3, 20, 15];
    let k = 3;
    let kth = kth_smallest(&mut arr, k - 1); // k-1 because k is 1-based
    println!("The {}-th smallest element is: {}", k, kth);
}
```

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

```
#[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 {
        Some(node) => {
            let left_depth = max_depth(node.left);
            let right_depth = max_depth(node.right);
            1 + std::cmp::max(left_depth, right_depth)
        },
        None => 0,
    }
}

fn main() {
    let mut root = Box::new(TreeNode::new(1));
    root.left = Some(Box::new(TreeNode::new(2)));
    root.right = Some(Box::new(TreeNode::new(3)));
    root.left.as_mut().unwrap().left = Some(Box::new(TreeNode::new(4)));
    root.left.as_mut().unwrap().right = Some(Box::new(TreeNode::new(5)));

    println!("The maximum depth of the tree is: {}", max_depth(Some(root)));
}
```

9. Reverse a string in Rust.

```
fn reverse_string(s: &str) -> String {
    s.chars().rev().collect()
}

fn main() {
    let original = "Hello, world!";
    let reversed = reverse_string(original);
    println!("Original: {}", original);
```

```
    println!("Reversed: {}", reversed);
}
```

10. Check if a number is prime in Rust

```
fn is_prime(n: u32) -> bool {
    if n <= 1 {
        return false;
    }
    if n == 2 {
        return true;
    }
    if n % 2 == 0 {
        return false;
    }
    let mut divisor = 3;
    while (divisor * divisor) <= n {
        if n % divisor == 0 {
            return false;
        }
        divisor += 2; // Increment by 2 to skip even numbers
    }
    true
}

fn main() {
    let num = 17;
    if is_prime(num) {
        println!("{} is a prime number.", num);
    } else {
        println!("{} is not a prime number.", num);
```

```
    }
}
```

11. 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;
        }
    }

    merged.extend_from_slice(&arr1[i..]);
    merged.extend_from_slice(&arr2[j..]);

    merged
}

fn main() {
    let arr1 = [1, 3, 5, 7, 9];
    let arr2 = [2, 4, 6, 8, 10];
    let merged = merge_sorted_arrays(&arr1, &arr2);

    println!("Merged array: {:?}", merged);
}
```

12. 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 = [-2, 1, -3, 4, -1, 2, 1, -5, 4];
    let max_sum = max_subarray_sum(&arr);
    println!("Maximum subarray sum: {}", max_sum);
}
```