1. Palindrome check

fn is\_palindrome(s: &str) -> bool {

let s = s.chars().collect::<Vec<char>>();

let n = s.len();

for i in 0..n / 2 {

if s[i] != s[n - 1 - i] {

return false;

}

}

true

}

1. Finding first occurrence in sorted array (binary search)

fn first\_occurrence(arr: &[i32], x: 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] == x {

result = Some(mid);

high = mid - 1;

} else if arr[mid] < x {

low = mid + 1;

} else {

high = mid - 1;

}

}

result

}

1. Find shortest string in of string of words

fn find\_shortest\_word(s: &str) -> Option<&str> {

let words: Vec<&str> = s.split\_whitespace().collect();

let mut shortest\_word: Option<&str> = None;

for word in words {

match shortest\_word {

Some(shortest) => {

if word.len() < shortest.len() {

shortest\_word = Some(word);

}

}

None => {

shortest\_word = Some(word);

}

}

}

shortest\_word

}

1. Prime or not

fn is\_prime(n: u64) -> bool {

if n <= 1 {

return false;

}

for i in 2..=(n as f64).sqrt() as u64 {

if n % i == 0 {

return false;

}

}

true

}

1. Median of sorted array

fn find\_median(arr: &[i32]) -> f64 {

let len = arr.len();

let mid = len / 2;

if len % 2 == 0 {

(arr[mid - 1] as f64 + arr[mid] as f64) / 2.0

} else {

arr[mid] as f64

}

}

1. Longest common prefix:

fn longest\_common\_prefix(strings: Vec<String>) -> String {

let len = strings.len();

if len == 0 {

return String::new();

}

let mut common\_prefix = strings[0].clone();

for i in 1..len {

while !strings[i].starts\_with(&common\_prefix) {

common\_prefix.pop();

}

}

common\_prefix

}

1. Kth smallest array

fn kth\_smallest(arr: &[i32], k: usize) -> i32 {

let mut sorted\_arr = arr.to\_vec();

sorted\_arr.sort();

sorted\_arr[k - 1]

}

1. Maximum depth of a binary tree

use std::cmp;

struct TreeNode {

val: i32,

left: Option<Box<TreeNode>>,

right: Option<Box<TreeNode>>,

}

impl TreeNode {

fn new(val: i32) -> Self {

TreeNode {

val,

left: None,

right: None,

}

}

}

fn max\_depth(root: Option<Box<TreeNode>>) -> i32 {

match root {

None => 0,

Some(node) => {

let left\_depth = max\_depth(node.left);

let right\_depth = max\_depth(node.right);

cmp::max(left\_depth, right\_depth) + 1

}

}

}

1. String reversal

fn reverse\_string(s: String) -> String {

let mut chars: Vec<char> = s.chars().collect();

let mut left = 0;

let mut right = chars.len() - 1;

while left < right {

chars.swap(left, right);

left += 1;

right -= 1;

}

chars.into\_iter().collect()

}

1. Repeated Prime or not

fn is\_prime(n: u64) -> bool {

if n <= 1 {

return false;

}

for i in 2..=(n as f64).sqrt() as u64 {

if n % i == 0 {

return false;

}

}

true

}

1. Merge two sorted arrays in Rust

fn merge\_sorted\_arrays(nums1: Vec<i32>, nums2: Vec<i32>) -> Vec<i32> {

let mut result = 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] {

result.push(nums1[i]);

i += 1;

} else {

result.push(nums2[j]);

j += 1;

}

}

while i < nums1.len() {

result.push(nums1[i]);

i += 1;

}

while j < nums2.len() {

result.push(nums2[j]);

j += 1;

}

result

}

1. Find the maximum subarray sum in Rust

fn max\_subarray\_sum(nums: Vec<i32>) -> i32 {

let mut max\_sum = nums[0];

let mut current\_sum = nums[0];

for i in 1..nums.len() {

current\_sum = std::cmp::max(current\_sum + nums[i], nums[i]);

max\_sum = std::cmp::max(max\_sum, current\_sum);

}

max\_sum

}