Leetcode in Rust

Table of Contents

# 1 Rust in a Nutshell

## 1.1 Why Rust?

## 1.2 Cargo

## 1.3 Cargo Doc

## 1.4 Crates

## 1.5 Basic Data Structures

### 1.5.1 Sequences

#### 1.5.1.1 Vec

#### 1.5.1.2 VecDeque

#### 1.5.1.3 LinkedList

### 1.5.2 Maps

#### 1.5.2.1 HashMap

#### 1.5.2.2 BTreeMap

### 1.5.3 Sets

#### 1.5.3.1 HashSet

#### 1.5.3.2 BTreeSet

### 1.5.4 Other

#### 1.5.4.1 BinaryHeap

## 1.6 Basic Algorithms

## 1.7 Other Useful things

## 1.8 Regex

## 1.9 Derive Macros

## 1.10 Counting in O(1) space with slices

# 2 Macros for Rust

## 2.1 test!

Unlike C and C++, a testing framework is built into rust. We can create our own tests by creating a mod block and letting cargo know that we want to test it.

Let’s say we create this function:

src/add.rs

fn add(a: i32, b: i32) -> i32 {  
 a + b  
}

We can test it at the bottom of the file:

src/add.rs

...  
#[cfg(test)]  
mod test {  
 use super::\*;  
  
 #[test]  
 fn add\_one\_and\_one() {  
 assert\_eq!(add(1, 1), 2);  
 }  
  
 #[test]  
 fn add\_one\_and\_two() {  
 assert\_eq!(add(1, 2), 3);  
 }  
}

Macros let us reduce most of the boilerplate:

src/lib.rs

#[macro\_export]  
macro\_rules! test {  
 ($($name:ident: $left:expr, $right:expr,)\*) => {  
 #[cfg(test)]  
 mod test {  
 use super::\*;  
 $(  
 #[test]  
 fn $name() {  
 assert\_eq!($left, $right);  
 }  
 )\*  
 }  
 }  
}

Test can then be called like so:

src/add.rs

test! {  
 add\_one\_to\_one: add(1, 1), 2,  
 add\_one\_to\_two: add(1, 2), 3,  
}

# 3 Introductory

## 3.1 Contains Duplicate

### 3.1.1 Problem

Given an integer array nums, return true if any value appears at least twice in the array, and return false if every element is distinct.

### 3.1.2 Intuition

### 3.1.3 Test Cases

[] == false  
[1] == false  
[1,1] == true  
[1,2,3] == false  
[1,2,1] == true

### 3.1.4 Using Sets

If a slice of numbers is the same length as the set of its numbers, we know that the slice **only contains** unique numbers. With this, we can find the solution to the problem:

### 3.1.5 Complexity

O(n) time, O(n) space. We take O(n) time to convert the slice into the HashSet, and the HashSet takes O(n) space as well.

### 3.1.6 Answer

use crate::\*;  
use std::collections::HashSet;  
  
test! {  
 test\_1: contains\_duplicate(&[1, 2, 3, 1]), true,  
 test\_2: contains\_duplicate(&[1, 2, 3, 4]), false,  
 test\_3: contains\_duplicate(&[1]), false,  
}  
  
/// Returns `true` if nums contains a duplicate, `false otherwise.`  
pub fn contains\_duplicate(nums: &[i32]) -> bool {  
 let num\_len = nums.len();  
 let s: HashSet<&i32> = HashSet::from\_iter(nums.iter());  
 s.len() != num\_len  
}

# 4 Trees

## 4.1 Maximum Path through a Binary Tree

use crate::\*;  
use std::cmp::max;  
  
/// Finds the maximum path sum through a binary tree.  
pub fn max\_path\_sum(root: BSTNode) -> i32 {  
 let mut max\_so\_far = i32::MIN;  
 fn helper(node: &BSTNode, max\_so\_far: &mut i32) -> i32 {  
 match node {  
 Some(n) => {  
 let val = n.borrow().val;  
 let l = max(0, helper(&n.borrow().left, max\_so\_far));  
 let r = max(0, helper(&n.borrow().right, max\_so\_far));  
 \*max\_so\_far = max(\*max\_so\_far, val + l + r);  
 val + max(l, r)  
 }  
 None => 0,  
 }  
 }  
 helper(&root, &mut max\_so\_far);  
 max\_so\_far  
}  
  
test! {  
 test\_1: max\_path\_sum(btree![1,2,3]), 6,  
 test\_2: max\_path\_sum(btree![-10, 9, 20, null, null, 15, 7]), 42,  
}

## 4.2 Validate Binary Search Tree

use crate::\*;  
  
pub fn is\_valid\_bst(root: BSTNode) -> bool {  
 fn helper(node: &BSTNode, possible\_min: i64, possible\_max: i64) -> bool {  
 if let Some(n) = node {  
 let borrowed = n.borrow();  
 let left = &borrowed.left;  
 let right = &borrowed.right;  
 let val: i64 = borrowed.val.into();  
 if val >= possible\_min && val <= possible\_max {  
 helper(&left, possible\_min, val) && helper(&right, val, possible\_max)  
 } else {  
 false  
 }  
 } else {  
 true  
 }  
 }  
 helper(&root, i64::MIN, i64::MAX)  
}  
  
test! {  
 test\_1: is\_valid\_bst(btree![2, 1, 3]), true,  
 test\_2: is\_valid\_bst(btree![5, 1, 3]), false,  
}

## 4.3 Same Tree

### 4.3.1 Test Cases

test! {  
 test\_1: is\_same\_tree(btree![1,2,3], btree![1,2,3]), true,  
 test\_2: is\_same\_tree(btree![1,2,3,4], btree![1,2,3]), false,  
}

use crate::\*;  
  
test! {  
 test\_1: is\_same\_tree(btree![1,2,3], btree![1,2,3]), true,  
 test\_2: is\_same\_tree(btree![1,2,3,4], btree![1,2,3]), false,  
}  
  
/// Calculates if two binary search trees have the same values.  
/// In this question, there are four possible cases:  
/// 1. Both left and right point to a `None` node. In this case, return true.  
/// 2. Either left or right points to a `None` node, but the other has a value. In which case, return false.  
/// 3. Both left and right point to a node with a value, but the values are different. return false.  
/// 4. Both left and right point to nodes with the same value. Return true.  
/// Afterwards  
pub fn is\_same\_tree(p: BSTNode, q: BSTNode) -> bool {  
 fn same(p: &BSTNode, q: &BSTNode) -> bool {  
 match (p, q) {  
 (Some(left), Some(right)) => {  
 let left = left.borrow();  
 let right = right.borrow();  
 left.val == right.val  
 && same(&left.left, &right.left)  
 && same(&left.right, &right.right)  
 }  
 (None, None) => true,  
 (None, \_) | (\_, None) => false,  
 }  
 }  
 same(&p, &q)  
}

