Leetcode in Rust

Contents

0.1 Rust in a Nutshell

0.1.1 Why Rust?

0.1.2 Cargo

0.1.3 Cargo Doc

0.1.4 Crates

0.1.5 Basic Data Structures

0.1.5.1 Sequences

0.1.5.1.1 Vec

0.1.5.1.2 VecDeque

0.1.5.1.3 LinkedList

0.1.5.2 Maps

0.1.5.2.1 HashMap

0.1.5.2.2 BTreeMap

0.1.5.3 Sets

0.1.5.3.1 HashSet

0.1.5.3.2 BTreeSet

0.1.5.4 Other

0.1.5.4.1 BinaryHeap

0.1.6 Basic Algorithms

0.1.7 Other Useful things

0.1.8 Regex

0.1.9 Derive Macros

0.1.10 Counting in O(1) space with slices

0.2 Macros for Rust

0.2.1 `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,
}
```

0.3 Introductory

0.3.1 Contains Duplicate

0.3.1.1 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.

0.3.1.2 Intuition

0.3.1.3 Test Cases

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

0.3.1.4 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:

0.3.1.5 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.

0.3.1.6 Answer

```rust
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
}
```

0.4 Trees

0.4.1 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,
}
```

0.4.2 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,
}
```

0.4.3 Same Tree

0.4.3.1 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)
}
```