

# Leetcode in Rust

# Contents

<b>1</b>	<b>Rust in a Nutshell</b>	<b>3</b>
1.1	Why Rust? . . . . .	3
1.2	Cargo . . . . .	3
1.3	Cargo Doc . . . . .	3
1.4	Crates . . . . .	3
1.5	Basic Data Structures . . . . .	3
1.5.1	Sequences . . . . .	3
1.5.2	Maps . . . . .	3
1.5.3	Sets . . . . .	3
1.5.4	Other . . . . .	4
1.6	Basic Algorithms . . . . .	4
1.7	Other Useful things . . . . .	4
1.8	Regex . . . . .	4
1.9	Derive Macros . . . . .	4
1.10	Counting in $O(1)$ space with slices . . . . .	4
<b>2</b>	<b>Macros for Rust</b>	<b>5</b>
2.1	test! . . . . .	5
<b>3</b>	<b>Introductory</b>	<b>7</b>
3.1	Contains Duplicate . . . . .	7
3.1.1	Problem . . . . .	7
3.1.2	Intuition . . . . .	7
3.1.3	Test Cases . . . . .	7
3.1.4	Using Sets . . . . .	7
3.1.5	Complexity . . . . .	8
3.1.6	Answer . . . . .	8
<b>4</b>	<b>Trees</b>	<b>9</b>
4.1	Maximum Path through a Binary Tree . . . . .	9

4.2	Validate Binary Search Tree . . . . .	10
4.3	Same Tree . . . . .	10

# Chapter 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**

# Chapter 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,
}

```

# Chapter 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 std::collections::HashSet;

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
}
```

# Chapter 4

## Trees

### 4.1 Maximum Path through a Binary Tree

```
type Node = Option<Rc<RefCell<TreeNode>>>>;

pub fn max_path_sum(root: Node) -> i32 {
    let mut max_so_far = i32::MIN;
    fn helper(node: &Node, 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
}
```

## 4.2 Validate Binary Search Tree

```
type Node = Option<Rc<RefCell<TreeNode>>>>;

pub fn is_valid_bst(root: Node) -> bool {
    fn helper(node: &Node, 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)
}
```

## 4.3 Same Tree

```
type Node = Option<Rc<RefCell<TreeNode>>>>;

pub fn is_same_tree(p: Node, q: Node) -> bool {
    fn is_same(p: &Node, q: &Node) -> 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,
        }
    }
}
```

```
    is_same(&p, &q)  
}
```

