# Leetcode in Rust

# Contents

# 1 Rust in a Nutshell

## 1.1 Why Rust?

### 1.1.1 Rust vs. C, C++, Objective-C

### 1.1.2 Rust vs. Java, Kotlin, C#, Swift

### 1.1.3 Rust vs. JavaScript, Python, Ruby

### 1.1.4 Rust vs. Haskell, OCaml, F#, Elixir

## 1.2 Ownership and References

Rust's most unique feature is its ownership system, which can be summed up thusly:

1. Each value in Rust has a variable that's called its owner.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value will be dropped.

Any data can be referred to by either:

1. A single mutable reference
2. Many immutable references

This is useful because having more than one *active* mutable reference can cause issues.

To demonstrate that, in this code below, we create a vector with one element, 5. Then we take an immutable reference to it, but mutably append the reference to it.

```
#include <vector>
using namespace std;

int main() {
  vector<int> vec = {5};
  const int &first = vec.front();
  for (int i = 0; i < 10; ++i) vec.push_back(first);
  for (const int item: vec) cout << item << '\n';
}
```

This causes iterator invalidation in C++, where `first` will eventually point to unowned memory, and will be pushed to vec. Vec will then point to unowned memory. This causes `Undefined Behavior`, where the program will behave non-deterministically.

Rust actually stops this in its tracks. First, we convert the above program to rust:

```rust
fn main() {
  let mut v = vec![5];
  let first = &v[0];
  for _ in 1..10 { v.push(*first); }
  for p in v { println!("{}", p); }
}
```

And if compiled, the compiler gives this error:

```
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable
 --> <source>:4:20
  |
3 |   let first = &v[0];
  |                - immutable borrow occurs here
4 |   for _ in 1..10 { v.push(*first); }
  |                    ^^^^^^^------^
  |                    |      |
  |                    |      immutable borrow later used here
  |                    mutable borrow occurs here

error: aborting due to previous error

For more information about this error, try `rustc --explain E0502`.
```

The error points us to the exact line and the error, which says that a mutable reference on `v` cannot exist at the same time as an immutable borrow on `v`. This could allow for iterator invalidation, and the compiler catches this for us.

If you're more used to a language with garbage collection, you might recall that there is a distinction between `value types` and `reference types`.

In JavaScript:

Numbers are value types so mutating the variable passed into a function doesn't have an effect on the variable.

```
> a = 10;
10
> b = 20;
20
> const add = (a, b) => a + b;
undefined
> add(a, b);
30
> a
10 // a is still 10
```

But for arrays, which are objects, and thus reference types:

```
> x = [];
[]
> x
[]
> const append = (array, item) => array.push(item);
undefined
> append(x, 10);
1
> x
[ 10 ] // x is now [ 10 ]
```

This is meant to ease programming, but it can be hard to remember which types are value types. (For example, strings in JavaScript are heap allocated, but are treated as value types).

In Rust, functions can take ownership of values by moving values into the function:

```
fn main() {
  let v = vec![5];
  fn push_and_print(mut v: Vec<i32>) {
    v.push(10);
    println!("{:?}", v);
  }
  push_and_print(v);
  // println!("{:?}", v); // This now causes a compiler error, as v is not in scope
}
```

If the last line is uncommented:

```
error[E0382]: borrow of moved value: `v`
 --> <source>:8:20
  |
```

```
2 |   let v = vec![5];
  |       - move occurs because `v` has type `Vec<i32>`, which does not implement the `Copy`
...
7 |   push_and_print(v);
  |                  - value moved here
8 |   println!("{:?}", v); // This now causes a compiler error, since v has been moved into
  |                    ^ value borrowed here after move

error: aborting due to previous error

For more information about this error, try `rustc --explain E0382`.
```

Functions can take a mutable reference, which does not transfer ownership:

```rust
fn main() {
  let mut v = vec![5];
  fn push_and_print(v: &mut Vec<i32>) {
    v.push(10);
    println!("{:?}", v);
  } // the mut reference is returned to the outer scope here
  push_and_print(&mut v);
  println!("{:?}", v); // this no longer causes a compiler error
}
```

Or they can take an immutable reference, which does not transfer ownership and disallows mutating the data behind the reference:

```rust
fn main() {
  let v = vec![5];
  fn push_and_print(v: &Vec<i32>) {
    // v.push(10); // mutating v is no longer allowed
    println!("{:?}", v);
  } // the reference is returned to the outer scope here
  push_and_print(&v);
  println!("{:?}", v); // no compiler error. V is in scope.
}
```

Value types (which in Rust implement a trait called `Copy`) can be treated similarly, except they are not moved into functions.

They are copied into a function by default:

```rust
fn main() {
  let num = 10;
```

```
  fn add_and_print(mut num: i32) {
    num += 10;
    println!("{}", num);
  }
  add_and_print(num); // 20
  println!("{}", num); // 10
}
```

They can be treated as a mutable reference:

```
fn main() {
  let mut num = 10;
  fn add_and_print(num: &mut i32) {
    *num += 10;
    println!("{}", num);
  } // the reference is returned to the outer scope here
  add_and_print(&mut num); // 20
  println!("{}", num); // 20 // no compiler error. num is in scope
}
```

They can be treated as an immutable reference:

```
fn main() {
  let num = 10;
  fn add_and_print(num: &i32) {
    // *num += 10; // mutation is no longer allowed
    println!("{}", num);
  } // the reference is returned to the outer scope here
  add_and_print(&num); // 20
  println!("{}", num); // 20 // no compiler error. num is in scope
}
```

All in all, Rust allows finer grained control of the lifetimes of variables, and functions show if they take ownership of variables that are passed in, or if they take a reference, or if they mutate the underlying data.

## 1.3 Cargo

## 1.4 Cargo Doc

## 1.5 Testing

## 1.6 Crates

## 1.7 Basic Types

### 1.7.1 Bool

### 1.7.2 Char

### 1.7.3 Floats

### 1.7.4 Integers

### 1.7.5 Saturing Operations

### 1.7.6 Unsigned Integers

### 1.7.7 Tuples

### 1.7.8 Structs

### 1.7.9 Enums

### 1.7.10 Unit Type

## 1.8 Pattern Matching

## 1.9 Error Handling at Compile Time

### 1.9.1 Option

### 1.9.2 Error

## 1.10 Impl



## 1.11 Traits

## 1.12 Iterators

To create a vector, one can use this syntax:

```
let mut v = Vec::new();
```

The main operation of a vector is `push`, which appends one element to the end of the vector.

```
let mut v = Vec::new();
v.push(5); // The vector now looks like this: [5].
```

Pushing has an amortized time complexity of O(1). Pushing to a vector has a worst case time complexity of O(n), because vectors dynamically grow.

If a vector is full, and you push back to a vector, the vector must do the following:

1. Allocate a buffer that is twice the size of its previous buffer
2. Copy over its current items to the new buffer
3. Add the new element to the buffer.
4. Free the previous buffer.

The first, third, and fourth step all take constant time, but copying over every item from the previous to the new buffer takes linear (O(n)) time. That being said, as long as you avoid this case as much as possible (which you can mitigate by doubling the buffer size every time) the time complexity for pushing to a vector is constant time on average.

If you want to check the contents of a vector at any given time, you can print it:

```
let mut v = Vec::new();
v.push(5);
println!("{:?}", v);
```

### 1.13.2 VecDeque

A VecDeque is a Doubly-Ended Queue implemented as a Vector. A VecDeque allows for O(1) appends and pops from either end of the queue, which basically makes it a stack and a queue in one data structure.

### 1.13.3 LinkedList

A LinkedList is a doubly linked list.

There are two Key-Value data structures in the Rust Standard Library.

#### 1.13.4 HashMap

A HashMap is an Unordered Map. That means that getting, inserting, updating, or deleting a value from this data structure is done in O(1) time.

#### 1.13.5 BTreeMap

A BTreeMap is an Ordered Map. That means that getting, inserting, updating, or deleting a value from this data structure is done in O(log n) time.

#### 1.13.6 HashSet

#### 1.13.7 BTreeSet

#### 1.13.8 BinaryHeap

### 1.14 Algorithms

#### 1.14.1 Binary Search

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

### 1.16 Regex

### 1.17 Derive Macros

### 1.18 Smart Pointers

# 2 Macros for Rust

## 2.1 A macro for testing

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:

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

We can test it at the bottom of the file:

```
#[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:

```
#[macro_export]
macro_rules! test {
    ($($name:ident: $left:expr, $right:expr,)*) => {
        #[cfg(test)]
        mod test {
        use super::*;
            $(
                #[test]
                fn $name() {
                    assert_eq!($left, $right);
```

```
                }
            )*
        }
    }
}
```

Our tests can then be rewritten like so:

```
test! {
  add_one_to_one: add(1, 1), 2,
  add_one_to_two: add(1, 2), 3,
}
```

And running them gives us this result:

```
$ cargo test
running 2 tests
test test::add_one_and_one ... ok
test test::add_one_and_two ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished
↪  in 0.01s
```

# 3 How to Approach Problems

Much has been said about how to become a better problem solver[1]. Here we'll go over some tips and tricks to solve a hard problem by using some of these techniques.

## 3.1 A Plan of Attack

1. Build an intuition about the problem. What should the code return?
2. Write some test cases. Note any edge cases your code should take care of.
3. Start writing out the code, being wary of edge cases.
4. Refactor your code. How can it be improved?

Let's go over our plan step by step, using an example problem called `Jewels and Stones`:

> You're given strings jewels representing the types of stones that are jewels, and stones representing the stones you have. Each character in stones is a type of stone you have. You want to know how many of the stones you have are also jewels.
>
> Letters are case sensitive, so "a" is considered a different type of stone from "A".

> Example 1:
>
> Input: jewels = "aA", stones = "aAAbbbb" Output: 3
>
> Example 2:
>
> Input: jewels = "z", stones = "ZZ" Output: 0

### 3.1.1 Build Intuition

First, let's jot down some notes about the problem. It says: `We want to know how many of the stones you have are also jewels`. This means that we want to return a count of our jewels. A count is going to be a unsigned integer. We can imagine that our return type would be some unsigned integer type, like `u32`.

We can start with that:

```
fn jewels_and_stones(/* TODO */) -> u32 { /* TODO */ }
```

Next, let's take note of the two inputs, which are `given [as] strings`. We can assume we are given one string for the jewels and one for the stones.

```
fn jewels_and_stones(jewels: String, stones: String) -> u32 { /* TODO */ }
```

The problem also notes that we want to return the number of jewels in our collection of stones, and that every character of jewels is a jewel, and every character of stones is a stone.

Let's reduce the problem to something easier. Let's say that instead of having a collection of stone(s), we have just one stone and one jewel. Does this make the problem easier?

It should. We now only need to check if the stone is a jewel, and return our counter at the end.

```
fn jewels_and_stones(jewel: char, stone: char) -> u32 {
  let mut count = 0;
  if jewel == stone {
    count += 1;
  }
  count
}
```

What if we make it so we have one stone but many jewels? What would we do?

Well for our one stone, we would want to check every jewel to make sure that it is a jewel, and return the count of jewels we have.

```
fn jewels_and_stones(jewels: String, stone: char) -> u32 {
  let mut count = 0;
  for jewel in jewels.chars() {
    if jewel == stone {
      count += 1;
    }
  }
  count
}
```

What happens if we have many stones but one jewel? We do the opposite, where every stone that counts as a jewel increments our count by one.

```
fn jewels_and_stones(jewel: char, stones: String) -> u32 {
  let mut count = 0;
  for stone in stones.chars() {
    if jewel == stone {
      count += 1;
    }
  }
```

```
  count
}
```

Now that we have some intuition about how to solve simpler problems, we'll start by writing test cases for this problem:

## 3.2 Writing test cases

We'll start off by writing test cases for our simplified problems:

If the jewels and stones have a length of one, either they are the same or not. If they are the same, this function should return 1. If not, this function should return 0.

```
assert_eq!(jewels_and_stones("a".to_string(), "a".to_string()), 1);
assert_eq!(jewels_and_stones("a".to_string(), "b".to_string()), 0);
```

If there is one jewel, we iterate through our stones and increment our count every time we find a jewel.

```
assert_eq!(jewels_and_stones("a".to_string(), "aac".to_string()), 2);
assert_eq!(jewels_and_stones("a".to_string(), "xyz".to_string()), 0);
```

Otherwise, if there's one stone, then we check every jewel to see if our stone is a jewel.

```
assert_eq!(jewels_and_stones("abc".to_string(), "a".to_string()), 1);
assert_eq!(jewels_and_stones("xyz".to_string(), "a".to_string()), 0);
```

Finally, if there's more than one jewel and more than one stone, for each stone, we check if it is a jewel in our set of jewels.

```
assert_eq!(jewels_and_stones("abc".to_string(), "cxx".to_string()), 1);
assert_eq!(jewels_and_stones("xyz".to_string(), "xxa".to_string()), 2);
```

## 3.3 Writing Code

Now we can begin writing some code to tackle our original problem:

We have an intuition that for every stone, we want to check if it is a jewel. To do this, we have to iterate through all the jewels, and compare our stone to it. If they're the same, we can increment the count.

```rust
fn jewels_and_stones(jewels: String, stones: String) -> u32 {
  let mut count = 0;
  for stone in stones.chars() {
    for jewel in jewels.chars() {
      if stone == jewel {
        count += 1;
        break;
      }
    }
  }
  count
}
```

This turns out to pass the tests outlined above, but it has some problems. Time to refactor!

## 3.4 Refactoring

When refactoring, let's discuss some things we can do to improve our code:

Our code is very concise. There's not much that can be done to improve its readability, which is a good thing. That being said, it can have a slow runtime. If we say the length of stones is N and the length of jewels is M, the runtime of our code grows in O(N*M) (polynomial time). We should be able to do better. But how?

```rust
// O(N*M)
fn jewels_and_stones(jewels: String, stones: String) -> u32 {
  let mut count = 0;
  for stone in stones.chars() { // O(N)
    for jewel in jewels.chars() { // O(M): Wouldn't it be nice if this was O(1)?
      if stone == jewel {
        count += 1;
        break;
      }
    }
  }
  count
}
```

We have a nested for loop, which contributes the slow runtime. Maybe we could represent either jewels or stones in a different fashion, and get rid of a for loop? Would there be a different way of representing jewels that would make this easier? Maybe a data structure that has O(1) time for if it contains an item?

We can use a set for this:

So our solution turns into this:

```
// O(N)
fn jewels_and_stones(jewels: String, stones: String) -> u32 {
  let mut count = 0;
  let jewels_set: HashSet<char> = HashSet::from_iter(jewels.chars());
  for stone in stones.chars() { // O(N)
    if jewels_set.contains(&stone) {
      count += 1;
    }
  }
  count
}
```

And we get down from O(N*M) to O(N) time.

# 4 Trees

## 4.1 Validate Binary Search Tree

```rust
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,
}
```

## 4.2 Same Tree

### 4.2.1 Problem

> Given the roots of two binary trees p and q, write a function to check if they are the same or not. Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

### 4.2.2 Intuition

This question tests your knowledge of recursion. To do so, start off with the base case:

- What happens when left is None and right has a value? Return false.
- What happens when left has a value and right is None? Return false.
- What happens when both left and right are None? Return true.
- What happens when left and right have different values? Return false.
- What happens when left and right have the same values? Test their left and right nodes for equality as well.

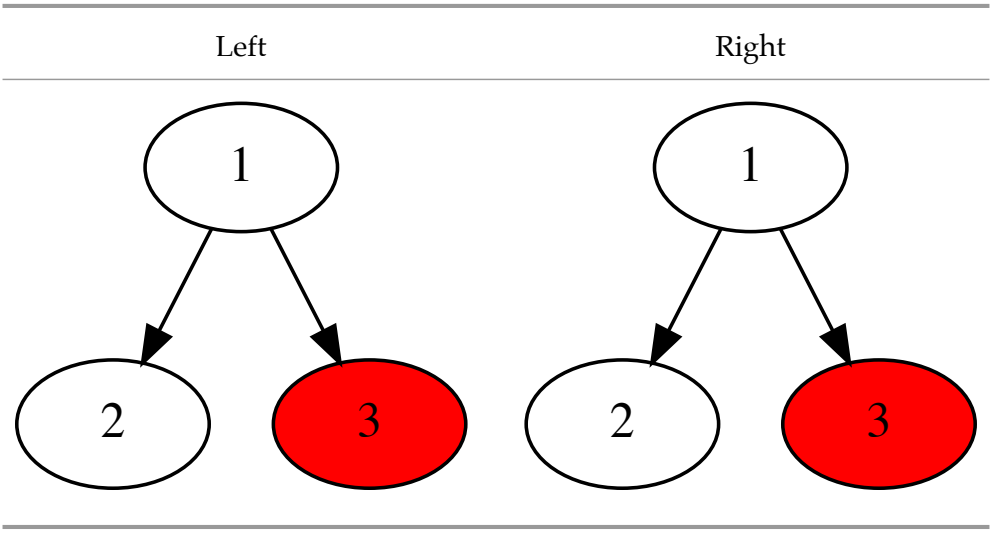
### 4.2.3 Test Cases

```
test! {
    test_1: is_same_tree(btree![], btree![]), true,
    test_2: is_same_tree(btree![1], btree![]), false,
    test_3: is_same_tree(btree![1,2,3], btree![1,2,3]), true,
    test_4: is_same_tree(btree![1,2,3,4], btree![1,2,3]), false,
}
```

### 4.2.4 Answer

```
/// 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. Both left and right point to nodes with the same value. Continue recursing
 ↪  through both
///    trees left and right subtrees.
/// 3. For any other case, return false.
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,
            _ => false,
```

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




## 4.3 Maximum Path through a Binary Tree

```rust
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,
}
```

## References

1. Polya, G. *How to solve it*. (Paperback; Princeton University Press, 1971).