# Assignment 7: Simple Rust program

## The Collatz Sequence — *Functional* Rust

## Overview

In this assignment you will implement two functions in Rust that compute properties of the Collatz sequence. The goal is not just to make the functions work — it is to make them work **without any mutable variables**, therefore without any loops, and without any arrays or strings. Pure numeric computation, pure functional style. The subset of Rust we discussed where "mut" does not appear in the source text.

This is a longish assignment description for what will be a little code. The goal again is to get you to demonstrate familiarity with making and running Rust programs, and using the functional subset.

You may use AI tools to ask about Rust syntax and concepts. You may not ask an AI tool to write or complete your functions for you. The distinction matters: "How do I write a recursive function in Rust?" is a syntax question. "Write me a collatz_length function" is not.

## The Collatz Sequence

The Collatz sequence starts from any positive integer n and applies a simple rule repeatedly:

| | |
|---|---|
| **If n is even** | `n → n / 2` |
| **If n is odd** | `n → 3n + 1` |

You keep applying the rule until you reach 1. For example, starting from 6:

```
6 → 3 → 10 → 5 → 16 → 8 → 4 → 2 → 1
That sequence has 9 steps (counting the starting number).
```

**The Collatz conjecture** states that this sequence always eventually reaches 1, for any positive integer you start with. This has been verified for all numbers up to at least 2^68. Nobody has ever proven it — or disproven it. Your program will verify it for all numbers up to one million.

## What You Must Implement

You are given a starter file (see below). You must implement exactly two functions:

### Function 1: collatz_length

```rust
fn collatz_length(n: u64) -> u64
```

Returns the number of steps in the Collatz sequence starting from n, counting n itself as step 1. The sequence ends when it reaches 1.

**Known answer:** `collatz_length(27) = 112` — the sequence from 27 takes 112 steps before reaching 1.

### Function 2: longest_collatz

```rust
fn longest_collatz(limit: u64) -> u64
```

Returns the starting number below limit that produces the longest Collatz sequence. If there is a tie, return any of the tied values.

**Known answer:** `longest_collatz(1_000_000) = 837799` — starting from 837,799 produces a sequence of 525 steps.

## Constraints

> These are not suggestions. They are requirements. Code that violates them will not receive full credit even if it produces correct output.

- **No let mut.** No mutable variables anywhere in your solution. The compiler will enforce this if you declare everything with let.
- **No loops.** No for, while, or loop constructs. Use recursion instead.  Tail recursion.
- **No arrays, vectors, or strings.** This assignment is purely numeric. You should not need any collection types.
- **Use u64 for all integers.** Intermediate Collatz values can grow large even for modest starting numbers. u32 will overflow. u64 is required.
- **No global variables or static state.** All state must live in function parameters or return values.

## Starter File

Save this as main.rs and compile with: rustc main.rs && ./main

**Rust — starter file (main.rs)**

```
fn main() {
    // Known-answer checks — these will panic if your functions are wrong
    assert_eq!(collatz_length(1),  1,  "length of 1 should be 1");
    assert_eq!(collatz_length(6),  9,  "length of 6 should be 9");
    assert_eq!(collatz_length(27), 112, "length of 27 should be 112");
    println!("collatz_length checks passed.");

    let answer = longest_collatz(1_000_000);
    assert_eq!(answer, 837799, "longest below 1M should be 837799");
    println!("longest_collatz(1_000_000) = {}", answer);
    println!("All checks passed.");
}

fn collatz_length(n: u64) -> u64 {
    todo!()
}

fn longest_collatz(limit: u64) -> u64 {
    todo!()
}
```

The todo!() macro compiles cleanly and panics at runtime with the message "not yet implemented". This means you can compile your program immediately and get one function working at a time. Start by getting collatz_length right — longest_collatz depends on it.

## Hints and Approach

### For collatz_length

Think about the base case and the recursive case. The base case is n == 1 — what should you return? The recursive case applies the rule and calls itself. The no-loops constraint makes recursion the natural choice here.

A match expression works well for selecting between the even and odd cases. You may ask an AI: “What is the syntax for a match expression in Rust?”

### For longest_collatz

You need to find the number in 1..limit that maximizes collatz_length. Without a loop, you have the option (as we did in Elixir/Erlang) of recursion:

1. Recursion: write a helper function that tries each number from 1 to limit, keeping track of the best so far as a parameter.

Recall that tail recursion is space efficient.  It uses one stack frame whereas other recursion might need millions of frames.

## AI Tool Policy

You may ask an AI tool about Rust syntax, concepts, and error messages. Examples of acceptable questions:

- "What is the syntax for a recursive function in Rust?"
- "What does the % operator do in Rust?"
- "How do I convert a u64 to check if it is even?"
- "What does this compiler error mean?"
- "What is max_by_key and how does it work?"

You may not ask an AI tool to implement your functions. Examples of unacceptable prompts:

- "Write me a collatz_length function in Rust"
- "Implement longest_collatz without using loops"
- "Complete this starter file for me"

## Discussion Question

> "Your collatz_length function calls itself potentially hundreds of times for a single input. What happens to the call stack each time it recurses? Is there a risk here that would not exist if you had used a loop instead? Does Rust do anything to help with this?"

Be prepared to answer this in a short written response or in conversation after the exercise. The answer connects to tail recursion, stack overflow, and why Rust's design makes certain tradeoffs that languages like Haskell do not.