Advent of Code 2025 Day 02

Deval Deliwala

Part 1

> Given a string of integer ranges, separated by commas:
>
> `11-22,95-115,998-1012,1188511880-1188511890,222220-222224,`
> `1698522-1698528,446443-446449,38593856-38593862,565653-565659,`
> `824824821-824824827,2121212118-2121212124,`
>
> the goal is to find all integers within each range made only of some sequence of digits repeated twice. In other words, integers that could be periodic with period 2. These integers are *"invalid"*.
>
> For example, in the range `998-1012`, the integer `1010` is invalid.
>
> We return the sum of all invalid integers.

Naive Implementation

Given an inclusive $[\min, \max]$, we iterate over all numbers inside the range and convert them to a `String`. If the length of the string is odd, it can't be periodic with period 2. If the length of the string is even, we split the string in the middle, and equate the left and right hand sides.

If the left and right hand sides are equal, the integer contained in the string is invalid. Hence we add it to an accumulating sum, and return this sum after the iterating over all ranges.

Therefore this algorithm is $O\left(\sum_{m=1}^{n} k(m)\right)$ where n is the number of ranges provided, and $k(m)$ returns the number of integers contained in the mth range.

We will divide this algorithm into two parts:
1. Parsing through the input `String` and collecting all the given $[\min, \max]$ ranges.
2. Iterating through all ranges, finding invalid integers, and summing them together.

```rust
use std::fs::File;
use std::io::{BufRead, BufReader, Error, ErrorKind};
use crate::_2025::DAY2_FILE; // the .txt file containing ranges


/// Given the text line as a [String], which is a list of ranges
/// `245284-286195, ...` separated by commas, this outputs a buffer
/// containing `(min, max)` bounds for each range.
///
/// Arguments:
/// - line: [String] - input text line of ranges separated by commas.
///
/// Returns:
/// - [Vec]<([usize], [usize])> of 2-tuples for min and max bounds for every range.
fn get_ranges(line: String) -> Result<Vec<(usize, usize)>, Error> {
    line
        // good practice, in case trailing comma at EOL
        .split_terminator(',')
        .map(|chunk| {
            let chunk = chunk.trim();
            let (lower, upper) = chunk
                .split_once('-')
                .ok_or_else(|| Error::new(ErrorKind::InvalidInput, "Expected `min-
max` integer bounds"))?;

            let min = lower.trim().parse().map_err(|_|
Error::new(ErrorKind::InvalidInput, "Expected an integer"))?;
            let max = upper.trim().parse().map_err(|_|
Error::new(ErrorKind::InvalidInput, "Expected an integer"))?;

            Ok((min, max))
        })
        .collect()
}


pub fn run_pt1() -> Result<usize, Error> {
    let mut buff_reader = BufReader::new(File::open(DAY2_FILE)?);
    let mut string_buf = String::new();
    buff_reader.read_line(&mut string_buf)?;

    let mut sum = 0;
    let ranges = get_ranges(string_buf)?;
    for &(lower, upper) in ranges.iter() {
        for number in lower..=upper {
            // allocates a String buffer for every number
            // crazy slowdown
            let num_str = number.to_string();
            let num_len = num_str.len();

            // unable to have period 2
            if num_len % 2 != 0 {
                continue;
            }

            // split at center and compare
            let mid_idx = num_len / 2;
            let (left, right) = num_str.split_at(mid_idx);
            if left == right {
                sum += number;
            }
        }
    }
    Ok(sum)
}
```

Analysis

This code is readable and clean and takes **142.15ms** on average. There are two things that jump out for optimization:

1. We build and *push* to a `Vec` to store all the ranges, *and afterwards* iterate through that buffer.

We could have avoided initializing the `Vec` (a heap allocated buffer) and just iterated through the ranges in the `String` directly: finding invalid integers and summing them in **one pass** instead of two separated ones.

Additionally, this line:

```rust
buff_reader.read_line(&mut string_buf)?;

// later parsing
.split_terminator(',')
.map(...)
```

takes time proportional to the input length, call it L. Therefore, while this algorithm elegantly separates function responsibilities, the total runtime is actually $O(L) + O\left(\sum_{m=1}^{n} k(m)\right)$. And from a hardware standpoint, creating the buffer and pushing to it is also expensive.

2. Do we even need a pass to check through all integers in a range? Is it possible to calculate what all invalid integers in a range are or their sum beforehand?

Optimized Implementation

Consider a $2k$-digit long *"invalid"* integer x. As x can be periodic with period two, let y denote the digit subsequence that makes up x. For example,

$$x = 123123, \quad \text{then} \quad y = 123, \quad k = 3. \tag{1}$$

Numerically,

$$\begin{aligned} x &= 123 \cdot 10^k + 123 \\ &= y(10^k + 1). \end{aligned} \tag{2}$$

So inside a range $[L, U]$, we want all y such that:

$$L \leq y(10^k + 1) \leq U \quad \Rightarrow \quad \left\lceil \frac{L}{10^k + 1} \right\rceil \leq y \leq \left\lfloor \frac{U}{10^k + 1} \right\rfloor. \tag{3}$$

Then x is $y \mid \mid y$ periodic and within $[L, U]$. y must also be k digits, so $10^{k-1} \leq y \leq 10^k - 1$. The intersection of these two bounds isolates y further. Let $[L', U']$ be the intersection "clip" of these two bounds.

Therefore, the total sum of all invalid x is

$$\sum x = (10^k + 1) \sum_{y=L'}^{U'} y, \tag{4}$$

summing over Equation 2 in the $[L', U']$ range. We can also avoid iterating over $\sum_{L'}^{U'} y$ if we use Gauss' formula:

$$\begin{aligned} \text{From} \quad \sum_{y=1}^{n} y = \frac{n(n+1)}{2} \quad \Longrightarrow \quad \sum_{y=L'}^{U'} y &= \sum_{y=1}^{U'} y - \sum_{y=1}^{L'-1} y \\ &= \left(\frac{U'(U'+1)}{2}\right) - \left(\frac{(L'-1)(L'-1+1)}{2}\right) \\ &= \frac{(U'^2 + U') - (L'^2 - L')}{2} \\ &= \frac{(U'+L')(U'-L') + (U'+L')}{2} \\ &= \frac{(U'+L')(U'-L'+1)}{2} \end{aligned} \tag{5}$$

This problem reduced to iterating over every range $[L', U']$, and accumulating

$$(10^k + 1) \cdot \frac{(U'+L')(U'-L'+1)}{2}, \tag{6}$$

in **one pass**.

```rust
use std::io::{BufRead, BufReader, Error, ErrorKind};
use std::fs::File;
use std::time::Instant;
use crate::_2025::DAY2_FILE;


/// Calculates the sum of invalid integers within `[l, u]`.
///
/// Arguments:
/// - l: &[str] - lower integer bound as a &[str].
/// - r: &[str] - upper integer bound as a &[str].
///
/// Returns:
/// - Ok([u64]) - sum of invalid integers.
/// - [Error]   - if invalid input given.
fn sum_range_pt1(l: &str, u: &str) -> Result<u64, Error> {
    let l = l.trim();
    let u = u.trim();
    let mut sum: u64 = 0;

    let num_digits_l = l.len();
    let num_digits_u = u.len();

    // lower bound for k
    // (num_digits_l + 1) / 2
    let kmin = num_digits_l.div_ceil(2);
    // upper bound for k
    let kmax = num_digits_u / 2;

    let u: u64 = u.parse().map_err(|_| Error::new(ErrorKind::InvalidInput, "Expected
an integer"))?;
    let l: u64 = l.parse().map_err(|_| Error::new(ErrorKind::InvalidInput, "Expected
an integer"))?;
    for k in kmin..=kmax {
        let factor = 10u64.pow(k as u32) + 1;

        // y bounds from inequality
        let y_lo = l.div_ceil(factor);
        let y_hi = u / factor;

        // y must be k-digit
        let k_lo = 10u64.pow((k - 1) as u32);
        let k_hi = 10u64.pow(k as u32) - 1;

        // clip
        let lprime = y_lo.max(k_lo);
        let uprime = y_hi.min(k_hi);

        if lprime <= uprime {
            // gauss
            sum += factor * ((uprime + lprime) * (uprime - lprime + 1))/2;
        }
    }
    Ok(sum)
}


pub fn run_pt1() -> Result<u64, Error> {
    let mut buff_reader = BufReader::new(File::open(DAY2_FILE)?);
    let mut string_buf = String::new();

    // convert .txt to String
    buff_reader.read_line(&mut string_buf)?;
    let mut string_buf = string_buf.as_str().trim();

    let mut sum = 0;
    while !string_buf.is_empty() {
        // get (l, r) from a string of the form "l-r, ..."
        let (l, rest) = string_buf
            .split_once('-')
            .ok_or_else(|| Error::new(ErrorKind::InvalidInput, "Expected a `min-
max, ...` input"))?;
        let (u, rest) = match rest.split_once(',') {
            Some((u, rest)) => (u, rest),
            None            => (rest, ""),
        };

        let range_sum = sum_range_pt1(l, u)?;
        sum += range_sum;

        // iteratively partitioning string to contain ranges leftover
        string_buf = rest;
    }

    Ok(sum)
}
```

This code takes **78.29µs**, which is **3095**× faster than the naive implementation.

Part 2

> Now, an ID is invalid if it is made only of some sequence of digits repeated at least twice. So, 12341234 (1234 two times), 123123123 (123 three times), 1212121212 (12 five times), and 1111111 (1 seven times) are all invalid IDs.

Naive implementation

Now the invalid integers are not just periodic with period 2, but with any period.

Though an invalid integer of length n can have any period, we still know the length of the repeated subsequence of digits, k, must divide n. But we don't know whether k is 1, 2, or any number for that matter.

Given an inclusive $[\min, \max]$, we iterate over all numbers in this range. For each number, we compute its length, n, and calculate all proper divisors $\{k_1, k_2, \ldots\}$ of n. We iterate through all possible k_i and determine if k_i produces a periodic string that matches the given number. If yes, we add it to a running sum. If no k_i work, we continue to the next number.

To determine if a k_i works, we look at an example string: `s = "121212"`. Here we see $k = 2$, and $n = 6$. It's critical to notice given an index i, `s[i] = s[i + k] = s[i + 2k]`. This is just modular arithmetic.

We can iterate over all indices $i \in [0, 6)$ and see if `s[i] == s[i % k]`. If, for any index, this equivalence fails, we move onto the next potential divisor k. If a k passes for all indices i, then the number is periodic, and we add it to a running sum.

```rust
/// Returns a [Vec] containing all proper divisors of a positive integer `n`.
fn get_divisors(n: usize) -> Vec<usize> {
    let mut divs = Vec::new();

    if n == 1 {
        return divs;
    }

    let mut i = 1;
    // only need to go until sqrt(n)
    while i * i <= n {
        if n % i == 0 {
            // main divisor
            divs.push(i);

            // pair divisor
            let q = n / i;
            if q != i && q < n {
                divs.push(q);
            }
        }

        i += 1;
    }

    divs
}

pub fn run_pt2() -> Result<usize, Error> {
    let mut buff_reader = BufReader::new(File::open(DAY2_FILE)?);
    let mut string_buf = String::new();
    buff_reader.read_line(&mut string_buf)?;

    let mut sum = 0;
    let ranges = get_ranges(string_buf)?;

    // preallocate a String buffer to reuse for all numbers
    // with enough capacity
    let mut num_buffer = String::with_capacity(21);
    for &(lower, upper) in ranges.iter() {
        for number in lower..=upper {
            // reuse buffer
            num_buffer.clear();
            write!(&mut num_buffer, "{}", number)
                .map_err(|_| Error::new(ErrorKind::InvalidInput, "Expected an
integer"))?;

            let num_len = num_buffer.len();
            let bytes = num_buffer.as_bytes();
            let divisors = get_divisors(num_len);
            for &p in divisors.iter() {
                let mut passed = true;

                // only iterate from p..num_len
                // because i == i % p trivially for i < p
                // hence s[i] == s[i % p] trivially for i < p
                for idx in p..num_len {
                    if bytes[idx] != bytes[idx % p] {
                        passed = false;
                        break;
                    }
                }

                if passed {
                    sum += number;
                    break;
                }
            }
        }
    }

    Ok(sum)
}
```

This code is elegant and readable, but its also crazy slow. The function has four nested loops, one coming from `.get_divisors()`. It takes **271.74ms** on average.