Advent of Code 2025 Day 03

Deval Deliwala

Part 1

> Given a sequence of lines called banks:
>
> ```
> 9876543211111111
> 8111111111111119
> 2342342342342278
> 8181819111122111
> ```
>
> find the largest possible two-digit number that exists within each bank. Return the sum of all two-digit numbers found from every bank without rearranging order. For example:
>
> In **98**7654321111111, the largest number is **98**. In 8181819 **9** 1111 **2** 111, the largest number is **92**. Across all four banks above, the sum is 357.

Naive Implementation

The challenge is to find the two ordered digits that form the largest number. Greedily, maximizing the tens place is a good start.

Given a bank "**818181911112111**", we want the largest number – 9 – to be on the left. Hence we place 9 in the tens place. Then we find the largest number *after* 9, which is 2, and place it in the ones place, producing 92. However, if the largest number *is the last digit*, then it must be in the ones place. Then, we choose the largest number that precedes it for the tens place.

Iterating the above procedure over all banks and accumulating every max number yields the result.

```rust
use std::fs::File;
use std::io::{BufReader, BufRead, Error, ErrorKind};
use std::time::Instant;
use crate::y2025::DAY3_FILE;


/// Finds the maximum number in a given [String]
/// containing digits 1-9.
fn find_max_in_bank(bank: &str) -> Result<(usize, u32), Error> {
    let mut max = 0;
    let mut max_idx = 0;

    // find largest number and its position in a bank
    for (i, c) in bank.chars().enumerate() {
        let c_num = c.to_digit(10)
            .ok_or_else(
                || Error::new(
                    ErrorKind::InvalidInput,
                    "Expected an integer between 1-9"
                )
            )?;

        if c_num > max {
            max = c_num;
            max_idx = i;
        }
    }

    Ok((max_idx, max))
}

fn run_pt1() -> Result<u32, Error> {
    let mut total_max = 0;
    let mut buff_reader = BufReader::new(File::open(DAY3_FILE)?);
    let mut string_buff = String::new();

    // iterate over all banks
    while buff_reader.read_line(&mut string_buff)? != 0 {
        let buffer = string_buff.trim();
        let (max_idx, max_val) = find_max_in_bank(buffer)?;

        // largest digit is the last element
        if max_idx == buffer.len() - 1 {

            // find max in prefix
            let (_, max_left) = find_max_in_bank(&buffer[..max_idx])?;
            total_max += max_left * 10 + max_val;

        } else {

            // find max in suffix
            let (_, max_right) = find_max_in_bank(&buffer[(max_idx + 1)..])?;
            total_max += max_val * 10 + max_right;
        }

        // clear buffer
        string_buff.clear();
    }

    Ok(total_max)
}
```

This implementation takes roughly **2.4 ms** to run. There are many places for optimization. First of all, this requires two unequal-size passes:
1. One pass to find the largest digit.
2. Another pass scanning its suffix or prefix to find the second largest digit.

This problem is screaming "*able to be done in one pass.*" though.

Optimized Implementation

We'll solve this in one-pass through the bank. We know the tens-digit must come before the ones-digit. And we want to maximize the two-digit number they form:

$$\max_{i<j}(10d_i + d_j) \tag{1}$$

The constraint $i < j$ suggests we iterate backwards over the bank. Why? Given a tens-digit, we want to find the largest digit that comes after for the ones-digit to satisfy Equation 1. Therefore, if we start from the left, then for every candidate tens-digit, we *have* to pass through the *entire bank* to find the largest digit that comes after. This is *already a pass over the entire bank*, for a single tens-candidate.

But, consider if we iterate backwards, starting from the last digit in the bank. Then we *already know* the digits that come after any digit *as we do the pass over the bank*.

```rust
fn run_pt1_2() -> Result<u32, Error> {
    let mut total_max = 0;
    let mut buff_reader = BufReader::new(File::open(DAY3_FILE)?);
    let mut string_buff = String::new();

    while buff_reader.read_line(&mut string_buff)? != 0 {
        let buffer = string_buff.trim();
        let bytes  = buffer.as_bytes();

        // keeps track of largest two-digit number found
        let mut best_number = 0;

        // keeps track of largest ones-digit found
        // starts all the way on the right, ensures i < j.
        let mut best_ones = (bytes[bytes.len() - 1] - b'0') as u32;

        // iterate backwards
        for &tens in bytes[..bytes.len() - 1].iter().rev() {
            let tens = (tens - b'0') as u32;

            let candidate_number = tens * 10 + best_ones;
            if candidate_number > best_number {
                best_number = candidate_number;
            }

            if tens > best_ones {
                best_ones = tens;
            }
        }

        // accumulate
        total_max += best_number;
        string_buff.clear();
    }

    Ok(total_max)
}
```

This runs in roughly **590 µs**, which is 4× faster than the naive implementation.

SIMD Implementation

I want to try something more interesting. The above optimized implementation is not really `simd`-friendly. It involves updating `best_ones` based on comparisons and is branch-dependent. However, the naive implementation is very `simd`-friendly. It involves finding the largest number in a sequence of digits – twice: one pass through the entire sequence for the tens-place, and another pass through the sequence *after* the tens-place for the ones-place.

This situation is perfect to understand memory-bound nature and if/when SIMD is worth it! Many times people choose to `simd`-optimize to make things faster. We'll do this and compare its performance to the above simple, but optimal one-pass implementation.

This algorithm is the exact same as the naive implementation. To find the largest number in a `String` of numbers 1-9, I convert the string into a `Vec<u8>`. This vector is divided into &[u8; 16] chunks of length 16, and a leftover &[u8] `tail` of length <16. `simd` operations will be performed on the chunks of set length. The leftover tail will be handled in a scalar loop.

```rust
use std::fs::File;
use std::io::{BufReader, BufRead, Error};

use std::simd::Simd;
use std::simd::cmp::SimdPartialOrd;

use std::time::Instant;
use crate::y2025::DAY3_FILE;

const LANES: usize = 16;

/// Finds the maximum number in a given [String]
/// containing digits 1-9.
fn find_max_in_bank(bank: &str) -> Result<(usize, u8), Error> {

    let mut max_idx = 0;
    let mut max_val = 0;

    // String -> Vec
    let bank_vec: Vec<u8> = bank
        .as_bytes()
        .iter()
        .map(|b| b - b'0')
        .collect();

    let (chunks, tail) = bank_vec.as_chunks::<LANES>();

    // fast path on chunks
    for (idx, chunk) in chunks.iter().enumerate() {
        let vec  = Simd::<u8, LANES>::from_array(*chunk);

        // simd greater than comparison with simd vector of `max_val`
        let mask = vec.simd_gt(Simd::splat(max_val));

        // true if a larger value than `max_val` was found in `vec`
        if mask.any() {
            // iterate through `vec` to find the larger value
            for lane in 0..LANES {
                let v = vec[lane];
                if v > max_val {
                    max_val = v;
                    max_idx = idx * LANES + lane;
                }
            }
        }
    }

    // scalar path on tail
    let simd_len = chunks.len() * LANES;
    for (idx, &val) in tail.iter().enumerate() {
        if val > max_val {
            max_val = val;
            max_idx = simd_len + idx;
        }
    }

    Ok((max_idx, max_val))
}

fn run_pt1() -> Result<u32, Error> {
    let mut total_max = 0;
    let mut buff_reader = BufReader::new(File::open(DAY3_FILE)?);
    let mut string_buff = String::new();

    // iterate over all banks
    while buff_reader.read_line(&mut string_buff)? != 0 {
        let buffer = string_buff.trim();
        let (max1_idx, max1_val) = find_max_in_bank(buffer)?;
        let max1_val = max1_val as u32;

        if max1_idx == buffer.len() - 1 {
            let (_, max2_val) = find_max_in_bank(&buffer[..buffer.len() - 1])?;
            let max2_val = max2_val as u32;
            total_max += max2_val * 10 + max1_val;
        } else {
            let (_, max2_val) = find_max_in_bank(&buffer[max1_idx + 1..])?;
            let max2_val = max2_val as u32;
            total_max += max1_val * 10 + max2_val;
        }

        string_buff.clear();
    }

    Ok(total_max)
}
```

This `simd` implementation takes roughly **1.596791 ms**, which is **1.5**× faster than the scalar naive implementation. However, it's still **2.7**1× slower than the scalar, but optimal implementation!

The simple nature of this problem means that the arithmetic intensity, or the ratio of total floating-point operations on the CPU to total data movement, is very low. The *bottleneck* in these simple problems are *how fast data is given to the CPU, not how fast the CPU can do operations.* This is "*memory-bound*".

And considering the naive solution is $O(2n)$ (not exactly, the second pass is smaller) memory reads vrsus the $O(n)$ optimal one-pass solution, the difference is big in a problem dominated by data movement. Additionally, the `simd` implementation also does a `Vec<u8>.collect()` which reads bank bytes, writes bank_vec digits, and reads bank_vec again in `simd` chunks, and this is done again for the second substring pass. This is extra memory bandwidth and cache pressure.

`simd` is great when arithmetic intensity is high and when you can do clean, branch-less algorithms. In other words, `simd` doesn't help much when the bottleneck is "how fast can I move bytes", not "how fast can I compare them."

It is *naive* to think `simd` will meaningfully speed up such a simple, memory-bound problem, relative to a cleaner one-pass solution.

But what if the strings were *really* long? There are two regimes:

1. Fits in cache (L1/L2/L3)
   - Extra passes are still painful
   - SIMD can help a *bit* because everything is hot and cache gives good prefetching
   - the one-pass still wins because it's *strictly* less work (also LLVM)
2. Doesn't fit in cache (streaming from DRAM)
   - One-pass impl: 1× DRAM read of the data
   - Two-pass impl: 1× DRAM read of the data + *another* DRAM read for the leftover data that couldn't fit in cache after the first pass.
   - We also allocate a `Vec` for the whole `String` for each pass.
   - You can imagine the slowdown relative to the simple one-pass solution.