Advent of Code 2025 Day 03

Deval Dilwala

Part 1

Given a sequence of lines called *banks*:

```
987654321111111
811111111111119
234234234234278
818181911112111
```

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** 765432111111, the largest number is **98**. In 818181 **9** 11112 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:

```
use std::fs::File;
use std::io::{BufReader, BufRead, Error, ErrorKind};
use std::time::Instant;
use crates::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:

$$\max_{i<j}(10a_i + a_j) \tag{1}$$

The constraint $i < j$ suggests we iterate backwards from 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.[1]

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 *this pass* over the bank.

[1]Note: *You can do this in one forward pass. You can set the tens place to the first digit, and the ones place to the second digit. Then move the ones place digit to the right until it reaches a value larger than the tens place digit. Then overwrite the tens place digit and place the ones place digit to the right of it. However, there are more edge cases with this implementation and the solution isn't as elegant.*

```
fn run_pt2_{i} -> 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 (ones in bytes[..bytes.len() - 1].iter().rev) {
            let tens = (ones - 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 next_ones based on comparisons and a 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 subsequence 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 `&[u8]` slice. 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.

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

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

use std::time::Instant;
use crates::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 {
                if lane == mask.first().unwrap() {
                    if v > max_val {
                        max_val = v;
                        max_idx = idx * LANES + lane;
                    }
                }
            }
        }
    }

    // scalar path on tail
    let simd_len = chunks.len() * LANES;
    for (idx, local) 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 (max_idx, max_val) = find_max_in_bank(buffer)?;
        let max2_val = max_val as u32;

        if max_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 + max2_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**× slower than the scalar one-pass implementation!

The simple nature of this problem means 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 is *how fast data is given to the CPU, not how fast the CPU can do operations*. This problem is "*memory-bound*."

And considering the naive solution is $O(2n)$ (not exactly, the second pass is smaller) memory reads versus 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 `.collect()` into a `Vec<u8>` which reads bank bytes and writes bank_vec digits. Then we read bank_vec again in SIMD chunks, and also again for the second subsequence 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 compute 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
   - But 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.

Part 2

The only difference is now there are **12** digits in each bank to maximize instead of two. Consider the same example:

```
987654321111111
811111111111119
234234234234278
818181911112111
```

In **987654321111** 111, the largest ordered 12-digit number is 987654321111. In **8 1 8 1 8 1 9 1 1 1 2 1 1 1**, the largest 12-digit number is **888911112111**. The sum of the every bank's largest 12-digit number is returned. These four banks together return 3121910778619.

My first instinct was to do a similar backwards pass like with Part 1. However, with 12 digits, this becomes more complicated.

Previously we were finding

$$\max_{i<j}(10a_i + a_j) \tag{2}$$

So given a d_i two-place digit, we choose the maximal d_j found in the suffix. Only the value mattered. Being to the right of the tens-place was already implicit from iterating backwards.

But with 12 digits, we need 11 digits from the suffix, and they must come from *distinct indices in increasing order*. The best set of 11 digits isn't determined by their values alone, but also on *where they occur*. There's more interaction to consider. Unfortunately the solution is tricky unless you know about the "monotonic stack" data structure. It's suited for problems just like this.

We look at the problem differently.

Maximizing a 12-digit number is the same as deleting

$$D = n - 12 \tag{3}$$

digits so that the remaining digits (in original order) form the largest possible number, where n is the length of the bank. That's a standard greedy: whenever we see a bigger digit, we want to move left (become more significant) by deleting smaller digits before it. But we are only allowed to do this at most D times total to ensure we still have a 12-digit number.

```
fn find_max_12_digit(bank: &[u8]) -> u64 {
    let n = bank.len();

    // number of digits we need to drop
    let mut drop = n - 12;
    let mut stack: Vec<u8> = Vec::with_capacity(n);


    for ib in bank {
        let d = b - b'0';

        // monotonic stack
        // decreasing
        while drop > 0 && !stack.is_empty() && stack[stack.len() - 1] < d {
            stack.pop();
            drop -= 1;
        }

        stack.push(d);

    }

    // only include the first 12 (most significant)
    if stack.len() > 12 {
        stack.truncate(12);
    }

    let mut num = 0;
    for ib in stack.iter() {
        num = num * 10 + ib as u64;
    }

    num
}

fn run_pt2() -> Result<u64, 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 bank = buffer.as_bytes();
        total_max += find_max_12_digit(bank);

        string_buff.clear();
    }

    Ok(total_max)
}
```

Without knowing the monotonic stack concept it's difficult. I don't have much to say about how *to go about* figuring that out.

This takes **634.200 µs** to run.