Advent of Code 2025

Jump to problem

Day 1: Secret Entrance

Today was fairly difficult as far as Day 1's go (or maybe I was tired from the 4 hour drive through a rainstorm I had just hours earlier). As in the past, I did the problems in Kotlin.

Part 1

I started part 1 off wonderfully, misreading it, and (attempting to) solve part 2. When I was getting a higher answer than expected, I went back, reread, and stashed my code for later, (correctly) assuming it may be useful for part 2. Other than that, my code wasn't that interesting, just a check for zero, and repeatedly incrementing my `dial` variable by 100 until it was positive, since modulo arithmetic wouldn't help me there.

Total time: **00:06:03**

Part 2

Part 2 is where I really began to struggle. At first, I attempted the "smarter" solution. It looked something like this (unfortunately my 12:30am brain decided not to save it for when I want to come back later to fix it):

```
(1) dial += value * direction
(2) if (dial == 0) {
(3)     zeroCounter++
(4) }
(5)
(6) while (dial < 100) {
(7)     dial += 100
(8)     zeroCounter++
(9) }
(10)
(11) while (dial > 100) {
(12)     dial -= 100
(13)     zeroCounter++
(14) }
```

The main issue (at least as far as I could tell at the time) was my code would "double count" the dial if it was initially zero before an instruction, and moved to the left. I added a check to cover this case, but it didn't seem to work. This solution is definitely one I would like to revisit after the calendar is over, as it seems like one that could be optimized heavily.

After this attempt, I moved onto the "brute-force" method. I ran a loop from 1 to the instruction's "value", counting every time the dial passed zero, and ensuring it stayed within the range reqired. This ended up working, and giving me the correct solution after many incorrect submissions.

Total time: **00:38:10**

Day 2: Gift Shop

Overall today was way easier than day 1 in my opinion. Most of my pitfalls were due more to me misreading the problem than anything else.

Part 1

My initial solution to this completely ignored everything except the first and last in the range. For some reason, I completely missed that the input was a range and not two unrelated numbers. This wasn't that large a change however, since I was already using `flatMap` to parse the input (just returning a list of the first and second numbers in the list).

To actually process the numbers, I initially used Kotlin's `take` and `drop` methods, roughly as follows:

```
(1) val first = id.take(id.length / 2)
(2) val second = id.drop(id.length / 2)
(3)
(4) if (first == second) count += id.toInt()
```

This worked for Part 1, but would later be changed after I introduced a helper function in Part 2.

Total time: **00:05:49**

Part 2

For this part, I quickly realized a helper function to "chunkify" a string and check the uniqueness of these "chunks" would be helpful, allowing me to refactor Part 1 as well. Luckily, Kotlin provides a `String.chunk` method that does just that. To check for a repeating pattern, I simply turned the chunks into a set, and checked if its length was 1, indicating there was only 1 unique "chunk":

```
(1) fun checkRepeats(string: String, num: Int): Boolean {
(2)     val chunks = string.chunked(num)
(3)
(4)     return chunks.toSet().size == 1
(5) }
```

After writing this function, all I needed to change from my part 1 code was loop from 1 to half the length of each id (any further and there was no possibility for a repeating pattern), checking for a repeating pattern iff the current size evenly divides the id's size (another minor optimization, albeit important for size=1 on part 1, where ids such as `111` would be mistakenly verified since `1`, the first "half" of the string, would cause the remainder to be split into thirds instead of halves).

Total time: **00:14:32**

Day 3: Lobby

This one was a *doozy*. Well at least Part 2 was. Part 1 was genuinely the fastest I've ever solved an AoC problem. It came as quite the shock when I spent close to 20x the time working on Part 2 as I did part 1.

Part 1

For Part 1, I created a simple helper function to brute-force all combinations of 2 digits. Simple, and allowed some Kotlin higher-order function magic to make my primary function (the one that actually processes the inputs) incredibly concise.

Total time: **00:03:34**

Part 2

Part 2 seemed like quite a simple change, but this was one of those problems where a simple change completely eliminates the ability to brute-force (to think its only day 3…). Were I to write a brute-force solution for this problem, I'd need to process roughly 1,050,420,000,000,000 possible combinations, or $\binom{100}{12}$ on the "real-deal" input. This was obviously not ideal.

Roughly in order, I tried these possible solutions:

1. A recursive solution (very briefly entertained this idea, this would go exactly nowhere)
2. Running my solution for part 1, excluding the two digits it selects, and repeating until 12 digits are chosen
3. Removing the smallest digits until 12 remain

It was at this point I began to grow hopeless and ask for help in a programming Discord server I frequent. Most of the help I recieved today came in the form of rubber duck debugging. While talking over my solution, I realized that if I could identify the first digit, I could likely apply my previous (at the time I believed this was the closest) solution, while forcing that digit to be the first.

Finding the first digit was easy. I'll talk more about the code for that shortly, since this *did* end up being a part of my final solution. This worked for the test input, but on the "real-deal" input, did not give the correct answer.

This solution led me to try what would become my final solution. If getting the first digit was so easy, why not just apply that same algorithm 12 times, shortening the input size each time? I tried to implement this, and for some reason, the code would get itself "stuck" between a smaller number of comparatively large digits, leading to outputs with not nearly enough digits.

Obviously, if the code isn't outputting enough digits to satisfy the problem description, that's an issue with the entire algorithm, and definitely not a bug in the code. I told myself this, and proceeded to bang my head against the "first digit then remove smallest digits" algorithm until I got sick and tired of it.

Finally, I decided to actually revisit the solution I tried earlier. This time, I ensured I wouldn't take from the end too early, forcing myself into a smaller amount of text. Truthfully, I have no clue how my method didn't work the first time I tried it. I already handled the logic to ensure enough digits remained when I first implemented the first digit. My final solution ended up making heavy use of Kotlin's `drop` family of methods to limit the "window" I could search for a first digit in, which was where the majority of the refinement for this algorithm took place.

```
(1) var currentFirst = -1
(2) val max = StringBuilder()
(3) for (i in 1 .. 12) {
(4)     val digit = line.drop(currentFirst + 1).dropLast(12 - i).max()
(5)     currentFirst = line.indexOf(digit, currentFirst + 1)
(6)
(7)     max.append(digit)
(8) }
```

The first call to `drop` removed the digits that were already "considered" and prevent them from being duplicated. The `dropLast` call ensures the last n digits remain, where n is the number of digits that still to be added to the string. Other than those calls, the rest is fairly straightforward, and after the loop exits, `max` contains a string representation of the biggest 12-digit number possible to make with the digits provided, solving the problem. Maybe next time I'll actually ensure something is implemented correctly before giving up on it.

Total time: **01:05:35**

Day 4

Today was probably the easiest day so far for me. In fact, the hardest part was problably being able to actually work on it. My apartment's power went out about 45 minutes before the problem release, so I had no Wi-Fi and a dead laptop when it came time for the problem to be released. As such, I waited until 4:30pm to solve it, and in the times I list here, will subtract accordingly. (Unfortunately, this means I was fully awake during the time I spent solving this problem, so my code makes significantly more sense and is less interesting as a result).

Part 1

This was a rather standard "count the neighbors" task on a grid. I threw together two helper functions, one to check if a cell, and another to count them. Other than that, theres not much interesting about my solution, other than my method of handling out of bounds errors (which is why checking a cell is a separate function):

```
(1) fun check(grid: List<String>, coords: Pair<Int, Int>): Boolean {
(2)     try {
(3)         return grid[coords.first][coords.second] == '@'
(4)     } catch (e: Exception) {
(5)         return false
(6)     }
(7) }
```

By wrapping the check in a `try` block, I don't need to check the bounds. If my coordinates are out of bounds, I simply catch the error, and return `false` .

Total time: **00:08:12**

Part 2

Part 2 really gave me a benefit from these helper functions. However, I had to make a few changes to my part 1 solution to allow me to adapt my helper functions to a mutable data structure (Since strings are indexable, I typically can represent a grid as a `List<String>` if it doesnt need to be mutable). After the changes, the grid was represented as a `List<List<Boolean>>` where `true` represented a roll being present.

Other than these changes, the rest was simply a loop over the part 1 solution, terminating when no further rolls are removed, and removing a roll (hence the requirement for mutablilty) if it has less than 4 neighbors.

Total time: **00:13:00**