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):

```
dial += value * direction
if (dial == 0) {
    zeroCounter++
}

while (dial < 100) {
    dial += 100
    zeroCounter++
}

while (dial > 100) {
    dial -= 100
    zeroCounter++
}
```

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:

```kotlin
val first = id.take(id.length / 2)
val second = id.drop(id.length / 2)

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":

```kotlin
fun checkRepeats(string: String, num: Int): Boolean {
    val chunks = string.chunked(num)

    return chunks.toSet().size == 1
}
```

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.

```
var currentFirst = -1
val max = StringBuilder()
for (i in 1 .. 12) {
    val digit = line.drop(currentFirst + 1).dropLast(12 - i).max()
    currentFirst = line.indexOf(digit, currentFirst + 1)

    max.append(digit)
}
```

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: Printing Department

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):

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

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**

Day 5: Cafeteria

Another fairly difficult task today! I think with the shortened calendar this year, we're definitely getting a steeper difficulty curve (or my programming skills got rusty after nearly a year of not coding much outside of classwork).

Part 1

Despite the difficulty of this problem overall, I found Part 1 pretty easy (Part 2 was where the real challenge lied). The main part of the task was to check if a number was with a range, so I extracted this logic into a helper function expecting to use it in Part 2.

Total time: **00:03:52**

Part 2

My first impression on Part 2 was that it looked a lot like `reduce` or `fold`. It did not. Well, at least not the built in `reduce` and `fold` methods. The "reduction" function I'd need to pass to these can either return one element (the ranges were merged successfully), or two (they were not). This isn't possible with the `reduce` or `fold` methods Kotlin provides on `Lists`. So I tried to implement it myself. I made a `mergeRanges` function that would return a boolean (whether the ranges were merged) and a range (either (-1,-1) or the actually merged range). At first, it looked like this:

```kotlin
fun mergeRanges(range1: Pair<Long, Long>, range2: Pair<Long, Long>): Pair<Boolean, Pair<Long, Long>> {
    if (!rangeIncludes(range1, range2.first)) return false to (-1L to -1L)

    return true to (range1.first to range2.second)
}
```

I made a simplifying assumption here. If I sorted the ranges by their first element, I could assume the ranges are mergable if the second ranges start is within the first, and that the merged range had the same start as the first range (still technically correct) and (this is where my assumption was wrong) the merged range's end would be the same as thhat of the second range.

Using this incorrect merge function, I tried 3 different algorithms to solve this, 2 of which worked (I didn't know because of the incorrect function)

First, I tried to implement `reduce` myself, using `map` and `filter` (to remove already-merged ranges, since each range other than the first and last appears in two pairs) over a list of every adjacent pair in the list of ranges, until it didn't change. Honestly, I am glad this didnt work, since the code was genuinely terrible.

My next stab at a solution ended up being an actually functional one. Due to my messed up merge logic, I ended up feeling forced to try one more solution after it, though. By iterating through every pair, I could "clump" as many ranges as I could together, then add their size to the running total. After each iteration, every range used in the clump was marked in a set of used ranges, so they could not be used again.

My final solution ended up using the fact that due to the list of ranges being sorted in my code, any range that could be merged with another would be adjacent to it. This creates "chains" of ranges that can be merged. Due to these "chains," I can keep merging ranges together until I find one that cannot be merged, then stash that “chain” and start a new one. The code is actually pretty simple, despite how much time it took me to come up with:

```
val mergedRanges = mutableListOf<Pair<Long, Long>>()
var current = ranges.first()
for (i in ranges.drop(1)) {
    val mergeResult = mergeRanges(current, i)
    if (!mergeResult.first) {
        mergedRanges.add(current)
        current = i
    } else {
        current = mergeResult.second
    }
}
mergedRanges.add(current)
```

To think essentially a 17 line solution (counting the merge logic) took me nearly an hour to write… Of course, to get this to work, I needed to fix my `mergeRanges` function. Since *technically* the assumption I made regarding the condition for merging ranges was still true, I didn’t need to change the logic there, but I did anyway, just to be safe. The only line that needed to be changed was the last one:

```
return true to (min(range1.first, range2.first) to max(range1.second, range2.second))
```

`range1.first` will always be less than `range2.first`, but I made the changes such that it would work on all pairs, even unsorted ones. With these changes made, I had (two) working solutions. I didn’t go too in-depth on my first solution, which isn’t in the `main` branch on the github repo, but since I commited with the commented-out version I had while testing, it is not lost to history. If you are curious, you can view it [here](#).

Total time: **00:59:06**

Day 6: Trash Compactor

This was another one I couldn't start on time. I had a lifeguarding shift early in the morning, and it is significantly easier to lifeguard on a full night's sleep, so I regrettably had to wait until my shift ended (in hindsight, I would have gone to sleep at the same time). Since I waited until 12:30pm to start today's problems, I will be subtracting that amount of time from the times I list.

Part 1

This was a fairly simple input parsing problem. Since the extra spaces between numbers didn't matter for this part, I used a regex (`\s+`) to split my lines, rather than my usual `" "`. This removed the extra spaces, and I could simply loop over the lists of numbers, applying the specified operation as I went.

Total time: **00:05:30**

Part 2

The main goal of this problem was the same, but the difference was the input's "formatting". Numbers to be added or multiplied were listed right to left, vertically. Since addition and multiplication are commutative, I could safely ignore the right-to-left nature of the values.

To parse the columns out of the file, I first figured out where the columns ended. Since a column ends where every line has a space in the same spot, this was fairly trivial with some list manipulation methods:

```
val columnEnds = (0 until input[0].lastIndex).filter { colmn ->
    input.all { line ->
        line[colmn] == ' '
    }
}
```

Technically, the lines don't have to be of the same length, but since they'll all have the space in the same spot, this was good enough to find all of the spaces. I prepended 0 to this list, since all of the columns begin at index 0.

Next, I split the list into columns using a loop over adjacent pairs of column start and end and `substring`. This excluded the very last pair, since the end of the last column is not a constant. I had to add that one manually.

Now that I had a list of columns, I had to transpose them into a list of numbers. One issue I ran into here was a shorter initial row lead to the end of a column being excluded in the last column. To solve this, I looped over 0 to the maximum index in all of the column's rows, using the `try-catch` trick I used in Day 4 to ignore out of bounds errors.

The rest of my code was essentially the same, except I made use of `reduce` since I had a list of numbers in each column, rather than a list of rows.

Total time: **00:27:16**

Day 7: Laboratories

Yet another day I couldn't start on time, due to the same reason as yesterday. I started at the same time as I did yesterday (12:30pm), so I subtracted from my times as I did yesterday.

Part 1

A simple simulation was enough to solve this one. By maintaining a set of all positions currently occupied by a beam, I could guarantee overlapping beams only counted for one beam. Each time I reached a splitter, I incremented a counter by one.

Total time: **00:03:56**

Part 2

This is my 5th year doing Advent of Code. Every year so far, theres been a similar problem to this one, which seems easy at first glance, but very quickly runs into massive memory issues due to the sheer size of what you're asked to simulate. For all of this style of problem, there's two main methods I've used to solve it. The first, using an alternate data structure to represent the current simulation state, takes a bit more thinking to craft a form of representing the solution. For this reason, I chose to go with a recursive solution, with a bit of memoization to save myself from evaluating what turned out to be on the order of 10^{14} realities

My solution was fairly simple, and very likely would have worked first try, were it not for some (rather stupid) bugs. I created a recursive function, which took in the rows of the manifold yet to be evaluated, and the current x-position of a beam. Upon reaching a splitter, I call the function once (at the same level) for each new beam created, and sum their results. When there are no rows remaining, I return 1.

Since the actual rows are constant, I could represent my function's cache's keys as a `Pair<Int, Int>`. This pair represents the beam's exact position on the grid, and is all that is needed to uniquely identify the result of a call to my recursive solution.

Now for the part where I talk about the stupid mistakes I made.

1. I used an `Int` instead of a `Long`. I don't know how I haven't learned this lesson by now, but when massive amounts of simulation are involved, the answer is very likely to exceed the 32-bit integer limit. Since the answer my solution gave me was a positive number, I saw no issue with it, submitted it, and proceeded to immediately realize what I did wrong when I was told my answer was too low.
2. This issue took me far longer to rectify. My answer was too low, and no matter what I did, I couldn't make it change, since even when I closely inspected my function, I couldn't find anything wrong with it. The issue was that I was not clearing the cache after executing the test case. Since the cache's keys were not based on the input, this lead to my solution falsely identifying parts of the test case to be relevant to the actual solution.

The issue of previous cache data being used was fixed with one line: `cache.clear()`. Had I not made these stupid mistakes, I would probably have solved this problem first try, since neither of them required any change to my actual solution, only how I was using it.

Total time: **00:22:04**

Day 8: Playground

Of the 16 solutions I've written so far this year(including today), 8 of them have required me to use 64-bit integers to store my solution. You'd think I'd have learned by now, just to use them for all of my solutions. No. I have not. And it lead to me spending way more time than necessary on Part 2 today.

Part 1

The main challenge for part 1 was figuring out how to best represent the current circuits. Since each singleton junction box formed its own circuit, I chose to represent the current circuits as a list of sets of `Point3s` (a custom class I made consisting of 3 `Ints`). Then, I looped over the smallest loops either 10 or 1000 times, depending on if I'm running the test or real input. After finding a pair, I merged the two sets together, removing one from the list:

```
val circuitA = circuits.first { shortestPair.first in it }
val circuitB = circuits.first { shortestPair.second in it }

if (circuitA == circuitB) continue

circuits.remove(circuitB)
circuitA.addAll(circuitB)
```

The check to make sure the found circuits are not the same was technically not necessary, and wasa removed in the code for Part 2.

Total time: **00:24:31**

Part 2

My first idea for this, as it is on many problems, was to brute-force it. I threw my code from part 1 in a loop until there was one circuit remaining, and stored which pair was connected last. This worked on the test input, but not on my real input. So, I ended up debugging, and overall making my code from Part 1 significantly better hoping I'd fix the bug. I did not. When I grab my inputs, I tend not to look too deeply at them. Had I done so, I would have quickly realized that multiplying two of the x-positions in the input would very likely result in an integer overflow if I was using a 32-bit integer. Yet again, I defaulted to 32-bit integers, and spent way too much time debugging working code, that happened to be experiencing an integer overflow. I changed my template code to use `Longs` as soon as I finished today's problems.

Total time: **00:47:27**