AOC 2021 Day 24

by Kamiel de Visser

Puzzle input

The puzzle input repeats the same piece of code 14 times, with slight alterations each time it's ran:

```
inp w
mul x 0
add x z
mod x 26
div z <DIVIDE>
add x <CHECK>
eql x w
eql x 0
mul y 0
add y 25
mul y x
add y 1
mul z y
mul y 0
add y w
add y <OFFSET>
mul y x
add z y
```

Some things to note about this code:

- `y` , `x` and `w` values can be 'forgotten' in between runs, as they are reassigned to `0` (for `y` and `x`) or `w` (for `w`) before using them in the 'INP-cycle'.
- `w` values are the digits from the model number we have to input one by one to check the model number for validation.
- `z` values change over the course of multiple runs, and eventually have to equal `0` after the fourteenth run for there to be a valid model number.

- `<DIVIDE>`, `<CHECK>` and `<OFFSET>` contain different values between runs, all other instructions in this 'INP-cycle' repeat exactly as they are 14 times.
    - `<DIVIDE>` is:
        - `1` in cases where `<CHECK>` is a positive number
        - `26` in cases where `<CHECK>` is a negative number
    - `<CHECK>` is always a digit larger than 9

Diving deeper

Lets split up the bit of code and look at what it's essentially doing:

The condition

```
inp w
mul x 0
add x z
mod x 26
div z <DIVIDE>
add x <CHECK>
eql x w
```

can be abbreviated to

```
w = input
x = ((z % 26) / <DIVIDE>) + <CHECK>
x = (((z % 26) / <DIVIDE>) + <CHECK>) == w ? 1 : 0
```

So this bit of code takes in an input `w`, then checks to see if a condition (now named `COND` using that input is true:

`COND: x = (((z % 26) / <DIVIDE>) + <CHECK>) == w ? 1 : 0`

It must be noted here that this condition can only become `1` if `<CHECK>` is a negative number. `w` is limited from a range of `1` to and including `9`, but as we already found out in the first part, `<CHECK>` is always a digit *larger than* `9`.

The double condition

Then follows the second bit of code:

```
eql x 0
```

This takes in the result from `COND`, and checks whether it was false. If it was, it sets `x` to `1` and if it was met, it sets `x` to `0`. In short; it does: `x = (x == 0) ? 1 : 0`

The addition

Now for the final bit of code:

```
mul y 0
add y 25
mul y x
add y 1
mul z y
mul y 0
add y w
add y <OFFSET>
mul y x
add z y
```

Can be abbreviated to:

```
z = z * ((25 * x)+1)
z = z + ((w + <OFFSET>) * x)
```

Now considering that `COND` was met, `x` is `0`, so this can be further abbreviated to:

```
z = z * 1
z = z
```

Which essentially does nothing to `z`. (meaning we can skip the rest of the code entirely if `CHECK` is positive (since that results into `COND` never being met.))

If `COND` was not met however, `x` is `1` and we can see that we can abbreviate the code to:

```
z = 26 * z + w + <OFFSET>
```

The stack

We can now conclude the code essentially does this:

```
if (<CHECK> is positive):   // <DIVIDE> = <CHECK> > 0 ? 26 : 1
        z = z / 1
        z = 26 * z + w + {OFFSET}
else:
        z = z / 26
        if (z + <CHECK> != w):
                z = 26 * z + w + {OFFSET}
```

Which looks eerily close to either pushing or popping from a stack of base-26 numbers. If we think of it that way, we can simplify the code even more:

```
if (<CHECK> > 0):
        stack.push(w, <OFFSET>)
else:
        leftOver = stack.pop()
        if (leftOver + <CHECK> != w)
                stack.push(w, <OFFSET>)
```

We need to get a `z` value of `0` at the end of the program to validate a model number. This means that the stack must be empty.

Here are the consecutive values for `<CHECK>` and `<OFFSET>` of my puzzle input respectively:

Run	1	2	3	4	5	6	7	8	9	10	11	12	13	14
<CHECK>	14	14	14	12	15	-12	-12	12	-7	13	-8	-5	-10	-7
<OFFSET>	14	2	1	13	5	5	5	9	3	13	2	1	11	8

Notice how there are an equal amount of positive (green) to negative (blue) offsets, further supporting our theory that we are popping/pushing to a stack.

If we now run the above code for fourteen digits stored in an array `digit`, we will do:

```
stack.push(digit[0] + 14)
stack.push(digit[1] + 2)
```

```
stack.push(digit[2] + 1)
stack.push(digit[3] + 13)
stack.push(digit[4] + 5)
stack.pop()  // digit[5] == stack.pop() - 12 must be true
stack.pop()  // digit[6] == stack.pop() - 12 must be true
stack.push(digit[7] + 9)
stack.pop()  // digit[8] == stack.pop() - 7 must be true
stack.push(digit[9] + 13)
stack.pop()  // digit[10] == stack.pop() - 8 must be true
stack.pop()  // digit[11] == stack.pop() - 5 must be true
stack.pop()  // digit[12] == stack.pop() - 10 must be true
stack.pop()  // digit[13] == stack.pop() - 7 must be true
```

Thus we can reverse engineer some requirements for all pops to pass the condition:

For every `pop()`, we look at the last value in the stack:

1. At our first `pop()`, our stack contains:

```
0: digit[0] + 14 // bottom of stack
1: digit[1] + 2
2: digit[2] + 1
3: digit[3] + 13
4: digit[4] + 5  // top of stack
```

So `digit[5]` must be a number that makes the condition `digit[5] == digit[4] + 5 - 12` true.

2. At our second `pop()`, our stack contains:

```
0: digit[0] + 14 // bottom of stack
1: digit[1] + 2
2: digit[2] + 1
3: digit[3] + 13
```

So `digit[6]` must be a number that makes the condition `digit[6] == digit[3] + 13 - 12` true.

3. At our third `pop()`, our stack contains:

```
0: digit[0] + 14 // bottom of stack
1: digit[1] + 2
2: digit[2] + 1
3: digit[7] + 9
```

So `digit[8]` must be a number that makes the condition `digit[8] == digit[7] + 9 - 7` true.

Following this process for the entire program, we can find these rules:

```
digit[5] =  digit[4] + 5        - 12
digit[6] =  digit[3] + 13       - 12
digit[8] =  digit[7] + 9        - 7
digit[10] = digit[9] + 13       - 8
digit[11] = digit[2] + 1        - 5
digit[12] = digit[1] + 2        - 10
digit[13] = digit[0] + 14       - 7
```

Or simplified:

```
digit[5] =  digit[4] - 7
digit[6] =  digit[3] + 1
digit[8] =  digit[7] + 2
digit[10] = digit[9] + 5
digit[11] = digit[2] - 4
digit[12] = digit[1] - 8
digit[13] = digit[0] + 7
```

The highest number a `digit[i]` can have is `9`, so we fill in values that satisfy the above conditions whilst maximizing the filled in digits. The key to doing this is filling in a `9` on the lefthand side if we are dealing with an addition, and `9` on the righthand side if we are dealing with a substraction:

```
digit[5] =  digit[4] - 7   // 2 = 9 - 7
digit[6] =  digit[3] + 1   // 9 = 8 + 1
digit[8] =  digit[7] + 2   // 9 = 7 + 2
digit[10] = digit[9] + 5   // 9 = 4 + 5
digit[11] = digit[2] - 4   // 5 = 9 - 4
digit[12] = digit[1] - 8   // 1 = 9 - 8
```

```
digit[13] = digit[0] + 7   // 9 = 2 + 7

digit:
29989297949519
```

Likewise, we can find the smallest possible model number by minimizing the values, filling in `1` instead of `9`, now first on the lefthand side when dealing with substraction, and the righthand side when dealing with addition, as the lowest `digit[i]` can be is `1`:

```
digit[5] =  digit[4] - 7   // 1 = 8 - 7
digit[6] =  digit[3] + 1   // 2 = 1 + 1
digit[8] =  digit[7] + 2   // 3 = 1 + 2
digit[10] = digit[9] + 5   // 6 = 1 + 5
digit[11] = digit[2] - 4   // 1 = 5 - 4
digit[12] = digit[1] - 8   // 1 = 9 - 8
digit[13] = digit[0] + 7   // 8 = 1 + 7

digit:
19518121316118
```