Advent of Code is a series of programming puzzles you can tackle to hone your coding skills each day in the run-up to Christmas.

This year I am attempting it using R, which can make some challenges easier or harder depending on whether they are more ‘computer sciencey’ or more ‘data sciencey’. Generally it makes parsing datasets easier but low-level string manipulation more fiddly.

Here are my solutions so far. Where possible, I’ve tried to strike a balance between efficiency and readability, and to avoid using the packages I might usually use (e.g. `dplyr`) if I think it makes the puzzle too easy.

The input data are different for each participant, so your numerical results may differ from mine.

# Day 1 – Report repair

## Two numbers

Find the two entries that sum to 2020, then multiply those two numbers together.

This can be a one-liner:

```
input <- as.integer(readLines('input01.txt'))
prod(input[(2020 - input) %in% input])
```

```
[1] 468051
```

## Three numbers

Find the three entries that sum to 2020, then multiply them together.

It might be tempting to go for a naïve solution like this:

```
prod(combn(input, 3)[, combn(input, 3, sum) == 2020])
```

```
[1] 272611658
```

It gives the right answer but involves a fair amount of unnecessary computation. It takes more than a second to run. If we assume all the inputs are non-negative, we can take advantage of this to reduce the number of operations.

```
. <- expand.grid(input, input[(2020 - input) > min(input)])
. <- transform(., Var3 = 2020 - Var1 - Var2)
. <- subset(., Var3 > min(input))
prod(.[which.max(.$Var3 %in% input), ])
```

```
[1] 272611658
```

This is approximately 2000 times faster than the one-liner, and works by successively discarding values that could only add up to more than 2020. The `.` notation is just so I can write this without using `dplyr`.

# Day 2 - Password philosophy

## Number of letters

How many passwords are valid according to the policies?

```
1-3 a: abcde
1-3 b: cdefg
2-9 c: ccccccccc
```

First read in the data. I like data frames and so should you.

```
input <- readLines('input02.txt')
passwords <- do.call(rbind, strsplit(input, '[- ]|\\: '))
passwords <- setNames(as.data.frame(passwords),
                      c('min', 'max', 'letter', 'password'))
passwords <- transform(passwords,
                       min = as.integer(min),
                       max = as.integer(max))
head(passwords)
```

```
  min max letter          password
1  14  15      v  vdvvvvvsvvvvvfpv
2   3  11      k  kkqkkfkkvkgfknkx
3   6  10      j        jjjjjjjjjj
4   5  10      s nskdmzwrmpmhsrzts
5  13  15      v   vvvvvvkvvvvjzvv
6  11  13      h    hhhhhbhhhhdhhh
```

String operations are a bit of a pain in base R so it's easier just to use a package, like `stringi` or `stringr` for this.

```
with(passwords, {
     n <- stringr::str_count(password, letter)
     sum(n >= min & n <= max)
})
```

```
[1] 625
```

You could also split each password with `strsplit` and count the letters with an `sapply`-type loop.

## Position of letters

Now the two digits describe two indices in the password, *exactly one* of which must match the given letter.

```
with(passwords,
     sum(xor(substr(password, min, min) == letter,
             substr(password, max, max) == letter))
)
```

```
[1] 391
```

Initially I got caught out here, by misreading the question as 'at least one' and then wondering why an inclusive or (`|`) was returning the incorrect answer.

## Day 3 - Toboggan trajectory

The input looks a bit like this:

```
..##.........##.........##.........##.........##.........##.......
--->
#...#...#..#...#...#..#...#...#..#...#...#..#...#...#..#...#...#..
.#....#..#..#....#..#..#....#..#..#....#..#..#....#..#..#....#..#.
..#.#...#.#..#.#...#.#..#.#...#.#..#.#...#.#..#.#...#.#..#.#...#.#
.#...##..#..#...##..#..#...##..#..#...##..#..#...##..#..#...##..#.
..#.##.......#.##.......#.##.......#.##.......#.##.......#.##.....
--->
.#.#.#....#.#.#.#....#.#.#.#....#.#.#.#....#.#.#.#....#.#.#.#....#
.#........#.#........#.#........#.#........#.#........#.#........#
#.##...#...#.##...#...#.##...#...#.##...#...#.##...#...#.##...#...
#...##....##...##....##...##....##...##....##...##....##...##....#
.#..#...#.#.#..#...#.#.#..#...#.#.#..#...#.#.#..#...#.#.#..#...#.#
--->
```

### Encountering trees

Starting at the top-left corner of your map and following a slope of right 3 and down 1, how many trees would you encounter?

```r
input <- readLines('input03.txt')
```

A complicated-sounding problem but the solution is mainly mathematical.

```r
positions <- (3 * (seq_along(input) - 1)) %% nchar(input) + 1
sum(substr(input, positions, positions) == '#')
```

```
[1] 268
```

The sequence of positions goes 1, 4, 7, …, and when it reaches the edge of the map, loops back round to the beginning. Using the modulo operator we can use the sequence modulo the width of the input map, then add one because R indexes from one rather than from zero.

### Different slopes

Simply wrap the above into a function.

```r
trees <- function(right, down = 1) {
  vertical <- seq(0, length(input) - 1, by = down) + 1
  horizontal <- (right * (seq_along(input) - 1)) %% nchar(input) + 1
  horizontal <- head(horizontal, length(vertical))
  as.double(
    sum(substr(input[vertical], horizontal, horizontal) == '#')
  )
}
trees(1) * trees(3) * trees(5) * trees(7) * trees(1, 2)
```

```
[1] 3093068400
```

The `as.double` bit is necessary only because multiplying large integer outputs together can cause an overflow when the product is larger than $10^9$.

## Day 4 - Passport processing

The example input is in this ragged format, where keys and values are separated by colons and records are separated by double newlines. The first step is to parse this unusual data format.

```
ecl:gry pid:860033327 eyr:2020 hcl:#fffffd
byr:1937 iyr:2017 cid:147 hgt:183cm

iyr:2013 ecl:amb cid:350 eyr:2023 pid:028048884
hcl:#cfa07d byr:1929

hcl:#ae17e1 iyr:2013
eyr:2024
ecl:brn pid:760753108 byr:1931
hgt:179cm

hcl:#cfa07d eyr:2025 pid:166559648
iyr:2011 ecl:brn hgt:59in
```

```
input <- strsplit(readLines('input04.txt'), ' ')
ids = cumsum(!lengths(input))
pairs <- lapply(strsplit(unlist(input), ':'), setNames, c('key',
'value'))
passports <- data.frame(id = rep(ids, lengths(input)),
                        do.call(rbind, pairs))
```

### Missing fields

Now the data are in a standard format, this is a simple split-apply-combine operation. I am using the base `aggregate` but this could be done equally well using `dplyr` or `data.table`.

```
required <- c('byr', 'iyr', 'eyr', 'hgt', 'hcl', 'ecl', 'pid')
valid <- aggregate(key ~ id, passports,
                   function(x) !length(setdiff(required, x)))
head(valid, 10)
```

```
   id   key
1   0  TRUE
2   1  TRUE
3   2  TRUE
4   3  TRUE
5   4  TRUE
6   5  TRUE
7   6  TRUE
8   7 FALSE
9   8 FALSE
10  9  TRUE
```

Then the answer is simply

```
sum(valid$key)
```

```
[1] 190
```

## Field validation

Thanks to the way we imported the data, this is quite straightforward. The rules are:

- `byr` (Birth Year) - four digits; at least 1920 and at most 2002.
- `iyr` (Issue Year) - four digits; at least 2010 and at most 2020.
- `eyr` (Expiration Year) - four digits; at least 2020 and at most 2030.
- `hgt` (Height) - a number followed by either cm or in:
  - If `cm`, the number must be at least 150 and at most 193.
  - If `in`, the number must be at least 59 and at most 76.
- `hcl` (Hair Color) - a # followed by exactly six characters `0-9` or `a-f`.
- `ecl` (Eye Color) - exactly one of: `amb` `blu` `brn` `gry` `grn` `hzl` `oth`.
- `pid` (Passport ID) - a nine-digit number, including leading zeroes.
- `cid` (Country ID) - ignored, missing or not.

The data are all different types (integer, double and categorical) so the first step will be to spread the table to a wider format, with one row per passport, and one column for each field.

Here is a `dplyr` + `tidyr` solution.

```
library(dplyr)
library(tidyr)
passports_wide <- passports %>%
  pivot_wider(names_from = key, values_from = value) %>%
  mutate(byr = as.integer(byr),
         iyr = as.integer(iyr),
         eyr = as.integer(eyr),
         hgt_value = as.numeric(gsub('cm|in$', '', hgt)),
         hgt_unit = gsub('\\d*', '', hgt))
head(passports_wide)
```

```
# A tibble: 6 x 11
     id   iyr cid   pid         eyr hcl     ecl       byr hgt   hgt_value
hgt_unit

1      0  1928 150   4761132~  2039 a5ac0f  #25f8~  2027 190         190
""
2      1  2013 169   9200769~  2026 #fffffd hzl     1929 168cm       168
"cm"
3      2  2011   3284128~  2023 #6b5442 brn     1948 156cm       156
"cm"
4      3  2019 279   6749079~  2020 #602927 amb     1950 189cm       189
"cm"
5      4  2015   4736300~  2022 #341e13 hzl     1976 178cm       178
"cm"
6      5  2020   6281139~  2023 #866857 blu     1984 163cm       163
"cm"
```

From here, we can filter out the invalid entries, using `filter` or `subset`.

```
passports_wide %>%
  filter(byr >= 1920, byr <= 2002,
         iyr >= 2010, iyr <= 2020,
         eyr >= 2020, eyr <= 2030,
         hgt_value >= 150 & hgt_value <= 193 & hgt_unit == 'cm' |
           hgt_value >= 59 & hgt_value <= 76 & hgt_unit == 'in',
         grepl('^#[0-9a-f]{6}$', hcl),
         ecl %in% c('amb', 'blu', 'brn', 'gry', 'grn', 'hzl', 'oth'),
         grepl('^\\d{9}$', pid)) -> valid_passports
nrow(valid_passports)
```

```
[1] 121
```

You could also use a filtering join, though since most of the fields are ranges of integer values, you would want to use a `data.table` *non-equi-join* rather than a simple `semi_join`.

# Day 5 - Binary boarding

## Highest seat ID

This task is easy, as soon as you recognise that it is just converting numbers from binary to decimal, where `F` and `L` denote ones and `B` and `R` are zeros. The distinction between rows and columns is a red herring, because you can parse the whole sequence at once.

```
input <- readLines('input05.txt')
binary <- lapply(strsplit(input, ''), grepl, pattern = '[BR]')
seat_ids <- sapply(binary, function(x) sum(x * 2^(rev(seq_along(x)) -
1)))
max(seat_ids)
```

```
[1] 874
```

## Finding an empty seat

Get the missing value, which isn't the minimum or the maximum in the list.

```
setdiff(seq(min(seat_ids), max(seat_ids)),
        seat_ids)
```

```
[1] 594
```

# Day 6 - Custom customs

## Questions with any 'yes'

Count the number of unique letters in each group, where a 'group' is series of strings separated from others by a blank line. This is a *union* set operation.

```
input <- readLines('input06.txt')
group <- cumsum(!nchar(input))

library(dplyr)
responses <- data.frame(group = group[nchar(input) > 0],
                        questions = input[nchar(input) > 0])
union <- aggregate(questions ~ group, responses,
                    function(x) length(unique(unlist(strsplit(x,
'')))))
sum(union$questions)
```

```
[1] 6551
```

## Questions with all ‘yes’

Similar, but now an *intersection* set operation.

```
intersection <- aggregate(questions ~ group, responses,
                          function(x) length(Reduce(intersect,
strsplit(x, ''))))
sum(intersection$questions)
```

```
[1] 3358
```

The solution to the first part could have used `Reduce(union, ...)`, which would achieve the same result as `unique(unlist(...))`.

Both of these could be made a bit more readable using `dplyr` or `data.table` instead. In particular, the base function `aggregate` doesn’t like list-columns as inputs, so the `strsplit` can’t be done before the aggregation. This is not a problem with `dplyr::summarise` or `data.table`:

```
library(dplyr)
responses %>%
  mutate(questions = strsplit(questions, '')) %>%
  group_by(group) %>%
  summarise(count = Reduce(intersect, questions) %>% length) %>%
  pull(count) %>% sum
```

```
[1] 3358
```

```
library(data.table)
setDT(responses)[, questions := strsplit(questions, '')]
responses[, .(count = length(Reduce(intersect, questions))),
          by = group][, sum(count)]
```

```
[1] 3358…
```