Advent of Code 2018: Some solutions

Katie LeVan

This is a narrative of my solutions to the Advent of Code 2018 Challenge.

— Day 1: Chronal Calibration —

"We've detected some temporal anomalies," one of Santa's Elves at the Temporal Anomaly Research and Detection Instrument Station tells you. She sounded pretty worried when she called you down here. "At 500-year intervals into the past, someone has been changing Santa's history!"

"The good news is that the changes won't propagate to our time stream for another 25 days, and we have a device" - she attaches something to your wrist - "that will let you fix the changes with no such propagation delay. It's configured to send you 500 years further into the past every few days; that was the best we could do on such short notice."

"The bad news is that we are detecting roughly fifty anomalies throughout time; the device will indicate fixed anomalies with stars. The other bad news is that we only have one device and you're the best person for the job! Good lu–" She taps a button on the device and you suddenly feel like you're falling. To save Christmas, you need to get all fifty stars by December 25th.

Collect stars by solving puzzles. Two puzzles will be made available on each day in the advent calendar; the second puzzle is unlocked when you complete the first. Each puzzle grants one star. Good luck!

After feeling like you've been falling for a few minutes, you look at the device's tiny screen. "Error: Device must be calibrated before first use. Frequency drift detected. Cannot maintain destination lock." Below the message, the device shows a sequence of changes in frequency (your puzzle input). A value like +6 means the current frequency increases by 6; a value like -3 means the current frequency decreases by 3.

For example, if the device displays frequency changes of +1, -2, +3, +1, then starting from a frequency of zero, the following changes would occur:

- Current frequency 0, change of +1; resulting frequency 1.
- Current frequency 1, change of -2; resulting frequency -1.
- Current frequency -1, change of +3; resulting frequency 2.
- Current frequency 2, change of +1; resulting frequency 3.

In this example, the resulting frequency is 3.

Here are other example situations:

- +1, +1, +1 results in 3
- +1, +1, -2 results in 0
- -1, -2, -3 results in -6

Starting with a frequency of zero, what is the resulting frequency after all of the changes in frequency have been applied?

```
# Input Data For Puzzle
chg = read.delim(paste(wd,'data/day-1-input.txt',sep='/'), header = F,
                 col.names = 'freq', encoding = 'UTF-8')
```

Because all additions and subtractions cancel each other out, the simple answer is to use the sum() function and return the net value.

Answer: `sum(chg$freq)` is 525

— Part Two —

You notice that the device repeats the same frequency change list over and over. To calibrate the device, you need to find the first frequency it reaches twice.

For example, using the same list of changes above, the device would loop as follows:

- Current frequency 0, change of +1; resulting frequency 1.
- Current frequency 1, change of -2; resulting frequency -1.
- Current frequency -1, change of +3; resulting frequency 2.
- Current frequency 2, change of +1; resulting frequency 3. (At this point, the device continues from the start of the list.)
- Current frequency 3, change of +1; resulting frequency 4.
- Current frequency 4, change of -2; resulting frequency 2, which has already been seen.

In this example, the first frequency reached twice is 2. Note that your device might need to repeat its list of frequency changes many times before a duplicate frequency is found, and that duplicates might be found while in the middle of processing the list.

Here are other examples:

- +1, -1 first reaches 0 twice.
- +3, +3, +4, -2, -4 first reaches 10 twice.
- -6, +3, +8, +5, -6 first reaches 5 twice.
- +7, +7, -2, -7, -4 first reaches 14 twice.

What is the first frequency your device reaches twice?

My first instinct was to loop through the first list and see if a duplicate number emerged.

```
# Determine the list of increments
t = 0 # initialize t
add = vector() # create a container for each time increment

# Loop through the list of time steps
for(n in 1:nrow(chg)){
  # Current time step
  t = (t + chg$freq[n])
  # Log the time step
  add = c(add, t)
}

rm(n,t) # cleanup extra objects
```

Evaluating `any(duplicated(add))` yielded `FALSE` - therefore every number in the list was unique, with no duplicates. Next, I tried implementing a while loop to cycle through the `chg` list, keep track of each increment, and start over if the duplicate was not found. I quickly realized that this approach was taking minutes and no solution had been found.

Clearly, this wasn't the greatest solution from a computational complexity perspective.

Looking at the patterns in the example solutions more closely, I realized that the second round of numbers in the list was just a repeat of the first set, where each number was added to the sum of all numbers in the original list.

For example, if the original list of offsets `+1, -2, +3, +1` had a sum of 3 and the increments at each time step were `+1, -1, +2, +3` where the offsets were applied to zero, then the next time steps would be:

- `+4, +2, +5, +6`
- `+7, +5, +8, +9`
- `+10, +8, +11, +12` etc.

```
# Part 2: Determine first duplicate number
# when duplicate is in existing list, can use 'duplicated' function to find dup
# when many iterations required, can use the fact that the sum() of the list
# is the new starting number whereby all list items are incremented

# If there is a duplicate in the first try
if(any(duplicated(add))){
  # store that duplicate
  dup = (add[duplicated(add)][1])
} else {
  # Increment entire list by the sum
  # until duplicate is found
  final = add;
  while(!any(duplicated(final))){
    # Log the time step
    # If a duplicate is found
    # then the while loop ends
    add = (sum(chg$freq) + add)
    final = c(final, add)

  }
  dup = (final[duplicated(final)][1]) # Else, this is the first duplicate
}
```

When this logic is applied, the first duplicate seen is at 75749 (requiring looping over the list 145 times), which explains in part why the slower one-by-one evaluation method was taking eons to complete (since the first duplicate is at position 146725 in the list).

— Day 2: Inventory Management System —

You stop falling through time, catch your breath, and check the screen on the device. "Destination reached. Current Year: 1518. Current Location: North Pole Utility Closet 83N10." You made it! Now, to find those anomalies.

Outside the utility closet, you hear footsteps and a voice. "...I'm not sure either. But now that so many people have chimneys, maybe he could sneak in that way?" Another voice responds, "Actually, we've been working on a new kind of suit that would let him fit through tight spaces like that. But, I heard that a few days ago, they lost the prototype fabric, the design plans, everything! Nobody on the team can even seem to remember important details of the project!"

"Wouldn't they have had enough fabric to fill several boxes in the warehouse? They'd be stored together, so the box IDs should be similar. Too bad it would take forever to search the warehouse for two similar box IDs..." They walk too far away to hear any more.

Late at night, you sneak to the warehouse - who knows what kinds of paradoxes you could cause if you were discovered - and use your fancy wrist device to quickly scan every box and produce a list of the likely candidates (your puzzle input).

To make sure you didn't miss any, you scan the likely candidate boxes again, counting the number that have an ID containing exactly two of any letter and then separately counting those with exactly three of any letter. You can multiply those two counts together to get a rudimentary checksum and compare it to what your device predicts.

For example, if you see the following box IDs:

- abcdef contains no letters that appear exactly two or three times.
- bababc contains two a and three b, so it counts for both.
- abbcde contains two b, but no letter appears exactly three times.
- abcccd contains three c, but no letter appears exactly two times.
- aabcdd contains two a and two d, but it only counts once.
- abcdee contains two e.
- ababab contains three a and three b, but it only counts once.

Of these box IDs, four of them contain a letter which appears exactly twice, and three of them contain a letter which appears exactly three times. Multiplying these together produces a checksum of 4 * 3 = 12.

What is the checksum for your list of box IDs?

```
# Input Data
boxes = read.delim(paste(wd,'data/day-2-input.txt',sep='/'), header = F,
                   col.names = 'IDs', encoding = 'UTF-8')
```

Using the `strsplit()` function, you can get a list of all single elements from the vector of boxes identifiers. From there, it's trivial to find duplicates within the list (using the `duplicated()` function). Any duplicates that are exactly double or triple repeats in a string can be logged.

```
# Part 1: Checksum; look for doubles & triples
# First create a list that is easier to work with
boxes.split = strsplit(boxes$IDs, split = '')

# make places to store info about detected doubles and triples
boxes[, c('dbl', 'tpl')] = NA
# For each box
for(n in 1:nrow(boxes)){
  # Get all the letters
  b = boxes.split[[n]]
  # Get the duplicates
  dups = b[duplicated(b)]

  if(length(b)==0){
    boxes$dbl[n] = boxes$tpl[n] = 0
  } else {
    # For each detected duplicate
```

```
    for(m in dups){
      # Was it repeated exactly twice?
      if(length(b[b%in%m])%in%2){
        boxes$dbl[n] = 1 # log the answer
      }
      # Was it repeated exactly three times?
      if(length(b[b%in%m])%in%3){
        boxes$tpl[n] = 1 # log the answer
      }
    }
  }
}
```

The the sums of those logs can be multiplied as the puzzle requires.

Answer: `sum(boxes$dbl, na.rm = T) * sum(boxes$tpl, na.rm = T)` is 5434.

— Part Two —

Confident that your list of box IDs is complete, you're ready to find the boxes full of prototype fabric.

The boxes will have IDs which differ by exactly one character at the same position in both strings. For example, given the following box IDs:

abcde fghij klmno pqrst fguij axcye wvxyz

The IDs abcde and axcye are close, but they differ by two characters (the second and fourth). However, the IDs fghij and fguij differ by exactly one character, the third (h and u). Those must be the correct boxes.

What letters are common between the two correct box IDs? (In the example above, this is found by removing the differing character from either ID, producing fgij.)

Using the `adist()` function from the `utils` library makes this solution super simple. `adist()` will evaluate the approximate Levenshtein string distances between two strings s and t such that "the minimal possibly weighted number of insertions, deletions and substitutions needed to transform s into t (so that the transformation exactly matches t)" is returned.

```
# Part 2: Figure out which box IDs are only one letter apart
# Use the distance function to assess differences

# A vector of all IDs
boxes = boxes$IDs

# a data frame of distances
# we are looking for the case where adist returns value of 1
mat = as.data.frame(adist(boxes))

# assign names of boxes to the rows
rownames(mat) = boxes

# Return rownames where the value of the data.frame was 1
box.list = (rownames(which(mat == 1, arr.ind = TRUE)))
```

For me, the box.list of similar names returns “agitmdjvlhedpsyoqfzuknpjwt” and “agirmdjvlhedpsyoqfzuknpjwt”, which differ in the 4th letter. Thus, my puzzle answer was agimdjvlhedpsyoqfzuknpjwt (removing that 4th different letter).

— Day 3: No Matter How You Slice It —

The Elves managed to locate the chimney-squeeze prototype fabric for Santa’s suit (thanks to someone who helpfully wrote its box IDs on the wall of the warehouse in the middle of the night). Unfortunately, anomalies are still affecting them - nobody can even agree on how to cut the fabric.

The whole piece of fabric they’re working on is a very large square - at least 1000 inches on each side.

Each Elf has made a claim about which area of fabric would be ideal for Santa’s suit. All claims have an ID and consist of a single rectangle with edges parallel to the edges of the fabric. Each claim’s rectangle is defined as follows:

The number of inches between the left edge of the fabric and the left edge of the rectangle. The number of inches between the top edge of the fabric and the top edge of the rectangle. The width of the rectangle in inches. The height of the rectangle in inches. A claim like #123 @ 3,2: 5x4 means that claim ID 123 specifies a rectangle 3 inches from the left edge, 2 inches from the top edge, 5 inches wide, and 4 inches tall. Visually, it claims the square inches of fabric represented by # (and ignores the square inches of fabric represented by .) in the diagram below:

```
...........

...........

...#####...

...#####...

...#####...

...#####...

...........

...........

...........
```

The problem is that many of the claims overlap, causing two or more claims to cover part of the same areas. For example, consider the following claims:

- `#1 @ 1,3: 4x4`
- `#2 @ 3,1: 4x4`
- `#3 @ 5,5: 2x2`

Visually, these claim the following areas:

```
........

...2222.

...2222.

.11XX22.

.11XX22.

.111133.
```

```
.111133.

........
```

The four square inches marked with X are claimed by both 1 and 2. (Claim 3, while adjacent to the others, does not overlap either of them.)

If the Elves all proceed with their own plans, none of them will have enough fabric. How many square inches of fabric are within two or more claims?

```
library(sp)

# Input Data
suit = read.delim(paste(wd,'data/day-3-input.txt',sep='/'), header = F,
                  col.names = 'loc', encoding = 'UTF-8')
```

The first thing I did was visualize the problem, using a function I wrote called `makePoly` to convert the claim format `#1 @ 1,3: 4x4` into a polygon.

```
# Make a Polygon for each claim
# makePoly will convert each claim into a Polygon
makePoly <- function(input){
  # Get the values
  vals = unlist(strsplit(input, split = ' '))
  # Set the beginning point
  beg = as.numeric(gsub(':','',unlist(strsplit(vals[3],split = ',')))) # x, y
  # What is the furthest point?
  inc = as.numeric(unlist(strsplit(vals[4],split = 'x'))) # x, y
  # Make a matrix
  m <- matrix(c(beg, # start corner
                beg[1]+inc[1],beg[2], # X corner
                beg[1]+inc[1],beg[2]+inc[2], # far corner
                beg[1],beg[2]+inc[2]), # Y corner
              ncol = 2, byrow = TRUE)

  out <- Polygons(list(Polygon(m)), gsub('#','',vals[1]))
  return(out)
}

# Covert each suit location into a polygon
locs = sapply(suit$loc, makePoly)

# Then the SpatialPolygons function can format those polygons
# so they are ready to plot
polys = SpatialPolygons((locs))
```

```
# Plotting the polygons
plot(polys, border = rainbow(n = 30),
     xlab = "X", ylab = "Y",
     main = "Claims on the Santa Suit material",
     xlim=c(0,1000),ylim=c(0,1000))
axis(1);axis(2); box()
```

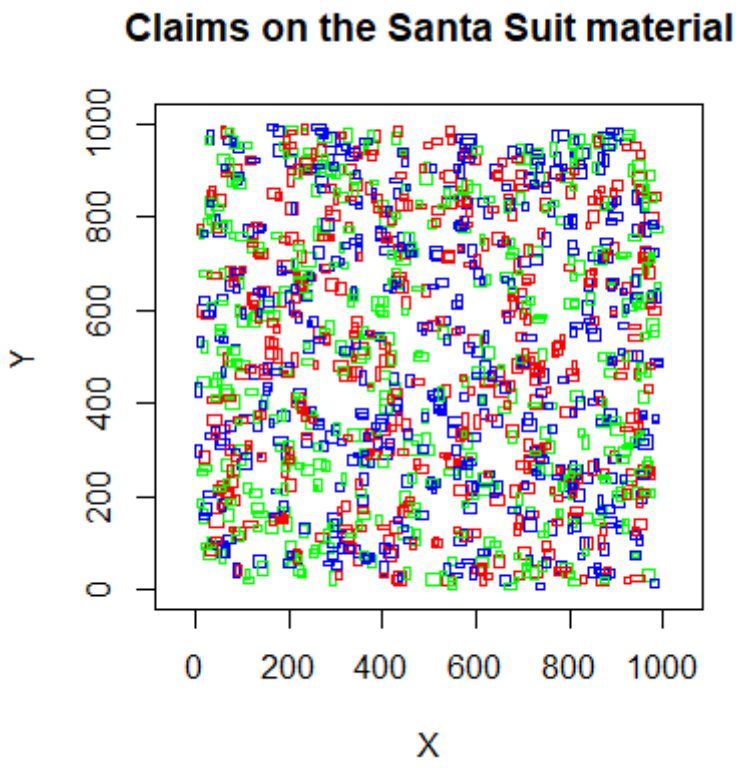

Figure 1: Graphical representation of the problem.

Then converted each claim into a list of coordinate locations it represented, using a function I wrote called `makeList`. From there it's as simple as applying the function and getting the list of duplicate locations.

```
# Convert each item into it's list of addresses,
# then look for ones with > 1 instance
makeList <- function(input){
  # Get the values
  vals = unlist(strsplit(input, split = ' '))
  # Set the beginning point
  beg = as.numeric(gsub(':','',unlist(strsplit(vals[3],split = ','')))) # x, y
  # What is the furthest point?
  inc = as.numeric(unlist(strsplit(vals[4],split = 'x'))) # x, y

  mini = (beg + 1) # x, y start corner
  maxi = (beg + inc) # x, y max corner
  coord = vector()
  for(x in mini[1]:maxi[1]){
    for (y in mini[2]:maxi[2]){
      coord = c(coord, paste(x,y,sep=','))
    }
  }
  return(coord)
}
```

```r
# Use the makeList function to transform each suit location
# into the list of coordinates is comprises
coords = (sapply(suit$loc, makeList))
# unlist for ease of use
cor = unlist(coords)
# determine which claim locations are repeated
dup = unique(cor[duplicated(cor)])

length(dup) # area of duplicate claims
```

The area is easy to calculate by looking at the length of the list of unique names of duplicate claim locations; `length(dup)` is 111266 and the solution to the puzzle.

— Part Two —

Amidst the chaos, you notice that exactly one claim doesn't overlap by even a single square inch of fabric with any other claim. If you can somehow draw attention to it, maybe the Elves will be able to make Santa's suit after all!

For example, in the claims above, only claim 3 is intact after all claims are made.

What is the ID of the only claim that doesn't overlap?

Since the suit location list wasn't long, it was quick to iterate over the list and find the claim that didn't have a value in the duplicate claim location list.

```r
# Part 2: Find the one square that doesn't overlap
# unduplicated coordinates
for (i in 1:nrow(suit)){
  if(length(coords[[i]][coords[[i]]%in%dup])==0){
    print(suit[i,])
  }
}
```

Revealing that claim # 266 had no overlap.