Not A Trick.. But Credit Where Credit Is Due

I can't imagine the amount of work that goes into creating these puzzles.

A bit a cop-out that the first item has nothing to do with R. But I did want to specifically give props to Eric Wastl for making these puzzles. As hard as it was at times to complete the puzzles, I found myself constantly thinking how difficult it must be to *create* them and ensure that they are solvable.

Now onto the R.

Trick #1: Break apart a string of text into a vector with `str_split()` and `unlist()`

The inputs for Advent of Code are usually flat files and its often necessary to break up the input in order to fill out a matrix or columns in a data frame.

Suppose there is an input like:

```
....#..
```

and we want to have each character as a vector element . A function like `readLines` will input each row as a vector, but in order to split the string into each element we'll call upon `str_split()` to break apart the string by a delimiter. Using the empty string ('') will separate each character to create a list. Then `unlist()` will break each character into its own element in the vector

```
input <- "....#..."

print(str_split(input, '') %>% unlist())
## [1] "." "." "." "." "#" "." "." "."
```

Now as opposed to having 1 string, we have a character vector with each character as its own element.

Trick #2: Combining `str_split()` with `unnest()` can turn a vector of strings into a tidy data frame.

One thing that I worked with more in Advent of Code than I have in the last few years have been **matrices**. As shown before, most of the input comes as a flat file needing to be processed. Sometimes it was helpful to represent the matrix as a tidy data-set with columns for `row_id`, `col_id`, and `value` vs. the traditional matrix format. The `unnest()` function will break apart each element of a list into its own row. Using a using a similar input to before but with more rows.

```
input <- c("....#.......",
           ".#..#....###",
           "....###.....")

tibble(raw = input) %>%
  mutate(
    row_id = row_number(), #Create Row ID
```

```
    value = str_split(raw, '') #Break Each Row Into A List Of Elements
  ) %>%
  unnest(value) %>% #Break Each Element Into Its Own Row
  group_by(row_id) %>%
  mutate(col_id = row_number()) %>% #Create Column ID
  head(10) %>%
  kable(align = 'c')
```

raw	row_id	value	col_id
….#…….	1	.	1
….#…….	1	.	2
….#…….	1	.	3
….#…….	1	.	4
….#…….	1	#	5
….#…….	1	.	6
….#…….	1	.	7
….#…….	1	.	8
….#…….	1	.	9
….#…….	1	.	10

Now each element of the character vector is its own row its with own `row_id` and `col_id`.

Trick #3: `extract()` is a powerhouse function for working with strings

I've mentioned before that I think regular expressions are amazing and opens up a world of possibilities. `extract()` allows for the use to regular expressions and capture groups to create any number of new columns. Its similar to `separate()` but to me seems more customizable. Given the inputs:

```
6-7 z: dqzzzjbzz 67
13-16 j: jjjvjmjjkjjjjjjj 123
5-6 m: mmbmmlvmbmmgmmf 5
```

And you wanted to create a data.frame that had columns for the number range, the character before the ':', the series of characters after the ':' and a final digit . This could be done with `str_match()` or similar but `extract()` just makes it so ***easy***. Just give `extract()` a regular expression and capture in parentheses the things to turn into columns.

```
input <- c("6-7 z: dqzzzjbzz 67",
           "13-16 j: jjjvjmjjkjjjjjjj 123",
           "5-6 m: mmbmmlvmbmmgmmf 5")

tibble(raw = input) %>%
  extract(raw,
          into = c('number_range', 'single_char',
                   'many_char', 'single_digit'),
          regex = '(\\d+-\\d+) (\\w+): (\\w+) (\\d+)',
          convert = T) %>%
  kable(align = 'c')
```

number_range	single_char	many_char	single_digit
6-7	z	dqzzzjbzz	67
13-16	j	jjjvjmjjkjjjjjj	123
5-6	m	mmbmmlvmbmmgmmf	5

Done and Done (and with convert=T it even turned the `single_digit` into an int)!

Trick #4: Memoization

Some of the puzzles in AoC use programming concepts I haven't thought about in a long-term (linked lists) and some used concepts I didn't know existed. Memoization is one of those terms that I'd heard before but had no idea what it meant. There were a number of puzzles where my initial brute force solutions would take hours or days to complete. But in certain cases, memoization sped things up immensely.

Memoization caches the results of function calls so that if the same call happens a second time, rather than doing the work again, the program can just recall the value from the cache.

Functions can be memoised in R using the `memoise::memoise()` function to wrap the function.

For this example, I'm borrowing the Fibonacci example from this post on IWNT Statistics.

```
library(memoise)

# Vanilla Function
fibb <- function(x){
  if(x==0){return(1)}
  else if(x==1){return(1)}
  else{return(fibb(x - 1) + fibb(x-2))}
}

# Same Function But Wrapped In Memoise
memo_fib <- memoise(function(x){
  if(x==0){return(1)}
  else if(x==1){return(1)}
  else{return(memo_fib(x - 1) + memo_fib(x-2))}
})
```

Running the original version:

```
tictoc::tic()
fibb(35)
## [1] 14930352
tictoc::toc()
## 26.58 sec elapsed
```

And the memoised version:

```
tictoc::tic()
memo_fib(35)
## [1] 14930352
tictoc::toc()
```

```
## 0.08 sec elapsed
```

The memoised version produces a ***way*** faster result! While hard to believe, the original function makes close to 30 million calls on its way to finding `fibb(35)`. However, the memoised version, only needs to solve for the 35 unique function calls and can recall the answer from cache for the recursive calls.

Trick #5 - String Replacement with Back References

Back to string manipulation!

Within regular expressions there is a concept of “capture groups” which is when you wrap something in parenthesis and then are able to extract it from the string match (like how `str_match()` can work). However, you can also reference what is in the capture group to use it for replacement in functions like `str_replace_all()`.

In our example, image we have a string of animals, `"the cat, a bird, the dog, ze goat"` and we want to insert the adjective **red** between “the” and each animal. There are many ways to do this, but I will use back-references, which will reference the contents of the capture group without knowing specifically what’s in it.

```
input <- "the cat, a bird, the dog, ze goat"

str_replace_all(input, '(\\w+) (\\w+)', '\\1 red \\2')
## [1] "the red cat, a red bird, the red dog, ze red goat"
```

The `\\1` is a back-reference to the first capture group in parenthesis (the, a, the, and ze) while `\\2` is a reference to the animals.

Trick #6 - Escaping stringR’s regular expression matching with `coll()`

More often than not, stringR’s use of regular expressions as the pattern is a blessing. One place where it was troublesome was when I was trying to use one variable as a pattern to replace another variable. In these cases, the special characters in my pattern (the ‘+’) were treated as part of a RegEx rather than the literal string I wanted to match.

For this example, suppose I want to replace an equation within parenthesis with the word ‘hi’ (not sure **why** I’d want to do this, but oh well).

```
tibble(
  eq = c("(1 + 1)", "(7 - 3)", "(12 * 1)")
) %>%
  mutate(ptrn = str_extract(eq, '\\(.+\\)'),
         new_eq = str_replace_all(eq, ptrn, 'hi'),
  ) %>%
  kable(align = 'c')
```

eq	ptrn	new_eq
(1 + 1)	(1 + 1)	(1 + 1)
(7 - 3)	(7 - 3)	(hi)
(12 * 1)	(12 * 1)	(12 * 1)

Notice that the `str_replace_all` either didn't work 100% correctly or didn't work at all for all three cases. Even though as a person this obviously should be a match, in computer-land the symbols "(", ")", "+", and "*" all are special characters for regular expressions and therefore aren't matching the literal symbols they are intended to match.

Fortunately, there is a function `coll()` which will compare strings using standard collation rules rather than using RegExp rules. Wrapping the pattern variable in `coll()` should solve all problems.

```
tibble(
  eq = c("(1 + 1)", "(7 - 3)", "(12 * 1)")
) %>%
  mutate(ptrn = str_extract(eq, '\\(.+\\)'),
         new_eq = str_replace_all(eq, ptrn, 'hi'),
         with_coll = str_replace_all(eq, coll(ptrn), 'hi')
  ) %>%
  kable(align = 'c')
```

eq	ptrn	new_eq	with_coll
(1 + 1)	(1 + 1)	(1 + 1)	hi
(7 - 3)	(7 - 3)	(hi)	hi
(12 * 1)	(12 * 1)	(12 * 1)	hi

Now everything works!

Trick #7 - Use the `assign()` function to programatically create new objects

I always struggle with doing programmatic naming of objects. In the course of one of the puzzles I came across the `assign()` function which takes a variable name, and a object that will be given the variable name.

Suppose we have data in a data.frame with a column for Player and a value for the cards help by the player and we want to create 2 vectors; one for player 1 and one for player 2. We can use assign to create those objects.

```
input <- tibble::tribble(
  ~Player, ~Cards,
       1L,     1L,
       1L,     2L,
       1L,     3L,
       2L,     4L,
       2L,     5L,
       2L,     6L
  )

# Generate the string for the variable name with paste and assign an
object
for(i in seq_len(n_distinct(input$Player))){
  assign(paste0('player_',i), input %>% filter(Player == i) %>%
pull(Cards))
}
```

```
print(player_1)
## [1] 1 2 3
print(player_2)
## [1] 4 5 6
```

Now there are two objected in the environment with names “player_1” and “player_2”