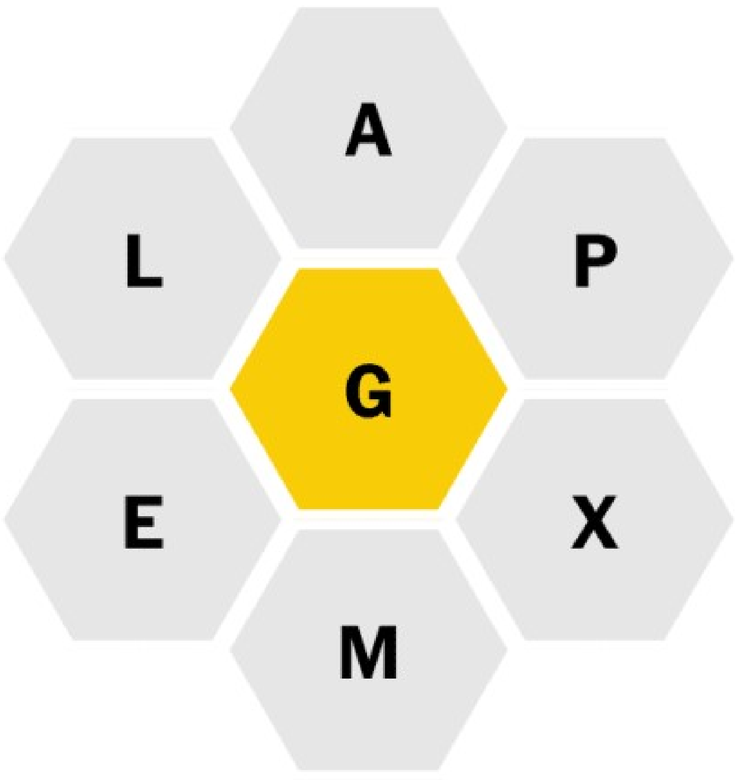
The New York Times recently launched some new word puzzles, one of which is Spelling Bee. In this game, seven letters are arranged in a honeycomb lattice, with one letter in the center. Here's the lattice from December 24, 2019:



The goal is to identify as many words that meet the following criteria:

- The word must be at least four letters long.
- The word must include the central letter.
- The word cannot include any letter beyond the seven given letters.

Note that letters can be repeated. For example, the words GAME and AMALGAM are both acceptable words. Four-letter words are worth 1 point each, while five-letter words are worth 5 points, six-letter words are worth 6 points, seven-letter words are worth 7 points, etc. Words that use all of the seven letters in the honeycomb are known as "pangrams" and earn 7 bonus points (in addition to the points for the length of the word). So in the above example, MEGAPLEX is worth 15 points.

Which seven-letter honeycomb results in the highest possible game score? To be a valid choice of seven letters, no letter can be repeated, it must not contain the letter S (that would be too easy) and there must be at least one pangram.

Solving this puzzle in R is interesting enough, but it's particularly challenging to do so in a computationally efficient way. As much as I love the tidyverse, this, like the "lost boarding pass" puzzle and Emily Robinson's evaluation of the best Pokémon team, this serves as a great example of using R's **matrix operations** to work efficiently with data.

I've done a lot of puzzles recently, and I realized that showing the end result isn't a representation of my thought process. I don't show all the dead ends and bugs, or explain why I ended up choosing a particular path. So in the same spirit as my Tidy Tuesday screencasts, I recorded myself solving this puzzle (though not the process of turning it into a blog post).

Setup: Processing the word list

Our first step is to process the word list for words that could appear in a honeycomb puzzle. Based on the above rules, these will have at least four letters, have no more than 7 unique letters, and never contain the letter S. We'll do this processing with tidyverse functions.

```
library(tidyverse)

words <- tibble(word = read_lines("https://norvig.com/ngrams/enable1.txt")) %>%
  mutate(word_length = str_length(word)) %>%
  filter(word_length >= 4,
         !str_detect(word, "s")) %>%
  mutate(letters = str_split(word, ""),
         letters = map(letters, unique),
         unique_letters = lengths(letters)) %>%
  mutate(points = ifelse(word_length == 4, 1, word_length) +
           15 * (unique_letters == 7)) %>%
  filter(unique_letters <= 7) %>%
  arrange(desc(points))

words

## # A tibble: 44,585 x 5
##    word              word_length letters   unique_letters points
##
##  1 antitotalitarian           16                        7     31
##  2 anticlimactical            15                        7     30
##  3 inconveniencing            15                        7     30
##  4 interconnection            15                        7     30
##  5 micrometeoritic            15                        7     30
##  6 nationalization            15                        7     30
##  7 nonindependence            15                        7     30
##  8 nonintervention            15                        7     30
##  9 nontotalitarian            15                        7     30
## 10 overengineering            15                        7     30
## # … with 44,575 more rows
```

There are 44585 that are eligible to appear in a puzzle. This data gives us a way to solve the December 24th Honeycomb puzzle that comes with the Riddler column.

```
get_words <- function(center_letter, other_letters) {
  all_letters <- c(center_letter, other_letters)

  words %>%
    filter(str_detect(word, center_letter)) %>%
    mutate(invalid_letters = map(letters, setdiff, all_letters)) %>%
    filter(lengths(invalid_letters) == 0) %>%
    select(word, points)
}

honeycomb_words <- get_words("g", c("a", "p", "x", "m", "e", "l"))
honeycomb_words

## # A tibble: 43 x 2
##    word    points
##
##  1 amalgam      7
##  2 agapae       6
##  3 agleam       6
##  4 allege       6
##  5 gaggle       6
```

```
##   6 galeae       6
##   7 gemmae       6
##   8 pelage       6
##   9 plagal       6
##  10 agama        5
## # … with 33 more rows
```

```
sum(honeycomb_words$points)
```

```
## [1] 153
```

Looks like the score is 153, and the pangram that uses all seven letters is AMALGAM.

Could we use this `get_words()` function to brute force every possible honeycomb? Only if we had a lot of time on our hands. Since S is eliminated, there are 25 possible choices for the center letter, and $24\choose 6$ ("24 choose 6") possible combinations of outer letters, making `25 * choose(24, 6)` = **3.36 million possible honeycombs**. It would take about 8 days to test every one this way. We can do a lot better.

Heuristics: What letters appear in the highest-scoring words?

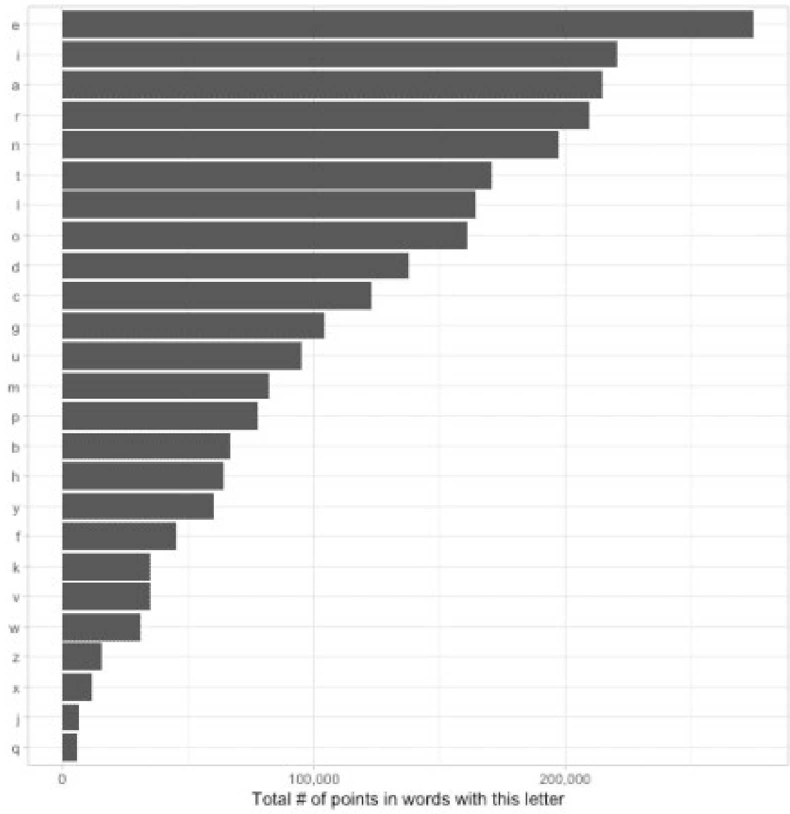
Would you expect the winning honeycomb to have letters like X, Z, or Q? Neither would I. The winning honeycomb is likely filled with common letters like E that appear in lots of words.

Let's quantify this, by looking at how many words each letter appears in, and how many total points they'd earn.

```
library(tidytext)

# unnest_tokens is a fast approach to splitting one-character-per-row
letters_unnested <- words %>%
  select(word, points) %>%
  unnest_tokens(letter, word, token = "characters", drop = FALSE) %>%
  distinct(word, letter, .keep_all = TRUE)

letters_summarized <- letters_unnested %>%
  group_by(letter) %>%
  summarize(n_words = n(),
            n_points = sum(points)) %>%
  arrange(desc(n_points))

letters_summarized
```

```
## # A tibble: 25 x 3
##    letter n_words n_points
##
##   1 e        28203   356775
##   2 i        21128   292785
##   3 a        21719   281514
##   4 r        20977   274395
##   5 n        18735   262133
##   6 t        16232   226985
##   7 l        16503   216576
##   8 o        16038   212751
##   9 d        13894   179331
##  10 c        11644   164720
## # … with 15 more rows
```

What letters have the most and least points?



Of course, any given honeycomb with one of these letters will get only a tiny fraction of the points available to the letter. There could be interactions between those terms (maybe two relatively rare letters go well together). But it gives us a sense of which letters are likely to be included (almost certainly not ones like X, Z, Q, and J), which may help us narrow down our search space in the next step.

Using matrices to score honeycombs

When you need R to be very efficient, you might want to turn to matrix operations, which when used properly are some of the fastest operations in R. Luckily, it turns out that a lot of this puzzle can be done through linear algebra. I'm presenting this as the finished product, but if you're interested in my thought process as I came up with it I do recommend running through the screencast recording to show how I got here.

We start by creating **a word-by-letter matrix**. There are 44K words and (without S) 25 letters. To operate efficiently on these, we'll want these in a 44K x 25 binary matrix rather than strings. The underrated `reshape2::acast` can set this up for us.

```
word_matrix <- letters_unnested %>%
  reshape2::acast(word ~ letter, fun.aggregate = length)

dim(word_matrix)

## [1] 44585    25

head(word_matrix)

##            a b c d e f g h i j k l m n o p q r t u v w x y z
## aahed      1 0 0 1 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## aahing     1 0 0 0 0 0 1 1 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
## aalii      1 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
## aardvark 1 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 1 0 0 0 0
## aardwolf 1 0 0 1 0 1 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 1 0 0 0
## aargh      1 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
```

A points-per-word vector. Once we have the set of words that are possible with each honeycomb, we'll need to score them to find the maximum.

```
points_per_word <- setNames(words$points, words$word)
head(points_per_word)
```

```
## antitotalitarian  anticlimactical  inconveniencing  interconnection
##               31               30               30               30
##  micrometeoritic  nationalization
##               30               30
```

At this point, we take a matrix multiplication approach to score all the honeycomb combinations. Since I went through it in the screencast, I won't walk through each of the steps except in the comments.

```
find_best_score <- function(center_letter, possible_letters) {
  # Find all 6-letter combinations for the outside letters
  outside <- setdiff(possible_letters, center_letter)
  combos <- combn(outside, 6)

  # Binary matrix with one row per combination, 1 means letter is
forbidden
  forbidden_matrix <- 1 - apply(combos, 2, function(.) {
    colnames(word_matrix) %in% c(center_letter, .)
  })

  # Must contain the center letter, can't contain any forbidden
  filtered_word_matrix <- word_matrix[word_matrix[, center_letter] ==
1, ]
  allowed_matrix <- (filtered_word_matrix %*% forbidden_matrix) == 0

  # Score all the words, and add them up within each combination
  scores <- t(allowed_matrix) %*% points_per_word[rownames(allow
ed_matrix)]

  # Find the highest score, and return a tibble with all useful info
  tibble(center_letter = center_letter,
         other_letters = paste(combos[, which.max(scores)], collapse =
","),
         score = max(scores))
}
```

In this approach, I hold the center letter constant, and try every combination of 6 within a pool of letters (the `possible_letters` argument) for the outer letters.

If we use all 25 available letters, this ends up impractically slow and memory-intensive (we end up with something like a 40,000 by 135,000 matrix). But if we bring it down to 15 letters, it can run in a few seconds per central letter. Here I'll use the heuristic we came up with above: it's likely that the winning solution is made up of mostly the ones that appear in lots of high-scoring words, like E, I, and A.

```
pool <- head(letters_summarized$letter, 15)

# Try each as a center letter, along with the pool
best_scores <- purrr::map_df(pool, find_best_combination, pool)

best_scores
```

```
## # A tibble: 15 x 3
##    center_letter other_letters score
##
##  1 e             i,a,r,n,t,g    4169
##  2 i             e,a,r,n,t,g    3806
##  3 a             e,i,r,n,t,g    3772
##  4 r             e,i,a,n,t,g    4298
```

```
##   5 n              e,i,a,r,t,g    4182
##   6 t              e,i,a,r,n,g    3821
##   7 l              e,i,a,n,t,g    2261
##   8 o              e,i,r,n,t,c    2031
##   9 d              e,i,a,r,n,g    2874
##  10 c              e,i,a,r,n,t    2214
##  11 g              e,i,a,r,n,t    3495
##  12 u              e,a,r,n,t,d    1342
##  13 m              e,i,a,r,n,t    1874
##  14 p              e,a,r,t,o,d    1911
##  15 b              e,i,a,r,l,d    1737
```

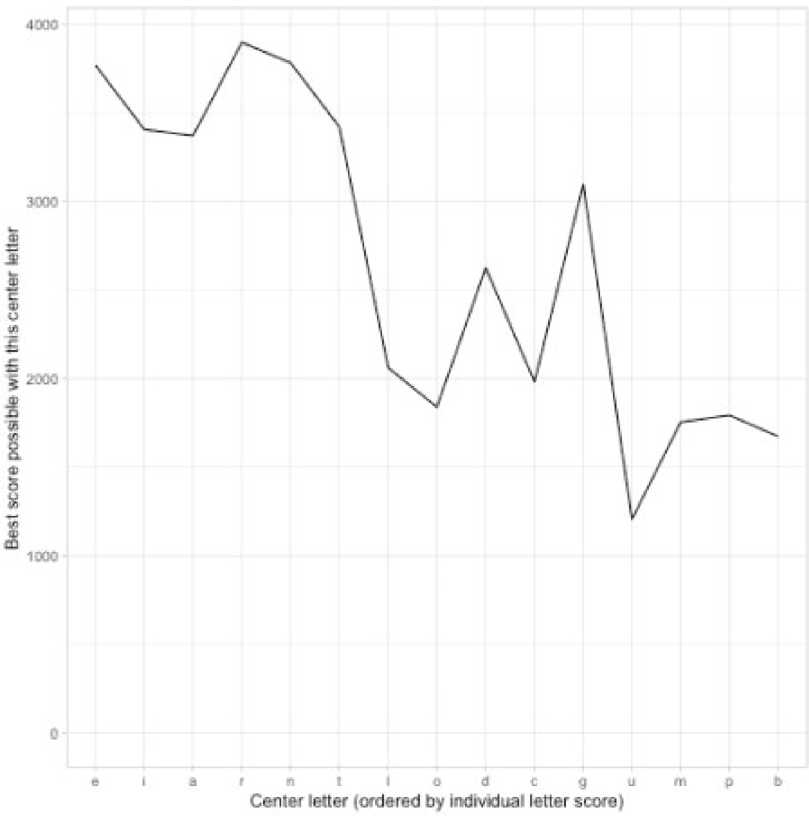
It takes about a minute to try all combinations of the top 15 letters. This makes it clear: **the best combination is R in the center, and E, I, R, N, T, G surrounding it, for a total score of 4298**.

How good was our heuristic at judging letters?

We took a shortcut, and therefore a bit of risk, in winnowing down the alphabet to just 15 letters. How could we get a sense of whether we still got the right answer?

We could start by looking at how good a predictor our heuristic was of how good a letter is in the center.

```
best_scores %>%
  mutate(center_letter = fct_inorder(center_letter)) %>%
  ggplot(aes(center_letter, score)) +
  geom_line(group = 1) +
  expand_limits(y = 0) +
  labs(x = "Center letter (ordered by individual letter score)",
       y = "Best score possible with this center letter")
```



We can see that the heuristic isn't perfect. For instance, G is a surprisingly good center letter (and makes it into the outer letters of our winning honeycomb) considering that overall it doesn't quite make the top 10 for our heuristic. However, it's unlikely we're missing anything with the rarer letters.

Offline, I tried running this code with the top 21 letters (that is, all but Z, X, J, and Q) and while it takes a long time to run it confirms that you can't beat R/EIRNTG, and that none of the letters after G are particularly strong.

Conclusion

After a decade programming in R, I still love the process of journeying through multiple approaches to a problem, and iterating before I find an efficient and elegant solution. (There's probably still a lot of optimization I can do! I'm still using a brute force approach, and I don't know if there's another algorithmic way to solve it). Just because we end up with a handful of matrix multiplications doesn't mean it's easy to get there, especially when you're out of practice with linear algebra like I am.