Jerry Cain
CS 106AX

Handout #18
November 5th, 2025

Section Handout #6

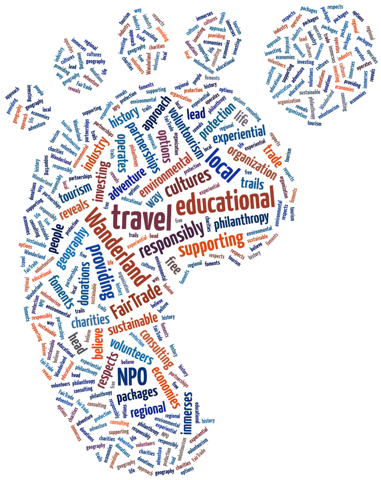
Problem 1: `Rationals` and Unit Fractions

For this problem, you may assume the **`Rational`** class we implemented in lecture has been extended to provide **`getNumerator`** and **`getDenominator`** methods. Implement a function called **`unitFractionSum`**, which accepts a **`Rational`** object called **`r`**—assumed to be between 0 and 1—and returns a list of strictly decreasing unit fractions—that is, fractions with a numerator of 1—whose sum is **`r`**. If **`r`** is already a unit fraction, then **`unitFractionSum`** should just return the list **`[r]`**. Otherwise, compute the largest unit fraction—we'll call it **`u`**—that's less than **`r`** and add **`u`** to a running list of unit fractions that eventually add up to **`r`**. It's an iterative process where you're always computing the largest unit fraction less than or equal to what's left.

```
def unitFractionSum(r):
   """
   Constructs a list of distinct unit fraction
   that add up to the supplied r.
   Examples:
      unitFractionSum(Rational(1, 3)) -> [1/3]
      unitFractionSum(Rational(2, 3)) -> [1/2, 1/6]
      unitFractionSum(Rational(21, 23)) -> [1/2, 1/3, 1/13, 1/359, 1/644046]
   """
```

Now extend the above function to generate a sum of unit fractions (with minimum denominator of 2) for any positive rational number whatsoever, ensuring that no denominator gets used more than once.

```
def unitFractionSum(r):
   """
   Constructs a list of distinct unit fraction
   that add up to the supplied r.
   Examples:
      unitFractionSum(Rational(21, 23)) -> [1/2, 1/3, 1/13, 1/359, 1/644046]
      unitFractionSum(Rational(13, 12)) -> [1/2, 1/3, 1/4]
      unitFractionSum(Rational(5, 2)) ->
         [1/2, 1/3, ..17 terms.. ,1/7894115294, 1/333156570077494116352]
   """
```



Problem 2: Defining and Implementing Classes

Tag clouds are data visualizations often used to convey information about the most prominent words in a large data set (e.g., presidential speeches, college admissions essays, New York Times news articles, or leaked diplomatic cables). The most important, common, and impactful words are typically drawn in larger fonts sizes, and less compelling words, while important enough to be included, are drawn less prominently in smaller font sizes.

Assume that a large subset of all U.S. Presidential speeches has been parsed, and word tag data for all of those speeches has been compiled and stored in a data file structured as follows:

```
Foundation of Government                    // title of first speech
1776-01-15                                  // date of first speech
affections #A19A7E 15                       // first word, color, weight
agreeable #BBB6A2 15                        // second word, color, weight
ambition #D9D6CB 13                         // third word, color, weight
ammunition #CFCCBE 19
antiquity #8C8361 15
approve #D4D0C4 17
assembly #DFDCD3 47
// more tag entries
youth #DEDCD2 15                            // last word, color, weight
                                            // blank line after all speeches, incl the last one
Importance of Property for the Suffrage     // title of second speech
1776-05-2615                                // date of second speech
accommodate #CECABC 19                      // first word, color, weight
// more tag entries
wise #DFDCD3 17                             // last word, color, weight. and so forth

Debate on Independence
1776-06-07
absolved #ADA78F 13
// more tag entries
volunteer #C7DED1 17
```

The first line of the data file contains the name of a speech, and the next line stores the date the speech was delivered. Each line that follows contains three space-delimited tokens: the word of prominence, the color that should be used to render the word in a tag cloud, and the font size that should be used to render the word. The list of prominent words continues until a blank line—literally the empty string—is encountered. The second speech appears after this first blank line, and its data is catalogued in the same manner as the first. All speech data is structured the same way, and even the last speech has a blank line marking the end of its list of prominent words. You can assume the file, for any given speech, stores the words in alphabetical order, and it's perfectly formatted so there's never any parsing drama.

For this problem, you're to provide the full implementation of the **`PresidentialWordCloud`** class, whose constructor accepts the name of a valid, properly structured file and makes a single pass through it to build an internal representation of the relevant data and configure the object so it responds to two methods, which are:

- **`getAllWords`**, which accepts **`title`** and **`date`** strings as arguments and returns the alphabetically sorted list of words that would contribute to that speech's word cloud, and
- **`getAllTags`**, which accepts **`title`**, **`date`**, and **`size`** parameters and returns a list of Python tuples, alphabetically sorted by word, for all tagged words in the speech identified by the supplied title and date that should be rendered in the supplied font size. Each tuple should be of length two: the 0th entry should be the word and the 1th entry should be the color. For example, a call to **`cloud.getAllTags("Foundation of Government", "1776-01-15", 15)`** would produce the following list as a return value:

```
[
  ("affections", "#A19A7E"), ("agreeable", "#BBB6A2"),
  ("ambition", "#D9D6CB"), ("antiquity", "#8C8361"),
  // several similarly structured tuples omitted for brevity
  ("youth", "#DEDCD2")
]
```

The two methods shouldn't need to do anything other than quickly lookup and return information stored in dictionaries fully built by the constructor. Whenever the speech/date and speech/date/size combinations can't be found, you should return the empty list.