420-406-AB Data Structures
Winter 2015

April 20th, 2015
Worth 3%

Lab VII: Anagrams

(Due on May 1st, 2015)

In this lab you will write an *anagram* finder. An anagram is a word formed by rearranging the letters of another word. For example, an anagram of "please" is "asleep". To list all anagrams of a word, you can first list all possible arrangements of a set of letters, called *permutations*, and second, figure out which are dictionary words.

1 Permutations

A *permutation* of a set of elements, $\{x_1, x_2, \ldots, x_n\}$, is an ordered arrangement of these elements.

Example: What are all possible permutations of the numbers {2, 3, 5}?

```
(2, 3, 5)
(2, 5, 3)
(3, 2, 5)
(3, 5, 2)
(5, 2, 3)
(5, 3, 2)
```

Write a recursive function that lists all the permutations of a String of characters. Treat duplicate characters as if they were different: for the string `"eel"` you would have two way to permute the letters to get `"lee"`, and similarly for `"ele"`.

Hint: Think about the permutations of `"abcd"`. They can all be listed as:

- `'a'` concatenated with each of the permutations of `"bcd"`.
- `'b'` concatenated with each of the permutations of `"acd"`.
- `'c'` concatenated with each of the permutations of `"abd"`.
- `'d'` concatenated with each of the permutations of `"abc"`.

2 Matching with a word dictionary

From the permutations generated in the first section and determine which are dictionary words, i.e.: anagrams. Print each word to `System.out`, no more than once. Use the English word dictionary provided with the assignment.

Example: `paeles` has anagrams: `please`, `elapse` and `asleep`

Algorithm: If you store the permutations in a sorted data structure, you can compare them with a sorted file of dictionary words with a *merge*-like operation. Recall that sorting strings results in a "phonebook" order (called *lexicographical order*). Use the following algorithm:

1. Start with the first permutation p and the first dictionary word w.
2. If $p < w$ then advance to the next permutation p'.
3. If $p > w$ then advance to the next word w'.
4. If $p = w$ then include p in the set of words and advance to the next permutation p' and word w'.
5. Repeat the above until either the permutations or words are exhausted.

Requirements

- Write a recursive function `permutations(...)` to generate all the premutations of an input string.
- Use a data structure to store the permutations.
- Use the algorithm described in Section 2.
- Use the provided `words-sorted.txt` dictionary, which is already sorted.

Hand-in Checklist

- ☐ The program is clear and well commented.
- ☐ The project directory contains all of the source files.
- ☐ Zip the project folder and submit it using Léa.

Bonus (33%): Unscrambler

A *combination* of a set of distinct elements $\{x_1, x_2, \ldots, x_n\}$, is way of choosing m elements from the the set, where the order does not matter.

Example 1: What are all the 3-combinations of the numbers $\{2, 3, 5, 7, 11\}$?

```
(2, 3, 5)
(2, 3, 7)
(2, 3, 11)
(2, 5, 7)
(2, 5, 11)
(2, 7, 11)
(3, 5, 7)
(3, 5, 11)
(3, 7, 11)
(5, 7, 11)
```

Example 2: Each 6/49 lottery result is a 6-combination, that is 6 winning numbers are chosen from the set $\{1, 2, \ldots, 49\}$.

Write a recursive function `combinations(...)` that lists all the m-combinations of a set of n characters in a string.

Use the functions `combinations` and `permutations` to write an *unscrambler*. Generate all possible combinations of the input characters ($m = 1..n$). For each combination, find all permutations, and collect them in a list. Finally, repeat the algorithm from Section 2 to pick out all words.

Example: `paeles` has the above anagrams, but also contains smaller words like `sleep`, `else`, `eel` and `ale`.