

Informatik II, SS 2008

Laboratory Sheet E – Due: Week #10

Preambles

Software for this Lab Sheet is provided under the ‘Lab Sheets’ link on the course webpage:

<http://ad.informatik.uni-freiburg.de/lehre/ss08/info2/lab/index.php#LabSheetE>

Word Scrambler

We shall refer to the Word Scrambler as a strategy emulator that we can use in a competitive word game of Scrabble. Thus far, we have been observing and experimenting with the generation of words, given a few constraints; and we have also tackled the permutations of letters within a word to find its anagrams.

In this lab, we shall use those techniques we accumulated to devise several efficient algorithms that we can use for our strategy emulator in the Word Scrambler. You should not be surprised that most of the solutions to the questions in this lab lie directly in the codes written for Lab Sheet C.

1. Preparing the Software

Fire up Eclipse and ensure that the `WordPuzzler` project is open.

- From the course webpage, download the file `wordPuzzler2.zip` and expand it in your workspace directory. The `wordPuzzler2.zip` contains two new files to be added to the current `WordPuzzler` project: `LetterDispenser.java` and `NoMoreTilesException.java`, which shall be deposited into the packages `wordPuzzler.util` and `wordPuzzler.lang`, respectively.
- Also, download the file `solns_lab_C.zip` and expand it in your workspace directory. Alternatively, you may wish to use your own codes that were submitted for Lab Sheet C to handle this week’s lab.
- Create a new package for this lab, following the package naming convention introduced in the previous lab sheet. For example, users ‘bob’ and ‘belinda’ working together for this Lab Sheet E, will create a package name `'bob_belinda_lab_E'`.
- Create a new class, in your new package, called `WordScrambler`. From the course webpage, copy and paste the skeleton codes for `WordScrambler.java`.

2. The Letter-Dispenser

The class `LetterDispenser` is the main backbone letter-supplier for this week’s Word Scrambler problem. That is, the `LetterDispenser` contains a fixed set of letters, called ‘tiles’, in its repository. Each tile has an associated point-value to it. Once a tile is dispensed, it no longer remains in the repository, and that the `LetterDispenser` can dispense as many tiles as it has left in its repository – you can query the complete current distribution of letters (and their point-values) in the `LetterDispenser` by calling the `toString()` method.

Take some time to review the `LetterDispenser` class, paying special attention on how you can manipulate it to solve the questions in the following sections.

Exercise 1:

Properly complete the procedure `getScrambledWord(int n)`, that should *cleanly* retrieve an n -letter word from the `LetterDispenser`.

That is, write codes to *safely* return a scrambled word, such that if there are fewer than n tiles left in the `LetterDispenser`, then we return all the remaining tiles that the `LetterDispenser` can dispense. Your procedure should return *null* if there are no more tiles left to dispense.

3. Point-Value of a Word

Each tile in the `LetterDispenser` has a point-value; following the standard English Scrabble tile distribution. In devising a strategy to scramble a word, we will need to know the value of scrambled word.

(*) Exercise 2 – [2 marks]:

Write a procedure in the method `getPointValue(String word)` in `WordScrambler` that returns the value of the given word, based on the values of its letters within.

(*) Exercise 3 – [4 marks]:

Complete the method `sortWordsByPointValue(List<String> words)` that takes in a `List` of words and sorts them in order of the point-value of each word. The word with the highest value shall be at the top of the list.

If several words with the same point-value exist, then these words must be arranged in alphabetical order in the returned `List`.

Your sorting procedure must run in time $O(n \log n)$, where n is the number of words to be sorted.

4. Generating Valid Words

Since the `LetterDispenser` only returns a word containing randomly scrambled letters, which we shall refer to as `refWord`, we must be able to determine all anagrammatic words that can be derived from `refWord`.

(*) Exercise 4 – [4 marks]:

Write a procedure in the method `generateValidWords(String refWord)` in `WordScrambler` that generates all *valid* words from the given n -letter `refWord` of shortest length 2 and of longest length n .

Your method must return the `List` of words in order of the point-value of each word, beginning with the highest value on the top of the `List`.

Example:

Input: `refWord = "fwnfe"`

Output: {"few"[9], "fen"[6], "new"[6], "we"[5]}

* Numbers in square brackets indicate the point-value of that word.

Once we have a list of valid words to work with, and once we know each of its point-value, we can attempt to determine a related anagram of each valid word, with an additional letter to improve its point-value.

For example, suppose we have a valid word "word", whose point-value is 8. Now, we can improve its point-value to become 9 if we add the letter 's', to form "words" or "sword". However, we can better improve the

point-value of the original word to **12**, if we add the letter 'y' instead, to form "dowry", "rowdy", or "wordy".

Thus, it now becomes our quest to determine these additional letters, and select the one that offers the best point-value improvement.

(*) Exercise 5 – [5 marks]:

Write a procedure in the method `findDirectAnagrams(String validWord, char letter)` in `WordScrambler` that finds all *direct* anagrams of the combined `validWord` and `letter`.

Example:

Input: `validWord = "word", letter = 's'`

Output: {"sword", "words"}

(*) Exercise 6 – [5 marks]:

Complete the method `suggestLetterForNewWord(String validWord)`, where given a `validWord`, we then *suggest* one letter that can be added into any position within the `validWord`, and still make the new word (or its resultant *direct* anagrams) valid.

Find all such *letters* and return them in a `char[]` array.

Example:

Input: `validWord = "word"`

Output: {'c', 'e', 'l', 'n', 's', 'y'}

Exercise 7:

Write a similar procedure to Exercise 6 in the method `suggestLetterForNewWord(String refWord, String validWord)` in `WordScrambler` that given an *n*-letter `refWord` and a `validWord` (which was generated from `refWord`, through the procedure in Exercise 4), suggest *the one* letter that can be added into any position within the `validWord`, and still make the new word (or its resultant anagrams) valid. The single returned `char` must comply to the following constraints:

- The suggested letter must NOT be in `refWord`; and
- The suggested letter must be the letter that will best improve the point-value of the original `validWord`. If more than one such letter exists, then we return the first one encountered.

5. Putting Together the Word Scrambler Round

We can now assemble the various methods developed above and conceive a Word Scrambler Round.

Exercise 8:

Complete the method `executeRound()` in `WordScrambler` that returns a `List` of valid words in order of their point values. The method `executeRound()` shall perform the following routines in search of the `List` of valid words to return:

- Retrieves a random `refWord` from the `LetterDispenser`;
- Generates all valid words from `refWord`;
- For each valid word, find the letter that will best increase the word's point-value, and generate all *direct* anagrams of the combined word and letter, and add them to the list of valid words.

4 Submitting Your Work

You will submit your solutions to the starred (*) questions via email to your tutor during your lab sessions. As usual, zip up the package you created for this Lab Sheet E, which should contain the main `WordScrambler.java` file (and any other class files you may have created to solve this week's lab).

***Note on Plagiarism**

Submitted solutions that are plagiarized or directly copied from other sources will not be accepted. Plagiarism is broadly defined to be when any portion of the work presented for assessment, can be attributed to another party. *The student making the submission should acknowledge what aspects of the presented work is not directly derived by them.* For the purposes of plagiarism it is irrelevant that you have been given permission by someone to copy their work and present it as your own.

Max: 20 marks