CIT 594 – Homework 3

Due – Feb 29, 2016 at 12pm

Part 1 – Theory (20 points)

Please do the following problems:

1. Exercise R-16.22 from Big Java (5 points)
2. Exercise R-16.24 from Big Java (5 points)
3. Exercise R-16.25 from Big Java (5 points)
4. Suppose you have a stack S containing n elements and a queue Q that is initially empty. Describe how you can use Q to scan S to see if it contains a certain element x, with the additional constraint that your algorithm must return the elements back to S in their original order. You may not use an array or linked list—only S and Q and a constant number of reference variables. What is the running time of your algorithm? (5 points)

Part 2 – Programming (80 points)

Evil Hangman: This assignment has been heavily inspired by, and borrowed from, Keith Schwartz's Nifty Assignment (http://nifty.stanford.edu/2011/schwarz-evil-hangman/)

It's hard to write computer programs to play games. When we as humans sit down to play a game, we can draw on past experience, adapt to our opponents' strategies, and learn from our mistakes. Computers, on the other hand, blindly follow a preset algorithm that (hopefully) causes it to act somewhat intelligently. Though computers have bested their human masters in some games, most notably checkers and chess, the programs that do so often draw on hundreds of years of human game experience and use extraordinarily complex algorithms and optimizations to outcalculate their opponents.

While there are many viable strategies for building competitive computer game players, there is one approach that has been fairly neglected in modern research – cheating. Why spend all the effort trying to teach a computer the nuances of strategy when you can simply write a program to play dirty and win handily all the time? In this assignment, you will build a mischievous program that bends the rules of Hangman to trounce its human opponent time and time again. In doing so, you'll cement your skills with various data structures, and will hone your general programming savvy. Plus, you'll end up with a piece of software which will be highly entertaining. At least, from your perspective ☺

In case you aren't familiar with the game Hangman, the rules are as follows:

1. One player chooses a secret word, then writes out a number of dashes equal to the word length.

2. The other player begins guessing letters. Whenever she guesses a letter contained in the hidden word, the first player reveals each instance of that letter in the word. Otherwise, the guess is wrong.
3. The game ends either when all the letters in the word have been revealed or when the guesser has run out of guesses.

Fundamental to the game is the fact the first player accurately represents the word she has chosen. That way, when the other players guess letters, she can reveal whether that letter is in the word. But what happens if the player doesn't do this? This gives the player who chooses the hidden word an enormous advantage. For example, suppose that you're the player trying to guess the word, and at some point you end up revealing letters until you arrive at this point with only one guess remaining:

```
D O – B L E
```

There are only two words in the English language that match this pattern: “doable” and “double.” If the player who chose the hidden word is playing fairly, then you have a fifty-fifty chance of winning this game if you guess 'A' or 'U' as the missing letter. However, if your opponent is cheating and hasn't actually committed to either word, then there is no possible way you can win this game. No matter what letter you guess, your opponent can claim that she had picked the other word, and you will lose the game. That is, if you guess that the word is “doable,” she can pretend that she committed to “double” the whole time, and vice-versa.

Let's illustrate this technique with an example. Suppose that you are playing Hangman and it's your turn to choose a word, which we'll assume is of length four. Rather than committing to a secret word, you instead compile a list of every four-letter word in the English language. For simplicity, let's assume that English only has a few four-letter words, all of which are reprinted here:

```
ALLY   BETA   COOL   DEAL   ELSE   FLEW   GOOD   HOPE   IBEX
```

Now, suppose that your opponent guesses the letter 'E.' You now need to tell your opponent which letters in the word you've “picked” are E's. Of course, you haven't picked a word, and so you have multiple options about where you reveal the E's. Here's the above word list, with E's highlighted in each word:

```
ALLY   BETA   COOL   DEAL   ELSE   FLEW   GOOD   HOPE   IBEX
```

If you'll notice, every word in your word list falls into one of five “word families:”

- `----,` which contains the word ALLY, COOL, and GOOD.
- `-E--,` containing BETA and DEAL.
- `--E-,` containing FLEW and IBEX.
- `E--E,` containing ELSE.
- `---E,` containing HOPE.

Since the letters you reveal have to correspond to some word in your word list, you can choose to reveal any one of the above five families. There are many ways to pick which family to reveal – perhaps you want to steer your opponent toward a smaller family with more obscure words, or toward a larger family in the hopes of keeping your options open. In this assignment, in the interests of simplicity, we'll adopt the latter approach and always choose the largest of the remaining word families. In this case, it means that you should pick the family `----`. This reduces your word list down to

`ALLY   COOL   GOOD`

and since you didn't reveal any letters, you would tell your opponent that his guess was wrong.

Let's see a few more examples of this strategy. Given this three-word word list, if your opponent guesses the letter O, then you would break your word list down into two families:

- `-OO-,` containing COOL and GOOD.
- `----,` containing ALLY.

The first of these families is larger than the second, and so you choose it, revealing two O's in the word and reducing your list down to

`COOL   GOOD`

But what happens if your opponent guesses a letter that doesn't appear anywhere in your word list? For example, what happens if your opponent now guesses 'T'? This isn't a problem. If you try splitting these words apart into word families, you'll find that there's only one family – the family `----` in which T appears nowhere and which contains both COOL and GOOD. Since there is only one word family here, it's trivially the largest family, and by picking it you'd maintain the word list you already had.

There are two possible outcomes of this game. First, your opponent might be smart enough to pare the word list down to one word and then guess what that word is. In this case, you should congratulate him – that's an impressive feat considering the scheming you were up to! Second, and by far the most common case, your opponent will be completely stumped and will run out of guesses. When this happens, you can pick any word you'd like from your list and say it's the word that you had chosen all along. The beauty of this setup is that your opponent will have no way of knowing that you were dodging guesses the whole time – it looks like you simply picked an unusual word and stuck with it the whole way.

The Assignment

Your assignment is to write a computer program which plays a game of Hangman using this “Evil Hangman” algorithm. In particular, your program should do the following:

1. Read the file `dictionary.txt`, which contains the full contents of the Official Scrabble Player's Dictionary, Second Edition. This word list has over 120,000 words, which should be more than enough for our purposes.
2. Prompt the user for a valid word length and number of guesses.

3. Prompt the user for whether she wants to have a running total of the number of words remaining in the word list. This completely ruins the illusion of a fair game that you'll be cultivating, but it's quite useful for testing (and grading!)
4. Play a game of Hangman using the Evil Hangman algorithm, as described below:
    a. Construct a list of all words in the English language whose length matches the input length.
    b. Print out how many guesses the user has remaining, along with any letters the player has guessed and the current blanked-out version of the word. If the user chose earlier to see the number of words remaining, print that out too.
    c. Prompt the user for a single letter guess.
    d. Partition the words in the dictionary into groups by word family.
    e. Find the most common "word family" in the remaining words, remove all words from the word list that aren't in that family, and report the position of the letters (if any) to the user. If the word family doesn't contain any copies of the letter, subtract a remaining guess from the user.
    f. If the player has run out of guesses, pick a word from the word list and display it as the word that the computer initially "chose."
    g. If the player correctly guesses the word, congratulate her.

Advice, Tips, and Tricks

Since you're building this project from scratch, you'll need to do a bit of planning to figure out what the best data structures are for the program. We have seen Arrays, Linked Lists, Stacks, Queues, and HashMaps so far. There is no "right way" to go about writing this program, but some design decisions are much better than others (e.g., you can store your word list in a stack or map, but this is probably not the best option). Here are some general tips and tricks that might be useful:

1. Letter position matters just as much as letter frequency. When computing word families, it's not enough to count the number of times a particular letter appears in a word; you also have to consider their positions. For example, "BEER" and "HERE" are in two different families even though they both have two E's in them. Consequently, representing word families as numbers representing the frequency of the letter in the word will get you into trouble.
2. Don't explicitly enumerate word families. If you are working with a word of length n, then there are 2^n possible word families for each letter. However, most of these families don't actually appear in the English language. For example, no English words contain three consecutive U's, and no word matches the pattern `E-EE-EE--E`. Rather than explicitly generating every word family whenever the user enters a guess, see if you can generate word families only for words that actually appear in the word list.

Extra Credit (10 points)

The algorithm outlined in this handout is by no means optimal, and there are several cases in which it will make bad decisions. For example, suppose that the human has exactly one guess remaining and that computer has the following word list:

```
DEAL   TEAR   MONK
```

If the human guesses the letter 'E' here, the computer will notice that the word family `-E--` has two elements and the word family `----` has just one. Consequently, it will pick the family containing DEAL and TEAR, revealing an E and giving the human another chance to guess. However, since the human has only one guess left, a much better decision would be to pick the family `----` containing `MONK`, causing the human to lose the game.

There are several other places in which the algorithm does not function ideally. For example, suppose that after the player guesses a letter, you find that there are two word families, the family `--E-` containing 10,000 words and the family `----` containing 9,000 words. Which family should the computer pick? If the computer picks the first family, it will end up with more words, but because it revealed a letter the user will have more chances to guess the words that are left. On the other hand, if the computer picks the family `----`, the computer will have fewer words left but the human will have fewer guesses as well. More generally, picking the largest word family is not necessarily the best way to cause the human to lose. Often, picking a smaller family will be better.

After you implement this assignment, take some time to think over possible improvements to the algorithm. You might weight the word families using some metric other than size. You might consider having the computer "look ahead" a step or two by considering what actions it might take in the future – this idea of looking ahead generalizes to a strategy called "**Minimax**", which can be shown to play a theoretically perfect game. Researching and implementing a minimax search can make your player substantially more powerful.

For the EC, you need to make significant improvements to the base algorithm to ensure that the human player loses faster.

If you attempt the EC, please submit a file called `ec.txt`. This file should outline what improvements you made to the base algorithm and show that it does indeed beat the human player faster.

As always, for the EC part, you cannot have any help from the TAs/instructor.

Unit Testing and Code Coverage

For this homework, there is no explicit requirement for unit testing and coverage. However, an important part of the grading criteria (see below) will be on correctness. We encourage and recommend that you perform unit testing for the “critical” parts where it’s likely there are bugs in your code.

Grading Criteria

0% for compilation – If your code compiles, you get full credit. If not, you get a 0.
60% for functionality – Does the code work as required? Does it crash while running? Are there bugs? …
15% for design – Is your code well designed? Does it handle errors well? …
15% for data structure design – Did you choose data structures appropriately? Please explain your decisions in the readme.txt file.
10% for style – Do you have good comments in the code? Are your variables named appropriately? ...

Programming – General Comments

Here are some guidelines with respect to programming style.

Please use Javadoc-style comments.

For things like naming conventions, please see Appendix I (Page A-79) of the Horstmann book. You can also install the Checkstyle plugin (http://eclipse-cs.sourceforge.net/) in Eclipse, which will automatically warn you about style violations.

Submission Instructions

We recommend submitting the theory part electronically also. However, you can turn in a physical copy at the start of class, if you prefer. Please **do not** print out the Java source.

In addition to the theory writeup, you should also submit a text file titled readme.txt. That is, write in plain English, instructions for using your software, explanations for how and why you chose to design your code the way you did. The readme.txt file is also an opportunity for you to get partial credit when certain requirements of the assignment are not met. Think of the readme as a combination of instructions for the user and a chance for you to get partial credit.

Please create a folder called YOUR_PENNKEY. Places all your files inside this – theory writeup, the Java files, the readme.txt file. Zip up this folder. It will thus be called YOUR_PENNKEY.zip. So, e.g., my homework submission would be swapneel.zip. Please submit this zip file via canvas.