CSE12 - Spring 2014 HW #3

Running time

(100 points)
Due 11:59pm 22 April 2014

In this assignment you will analyze the running time of code and algorithms, practice with Big-O (and Big-Omega and Big-Theta) notation, and run empirical tests on your MyLinkedList class.

This assignment is an individual assignment. You may ask Professors/TAs/Tutors for some guidance and help, but you can't copy code. You can, of course, discuss the assignment with your classmates, but don't look at or copy each others code or written answers.

The following files are provided for you and can be found on the HW page:

- MyLinkedList.java
- MRUList.java
- CollectionTimer.java
- pride-and-prejudice.txt
- big-wordlist.txt
- small-wordlist.txt
- medium-wordlist.txt
- large-wordlist.txt
- hugh-wordlist.txt

You will submit the following file for this assignment:

- **HW3.txt**

This will be a plain text file (NOT .docx or .pdf or anything else). You should use the ^ symbol to indicate exponentiation (e.g. n^2 should be written n^2) and write out the words Theta (Θ) and Omega (Ω) where appropriate. At the top of your HW3.txt file, please include the following header:

```
CSE 12 Homework 3
Your Name
Your PID
Section [Your section] (A00 for Alvarado, B00 for Papadopoulos)
The date
```

Part 1 - Analyzing running time (42 points)

Part 1A: Warm up with Big-O, Big-Theta and Big-Omega (18 points)

True/False. In a section labeled Part 1A in your HW3.txt file, state whether each of the following equalities are true or false. You should put the number of the question followed by either the word True or False. One answer/line. eg.

```
Part 1A
1. True
2. False
```

and so on.

1. $n^2 + 100 = O(n^4)$
2. $n^2 + 100 = O(n^2)$
3. $n^2 + 100 = O(n)$
4. $n^2 - 1,000,000,000 = O(n)$
5. $n^2 + n = O(n)$
6. $n^2 + n = O(n^2)$
7. $n^2 * n = O(n^2)$
8. $n^2 * n = O(n^3)$
9. $n^2 + log(n) * n^2 = O(n^2)$
10. $n^2 + 100 = \Omega(n^4)$
11. $n^2 + 100 = \Omega(n^2)$
12. $n^2 + 100 = \Omega(n)$
13. $n^2 + n = \Omega(n)$
14. $n + 100,000 = \Omega(n)$
15. $n^2 + n + 100 = \theta(n)$
16. $n^2 + n + 100 = \theta(n^2)$
17. $n^2 + n + 100 = \theta(n^3)$
18. $n^2 * n + 100 = \theta(n^3)$

Part 1B: Analyzing running time of code (24 points)

In this part you will practice your skills of estimating running time of code snippets. For each piece of code below, state the running time of the snippet in terms of the loop limit variable, *n*. You should assume that the variables n and sum are already declared and have a value. You should express your answer using Big-O or Big-Θ (Theta) notation, though your Big-O bound will only receive full credit if it is a tight bound. We allow you to use Big-O because it is often the convention to express only upper bounds on algorithms or code, but these upper bounds are generally understood to be as tight as possible. In all cases, your expressed bounds should be simplified as much as possible, with no extra constant factors or additional terms (e.g. O(n) instead of O(2n+5))

For each piece of code, state the running time and then give a short explanation of why that running time is correct. We are not asking for a formal proof--you'll learn how to do that in CSE 101. For now your explanation should simply include an (approximate but reasonable) equation for how many instructions are executed, and then a relationship between your equation and your stated bound. Place your answers in in your **HW3.txt** file in a section labeled Part 1b.
Here is an example:

Ex.

```
for ( int i = 5; i < n; i++ )
  sum++;
```

Most precise answer:
Running time O(n)
Explanation: There is a single loop that runs n-5 times. Each time the loop runs it executes 1 instruction in the loop header and 1 instruction in the loop header and 1 instruction in the body of the loop. The total number of instructions is 2*(n-5) + 1 (for the last loop check) = 2n-9 = O(n).

A slightly less precise but still acceptable for full credit answer:
Running time: O(n).
Explanation: There is a single loop that runs n-5 times. Each time the loop runs it executes 1 instruction, so the total number of instructions executed is 1* (n-5) = O(n) (also OK: Θ(n)).

```
1. for ( int i = 0; i < n; i+=2 )
       sum++;

2. for ( int i = 1; i < n; i*=2 )
       sum++

3. for ( int i = 0; i < n; i++ )
       for ( int j = 0; j < n; j++ )
           sum++;

4. for ( int i = 0; i < n; i++ )
       sum++
   for ( int j = 0; j < n; j++ )
```

```
       sum++
   // The above are two loops one after the other, NOT nested

5. for ( int i = 0; i < 2*n; i++ )
       sum++

6. for ( int i = 0; i < n*n; i++ )
       sum++;

7. for ( int i = 0; i < n; i++ )
       for ( int j = 0; j < n*n; j++ )
           sum++;

8. for ( int i = 0; i < n; i++ )
       for ( int j = 0; j < 10000; j++ )
           sum++;
```

Part 2: Running time of Linked List functions (20 points)

In this section, you will analyze minimum time needed to implement several operations on a Linked List. Hopefully these minimum times should match your *actual* implementation that you created in HW2 (though of course a bad implementation can run arbitrarily slowly).

In your **HW3.txt** file, in a section labeled Part 2, give the minimum required running time to execute each of the following operations in a doubly linked list with head and tail pointers *in the worst case*, using Big-Ω notation, assuming n is the number of elements in the linked list. Following each expression, include a 1-2 sentence argument about why an algorithm to perform the given operation could not run faster than the bound you give. As above, full credit will be given only for tight Big-Ω bounds. That is, it is not sufficient to say that all operations take Ω(1). This is trivially true for any piece of code. (Though in some cases this will be the tightest Big-Ω bound).

As above, the first problem is given as an example:

Ex. Adding a value to the start of the list
Running time: Ω(1)
Explanation: You have a pointer to the head node in the list, so adding an element involves creating a new node (which is not dependent on the length of the list), setting the links in the new node, and changing the values of the head references. This takes somewhere around 10 steps to perform, and 10=Ω(1).

1. Making a copy of the list
2. Adding a value to the end of the list
3. Removing the first value from the list

4. Removing the last value from the list
5. Determining whether the list contains some value V

Part 3: Runtime Exploration (38 points)

Credit: Part 3 based on CSCI 151 - Lab 4 at Oberlin College, by Benjamin Kuperman.

For this last part you will need to use your MyLinkedList implementation as well as the provided MRUList class (defined in MRUList.java). You are also welcome to use our implementation of MyLinkedList, which will be provided on Thursday (after the slip day deadline has passed). You will also need the provided CollectionTime.java file, as well as at least the small and medium wordlist files. IMPORTANT: If you choose to use your own implementation you will need to make sure that the head field is protected and not private or MRUList will not compile.

Looking at MRUList.java, you will notice that **`MRUList<T>`** is a Most Recently Used (MRU) List which is derived from your existing MyLinkedList class. The idea behind a MRU list is that when an item is looked up in a list, it is often looked up again in the near future. To try and improve lookup times, whenever an item is "found" in the list, it is moved to the front of the list so that subsequent searches for it might be faster.

Notice that to achieve the MRU behavior, MRUList overrides a few inherited methods:

public boolean add(T x)
public void add(int index, T x)

Any item that has been added is considered to have been recently found, so just found items are always added at index 0.

public boolean contains(Object o)

Each time you check to see whether an MRUList contains an element, you remove that element from its position in the list and add it to the front. Note that we do this by accessing and modifying the internal Node objects, rather than using calls to remove and add, to avoid the overhead of creating and destroying Node objects.

In this part you will use a provided class called **`CollectionTimer`** that will let you compare the running time of using your `MyLinkedList` and `MRUList` to do a spell checking task.

What the `CollectionTimer` does is read in a list of known "good" words which it stores in a collection such as your `MyLinkedList`. It then reads a fixed number of words from a second file and checks to see if they are contained in the "good" list or not. The program will keep track of the number of words that matched or not, but that isn't displayed unless you enable debugging information in the `CollectionTimer`. Instead, it keeps track of the number of milliseconds that have elapsed during the performance of this task. It only starts timing once it is doing the word list lookup, so setup time is not included.

The program takes 6 arguments as described below. The first 2 are required. The other 4 are optional and are used to change the amount of work performed for each iteration.

1. The name of the dictionary file
2. The name of the document to be checked
3. The number of words to initially read from the document (Default: 5000)
4. The number of words to increase by for each run (Default: 5000)
5. The number of times to increase (Default: 5)
6. The number of times to run each test (average time reported) (Default: 5)

Here is the output on the lab machines with the default parameters:

```
[cs12e@ieng6-250]:HW3:557$ java CollectionTimer small-wordlist.txt
pride-and-prejudice.txt
Wordlist: small-wordlist.txt  Document: pride-and-prejudice.txt
Class: MyLinkedList
======================================
  1: 5000 words in   651 milliseconds
  2:   10000 words in1307 milliseconds
  3:   15000 words in1976 milliseconds
  4:   20000 words in2449 milliseconds
  5:   25000 words in3860 milliseconds

Wordlist: small-wordlist.txt  Document: pride-and-prejudice.txt
Class: MRUList
======================================
  1: 5000 words in   308 milliseconds
  2:   10000 words in 548 milliseconds
  3:   15000 words in 964 milliseconds
  4:   20000 words in1356 milliseconds
  5:   25000 words in1412 milliseconds
```

We've included a number of wordlists for you to try comparing against. We also included a copy of "Pride and Prejudice" from Project Gutenberg which has 121557 words which should contain more than enough text for you to test against for your loops. The wordlist files have the following word counts:

- small-wordlist.txt (13566 words)
- medium-wordlist.txt (119805 words)
- large-wordlist.txt (339884 words)
- huge-wordlist.txt (653669 words)

You should not need to make any modifications to CollectionTimer, but we will ask you to look at this code and answer questions about it in Question 1. There is a debug flag that you can enable to let you see some of the inner workings. This might be useful if you want to see some of the details of the inner workings.

Now, in your **HW3.txt** file, answer the following questions in a section labeled Part 3:

Question 1

Before running any of the benchmarks, please answer the following to help you better understand the CollectionTimer code itself. You will need to read the code to understand it. Generating the javadoc for CollectionTimer and looking at it may assist you.

A. What is a unix command that could be used to determine the number of words in the medium-wordlist.txt file? Show the output of that command and point out which part of the output gives the number of words.

B. Write a method call "outline" for CollectionTimer. The idea is to strip away the detail of the code and just see in what order the methods (just those *defined* in CollectionTimer) are being invoked. Indent when methods are called within other methods. It is up to you, but often it is useful to write down the critical arguments given to methods. The goal of the outline is to "de-mystify" what CollectionTimer is doing. Too much detail makes it so that you can't see the big picture of the code, too little detail and the outline isn't helpful. What we are looking for is the proper order of methods (and indentation), but you may want to add some detail or notes for your own understanding.
This part is fairly open-ended and there's no one "right" answer, so asking "is this right?" is not likely to be productive. What we are looking for is enough of a skeleton that we can see that you understand what's going on in the CollectionTimer class. In particular, you might want to add some high-level explanation of the main behavior chunks within the methods themselves. If you have a good understanding of how the code is running and your outline communicates that understanding, you'll get credit for this part. To get you started:

```
main()
        doLoops(MyLinkedList, wordlist, input file, loop bounds and steps)
                … TODO: fill out the methods called from within doLoops()
        doLoops(MRUList, <same parameters as previous call)
                <same call outline as above, detail is not needed>
```

C. Where is the class HashSet defined? What interface that we have looked at does HashSet implement? You should also look at the printf statements that use the variables good and goodwords.size() and then look at how good is incremented in the code. After you run the code in the next steps, come back to this question and write a one-sentence reason as to why you think good and goodwords.size() are different.

D. The small wordlist is approximately sorted alphabetically, do you think it makes any difference to the overall performance that you will observe? Why or why not (please, just a few sentences as to why you reached your particular conclusion)

Question 2

Copy and paste the output from one run of CollectionTimer using small-wordlist.txt and pride-and-prejudice.txt with default for the rest. If you find that it is running too quickly/slowly, you may need to modify things from the default parameters. If you do so, be sure to document what settings you are using in your writeup. Try and have the number of lookups double, and then double again, to get a good range of observations.

Question 3

The running time for this task should be linear in terms of the number of items in our wordlist. (To determine a "miss" you would need to look at every word in the list.) Let's call this n. It is also linear in terms of the number of words to be read in. Let's call this m. Taken together, you might express this as O(m n).
Looking at the table you've generated, as the size of m doubles, you would expect the worst case running time to also double. Does this hold for your observations? Be sure to consider MyLinkedList and MRUList separately. Why do you think this is?

Question 4

Now re-run your experiment from Question 2, but this time using the medium-wordlist.txt. Include this table in your **HW3.txt**. WARNING: This will take a while, and you probably want to set the number of times to run each test to something smaller than the default.

This wordlist is about ten times the size of small-wordlist.txt. How does this change the performance of each type of list? Why do you think this is?

Question 5

Now re-run your experiment but use pride-and-prejudice.txt as both the wordlist and the document to be checked. Include this table in your **HW3.txt**. What do you observe about the running times now? Why do you think this is?

IMPORTANT NOTE: The performance of these tests can be highly dependent on your specific machine, your memory set up, how Java is running on your machine, how many other people are using the computer, how many other programs you're running. If you are getting strange behavior for this part try the following:

- Log in and run your code on one of the lab machines. This is strongly recommended, though remember, if you're doing this at the last minute, chances are the machines are under a heavy load, so you might still get strange behavior
- Close all other programs on your computer

Turning in your assignment

For this assignment you will only turn in your **HW3.txt** file using the command **bundleHW3**.