Project 2 – MSCI 240, Fall 2016[1]

Assigned November 14th, Due November 22 at 12:20pm.

Instructions

One way that software companies use to investigate usage of their applications is called *telemetry*. Telemetry data is data collected from use of an application and sent to the company. You've probably seen applications ask for your permission to send this data. When you do, it's collected and analysed by a data team. If you've ever used the Mozilla Firefox browser, this data is often made public so that anyone can analyze these usage patterns. Here are some examples of analysis: http://www.icir.org/christian/testpilot/, https://blog.mozilla.org/ux/2012/06/firefox-heatmap-study-2012-results-are-in/. In this project, you will use data that I downloaded from here: https://datahub.io/dataset/a-week-in-the-life-of-a-browser-version-2 and count the frequency of events that came from real Firefox users. Specifically, you will investigate: (a) how many bookmarks they had in their bookmarks folder, and (b) how many tabs they kept open. However, this purpose is secondary to the primary purpose of the project, which is to investigate the performance of 3 different dictionary designs to determine frequencies:

1. A dictionary implemented with an unsorted list. Your unsorted list should be of type `ArrayList<ValueFrequency>` where `ValueFrequency` is a class with at least an integer to hold the value being considered (number of bookmarks or number of tabs) and an integer to hold the value's frequency in the dataset (number of events that have that many bookmarks/tabs):

   ```
   public class ValueFrequency {
       public int value;
       public int frequency;
   }
   ```

   You may make `ValueFrequency` different or more complex as needed, but the above is a minimum implementation. **You should not maintain a separate list for values and separate list for frequencies.** For information on using the `ArrayList` class, see: https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

2. A dictionary implemented with a binary search tree. Your dictionary should be of type: `TreeMap<Integer, Integer>`. See: https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html
   Aside: this is actually a Red-Black Tree, which is a type of binary search tree: https://en.wikipedia.org/wiki/Red%E2%80%93black_tree

3. A dictionary implemented with a hash table. Your dictionary should be of type `HashMap<Integer, Integer>`. See: https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

Please note that the manual pages for the ArrayList, TreeMap, and HashMap classes provide lots of information and even examples! One of the goals of this project is for you to gain experience with learning to use built-in classes from a class libary and the work required to learn how to use these classes by reading available documentation. We have not taught you how to use these classes beyond teaching you the concepts of how each is implemented and the performance to expect with such implementations. This project simulates the real world where you have to do this sort of self-learning on a regular basis (e.g., you know you want to use a dictionary implemented as a hash table, but you don't know the details of how it is made available to you in the current programming language of choice).

We have provided a dataset from Firefox and some code that will read this dataset and convert them into a long list of events. You will go through this list from index `0` to index `events.size()-1` and use a dictionary implementation to count the number of events that refer to (a) each quantity of bookmarks and (b) each qauntity of tabs (see sample output below). You will then output the 20 most frequently occuring values for both (a) and (b) in descending order.

[1] This project is a (heavily) modified version of a previous project by Mark Smucker, used with permission

Step-by-step

1. Download p2-code.zip and p2-data.zip from Learn and unzip.
2. Create a new Java project in Eclipse.
3. Copy the "project" directory from the unzipped p2-code.zip contents to the "src" folder.
4. Copy the events.csv and events_small.csv from the unzipped p2-data.zip to the root folder of the project (whatever you named it).
5. Compile the program.
6. Run the program. It should output:

```
Processed 5,000,000 events so far...
Processed 10,000,000 events so far...
Finished reading 12,778,416 events.
```

7. For each of the 3 dictionary implementations discussed above, do the following:
   a. In the part of the `main` method that says "// TODO: Add your code here...", add code that uses the dictionary to count the number of events with (a) the same number of tabs and (b) the same number of bookmarks. You should only iterate through the events list (not modify it):

```
for ( int i = 0; i < events.size(); i++ ) {
    // increment count in dictionary for events.get(i)
    // for bookmarks and tabs
}
```

   b. Using the code and good practices learned in Lab this term, time how long it takes you to count the frequency of events with specific numbers of bookmarks and tabs using the dictionary. You should only measure the time of the loop shown in 7a above, and not the time it takes to read the events into the events list.
   c. Collect the following information for the dictionary **and output it to the Console**:
      i. how many unique values there were of number of bookmark and number of tabs
      ii. how long it took to count these events with the dictionary
      iii. The top 20 values of bookmarks sorted by frequency in descending order. For example, one solution program outputs **on the small event list**:

```
Finished reading 202,460 events.
To count the events with unsorted lists took 0.053258193 seconds.
Found 242 unique events.
Found 120 unique tab events.
The 20 most frequent number of bookmarks and their counts are:
9       650
11      551
13      386
10      375
12      373
14      332
21      292
18      287
31      247
33      240
22      218
15      211
23      206
17      197
30      183
34      178
19      170
35      167
24      167
16      159
```

8. Write a report. See the following for information about the report.

Report

Implement, test, and debug your program until you believe it is working correctly. Visit the TAs for help.

Write a report regarding your project. You report should be typed and consist of the following parts:

1. A cover page that includes the following information:
    a. Your name.
    b. The course info: "MSCI 240 Fall 2015, Instructor: Hancock"
    c. The date on which you are submitting your assignment.
    d. Which assignment this is.
2. A report consisting of the following sections:
    a. Introduction: A brief summary of the whole report.
    b. A section for each dictionary implementation that reports:
        i. Based on your code and the manual pages of the classes, give an explanation of the worst case growth rate of the code needed to count frequencies of bookmarks and tabs given the dictionary. Be sure to note what information in your report is from the manual pages and what is based on your own analysis of your code.
        ii. The data collected regarding empirical time to count. In addition to reporting the collected data, you should explain all of the steps you took to obtain accurate measurements.
        iii. Assuming you found that same number of unique frequencies and top 20 frequencies, report this information once (no need to repeat identical information for all dictionaries), **but if your dictionaries produced different answers, you must truthfully report the differences.**
    c. A discussion about the performance of each dictionary and your opinion about what is the correct design choice for this problem.
    d. A discussion about how you selected the method to determine the top 20 frequencies of bookmarks and what criteria you used to make that design choice.
    e. A brief discussion about any issues you encountered in completing the project.
    f. Acknowledgments. You must acknowledge all help received. You must cite and quote all sources of material that you did not create yourself.
    g. Help Given. Please explain any help that you gave to other students.
    h. Your code. Provide a printout from Eclipse of your code including line numbers. Be sure to print out the final version and not some version that does not match what you will upload to Learn. Please do not print from Notepad. Please do not copy and paste into MS-Word. Print from Eclipse.

All pages of your report must be letter sized pieces of paper. You must staple multiple pages together with one metal staple in the upper left hand corner. Your stapling should be neat and tidy. Mangled staples are not permitted. Do not use report binders. You are welcome to use duplex printing to save paper.

What to hand in:

1. You should copy and rename your program files (the **.java** files, often named something like Program.java) to be nexususername-proj2-Filename.java where nexususername is your nexus username and Filename.java is the original filename of the code, and **upload it to the UW-Learn** drop box for Project 2. Do not make mistakes naming your file. We will compile your program and run it.
2. A soft-copy of your report should also be turned into the Learn dropbox. Your report should be a Microsoft Word or Adobe PDF document.
3. Turn in a printed copy of your project report as per above. You should hand in this paper portion of the assignment to the dropbox on the **third** floor between E2 and CPH.

Failure to upload your file correctly will result in a significant loss of points. The timestamp that UW-Learn gives your upload will determine when you have turned in the homework, but your hardcopy version must also be placed in the dropbox on time.

An Important Note about Finding the 20 Most Frequently Number of Bookmarks

As with most problems, there are multiple solutions to finding the 20 most frequent number of bookmarks. In this project, you first are going to count the number of events that have each quantity of bookmarks. You will then have a dictionary that pairs together a quantity of bookmarks and its count (frequency), but the dictionary itself will not help you find the 20 most frequently occurring numbers of bookmarks.

The best solution is to use a priority queue implemented with a min heap (http://en.wikipedia.org/wiki/Priority_queue). We will discuss heaps before the end of the term, but we have not yet done so. Java provides a PriorityQueue class which can be used to solve this part of the project:
http://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html

To use this class, you will need to have a firm grasp of the discussion below on Comparator and Comparable. Alternatively, in place of Java's default implementation of a priority queue that uses a heap, we could use a balanced binary search tree (BST) to implement a priority queue. The BST will keep the top scores in sorted order for us and it will be possible to find the minimum, delete it if needed, and add a new element all in $O(\log n)$ time. An advantage of the heap over the BST is that the heap can be easily implemented and stored efficiently using an array while the BST is a linked data structure that uses more space.

A naive priority queue implementation could be had by using an array of `ValueFrequency` objects. You would operate on the array as you did for your high score table in Project 1. The first 20 value-frequency pairs would be inserted into the array to maintain the sorted order. Then for each additional word, you would decide if it was a new top 20 frequency or not. In fact, if your Project 1 code works, you should be able to use it with a little modification. This is a naive implementation of a priority queue because the insertion costs $O(n)$ where n is the number of elements in the queue. Heaps lower the insertion cost to $O(\log n)$. But, your queue has a small n, and the naive solution should be plenty fast.

Alternatively, you could write some code to iterate through all of the key-value pairs in a dictionary and find the maximum frequency. Once you have that value of bookmarks, you could iterate through all the pairs again, but this time finding the maximum that is less than or equal to your previously found max. You would repeat this process until you had the top 20 frequencies. This isn't too bad for only the top 20, and this process of extracting the max, and then the max of the remaining elements, and so forth, is essentially selection sort (and we know from class why it's not the best choice). I suspect that a naive priority queue would be faster than this method, given that we only need to take one pass over the word count pairs with the priority queue, while this selection of the max process requires 20 passes over the pairs. In other words, the worst case for the naive priority queue involves comparing each value-frequency pair to 20 other pairs (to a first order approximation), which is equal to the number of comparisons that the selection process does (it compares a max value to each value-frequency pair 20 times). But, in reality it is likely that we'll do fewer than 20 comparisons with the priority queue for each pair. The max selection process will always do the same number of comparisons. Thus, the naive priority queue has the upside of possibly running faster with no downside compared to the max selection process.

Most students, even after completing Project 1, will never think of implementing a priority queue. Instead, their natural solution to the problem is to want to sort all of the value-frequency pairs in either ascending or descending order given the frequencies. Once sorted, the 20 most frequently occurring values will be at one end of the list.

Unfortunately, the dictionary classes cannot help here. Even the TreeMap, which uses a binary search tree, will not help, since they maintain the words in sorted order, not the counts (unless you make effective use of the Comparator class, discussed below).

An Important Note about Comparing Objects

To sort all of the value-frequency pairs, we first need to copy them out of the dictionary and into a Java Collection. We could declare such a list as follows:

```
ArrayList<ValueFrequency> counts = new ArrayList<>() ;
```

where `ValueFrequency` is the class as given above that pairs a value (int) with frequency (int). Once we have all of the values and their frequencies in a list, we need some way of sorting the list by the frequency field of the `ValueFrequency` class.

If a Java Collection is a List of some simple type such as an int or double, we can call the sort method from the Collections class. For example:

```
ArrayList<int> myNumbers = new ArrayList<>();
myNumbers.add(6);
myNumbers.add(1);
// etc.
Collections.sort(myNumbers) ;
```

Unfortunately, if you try this with `ValueFrequency` objects, it will not work right away. This code:

```
ArrayList<ValueFrequency> vf = new ArrayList<>();
vf.add( new ValueFrequency( 5, 20 ) );
vf.add( new ValueFrequency( 9, 5 ) );
Collections.sort(vf);
```

Will not compile with the explanation "`The method sort(List<T>) in the type Collections is not applicable for the arguments (ArrayList<ValueFrequency>)`"

The problem is that to sort items, we need some way to compare the items. Java knows how to compare ints and doubles (every time you do < or > or <= or >= or == or !=, you're doing comparisons). If we want to sort `ValueFrequency` objects, we need to choose between one of two options:

1. Have `ValueFrequency` implement the **`Comparable`** interface. See: https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html
2. Provide a **`Comparator`** object as a parameter to the Collections.sort method, that knows how to compare two `ValueFrequency` objects. See: https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

In both cases, the solution amounts to **overloading** a method called `compareTo` that can compare two `ValueFrequency` objects and report whether one is less than, equal to, or greater than the other. You will get / should have had practice doing this in Lab 4. We have not talked about overloaded methods in Java. An overloaded method is actually multiple methods that share the same name. We can tell the difference between the methods because the object it belongs to and the arguments that are passed to the methods are different. Another example of a method that you may have overriden is the `toString` method. The various `compareTo` methods give you different ways to tell sort how to compare two objects.

The `compareTo` function used to compare two objects A and B behaves according to the convention that if A is less than B, then our comparison function returns a value less than zero, if A is equal to B it returns 0, and if A is greater than B it returns a value greater than zero. Full details can be gleaned from either the `Comparable` or `Comparator` Java API documentation provided above.

We want to compare the counts of two `ValueFrequency` objects. We could do this by first changing the class definition for `ValueFrequency` as follows:

```
public class ValueFrequency implements Comparable<ValueFrequency> {
    // ...
}
```

And then writing the following method inside the WordCount class:

```
        @Override
        public int compareTo( ValueFrequency other ) {
            return Integer.compare( this.frequency, other.frequency) ;
        }
```

To make our life easy, since integers already have a built in `compare` method, we compare the count of the current `ValueFrequency` object (`this.frequency`) to the other `ValueFrequency` object passed in (`other.frequency`) as shown above. This makes the above code that wouldn't compile work and sorts the array by frequency.

Is sorting and finding the top 20 words better than a naive priority queue implementation? I leave that to you to decide and discuss in this project.

Collaboration and Academic Honesty

1. Please review the syllabus for guidelines and extensive rules on academic honesty in this course.
2. You are to work alone on this assignment except that:
    a. You may talk with the instructor and the TA regarding this homework.
    b. You may talk with students about general programming questions, but you may not discuss ways to solve this homework. For example, you may have another student help explain loops or conditionals or variables to you, but you may not discuss how to write parts of a program to solve this homework. **You may discuss how to use the Java classes.**
3. All collaboration other than that allowed above (2a, 2b) is forbidden. For example:
    a. You may not discuss the problems with people.
    b. You may not sketch out pseudo-code on a whiteboard.
    c. You may not share code with others.
    d. You may not give answers to others.
    e. You may not compare answers with others.
4. In addition, you may not use any materials other than your textbook or books you can find in the library and the Java documentation. You must cite any source that you use. You may not search for answers on the internet, but **you are allowed to search the internet for help with the Java classes**.
5. We will use automatic methods to detect cheating as well as human inspection of solutions.
6. Violating these rules will be considered academic misconduct and subject to penalties.

Marking Rubric

Correctness of the Java code is important. Incorrect code will prevent you from obtaining correct answers to the experiments. 40 points will be awarded for correct implementation. 40 points will be awarded for analytical growth rate explanations, good empirical measurements, and a high quality discussion of the results. 20 points will be awarded for good Java style and report presentation quality and other miscellaneous attributes as needed.