P05 Simple Benchmarking

Overview

Analyzing runtime complexity of a program can, as you're learning this week, take the form of a theoretical analysis expressed in Big-O notation and mathematical formulas, but it can *also* be a more informal measure of "how long does this program take to run on controlled inputs?" Broadly, this timed version is called **benchmarking**.

There are many advanced tools for profiling system usage and time for algorithmic components, but in this assignment you will focus on strict runtime comparisons: how many milliseconds elapse between calling a method and receiving a return value?

Grading Rubric

5 points	**Pre-assignment Quiz**: Accessible through Canvas until 11:59PM on **03/15**.
20 points	**Immediate Automated Tests**: Accessible by submission to Gradescope. You will receive feedback from these tests *before* the submission deadline and may make changes to your code in order to pass these tests. Passing all immediate automated tests does **not** guarantee full credit for the assignment.
15 points	**Additional Automated Tests**: These tests will also run on submissions to Gradescope, but you will not receive feedback from these tests until after the submission deadline.
10 points	**Manual Grading Feedback**: TAs or graders will manually review your code, focusing on algorithms, use of programming constructs, and style/readability.

Learning Objectives

The goals of this assignment are:

- Implement multiple algorithms to solve the same problem
- Learn a simple benchmarking technique using built-in Java methods
- Practice file I/O and exception handling

Additional Assignment Requirements and Notes

Keep in mind:

- You are allowed to define any local variables you may need to implement the methods in this specification.
- All methods, public or private, must have their own Javadoc-style method header comments in accordance with the CS 300 Course Style Guide.
- Any source code provided in this specification may be included verbatim in your program without attribution.
- You are allowed to import ONLY the following classes:
  - java.io.File
  - java.io.FileWriter or java.io.PrintWriter
  - java.util.Random
  - java.util.Scanner
  - Any relevant exceptions
- You are allowed to define additional **private** helper methods to help implement the methods in this specification.
- Do not add any global constants or variables other than those listed in this specification.
- Note that this assignment does NOT include a required tester class! We still strongly recommend that you test your program's correctness; remember that Gradescope includes test cases that are not available to you until after grades are published.

CS 300 Assignment Requirements

You are responsible for following the requirements listed on both of these pages on all CS 300 assignments, whether you've read them recently or not. Take a moment to review them if it's been a while:

- Academic Conduct Expectations and Advice, which addresses such questions as:
  - How much can you talk to your classmates?
  - How much can you look up on the internet?
  - What do I do about hardware problems?
  - and more!
- Course Style Guide, which addresses such questions as:
  - What should my source code look like?
  - How much should I comment?
  - and more!

Getting Started

1. Create a new project in Eclipse, called something like **P05 Benchmarking**.
   a. Ensure this project uses Java 11. Select "JavaSE-11" under "Use an execution environment JRE" in the New Java Project dialog box.
   b. Do **not** create a project-specific package; use the default package.
2. Create three (3) Java source files within that project's src folder:
   a. **SimpleBag**.java (instantiable; does NOT include a main method)
   b. **CleverBag**.java (instantiable; child class of SimpleBag; does NOT include a main method)
   c. **Benchmark**.java (includes a main method)
3. Download the sample data file, frank.txt, and add it to your project.

All methods and fields in the Benchmark class will be class methods (**static**); methods and fields in the Bag classes will be (non-static) **instance** methods.

Implementation Requirements Overview

Your **Benchmark** class must contain the following methods. Implementation details, including required output formatting, are provided in later sections.

- `public static String compareLoadData(File f, SimpleBag s, CleverBag c)`
  - Runs both classes' loadData() implementations on the same text file.
  - Tracks the time spent in milliseconds to complete each loadData().
  - Returns a formatted String with the elapsed times for each of the bag types.
- `public static String compareRemove(int n, SimpleBag s, CleverBag c)`
  - Runs both classes' removeRandom() method n times .
  - Tracks the time spent in milliseconds to complete each type of remove
  - Returns a formatted string with n and the elapsed times for each of the bag types.
- `public static void createResultsFile(File in, File out, int[] nValues)`
  - Creates one instance each of a SimpleBag and a CleverBag.
  - Calls compareLoadData() to compare the two different data loads using the `in` parameter.
  - Calls compareRemove() on each of the provided nValues to compare the two different remove implementations.
  - Writes the results of the data load comparison followed by the remove comparisons to a file specified by the `out` parameter.
  - Handles any exceptions raised by the methods it uses.

Benchmark.java should also contain a `main` method, which you can use to test your code.

The classes you'll be comparing are detailed on the **next page**...

Your **SimpleBag** class must contain the following methods. Any referenced data fields must be **instance** fields unless otherwise stated. Implementation details are provided in later sections.

- `public SimpleBag(int seed)`
  - Initializes a protected field, `data`, which is an array of Strings with capacity 80,000. We will not provide files with more than 80,000 words.
  - Initializes a protected Random object, `random`, using the provided seed value.
- `public void loadData(File f)`
  - Reads the text contents of the provided file, inserting each new space-separated word at the *beginning* of the `data` array.
  - All strings currently in the array should be shifted to the **right** by one index to make room. That is, the string at index N should be moved to index N+1, and so forth.
  - If you encounter any exceptions while reading the File, simply return from the method.
- `public String removeRandom()`
  - Counts the number of Strings (i.e. non-null) values in the data array and generates a random index between 0 and the number of Strings stored in this bag (exclusive)..
  - Removes and returns the String at that index.
  - Fills gaps by moving all following strings to the **left** by one index. N -> N-1, etc.
  - If the bag contains no strings, this method returns null.

Your **CleverBag** class must inherit from the SimpleBag class and contain the following methods. Any referenced data field must be **instance** fields unless otherwise stated. Implementation details are provided in later sections.

- `public CleverBag(int seed)`
  - Calls the super class' constructor with appropriate arguments.
  - Initializes the private integer field , `size`, which will track the current number of initialized Strings in the parent class' `data` array.
- `@Override`
  `public void loadData(File f)`
  - Reads the contents of the file as in the parent class, but instead inserts the new words at the *end* of the array and then updates the `size` field accordingly.
  - If you encounter any exceptions while reading the File, simply return from the method.
- `@Override`
  `public String removeRandom()`
  - Generates a random integer between 0 and the current `size`.
  - Removes and returns the String at that index.
  - Fills gaps by moving the last String into the gap and decrementing `size`.
  - If the bag contains no strings, this method returns null.

Implementation Details and Suggestions

We strongly recommend creating another, shorter text file with the same formatting as frank.txt to help you verify that your **loadData()** methods are working. The first line should be an integer count of the number of words in the file and the rest of the file should be space-separated, punctuation-free words. You may assume that the number on the first line is always an accurate word count.

SimpleBag

We recommend starting with the SimpleBag class, since (**a**) it's Simple and (**b**) CleverBag inherits from it.

First implement the **constructor**, as detailed above, like you would for any instantiable class. (Don't forget to declare the fields mentioned there too!)

Then move onto the **loadData()** method. Use the File parameter to create a Scanner. Remember the formatting of the file! The first line is the total word count (so you can verify that you got everything), and every line after that contains space-separated words to be added to `data`. Use any Scanner methods you like, but make sure to load every word before you end the method. Handle any exceptions you need to here; *don't* throw them.

> ⇛ **STOP**: Make sure loadData() is working correctly before you continue. Using a shorter file will help with debugging, we promise. You can add a main method to the SimpleBag class for testing purposes, since we don't have traditional accessor methods, but **remove it before you submit**!

Finally, implement **removeRandom()**. Use the Random.nextInt(int bound) method to generate a random index once you've figured out what the bound should be, and remove the string at that index per the details provided above. Again, **STOP** and verify that everything works!

Next onto better (and more efficient) things...

CleverBag

CleverBag does the same tasks as SimpleBag, but better (using a different algorithmic approach).

Write the constructor according to the specifications. (Don't forget to declare the `size` field!)

Now, override the **loadData()** method. CleverBag will also read data from the text file (so it'll start the same way), but pay attention to *how* you're storing the data in the array. Again, handle the exceptions here; don't throw them. Once you're confident you've followed the spec, **STOP** and test it out.

Finally, override **removeRandom().** Again, the idea of this method is similar to SimpleBag's; the only difference is *how* you remove from and fill holes in the data array.

Now comes the actual benchmarking bit, where you'll compare the performance of these two bags...

Benchmark: compareLoadData()

The purpose of the **compareLoadData()** method is to call each of the Bag classes' loadData() methods on the same input File and compare how long it takes each one to complete.

Use the `System.currentTimeMillis()` method to get the current system time in milliseconds. If you record the time both before and after you call a method (or multiple methods), you can use simple arithmetic to see how much time has elapsed.

Record the elapsed time for the SimpleBag and the CleverBag loadData() methods, and return the values in a tab-separated String with a newline terminator, formatted as follows:

```
"load:\t" + simpleBagLoadTime + "\t" + cleverBagLoadTime + "\n"
```

For example, the output from this method might be the string:

```
"load:98    15\n"
```

Benchmark: compareRemove()

The purpose of the **compareRemove()** method is to call each of the Bag classes' removeRandom() methods n times, and compare how long it takes each one to complete those calls.

Use the `System.currentTimeMillis()` method to get the current system time in milliseconds. If you record the time both before and after you call a method (or multiple methods), you can use simple arithmetic to see how much time has elapsed.

Record the total elapsed time for all calls to the SimpleBag and the CleverBag removeRandom() methods, and return the totals in a tab-separated String with a newline terminator, formatted as follows:

```
queries.length + "\t" + simpleBagRemoveTime + "\t" + cleverBagRemoveTime + "\n"
```

For example, for an n of 100 (meaning you remove 100 random words from each bag), the output from this method might be the string:

```
"100    28    0\n"
```

Benchmark: createResultsFile()

This method creates the actual objects, passes them to each of the compare methods, and writes the results to a file.

1. Create a SimpleBag and a CleverBag object.
2. Pass the `in` File and bags to the `compareLoadData()` method, and save the output.

3. Pass each of the `nValues` and bags to the `compareRemove()` method, and save each line of output.
4. Use the `out` File parameter to open a FileWriter or PrintWriter and write the results of 2 and 3 to the file (compareLoadData()'s first, then each of the compareRemove() results in order).
5. Close the FileWriter or PrintWriter.

For example, for nValues of `{10, 100, 1000, 10000}` your output file might look like this:

```
load:   98     15
10      12     0
100     34     0
1000    148    1
10000   1352   1
```

Any exceptions encountered when writing to the file should be handled *without crashing* within the `createResultsFile()` method.

Use the Benchmark class' main method to create some test query word arrays and run the methods to verify that everything works as expected.

Feel free to add any printed console output that you like for debugging purposes - we will only be testing return values and file contents.

Complexity Commenting

In addition to our usual commenting requirements, add the following line to the Javadoc for each of the methods in the Bag classes:

```
/**
 * Complexity: O(___)
 */
```

Include this line for both the `loadData()` and `removeRandom()` methods in SimpleBag and CleverBag, filling in the blank with the appropriate *worst-case* Big O notation complexity for the algorithm in the method.

This value will be manually graded!

Assignment Submission

Once you're satisfied with your work, both in terms of adherence to this specification and the academic conduct and style guide requirements, submit your source code through Gradescope.

For full credit, please submit ONLY the following files (source code, *not* .class files):

- SimpleBag.java
- CleverBag.java
- Benchmark.java

Your score for this assignment will be based on the submission marked "**active**" prior to the deadline. You may select which submission to mark active at any time, but by default this will be your most recent submission.

Copyright Notice