P03 Simple Benchmarking

Overview

Runtime complexity can be a formal analysis of the number instructions in an algorithm, but it can also be a more informal analysis of the time it takes to execute an algorithm. Broadly, this time analysis 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 9:59PM on **09/27**.
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 will also run on submission 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 two algorithms for solving 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 may not have any import statements besides java.io.FileWriter or java.io.PrintWriter, java.io.File, and the relevant exceptions.
- You are NOT allowed to add any constants or variables outside of any method.
- You are allowed to define any local variables you may need to implement the methods in this specification.
- You are allowed to define additional **private static** helper methods to help implement the methods in this specification.
- You are NOT required to implement any test methods for this program, but we strongly recommend that you test your program's correctness.
- 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.

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 **P03 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 two (2) Java source files within that project's src folder:
   a. ComparisonMethods.java (does NOT include a main method)
   b. Benchmarker.java (includes a main method)

All methods in this program will be **static** methods, as this program focuses on procedural programming.

Implementation Requirements Overview

Your **ComparisonMethods.java** program must contain the following methods. Implementation details are provided in later sections.

- `public static long bruteForce(long n)`
  - Calculates and returns the sum of all integers 1 to n
  - Uses a loop and running total to calculate
- `public static long constantTime(long n)`
  - Calculates and returns the sum of all integers 1 to n
  - Uses a formula to calculate

ComparisonMethods.java must NOT contain a main method.

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

- `public static String compare(long n) throws NoSuchElementException`
  - Runs both methods from ComparisonMethods.java on the same n
  - Tracks the time spent in milliseconds to complete each method
  - Returns a formatted string with n and the elapsed times
  - Throws a NoSuchElementException with a descriptive error message if the return values of the two comparison methods are different
- `public static void createResultsFile(File f, long[] queryNs)`
  - Calls compare(n) using each of an array of values
  - Writes the results to a file specified by the parameter
  - Handles any exceptions raised by the methods it uses

Benchmarker.java should also contain a main method, which you can use to create a sample array of values, open a java.io.File and call createResultsFile() to test your code.

Implementation Details and Suggestions

The first class, ComparisonMethods, is not intended to be difficult to implement. It is merely an illustration of multiple ways to solve a single problem with very different runtime efficiencies.

As such, both methods described here should, given identical inputs, produce identical results. We will manually inspect your code to ensure that you've implemented them as described here.

Note: while we're referring to integer values, all values in this class must be **long**s. In order to produce interesting results in this comparison, we'll have to use numbers so large that int primitives are too small to hold them.

Brute Force

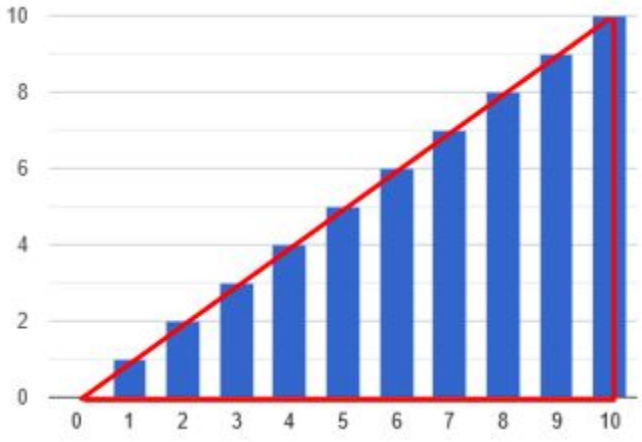
The first implementation employs what algorithm analysis refers to as a "brute force" algorithm. Brute force algorithms prioritize ease of understanding over efficiency.

When asked to add up all the numbers between 1 and N, this is likely the first approach you would think of: counting from 1 to N and adding each intermediate number to a running total in a loop.

This method must have one **long** parameter (N) and return the calculated value as a **long**.

Constant-Time Formula

The second implementation uses a clever observation based on, of all things, triangles.



The area of the triangle in red is equal to the sum of all numbers between 1 and 10. The triangle itself has height 10 (N) and base 11 (N+1), and using the formula for calculating the area of a triangle (½)(base*height) we can see that the sum of all numbers 1 to 10 is (½)(10*11) = (½)(110) = 55.

This method must also have one **long** parameter (N) and return the calculated value as a **long**.

The interesting part of this programming assignment is the comparison of these two methods, which you will do in the Benchmarker class.

Comparing Methods

The purpose of the compare() method is to call each of the methods from ComparisonMethods using the same value of N and time how long it takes each one to complete.

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

Record the elapsed time for both the brute force and constant time formula implementations, and return it in a tab-separated String with a newline terminator, formatted as follows:

```
inputN + "\t" + bruteForceTime + "\t" + formulaTime + "\n"
```

For example, for an N of 100, my output from this method might be the string “100 1 0\n”.

Note: this method should verify that the results of each of the algorithms are equal, and should throw an NoSuchElementException with a descriptive error message if they are NOT.

Creating a File

This method will collect a series of these comparison results and write them to a file. Handle any exceptions which may be thrown at the time they are thrown; this method should NOT throw exceptions.

1. Use the java.io.File provided in the parameter to open a FileWriter or PrintWriter.
2. Iterate through the parameter array of long values in order and write the results of compare() to the file. Note that each String produced by compare() should already have a newline at the end, so do not add another.
3. Close the FileWriter or PrintWriter.

For an input array of five long values generated by starting at 100,000 and increasing by a factor of 10 for each element, my laptop produced the following results:

```
100000    1    0
1000000    1    0
10000000    4    0
100000000    41    0
1000000000    411    0
```

Results vary from computer to computer, and even from run to run! Don’t expect to get these exact values.

If you are unable to create the FileWriter or PrintWriter object, your method should print the message "`Exception encountered, unable to complete method.`" to the console and end the method.

If you encounter an error while writing to the FileWriter or PrintWriter object, your method should print the message "`Exception encountered while writing for value N = `" ending with the corresponding value of N to the console and continue running.

If you encounter an error while closing the FileWriter or PrintWriter object, your method should print the message "`Exception encountered while closing file.`" to the console and end the method.

If your compare() method throws a NoSuchElementException, your method should print the message associated with the exception and continue running.

Use your Benchmarker class' main method to create test arrays and run the createResultsFile() method. This class takes the place of a tester class for this programming assignment.

Commenting

In addition to our usual commenting requirements, add the following line to the Javadoc for each method in the ComparisonMethods.java file:

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

Include this line for both constantTime() and bruteForce(), filling in the blank with the appropriate big-O notation complexity for the algorithm in the method. This value will be manually graded.

~~We've already basically given away the answer for constantTime()~~.

Assignment Submission

Since you're working remotely, it's a good idea to keep a copy of your current work on Gradescope. Please feel free to make multiple submissions of your work to Gradescope as you progress through this specification!

Ensure that your final submission adheres to this specification and the academic conduct and style guide requirements as closely as you can!

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

- ComparisonMethods.java
- Benchmarker.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