### CS 1181

### Programming Assignment 4

**NOTICE: There are NO static variables allowed in this lab. The only class that contains a static method is the primary test class. Member variables are all private. Output statements are only in the primary test class.**

**PURPOSE:**

The purpose of this lab is to validate the Big-O that is expected for a variety of software structures.

**PROCEDURE:**

There are 8 code fragments at the end of this document. Each code fragment has a different software structure. The first part of your assignment is to examine the 8 code fragments and determine what you expect the Big-O behavior of the each fragment to be. **Use the following table to document your initial determination of the Big-O for each fragment.** This table will be completely filled in and turned in as part of this assignment.

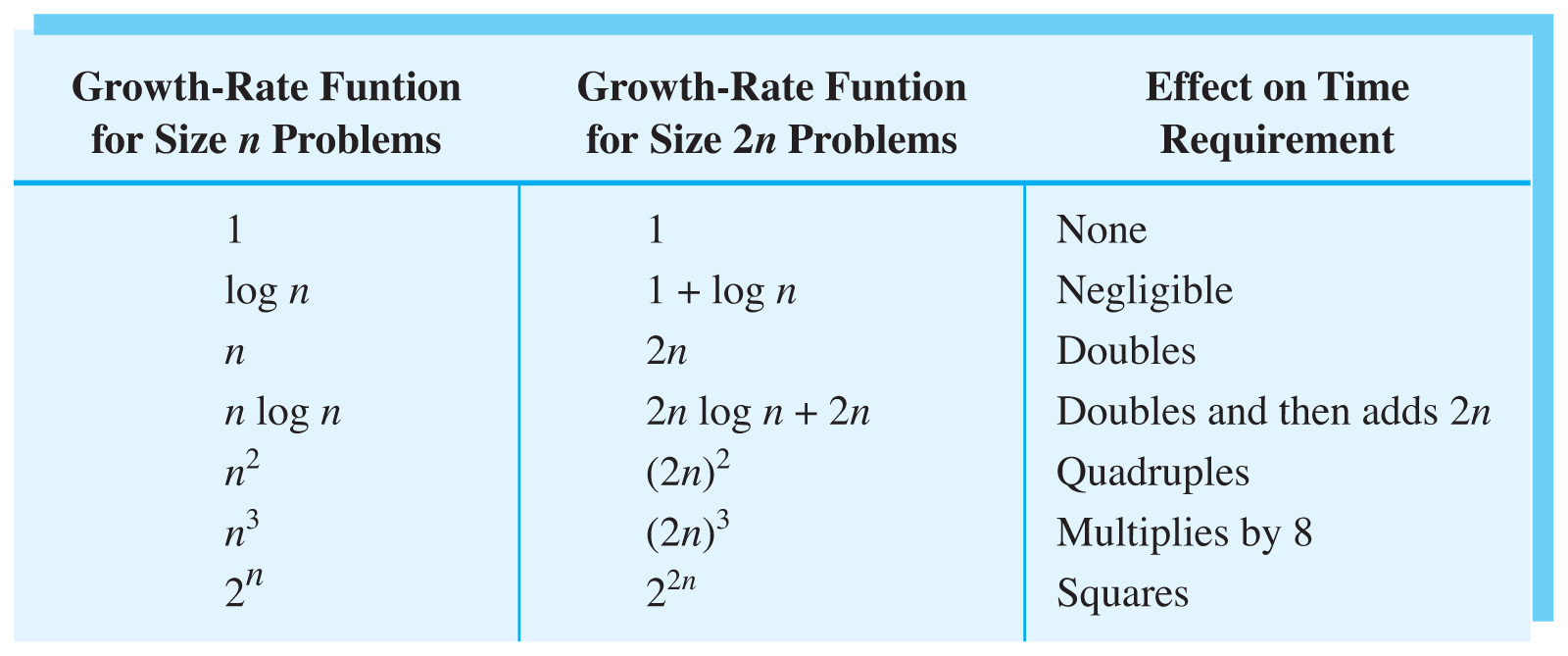
|  |  |  |  |
| --- | --- | --- | --- |
| Fragment # | Big-O | Expected Growth Factor | Observed Growth Factor |
| 1 | O(n) | 2n |  |
| 2 | O(n^2) | 2n^2 |  |
| 3 | O(2n) | 4n |  |
| 4 | O(n^3) | 2n^3 |  |
| 5 | O(n^2) | 2n^2 |  |
| 6 | O(n^5) | 2n^5 |  |
| 7 | O(log n) | 1 + log n |  |
| 8 | O(2^n) | 2^2n |  |

The overall approach of this assignment will be to collect average execution time data for each of the code fragments. Four different values of N will be selected as test points where execution time measurements are collected. For all but one of the fragments, the value of N will double between each test point. For instance, if the value of N for the first test of a fragment is 50, then the second test would use N = 100, the third would use N = 200, and the fourth test would use N = 400.

Each time a test of a fragment is performed at a value of N, the average execution time is measured. After running the test on 4 different values of N, there are 4 different measured time results. The definition of growth factor in the table above is the factor the execution time grows when the value of N is doubled. For example, if the second measured time is 200 and the first measured time is 100, then the growth factor is as follows: 1 + (200 – 100) / 100 or in this case 2. So by doubling the size of N, the execution time increases by a factor of 2. This is the expected behavior of software that is O(n).

When filling in the table above with the Big-O of each code fragment, **fill in the Expected Growth Factor column assuming the value of N doubles between test points (ie test N, 2N, 4N, 8N).** The following table may be helpful in understanding the expected growth. The first column is the Big-O, while the second column shows the same function when the value of N is doubled. The third column shows the actual growth to expect between test points N and 2N. For instance, for O(log n), the difference between column 1 and column 2 is very small, so the expected growth factor would be 1 + a very small amount. Whereas for the O(n3) case, the difference between column 1 and 2 is 23 which is a growth factor of 8!

The final column of the table above will be filled in with the actual measured results as described below.



The only exception to the doubling of N between test points should be in the case of an O(2n) fragment. This growth rate is so extreme that doubling the value of N is impractical for taking measurements. Instead, for testing a fragment that is O(2n) just add 2 to the value of N for each test case. This results in an expected increase in the growth rate between test points of 22 or a factor of 4 (quadrupling).

**Implementing the Test Software**

The Java programming environment presents an especially challenging environment when it comes to getting valid, consistent timing information. The JVM will do optimizations to your code behind your back that affect execution speed. The Java garbage collector will run whenever the JVM decides it needs to reclaim memory and that will significantly impact execution times. Windows will also interfere with trying to get timing data. Windows can interrupt the execution of a piece of code at any time to let some other task run thereby interfering with the time it takes for a block of code to execute.

So, in order to give ourselves the best chance to get reasonable timing data we need to take several steps to mitigate these external effects that are working against us. This involves taking periodic breaks from execution, explicitly running the Java garbage collector, and declaring some variables as volatile to keep the JVM from optimizing access to them. We also need to repeats tests and take an average to improve the chances of getting consistent timing numbers.

You are going to develop two classes. One class is responsible for repeatedly executing functions, collecting timing data, and calculating average execution times. The other class is responsible for deciding what function to test, what value of N to use, and calculating and reporting average growth rates between test points.

**FunctionTimer Class**

This class is responsible for repeatedly executing the code fragments, collecting timing data, and returning the average execution times. This class contains the following:

**A private method for each of the 8 code fragments** listed at the end of this document. Each method has a **void** return type and takes an *integer* parameter ***n*** that controls the repetition of the code fragment. Each code fragment / function contains the following block of code:

sum = 0;

for(x = 0; x < REP\_COUNT; x++)

sum += x;

There are 2 variables and a constant that control the execution of this code block. It is critical that you define these items as members of the class, not as local variables. The variables **sum** and **x** must be declared as ***volatile long*** member variables. This will force the JVM to write actual values out to real memory every time these variables are touched. This is critical to getting reasonably accurate and consistent timing data for these functions. **REP\_COUNT** should be declared as a ***final int*** member and should be set to 25 initially. This constant can be modified if needed to adjust the function execution times for faster or slower computing environments.

An **integer** **member variable** is needed to control the number of times the test of a function should be repeated. There also must be a **setter** for this variable to allow this number to be modified by the program coordinating the testing activities. The **setter** method must verify that its parameter is greater than zero. If not, the **setter** should throw an **IllegalArgumentException**.

The **final method** needed for this class is the method that repeatedly executes the requested function for the requested value of N and collects timing data for that function. This method takes 2 parameters, the number of the function to be executed and the size of N. The function number must be between 1 and 8 inclusive. The number corresponds to the fragment numbers at the end of this document. The size of N must be greater than 0. If any of these preconditions are violated, an IllegalArgumentException should be thrown.

This method requires three local variables, all of which are of the **long** data type. One variable is needed to hold the time a test **starts**. Another variable is needed to hold the time a test **stops**. A third variable is needed which will contain the running sum of how long each iteration of the test takes to execute. This method returns a long value which is the average execution time of the function being exercised, which is calculated by dividing the running sum by the number of times the test is repeated.

The structure of this method is critical to improving the chances of getting consistent timing data. The main work loop of this method controls the number of times the selected function executes. Inside this loop, the first thing to be done is to briefly sleep. This will reduce the chances that the JVM or Windows will interfere with the execution of the function. The following code sleeps for 5 milliseconds:

try

{

Thread.sleep(5);

}

catch(InterruptedException e){ }

The next thing to do to further reduce the likelihood of interference from the JVM is to call the garbage collector. The following line of code does this.

System.gc();

It is finally time to execute the requested function. Java provides a method to get the current time as a running counter value. The following statement is used to get the current time value to mark the start of the test execution.

start = System.nanoTime();

Now use a switch statement to select the function to call and pass the provided value of N into that function. When the function completes execution, get the current time again and store that into the stop variable. Add the difference between stop and start to the running sum variable.

When the loop has completed, calculate and return the average execution time as previously described.

**Main Test Program Class**

This class contains the main method which controls the process of testing each method at 4 different values of N, collects the average execution time values for those 4 different values of N, and then calculates and displays the average growth factor between the execution times.

The main method will need three local variables. There will need to be a FunctionTimer object that is used to get timing data on function executions. One variable contains the integer value of N that is passed into the functions being tested. Another variable is an array of 4 long values which holds the 4 average execution times returned from calling the FunctionTimer object 4 times for the same function with different values of N.

There should be a second static method in this program that calculates and outputs the average growth rate between execution times for different values of N. This method takes two parameters, the array of average execution time values, and the number of the function that was tested to produce these timing numbers. This method does the following calculation to determine the average growth rate (assume **d** is the array parameter):

1. + ((d[1] – d[0] ) / d[0] + (d[2] – d[1]) / d[1] + (d[3] – d[2]) / d[2] ) / 3.0

Care should be taken to cast the denominator of each division to a **double** in order to achieve the most accuracy. This method should output the number of the function and the observed growth rate with 3 digits after the decimal point.

The basic structure of the main method is as follows:

1. Set the FunctionTimer’s times to repeat value
2. Set the variable **n** with the largest value of N the function is tested with
3. A *for* loop executes 4 times, start at 3, then 2, then 1, then 0
   1. Call the FunctionTimer’s function execution / timing method with the number of the function to be executed and the **n** variable
   2. Store the result of the previous step into the array using the loop control variable as the index
   3. Cut the value of the **n** variable by a factor of 2
4. Call the function to calculate and output the average growth rate passing in the function number and the array.

The reason the testing starts with the largest value of N is because this will minimize the effect of anything special the JVM does when the function is executed for the first time.

NOTE: 7 of the 8 functions to be tested have a factor of 2 between the sizes of N for each test point. However, the 8th code fragment / function needs to be handled differently. For testing function 8, instead of dividing **n** by 2 in step c, do **n = n – 2** instead.

The following table indicates the times to repeat each function and the starting values of N that you should use.

|  |  |  |
| --- | --- | --- |
| Function Number | Times to Repeat | Starting Value of N |
| 1 | 500 | 40000 |
| 2 | 500 | 400 |
| 3 | 500 | 40000 |
| 4 | 250 | 64 |
| 5 | 500 | 400 |
| 6 | 50 | 16 |
| 7 | 500 | 800000 |
| 8 | 500 | 10 |

**How to Submit Lab 3**

You will need to submit a (Word) document. Write a brief introduction to describe what you are doing in this programming exercise. **This should be at least a significant paragraph.** Next, include the table from the first part of this document. Fill in the last column of the table with the growth factor results produced by your program. For each of the 8 program fragments, explain how the fragment’s software structure led you chose the Big-O function. Also, for each of the 8 program fragments, explain whether the observed growth factor confirms the growth factor you predicted based on your choice of Big-O. Explain any significant discrepancies between what you expected to see and what your measurements showed.

For this assignment you are submitting three things to the drop box. The Word document described above, ONE text document (Wordpad or Notepad) with your 2 classes copied and pasted into the text file, and a ZIP file containing your entire Netbeans project.

**Failure to properly submit these 3 files will result in a grade of zero! If your project does not compile, you will receive a zero. Use of any static variables will result in an automatic 25% point penalty. Use of static methods in any class other than your test program will result in an automatic 25% point penalty**

**Rubric ( 60 pts)**

**FunctionTimer Class (25 pts)**

* **Must be implemented as described above**

**Main Test Program Class (25 pts)**

* **Must be implemented as described above**
* **Produces valid timing results which closely agree with theoretical predictions**

**Word document containing table and writeup (10 pts)**

* **Introductory paragraph**
* **Table completely filled in with predictions and measured results**
* **Explanation of why Big-O function was chosen for each fragment**
* **Comparison of measured growth factor with expected growth factor**

**Code:**

**// fragment #1**

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

**{**

**sum = 0;**

**for(x = 0; x < REP\_COUNT; x++)**

**sum += x;**

**}  
 //end fragment #1**

**// fragment #2**

**for (int i = 1; i <= n; i++ ) {**

**for (int j = 1; j <= n; j++ ) {**

**sum = 0;**

**for(x = 0; x < REP\_COUNT; x++)**

**sum += x;**

**}**

**}**

**//end fragment #2**

**// fragment #3**

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

**{**

**sum = 0;**

**for(x = 0; x < REP\_COUNT; x++)**

**sum += x;**

**}**

**for (int j = 1; j <= n; j++ )**

**{**

**sum = 0;**

**for(x = 0; x < REP\_COUNT; x++)**

**sum += x;**

**}**

**//end fragment #3**

**// fragment #4**

**int nsquared = n \* n;**

**for (int i = 1; i <= n; i++ ) {**

**for (int j = 1; j <= nsquared; j++ ) {**

**sum = 0;**

**for(x = 0; x < REP\_COUNT; x++)**

**sum += x;**

**}**

**}**

**//end fragment #4**

**// fragment #5**

**for (int i = 1; i <= n; i++ ) {**

**for (int j = 1; j <= i; j++ ) {**

**sum = 0;**

**for(x = 0; x < REP\_COUNT; x++)**

**sum += x;**

**}**

**}**

**//end fragment #5**

**// fragment #6**

**int nsquared = n \* n;**

**for (int i = 1; i <= n; i++ ) {**

**for (int j = 1; j <= nsquared; j++ ) {**

**for (int k = 1; k <= nsquared; k++ ) {**

**sum = 0;**

**for(x = 0; x < REP\_COUNT; x++)**

**sum += x;**

**}**

**}**

**}**

**//end fragment #6**

**// fragment #7**

**while ( n >= 1 )**

**{**

**sum = 0;**

**for(x = 0; x < 100 \* REP\_COUNT; x++)**

**sum += x;**

**n /= 2;**

**}**

**//end fragment #7**

**// fragment #8**

**int end = (int)Math.pow(2,n);**

**for (int i = 1; i <= end; i++ )**

**{**

**sum = 0;**

**for(x = 0; x < 100 \* REP\_COUNT; x++)**

**sum += x;**

**}**

**//end fragment #8**