Schedule
Calendar
Syllabus
Questions
Material
Assignments
Help Hours

PA0
PA1
PA2
PA3
Exam 1
PA4
PA5
PA6
Exam 2
PA7
PA8
Exam Final - Part 1
Exam Final - Part 2
Exam Final - Part 3

CSE 12 Programming Assignment 4

Runtime, Measured and Modeled

This assignment is open to collaboration.

This assignment will give you experience working with big-O/θ/Ω representations, practice matching them to implementations, and perform measurements of the runtime of different methods.

This assignment is inspired by a combination of a lab in Swarthmore College's CS35, and by a similar assignment by Marina Langlois in CSE12 at UCSD

This PA is due on ** **Tuesday, April 26 at 10:00pm** **

Getting the Code

The starter code is at here. If you are not familiar with Github, here are two easy ways to get your code.

1. Download as a ZIP folder

   If you scroll to the top of Github repository, you should see a green button that says *Code*. Click on that button. Then click on *Download ZIP*. This should download all the files as a ZIP folder. You can then unzip/extract the zip bundle and move it to wherever you would like to work. The code that you will be changing is in the folder called *pa4-starter*.

2. Using git clone (requires terminal/command line)

   If you scroll to the top of the Github repository, you should see a green button that says *Code*. Click on that button. You should see something that says *Clone with HTTPS*. Copy the link that is in that section. In terminal/command line, navigate to whatever folder/directory you would like to work. Type the command `git clone _` where the `_` is replaced with the link you copied. This should clone the repository on your computer and you can then edit the files on whatever IDE you see fit.

If you are unsure or have questions about how to get the starter code, feel free to make a Piazza post or ask a tutor for help.

Part 1: Runtime Analysis Questions

Big-O Justification

Indicate whether the following assertions are true or false, and give a justification:

- $n + 5n^2 + 8n^4$ is $O(n^3)$
- $n! + n$ is $O(n * log\ n)$

- $log\ n + n * log\ n + log(log\ n)$ is $\Omega(n)$
- $n^2 + n/4 + 6$ is $\Theta(n^2)$
- $1/(n^{50}) + log32$ is $\Omega(1)$
- $1/(n^{50}) + log2$ is $O(1)$

If you are justifying the positive direction, give choices of `n0` and `C`. For big-Θ, make sure to justify both big-O and big-Ω, or big-O in both directions.

If you are justifying the negative direction, indicate which term(s) can't work because one is guaranteed to grow faster or slower than the other.

As a quick guide, here is an ordering of functions from slowest-growing (indeed, the first two *shrink* as n increases) to fastest-growing that you might find helpful:

- f(n) = $1/(n^2)$
- f(n) = $1/n$
- f(n) = 1
- f(n) = log(n)
- f(n) = sqrt(n)
- f(n) = n
- f(n) = n^2
- f(n) = n^3
- f(n) = n^4
- ... and so on for constant polynomials ...
- f(n) = 2^n
- f(n) = n!
- f(n) = n^n

This portion will be submitted via Gradescope. It can be found in the *Programming Assignment 4 - questions* assignment (this is question 1).

List Analysis

Consider the two files `ArrayStringList.java` and `LinkedStringList.java`, which are included in this repository. Answer the following questions, and justify them with one or two sentences each:

- Give a tight big-O bound for the *best case* running time of `prepend` in ArrayStringList
- Give a tight big-O bound for the *worst case* running time of `prepend` in ArrayStringList
- Give a tight big-O bound for the *best case* running time of `prepend` in LinkedStringList
- Give a tight big-O bound for the *worst case* running time of `prepend` in LinkedStringList
- Give a tight big-O bound for the *best case* running time of `add` in ArrayStringList
- Give a tight big-O bound for the *worst case* running time of `add` in ArrayStringList
- Give a tight big-O bound for the *best case* running time of `add` in LinkedStringList
- Give a tight big-O bound for the *worst case* running time of `add` in LinkedStringList

In all cases, give answers in terms of the *current size of the list*, and assume that the list has some non-empty size *n*. That is, you shouldn't consider the empty list as a best case; instead think about the best case based on other factors like size, capacity, and nodes.

Notable points to consider:

- Creating an array takes time proportional to the length of the array
- When considering the running time of a method, make sure to take into account any helpers methods it uses!

Example for `get` in the `LinkedStringList` class:

```
The get method is O(1) in the best case, when the index is 0. In this case
it will do constant work checking the index and immediately return the
first element, never entering the while loop.

The get method is O(n) in the worst case, because the index could be at
the end of the list (for example, index n - 1). In this case, the while
loop will run n times, spending constant time on each iteration, resulting
in overall O(n) number of steps taken.
```

This portion will be submitted via Gradescope. It can be found in the *Programming Assignment 4 - questions* assignment (this is question 2). Make sure to following the formatting instuctions!

Mystery Functions

We have provided you with a `.jar` file that contains implementations of the following methods:

```
public static void f1(int n) {
        int a = 0;
        int i = 0;
        while (i < n) {
                a = i;
                i++;
        }
}
public static void f2(int n) {
        int a = 0;
        for(int i = 0; i < n; i += 3) {
                a = i;
        }
}
public static void f3(int n) {
        int a = 0;
        int i = 0;
        while (i < n) {
                for (int j = n; n > 0; n = n/10) {
                        a = i + j;
                }
                i++;
        }
}
public static void f4(int n) {
        int a = 0;
        for(int i = 0; i < n; i += 1) {
                for(int j = i; j < n; j += 1) {
```

```
                        a = i + j;
                }
        }
}
public static void f5(int n) {
        int a = 0;
        for(int i = 0; i < n * n; i += 1) {
                for(int j = 0; j <= n; j += 1) {
                        a = i + j;
                }
        }
}
public static void f6(int n) {
        int k = 1, a = 0;
        for(int i = 0; i < n; i += 1) {
                for(int j = 0; j <= k * 2; j += 1) {
                        a = i + j;
                }
                k = k * 2;
        }
}
```

However, in that file, they are called `mysteryA-F`, and they are in a different order, and we don't provide the source of that file. You have two tasks: determining a big-O bound for each method labeled 1-6 analyzing the source above, and determining which mystery method A-F corresponds to the implementations above by measuring against provided (but hidden) implementation.

Identifying Bounds from Code

Determine a big-O bound for each function, and justify it with a few sentences. Give only the most relevant term, so use, for example *O(n)*, not *O(4n + 2)*. You will be submitting this via Gradescope. It can be found in the *Programming Assignment 4 - questions* assignment (this is also part of question 3). Make sure to following the formatting instuctions!

Part 2: Measuring Implementations

You will write a program to:

- Measure the mystery methods
- Use your measurements to match the mystery methods to the sources above
- Generate several graphs to justify your work

You have a lot of freedom in how you do this; the deliverables you need to produce are specified at the end of this section. There are a few methods that we *require* that you write in order to do this, and they will help guide you through the measurement process.

The `measure` Method

You *must* write the following two methods in the `Measure` class:

```
public static List<Measurement> measure(String[] toRun, int startN, int stopN)`
public static String measurementsToCSV(List<Measurement> measurements)
```

where `Measurement` is defined in `Measurement.java`.

- `measure` should work as follows:

    1. It assumes each string in `toRun` is one of the letters A-F.
    2. For each of the implementations to run, it runs the corresponding `mysteryX` method `stopN - startN` times, providing a value of `n` starting at `startN` and ending at `stopN` each time.
    3. For each of these runs, it *measures* the time it takes to run. You can do this by using the method `System.nanoTime()` (see here for its official Java documentation)
    4. For each of the measured runs, it creates a `Measurement` whose `valueOfN` field is the value that was used for the given run, whose `name` field is the single-letter string of the implementation that ran, and whose `nanosecondsToRun` field is a measurement, and adds it to a running list of measurements.
    5. The final result is the list of measurements.

Example:

This call:

```
measure(new String[]{"A", "B"}, 40, 100);
```

Should produce a list that has 122 measurements, 61 of which will have `name` equal to `"A"` and 61 of which will have `name` equal to `"B"`. Each of the 61 for each name will have a different `valueOfN` from 40 to 100, and each will have a different number of nanoseconds (as was measured).

The `measurementsToCSV` method

The `measurementsToCSV` method takes a list of measurements (for example, as returned from `measure`) and generates a comma-separated-values `String` of the measurements. It should have the following format, where the first row is a literal header row and the other rows are example data. Note that this data is completely made up, and may not match your measurements.

You might choose to put all of the measurements for a single letter together:

```
name,n,nanoseconds
A,40,1034
A,41,1039
A,42,2033
... many rows for A ...
A,100,432
B,40,1034
B,41,4038
... many rows for B ...
```

You might also choose to put all of the measurements for a single round of `n` together:

```
name,n,nanoseconds
A,40,1034
B,40,1034
```

```
A,41,1039
B,41,4038
A,42,2033
B,42,4038
... many alternating rows of A, B ...
A,100,432
B,100,8038
```

Either layout is fine, do what makes sense to you, or what matches your `measure` function best, etc.

Strategies for Measuring

You can use the `measure` and `measurementsToCSV` methods to produce data about how the functions behaved in terms of their runtime. You should fill in the `main` method with whatever you find useful for using your measuring methods to compare the mystery implementations. You have total choice in how you implement this – you can add new helpers, print the CSV format out to a file, copy/paste it into a spreadsheet, use a tool you like for plotting, etc. The goal is to use measurements to identify the different implementations. Feel free to look up documentation for writing Strings out to files and use it, or use `System.out.println` and copy/paste the output, etc. It's probably pretty expedient to copy the data into Excel or a Google Sheet.

There are a few high-level strategies to consider:

- If an implementation is very slow, it could take a really long time to measure it for large n. If you notice something is taking a long time, stop the program and try the same mystery methods on a smaller input range. Does the smaller range tell you anything useful?
- Some of the methods might have similar big-O bounds, but have different constants that can be measured in terms of absolute time.
- Some of the methods might take vastly different times to run on certain inputs, so plotting them next to one another will show one with a flat line at 0 and the other with some interesting curve. Make sure to check what the relative numbers are when inspecting the output.

You will use these measurements to figure out which mystery method matches the implementations above, and generate three graphs to justify your answers.

Avoiding Obscuring Optimizations

On many platforms and Java versions, simple methods like the above get *optimized* to run much faster than their theoretical number of steps might suggest. Java is pretty smart – it can, while running, figure out how to make them run quickly enough that empirical measurements become hard to make. If you're seeing that even on values of n in the hundreds of thousands, you get effectively constant behavior, you should try *disabling* these optimizations to get more useful measurements for distinguishing the implementations.

Instructions for doing this are in the *Turning Off Java Optimizations* section of this Google Doc (scroll to the end):

https://docs.google.com/document/d/1vwckO76TrBT8B5E4xQ2-v2OXncLa6SQWuaQkNZaCPB0/edit

Note that this will make all the mystery methods run *a lot* slower, so you may want to *decrease* the values of n you use after making this change to avoid waiting a long time.

Submitting

Part 1

You may submit as many times as you like till the deadline.

- The `Programming Assignment 4 - questions` assignment in Gradescope, where you will submit the written part of this PA.
  - The first question your big-O justifications.
  - The second question is for your List analysis.
  - The third question is for your matchings for the mystery functions, along with your graphs and justifications. The following are what need to be answered in the subquestions.
    - The BigO bounds for each implementation f1-6.
    - A listing that matches each of mysteryA-F to an implementation f1-6 above
    - Three graphs that justify a few choices above. These don't need to exhaustively describe all of your matchings, but they must be generated from real data that you measured using `measure`, and they must show an interesting relationship that helps justify the matching.
  - The last section gives you a space to indicate who you collaborated with (if you collaborated with anyone).

If you want a guide on how to get from the CSV data to a graph, look here:

https://docs.google.com/document/d/1vwckO76TrBT8B5E4xQ2-v2OXncLa6SQWuaQkNZaCPB0/edit

Part 2

The `Programming Assignment 4 - code` assignment in Gradescope is where you will submit your final code for performing measurements. Please submit the following file structure:

- *pa4-student*
  - *src*
    - *cse12pa4student*
      - Main.java
      - Measure.java
      - Measurement.java
    - mysteries.jar

The easiest way to submit your files is to zip the `pa4-student` folder and upload that to Gradescope. **Make sure you are including the folder `pa4-student` in your zip file!!!** You

may also use the bash script provided, `prepare-submission.sh`. You may encounter errors if you submit extra files or directories. You may submit as many times as you like till the deadline.

Scoring (70 points total)

(70 total points)

- 16 points `measure` and `measurementsToCSV` [autograded]
- 12 points initial big-O justifications [manually graded]
- 16 points list method analysis [manually graded]
- 26 points matching activity [manually graded]
  - 12 points for complexity bounds on f1-6
  - 6 points for a correct matching
  - 6 points for 3 relevant graphs
  - 2 points describing how you measured