

Bryan Arnold

CSE 2100

Programming Assignment 3

10/2/16

Description

The follow program is two simple classes. Each class is responsible for the java application for the given algorithms of the assignment. The programs work by taking the algorithms, implementing them as recursive methods, and displaying the runtimes and total runtime of given numbers. The first method in each class is the given algorithm to recursive application, while the second is my improved version of each. The improved versions have different operations to save runtime. The algorithms rules are described in full detail in the comments of each method.

Tradeoffs

The tradeoffs I considered while making this program were how to first implement the programs, but mostly on how to improve them. I wanted to keep the core design of the original algorithms mostly intact, while making the runtimes quicker. I was able to achieve this by only adding one simple primitive operation to one condition of both algorithms, but it was not as efficient as it could be.

Extensions

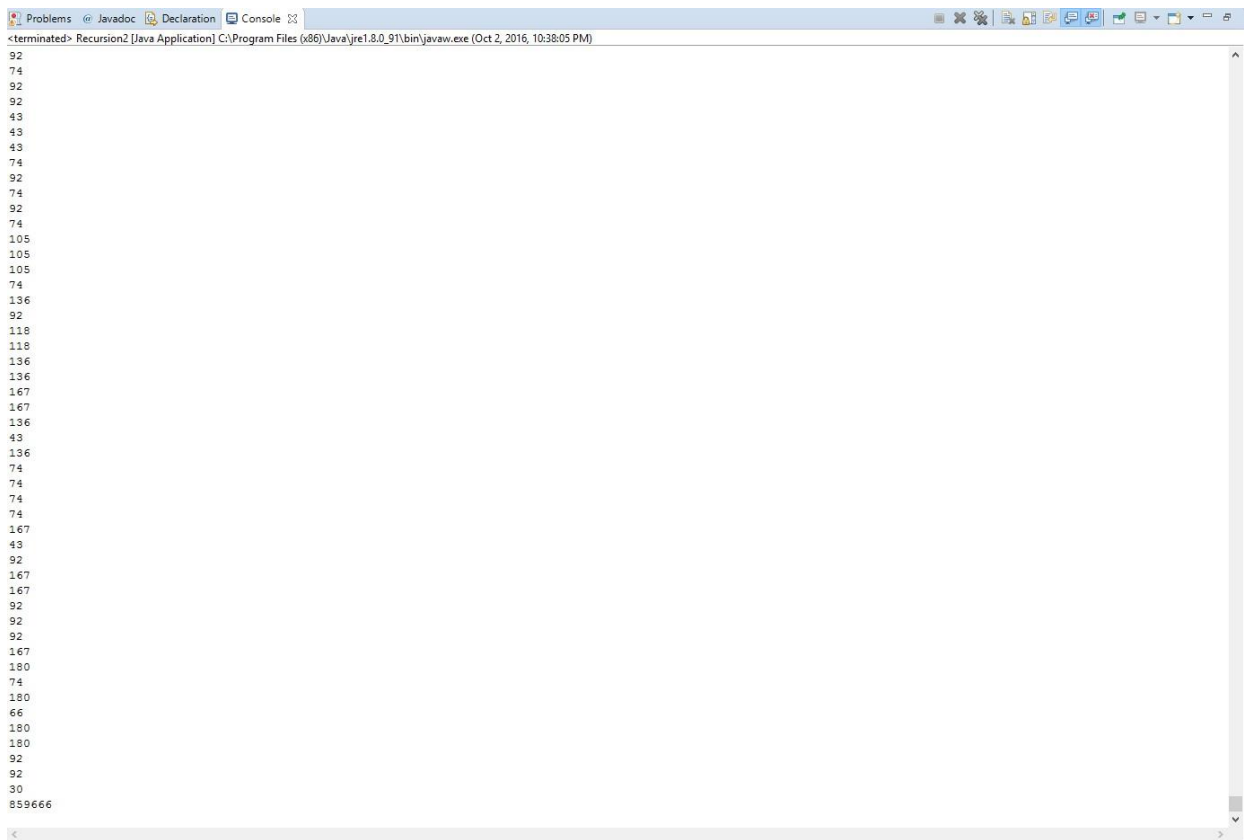
An improvement of these algorithms is entirely possible, such as adding more primitive methods to when the number is even, like dividing by 4. This could potentially lower the runtime even further of the algorithm. Another possible improvement, would be to convert the numbers to doubles, so the number could always be divided by 2 and converted back to an int. This could also cut down on runtime, but gain the same result.

Test Cases

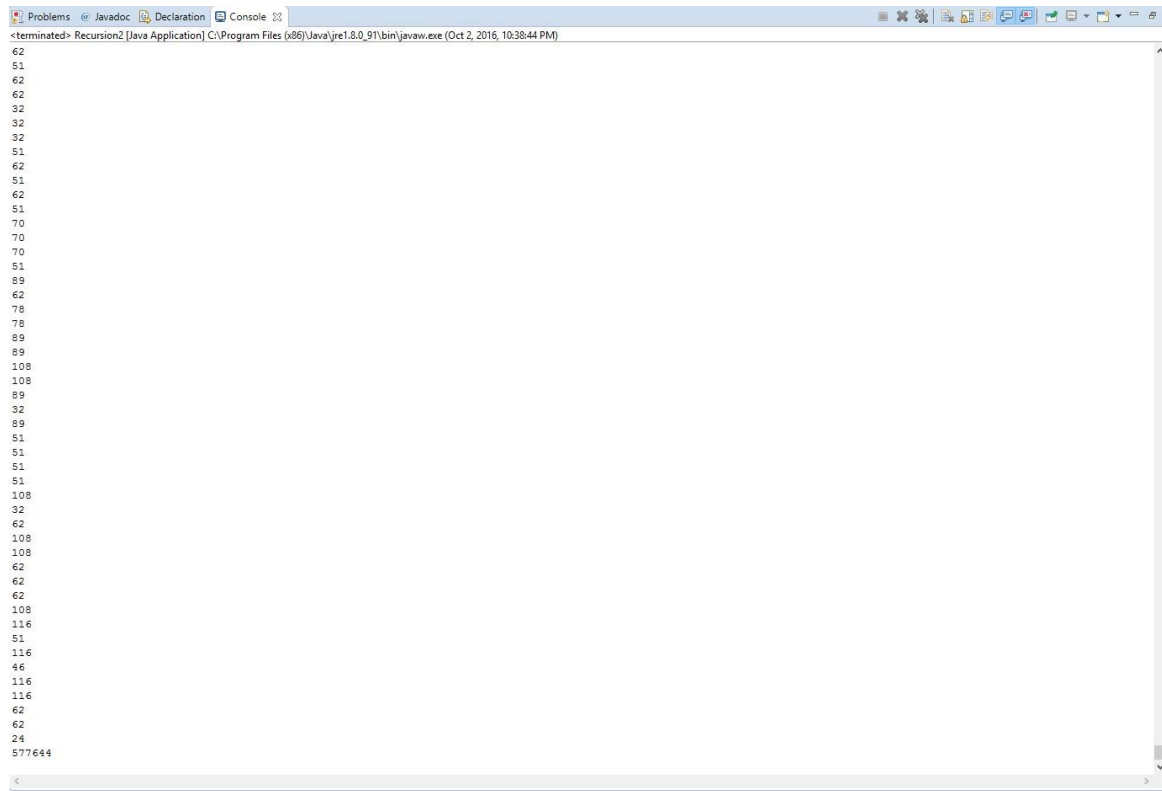
The wasn't a need to develop my own test cases other than printing the total runtime for all the algorithms. The test cases were given to us as part of the assignment. They included running the algorithms for all the numbers between 1 and 10000. To find the runtimes and total runtime of every algorithm, I did this in the main method. I totaled the total runtime of 1-10000 in one variable and printed each runtime of each number, not of all algorithms simultaneously though. This total runtime would be my comparison if my improvements of all algorithms were improvements, as the runtime should be lower. The following screenshots are how I did this:

The first screenshot is the runtime for the last 49 numbers leading up to 10000 of the first algorithm before it was improved, to show individual test cases. The final number at the bottom is the total runtime of the algorithm for the numbers 1 to 10000.

The second screenshot is also the last 49 numbers leading up to 10000 of the improved version of the second algorithm, to show individual test cases. The final number at the bottom is the total runtime of the algorithm for the numbers 1 to 10000. As you can see, the second total runtime is smaller, indicated my algorithm is an improvement for overall runtime. The following two screenshots are for the second part of the assignment:



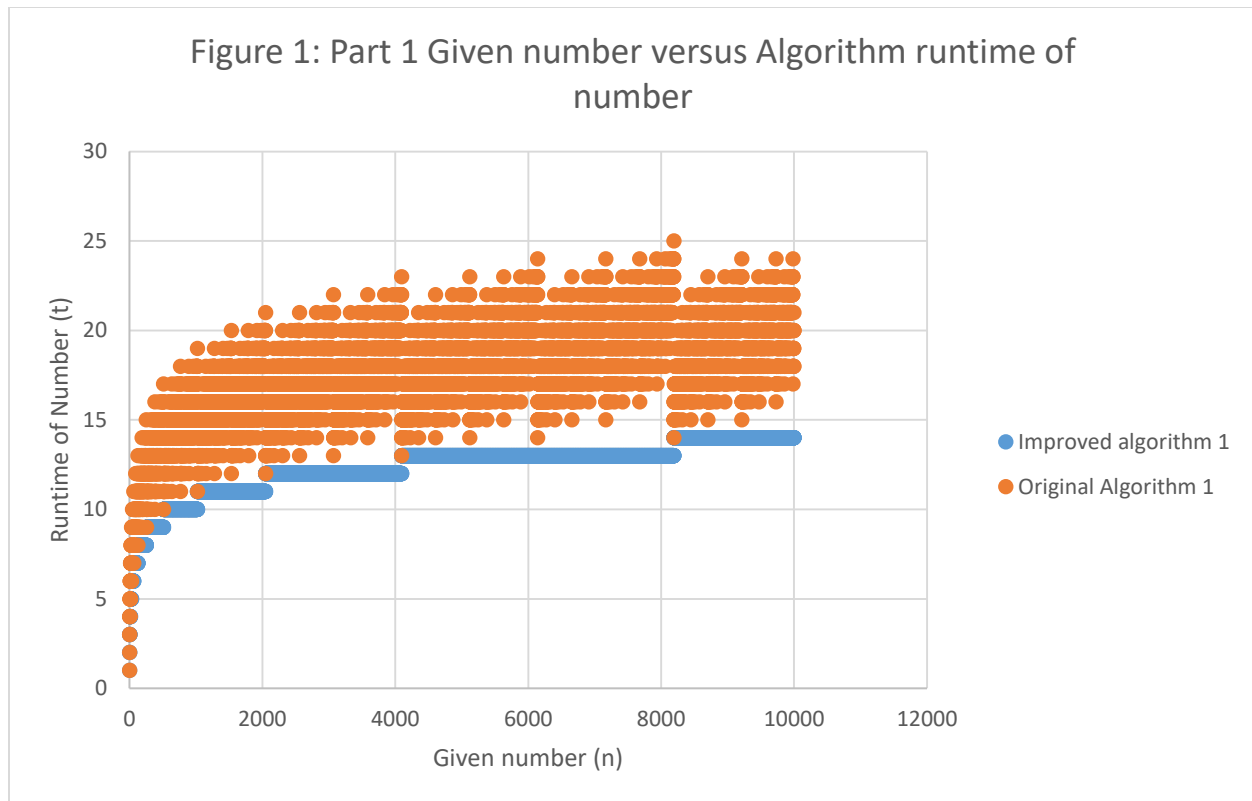
```
92
74
92
92
43
43
43
74
92
74
92
74
105
105
105
74
136
92
118
118
136
136
167
167
136
43
136
74
74
74
167
43
92
167
167
92
92
92
167
180
74
180
66
180
180
92
92
80
859666
```



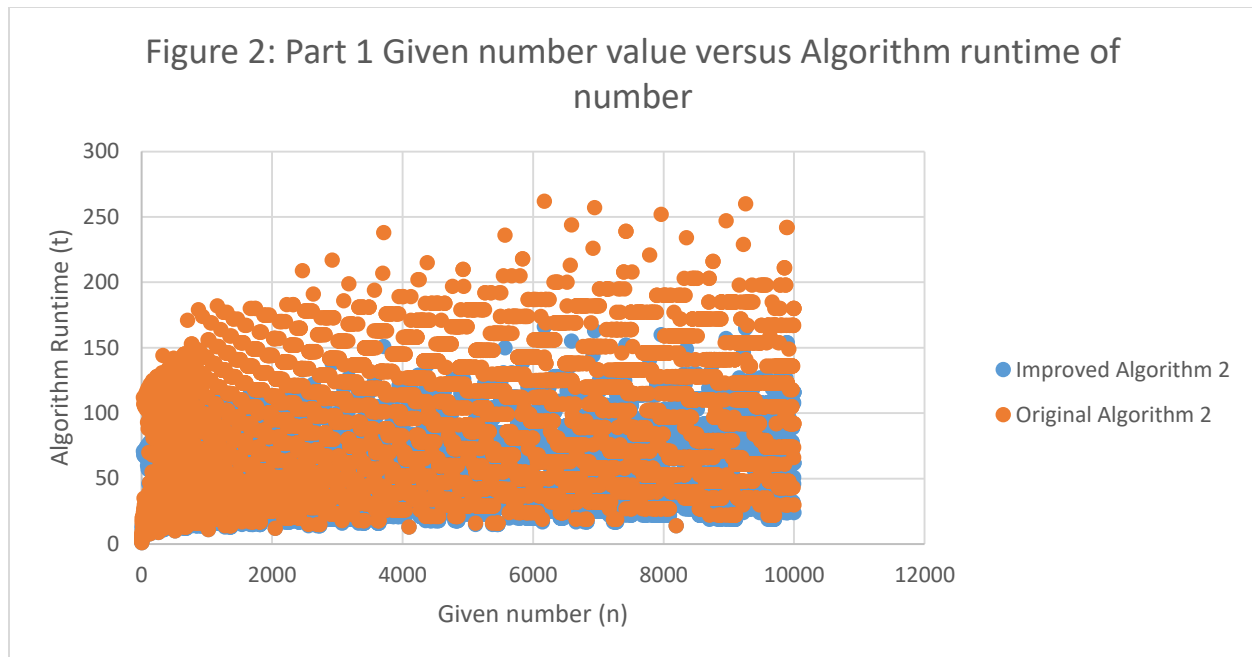
The screenshot shows a Java IDE window with a console tab. The console displays a list of numbers: 62, 51, 62, 62, 32, 32, 51, 62, 51, 62, 51, 70, 70, 51, 89, 62, 78, 78, 89, 89, 108, 108, 89, 32, 89, 51, 51, 51, 108, 32, 62, 108, 108, 62, 62, 108, 116, 51, 116, 46, 116, 116, 62, 62, 24, and 577644. The window title is "Recursion2 [Java Application] C:\Program Files (x86)\Java\jre1.8.0_91\bin\javaw.exe (Oct 2, 2016, 10:38:44 PM)".

These screenshots are the same format as the previous two, but with the second part of the assignment's algorithm and its improvement. As you can also see, my improved version, second screenshot, is less than the total runtime of the original algorithm, indicating an improved total runtime of the algorithm.

To display these runtimes better, since all values cannot be seen or a trend cannot be seen just by looking at them, plots were made for both. The following plot is the plot of the given number versus its runtime:



This plot will help us analyze and verify the runtime of the original algorithm of part one and its improved version. First, the condition of the algorithm for its base case is just $O(1)$, or 1. So this runtime is not really looked at. Next, when the number is odd it is simply subtracted by one, then put back into the method. This type of run method makes it have a runtime of $O(n)$. And finally, the runtime of the even condition is dividing by 2, so its runtime is $O(\log n)$. Looking at all these runtimes and the plot of the original algorithm, we can see that the true runtime of the algorithm is logarithmic. I wasn't able to find the exact/true big-oh/runtime equation of the first algorithm, but it is for sure logarithmic. Next, we look at the trend in the improved algorithm in the graph. The runtime for even numbers and 1 are the same, $O(1)$ and $O(\log n)$, but not looking at the new odd condition, it divides by two after it subtracts 1. This overrides the $O(n)$ runtime, and makes it $O(\log n)$. I also could not find the exact big-oh or runtime equation of this equation, but it is definitely logarithmic by this logic and the look of the graph. So, overall, the runtime/big-ohs of both the first two algorithms, are logarithmic with my improved version being more efficient and faster. Now, for the second plot for part two of the assignment:



This plot will help us analyze the runtime of the original algorithm of part two of the assignment in its original algorithm as well as my improved version. First, both algorithms have a base case of $O(1)$, or 1. This runtime is not really considered but will always add 1 to the total runtime for every number. Next, we look at the original algorithm, or the orange points of the graph. First, the even condition is the same as the previous part, $O(\log n)$, but the new odd condition vastly changes its runtime. This runtime is part of the Collatz Conjecture, which is by multiplying by 3 and adding one, the number will always reach 1 by following the above conditions. This makes the runtime much longer, but by looking at the given plot, a slight logarithmic trend can be seen due to the division of 2. So, the runtime/big-oh would be close to the actual runtime/big-oh of this algorithm (I was not able to properly analyze this algorithm as Stirling's Approximation confused me more than I already was). Next, we look at my improved version, or the blue points. It also follows the previous two runtimes/big-ohs for even and 1, $O(1)$ and $O(\log n)$, but now the odd condition is multiply by three subtract 1 then divide by 2. This cuts down on the runtime dramatically by making the big-oh/runtime of the improved version logarithmic whilst the previous odd condition was not. $O(\log n)$ is considered much more efficient than most runtimes, so having $O(\log n)$ for two conditions, my improvement runs faster. So, in the end and by looking at the plots and logic, we can see that the runtimes are somewhat logarithmic in nature, but I could not figure out their proper big-oh/runtime due to mass confusion of Collatz Conjecture (just being honest here).