Recursion and 3n + 1 Problem

Mark Cabanero

October 2, 2016

Program Design

Description

This program tests and plots the stopping time of two algorithms, defined:

$$g(n) = \begin{cases} 1 + g(n/2) & \text{if } n \text{ is even} \\ 1 + g(n-1) & \text{if } n \text{ is odd} \\ 1 & \text{if } n = 1 \end{cases}$$

$$g(n) = \begin{cases} 1 + g(n/2) & \text{if } n \text{ is even} \\ 1 + g(3n+1) & \text{if } n \text{ is odd} \\ 1 & \text{if } n = 1 \end{cases}$$

Hence referred to as problem 1 and problem 2, respectively.

By using a separate class named Recursion, cases are defined to call the function again with a smaller integer in order to reach the base case of $n = 1$.

```java
public static int problem1a(int n) {
    if (n == 1) {
        return 1;
    } else if (n % 2 == 0) {
        return 1 + problem1a(n / 2);
    } else {
        return 1 + problem1a(n - 1);
    }
}
```

Listing 1: Problem 1 in Java

Trade Offs

Since this assignment was straightforward for implementing recursion, there were no obvious trade offs to be made for the first part.

In optimizing the problems came the tricky part, with the stipulation to "keep the original problem intact." This means that tasks like rounding or dividing by larger integers are out of the question: they do not optimize the problem, but either create an approximation or bypass what the original problem was.

Hence a method that would create a hash map that kept track of the amount of steps it took from that particular $g(n)$ to 1 was out of the question; it modified what it meant to perform the recursion. This lead to the optimization in both problems – if $g(n)$ was ever odd, then divide the number first by 2, then call the recursion again. This lead to a reduction in calls overall, but could have been optimized even further.

Extensions

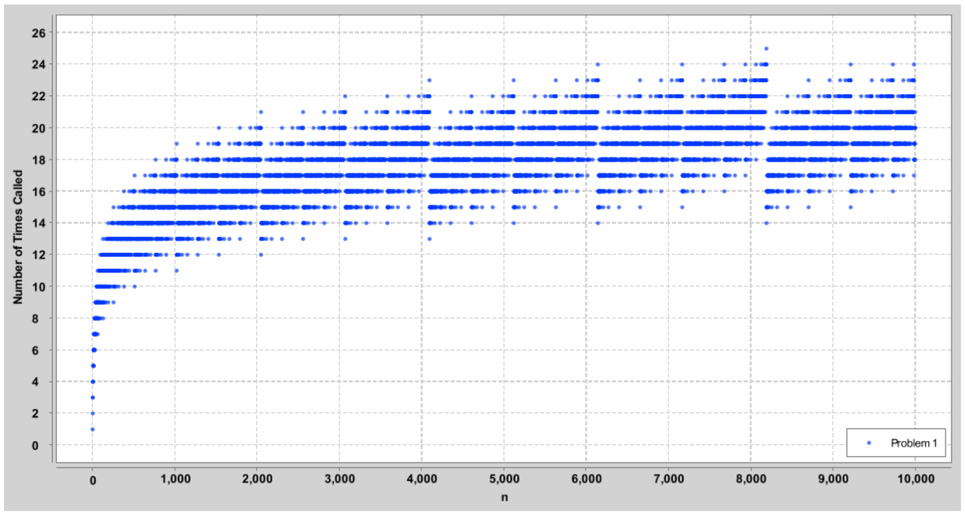
As discussed in **Trade Offs**, improvements could have come from:

- dividing n by larger even integers if they were known to be divisible by those integers
- keeping track of the steps it took to get to 1 from various $g(n)$s

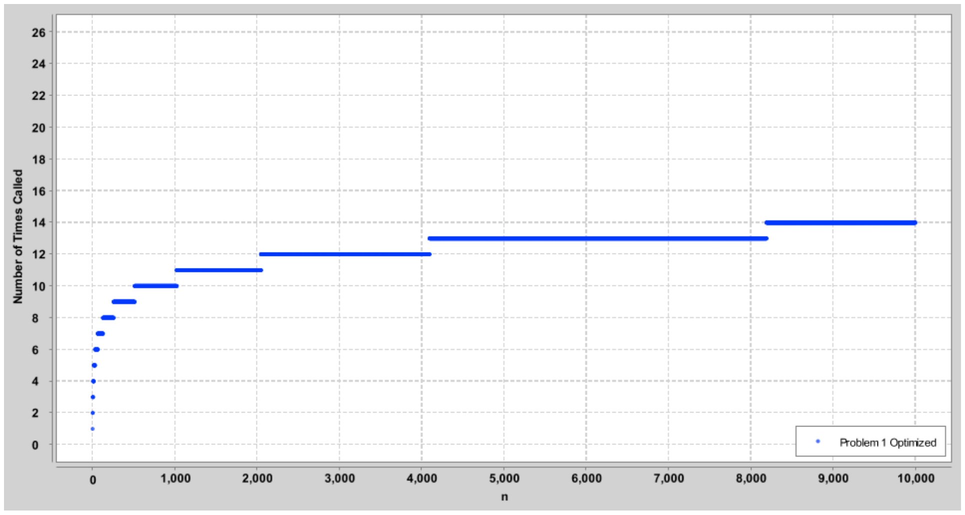
if they were in the scope of the problem.

Test Cases

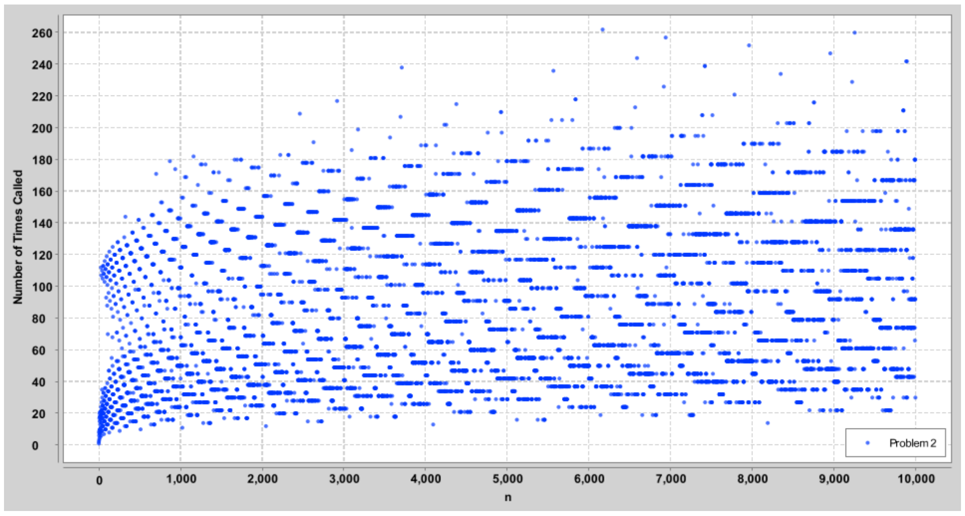
Since there are differences in stopping time between the original and optimized problems, there is no straightforward comparison between the two besides $g(1)$. Because of this, the graphs are observed to see if general trends are the same: problem 1 and optimized problem 1 are roughly logarithmic, while problem 2 and optimized problem 2 are abnormal in distribution, but it is observed that the calls made are roughly half. In addition, the original problems are compared to calculated stopping time distributions, and are observed to be the same.



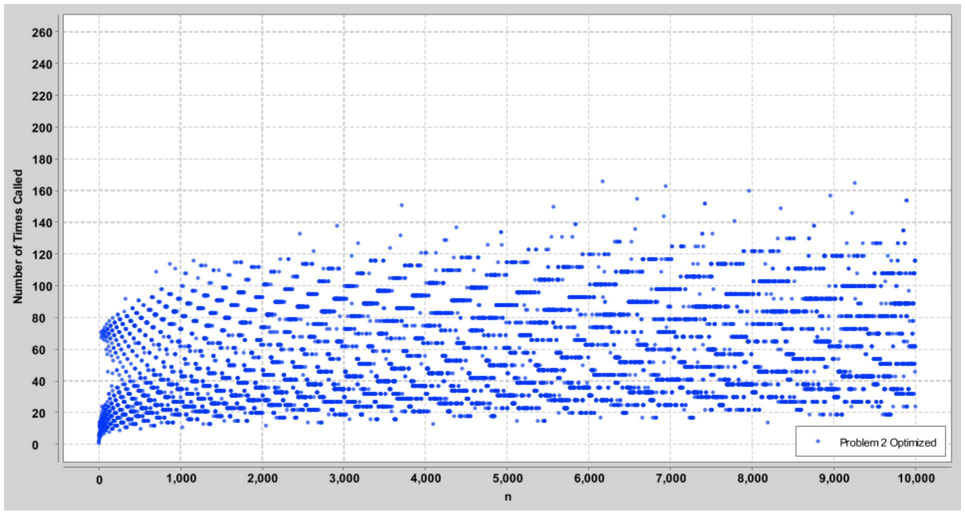
(a) Problem 1



(b) Problem 1 Optimized



(a) Problem 2



(b) Problem 2 Optimized

Questions

1. - For this algorithm, $g(n)$ would run close to O(log n) time. This is because the amount of numbers between 1 and 10,000 are half even and half odd. Half of the time, you'll be dividing n in half, which would result in a O(log n) runtime. However, the other half of the time, there would only be a subtraction of 1. This increases the estimation of O(log n), because subtracting by 1 adds to the stopping time.
   - For the improved algorithm, the graph now looks like a proper O(log n) graph. Since the calls to odd numbers are always made even, there is no need for an intermediary step of subtracting just 1, which cuts down on the amount of calls that needs to be done. This reduces down to O(log n).

   The graph also confirms this, where the random-esque looking O(log n) looks now more like a O(log n) graph.

2. This algorithm is a lot harder to analyze. Considering that there is a step where the number can triple, there are some certain ns that throw any sort of analysis off immensely: $n = 6, 19, 27, 97, 871$, etc. Even looking at the graph, there is no seeming pattern to discern, considering the wave-like picture that's constructed due to divisibility and multiplication that occurs.