

# Theoretical analysis: Experimental evidence of the Collatz conjecture

Student: Guillaume Leclerc (224338)

## Specification of the program

In this project we want to determine the number of steps it takes to go from a number to 1 using the Syracuse conjecture (see project description).

The output of the program will consist in a list of integer, the number of steps to go from the starting point to 1. The program will compute this value for each number starting at 2 and up to T the first and only argument given to the program.

Here is an example of a call to the program:

```
$ ./a.out 10
```

```
1
7
2
5
8
16
3
19
6
```

## Preliminary

Before we can do a proper analysis we need to do some empirical tests. Indeed for each number we don't know in advance how many steps it will take before we reach 1. So we can not really determine the algorithmic complexity of the algorithm in advance. For this analysis we will use the naive sequential approach to determine the number of steps. For example to reach 1 starting from 3 it take 7 steps to reach 1 :

```
3 -> 10 -> 5 -> 16 -> 8 -> 4 -> 2 -> 1
```

Using the very naive approach we computed the average number of steps from  $10^1$  to  $10^8$ . Let's note  $f(x)$  the average number of steps per integer to compute from 1 to  $10^x$ . For example  $f(1) = 7.44$  because as you can see in the example we gave before, the average of the values is 7.44. Here are the values we found for  $f(1)$  to  $f(8)$  (It takes too much time on a single machine to compute more).

```

7.444444444444444
31.73737373737374
59.601601601602
84.975097509751
107.53947539475
131.43455543456
155.27249862725
179.23493762235

```

As you might remark, the values are almost perfectly linear. It was not expected but it will be very convinient for our analysis. As we are in exponential scale we can deduce that the number of steps for each starting point is logarithmic. To be more precise here is the approximate formula :

$$f(x) \simeq 7.44 \times 24.6(x - 1)$$

We now have the number of steps of the naive algorithm :

$$T \times f(\log_{10}(T)) = \mathcal{O}(T \log(T))$$

## The Euristic

A simple (yet powerfull) euristic for this problem is to apply a time-memory tradeof. We can save for each number we already computed the number of iterations before reaching 1.

For example we compute the number of steps to go from 5 to 1 it take 5 steps, we memorize it. When we want to comput starting at 10. We apply the recurrence once and we find 5 (because  $10/2 = 5$ ) we can deduce that the number of steps from 10 to 1 is  $1 + 5 = 6$ . And it took only 2 iterations instead of 6 for the naive algorithm. The problem to evaluate this new algorithm is that we cannot predict how many iterations it will take before we reach a value we already know.

To overcome this we also did an empirical approach to estimate this value. We note  $g(x)$  the number of **iterations** it took with this new algorithm.  $x$  goes also from 1 to 8.

```

4
9.1010101010101
6.3833833833834
6.1842184218422
6.2091520915209
6.2262642262642
6.2359355235936
6.2389850023899

```

As we can see it seems that the values of  $g(x)$  are converging to a value near 6.24. This is great because we can assume that with this algorithm the amortized complexity for each integer is constant. We now have the complexity for the new algorithm:

$$T \times g(\log_{10}(T)) = \mathcal{O}(T)$$

This is great because we got rid of the  $\log_{10}(T)$  term to reach linear complexity. An important thing to note is that the constant of our algorithm is quite small too (way better than the constant of the naive algorithm).

## Distributing the improved algorithm

The naive algorithm is very easy to parallelise because there is no shared state between the iteration. On the other side in the improved algorithm we memorize the previous values. The problem is that nodes cannot share all their memory. If we distribute the algorithm, at some point precomputed values will not be available we needed. We had to evaluate the how the algorithm behaves when only part of memorized values are available.

Here are the number of **iterations** needed to compute the values from  $10^1$  to  $10^8$  if only 50% of the precomputed values are availables:

```
7.66666666666667
11.058823529412
7.7884231536926
8.0161967606479
8.1495170096598
8.1739256521487
8.1845663630867
8.1816952563661
```

And here are the result if only 1% of the values are availables:

```
7.44444444444444
31.737373737374
54.368314833502
40.619634380366
41.315612973606
41.427014720187
42.121434533188
42.143153897544
```

As the experiment shows, it seems that the value is still constant, Only the the constant change. This is great for us because it means we still have a linear algorithm even if all the memory is not shared between nodes.