On Evaluating the empirical complexity of the Miller-Rabin primality test

Garrenlus de Souza
Instituto de Informática
UFRGS - Universidade Federal do Rio Grande do Sul
Porto Alegre, Brasil
gsouza@inf.ufrgs.br

I. Introduction

The algorithm under analysis is the Miller-Rabin [1] primality test. The overarching goal of the presented work is to assess how empirical measurements over a real implementation of such algorithm behaves relative to its known worst-case asymptotic runtime complexity. The following formulation is an adaptation of [2].

Algorithm 1 Miller-Rabin Primality Test

Require: An odd integer *n*.
Ensure: If the answer is *PP* {*Possibly Prime*} then *n* is not a prime with at most $1/4$ probability. Furthermore, a *C* {*Composite*} answer is always correct.

1: $a \leftarrow$ a random value $v \in [1, n-1]$
2: **if** a and n are not coprimes **then**
3: **decide** *C*
4: **end if**
5: **find** u, t such that: $n - 1 = 2^t u$
6: **if** $a^u \equiv 1 \mod n$ **then**
7: **decide** *PP*
8: **end if**
9: **for** $i \in [0, t-1]$, $i \in \mathbb{Z}$ **do**
10: **if** $(a^u)^{2^i} \equiv -1 \mod n$ **then**
11: **decide** *PP*
12: **end if**
13: **end for**
14: **decide** *C*

II. Methodology

A. Environment

For the current study, the applied methodology consisted primarily into the analysis of running time statistics made over a reference implementation written in C++. The operating system is Ubuntu 20.04 LTS, running over a 32 GB RAM, 11th Gen Intel Core i7-1185G7 at 3.00GHz clock-speed system. All the code was compiled using g++ 9.4.0 (built for this very platform) through a CMake enabled build pipeline set to *"Release"* mode. The instances evaluated are available at [3] the GMP [4] library was employed in order to handle large numbers. The timing fixtures are native to C++. All the data structures employed come from the *STL*[5] unless otherwise stated.

B. Implementation

Translating the pseudocode into the structure of C++ was quite straightforward with exceptions for the steps *5* and *10*. The former demanded the implementation of Algorithm 2).

Algorithm 2 $\{2^t u\}$-Factorization

Require: An odd integer *n*.
Ensure: t and u are such that $n = 2^t u$.

1: $u \leftarrow n$
2: $t \leftarrow 0$
3: **while** $isEven(u)$ and $u > 0$ **do**
4: $u \leftarrow u >> 1$
5: $t \leftarrow t + 1$
6: **end while**
7: **return** t, u

It is quite obvious that such algorithm finds t and u in at most $O(logn)$ time. For the latter step, it was necessary to lay hands over some important properties of modular arithmetic, more specifically Equation 1.

$$a^k \equiv b^k \mod c \tag{1}$$

Such that: $k : \{\in \mathbb{N}_{>0}\}$ and $a, b \in \mathbb{Z}$.

This was necessary as to keep the calculation operands as small as possible given the fact that such algorithm is meant to deal with numbers thousands of digits long (base 10), otherwise GMP would possibly not be able to handle such huge numbers given the amount of memory available. This property was applied through the following modification applied over Algorithm X.

C. Instances

For the empirical evaluation of the presented algorithm, some test instances were generated. Each instance consists in a odd integer v. Each value v was built through the repeated juxtaposition of random values in the $[0, 9]$ range with the goal of getting values with exactly k digits ($3 \leq k \leq 3000$) as this information is key for the proposed analysis. Converting such values to a numeric representation was hurdle free thanks to the GMP API that allowed such task to be performed directly.

Algorithm 3 Modified Miller-Rabin Primality Test

Require: An odd integer n.

Ensure: If the answer is *PP* {*Possibly Prime*} then n is not a prime with at most $1/4$ probability. Furthermore, a *C* {*Composite*} answer is always correct.

1: $a \leftarrow$ a random value $v \in [1, n-1]$
2: **if** a and n are not coprimes **then**
3: **decide** *C*
4: **end if**
5: **find** $t, u \in \mathbb{Z}$ such that: $n - 1 = 2^t u$
6: $r \leftarrow a^u \mod n$
7: **if** $r \equiv 1 \mod n$ **then**
8: **decide** *PP*
9: **end if**
10: **for** $i \in [0, t-1]$, $i \in \mathbb{Z}$ **do**
11: **if** $r^{2^i} \equiv -1 \mod n$ **then**
12: **decide** *PP*
13: **end if**
14: **end for**
15: **decide** *C*

D. Hypothesis

The algorithm is known to have

$$O(klog^3 n) \tag{2}$$

runtime, with k being the number of *rounds*, each *round* an execution of the algorithm concerning one base a selected uniformly at random. The presented hypothesis is that the current implementation abides to this upper bound. Furthermore, it is also of interest to evaluate the distribution of bases that induce the algorithm to report a composite number as possibly prime. This distribution is known to be at most $1/4$ for a single round.

The upper bound analysis is conducted over the number of digits (D), and not the magnitude of the numbers themselves (their modulo). Hence, it is necessary to adapt the previously mentioned worst-case limit accordingly. Since d equals the floor of $logn$ added by 1. For the purpose of this work we can take $logn = d-1$, given that such function never overestimates 2 for all valid k. This way the new limit can be stated as $(D(n) - 1)^3$.

In order to assess the runtime growth, a basic runtime unit t was defined as $T(m)^{1/3}/(D(m)-1)$ (for m as the smallest instance evaluated). So the order of T, the expected time for input n can be written as $O(((D(n) - 1)t)^3)$, or more specifically:

$$O(((D(n) - 1)T(m)^{1/3}/(D(m) - 1))^3) \tag{3}$$

III. Results

A. Runtime

In order to offset some of the noise in the experimental environment, each black dot in Fig. 1 corresponds to the median of a total of 20 executions of the algorithm. A mean was evaluated but such metric presented itself too sensitive to outliers and was dropped in favor of a more representative value.

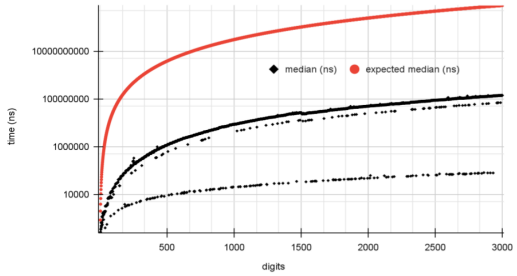

Fig. 1. Runtime as a function of the number of digits. The red dots represent the expected (estimated) time and the vertical axis is in logarithmic scale.

Notice that the observed values diverge greatly from the expected estimates. Another curious aspect of the numbers plotted in Fig. 1 is the existence of three seemingly distinct distributions. This could in turn indicate some interesting property of such numbers given the fact that such differences (between values compared from any two of them) are presented in a logarithmic scale. Though eerily inviting for further investigation, the authors decided not to take this path and withdrawn attention, focusing on delivering the report as soon as possible.

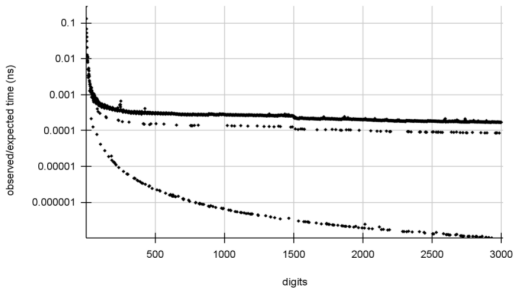

Fig. 2. Ratio between the observed and the expected (estimated) time as the number of digits grow (x axis). The y axis is logarithmic scale.

This division pattern can also be found in Fig. 2, where one can see the that values quite spread-out in the range of digits seem to plateau (the top two "curves") while the third (bottom "curve") one seems to converge to $y = 0$.

B. False Positives

In order to evaluate the distribution F of false positives, the first 1000 instances got their whole range of possible bases a tested. As each one of them is known to be composite, it was enough to count the number of false positives (answer is PP) and divide it by $(n-1)+1$. The results are in Fig. 3.

Covering the same range of the previous experiment was not feasible as the running time grow quite quickly when we apply each possible base a ($n-1$ possible values), so an 8 hours limit was set. This limit allowed the employed machine to reach up to the instance with 10 digits.

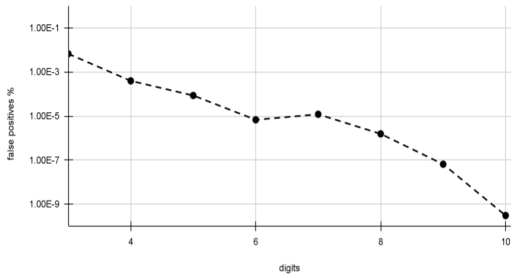

Fig. 3. The observed probability of the algorithm on choosing a base that erroneously map a composite instance to PP for instances with up to 10 digits.

TABLE I
FAILURE RATE FOR $d \in [3, 10]$

digits	**lying bases%**
3	0.00668896321070234
4	0.000395022713806043
5	0.0000856494368549526
6	0.00000674989284545107
7	0.0000119907404837375
8	0.0000015490490459622
9	6.48523401387009e-8
10	3.04212415552544e-10

IV. CONCLUSION

In summary, the runtime grew by an order that closely resembles d^3 as expected, though scaled down by a factor that ranged between $3\times$ and $9898610\times$, with a median of $4631\times$. The running time grows in a curious alternating pattern, though the reason for this behavior is not quite understood (nor was investigated), leaving such detail to get further attention in future works. The observed ratio of bases that lead the algorithm to lie ($A(n) \to PP$) ranged between $6.69E{-}03$ and $3.04E-10$ for the evaluated instances, with up to 10 digits, far away from the 0.25 worst-case found in literature [1]. The entire source-code is available at [3] (including deterministic test-generation scripts), raw numbers and presented plots can be found at [6].

REFERENCES

[1] "The miller–rabin test," https://kconrad.math.uconn.edu/blurbs/ugradnumthy/millerrabin.pdf, Accessed: 2023-4-14.
[2] "Inf05010 – algoritmos avançados, notas de aula," https://www.inf.ufrgs.br/~mrpritt/lib/exe/fetch.php?media=inf05016:notas-1310661.pdf, Accessed: 2023-4-14.
[3] "Assessing the miller-rabin primality test," https://github.com/GarrenSouza/inf05016-primality, Accessed: 2023-4-14.
[4] "Gnu multiple precision," https://gmplib.org/manual/, Accessed: 2023-4-14.
[5] "Containers," https://cplusplus.com/reference/stl/, Accessed: 2023-3-26.
[6] "Miller-rabin statistics," https://docs.google.com/spreadsheets/d/1TiZ14DQQNh4r0NkYZcYWDGV8MUM9a0ZdzNzNOXV-Xz4/edit?usp=sharing, Accessed: 2023-4-14.