DLC Project Report
Simple Methods for Factorization

DOAN Thi Van Thao
NGUYEN Thi Mai Phuong
TRAN Danh Nam

January 2022

Abstract

This report shall give a brief description of the three simple methods for factorization: trial division, Pollard's *rho* and Pollard's $p-1$ algorithms. We provide experimental results of each method, and test the performance of the general factorization tool which applies all the three methods above. The main sources of reference are [1] and [2].

Contents

1 Description of the Methods **2**
1.1 The method of trial division by prime numbers 2
1.2 Pollard's *rho* algorithm 2
1.3 Pollard's $p-1$ algorithm 4

2 Experimental results **6**
2.1 The method of trial division by prime numbers 6
2.2 Pollard's *rho* algorithm 7
2.3 Pollard's $p-1$ algorithm 9
2.4 General tool for factorization 11

3 Conclusion **12**

A Graphic User Interface (GUI) **13**

1 Description of the Methods

1.1 The method of trial division by prime numbers

The trial division was first introduced by Fibonacci in 1202. It is the simplest method of factorization, but also the most time-consuming.

Given an integer to be factored n, the trial division method will systematically test all integers smaller than n and see if they divide n. However, this brute force variant is an overstatement with a complexity of $O(n) = n$. Obviously, there is no need to test divisibility by 4 if n is not divisible by 2. Therefore, the task is simply selecting prime numbers as candidate divisors.

Suppose that p_i is the i^{th} prime number, i.e. $p_1 = 2, p_2 = 3, p_3 = 5, p_4 = 7, \ldots$ It is known that if n has any non-trivial divisors d then d is at most $\lfloor\sqrt{n}\rfloor$. For that reason, we only need to test divisibility by prime numbers up to $\sqrt{n}$. Apparently, the method of trial division by prime numbers is more efficient than the previous one with complexity $O(\sqrt{n})$.

Algorithm 1: The method of trial division by prime numbers

Input: a composite number n, an upper bound p_{max}
Output: a nontrivial prime factor of n

```
for prime numbers p_i ⩽ p_max do
    if n mod p_i = 0 then
        return p_i ;                  // non-trivial prime factor found
    end
end
return failure ;                      // no prime factor at most p_max
```

In general, we want to compute the factorization of n as

$$n = \prod_{i=1}^{k} p_i^{e_i}$$

where p_i are the prime factors and e_i are the corresponding exponents. Hence we may apply Algorithm 1 to find a prime factor p of n, then find the largest prime power p^e which divides n. Then we continue applying the algorithm to find a prime factor of n/p^e. Note that a prime factor of n/p^e must be larger than p, thus we can modify Algorithm 1 to start testing prime numbers from a lower bound p_{min}. Continue the process until all prime factors of n is found or failure, we can find the complete factorization, or partial factorization of n.

The value p_{max} determines the upper limit of prime numbers that we want to test for divisibility. Hence, in the worst case, the larger p_{max} results in the longer running time, but higher possibility of finding a factor of n.

1.2 Pollard's *rho* algorithm

Consider a random function $f : S \to S$, where S is a finite set of n elements. Let $x_0, x_1, x_2, \ldots$ be a sequence defined by: $x_0 \in S$ and $x_{i+1} = f(x_i)$ for $i \geqslant 0$. This sequence must be periodic starting from some index, hence we can find collision of the function f in this sequence (with complexity of time and space $O(\sqrt{n})$ by the birthday paradox). In fact, the non-periodic part of f has expected length $\sqrt{n\pi/8}$ and the periodic part has expected length $\sqrt{n\pi/8}$. One can visualize the non-periodic part as the tail of the Greek letter ρ, and the periodic part as the cycle of ρ.

In 1985, John M.Pollard described a method of factorization, the idea of which is essentially finding a collision in the sequence (x_i) for a well-chosen function f. Let n be a composite number and p be its prime factor. Pollard considered the function $f(x) = x^2 + 1 \mod p$. Assuming f is "random" enough, given $O(\sqrt{p})$ values of the sequence one may find a collision $f(x_i) = f(x_j)$. However, since p is not known, $f(x)$ is implicitly computed via the function $F(x) = x^2 + 1 \mod n$ (note that $F(x) \equiv f(x) \mod p$). Thus we have $F(x_i) \equiv F(x_j) \mod p$ or p divides $\gcd(F(x_i) - F(x_j), n)$. Then one may find a nontrivial factor of n by computing $\gcd(F(x_i) - F(x_j), n)$.

To reduce the memory cost, Pollard applied the idea of Floyd's cycle detection algorithm: two pointers, namely x and y are used; pointer x holds the values of x_i's and pointer y holds the values of x_{2i}'s. Each iteration updates the values of x and y by computing $f(x)$ and $f(f(y))$, then checks if $\gcd(x_i - x_{2i}, n) = \gcd(x - y, n)$ is a nontrivial factor of n. This reduces the memory cost to $O(1)$.

Assuming f is a random function, then the expected number of evaluations to the function f performed by Pollard's *rho* algorithm is $O(\sqrt{p}) = O(\sqrt[4]{n})$, where p is the smallest prime factor of n.

Algorithm 2: Pollard's *rho* algorithm using Floyd's cycle detection

---

Input: a composite number n, a bound B for the number of iterations
Output: a nontrivial factor of n or failure

$x \leftarrow 2$; `// Set` $x = x_0 = 2$ `to be the initial value`
$y \leftarrow 2$; `// Set` $y = x_0 = 2$
$d \leftarrow 1$;
$i \leftarrow 0$;
while $d = 1$ **OR** $d = n$ **do**
    if $i \geqslant B$ **then**
        return *failure*; `// Maximum number of iterations reached`
    end
    $x \leftarrow f(x)$; `//` $x = x_i$
    $y \leftarrow f(f(y))$; `//` $y = x_{2i}$
    $d \leftarrow \gcd(|x - y|, n)$;
    $i \leftarrow i + 1$;
end
if $d = 1$ **OR** $d = n$ **then**
    return *failure* ;
end
else
    return d ;
end

It is recommended to choose $f(x) = x^2 + c \mod N$ where $c \neq 0, -2$; since such functions f are conjecture to be "random".

In 1980, Richard Brent proposed a variant of Pollard's *rho* algorithm, which used a different method of cycle detection (named after Brent himself).

Algorithm 3: Pollard's *rho* algorithm using Brent's cycle detection

Input: a composite number n, a bound B

Output: a nontrivial factor of n or failure

```
y ← 2 ;                    // Set y = x_0 = 2 to be the initial value
d ← 1;
r ← 1;                                          // r is a power of 2
while d = 1 OR d = n do
    x ← y ;                                             // x = x_i
    for j = 1 to r do
        y ← f(y) ;                              // Compute y = x_{i+r}
    end
    k ← 0;
    while k < r AND d = 1 do
        y ← f(y) ;           // Compute y = x_{i+r+k+1} for 0 ⩽ k < r
        d ← gcd(|x − y|, n);
        k ← k + 1;
    end
    r ← 2r ;                               // go to the next power of 2
    if r ⩾ B then
        return failure ;                       // power of 2 exceeds B
    end
end
if d = 1 OR d = n then
    return failure;
end
else
    return d;
end
```

We summarize the core ideas of Brent as following

- Find the smallest power of 2 (variable r in the algorithm) which is larger than the period of f
- Use only one pointer to keep track of the sequence (x_i) and make only one evaluation to f to update the sequence
- Only test $\gcd(x_i - x_{i+k}, n)$ where $r < k < 2r$. Note that when r is less than the period of f, there is no collision within $x_i, x_{i+1}, \ldots, x_{i+r-1}$ and thus it is not worth testing $\gcd(x_i - x_{i+k}, n)$ when $k < r$. This reduces the number of $\gcd$ computations

With these changes, Brent claimed that his method worked 24 per cent faster on average comparing to the original version of Pollard. Details are presented in [3].

1.3 Pollard's $p - 1$ algorithm

The Pollard's $p-1$ algorithm is a number theoretic integer factorization algorithm, proposed John M. Pollard in 1974 [4]. The idea is based on Fermat's Little Theorem

Fermat's Little Theorem. *Let p be a prime and $a \in \mathbb{Z}$ such that $p \nmid a$, then*

$$a^{p-1} \equiv 1 \mod p$$

It follows that p divides $d = a^{p-1} - 1$, for any a relatively prime to p. Thus if p is a prime factor of n, p divides $\gcd(d, n)$. Hence by computing $\gcd(d, n)$ we may find a nontrivial factor of n. However, we can not directly calculate d because p is unknown at first. The idea is to replace $p - 1$ by a number M such that $p - 1 \mid M$ or $M = k(p-1)$. It follows that $a^M \equiv 1 \mod p$ and then we may compute $\gcd(a^M - 1, n)$ instead. Usually, M is chosen to be a product of many prime powers no greater than a value B, called *smoothness bound*.

Unlike the two previous methods, the possibility of finding a factor p of given size is not determined solely by its size, but rather by the smoothness of $p-1$. This is also the reason why this algorithm is named Pollard's $p - 1$.

If B is the smoothness bound; $p_1 < p_2 < \cdots < p_m \leqslant B$ are all the prime numbers up to B; e_i is the maximum exponent such that $p_i^{e_i} \leqslant B$, then

$$M = \prod_{i=1}^{m} p_i^{e_i}$$

Since M grows very large compare to the smoothness bound B, we avoid computing M explicitly by computing $a^M \mod n$ instead. This can be done by iteratively computing $a^{p_i^{e_i}} \mod n$ for each prime numbers p_i.

Algorithm 4: Pollard's $p - 1$ algorithm

Input: a composite number n, a bound B
Output: a nontrivial factor of n or failure

1 Choose a positive integer base a randomly between 1 and n.;
2 Compute $d = \gcd(a, n)$;
3 **if** $d \neq 1$ **then**
4   **return** d;
5 **end**
6 **for** prime numbers $p_i \leqslant B$ **do**
7     $q \leftarrow 1$;
8   **while** $q \leqslant B$ **do**
9         $a \leftarrow a^{p_i} \mod n$;
10         $q \leftarrow q \times p_i$;
11   **end**
12     $c \leftarrow a - 1$;
13     $d \leftarrow \gcd(c, n)$;
14   **if** $d \neq 1$ **AND** $d \neq n$ **then**
15   **return** d;
16   **end**
17   **if** $d = n$ **then**
18   Go to line 1 and choose a new value for a;
19   **end**
20 **end**
21 **if** $d = 1$ **then**
22   **return** *failure*;
23 **end**

Here, the bound B plays an important role in defining indirectly the size of the exponent M. A larger B increases the the probability of finding a factor of n, but it also increases the algorithm's complexity, and so the time needed to perform the algorithm.

There are two cases of failure: $d = 1$ or $d = n$. In the first case, $a^M - 1$ is coprime with n, which implies that the search bound B is too small, and thus one should rerun the algorithm with a larger B. In the second case, $d = n$ implies that n has a B-smooth prime factor p, but the randomized base a has order less than $p - 1$ modulo p (hence omitted for $\gcd$ computation in the loop from line 8 to 11). In this case we choose another base a and restart the algorithm.

There exists a *two-stage* variant of Pollard's $p - 1$ algorithm. The second-stage is performed by choosing a second bound $B_2 > B$, normally $B_2 = 100B$. When the first stage of the algorithm fails, it may be the case that n has a prime factor p such that $p-1 = u \cdot q$, where u is B-smooth and q is a prime number in the interval $(B, B_2]$. The idea is to start from the value $a^M \mod n$ computed at the end of first stage and proceed to compute $a^{Mq_1} \mod n, a^{Mq_2} \mod n, \dots, a^{Mq_t} \mod n$, where $B < q_1 < q_2 < \cdots < q_t \leqslant B_2$ are all the primes between B and B_2. Then we test $d = \gcd(a^{Mq_i} - 1, n)$ for a nontrivial factor as in the first stage.

To saving time computing modular exponent, $a^{Mq_{i+1}}$ is iteratively calculated as $a^{Mq_{i+1}} = a^{Mq_i} a^{M(q_{i+1}-q_i)}$, where $q_{i+1} - q_i$ is the prime gap between q_{i+1} and q_i. Note that the prime gap between prime numbers is quite small (no larger than 1000 up to 18-digit prime numbers). Hence we can precompute the values $a^{M(q_{i+1}-q_i)} \mod n$ as $a^{Md} \mod n$, where $d = 2, 4, 6, \dots$ up to a few hundreds. Then the calculation of each a^{Mq_i} for $1 < i \leqslant t$ costs only one modular multiplication.

2 Experimental results

2.1 The method of trial division by prime numbers

The following table shows the results when we compute the factorization of 100 integers of 40 digits and 100 integers of 50 digits, using Algorithm 1. For each testing set, we increase the different values of $p_{\max}$ to evaluate its effect on the final result. The final results is either 1 of 3 possibilities: *Fully factored*, *Partially factored* or *Failure*.

Size n	p_{max}	Average time(s)	Fully factored	Partially factored	Failure
40	10^3	0.0011862	5	34	61
40	10^4	0.0104638	6	46	48
40	10^5	0.0959057	11	52	37
40	10^6	0.8707471	12	58	30
40	10^7	7.4829546	21	58	21
50	10^3	0.0011692	3	31	66
50	10^4	0.0105942	7	44	49
50	10^5	0.1046316	13	51	36
50	10^6	0.8542737	14	61	25
50	10^7	7.7956228	16	62	22

Table 1: Experimental results of method of trial division by prime numbers.

The result and execution time depend on the value of $p_{\max}$. When $p_{\max}$ increases, the number of failure cases are lower, but the executions time are longer. In practice, the method of trial division by prime numbers is used to find small prime factors up to 10^7, any other factor of larger size is found by applying more efficient methods.

2.2 Pollard's *rho* algorithm

We present experimental results of Pollard's *rho* algorithm. The first is to test if certain choices of initial value x_0 or function $f(x)$ affect the running time.

We randomize the initial values x_0 100 times and run the Pollard's *rho* algorithm (using Floyd's cycle detection) to find a non-trivial factor of the following three numbers

$$\begin{aligned}
n_1 &= 1125939825397831601 \\
&\quad \text{(a 60-bit RSA modulus)} \\
n_2 &= 925276410789441750962080530947 \\
&\quad \text{(a 100-bit RSA modulus)} \\
n_3 &= 115792089237316195423570985008687907853269984665640564039457584007913129639937 \\
&\quad \text{(Fermat number } F_8 = 2^{2^8} + 1\text{)}
\end{aligned}$$

For each run we use the same function $f(x) = x^2 + 1 \mod n$ and same maximum number of iterations $B = 10^8$. The results are in Table 2

Input number	Average running time (s)	Standard deviation
n_1	0.0054	0.0016
n_2	5.9370	3.3053
n_3	67.0664	43.6244

Table 2: Average running time with different initial values

It can be seen that when the size of input number is small (around 100 bits or 30 digits), the standard deviation is rather small compare to the average running time. However when the size of input number grows large (in our test cases, n_3 has 256 bits or 78 digits), standard deviation grows relative large compare to the average running time. Note that the smallest prime factors of n_2 and n_3 are roughly same size (15 digits compare to 16 digits), but factoring n_3 takes more time due to the higher complexity of modular arithmetic (which grows with respect to the size of the input number). The results of Table 2 suggest that certain initial values speed up the running time of the algorithm, but these values depend on the input number.

Next we test the Pollard's *rho* algorithm with different choices of the function $f(x)$. Using the same numbers n_1, n_2, n_3; $B = 10^8$ and initial value $x_0 = 2$; we run the algorithm with $f(x) = x^2 + 1 \mod n$, $f(x) = x^2 + 2 \mod n$ and $f(x) = x^2 + c \mod n$, where c is randomized for 100 runs. Then we compare the average running time of these 100 runs with the two runs using $x^2 + 1 \mod n$ and $x^2 + 2 \mod n$. The results are in Table 3.

Input number	Running time (s) for $x^2+1 \mod n$	Running time (s) for $x^2+2 \mod n$	Average of 100 runs	Standard deviation
n_1	0.0114	0.0181	0.00536	0.00306
n_2	2.8195	3.7504	8.4493	4.8954
n_3	27.84818	35.18842	68.9762	35.0732

Table 3: Running time with different choices of function f

Table 3 shows that the two function $f(x) = x^2 + 1 \mod n$ and $f(x) = x^2 + 2 \mod n$ give faster running time compare to the average running time when f is randomized. Also, the large value of standard deviation implies that certain choices of f speed up the algorithm, while some others take longer time.

We also compare the running time of the version based on Floyd's cycle detection algorithm with the version based on Brent's cycle detection algorithm. We run both version with 1000 numbers (500 numbers with 40 digits and 500 numbers of 50 digits), using $f(x) = x^2 + 1 \mod n$ and $B = 10^9$, count the number of success (a non-trivial factor found) and average running time of successful attempts for every 100 runs. The results are in Table 4

Size of input	Avg running time (s) (Floyd)	Success (Floyd)	Avg running time (s) (Brent)	Success (Brent)
40 digits	13.8400	99	9.7651	99
40 digits	5.9126	99	4.4527	99
40 digits	7.8075	99	11.6436	100
40 digits	6.9863	97	13.3923	98
40 digits	4.9465	98	3.9956	98
50 digits	2.0018	100	1.9188	100
50 digits	2.7532	95	7.8230	96
50 digits	11.9425	94	10.9833	94
50 digits	10.4239	99	9.8133	99
50 digits	3.5810	96	2.3657	96

Table 4: Average running time of different cycle detection method

From Table 4, we see that for any instance of 100 runs for which both methods have the same number of success, Brent's method performs better than Floyd's method on average. There are some instance of 100 runs such that Brent's method takes longer time on average. However, this happens because in these 100 runs there exists some input numbers of which the smallest prime factor is very large, hence take a lot of time for Brent's method (but result in failure for Floyd's method).

We apply the Pollard's *rho* algorithm (with Brent's cycle detection) to compute the factorization as follows: we run the algorithm twice with $f(x) = x^2+1 \mod n$ and $f(x) = x^2 + 2 \mod n$ to find a nontrivial factor d of n. If d is prime, compute the largest prime power d^e dividing n and continue the process with n/d^e. If d is not prime, we continue from the start with d as the input, until finding a prime factor of d (which is also a prime factor of n). With the strategy described, we compute the factorization of 100 numbers of 40 digits and 100 numbers of 50 digits, using different value of bound B. The results are presented in Table 5

Size n	B	Avg time (s) (of success)	Fully factored	Partially factored	Failure
40	10^6	0.1378	81	12	7
40	10^7	0.7019	94	2	6
40	10^8	2.3958	95	1	4
40	10^9	14.3434	99	0	1
50	10^6	0.5238	54	33	13
50	10^7	5.6196	71	19	10
50	10^8	26.4083	84	8	8
50	10^9	81.0459	91	2	7

Table 5: Performance of Pollard's *rho* when computing factorization

In general, the running times grows with respect to the value of B. Choosing $B = 10^9$ will result in factorization for almost every input number, but the running time is quite long. It seems reasonable to choose $B = 10^6$ or 10^7 when computing the factorization using Pollard's *rho* only.

Table 6 shows the running time and size of factor when we run the algorithm with $B = 10^{10}$ to find a medium-size factor of some worst-case numbers

$$\begin{aligned}
n_4 &= 237130450584081431781941097598542348001 \\
&= 15351399207396244631 \times 15446829789289363271 \\
&\quad \text{(a 128-bit RSA modulus)} \\
n_5 &= 304075290252258958535257891241265214597 \\
&= 18229633569899862109 \times 16680274405204585033 \\
&\quad \text{(a 128-bit RSA modulus)} \\
n_6 &= 34! - 1 \\
&= 295232799039604140847618609643519999999 \\
&= 10398560889846739639 \times 28391697867333973241
\end{aligned}$$

Input number	Running time (s)	Digits in factor
n_4	970	20
n_5	819	20
n_6	27.1042	20

Table 6: Running time to find medium-size factor using Pollard's *rho*

It can be seen that Pollard's *rho* takes a lot of time to find a factor of medium size (around 20 digits).

2.3 Pollard's $p - 1$ algorithm

In this part, we present experimental results of Pollard's $p - 1$ method for factorization problem. Both search bounds B and B_2 for each stage are entered as inputs. In case the algorithm successfully finds prime factors of n, the result is in the form of *partially factored* or *fully factored* if possible.

To compare each method's performance, the test numbers used is as same as the one in two previous methods. The table below shows the results of testing 100 integers of 40 digits and 100 integers of 50 digits. The average running time is calculated over the success attempts (complete or partial factorization found) of each 100 runs.

Size n	B	B_2	Average time(s)	Fully factored	Partially factored	Failure
40	10^3	10^5	0.0185053	39	45	16
40	10^4	10^6	0.0789924	49	38	13
40	10^5	10^7	1.06857792	56	37	7
40	10^6	10^8	8.18924395	75	21	4
50	10^3	10^5	0.0266136	25	54	21
50	10^4	10^6	0.2113204	36	52	12
50	10^5	10^7	1.91126	50	42	8
50	10^6	10^8	12.79705	72	22	6

Table 7: Experimental results of Pollard's $p-1$ algorithm.

When the search bounds increase, the average running time is also increased. For the same bounds, the increase of size n leads to the increase of factorization time. The bigger the bounds are, the bigger the time difference between runs is. For example, with the bounds (B, B_2) of $(10^3, 10^5)$, the processing time range of 40-digit and 50-digit n is $[0.0004s, 0.04s]$ and $[0.0009s, 0.04s]$ respectively. However, when the bounds (B, B_2) change to $(10^6, 10^8)$, these ranges reach $[0.004s, 26s]$ and $[0.0005s, 32s]$ for 40-digit and 50-digit n respectively.

Besides, Table 7 also illustrates how many times out of 100, the Pollard's $p-1$ algorithm returns results as success and failure. It is obvious that when the bounds increases, the possibility of full factorization is increased. Once the bound B reaches to 10^6, almost 72 to 75% the integers n are fully factored.

We also tried finding large factor of some numbers with record factors found by Pollard's $p-1$ method listed in [5]. The running time for each is displayed in Table 8.

Digits of factor	n	B_1	B_2	Time(s)
32	$2^{977}-1$	10^7	10^8	14.9582
34	575th Fibonacci number	10^7	10^8	14.3806
66	$960^{119}-1$	10^8	10^{10}	1076

Table 8: Some record factors by Pollard's $p-1$ method.

We list the factor of each number below

$4985899058078884305401269007884 1$
(32-digit factor of $2^{977} - 1$)

$7146831801094929757704917464134401$
(34-digit factor of 575th Fibonacci number)

$672038771836751227845696565342450315062141551559473564642434674541$
(66-digit factor of $960^{119} - 1$)

2.4 General tool for factorization

The general tool is designed as following: first we test if the input number n is a perfect power and find the maximum k such that $n = m^k$ for some integer k. If such m and k exist, the task reduces to computing the factorization of m. Then we apply the three methods: trial division, Pollard's *rho* and Pollard's $p-1$ in that order to find all the prime factors of n (or m).

Using the following parameters

- $p_{\text{max}} = 10^7$ for trial division;
- $B = 10^7$ for Pollard's *rho* (Brent's cycle detection);
- $B_1 = 10^6$ and $B_2 = 10^7$ for two-stage Pollard's $p - 1$;

we compute the factorization of 300 numbers of 40 digits and 300 numbers of 50 digits. We compute the average running time for every success attempts (completely or partially factorization) in every 100 runs.

Size n	Avg time (s) (of success)	Fully factored	Partially factored	Failure
40	27.2430	100	0	0
40	27.0013	100	0	0
40	39.8608	100	0	0

Table 9: Performance of the general factorization tool for 40-digit numbers

Quite interestingly, the tool succeeded in factorizing completely 300 numbers with the average running time less than 40 seconds. However, there are some instances of input number which take up to nearly 50 minutes of running time, while for almost every other case it only takes less than 15 - 20 seconds.

Next we compute the factorization of 300 numbers of 50 digits, and compute the average running time for successful attempts in every 100 runs. We use the following parameters

- $p_{\text{max}} = 10^6$ for trial division;
- $B = 10^6$ for Pollard's *rho* (Brent's cycle detection);
- $B_1 = 10^6$ for one-stage Pollard's $p - 1$;

We reduce the value of all parameters, since the previous choices of parameters result in longer running time for some cases. The result is shown in Table 10.

Size n	Avg time (s) (of success)	Fully factored	Partially factored	Failure
50	30.8103	89	4	7
50	49.9791	82	10	8
50	56.378	84	12	4

Table 10: Performance of the general factorization tool for 50-digit numbers

The tool succeeded in factoring nearly every number in each 100 runs (around 90 numbers), with the average running time less than a minute. We also observed that the failure cases took approximately 7 minutes of running time. Hence it seems that the choices of parameters above are more reasonable for computation.

3 Conclusion

In this project, we implemented 3 simple methods of factorization. For each method, we measured the average execution time and chance of success by changing some parameters. We also implemented a factorization tool using in combination the 3 methods, and tested with numbers with 40 digits and 50 digits. We conclude that our program can find the full factorization of almost every number around 40 and 50 digits (with suitable choices of parameters) in a somewhat reasonable amount of time.

We also tried implementing a straightforward version of the Elliptic Curve Method (ECM) for factorization. However, our implementation did not give a good performance compare to Pollard's *rho* and Pollard's $p - 1$. This may be due to the fact that ECM uses more sophisticated technique for optimization (thus a straightforward implementation is not enough).

A Graphic User Interface (GUI)

In this project, we use the library **Python tkinter** to build the GUI and the library **Python ctypes** to call functions of shared **C** library.

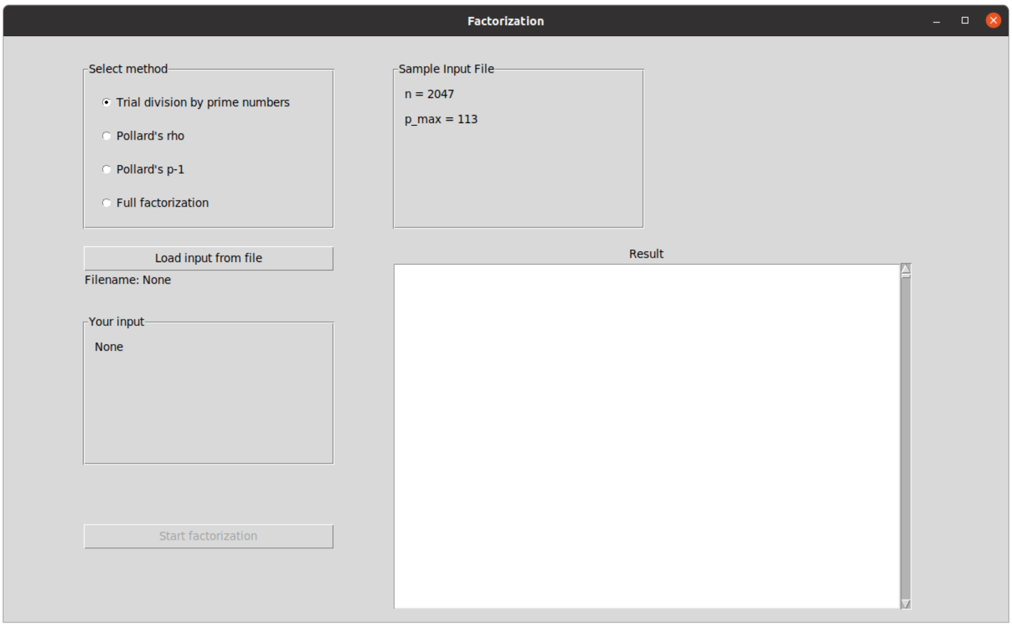

Figure 1: Graphical User Interface - Initialization

To begin, we choose the method that we want to use. Then, we select the input file which specifies the input number and parameters for each method (hence the format of the input file is different depending on the method chosen).

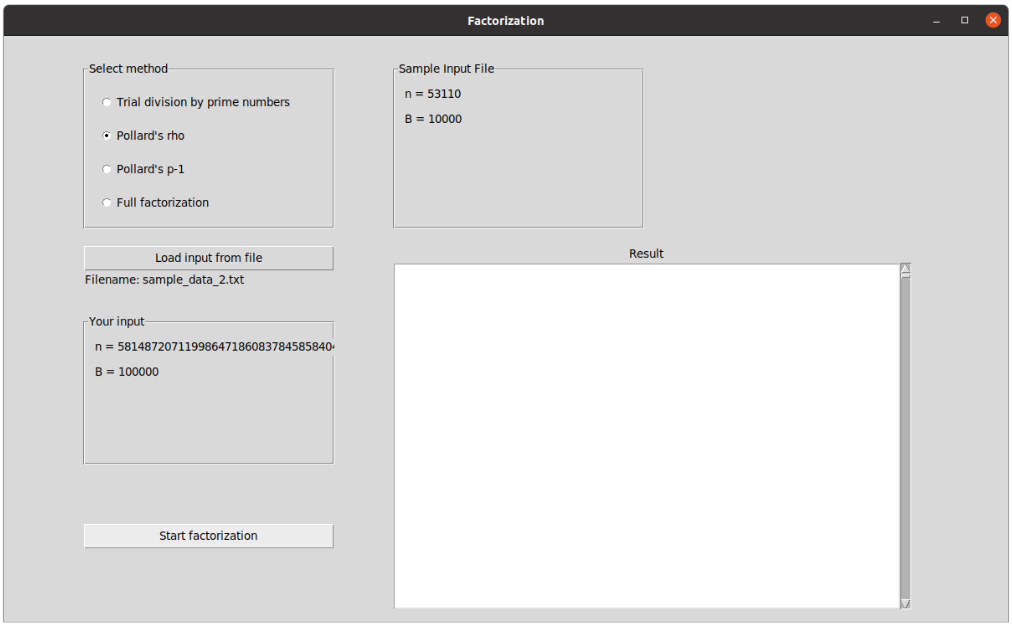

Figure 2: Graphical User Interface - Setup

The result will be displayed on the right side of the window. Within it, we list all the necessary information like input values, final results, and the total execution time as illustrated on Figure 3 below. We round the execution time to 9 decimal digits for easier to compare results between testing cases.

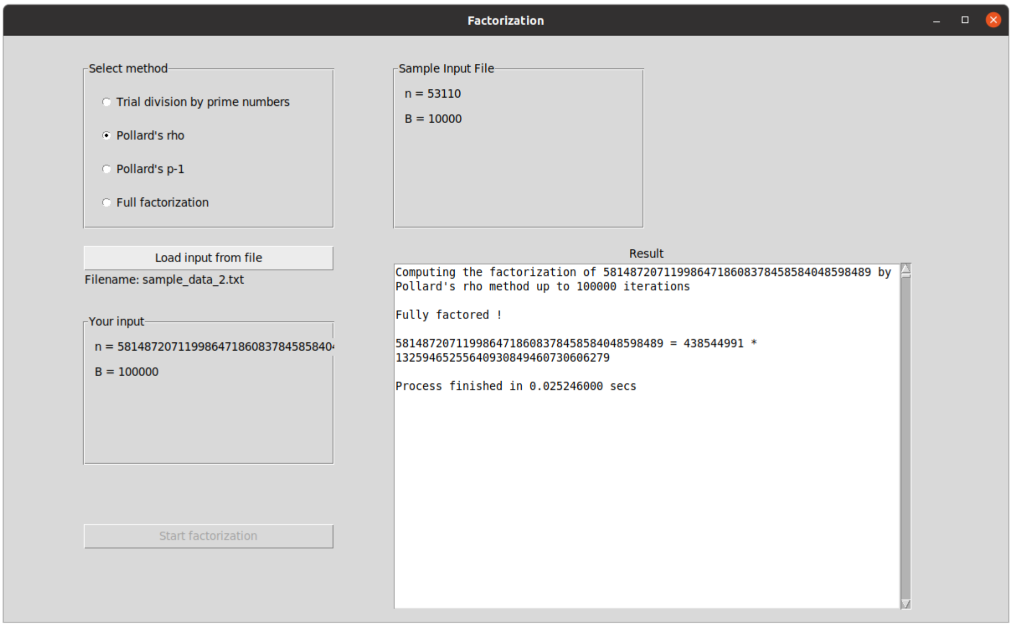

Figure 3: Graphical User Interface - Result

References

[1] Richard Crandall and Carl Pomerance. *Prime Numbers: A Computational Perspective*. Springer, 2nd edition, 2005.

[2] A.Menezes, P.van Oorschot, and S.Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[3] Richard P.Brent. An Improved Monte Carlo Factorization Algorithm. *BIT Numerical Mathematics*, 1980.

[4] J. M. Pollard. Theorems on factorization and primality testing. *Mathematical Proceeding of the Cambridge Philosophical Society*, 1974.

[5] Record factors found by Pollard's $p-1$ method. `https://members.loria.fr/PZimmermann/records/Pminus1.html`.