Integer Factorization

Kien Le

Contents

1 Abstract

This project is concerned with the problem of integer factorization and some algorithms that have been developed to solve it. First, the paper introduced the notations that will be used to analyze the time complexity of the presented algorithms. Then, four algorithms used to factor integers will be discussed and their theoretical run time will be analyzed. Then, those four algorithms will be implemented and tested against each other. One surprising result is that one of the algorithms perform much worse than the others, which indicates that its implementation is not optimized. Afterwards, the paper presents the state-of-the-art algorithms for factoring integers and some milestones that have been reached using those algorithms. Finally, a discussion on possible improvements that can be made and some problems of this project are laid out.

2 Introduction

2.1 Integer Factorization

The problem of integer factorization can be phrased simply as given an integer, returns a product of two integers that is equal to the original integer. We can describe this more formally as given an arbitrary number n, find some $x, y \leq n$ such that $n = xy$. This formulation of integer factorization, however, allows for the trivial answers of $x = \pm 1$ and $y = \pm n$ for all n. Furthermore, in cases where n is prime, the only answer possible is the trivial one. Instead, for this project, we will use a stronger formulation of integer factorization. Let n be a positive composite number. Find positive $x, y \neq 1, n$ such that $n = xy$. Here, n is a

positive number because solving integer factorization for a negative number can be reduced to factoring its inverse and multiply one of the factors by -1.

2.2 Significance of the Problem

One important application of integer factorization is in cryptanalysis of the RSA cryptosystem, one of the most important and widely-used cryptographic system. Its most important use is in encrypting data sent over the internet. Cryptanalysing RSA can be reduced mostly to factoring a number $n = pq$, where p, q are primes. Thus, solving integer factorization is equivalent to cracking RSA. If a method to factor integer efficiently is ever found, it would have massive ramifications for internet communication. Because a message encrypted by RSA can be (somewhat) efficiently decrypted in this hypothetical scenario, either larger primes have to be used, which will increase encryption time, or a new encryption scheme must be devised in order to ensure that information sent over the internet stays secure.

Aside from its use cryptanalysis of RSA, integer factorization also has some usage in solving mathematical problem. For example, [13] shows how it can be used in a combinatorics problem. Furthermore, integer factorization is also of theoretical interest. A fair number of number theory and abstract algebra is used in designing algorithms to factor integers. It is possible that in the process, some deeper insights can be made into those two fields.

3 Asymptotic notation and size of input

Throughout this paper, big-O notation will be used to describe the time complexity of algorithms. In order to make use of big-O notation, however, we first have to determine how to define the size of the input. For most algorithms, such as sorting a list, the object being worked on is a data structure, which has a easy-to-conceive notion of size. Seeing that the object being considered in this project are integers, however, it does not make much sense to define size in this way. Instead, we will make use of the definition of size presented at the

beginning of chapter 31 in [?], which is the number of bits needed to represent an integer. This number can be calculated as $\lfloor \log_2 n \rfloor + 1$ for some integer n. For the sake of simplicity, we will simplify this expression to $\log_2 n$.

One thing of note here is that in order to convert between two bases of a logarithm, we use the formula $\log_b n = \frac{\log_a n}{\log_a b}$[?]. Seeing that $\log_a b$ is a constant, we have $O(\log_b n) = O(\log_a n)$. In other words, in big-O notation, $\log_a n$ is equivalent to $\log_b n$ for any a, b. This equivalence means that we can omit the base when working with logarithms in big-O notation as a notational shortcut. Combining this with the above definition of input size, we can define a polynomial-time algorithm for factoring integers as an algorithm that takes an integer n as an input and produces two factors of n in time $O(\log^k n)$, where k is a constant. Along the same lines, we can define an exponential-time algorithm for factoring integers as an algorithm that takes an integer n as an input and produces two factors of n in time $O(k^{\log n})$.

Another tool that is utilized in this paper is the L-notation, which is also used to describe the time complexity of an algorithm. Given an input n, the L-notation is defined as $L_n[\alpha, c] = e^{(c+o(1))(log(n))^{\alpha}(log(log(n)))^{1-\alpha}}$, where $o(1)$ is a function that tends to 0 when n goes to ∞ [11]. It is known that $L_n[0, c]$ is polynomial and $L_n[1, c]$ is exponential in $\log n$. The remaining class of functions, where $0 < \alpha < 1$, are called sub-exponential. A sub-exponential function is defined as a function that is, asymptotically, greater than all polynomials and less than all exponential function [11].

4 Algorithms

Algorithms used to factor integers are commonly split into two categories, special-purpose and general-purpose. Special-purpose algorithms are algorithms whose run time varies based on whether the number to be factored exhibit some special structure. Meanwhile, general-purpose algorithms are algorithms whose run time depends solely on the size of the input number. Generally, a special-purpose algorithm is faster than a general-purpose algorithm

when the number being factored fulfills its special conditions, while the reverse holds true if the number does not fulfill those conditions.

4.1 Special-purpose Algorithms

4.1.1 Trial division

The simplest algorithm for factoring integers is trial division. The idea behind trial division is that, if x is a factor of n, it must be the case that $x < n$. So, in order to find a factor of n, we check by all integers less than n until a number that divides n is found. With this in mind, the pseudocode for trial division is as follow.

```
TRIAL-DIVISION(x)
for i = 2 to x
    if i | x
        return i, x/i
```

An easy optimization can be made to this algorithm. It can be noticed that if a number is not divisible by 2, it is not going to be divisible by any even number. Thus, it is possible to eliminate half of the divisibility checks by checking if n is even at the start. If n is even, a factorization of n is found, while if n is odd, the algorithm proceeds by dividing n by all odd numbers from 3 to n. With this optimization, we obtain the algorithm

```
TRIAL-DIVISION(n)
if 2 | n
    return 2, n / 2
for i = 3 to n
    if i | n
        return i, n / i
```

The algorithm above runs in $O(1)$ in the best case, when n is even, seeing that n would then be divisible by 2, and the algorithm will halt after one divisibility check, which runs in constant time. Meanwhile, in the worst case, where n is prime, the loop will check the divisibility of n and every odd integer from 3 to n. Suppose that n is an b-bit integer. Seeing that $b \approxeq \log_2 n$, we know that $n \approxeq 2^b$. So, there are $2^b + c_1$ numbers less than n, where c_1 is a constant. Seeing that approximately half of those numbers are odd, there will be $\frac{2^b+c_1}{2} + c_2$ comparisons made in the for loop. Furthermore, an additional divisibility check was done before the for loop, and so $\frac{2^b+c_1}{2} + c_2 + 1$ comparisons are made in the worst case, which results in a time complexity of $O(2^b)$. Because $b = \log_2 n$, we can rewrite this as $O(2^{\log_2 n}) = O(2^{\log n})$, which is exponential in $\log n$.

Trial division works best when n has at least one small factor. In such a situation, the for loop will only run a small number of times. It should be noted that a small factor with respect to trial division can still be quite large.

4.1.2 Fermat's Factorization Algorithm

Fermat's algorithm proceeds as follows. Suppose that the input integer is n. Let $x = \lceil \sqrt{n} \rceil$ and $ySquared = x^2 - n$. As long as $ySquared$ is not a perfect square, increment x by 1 and set $ySquared = x^2 - n$. The pseudocode for this algorithm is

```
FERMAT-FACTORIZATION(n)
x = ceiling(sqrt(n))
ySquared = x * x - n
while not IS-PERFECT-SQUARE(ySquared)
    x = x + 1
    ySquared = x * x - n
return x - sqrt(ySquared), x + sqrt(ySquared)
```

The idea behind Fermat's algorithm is that, because $a^2 - b^2 = (a-b)(a+b)$, if an integer

n can be written as $n = x^2 - y^2$ for some integers x, y, then it is trivial to construct a factorization of n.

In the best case, the algorithm will have a time complexity of $O(1)$, seeing that $ySquared$ is a perfect square with the first x tried and so the while loop is never entered. Now, consider the worst case. Let n be an b-bit integer. We know that $b = \log_2 n$. Now, consider $\log_2 \sqrt{n}$, the number of bits in $\sqrt{n}$. We have $\log_2 \sqrt{n} = \frac{1}{2} \log_2 n = \frac{b}{2}$. With this, we know that $\sqrt{n} = 2^{\frac{b}{2}}$. Furthermore, we have shown that $n = 2^b$ in section 4.1.1. Because x goes from $\sqrt{n}$ to n, the algorithm runs in $n - \sqrt{n} = 2^b - 2^{\frac{b}{2}}$ time. We know that for all $b > 0$, $b > \frac{b}{2}$. Therefore, $2^b > 2^{\frac{b}{2}}$. So, by the definition of big-O, $O(2^{\frac{b}{2}}) = O(2^b)$. So, the Fermat's method runs in $2^b - 2^{\frac{b}{2}} = O(2^b)$ time. Substituting $b = \log_2 n$ into this expression, we get the time complexity of the algorithm to be $O(2^{\log n})$, which is exponential in $\log n$.

Fermat's algorithm works best when a number has two factors are close to each other. This is because if two factors are close, they must necessarily be close to the square root of their product, and so the first factor is near the square root of the number being factored. Seeing that the first factor starts at the square root, being near its actual value means that it only has to be incremented a small number of times, and thus the number of executions of the while loop is small. As a result, Fermat's method runs efficiently.

The reverse, however, is also true. If the two factors are far from each other, the number of times that the for loop needs to run will also be high. This is why in practice, Fermat's algorithm is not used for factoring large numbers. However, that is not to say that Fermat's algorithm does not have any utility when factoring large integers. As long as it is known that the distance between a number's two factors is sufficiently small, Fermat's algorithm is likely to be one of the best algorithms.

4.1.3 Pollard's $p-1$ Algorithm

Let n be the input integer. The first step in Pollard's $p-1$ algorithm is to pick a number B, called the smoothness bound. Then, we calculate M as the product of all $q^{\lfloor \log_q(B) \rfloor}$,

where q are primes less than B. After that, randomly pick a coprime a of n, and calculate $g = \gcd(a^M - 1, n)$. Finally, if $1 < g < n$, returns g, otherwise raises an error. The pseudocode for this algorithm is as follow

```
POLLARD-ALGORITHM(n)
B = RANDOM-NUMBER()
primesList = FIND-ALL-PRIME-LESS-THAN(B)
M = 1
for i = 1 to primesList.length
    q = primesList[i]
    M = M * q^(FLOOR(LOG_q(B)))
a = PICK-COPRIME(n)
g = GCD(a^M - 1, n)
if g > 1 and g < n
    return g
else
    error "Algorithm failed"
```

The main idea of this algorithm is derived from Fermat's Little Theorem.

Theorem (Fermat's Little Theorem)**.** *[10, Section 6.3] Let p be a prime number and a be coprime to p. Then, $a^{p-1} \equiv 1 (mod\ p)$.*

In the algorithm above, a was chosen to be coprime to n, and is consequently also coprime to all factors of n. Let p be a factor of n. Then, by Fermat's Little Theorem, we have $a^{p-1} \equiv 1 \pmod{p}$, which is equivalent to $a^{k(p-1)} \equiv 1 \pmod{p}$ for some integer k. Then, we set $M = k(p-1)$. Now, a consequence of $a^M \equiv 1 \pmod{p}$ is that $p \mid gcd(a^M - 1, n) = g$. Seeing that g is a multiple of a factor of n, if $1 < g < n$, then g must be a factor of n.

At this point, there is still one issue that needs to be addressed. In the above paragraph, M is set as a multiple of $p - 1$, which is impossible to determine without knowledge of p,

the thing that is being searched for. This is where B is used. The algorithm assumes that $p-1$ is B-powersmooth.

Definition 1. *An integer n is called k-**powersmooth** if, when n is written as a product of prime powers, all of those prime powers are at most k.*

Now, instead of calculating M as a multiple of $p-1$, the algorithm calculates M as the smallest number that is a multiple of all B-powersmooth numbers, which results in the formula for M used in the algorithm. However, this method is not guaranteed to work. If there is no factor p of n such that $p-1$ is B-powersmooth or $p-1$ is B-powersmooth for all p, the algorithm won't be able to give a nontrivial factor, seeing that the former situation results in a factor of 1, while the latter results in a factor of n.

If it is desired, however, it is possible to modify the algorithm so that a factor is always given. At the final step, instead of raising an error, consider the value of g. If $g = 1$, B should be increased. On the other hand, If $g = n$, B should be decreased. This way, B should eventually reach a value where $1 < g < n$ [1].

The time complexity of Pollard's $p-1$ algorithm is $O(B \log^2 n)$ [14]. The idea of the derivation is that there are three main steps in the algorithm: finding all primes in $[1, B]$, calculating $a^M - 1$, and computing $\gcd(a^M - 1, n)$. In most cases, where B is not small, the second step would dominate the computation, and thus its time complexity is the algorithm's complexity. Here, it should be noted that the value of M is not calculated directly. Instead, $a^{q^{\lfloor \log_q(B) \rfloor}}$ is calculated for each q, and then these values are combined to form a^M.

4.2 General-purpose Algorithms

4.2.1 Dixon's factorization method

Let n be the number to be factored. The idea behind Dixon's method is to find a congruence of squares $x^2 \equiv y^2 \pmod{n}$. It is known that if $x \not\equiv y \pmod{n}$, then $\gcd(x+y, n)$ is a proper factor of n. Furthermore, it is also known that at least half of the pairs (x, y)

that satisfies $x^2 \equiv y^2 \pmod{n}$, where xy has no common factor with n, must also satisfy $x \not\equiv y \pmod{n}$ [15]. From these two statements, it is easy to see that given a congruence of squares mod n, there is at least a 50% chance that a nontrivial factor of n can be found.

In order to find a congruence of square, Dixon's algorithm makes use of the concept of smooth numbers.

Definition 2. *An integer n is called k-**smooth** if all prime factors of n are less than k.*

The first step in Dixon's algorithm is to choose a number B, called the smoothness bound. As discussed in [6], the optimal value for B is $\exp(\frac{1}{2}\sqrt{\ln n \ln\ln n})$. After that, the list P of all prime numbers less than or equal to B is calculated. After that, for all z in the range $[\sqrt{n}, n]$, we determine if $z^2 \mod n$ is B-smooth. Here, $z^2 \mod n$ denotes the smallest positive integer x such that $z^2 \equiv x \pmod{n}$. In order to check if $z^2 \mod n$ is B-smooth, we can, for each p in P, divide $z^2 \mod n$ by p as many time as possible, then take the result and repeat the process with the next p. If this process gives a final value of 1, we know that $z^2 \mod n$ is B-smooth. In that case, we have $z^2 \mod n = \prod_{p_i \in P} p_i^{a_i}$. We will save z and $(a_1, a_2, \ldots, a_{|P|})$, called an exponent vector, as a pair of relations in the set Z.

It is not necessary, however, to check every number in $[\sqrt{n}, n]$. The theorem below gives some guidance for when to stop checking.

Theorem. *[15] If $m_1, m_2, \ldots, m_k$ are positive B-smooth integers, and if $k > \pi(B)$, where $\pi(B)$ denotes the number of primes in the interval $[1, B]$, then some non-empty subsequence of (m_i) has product a square.*

The important point to notice is that for each $z_i \in Z$, we have $z_i^2 \equiv u_i \mod n$, where u_i is B-smooth. Thus, by the theorem above, we can find a set of indices $I \subseteq [1, |P|]$ such that $\prod_{i \in I} u_i = x^2$ if $|Z| > |P| = \pi(B)$. Seeing that this would give us a congruence of square $\prod_{i \in I} z_i \equiv x^2 \pmod{n}$, we know that the earliest point we should stop checking for smooth $z^2 \mod n$ is when $|Z| = |P| + 1$. However, seeing that each congruence of square only has (at least) a 50% chance to produce a non-trivial factor of n, we would want to have a few

more relations in Z to ensure that the algorithm gives out a factor. The implementation in this project uses $|Z| = |P| + 5$.

For the sake of brevity, let $k = |P|$. Now that we have shown that it is possible to obtain a congruence of square, we will need to discuss the method by which this congruence of square can be obtained. A way to multiply two B-smooth numbers u_1, u_2 is to take the dot product of their exponent vectors. This in be seen by noticing that $u_1 u_2 = \prod_{1 \leq i \leq k} p_i^{a_{i1}} \prod_{1 \leq i \leq k} p_i^{a_{i2}} = \prod_{1 \leq i \leq k} p_i^{a_{i1}+a_{i2}}$. Furthermore, we know that for a B-smooth number x^2, its exponent vector would contain all even values. Thus, in order to obtain the set of indices I such that $\prod_{i \in I} u_i = x^2$, we find a linear combination $x_1 v_1 + x_2 v_2 + \ldots + x_k v_k = v$, where v_i is the exponent vector of u_i and v is the exponent vector of x^2. This is equivalent to solving the matrix equation

$$\begin{bmatrix} v_1 & | & v_2 & | & \ldots & | & v_k \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \end{bmatrix} = v$$

Because we are only concern with the parity of the values of v, we can solve this equation mod 2, in which case $v = \vec{0}$. In order to do this, we will use Gaussian elimination. For more details on this approach, refer to [6]. Once we have obtained $(x_1, x_2, \ldots, x_k)$, we can get $I = \{i : x_i = 1\}$. Using this set of indices, we can produce a congruence of squares and obtain a factor of n.

The time complexity of this Dixon's method is $L[\frac{1}{2}, 2\sqrt{2}]$ [7], which indicates that it has sub-exponential run time in $\log n$. It should be noted, however, that the algorithm does not guarantee a factor. As noted in the preceding paragraphs, each congruence of squares only have a 50% chance to produce a non-trivial factor of n. Even if the chance of obtaining at least one such factor of n rapidly gets close to 1, there is still a possibility that none of them is found. We can makes it so that Dixon's algorithm basically guarantees a factor by

checking every number in $[\sqrt{n}, n]$. However, this would increase the execution time by a significant amount, which is not preferable.

5 Implementation and Bench-marking

5.1 Implementation and Benchmark details

The algorithms described are implemented in the Python to leverage the native support for arbitrary-precision arithmetic. Furthermore, Python was also used to take advantage of the numpy and sympy library, which contains implementations of some important algorithms, one of which being Gaussian elimination. For testing, the native time library in Python is used since the inputs are large enough that the run time of most algorithm fits in the resolution of the time library.

For the measurement, four sets of number, formed by multiplying two primes in the range $[10^6, 10^7]$, were used as inputs. The first set was selected so that each number has at least one small prime factor. Each member of the second set was formed by picking two primes with a low difference. Each element of the third set of number is formed by choosing one prime factor p so that $p-1$ is B-powersmooth for some small B, while the other prime factor can be any number. Finally, the fourth set was formed from products of arbitrary primes.

The purpose of the splitting the test into four separate sets is to test if the special-purpose algorithms are the fastest if the requirements for their being efficient is fulfilled. Furthermore, this split also has a secondary purpose of approximating the point at which a number is not considered easy to factor for a special-purpose algorithm. In our case, the first set tests trial division, the second set tests Fermat's method, and the third set tests Pollard's algorithm. Meanwhile, the fourth set is a general test to compare the efficiency of the Dixon's method, the only general-purpose algorithm, against the three special-purpose algorithms.

5.2 Results

The result of the test is presented in table 1 below. Each cell in the table contains the average run time of each algorithm over each set of data in seconds. It can be seen that some of the cells are empty. This is due to the test corresponding to that cell takes too long to find a factor for its number set, and thus has to be halted prematurely. This was done because in testing with random numbers, some algorithms, mainly Pollard's $p-1$, can take as much as 10 minutes to produce a factor. Because it is not necessary to test an algorithm completely when its special condition does not apply (it can already be noticed that it runs much slower than every other algorithm), it is decided that it is more prudent to halt an algorithm after a period of time. The maximum amount of time an algorithm is allowed to run is chosen to be 60 seconds.

	Set 1	Set 2	Set 3	Set 4
Trial division	1.42	3.58	1.49	1.87
Fermat's method	1.98	0.08	0.45	1.05
Pollard's $p-1$ algorithm				
Dixon's method	14.34	30.03	12.60	16.74

Table 1: Test result

At a glance, it can be seen that in the first set, trial division outperforms Fermat's method and Dixon's method. For all the other sets, however, Fermat's method is faster than every other algorithms. It should also be noted that while trial division outperforms Fermat's method in the first set, the difference in run time is small. From this, it can be said that generally, Fermat's method should be better than trial division in cases where the factors have the same order.

Another thing of note is that Dixon's method is inferior to the two methods discussed above. This points to one of two scenarios, either the implementation of the algorithm can

be optimized further or the input size is not large enough for the asymptotic portion of the run time to exceed the constant part. While it is possible that the first situation is true, the problem is more likely the latter, seeing that 10^12 is not a very large number in the grand scheme of things.

One interesting result is that Pollard's $p-1$ algorithm never manages to factor n in under 60 seconds, even in the set of numbers where it should be efficient on. This indicates that there might be a problem in the implementation. After some investigation of the code, it has been determined that there are some computations that are being unnecessarily repeated. Knowing this, the implementation can be modified so that it runs faster.

6 Discussion

6.1 Current state of the art

As of the moment, the most efficient algorithm, asymptotically, for factoring any integer n is the general number field sieve, which has a time complexity of $L_n\left[\frac{1}{3}, \sqrt[3]{\frac{64}{9}}\right]$. Compared to this algorithm, all other general-purpose algorithms have a complexity of at least $L_n\left[\frac{1}{2}, 1\right]$ [3]. As a result, the general number field sieve is substantially faster than those algorithms. It is the case, however, that there are still some special-purpose algorithms that are faster than it asymptotically, an example of which will be mentioned below.

The current record in factoring a number that has no special structure is the factorization of RSA-250, a 250-digit number. This number was factored using the general number field sieve, using up 2700 core years of a 2.1GHz CPU [17]. Counting numbers that exhibit special properties, however, some much larger numbers have been factored. For instance, the number $2^{1061} - 1$ was factored in 2012 using the special number field sieve [4], a version of the general number field sieve with complexity $L\left[\frac{1}{3}, \sqrt[3]{\frac{32}{9}}\right]$ for $n = r^e - s$, where r is a small positive integer and s is an integer with small absolute value greater than 0 [12].

No classical algorithms have been found to factor an arbitrary integer in polynomial

time. Even the general number field sieve, the current fastest algorithm, still only has sub-exponential time complexity. On quantum computers, however, one such algorithm exists, called Shor's algorithm. Shor's algorithm has a complexity of $O(\log^3 n)$. It has been used to factor small numbers such as 15 [16].

6.2 Future goals and possible improvements

One idea that is worth further pursuing regarding the algorithms discussed above is the possibility to extend them for prime factorization. Prime factorization is simply the factorization of integers into a products of powers of primes. As can be noticed, all of the algorithms presented above stop after a factor is found. While this behavior is enough for applications where the integer to be factored is a product of two primes, such as in cryptanalysing RSA, there are cases where a prime factorization of a number is required. Even without the application, however, it might still be enlightening to look into this matter. All of the algorithms above halt when after finding one factor, and so it is impossible to see how they might scale when having to deal with multiple factors. By adapting the algorithms to factor an integer into prime powers, it might be possible to better understand their properties and reasons why an algorithm is better than another.

One deficiency of this project is that the implementation is not as efficient as it can theoretically be. This problem is caused by two reasons, the first of which being that the language that was used. While Python's speed is sufficient for most purposes, in situations such as this project where a lot of computation needs to be done, Python being slower than other languages is a significant problem. Furthermore, Python's data model makes it more prone to running out of memory when working with a large amount of data. One concrete example in this project is in Pollard's algorithm and Dixon's method, where a list of primes have to be calculated. The method used to accomplish this is the Sieve of Eratosthenes (for more details, refer to [9]), which needs to store a list of boolean values. In Python, a list is an array of references [8]. As a result, for each element in a list, there is an overhead of storing

the reference alongside with the value. With a large list, this overhead causes the list to take up a lot more memory than an equivalent structure in another language, such as a boolean array that stores the value directly in C++. As such, Python will run out of memory with a smaller list. One method to get around this would be to make use of numpy's array, which stores the value directly instead of references.

The other issue with the efficiency of the code written is due to the algorithms used not being the most efficient. The most obvious example of this is Pollard's $p-1$ algorithm, which timed out in all of the tests done. In section 5.2, it has been found that the implementation of this algorithm is not optimized. A more subtle example can be observed in Dixon's method. It can be noticed that the matrix used in that algorithm is a sparse matrix because the exponent vectors that form it are sparse themselves [15]. As a result, algorithms that are designed to work with sparse matrix, such as the block Lanczos algorithm, would be more appropriate.

References

[1] Pollard's $p-1$ algorithm. `https://en.wikipedia.org/wiki/Pollard%27s_p_%E2%88%92_1_algorithm`.

[2] Change of base of logarithm, 2019. `https://proofwiki.org/wiki/Change_of_Base_of_Logarithm`.

[3] Buhler, J. P., Lenstra, H. W., and Pomerance, C. Factoring integers with the number field sieve. In *The development of the number field sieve* (Berlin, Heidelberg, 1993), A. K. Lenstra and H. W. Lenstra, Eds., Springer Berlin Heidelberg, pp. 50–94.

[4] Childers, G. Factorization of a 1061-bit number by the special number field sieve. Cryptology ePrint Archive, Report 2012/444, 2012. `https://eprint.iacr.org/2012/444`.

[5] Dixon, J. D. Asymptotically fast factorization of integers. *Mathematics of Computation 36*, 153 (1981), 255–255.

[6] Filaseta, M. Implementing two general purpose factoring algorithms, 2017. https://crypto.stanford.edu/cs359c/17sp/projects/BrendonGo.pdf.

[7] Filaseta, M. Math 788: Computational number theory - lecture 18, 2018. https://people.math.sc.edu/filaseta/gradcourses/Math788Lecture18.pdf.

[8] Foundation, P. S. Design and history faq - how are lists implemented in cpython?, 2021. https://docs.python.org/3.8/faq/design.html#how-are-lists-implemented-in-cpython.

[9] Horsley, S. Koσ kinon epatoσ θ enoy σ . or, the sieve of eratosthenes. being an account of his method of finding all the prime numbers, by the rev. samuel horsley, f. r. s. *Philosophical Transactions (1683-1775) 62* (1772), 327–347.

[10] Judson, T. W. *Abstract Algebra: Theory and Applications*. ORTHOGONAL Publishing L3C, 2018.

[11] Lenstra, A. K. *L Notation*. Springer US, Boston, MA, 2011, pp. 709–710. "https://doi.org/10.1007/978-1-4419-5906-5_459".

[12] Lenstra, A. K., Lenstra, H. W., Manasse, M. S., and Pollard, J. M. The number field sieve. In *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1990), STOC '90, Association for Computing Machinery, p. 564–572.

[13] Miguel, J., 2015. https://www.quora.com/How-do-I-count-how-many-triples-a-b-c-satisfy-a-2-+-b-2-equiv-c-2-mod-n-1-leq-a-b-c-leq-n-1-a-leq-b.

[14] Neves, S., 2015. https://crypto.stackexchange.com/questions/72698/pollards-p-1-factorization-method-runtime.

[15] Pomerance, C. Smooth numbers and the quadratic sieve.

[16] Vandersypen, L. M. K., Steffen, M., Breyta, G., Yannoni, C. S., Sherwood, M. H., and Chuang, I. L. Experimental realization of shor's quantum factoring algorithm using nuclear magnetic resonance. *Nature 414*, 6866 (Dec 2001), 883–887. http://dx.doi.org/10.1038/414883a.

[17] Zimmerman, P. Factorization of rsa-250?, 2021. https://listserv.nodak.edu/cgi-bin/wa.exe?A2=NMBRTHRY;dc42ccd1.2002.