Two Integer Factorization Methods

Christopher Koch

April 22, 2014
Abstract

Integer factorization methods are algorithms that find the prime divisors of any positive integer. Besides studying trial division and Pollard’s $p - 1$ method, we discuss the analysis of algorithms, why integer factorization is generally seen as hard, and we illustrate why additional computing power is not a solution for faster integer factorization. We see that both discussed algorithms are exponential time algorithms and that Pollard’s $p - 1$ method only works fast for integers with a certain property. We make the assertion that many other methods work this way, which leads to the concept of strong primes in cryptography: primes that are not easy to factor with Pollard’s $p - 1$ method and other specialized methods like it.

Keywords: number theory, integer factorization, pollard, $p - 1$ method, trial division, algorithms
List of Theorems and Definitions

2.1 Theorem (Fundamental theorem of arithmetic) ........................................ 2
3.1 Definition (Landau notation) ................................................................. 6
3.1 Theorem (Division algorithm) ................................................................. 7
3.2 Definition (Non-trivial divisor) ............................................................... 7
3.3 Definition (Greatest common divisor) ..................................................... 7
4.1 Theorem (Prime number theorem) .......................................................... 11
4.2 Theorem (Fermat’s little theorem) ........................................................... 13

List of Algorithms

3.1 Example algorithm .................................................................................. 4
3.2 Hello World repetition algorithm ............................................................ 5
4.1 Trial Division ........................................................................................... 10
4.2 Pollard’s $p-1$ method ........................................................................... 13

List of Figures

4.1 Number of primes less than $n$ versus $n$: the prime counting function $\pi(n)$. For example, there are 550 primes less than or equal to 4000. ............................................. 11

List of Tables

4.1 Worked example of trial division for $n = 14$ ......................................... 10
1 Executive Summary

According to the fundamental theorem of arithmetic, any positive integer greater than one can be written as the multiplication of prime numbers, for example \(20 = 2 \times 2 \times 5\). While the factorization must exist, finding it is an incredibly hard problem. It could take several months or years to factor a 200-digit number, depending on the computing power available. Integer factorization methods are algorithms that find the prime factors of any given integer.

Some public-key cryptographic systems such as RSA encryption are based on the difficulty of factoring large integers into their prime divisors. If someone was to discover a method to factor integers up to 1000 digits quickly, we would be able to read most encrypted communication on the internet, including online banking sessions and confidential emails.

This report introduces an algorithm to be a procedure or list of steps which will produce some output when given some input. For example, integer factorization methods are procedures that have any positive integer greater than one as input and a list of its prime factors as output. To compare algorithms, we want to analyze how much time they take to run given the same input. This time usually depends on the power of the specific computer used to run the algorithm, though. This is why instead, we choose to analyze the number of steps that an algorithm takes. The power of any computer can be measured as the number of steps it can execute per second, so if we analyze the number of steps of an algorithm, we can find the specific run time for it on any computer. The number of steps is then usually called the runtime.

The number of steps that an algorithm takes usually depends on the input in some form. For example, an algorithm that finds the average of a list of numbers would depend on the length of that list, since it must add every element and then divide by the number of elements in the list. Integer factorization methods generally depend on the value of the given integer to be factored. Because counting the exact number of steps that an algorithm takes is extensive work, we usually find the “order” of steps that an algorithm takes, denoted in Landau notation (Big O notation).

The two integer factorization methods analyzed in this paper are trial division and Pollard’s \(p \cdot 1\) method. Trial division works by considering the natural, “human” way of trying to factor an integer: given an integer \(n\), it will try every prime number between 2 and \(n/2\) to see whether \(n\) is divisible by it. While this is a sure way to find all prime factors of \(n\), it is also a really slow. Finding how many steps it takes involves knowing how many prime numbers there are between 2 and \(n/2\): the prime number theorem gives us the “order” of prime numbers less than \(n/2\).

Given an integer \(n\) to factor, Pollard’s \(p \cdot 1\) method works by knowing that if \(p\) is a prime divisor of \(n\), then \(2^{p-1} - 1\) is a multiple of \(p\). Then, \(p\) is a common divisor of both \(2^{p-1} - 1\) and \(n\), so if we find the greatest common divisor of \(2^{p-1} - 1\) and \(n\), we have a multiple of \(p\) and thus a divisor of \(n\). Since we do not know the value of \(p - 1\) beforehand and thus could not compute \(2^{p-1} - 1\), we multiply many small prime numbers into a number \(K\). We then hope that \(K\) is a multiple of \(p - 1\) so that \(2^K - 1\) is divisible by \(p\). We see that the method only works if \(K\) is a multiple of \(p - 1\), which depends on the choice of \(K\).

This leads to the concept of strong primes: an integer that only has large prime factors would need a large value of \(K\) to be factored, at which point Pollard’s \(p \cdot 1\) method is even slower than trial division. This makes Pollard’s \(p \cdot 1\) method infeasible to use, while the trial division algorithm is already slow. There are other algorithms like Pollard’s \(p \cdot 1\) method that depend on certain properties like this; if we combine all these properties, a strong prime is a prime which does not have any of them and is thus hard to factor with most known integer factorization algorithms.
2 Introduction

Every positive integer (whole number) greater than one can be written as the multiplication of prime numbers, for example $10 = 2 \times 5$. The fundamental theorem of arithmetic details that every positive integer greater than 1 must have such a “unique prime factorization;” however, it does not specify how to find such factorization. Integer factorization methods are procedures that find a given integer’s unique prime factors.

**Theorem 2.1** (Fundamental theorem of arithmetic). Let $n$ be an integer greater than 1. Then, there exist not necessarily distinct prime numbers $p_1, p_2, \ldots, p_i$ such that

$$ n = p_1 \times p_2 \times \cdots \times p_i. $$

The fundamental theorem of arithmetic was first proven by the Greek mathematician Euclid around 300 BC in his treatise *Elements* [1].

While every positive integer greater than 1 has a prime factorization, finding it is a hard problem. The naive approach would be to take the given integer $n$ and to try to divide it by every prime between 1 and $n$. If it is divisible by a prime, then that prime is one of the prime factors of $n$. One must consider that an integer $n$ may be divisible by a prime more than once, for example $20 = 2 \times 2 \times 5$ has the prime 2 twice in its prime factorization. One must also consider that finding the prime numbers between 1 and $n$ is also a difficult problem. While simple, this approach takes a long time for large numbers, even when a computer does it: it could be hundreds or thousands of years.

Traditionally, prime numbers and all that they involve, such as integer factorization, had few applications outside of pure mathematics. Even the community of number theorists viewed colleagues that occupied themselves with integer factorization as incurably obsessive [2, p. 675]. Some results and theorems known about prime numbers were only discovered due to some mathematician’s obsession with primes; for example, Gauß conjectured the famous prime number theorem when he was 15, but did not discuss it until almost 60 years later because he thought it was not a significant result [3, p. 54].

All this changed entirely in the 1970s, when the concepts of public-key cryptography were invented through RSA [4], an encryption scheme named after its inventors Ronald Rivest, Adi Shamir, and Leonard Adleman. RSA specifically relied on the difficulty of factoring large integers into primes. In addition to that, complexity theory became popular [5]: complexity theory enabled researchers to express rigorously that using a certain method to factor a number $n$ will take at most time $f(n)$ for some function $f$. While the notation of some of complexity theory had been around since the beginning of the 20th century, it was now being applied to computational complexity: this meant that algorithms, such as integer factorization algorithms, could be compared to each other in terms of speed.

The security of RSA encryption lies in the difficulty of factoring large integers. If anyone was to suddenly be able to find prime factors in a quick way, RSA encryption would be unusable. While the mathematics behind RSA cannot be broken, the ability to factor large integers quickly would make the use of RSA infeasible. This would be fatal: RSA is the “most widely deployed public-key cryptosystem” [6]; it is for example used for secure web browsing, digital signatures, and secure e-mail communications.

To this day, a lot of research has been done in integer factorization methods. In the last 40 years, a number of methods were developed and optimized; for example, Pollard’s $\rho$ algorithm in 1975 [7] [8,
ch. 31], Pollard’s $p - 1$ algorithm in 1974 [9], or Lenstra’s elliptic curve factorization algorithm in 1987 [10]. To this day, the three best practical methods of integer factorization are the general number field sieve, the quadratic sieve, and the elliptic curve factorization algorithm. There is also Shor’s algorithm for quantum computers; however, to date no quantum computer has been built that can run Shor’s algorithm.

Contrary to what we might expect, the difficulty in finding prime factorization of integers does not actually lie in the lack of availability of computing power, but in the growth of running time that each factorization algorithm exhibits: the best-known algorithms for factorization are exponential time algorithms, which means that if we know the time to factor a $b$-digit number, then the time to factor a number with one more digit (a $b + 1$-digit number) will take 10 times as much time as the $b$-digit number. For example, if it takes 200 years to factor a 200-digit number, a 201-digit number will take 2000 years to factor. Even if a supercomputer is used to try to factor an integer, we will hit a feasible limit fairly quickly. For example, a cluster of computers was used in 2009 to factor a 232-digit number over several months [11], while it would have taken 2000 years to factor this number on a regular computer in 2009.
3 Preliminaries

In order to study different integer factorization methods, we have to establish some standards about algorithms and how their performance is measured. In addition to that, we remind the reader about some mathematical concepts that were likely taught before high school.

3.1 Algorithms and Their Analysis

Generally, an algorithm is a procedure which given some input will produce some output by following a list of steps. An algorithm is also sometimes called a procedure or a function. For example, the following algorithm takes in an integer and outputs one of two things: if that integer is even, it outputs its square value, and if it that integer is odd, it outputs its cube value. An algorithm is usually denoted by its name followed by a list of input values enclosed by parentheses and separated by commas, for example \textsc{Example}(n) as in Algorithm 3.1.

\begin{algorithm}
\caption{Example algorithm}
\begin{algorithmic}[1]
\State \textbf{Input:} an integer called \textit{n}
\State \textbf{Output:} its square \textit{n}^2 if \textit{n} is even and its cube \textit{n}^3 if \textit{n} is odd
\Procedure{Example}{\textit{n}}
\If{\textit{n} is even}
\State output \textit{n} \times \textit{n}
\ElsIf{\textit{n} is odd}
\State output \textit{n} \times \textit{n} \times \textit{n}
\EndProcedure
\end{algorithmic}
\end{algorithm}

When we consider different algorithms that accomplish the same task or result, we want to compare them in terms of the time they need to run. Traditionally, this is accomplished by counting the number of steps that an algorithm takes to produce a result in all possible cases. For example, in Algorithm 3.1, if \textit{n} is even, then the algorithm takes two steps (lines 2 and 3 are executed), while if \textit{n} is odd, the algorithm takes three steps (lines 2, 4, and 5 are executed). This does not incorporate the number of steps taken by multiplication, but that is insignificant to this example.

Traditionally, the number of steps an algorithm takes is measured with respect to the \textit{length} of its input. In the case of Algorithm 3.1, the number of steps does not depend on the input at all: the algorithm will take two or three steps in all cases. Given the specifics of any computer, one could calculate the time it takes to run an algorithm given the number of steps it takes by multiplying the number of steps with the steps per minute that the computer can handle. For this reason, the number of steps that an algorithm takes is usually called the running time of an algorithm.

For research in number theory, analyzing an algorithm in terms of its input length becomes rather lengthy. Instead, the running time is measured with respect to the \textit{value} of the input. The next algorithm will serve to show the difference between those two versions of analysis.
Algorithm 3.2 Hello World repetition algorithm

**Input:** an integer called \( n \)

**Output:** no output value; the algorithm will print something instead

1: procedure `ExampleTwo(n)`
2: \( i \leftarrow 1 \)
3: print "Hello!"
4: \( i \leftarrow i + 1 \)
5: if \( i \leq n \) then
6: go back to line 3
7: else
8: done

Algorithm 3.2 will, given an integer \( n \), print “Hello!” \( n \) times. For example, the output of `ExampleTwo(2)` (with \( n = 2 \)) would be:

Hello! Hello!

If we analyze this algorithm, we find that lines 3, 4, 5, and 6 are executed \( n \) times, while lines 2, 7, and 8 are executed once. Thus, the algorithm takes approximately \( 4n + 3 \) steps. As said previously, an algorithm is traditionally analyzed in terms of the length of its input, though: \( n \) is the input value. The length of \( n \) in binary is \( k = \lfloor \log_2(n) \rfloor \), and then \( n \) can be approximated by \( 2^k \). This means the algorithm actually executes approximately \( 4n + 3 \approx (4)2^k + 3 \) steps, where \( n \) is an arbitrary input value and \( k \) is \( n \)'s length. These are two completely different formulas: therefore, it makes a difference whether we choose to analyze an algorithm with respect to its input length or its input value.

Because a lot of algorithms in number theory are related to the value of some input, we choose to analyze algorithms with respect to input value instead of input length in this paper.
3.2 Algorithmic Complexity

As we just noticed, counting the exact number of steps of an algorithm is a non-trivial problem, as simple as it may sound. That is why the running time for most algorithms will not be denoted by the exact number of steps, but by the “order” of steps. For example, Algorithm 3.2 would be considered of order $n$ because the “largest” term in $4n + 3$ involves $n$.

When we say “largest,” we mean that $n^2$ is the fastest-growing term in $n^2 + 100n + 3$; i.e. as $n$ grows larger, the term $n^2$ “contributes” the higher number to the result of $n^2 + 100n + 3$. For example, for $n = 1000$, we have that $n^2 + 100n + 3 = 1100003$, of which $n^2$ “contributed” 1000000, 100n contributed 100000, and 3 only contributed 3.

Formally, the order, also called time complexity, of an algorithm is denoted by Big O notation, also called Landau notation.

**Definition 3.1** (Landau notation). We say that a function $f(n)$ is $O(g(n))$ as $n$ goes to $\infty$ (infinity) if

$$
\lim_{n \to \infty} \left| \frac{f(n)}{g(n)} \right| < \infty.
$$

We write $f(n) \in O(g(n))$. Often, the notation of $f(n) = O(g(n))$ is also used.

Equivalently, we may say that $f(n)$ is $O(g(n))$ as $n$ goes to $\infty$ if there exist integers $c, n_0$ ("witnesses") such that $f(n) \leq c \times g(n)$ whenever $n \geq n_0$ [8, ch. 3.1].

Intuitively, we take $f(n)$ to be the actual number of steps that an algorithm runs, and $g(n)$ to be the order of the algorithm. If a function $f(n)$ is $O(g(n))$, then $g(n)$ is an upper bound on $f(n)$. In theoretical computer science, we always consider Big O notation as $n$ approaches infinity, while in mathematics $n$ could also approach any other constant.

For example, $n^2 + 100n + 3 \in O(n^2)$. But also, $n^2 + 100n + 3 \in O(n^3)$. Since Big O notation serves as an upper bound, $n^2 + 100n + 3$ is $O(n^3)$ and $O(n^4)$ as well as $O(n^2)$, but $O(n^2)$ is a tighter bound: the lowest one of all that are possible. In general, the upper bound tends to be the $n$-term with highest exponent independent of the coefficients; for example, $n^3 + 100000n^2 \in O(n^3)$.

Instead of saying that the number of steps of an algorithm are $O(f)$, where $f$ is some function, we will simply say that the algorithm is $O(f)$ and drop the phrase “number of steps.” This is common in computer science.

Landau notation allows us to compare algorithms in terms of their speed. The order of an algorithm tells us how the running time grows when the input grows. In general, the following is an ordering of algorithmic complexity of some example functions:

$$
\log n < \sqrt{n} < n < \log \log n < n \log n < n^2 < n^3 < e^n < n!
$$

That is, if we are comparing an algorithm that is $O(\log n)$ and an algorithm that is $O(n)$, we can usually say that the $O(\log n)$ algorithm performs better (faster) than the $O(n)$ algorithm, at least for large values of $n$.

An algorithm that is of order $O(n^k)$ for any real number $k$ is said to be a polynomial time algorithm. An algorithm that is of order $O(k^n)$ for some real number $k$ is said to be an exponential time algorithm. Exponential time algorithms are generally slower than polynomial time algorithms. For example, $O(n^3)$ is an polynomial time algorithm, while $O(2^n)$ or $O(e^n)$ are exponential time algorithms, where $e$ is Euler’s number ($e \approx 2.71828$).
3.3 Concepts of Division

For some of the division concepts in this paper, we need to go back to the way that division was likely taught in elementary schools: with quotient and remainder. While not an algorithm in the traditional sense; formally, the following theorem is called the division algorithm.

**Theorem 3.1** (Division algorithm). Let $a$ be an integer called the dividend. Let $d$ be an integer, $d > 0$, called the divisor. Then, there exist unique integers $q$, called quotient, and $r$, called remainder, such that

$$a = qd + r \text{ where } 0 \leq r < d.$$ 

Essentially, the division algorithm states that when an integer is divided by another integer, we have a unique quotient and remainder. For example, 21 when divided by 5 has a quotient of 4 and a remainder of 1 since

$$21 = 4 \times 5 + 1.$$ 

Sometimes, the remainder is called the modulus and regarded as the fifth basic mathematical operation next to addition, subtraction, division, and multiplication. Since the remainder of 21 divided by 5 is 1, we write that

$$21 \mod 5 = 1.$$ 

Please note that this is not exactly standard notation and as such is only used in this way in this paper.

**Definition 3.2** (Non-trivial divisor). Let $n$ be an integer and $d$ be a divisor of $n$; that is, $n$ is a multiple of $d$. Then, $d$ is called a non-trivial divisor if $d \neq 1$ and $d \neq n$.

For example, 2 is a non-trivial divisor of 6, since it is neither 1 or 6.

**Definition 3.3** (Greatest common divisor). Let $b$ and $c$ be integers. An integer $d$ is the greatest common divisor of $b$ and $c$ denoted $d = \gcd(b, c)$ if

(a) $d$ is a common divisor of $b$ and $c$; i.e. $b$ is a multiple of $d$ and $c$ is a multiple of $d$. ($d$ is a common divisor.)

(b) For every common divisor $e$ of $b$ and $c$, $d \geq e$. ($d$ is greater than every other common divisor.)

As the name states, the greatest common divisor of two integers is the largest number that is a divisor of both integers. For example, $\gcd(40, 6) = 2$, since the divisors of 40 are 1, 2, 4, 5, 8, 10, 20, and 40, while the divisors of 6 are 1, 2, 3, and 6. The largest one they have in common is 2, so it is their greatest common divisor.

If we are given a number $n$ that we want to find prime divisors of and we are given $d = \gcd(a, n)$ where $a$ is some arbitrary number, then we know that $d$ is a divisor of $n$. We have that $d$ may not be prime, but if $d \neq 1$ and $d \neq n$, then $d$ is a non-trivial divisor of $n$, which is what a lot of integer factorization algorithms actually look for.
3.4 The Cost of Multiplication and GCD

Most integer factorization algorithms perform a lot of multiplications and the time cost of them cannot be ignored. There are many multiplication algorithms, for example the traditional schoolbook long multiplication that everyone learns in elementary school, which is of order $O(n^2)$ for two numbers with $n$ digits. The time taken to multiply two $n$-digit numbers will be denoted by $M(n)$ in this paper.

While not the fastest known algorithm in terms of algorithmic order, the multiplication algorithm by Strassen and Schönhage is the most used multiplication method in practice and its runtime $M(n)$ is of complexity $O(n \log n \log \log n)$ [12]. The method by Fürer with better algorithmic complexity has so far only theoretical implications and has not been used anywhere [13]; it is in practice only faster for very very large integers.

Many integer factorization algorithms also make use of the GCD. The simplest GCD algorithm is the Euclidean algorithm which takes $O((\log n)^2)$ time for two integers $n$ of the same length. A faster GCD algorithm would be one by Schönhage in $O(\log n (\log \log n)^3 \log \log \log n)$ time [14].
4 Integer Factorization

Given an integer \( n \), many integer factorization methods will only find one divisor of \( n \) and it may not be a prime. It is then necessary to test whether the resulting divisor is a prime or not. To find more than one factor of \( n \), the respective algorithm would have to be used repeatedly. In this paper, only the trial division algorithm will give all prime divisors of \( n \).

4.1 Why is Integer Factorization Hard?

Integer factorization is traditionally seen as a hard problem by computer scientists and mathematicians [2, p. 676]. In this context, “hard” means that the problem is computationally intensive: given a really large integer, a computer will take several hundred or thousand years to factor it. The difficulty of integer factorization does not lie in the lack of computing power, but in the lack of a fast algorithm.

Formally, it is said that there is no known \textit{polynomial time} algorithm for integer factorization, where we consider running time in terms of input length. The best known algorithms for integer factorization for some integer with \( b \) digits are of order \( O(e^b) \), which is exponential time.

Intuitively, this makes sense: it means that a \( b \)-digit number \( n \) can be factored in approximately \( e^b \) time, where \( b = \lceil \log_2(n) \rceil \), so \( e^b \) is proportional to \( n \). In essence, this means that the time taken to factor a number \( n \) is proportional to \( n \) itself.

To illustrate, this is what it means that all known integer factorization algorithms are exponential time algorithms: if a 200-digit number takes 200 years to factor on one computer, then a 201-digit number will take 10 times as much time: 2000 years. Similarly, a 202-digit would take 20000 years. Of course, the factor might not be 10, but any multiplication factor will produce a cascading effect like this making it infeasible to factor large numbers.

If we had ten times the computing power, a 200-digit number would take 20 years, while a 201-digit number would take 200 years. We see that adding more computing power does not solve the underlying problem of slow integer factorization algorithms, it simply delays the inevitable infeasibility of factorization to numbers with more digits.

The attentive reader may have noticed that almost no definite statements were made about the algorithmic running time that is possible for integer factorization. It was only said that no known polynomial time algorithms exist that solve the factorization problem. This is because formally, it has not been proven that no such algorithm exists [8, ch. 34].
4.2 Trial Division

Trial division is the naïve method of integer factorization. Given an integer \( n > 1 \), we know that factors of \( n \) can be between 2 and \( n/2 \): for example, \( 14 = 7 \times 2 \) has both 2 and \( 14/2 = 7 \) as prime factors. Then, we simply test every prime number between 2 and \( n/2 \) inclusive to see whether it divides \( n \). Algorithm 4.1 details the steps to be taken for the trial division algorithm.

Algorithm 4.1 Trial Division

Input: an integer called \( n \) to be factored into primes
Output: list of prime factors of \( n \)

1: procedure TrialDivision\((n)\)
2: \( L \leftarrow \text{PRIMES}(n/2) \) \( \triangleright \) \( L \) contains a list of primes between 2 and \( n/2 \)
3: \( P \leftarrow () \) \( \triangleright \) \( P \) is the list of prime factors, initially empty
4: \( p \leftarrow \text{first prime in list } L \) \( \triangleright \) \( a \) is the first prime number in the list \( L \)
5: if \( n \) is divisible by \( p \) then \( \triangleright \) if the remainder of \( n/p \) is 0
6: Add \( p \) to the list \( P \) \( \triangleright p \) is a prime factor of \( n \)
7: if \( L \) has more primes then \( \triangleright \) Repeat for the next prime
8: \( p \leftarrow \text{next prime in list } L \)
9: Go back to line 5
10: output list \( P \) \( \triangleright \) \( P \) is a list of prime factors of \( n \)

When finished, \( P \) is a list that contains all prime factors of the integer \( n \). Note that this is not the ideal version of trial division; one can make a few improvements to it that also improve its running time.

4.2.1 Worked Example

We follow the steps of the algorithm, going from line to line:

<table>
<thead>
<tr>
<th>Line</th>
<th>( L )</th>
<th>( p )</th>
<th>( P )</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>( (2, 3, 5, 7) )</td>
<td>2</td>
<td>()</td>
<td>At line 4 before we start</td>
</tr>
<tr>
<td>5/6</td>
<td>( (2, 3, 5, 7) )</td>
<td>2</td>
<td>(2)</td>
<td>14 is divisible by 2, add to list of divisors ( P )</td>
</tr>
<tr>
<td>7/8</td>
<td>( (2, 3, 5, 7) )</td>
<td>3</td>
<td>(2)</td>
<td>Consider next prime in list of primes ( L ): 3</td>
</tr>
<tr>
<td>5</td>
<td>( (2, 3, 5, 7) )</td>
<td>3</td>
<td>(2)</td>
<td>14 is not divisible by 3, so do not add to list of divisors ( P )</td>
</tr>
<tr>
<td>7/8</td>
<td>( (2, 3, 5, 7) )</td>
<td>5</td>
<td>(2)</td>
<td>Consider next prime in list of primes ( L ): 5</td>
</tr>
<tr>
<td>5</td>
<td>( (2, 3, 5, 7) )</td>
<td>5</td>
<td>(2)</td>
<td>14 is not divisible by 5, so do not add to list of divisors ( P )</td>
</tr>
<tr>
<td>7/8</td>
<td>( (2, 3, 5, 7) )</td>
<td>7</td>
<td>(2)</td>
<td>Consider next prime in list of primes ( L ): 7</td>
</tr>
<tr>
<td>5/6</td>
<td>( (2, 3, 5, 7) )</td>
<td>7</td>
<td>(2, 7)</td>
<td>14 is divisible by 7, so add to list of divisors ( P )</td>
</tr>
</tbody>
</table>

Table 4.1: Worked example of trial division for \( n = 14 \)

Then, at the end, we have that \( 14 = 7 \times 2 \) since the list \( P \) contains 2 and 7.
4.2.2 Analysis

Line 2 is considered to not take any time, because it is usually assumed that a list of primes is available for use. It is cheap to find a list of primes once and keep it in storage. It is then the question how often lines 5 through 9 repeat.

Lines 5 through 9 repeat for as many times as there are primes in the list \( L \). In essence, we need to find the number of primes between 2 and \( n/2 \). This is where the following theorem comes into play.

**Theorem 4.1** (Prime number theorem). Let \( \pi(x) \) be the number of primes less than \( x \); it is called the prime-counting function. Then,

\[
\lim_{x \to \infty} \frac{\pi(x)}{x/\ln(x)} = 1.
\]

This means that \( \pi(x) \in O \left( \frac{x}{\ln(x)} \right) \) by the definition of Landau notation (Definition 3.1).

For example, \( \pi(10) = 4 \), since the primes less than or equal to 10 are 2, 3, 5, and 7. Thus, there are four primes less than 10.

---

**Figure 4.1**: Number of primes less than \( n \) versus \( n \): the prime counting function \( \pi(n) \). For example, there are 550 primes less than or equal to 4000.

Figure 4.1 shows how close the values of \( \pi(n) \) and \( \frac{n}{\ln(n)} \) lie together. The prime number theorem was conjectured by Carl Friedrich Gauß in the year 1792 [3, p. 54], but it was not proved until 1896 by Jacques Hadamard and Charles Jean de la Vallée-Poussin. In fact, Gauß did not reveal his
conjecture until almost 60 years later in a letter, because he did not think his discovery was very important [3, p. 54].

This means that the number of primes lower than \( n/2 \) is of order \( O\left( \frac{n/2}{\ln(n/2)} \right) \). Therefore, the steps in lines 5-9 in the trial division algorithm are repeated \( O\left( \frac{n/2}{\ln(n/2)} \right) \) times.

Each time that those lines are repeated, we test the divisibility of two numbers, add a number to a list, test whether a list has more numbers, and read the next number in a list. These are four different steps, but we may neglect the time cost of three of them because it is relatively small. Only testing divisibility takes a lot of time, namely \( M(\log n) \).

Then, we can multiply the number of times that lines 5-9 repeat by the time each repetition takes and obtain the running time of the trial division algorithm presented here to be

\[
O\left( \frac{n/2}{\ln(n/2)} M(\log n) \right) = O\left( \frac{n}{\ln(n)} M(\log n) \right).
\]
4.3 Pollard’s $p - 1$ Method

We remember that the greatest common divisor (gcd) of two numbers $a$ and $b$ is the largest integer that divides both $a$ and $b$; see Definition 3.3 on page 7. Pollard’s $p - 1$ method is based on a famous theorem by Pierre de Fermat called Fermat’s little theorem [9, 10]. As with most of his number theory conjectures, Fermat did not actually prove this theorem [15]. Many years after its conjecture, Leonhard Euler proved it in 1736 and generalized it to a theorem known as the Euler totient theorem today [15].

**Theorem 4.2** (Fermat’s little theorem). Let $p$ be a prime number. Then, for any integer $a$, we have that $a^{p-1} - 1$ is a multiple of $p$. That is, $a^{p-1} - 1$ is divisible by $p$.

While it may not be intuitive at first, the theorem also implies that for some prime $p$ and any number $K$, we have that $a^{K(p-1)} - 1$ is a multiple of $p$.

The idea of Pollard’s $p - 1$ method is that if $p$ is a prime, and $K$ is some multiple of $p - 1$, then $p \leq \text{gcd}(a^{K} - 1, n) \leq n$, since $p$ divides both $a^{K} - 1$ and $n$. We then select $K$ as the multiplication of many small primes and calculate the $\text{gcd}(a^{K} - 1, n)$ in the hope that $K$ is some multiple of a prime factor of $n$ minus 1.

**Algorithm 4.2** Pollard’s $p - 1$ method

**Input:** an integer called $n$ and a smoothness bound $B$

**Output:** a non-trivial divisor of $n$ or failure

1: **procedure** POLLARDP-1($n, B$)
2: $K \leftarrow$ product of primes $p \leq B$ to the power $\left\lfloor \log_p(n) \right\rfloor$:
3: \[ K = \prod_{\text{primes } p \leq B} p^{\left\lfloor \log_p(n) \right\rfloor}. \]
4: $K \leftarrow (2^K - 1) \mod n$ \hspace{1cm} ▷ remainder of $2^K - 1$ divided by $n$
5: $g \leftarrow \text{gcd}(K, n)$ \hspace{1cm} ▷ greatest common divisor
6: if $g = 1$ then
7: \hspace{1cm} either select larger $B$ and go to line 2
8: \hspace{1cm} or output failure
9: else
10: output $g$ \hspace{1cm} ▷ $g$ must be a divisor of $n$

We notice that to find a prime divisor $p$ of $n$, the number $p - 1$ must be a multiple of $K$. For the algorithm to run quickly, we want $K$ to be a small number, but to always find all prime factors of $n$, we need $K$ to be $\sqrt{n}$. The algorithm then only works fast for numbers $n$ where a prime divisor $p$’s neighbor $p - 1$ is divisible by small primes. Since we cannot predict whether any number $n$ has this property, we cannot say whether it is feasible to use this algorithm or not for that number.

This introduces the concept of **strong primes**: in certain cryptographic schemes such as RSA, we need to use an integer that is hard to factor. If we consider all known integer factorization algorithms and their properties, then we can designate a strong prime to be a prime number that exhibits none of the properties that the factorization algorithms need to work well. For the $p - 1$ algorithm that means that a strong prime is a prime whose prime factor $p$’s neighbor $p - 1$ has at least one large prime factor, so that $K$ has to be a big number and the $p - 1$ algorithm runs slowly.
It is to note that Pollard’s $p - 1$ algorithm only finds one divisor of $n$ and this divisor is not necessarily prime. It may either be that the divisor is prime or it needs to be factored further. To find all factors of $n$, the algorithm needs to be used repeatedly.

4.3.1 Worked Example

We want to factor $n = 540,143$ and we choose $B = 8$.

Since the computation of $K$ is extensive, for the example we choose a similar measure: $K = \text{lcm}(2, 3, \ldots, B)$, the lowest common multiple of the numbers between 2 and $B$. For example, the lowest common multiple of 2, 3, and 4 is 12, because 12 is a multiple of all 2, 3, and 4 and it is the lowest such number.

Then, we have

$$K = \text{lcm}(2, 3, 4, 5, 6, 7, 8) = 840.$$  

We also have

$$(2^K - 1) \mod 540143 = (2^{840} - 1) \mod 540143 = 53046.$$  

To find a divisor, we compute

$$d = \gcd(53046, 540143) = 421.$$  

Indeed, we have that $540143 = 421 \times 1283$. Just as noted above, we notice that this only worked because $421 - 1 = 420$ was the factorization of many small primes: $420 = 2 \times 2 \times 3 \times 5 \times 7$. All these factors were also factors of $K$ and this is why the method worked.

If we now tried to factor $491389$ into its prime factors $383$ and $1283$, we notice that $383 - 1 = 382$ is not the factorization of many small prime factors, since $382 = 2 \times 191$. Then, we would have to choose at least $B = 191$ for the method to work. At this point, the computation of finding $K$ becomes really expensive.

4.3.2 Analysis

The running time of Pollard’s $p - 1$ algorithm depends entirely on the exponentiation in line 2/3 and the calculation of $2^K - 1$ in line 4. The GCD calculation on line 5 is not free, but it is negligible in comparison.

An exact analysis requires another algorithm that merges the computation of lines 2-4 and can be found in [16, p. 17/18], but it requires the knowledge of some more algorithmic notation. The analysis in [16,17] concludes that the running time of one iteration of Pollard’s $p - 1$ algorithm is

$$O(\pi(B) \log(n) M(\log(n))) = O\left(\frac{B}{\log(B)} \log(n) M(\log(n))\right).$$

Here, $\pi(B)$ refers to the prime-counting function as in Theorem 4.1.

Of course, this does not take into account the choice of $B$ and whether the algorithm will find a divisor or not. This only depends on the prime factors itself, which we cannot know beforehand.
5 Methods

The research conducted for this paper did not contain any novel results. The entire paper is based on internet and library research as well as some standard textbooks in number theory and theoretical computer science such as [2, 8].

For verification, the material was presented to a seminar class on algorithms under supervision of Dr. Subhasish Mazumdar, associate professor of Computer Science in the Department of Computer Science and Engineering at the New Mexico Institute of Mining and Technology.
6 Conclusion

In this paper, we introduced two different integer factorization methods. Each of the methods does not run in a feasible time for integers with a lot of digits. In addition to that, Pollard’s $p - 1$ method only works fast if the given integer has some certain properties that one cannot predict before using the method. In the worst case, Pollard’s $p - 1$ algorithm will take more time to find just one divisor than trial division will take to find all prime divisors.

While there are better integer factorization algorithms than the two presented here, we have seen that integer factorization algorithms in general are slow and not feasible for large numbers. While better algorithms require extensive knowledge of abstract algebra and number theory, their running time is still exponential time and will eventually hit the same boundaries as trial division and Pollard’s $p - 1$ algorithm.

Some cryptographic schemes take advantage of these difficulties: the difficulty of integer factorization is what makes some cryptographic schemes safe.

Many other integer factorization methods work similarly to Pollard’s $p - 1$ method in that they tend to only work relatively fast if the given integer exhibits a certain property that we cannot predict beforehand. This is also advantageous to cryptography: we can introduce the concept of strong primes. A strong prime is a prime number which does not exhibit any of the properties needed to factor it relatively quickly using any known integer factorization algorithm. These strong primes are then the primes used in cryptographic schemes that depend on the difficulty of integer factorization.
7 References


