An Efficient Variant of Pollard’s $p − 1$ for the Case That All Prime Factors of the $p − 1$ in B-Smooth

Kritsanapong Somsuk

Department of Computer and Communication Engineering, Faculty of Technology, Udon Thani Rajabhat University, UDRU, Udon Thani 41000, Thailand; kritsanapong@udru.ac.th

Abstract: Due to the computational limitations at present, there is no efficient integer factorization algorithm that can break at least 2048 bits of RSA with strong prime factors in polynomial time. Although Shor’s algorithm based on a quantum computer has been presented, the quantum computer is still in its early stages of the development. As a result, the integer factorization problem (IFP) is a technique that is still being refined. Pollard’s $p − 1$ is an integer factorization algorithm based on all prime factors of $p − 1$ or $q − 1$, where $p$ and $q$ are two distinct prime factors of modulus. In fact, Pollard’s $p − 1$ is an efficient method when all prime factors of $p − 1$ or $q − 1$ are small. The aim of this paper is to propose a variant of Pollard’s $p − 1$ in order to decrease the computation time. In general, the proposed method is very efficient when all prime factors of $p − 1$ or $q − 1$ are the members of B-smooth. Assuming this condition exists, the experimental results demonstrate that the proposed method is approximately 80 to 90 percent faster than Pollard’s $p − 1$. Furthermore, the proposed technique is still faster than Pollard’s $p − 1$ for some values of modulus in which at least one integer is a prime factor of $p − 1$ or $q − 1$ while it is not a member of B-smooth. In addition, it is demonstrated that the proposed method’s best-case running time is $O(x)$, where $x$ is represented as bits length of $\sqrt{n}$.

Keywords: RSA; integer factorization; pollard’s $p − 1$; modular exponentiation

1. Introduction

Research and development continue since we are on the verge of approaching the quantum computer era. Indeed, if the quantum computer is fully functional, almost asymmetric key cryptography algorithms will be rendered obsolete, except post-quantum cryptography, which is an active field. Furthermore, the world’s first completely quantum computer is expected to be released in 2030 [1]. However, with the incomplete quantum computer, asymmetric key cryptography or public key cryptography [2] remains one of the most effective methods for protecting the secret information transmitted via an insecure channel. RSA [3] is the most widely used public key cryptography. For encryption and decryption, this algorithm employs a pair of different keys. The first key, $e$, is the public key and must be shared with everyone. The other key which is mathematically related to the public key and is kept secret by the owner is the private key, $d$. Assuming the modulus $n = p \times q$ and Euler’s totient function $\Phi(n) = (p − 1) \times (q − 1)$, where $p$ and $q$ are two distinct prime factors. The key generation procedure is to generate $e$ and $d$. In fact, $e$ which must be assigned in the following conditions: $3 ≤ e < \Phi(n)$ and $\gcd(e, \Phi(n)) = 1$ can be chosen independently by the owner. After $e$ is found, $d \equiv e^{-1} \pmod{\Phi(n)}$ can be computed. Assuming senders want to send the secret message, $m$, also known as the plaintext to receivers, $m$ should be encrypted as $c \equiv m^e \pmod{n}$, where $c$ is the ciphertext. On the other hand, receivers who have $d$ can recover $m$ by using decryption equation as $m \equiv c^d \pmod{n}$.

To be deemed secure against intruders, $n$ should be allocated at least 2048 bits. The most efficient integer factorization approach in a classical computer, such as number field sieve, an algorithm with sub-exponent computation time complexity, is currently infeasible
to crack RSA in polynomial time without a completely quantum computer. In addition, RSA is based on the hardness of factoring $n$ \cite{4,5}. If $n$ is factored by using some integer factorization algorithms, RSA is broken. However, no effective technique exists and can break 2048 bits of RSA in polynomial time at present. As a result, the factorization problem is still evolving.

All integer factorization algorithms are now split into two classes. The first group is known as the special-proposal factoring group. The efficiency of each algorithm is determined by the various weak points of $p$ or $q$. Pollard’s $p - 1$ \cite{6}, which was proposed by Pollard, J. in 1974, is one of the algorithms in this group. In fact, this algorithm is very efficient whenever all prime factors of $p - 1$ or $q - 1$ are small. The second group is called the general-proposal factoring group. The computation time to finish the process by using all algorithms in this group is based on the size of $n$.

The goal of this study is to present a variant of Pollard’s $p - 1$ that is particularly efficient when all prime elements of $p - 1$ or $q - 1$ belong to $B$ (or also called $B$-smooth).

Assuming $P = \left( \prod_{p \in B} p \right)$, the main idea behind the proposed method is to find $r = \gcd(a^P - 1, n)$, where $1 < r < n$. In fact, $i$ is always less than bit length of $\sqrt{n}$ when all prime factors of $p - 1$ are members of $B$ and it is extremely likely that the proposed method will be faster than Pollard’s $p - 1$. Furthermore, the experimental results demonstrate that although there is at least one number which is a prime factor of $p - 1$ or $q - 1$ while it is not a member of $B$, this method may be faster than Pollard’s $p - 1$.

2. Overviews of Integer Factorization Algorithms

RSA is based on the integer factorization problem because the private key which must be kept secret, can be easily recovered after prime factors of the modulus are found. There is currently no efficient factorization algorithm that can break at least 2048 bits of RSA in polynomial time. As a result, RSA is still commonly employed to protect the secret data transferred over an insecure channel. Integer factorization techniques, on the other hand, are still being researched in an attempt to crack RSA. Factorization algorithms come in a variety of forms, as seen below:

The special-proposal factoring group is the group of numerous integer factorization algorithms. The performance of each algorithm in this group is based on the weak point of prime factors. Some of algorithms in this group are listed below.

1. The simplest method in this group is the trial division algorithm (TDA). The lowest or largest integer, which may or may not be a prime factor, is chosen as the initial divisor. This integer is almost probably the prime factor if there is not a remainder. However, in order to achieve the right answer, the divisor must be adjusted. In fact, TDA is divided into two techniques. This first technique \cite{7} is to select $x = 3, x \in \mathbb{Z}^+$, as the initial divisor for the process. If $x$ is not a divisor, it needs to be increased until the target is discovered. A small prime factor is the weak point of this case. The second technique \cite{8} is to start with the maximum integer, $x = \lfloor \sqrt{n} \rfloor$, which may be a prime factor. It differs from the first strategy in that $x$ must be reduced when it is not a factor of $n$. As a result, the case’s weakness is that a prime factor is relatively close to $\lfloor \sqrt{n} \rfloor$.

2. Fermat’s factorization algorithm (FFA) \cite{9} is a special algorithm proposed by Fermat, P. in 1600. At that time, he found that $n$ can be rewritten as the other form stated in the Equation (1).

$$n = \left( \frac{p + q}{2} \right)^2 - \left( \frac{p - q}{2} \right)^2$$

The solution to Fermat’s problem is to discover two perfect square integers in the Equation (1). Furthermore, several techniques have been developed to find both integers. In 2014, Wu, W.E. et al. presented the new initial values of $p + q$ and $p - q$ that are both closer to the targets when they are compared to the traditional initial values. However, this method, which is called the estimated prime factor (EPF) \cite{10}, cannot be chosen to solve
n derived from the balanced primes. In 2021, EPF was improved [11] to solve n which is generated from unbalanced primes or balanced primes. Therefore, the modified method can be chosen to recover p and q without the mistake. In 2006, Omar, K. and Szalay, L. proposed the improvement of FFA [12]. The key is that if the integer, r, which is in the condition is found, then n can be factored in at most \(2^L\) comparisons, where \(L \in \mathbb{Z}^+\). In addition, the specific Fermat’s factorization algorithm considered from X (SFFA-X) [13] where X is represented as the last m digits, \(m \in \mathbb{Z}^+\), of n was proposed to decrease steps to find p + q and p − q. The main process is to consider the last m digits of n to choose only integers which may be p + q or p − q. Furthermore, the technique [14] to estimate the new initial value of p − q which is begun as 2\(\left\lceil \sqrt{\frac{d^2}{4} \times n} \right\rceil\) where d is the distance between the initial value to find p + q and the new initial value to p + q was proposed to reduce steps to find p − q.

The idea behind these modified techniques is to eliminate computation loops. In fact, the weak point that the algorithms based on FFA can factor n very fast is a small result from \(|p − q|\).

3. Generalized Trial Division [15] is another factoring algorithm in the special proposed-factoring group. It was proposed by Sahin, M. in 2011. The integers \(a = b = x = \lfloor \sqrt{n} \rfloor\) are used as the starting point for computing \(r = \gcd(x, n)\). If \(r\) belongs to the scope, \(1 < r < n\), then it is a prime factor of n. On the other hand, \(a = a + 1\) and \(b = b - 1\) are assigned to find \(\gcd(a, n)\) and \(\gcd(b, n)\) until the result in the condition is found. Assuming \(i \times p\) or \(j \times q\) is very close to \(\lfloor \sqrt{n} \rfloor\), where \(i, j \in \mathbb{Z}^+\), they are the weak point which is suitable for generalized trial division.

4. VFactor [16] is a special factoring algorithm proposed in 2012. Assuming y is a maximum odd integer which is less than \(\lfloor \sqrt{n} \rfloor\) and x is a minimum odd integer which is larger than \(\lfloor \sqrt{n} \rfloor\). The idea behind VFactor is to think about the consequence of \(m = x \times y\). If m is equal to n, x and y are two prime factors of n. On the other hand, the result is divided into two cases. In the first situation, if m is larger than n, y must be reduced by two in order to compute \(m = x \times y\) once more. When m is smaller than n, x is raised by two to compute \(m = x \times y\) once again. It is similar to FFA that the small result of \(|p − q|\) is the weak point for this algorithm.

5. Pollard’s Rho or Monte Carlo factorization [17] was proposed by Pollard, J. in 1975. Moreover, its variants [18] were also proposed by Richard, B. in 1980. The key is to find \(m_1 = m_1^2 - 1 (\mod n)\) and \(m_2 = m_2^2 - 1 (\mod n)\) that \(\gcd(m_2 - m_1, n) \neq 1\). In addition, Pollard’s Rho and its variants are highly efficient when prime factors are small.

The general-propose factoring group is the group of integer factorization algorithms for the general case of n. The algorithms in this category depend on the size of n; prime factors are not a weak point. The following examples are the algorithms in this category:

1. Lenstra elliptic-curve factorization or elliptic curve factorization (ECM) [19,20] is the factoring algorithm which is modified from elliptic curve cryptography (ECC) [21,22]. To determine a prime factor, ECM employs the elliptic curve equation modulo n. The objective is to continuously discover new points on the curve until they cannot be calculated. In fact, \(\gcd(t, n) = 1\), is the only condition that allows the new point to be calculated from the addition of two points while t is the relation between two points for creating a new point. As a result, it cannot be computed when \(\gcd(t, n) \neq 1\) However, the result becomes a prime factor of n.

2. Quadratic sieve [23] is an efficient method for the medium size of n; it is a very efficient method when the digits of n are smaller than 100. To find two prime factors, two integers, X and Y, with the conditions in (2) and (3) must be found.

\[
X^2 \equiv Y^2 (\mod n) \quad (2)
\]
And
\[ X \not\equiv Y \pmod{n} \rightarrow \text{or } X \not\equiv -Y \pmod{n} \] (3)

In fact, a prime factor is calculated from \( p = \gcd(X - Y, n) \) after both of them are found.

3. Number field sieve (NFS) [24] is known as the most efficient factorization algorithm at present. In fact, NFS is an algorithm with sub-exponential computation time complexity. Moreover, NFS can factor \( n \) that is larger than \( 10^{100} \) rapidly in some cases.

Shor’s algorithm [25] is the factorization algorithm presented by Shor, P. in 1994. This method differs from the others because it is based on a quantum computer, whereas the others are based on a classical computer. In that time, he showed that his algorithm could break the large size of RSA in polynomial time when a quantum computer is practical. As quantum computers are still being developed, RSA is still frequently used to handle secret information until an efficient quantum computer is fully deployed.

3. Smooth Number

Smooth number is the group containing all prime numbers which are less than the smooth number. In fact, it is an important part for many factorization algorithms such as QS and Pollard’s \( p - 1 \). Assuming \( B \) is represented as the smooth number, which is also called \( B \)-Smooth, all prime numbers which are less than \( B \) are the members of \( B \).

4. Overviews of Pollard’s \( p - 1 \)

Pollard’s \( p - 1 \) [6] is a factorization algorithm in the special proposed-factoring group. This method was proposed by Pollard, J. in 1974. In fact, Pollard’s \( p - 1 \) can rapidly recover one of two prime factors when one of the two following circumstances occurs. The first requirement is that all prime factors of \( p - 1 \) are small. Another point is that all prime factors of \( q - 1 \) are small. In addition, there are two techniques to implement Pollard’s \( p - 1 \) algorithm.

Assigning \( b \in \mathbb{Z}^+ \) and \( \gcd(b, n) = 1 \), the first technique [26] is to assign \( B \) and compute
\[ A = \prod_{\text{prime } p \leq B} p^{\left\lfloor \log p \right\rfloor B} \].

After \( B \) and \( A \) are found, the next process is to compute \( t = \gcd(b^A - 1, n) \). The outcome is split into two categories. The first requirement is that \( t \) is less than \( n \) and not equal to 1. If this happens, then \( t \) is one of the two prime factors of \( n \). On the other hand, some prime factors of \( p - 1 \) or \( q - 1 \) are not the member in \( B \). Therefore, \( B \) must be assigned larger in order to find the new value until \( t \neq 1 \) is found. The algorithm of Pollard’s \( p - 1 \) in [26] is shown below.

The drawback of this approach, however, is that it assigns a too small value of \( B \), \( B \) must be expanded. If this occurrence occurs, there are two issues that will result in high costs.

Problem 1: The new prime member must be included in \( B \). That is, the method for determining whether a new member is a prime number is necessary such as trial division algorithm and Miller–Rabin test [27]. However, these algorithms consume at a high cost. For example, assuming \( B = \{2, 3, 5, 7\} \) and \( t = 1 \), therefore, \( B \) must be expanded. The first integer used in the verification procedure is 9. Nonetheless, after completing the prime verification method, 9 is not a prime number. The following digit should be 11. As this is a prime number, \( B \) is reassigned to \( B = \{2, 3, 5, 7, 11\} \). However, \( B \) may be reassigned again when \( t \) is still equal to 1. Therefore, the purpose of this example demonstrates that searching for the new member, which must be a prime integer, is a waste of time.
Algorithm 1 Pollard’s $p − 1$ in [26]

Input: $n$

Output: $p$, $q$ or failure

1. Select $B = \{p_1, p_2, p_3, \ldots, p_i\}$
2. Select $b$ randomly
3. $k ← 1$
4. While $p_k ≤ p_i$ do
5.   $e ← \left\lfloor \frac{\log B}{\log p_k} \right\rfloor$
6.   $b ← b \times b^{p_k} \pmod n$
7.   $k ← k + 1$
8. End While
9. $t ← \gcd(b − 1, n)$
10. IF $t == 1$ Then
11.   Select the new $B$ and go to Step 2 or return failure
12. ELSE IF $t == n$ Then
13.   Go to Step 2 or return failure
14. ELSE
15.   $p ← t$
16.   $q ← \frac{n}{p}$
17.   EndIF

Problem 2: all prime factors of $p − 1$ are a member of $B$ but one of their exponents is too small. For example, assuming $p − 1 = 2^2 \times 3 \times 5^{23}$ that they are not still found and $B = 5$, $B = \{2, 3, 5\}$ is chosen, then $A$ is calculated from $A = 2^{\left\lfloor \log 2 \right\rfloor} \times 3^{\left\lfloor \log 3 \right\rfloor} \times 5^{\left\lfloor \log 5 \right\rfloor} = 2^2 \times 3 \times 5$. Despite the fact that all of the prime factors in this example are members of $B$, the exponent of 5 is too small. As a result, $B$ needs to be reallocated. However, it is a waste of time to search for a new member of $B$ until the exponent of 5 equals 23 is discovered.

As a result, when one of the two issues listed above occurs, the first strategy may be ineffective. Therefore, the second technique may be preferable.

As $B$ is not picked in the second approach, it varies from the first technique. As a result, the prime checking procedure is unnecessary.

Assigning $k ∈ \mathbb{Z}^+$ and $k! = (p − 1) \times q$, from Fermat’s little theorem, $b^{p − 1} \equiv 1 \pmod p$.

Then,

$$b^{k!} \equiv b^{(p−1)\times q} \equiv 1^q \equiv 1 \pmod p$$

That mean, $p \mid b^{k!} − 1$ and $p \mid n$. Therefore $\gcd(b^{k!} − 1, n) = p$

For the implementation, the initial value of $k$ is 1 and compute $b = b^k \pmod n$ to check the result of $\gcd(b^{k!} − 1, n)$. If the result is equal to 1, $k$ must be increased by 1 to compute the new value of $b$: $b = b^k \pmod n$ until the result of $\gcd(b^{k!} − 1, n) ≠ 1$ is found. On the other hand, the result of $\gcd(b^{k!} − 1, n)$ is a prime factor of $n$ when it is not equal to 1. The algorithm of Pollard’s $p − 1$ [28] is shown below,

Algorithm 2 Pollard’s $p − 1$ [28]

Input: $n$

Output: $p$, $q$

1. Select $b$ randomly
2. $k ← 1$
3. $t ← \gcd(b^k − 1, n)$
4. While $t$ equals to 1 do
5.   $k ← k + 1$
6.   $b ← b^k \pmod n$
7.   $t ← \gcd(b − 1, n)$
8. End While
9. $p ← t$
10. $q ← \frac{n}{p}$
In fact, this algorithm is chosen as the compared method to the proposed method.

5. The Proposed Method

The purpose of this work is to present a specific method that is a variant of Pollard’s P − 1. In fact, when all prime factors of P − 1 or Q − 1 are members of B, the suggested techniques are highly efficient. Assuming all prime factors of P − 1 or Q − 1 are the members of B, the bit length of \( \sqrt{N} \) is represented as the maximum loops to finish the proposed methods.

Assuming P − 1 = \( p_1^{a_1} \times p_2^{a_2} \times p_3^{a_3} \times \ldots \times p_i^{a_i} \) where \( a_1, a_2, a_3, \ldots, a_i \) and \( a_i \in \mathbb{Z}^+ \) and \( a_1, a_2, a_3, \ldots, a_i < a_j \). Theorem 1 shows that P is also recovered by computing \( \gcd(b^{p_1 \times p_2 \times p_3 \times \ldots \times p_i} - 1, n) \), \( b \in \mathbb{Z}^+ \) and \( \gcd(b, n) = 1 \).

**Theorem 1.** Assuming \( p \) is prime, \( P − 1 = p_1^{a_1} \times p_2^{a_2} \times p_3^{a_3} \times \ldots \times p_i^{a_i} \), where \( p_1, p_2, p_3, \ldots, p_i \) are a prime number and \( a_1, a_2, a_3, \ldots, a_i < a_j \), then

\[
b^{p_1 \times p_2 \times p_3 \times \ldots \times p_i} \equiv 1 \pmod{P}
\]

where \( b \in \mathbb{Z}^+ \) and \( \gcd(b, p) = 1 \).

**Proof**

\[
b^{p_1 \times p_2 \times p_3 \times \ldots \times p_i} \equiv b^{p_1 \times p_2 \times p_3 \times \ldots \times p_i} \pmod{P}
\]

Assigning, \( a_j = a_1 + x_1 = a_2 + x_2 = a_3 + x_3 = \ldots = a_i + x_i \), where \( x_1, x_2, x_3, \ldots, x_i \in \mathbb{Z} \), then

\[
b^{p_1 \times p_2 \times p_3 \times \ldots \times p_i} \equiv b^{p_1 \times p_2 \times p_3 \times \ldots \times p_i} \equiv b^{p_1 \times p_2 \times p_3 \times \ldots \times p_i} \equiv b^{p_1 \times p_2 \times p_3 \times \ldots \times p_i} \equiv b^{(p_1 \times p_2 \times p_3 \times \ldots \times p_i)} \pmod{P}
\]

From Fermat’s little theorem, \( b^{p-1} \equiv 1 \pmod{P} \), then

\[
b^{p_1 \times p_2 \times p_3 \times \ldots \times p_i} \equiv (1)^{p_1 \times p_2 \times p_3 \times \ldots \times p_i} \equiv 1 \pmod{P} \]

Then, \( P \mid b^{p_1 \times p_2 \times p_3 \times \ldots \times p_i} - 1 \) and \( P \mid n \). Therefore, if \( b^{p_1 \times p_2 \times p_3 \times \ldots \times p_i} - 1 \) is less than \( n \), then

\[
\gcd\left( b^{p_1 \times p_2 \times p_3 \times \ldots \times p_i} - 1, n \right) = p
\] (5)

Example 1: Assuming \( n = 498592715639 = p \times q = 83952301 \times 5939 \), where 83952300 = \( 2^2 \times 3 \times 5^2 \times 23^4 \).

As the largest exponent of \( 2^2 \times 3 \times 5^2 \times 23^4 \) is 4, therefore

\[
\gcd\left( 3^{(2 \times 3 \times 5 \times 23)^4} - 1, 498592715639 \right) = 83952301
\]

In fact, \( 3^{(2 \times 3 \times 5 \times 23)^4} - 1 \pmod{n} = 264533700451 = 3151 \times 83952301 = 3151 \times p \).

In fact, none of the prime factors of \( P - 1 \) are revealed. As a consequence, B is split into two distinct instances. The first instance is that B is assigned too large. On the other hand, the second example is that B is assigned too small. The occurrence of the first instant is indicated in Theorem 2.
**Theorem 2.** Assuming, \( p - 1 = p_1^{a_1} \times p_2^{a_2} \times p_3^{a_3} \times \ldots \times p_i^{a_i} \), where \( p_1, p_2, p_3, \ldots, p_i, p_i \) are prime numbers that \( p_1, p_2, p_3, \ldots, p_i < p_i \) and \( a_1, a_2, a_3, \ldots, a_i \leq a_j \), then

\[
b^{(p_1 \times p_2 \times p_3 \times \ldots \times p_i \times p_j)} \equiv 1 \pmod{p}
\]

where \( b \in \mathbb{Z}^+ \) and \( \gcd(b, p) = 1 \).

**Proof**

Assigning, \( a_j = a_1 + x_1 = a_2 + x_2 = a_3 + x_3 = \ldots = a_i + x_i \), where \( x_1, x_2, x_3, \ldots, x_i \in \mathbb{Z}^+ \), then

\[
b^{(p_1 \times p_2 \times p_3 \times \ldots \times p_i \times p_j)} \equiv b^{(p_1^{(a_1+x_1)} \times p_2^{(a_2+x_2)} \times p_3^{(a_3+x_3)} \times \ldots \times p_i^{(a_i+x_i)} \times p_j)}
\]

\[
\equiv b^{(p_1^{a_1} \times p_2^{a_2} \times p_3^{a_3} \times \ldots \times p_i^{a_i} \times p_j)}
\]

\[
\equiv b^{(p-1)^\kappa \times p_1^{a_1} \times p_2^{a_2} \times p_3^{a_3} \times \ldots \times p_i^{a_i} \times p_j}\pmod{p}
\]

From Fermat’s little theorem, \( b^{p-1} \equiv 1 \pmod{p} \), then

\[
b^{(p_1 \times p_2 \times p_3 \times \ldots \times p_i \times p_j)} \equiv (1)^\kappa \times p_1^{a_1} \times p_2^{a_2} \times p_3^{a_3} \times \ldots \times p_i^{a_i} \times p_j
\]

\[
\equiv 1 \pmod{p} \square
\]

According to the information in Theorem 2, the greatest exponent is still \( a_j \), which is the same as the conclusion in Theorem 1.

**Example 2:** Assuming \( n = 498592715639 \) and \( B = \{2, 3, 5, 7, 11, 13, 17, 19, 23, 29\} \)

\[
\gcd(3^{(2 \times 3 \times 5 \times 7 \times 11 \times 13 \times 17 \times 19 \times 23 \times 29)^4} - 1, 498592715639) = 83952301
\]

In fact, \( 3^{(2 \times 3 \times 5 \times 7 \times 11 \times 13 \times 17 \times 19 \times 23 \times 29)^4} - 1 \pmod{n} = 15531175685 = 185 \times 83952301 = 185 \times p \)

Although, the multiple of \( p \) from the result of \( \left( \prod_{\text{prime} p \in B} p \right)^4 - 1 \pmod{n} \) in Example 2 is different from the result in Example 1, the smallest exponent to find the target is the same.

Assuming all prime factors of \( p - 1 \) are the members of \( B \), the first algorithm of the suggested technique is based on the notion of finding the lowest exponent. However, the starting value of the exponent (\( k \)) should be 1, and it should be increased by one every time that the result from\( \gcd \left( \left( \prod_{\text{prime} p \in B} p \right)^k, n \right) \) is equal to 1, until a result that is not equal to 1 is obtained. The first algorithm which is called the Improvement of Pollard’s \( p - 1 \) version 1 (IP1V1) is as follows:
Algorithm 3 IPP1_V1

**Input:** \( n, B = \{p_1, p_2, p_3, \ldots, p_i\} \)

**Output:** \( p, q \)

**Condition:** All prime factors of \( p - 1 \) are the members of \( B \)

1. Select \( a \in \mathbb{Z}_n \)
2. \( P \leftarrow \prod_{\text{prime} \ p \in B} p \)
3. \( a \leftarrow a^P \pmod{n} \) \( q \leftarrow \frac{n}{P} \)
4. \( t \leftarrow \gcd(a - 1, n) \)
5. While \( t \) equals to 1 do
6. \( a \leftarrow a^t \pmod{n} \)
7. \( t \leftarrow \gcd(a - 1, n) \)
8. End While
9. \( p \leftarrow t \)
10. \( q \leftarrow \frac{n}{P} \)

Example 3: Assuming \( n = 498592715639 \) and \( B = \{2, 3, 5, 7, 11, 13, 17, 19, 23, 29\} \), factoring \( n \) by using IPP1_V1

From IPP1_V1,

- **Step 1:** \( a = 3 \)
- **Step 2:** \( P = 6469693230 \)
- **Step 3:** \( a = 3^{6469693230} \pmod{498592715639} = 429172125187 \)
- **Step 4:** \( t = \gcd(429172125186, 498592715639) = 1 \)
  
  As \( t = 1 \), the process in loop is required.

**Loop 1:**

- **Step 6:** \( a \equiv 429172125187^{6469693230} \pmod{498592715639} = 175129628647 \)
- **Step 7:** \( t = \gcd(175129628646, 498592715639) = 1 \)
  
  As \( t = 1 \), the process in loop is required.

**Loop 2:**

- **Step 6:** \( a \equiv 175129628647^{6469693230} \pmod{498592715639} = 125962556910 \)
- **Step 7:** \( t = \gcd(125962556909, 498592715639) = 1 \)
  
  As \( t = 1 \), the process in loop is required.

**Loop 3:**

- **Step 6:** \( a \equiv 125962556910^{6469693230} \pmod{498592715639} = 15531175686 \)
- **Step 7:** \( t = \gcd(15531175685, 498592715639) = 83952301 \)
  
  As \( t = 83952301 \), jump to step 9
- **Step 9:** \( p = 83952301 \)
- **Step 10:** \( q = 5939 \)

Therefore, \( p = 83952301 \) and \( q = 5939 \)

In this example, the last value of \( a \) is that \( a \equiv ((a^P)^P)^P \equiv a^P \pmod{n} \). In this example, four modular exponentiations are necessary. When Pollard’s \( p - 1 \) in [28] is used, however, this work needs 92 modular exponentiations.

Table 1 compares the number of modular exponentiations in Example 3 between IPP1_V1 and Pollard’s \( p - 1 \). It means that the expenses of computing modular exponentiation are decreased about 96 percent.
Table 1. Comparison of number of modular exponentiations in Example 3 between IPP1_V1 and Pollard’s $p - 1$.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Number of Modular Exponentiations</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPP1_V1</td>
<td>4</td>
</tr>
<tr>
<td>Pollard’s $p - 1$</td>
<td>Cannot be solved (only members in B are considered)</td>
</tr>
<tr>
<td>Pollard’s $p - 1$</td>
<td>92</td>
</tr>
</tbody>
</table>

Assuming B is not changed, it implies that although all prime factors of $p - 1$ are the member of B, Pollard’s $p - 1$ in [26] cannot solve the problem in Example 3. The reason is that $p - 1 = 83952300 = 2^2 \times 3 \times 5^2 \times 23^4$ but the result of $\left\lfloor \frac{\log 29}{\log 23} \right\rfloor$ is 1. Therefore, the exponent of 23 is 1 and the result from Algorithm 1 is always a fail whenever B is not increased.

However, IPP1_V1 has the drawback of being unable to factor n when at least one of the prime factors of $p - 1$ is not a member of B. In fact, if the second situation occurs, IPP1_V1 must be adjusted to handle this circumstance, which is known as improvement of Pollard’s $p - 1$ version 2 (IPP1_V2). In addition, it indicates that the highest feasible value of $p - 1$ is $\lfloor \sqrt{n} \rfloor$. Assigning x is represented as the bit length of $\lfloor \sqrt{n} \rfloor$, if the result of $\gcd(a^{p_x}(\mod n) - 1, n)$ is still equal to 1, then B is surely allocated too small. Therefore, the idea of IPP1_V2 is to expand the member of B. Nevertheless, the new member for the proposed method may not be a prime number to avoid the prime verification process.

Assuming $p_i$ is the maximum member of B and the result is still not found, the next value of a should be assigned as follows:

$$a \equiv \left( (a^{p_x})^{p_i+1} \right) \pmod{n} \quad (6)$$

In fact, $p_i + 1$ is always an even number, then

$$p_i + 1 = 2 \times p_x \quad (7)$$

where $p_x \in \mathbb{Z}^+$

As $p_x < p_i$, all prime factors of $p_x$ are the member of B. Then,

$$\left( (a^{p_x})^{p_i+1} \right) \equiv a^{p_x \times (p_i+1)}$$

As 2 and prime factors of $p_x$ are the member of B, therefore, it is certain that

$$\gcd\left( \left( (a^{p_x})^{p_i+1} \right) \pmod{n} - 1, n \right) = 1 \quad (8)$$

As a result, when the new exponent comes between the range $p_i + 1$ to $2 \times p_x$, it should be allocated an odd integer. On the other hand, if the maximum exponent, $p_y$, is larger than $2 \times p_i$ and the target is still not found, the new exponent should be assigned as $p_y + 1$, $p_y + 2$, $p_y + 3$, … until the target is found.

The algorithm of IPP1_V2 is as follows:
Algorithm 4 IPP1_V2

Input: \( n, B = \{ p_1, p_2, p_3, \ldots, p_i \} \)

Output: \( p, q \)

1. Select \( a \in \mathbb{Z}_n \)
2. \( P \leftarrow \left( \prod_{\text{prime} p \in B} p \right) \)
3. \( x \leftarrow \text{bit length of } \lfloor \sqrt{n} \rfloor \)
4. \( i \leftarrow 0 \)
5. \( t \leftarrow 1 \)
6. While \( t == 1 \) and \( i < x-1 \) do
7. \( i \leftarrow i + 1 \)
8. \( a \leftarrow a^P \pmod{n} \)
9. \( t \leftarrow \gcd(a - 1, n) \)
10. End While
11. IF \( t == 1 \) Then
12. \( p_x \leftarrow p_i \)
13. \( \text{max} \leftarrow 2 \times p_x \)
14. While \( t == 1 \) Do
15. IF \( p_x < \text{max} \) Then
16. \( p_x \leftarrow p_x + 2 \)
17. Else
18. \( p_x \leftarrow p_x + 1 \)
19. End IF
20. \( a \leftarrow a^{p_x} \pmod{n} \)
21. \( t \leftarrow \gcd(a - 1, n) \)
22. End While
23. End IF
24. \( p \leftarrow t \)
25. \( q \leftarrow n \)

There are two while-loops in IPP1_V2. The first loop runs from step 6 until step 9. In actuality, when all prime factors of \( p - 1 \) are members of \( B \), just the first loop is necessary. On the other hand, the second loop is required to finish the process. The second loop is required whenever \( t \) is equal to 1 after computing \( x - 1 \) round in the first loop. Furthermore, each iteration of the while-loop requires one modular exponentiation.

Example 4: Assuming \( n = 498592715639 \) and choosing \( B = \{2, 3, 5, 7, 11, 13, 17, 19\} \), factoring \( n \) by using IPP1_V2

From IPP1_V2,

Step 1: \( a = 3 \)
Step 2: \( P = 9699690 \)
Step 3: \( x = 20 \)
Step 4: \( i = 0 \)
Step 5: \( t = 1 \)

As \( t = 1 \) and \( i < 19 \), the process in loop is required (Step 7 to Step 9).

Loop 1 (1st While Loop):

Step 7: \( i = 1 \)
Step 8: \( a \equiv 3^{9699690} \pmod{498592715639} = 269545421502 \)
Step 9: \( t = \gcd(269545421501, 498592715639) = 1 \)

As \( t = 1 \) and \( i < 19 \), the process in loop is required (Step 7 to Step 9).

Loop 2 (1st While Loop):

Step 7: \( i = 2 \)
Step 8: \( a \equiv 269545421502^{9699690} \pmod{498592715639} = 217253801708 \)
Step 9: \( t = \gcd(217253801707, 498592715639) = 1 \)

In fact, Loop 3 to Loop 19, \( t \) is still equal to 1 and \( a = 13501756563 \) after finish Loop 19, then
Steps 11–23: As t = 1, then
Step 12: $p_x = 19$
Step 13: max = 38
As t = 1, the process in the second loop is required (Step 15 to 21).

Loop 1 (2nd While Loop):
Steps 15–19: As $p_x < 38$, then $p_x = 21$
Step 20: $a \equiv 13501756563^{21} \pmod{498592715639} = 329268277840$
Step 21: $t = \gcd(329268277839, 498592715639) = 1$
As $t = 1$, the process in the second loop is required (Step 15 to 21).

Loop 2 (2nd While Loop):
Steps 15–19: As $p_x < 38$, then $p_x = 23$
Step 20: $a \equiv 329268277840^{23} \pmod{498592715639} = 460199361777$
Step 21: $t = \gcd(460199361776, 498592715639) = 1$
In fact, Loop 3 to Loop 62, t is still equal to 1 and $a = 31575775984$ and $p_x = 91$ after finish Loop 62, then

Loop 63 (2nd While Loop):
Steps 15–19: As $p_x > 38$, then $p_x = 92$
Step 20: $a \equiv 31575775984^{92} \pmod{498592715639} = 403474758607$
Step 21: $t = \gcd(403474758606, 498592715639) = 83952301$
As t is still equal to 1 after 19 rounds in the first loop, the second loop is necessary in Example 4. In addition, the first and second loops typically require 19 and 63 loops, respectively. As a result, there are 82 modular exponentiations. Pollard’s $p - 1$ [28], on the other hand, needs 92 modular exponentiations. As a result, utilizing IPP1_V2 to complete the operation costs are less than Pollard’s $p - 1$. Table 2 compares the number of modular exponentiations between IPP1_V1, IPP1_V2 and Pollard’s $p - 1$ to identify a prime factor in Example 4.

### Table 2. Comparison of number of modular exponentiations in Example 4 between IPP1_V1, IPP1_V2 and Pollard’s $p - 1$.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Number of Modular Exponentiations</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPP1_V1</td>
<td>Cannot be solved (only members in B are considered)</td>
</tr>
<tr>
<td>IPP1_V2</td>
<td>82</td>
</tr>
<tr>
<td>Pollard’s $p - 1$ [26]</td>
<td>Cannot be solved (only members in B are considered)</td>
</tr>
<tr>
<td>Pollard’s $p - 1$ [28]</td>
<td>92</td>
</tr>
</tbody>
</table>

When all prime factors of $p - 1$ or $q - 1$ are a member of B, it follows that both IPP1_V1 and IPP1_V2 are extremely efficient. Indeed, if this condition is met, the maximum number of loops required to complete the process is $x$. Assuming bit length of $n$ is 2048, the bit length of $\left\lfloor \sqrt{n} \right\rfloor$ is about 1024, $x \approx 1024$. As a result, the greatest number of loops required to retrieve a prime factor of $n$ are 1024. IPP1_V1 cannot be chosen to complete the job if at least one value that is a prime factor of $p - 1$ or $q - 1$ is not a member of B. On the other hand, IPP1_V2 can be selected to finish the task for all case of $n$. In addition, as demonstrated in Example 4, IPP1_V2 is still more efficient than Pollard’s $p - 1$ in some instances where at least one value is not a member of B.

However, it is possible that all prime factors of $p - 1$ and $q - 1$ are the members of B, B is assigned too large. If this happens, IPP1_V2 always returns $n$ as a result. Then, B should be decreased in size. Therefore, IPP1_V2 has to be tweaked a little to accommodate all $n$ values:
Algorithm 5 IPP1_V2 (The improvement)

Input: n, B = \{p_1, p_2, p_3, \ldots, p_i\}
Output: p, q
1. Select a ∈ \mathbb{Z}_n
2. \(P \leftarrow \prod_{\text{prime } p \in B} p\)
3. s ← bit length of \(\lfloor \sqrt{n} \rfloor\)
4. i ← 0
5. t ← 1
6. While t == 1 and i < s-1 do
7. i ← i + 1
8. a ← \(a^P \mod n\)
9. t ← gcd(a − 1, n)
10. End While
11. IF t == 1 Then
12. \(p_x \leftarrow p_i\)
13. max ← 2\(Xp_x\)
14. While t == 1 Do
15. IF \(p_x < \text{max}\) Then
16. \(p_x \leftarrow p_x + 2\)
17. Else
18. \(p_x \leftarrow p_x + 1\)
19. EndIF
20. a ← \(a^{p_x} \mod n\)
21. t ← gcd(a − 1, n)
22. EndWhile
23. Else IF t == n then
24. B ← Reduce the members of B
25. Go Back to Step 1
26. EndIF
27. p ← t
28. q ← \(n/p\)

In fact, when all prime factors of \(p - 1\) or \(q - 1\) are members of B, both IPP1_V1 and IPP1_V2 have the same efficiency. In the experimental result section, only IPP1_V2 is chosen for the implementation since this technique can finish all values of n.

6. Results and Discussion

The experiment to prove that IPP1_V2 is an efficient approach when all prime factors of \(p - 1\) or \(q - 1\) are members of B will be explained in this section. The various sizes of n included in this experiment are 64 bits, 128 bits, 256 bits, 512 bits, 1024 bits, and 2048 bits. However, all prime factors of \(p - 1\) or \(q - 1\) are the members of B = \{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97\}, a prime number which is between 1 and 100. In addition, IPP1V2 is compared to Pollard’s \(p - 1\) [26] and Pollard’s \(p - 1\) [28].

In addition, BigInteger class in Java is chosen to implement each algorithm because this class can manage an infinite number of integers generated by the String class. All experiments were performed on a 2.53 GHz Intel® Core i5 with 8 GB memory to control the same resource.

Figure 1 shows that IPP1_V2 outperforms Pollard’s \(p - 1\) for all values of n for which all prime factors of \(p - 1\) or \(q - 1\) are members of B. Using IPP1_V2, the average time is lower at about 80 to 90 percent in comparison to Pollard’s \(p - 1\) [28]. However, it becomes lower at about 40–50 percent when IPP1_V2 is compared to Pollard’s \(p - 1\) [26].
Figure 1. Comparison of time to recover a prime factor between Pollard’s $p - 1$ and IPP1_V2.

The following portion of this section will provide instances of $n$ to calculate the number of modular exponentiations and to complete the procedure using Pollard’s $p - 1$ [26], Pollard’s $p - 1$ [28], and IPP1_V2. In addition, two different examples in which the bit length of $n$ is 2048 are selected. The first example is that all prime factors of $p$ are a member of B (Example 5). In Example 6, at least one value is not a member of B. However, all of them are very close to the maximum value, which is a member of B.

Example 5: Assigning $n = p \times q$, when

$$n = 15246598209223733987894865238351873089972559091999188948235282912261810275631212791979079726122145412577328939010825657858675300679748504712776398335193906218885463067713490513884086503294093909704568713305865948743792347039475337631650933194458448262801770368979183551431433256933455309690231420838052270985298288712369917845526208267449508256804711179047997966533771475760304613663063286503061154605823209051951630164796392617955040771752044377188584834822430797731986696767900144307015222944405697235657118735140430611898622418009330338506410921878492137871947579101902468352390554579684919764299759335838310680231326332731450550436130809259824662100048686744686192855777939171223270599998877882006674235584855136293641256629778245757047675011067821530659540646763668429293033370216872855657225715754848894594748980147162770819982994596735304668154826197487144058999817443714145055623072797694023079687500001,q=142747142533793191172464433735174517618407613900058636260168923931064621007853272820043506706137493692760841055875749038586749271514202520389299000607230884415912019404745926278078018690059973143401068590218383426571457858113308418474880226705153959131351394468263799552328667300672121550203784663433583

In fact, this example, it shows that $p = 2^3 \times 3^{11} \times 5^{11} \times 7^{14} \times 11^{10} \times 13^{5} \times 17^3 \times 19^7 \times 23^8 \times 29^7 \times 31^{11} \times 37^{10} \times 41^9 \times 43^{11} \times 47^9 \times 53^7 \times 59^8 \times 61^9 \times 67^{11} \times 71^7 \times 73^4 \times 79^7 \times 83^7 \times 89^{11} \times 97^{10}$, all prime factors are the members of B.

Table 3 shows the comparison about number of modular exponentiations in Example 5 between IPP1_V2 and Pollard’s $p - 1$ to find a prime factor. Moreover, IPP1_V2 is also compared to the other algorithms in special-propose factoring group.
Table 3. Comparison of number of loops in Example 5 between IPP1_V2 and the algorithms in special-propose factoring group.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Loops of Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPP1_V2</td>
<td>14</td>
</tr>
<tr>
<td>Pollard’s $p - 1$</td>
<td>979</td>
</tr>
<tr>
<td>Pollard’s $p - 1$</td>
<td>25</td>
</tr>
<tr>
<td>Improved FFA</td>
<td>$\approx 2.52 \times 10^{307}$</td>
</tr>
<tr>
<td>TDA</td>
<td>$\approx 1.06 \times 10^{308}$</td>
</tr>
<tr>
<td>VFactor</td>
<td>$\approx 1.70 \times 10^{307}$</td>
</tr>
</tbody>
</table>

Table 3 demonstrates that when IPP1_V2 is used instead of Pollard’s $p - 1$, the number of modular exponentiations is greatly decreased. Furthermore, they are predicated on the greatest exponent of a prime factor of $p - 1$ or $q - 1$ for IPP1_V2. The highest value in the preceding example is 14, which is the exponent of 7; therefore, 14 modular exponentiations are required to complete the task. Moreover, in Example 5, the other algorithms in the special-propose factoring group require huge loops of computation when they are compared to IPP1_V2 and Pollard’s $p - 1$. The reason is that all prime factors of $p - 1$ are the member of B that is suitable for both of IPP1_V2 and Pollard’s $p - 1$.

Assuming $y$ is represented as the maximum exponent of a prime factor of $p - 1$ or $q - 1$, the total loops to finish the process is $T_A = y$. In fact, $y$ is always smaller than bit length of $\lfloor \sqrt{n} \rfloor$, $y < x$. It implies that IPP1_V2 is an efficient factorization algorithm when all prime factors of $p - 1$ or $q - 1$ are members of B.

Example 6:

$$n = 179814671178885285343673324723544418811647306042292907350570740123929054703909595149595761980556436038107768899203031258253216140621918752485299293669274901735796363957644999366003002626054250619274314398008881835666222672467369618652384566621754213918665681824520624658997817538566482333723226779405959741574686473713709939241547542368261645033771898314129410461107851153254068619609643173799553148709318437875804452804492614290827574763841764785870286081901507378257010382404607536741906634497933536035910396487161037952514310544119956806219030094013525813276343417335783819683615820439975874212873156044665337,
$$

$$p = 1275364852518522004987927193584319778356890901990821945863317096431858036639770776568589174103121964022670190267591227806565500633455787590630176291120804180394985344070440384272757597974447679073582515529882958740416620278779792145823603556468850310508131442875135550959292904050213372209203649404928000001,
$$

$$q = 140990768954810850705492505007650337537512522816130254048645374654640504396183970805951940107274979065453345119798244297352718473041345180420866626661829571319534466230367793012633481692758361543787060689421832681924658418892315043479581694033586128892761330407992065029561257362360533098313995134868465337
$$

In fact, in this example, it shows that, $p - 1 = 2^{20} \times 3^{17} \times 5^7 \times 7^{15} \times 11^{16} \times 13^{16} \times 17^{12} \times 19^9 \times 23^{19} \times 29^{11} \times 31^{12} \times 37^{14} \times 41^{14} \times 43^{11} \times 47^{19} \times 53^{10} \times 97^{20} \times 101$, a prime number which is 101 is not a member of B.

Table 4 shows the comparison of the number of modular exponentiations in Example 6 between IPP1_V2 and Pollard’s $p - 1$ to find a prime factor. Moreover, IPP1_V2 is also compared to the other algorithms in the special-propose factoring group.
Table 4 demonstrates that using IPP1_V2 to compute modular exponentiation costs in Example 6 is less than using Pollard’s p – 1 method. As a result, in some situations, it indicates that IPP1_V2 is quicker than Pollard’s p – 1 in [28], despite the fact that some prime factors of p – 1 or q – 1 are not members of B. In addition, in Example 6, the other algorithms in the special-propose factoring group require huge computation loops when they are compared to IPP1_V2 and Pollard’s p – 1. The reason is that almost all prime factors of p – 1 are a member of B. Furthermore, the other prime factor of p – 1, 101, which is not a member of B, is very close to the maximum value of B, 97. Therefore, IPP1_V2 is the best algorithm to recover prime factors of n in Example 6. Nevertheless, Pollard’s p – 1 in [26] cannot solve this problem, because all prime factors of p – 1 or q – 1 must be a member of B.

Both Example 5 and Example 6 imply that IPP1_V2 is an efficient algorithm when all prime factors of p – 1 or q – 1 are members of B. However, if some prime factors of p – 1 or q – 1 are not members of B but are very close to the maximum value of B, IPP1_V2 is still an efficient algorithm.

Assuming \( p_w \) is the maximum factor of p – 1 or q – 1 and \( p_w \) is not the member of B, the total loops are equal to the Equation (9):

\[
T_B = p_w \times (z - 1) + (p_w - p_i) - \frac{(p_i + 1)}{2} + (x - 1)
\]  

(9)

where

- \( T_B \) is total loops,
- \( p_w \) is the maximum factor of p – 1 or q – 1 that \( p_w \) is not a member of B,
- \( p_i \) is the maximum member of B which is a prime factor of p – 1 or q – 1,
- \( x \) is bit length of \( \sqrt{n} \).

However, total loops to finish Pollard’s p – 1 in [28] are \( z \times p_w \). As \( 97^{20} \) is the maximum factor of p – 1 in Example 6, then \( 20 \times 97 = 1940 \) is represented as the number of loops (or number of modular exponentiations) to finish Pollard’s p – 1.

Assuming there are some prime factors of p – 1 or q – 1 that are not members of B, the condition that the costs to compute modular exponentiations by using IPP1_V2 are less than Pollard’s p – 1 is that \( T_B \) must be less than \( z \times p_w \).

For time complexity analysis, there are two cases to evaluate the time complexity in the best-case and worst-case scenarios.

(1) Analysis of the best-case scenario

In fact, it is separated into two scenarios in order to examine the best-case scenario for the proposed method. For the first situation, assuming \( p_1, p_2, p_3, \ldots, p_i \) are members of B and \( p_1 \times p_2 \times p_3 \times \ldots \times p_i = p – 1 \), all exponents are equal to 1. It requires one modular exponentiation to find p. However, in the second case, all prime factors of p – 1 are still

Table 4. Comparison of number of loops in Example 6 between IPP1_V2 and the algorithms in special-propose factoring group.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Number of Modular Exponentiations</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPP1_V2</td>
<td>1025</td>
</tr>
<tr>
<td>Pollard’s p – 1 [28]</td>
<td>1940</td>
</tr>
<tr>
<td>Pollard’s p – 1 [26]</td>
<td>Cannot be solved (only members in B are considered)</td>
</tr>
<tr>
<td>Improved FFA [14]</td>
<td>( \approx 2.68 \times 10^{307} )</td>
</tr>
<tr>
<td>TDA</td>
<td>( \approx 1.27 \times 10^{307} )</td>
</tr>
<tr>
<td>VFactor</td>
<td>( \approx 6.72 \times 10^{306} )</td>
</tr>
</tbody>
</table>
members of B, but there is at least one exponent of $p_j$, where $1 \leq j \leq i$, which is greater than 1. As $p - 1$ is always an even number, 2 is one of prime factors of $p - 1$. Furthermore, there is at least one prime number which is greater than 2, and it is a prime factor of $p - 1$. As $p - 1 \leq \sqrt[2]{n}$, then the maximum loops are always less than $x - 1$. Therefore, the running time of the second case is $O(x)$.

(2) Analysis of the worst-case scenario

Assuming at least one prime number is not a member of B but it is a prime factor of $p - 1$, the highest value that can be a prime factor of $p - 1$ must be extremely close to $2x^{-1}$, $p_w^2 \approx 2^{x-1}$. Therefore, the running time of this case is about $O(2^{x-1})$.

In fact, it implies that the proposed method is very efficient when all prime factor of $p - 1$ or $q - 1$ are small, the running time for the best-case is $O(x)$. On the other hand, the proposed method becomes an inefficient method when there is at least one integer that is very large, and it is a prime factor of $p - 1$. Therefore, the proposed method is categorized in the special-propose factoring group.

Assuming $p < q$ and the maximum prime factor of $q - 1$ is larger than the maximum prime factor of $p - 1$, then $p \leq \sqrt[2]{n}$. In addition, $p - 1$ is an even number.

Then,

$$p - 1 = 2 \times \frac{p - 1}{2} \leq 2 \times \frac{\sqrt[2]{n} - 1}{2}$$

If $\frac{\sqrt[2]{n} - 1}{2}$ is a prime number, it implies that this integer is the highest prime number that may be a prime factor of $p - 1$. However, if it is not a prime number, it implies that all prime factors of $p - 1$ are less than $\frac{\sqrt[2]{n} - 1}{2}$. Therefore, IPP1_V2 is an efficient method when the maximum value of B is close to $\frac{\sqrt[2]{n} - 1}{2}$. However, this is the worst case of B that should be selected. In fact, it is very possible that $\frac{\sqrt[2]{n} - 1}{2}$ is not a prime number. Therefore, IPP1_V2 with a smaller size of B may still be an efficient method. Moreover, it is shown in Example 5 and Example 6 that although the bit length of n is 2048, IPP1_V2 can recover p and q rapidly with a small size of B. The reason is that all prime factors of $p - 1$ are close or equal to one of the members of B.

7. Conclusions

In this paper, a variant of Pollard’s $p - 1$ is presented. The proposed algorithm is an efficient method when all prime factors of $p - 1$ or $q - 1$ are members of B-smooth, where p and q are prime factors of the modulus. This is assuming all prime factors of $p - 1$ are the members of B. The proposed technique is quicker than Pollard’s $p - 1$ for virtually all values of the modulus, according to the experimental findings. Furthermore, even if at least one number that is a prime factor of $p - 1$ is not a member of B, the proposed method is still quicker than Pollard’s $p - 1$ for some values of the modulus. In fact, the proposed method is suitable for the case that all prime factors of $p - 1$ are small. In addition, it is shown that the running time for the best case of the proposed method that all prime factors of $p - 1$ are the member of B is $O(x)$, where x is represented as bits length of $\sqrt[2]{n}$.

Funding: This research was funded by Department of Computer and Communication Engineering, Faculty of Technology, Udon Thani Rajabhat University, Udon Thani, 41000, Thailand.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The author declares no potential conflict of interest.
References


