Integer Factorization Algorithms

Connelly Barnes
Department of Physics, Oregon State University

December 7, 2004

This document has been placed in the public domain.

Contents

I. Introduction 3
1. Terminology 3
2. Fundamental Theorem of Arithmetic 3
3. Practical Motivation 3

II. Algorithms 5
1. Algorithm: Trial Division 5
2. Pseudocode: Trial Division 5
3. Algorithm: Fermat Factorization 5
4. Pseudocode: Fermat Factorization 6
5. Algorithm: Pollard rho Factorization 6
6. Pseudocode: Pollard rho Factorization 7
7. Algorithm: Brent's Factorization Method 7
8. Pseudocode: Brent's Factorization Method 8
9. Algorithm: Pollard *p*-1 Factorization 9
10. Pseudocode: Pollard *p*-1 Factorization 10

III. Running times 11
1. Running time: Trial Division 11
2. Running time: Fermat Factorization 11
3. Running times: Empirical Results 11

IV. Failures of Probabilistic Algorithms 12

V. Conclusion 13

Appendix A. Maple Source Code for Simulation 14

Appendix B. References 17

I. Introduction

This paper gives a brief survey of integer factorization algorithms. We offer several motivations for the factorization of large integers. A number of factoring algorithms are then explained, and pseudocode is given for each. Bounds in running time are found for algorithms which are always successful, and failure cases are shown for probabilistic algorithms. Finally, the run times of all presented algorithms are plotted for certain prime products and compared.

1. Terminology

Big *O* notation:

The function $f(x)$ is $O(g(x))$ as $x \to \infty$ if and only if there are positive real constants *c*, *k* such that for every $x > k$, $0 \le f(n) \le cg(n)$.

Example: $f(x) = 2x^2 + x + 1$ is $O(x^2)$ as $x \to \infty$ for $c = 3, k = 0$.

When Big *O* notation is applied to the running time or storage requirements of an algorithm, one may write simply $O(g(x))$, and it is assumed that $x \to \infty$. If multiple variables are present, the variable which goes to infinity is indicated. As part of the definition of $O(g(x))$, all possible executions of the algorithm must be considered as $x \to \infty$.

Trivial factor:

A positive integer factor *s* of *N* such that *s* = 1 or *s* = *N*.

Nontrivial factor:

A positive integer factor *s* of *N* such that $1 < s < N$.

Prime number:

A positive integer greater than 1 that is divisible by no positive integers other than 1 and itself.

2. Fundamental Theorem of Arithmetic

The fundamental theorem of arithmetic states that every positive integer can be written uniquely as a product of primes, when the primes in the product are written in nondecreasing order.

3. Practical Motivations

The fundamental theorem of arithmetic implies that any composite integer can be factored. Given the number N = 21, it is straightforward to find the factors of N: $21 = 3 \cdot 7$. Now consider a larger composite number:

$$
\begin{aligned}
N = {} & 25195908475657893494027183240048398571429282126204 \\
& 03202777713783604366202070759555626401852588078440 \\
& 69182906412495150821892985591491761845028084891200 \\
& 72844992687392807287776735971418347270261896375014 \\
& 97182469116507761337985909570009733045974880842840 \\
& 17974291006424586918171951187461215151726546322822 \\
& 16869987549182422433637259085141865462043576798423 \\
& 38718477444792073993423658482382428119816381501067 \\
& 48104516603773060562016196762561338441436038339044 \\
& 14952634432190114657544454178424020924616515723350 \\
& 77870774981712577246796292638635637328991215483143 \\
& 81678998850404453640235273819513786365643912120103 \\
& 97122822120720357.
\end{aligned}
$$

This integer is known as RSA-2048. On March 1991, RSA Laboratories announced a USD 200,000 award for the successful factorization of this number. As of November 2004, this number has not yet been factored [1].

If one is given two large prime numbers, there are fast algorithms for multiplying them together. However, if one is given the product of two large primes, it is difficult to find the prime factors. The fastest known general-purpose factoring algorithm is the General Number Field Sieve (GNFS), which in asymptotic notation takes

$$S = O\left(\exp\left[\left(\frac{64}{9}n\right)^{1/3}(\log n)^{2/3}\right]\right)$$

steps to factor an integer with n decimal digits. The running time of the algorithm is bounded below by functions polynomial in n and bounded above by functions exponential in n [2].

The apparent difficulty of factoring large integers is the basis of some modern cryptographic algorithms. The RSA encryption algorithm [3], and the Blum Blum Shub cryptographic pseudorandom number generator [4] both rely on the difficulty of factoring large integers. If it were possible to factor products of large prime numbers quickly, these algorithms would be insecure.

The SSL encryption used for TCP/IP connections over the World Wide Web relies on the security of the RSA algorithm [5]. Hence if one could factor large integers quickly, "secured" Internet sites would no longer be secure.

Finally, in computational complexity theory, it is unknown whether factoring is in the complexity class P. In technical terms, this means that there is no known algorithm for answering the question "Does integer N have a factor less than integer s?" in a number of steps that is $O(P(n))$, where n is the number of digits in N, and $P(n)$ is a polynomial function. Moreover, no one has proved that such an algorithm exists, or does not exist. In layman's terms, one can simply ask the question, "What is the fastest algorithm for factoring large numbers?" This is an important open question in mathematics [6].

II. Algorithms

1. Algorithm: Trial Division

Trial division is the simplest algorithm for factoring an integer. Assume that s and t are nontrivial factors of N such that $st = N$ and $s \le t$. To perform the trial division algorithm, one simply checks whether $s \mid N$ for $s = 2,\ldots,\lfloor\sqrt{N}\rfloor$. When such a divisor s is found, then $t = N/s$ is also a factor, and a factorization has been found for N. The upper bound of $s \le \lfloor\sqrt{N}\rfloor$ is provided by the following theorem:

Theorem. If N has nontrivial factors s, t with $st = N$ and $s \le t$, then $s \le \sqrt{N}$.
Proof. Assume $s > \sqrt{N}$. Then $t \ge s > \sqrt{N}$, and $st > N$, which contradicts the assumption that $st = N$. Hence $s \le \sqrt{N}$.

2. Pseudocode: Trial Division

```
function trialDivision(N)
  for s from 2 to floor(sqrt(N))
    if s divides N then
      return s, N/s
    end if
  end for
end function
```

If this algorithm is given composite N, then it returns a pair of nontrivial factors s, t with $s \le t$. The statement $s \mid N$ is equivalent to $s \equiv 0 \pmod{N}$, and so it can be implemented via modular arithmetic in most languages.

3. Algorithm: Fermat Factorization

This algorithm was discovered by mathematician Pierre de Fermat in the 1600s [7]. Fermat factorization rewrites a composite number N as the difference of squares:

$$N = x^2 - y^2$$

This difference of squares leads immediately to the factorization of N:

$$N = (x + y)(x - y)$$

Assume that s and t are nontrivial odd factors of N such that $st = N$ and $s \le t$. We can find x and y such that $s = (x - y)$ and $t = (x + y)$. Solving this equation, we find that $x = (s + t) / 2$ and $y = (t - s) / 2$. Here x and y are integers, since the difference between any two odd numbers is even, and an even number is divisible by two. Since $s > 1$ and $t \ge s$,

we find that $x \geq 1$ and $y \geq 0$. For particular x, y satisfying $s = (x - y)$ and $t = (x + y)$, we thus know that $x = \sqrt{N + y^2}$, and hence $x \geq \sqrt{N}$. Also, $x \leq (s+t)/2 \leq 2t/2 \leq N$.

For an algorithm, we choose $x_1 = \left\lceil \sqrt{N} \right\rceil$, and $x_{i+1} = x_i + 1$. For each i, we check whether $y_i = \sqrt{x_i^2 - N}$ is an integer and whether $(x_i + y_i), (x_i - y_i)$ are nontrivial factors of N. If both of these conditions hold, we return the nontrivial factors. Otherwise, we continue to the next i, and exit once $x_i = N$.

4. Pseudocode: Fermat Factorization

```
function fermatFactor(N)
  for x from ceil(sqrt(N)) to N
    ySquared := x * x - N
    if isSquare(ySquared) then
      y := sqrt(ySquared)
      s := (x - y)
      t := (x + y)
      if s <> 1 and s <> N then
        return s, t
      end if
    end if
  end for
end function
```

Here the `isSquare(z)` function is `true` if `z` is a square number and `false` otherwise. It is straightforward to construct an `isSquare` function by taking a square root, rounding the answer to an integer, squaring the result, and checking if the original number is reproduced.

5. Algorithm: Pollard rho Factorization

Pollard's rho method is a probabilistic method for factoring a composite number N by iterating a polynomial modulo N. The method was published by J.M. Pollard in 1975. Suppose we construct the sequence:

$$x_0 \equiv 2 \pmod{N}$$
$$x_{n+1} \equiv x_n^2 + 1 \pmod{N}$$

This sequence will eventually become periodic. It can be shown that the length of the cycle is less than or equal to N by a proof by contradiction: assume that the length L of the cycle is greater than N, however we have only N distinct x_n values in our cycle of length $L>N$, so there must exist two x_n values are congruent, and these can be identified as the “starting points” of a cycle with length less than or equal to N. Probabilistic arguments show that the expected time for this sequence (mod N) to fall into a cycle and expected length of the cycle are both proportional to $\sqrt{N}$, for almost all N [8]. Other

initial values and iterative functions often have similar behavior under iteration, but the function $f(n) = x_n^2 + 1$ has been found to work well in practice for factorization.

Assume that *s* and *t* are nontrivial factors of *N* such that $st = N$ and $s \leq t$. Now suppose that we have found nonnegative integers *i*, *j* with $i < j$ such that $x_i \equiv x_j \pmod{s}$ but $x_i \not\equiv x_j \pmod{N}$. Since $s \mid (x_i - x_j)$, and $s \mid N$, we have that $s \mid \gcd(x_i - x_j, N)$. By assumption $s \geq 2$, thus $\gcd(x_i - x_j, N) \geq 2$. By definition we know $\gcd(x_i - x_j, N) \mid N$. However, we have that $N \nmid (x_i - x_j)$, and thus that $N \nmid \gcd(x_i - x_j, N)$. So we have that $N \nmid \gcd(x_i - x_j, N)$, $\gcd(x_i - x_j, N) > 1$, and $\gcd(x_i - x_j, N) \mid N$. Therefore $\gcd(x_i - x_j, N)$ is a nontrivial factor of *N*.

Now we must find *i*, *j* such that $x_i \equiv x_j \pmod{s}$ and $x_i \not\equiv x_j \pmod{N}$. Observe that the sequence $x_n \pmod{s}$ is periodic with the length of the cycle proportional to $\sqrt{s}$. Pollard suggested that x_n be compared to x_{2n} for *n* = 1, 2, 3, …. For each *n*, we check whether $\gcd(x_n - x_{2n}, N)$ is a nontrivial factor of *N*. If $\gcd(x_n - x_{2n}, N)$ is a trivial factor of *N*, we repeat the iterative process until a factor is found. If no factor is found, the algorithm does not terminate.

6. Pseudocode: Pollard rho Factorization

```
function pollardRho(N)
  # Initial values x(i) and x(2*i) for i = 0.
  xi  := 2
  x2i := 2
  do
    # Find x(i+1) and x(2*(i+1))
    xiPrime  := xi ^ 2 + 1
    x2iPrime := (x2i ^ 2 + 1) ^ 2 + 1
    # Increment i: change our running values for x(i), x(2*i).
    xi  := xiPrime % N
    x2i := x2iPrime % N
    s := gcd(xi - x2i, N)
    if s <> 1 and s <> N then
      return s, N/s
    end if
  end do
end function
```

Here `a % m` is a modulo operation, which returns the least nonnegative integer *y* such that $a \equiv y \pmod{m}$.

7. Algorithm: Brent's Factorization Method

Brent's factorization method is an improvement to Pollard's rho algorithm, published by R. Brent in 1980 [9]. In Pollard's rho algorithm, one tries to find a

nontrivial factor s of N by finding indices i, j with $i < j$ such that $x_i \equiv x_j \pmod{s}$ and $x_i \not\equiv x_j \pmod{N}$. The x_n sequence is defined by the recurrence relation:

$$x_0 \equiv 2 \pmod{N}$$
$$x_{n+1} \equiv x_n^2 + 2 \pmod{N}$$

Pollard suggested that x_n be compared to x_{2n} for n = 1, 2, 3, …. Brent's improvement to Pollard's method is to compare x_n to x_m, where m is the largest integral power of 2 less than n.

8. Pseudocode: Brent's Factorization Method

```
function brentFactor(N)
  # Initial values x(i) and x(m) for i = 0.
  xi  := 2
  xm  := 2
  for i from 1 to infinity
    # Find x(i) from x(i-1).
    xi := (xi ^ 2 + 1) % N
    s := gcd(xi - xm, N)
    if s <> 1 and s <> N then
      return s, N/s
    end if
    if integralPowerOf2(i) then
      xm := xi
    end if
  end do
end function
```

Here the function `integralPowerOf2(z)` is `true` if `z` is an integral power of 2 and `false` otherwise. An inefficient implementation for this function can be made by checking successive powers of 2 until a power of 2 equals or exceeds z:

```
function integralPowerOf2(z)
  pow2 := 1
  while pow2 <= z do
    if pow2 = z then
      return true
    end if
    pow2 := pow2 * 2
  end while
  return false
end function
```

In terms of more efficient operations, `integralPowerOf2(z)` is true if and only if `(z&(z-1))` is zero, where `&` is the bitwise AND operation [10]. A proof follows.

Theorem. If z is a positive integer, then z is an integral power of 2 if and only if $z \,\&\, (z-1) = 0$, where $a \,\&\, b$ denotes the bitwise AND operation of a and b.

Proof. Let there be d binary bits in z, and let $(\cdot)_i$ be an operator which gives the ith binary bit of $(\cdot)$, where $i = 1$ is the least significant bit. If z is an integral power of 2, then clearly $z_k = 0$ for $k = 1, 2, \ldots, d-1$, and $z_d = 1$. We also have that $z - 1 < z$, so clearly $(z-1)_d = 0$. Using the truth table for the logical AND operator, we find that $(z \,\&\, (z-1))_k$ must be 0 for $k = 1 \ldots d$. Hence $(z \,\&\, (z-1))_k = 0$. In the case that z is not an integral power of 2, $z_d = 1$. Let α be the largest integral power of 2 that is less than z. Then $z > \alpha$, hence $z - 1 \geq \alpha$, and thus $(z-1)_d = \alpha_d = 1$. Using the truth table for the logical AND operator at bit d we find that $(z \,\&\, (z-1))_d = 1$, hence $(z \,\&\, (z-1))_k \neq 0$. Therefore, z is an integral power of 2 if and only if $z \,\&\, (z-1) = 0$.

9. Algorithm: Pollard p-1 Factorization

Pollard's p-1 factorization method was published by J. M. Pollard in 1974 [11]. It is based on Fermat's little theorem, which states:

If p is prime, a is a natural number, and $p \nmid a$, then $a^{p-1} \equiv 1 \pmod{p}$.

Suppose we have a positive integer $k \geq 1$ and a prime $p>2$ such that $(p-1) \mid k!$. Now we can apply Fermat's little theorem with $a = 2$:

$$2^{p-1} \equiv 1 \pmod{p}$$

But since $(p-1) \mid k!$, we can write $k! = (p-1)q$ for some positive integer q. We have:

$$2^{k!} \equiv \left(2^{p-1}\right)^q \equiv 1^q \equiv 1 \pmod{p}$$

Hence $p \mid 2^{k!} - 1$. If N is an integer which has nontrivial prime factor p, then p also divides $2^{k!} - 1 + Nt$ for all integers t. We can compute $x_k \equiv 2^{k!} - 1 \pmod{N}$ for k = 1, 2, 3, …, and for each x_k check whether there exists an integer $r_k = \gcd(x_k, N)$ which divides both x_k and N. If $(p-1) \mid k!$, then we know $p \mid x_k$ and hence r_k is a nontrivial factor of N. If r_k is not a nontrivial factor of N, then it is a trivial factor of N, i.e. $r_k = 1$ or $r_k = N$. The algorithm is then:

Compute $r_k = \gcd(2^{k!} - 1, N)$ for k = 1, 2, 3…. If $r_k \notin \{1, N\}$, then r_k is a nontrivial factor and we are done.

For efficiency purposes, we can write $2^{k!} \equiv \left(2^{(k-1)!}\right)^k \pmod{N}$, so that if $2^{(k-1)!}$ is known (mod N), $2^{k!}$ can be computed by a single modular exponentiation operation.

10. Pseudocode: Pollard p-1 Factorization

```
function pollard_p1(N)
  # Initial value 2^(k!) for k = 0.
  two_k_fact := 1
  for k from 1 to infinity
    # Calculate 2^(k!) (mod N) from 2^((k-1)!).
    two_k_fact := modPow(two_k_fact, k, N)
    rk := gcd(two_k_fact - 1, N)
    if rk <> 1 and rk <> N then
      return rk, N/rk
    end if
  end for
end function
```

Here `modPow(a, b, m)` returns the least nonnegative integer y such that $a^b \equiv y \pmod{m}$. This function is typically provided in languages with big integer operations, and is known as "modular exponentiation."

For languages without modular exponentiation, we present an efficient algorithm for modular exponentiation. Write b in terms of its binary digits $b_0 \ldots b_{n-1}$, so $b = b_0 2^0 + b_1 2^1 + \ldots + b_{n-1} 2^{n-1}$ and observe that a^b can be rewritten as

$$a^b = a^{b_0 2^0} a^{b_1 2^1} \cdot \ldots \cdot a^{b_{n-1} 2^{n-1}} = \left(a^{2^0}\right)^{b_0} \left(a^{2^1}\right)^{b_1} \cdot \ldots \cdot \left(a^{2^{n-1}}\right)^{b_{n-1}}.$$

Note that for any k, $\left(a^{2^k}\right)^{b_k}$ is simply 1 if $b_k = 0$, and a^{2^k} otherwise. Thus we have:

$$a^b = \prod_{\substack{k=0 \\ b_k \neq 0}}^{n-1} a^{2^k}$$

Also note that $a^{2^{k+1}} = a^{2 \cdot 2^k} = \left(a^{2^k}\right)^2$. Via a process of repeated squaring, we can thus construct an algorithm which returns the least nonnegative integer y such that $a^b \equiv y \pmod{m}$.

```
function modPow(a, b, m):
  ans := 1
  a := a % m
  for k from 0 to infinity
    if 2^k>b then
      return ans
    end if
    if (bit k of b is nonzero) then
      ans := (ans * a) % m
    a := (a * a) % m
  end for
end function
```

Here `a % m` is a modulo operation, which returns the least nonnegative integer y such that $a \equiv y \pmod{m}$.

III. Running Times

1. Running Time: Trial Division

The worst case running time for the trial division algorithm occurs when $s = t = \sqrt{N}$, and $N = s^2$. In this case, we test divisibility for exactly $\sqrt{N} - 1$ integers. Thus the algorithm takes $O(\sqrt{N})$ steps, or when written in terms of the number of digits n of N, it requires $O(e^{n/2})$ steps.

Each divisibility test can be carried out in $O(\log N)$ time [13]. There are no more than $\sqrt{N}$ such tests, so at worst the trial division algorithm takes $O(\sqrt{N} \log N)$ time. When written in terms of the number of digits n of N, trial division takes $O(ne^{n/2})$ time.

2. Running Time: Fermat Factorization

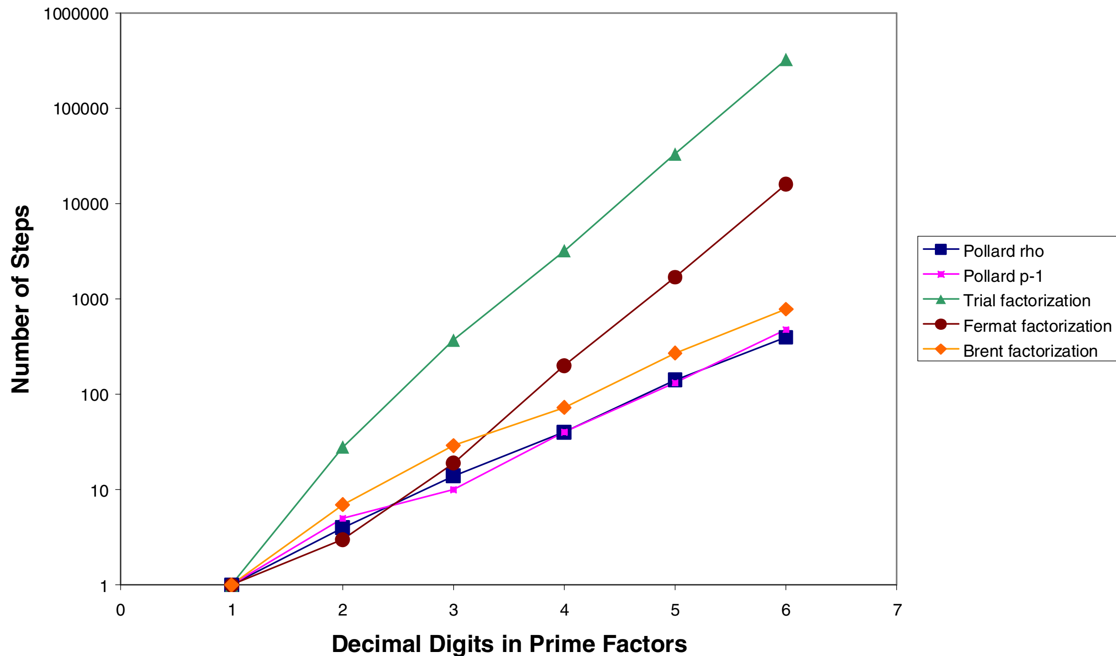
Assuming that N is the product of odd primes, the Fermat factorization as presented in Section II.4 makes no more than N steps through the `for` loop. Hence Fermat factorization takes $O(N)$ steps. When written in terms of the number of digits n of N, the algorithm takes $O(e^n)$ steps.

3. Running Time: Empirical Results

Figure 1 shows a plot of the median number of steps for each algorithm versus the number of decimal digits d in the prime factors, where "steps" is defined as the number of iterations through the `for` loop.

For each value of d, each algorithm was tested 100 times. For each test, integers s, t were chosen in a uniform random manner from the set of integers having d decimal digits. If s was composite, or t was composite, or s equaled t, then the numbers were re-selected. Once a valid pair s, t was found, the algorithm was run on the product st for up to 10^6 steps. The median number of steps is plotted for each algorithm.

Figure 1



Although the Brent factorization algorithm was touted as an improvement to the Pollard rho method, it appears to be slower in this simulation. In terms of median running times for these data, the Pollard rho and Pollard p-1 methods are fastest, and the trial factorization method is slowest.

The Maple source code used to produce these data is presented in Appendix A.

IV. Failures of Probabilistic Algorithms

The trial division and Fermat factorization algorithms always terminate, and upper bounds can be derived for the running times of these algorithms in terms of N, the number to be factored. The Pollard rho algorithm, Brent's method, and the Pollard p-1 algorithm are probabilistic, and may not finish, even for small values of N.

Example. Consider the Pollard rho algorithm for $N = 21 = 3 \cdot 7$. The sequence of x_n values generated by the algorithm is

$$x_0 \equiv 2 \pmod{21}$$
$$x_1 \equiv x_0^2 + 1 \equiv 5 \pmod{21}$$
$$x_2 \equiv x_1^2 + 1 \equiv 5 \pmod{21}$$
$$x_n \equiv x_{n-1}^2 + 1 \equiv 5 \pmod{21} \qquad \text{for } n \geq 1$$

If $n \geq 1$, $x_{2n} - x_n = 0$. The algorithm at each step for n = 1, 2, … computes $\gcd(x_{2n} - x_n, N) = \gcd(0, N) = N$. The algorithm never finds a nontrivial factor, and never terminates.

Example. Consider the Pollard p-1 algorithm for N = 65 = $13 \cdot 5$. The sequence of x_k values generated by the algorithm is:

$$x_k \equiv 2^{k!} - 1 \pmod{65} \qquad k = 1, 2, 3, \ldots$$
$$x_1 \equiv 2^1 - 1 \equiv 1 \pmod{65}$$
$$x_2 \equiv 2^2 - 1 \equiv 3 \pmod{65}$$
$$x_3 \equiv 2^6 - 1 \equiv 63 \pmod{65}$$
$$x_4 \equiv 2^{24} - 1 \equiv 0 \pmod{65}$$
$$x_{n+1} \equiv (x_n + 1)^{n+1} - 1 \equiv 1^{n+1} - 1 \equiv 0 \pmod{65} \quad \text{for } k \geq 5$$

The Pollard p-1 algorithm computes at each step $\gcd(x_k, N)$. For the first three steps, we find that gcd(1, 65) = 1, gcd(3, 65) = 1, and gcd(63, 65) = 1. For steps $k \geq 4$ we find gcd(0, 65) = 65. Hence the algorithm never finds a nontrivial factor, and never terminates.

V. Conclusion

There are no known algorithms which can factor arbitrary large integers efficiently. Probabilistic algorithms such as the Pollard rho and Pollard p-1 algorithm are in most cases more efficient than the trial division and Fermat factorization algorithms. However, probabilistic algorithms can fail when given certain prime products: for example, Pollard's rho algorithm fails for $N = 21$. Integer factorization algorithms are an important subject in mathematics, both for complexity theory, and for practical purposes such as data security on computers.

Appendix A. Maple Source Code for Simulation

```
> # Define each factorization algorithm

> # Trial division.  Factor N, return s, t, iters, where s*t = N, and
  # iters is the number of iterations made through the for loop.  If
  # more than maxsteps iterations are made, returns 1, N, maxsteps.
  trial_factor := proc(N, maxsteps)
    local x, y, iters;
    iters := 1;
    for x from 2 to floor(sqrt(N)) do
      if modp(N, x) = 0 then        # If y is an integer, return factors.
        return x, N/x, iters;
      fi;
      if iters >= maxsteps then
        return 1, N, maxsteps;
      fi;
      iters := iters + 1;
    od;
  end;

> # Fermat factorization.  Same arguments and return value as trial_factor.
  fermat_factor := proc(N, maxsteps)
    local x, y, iters;
    iters := 1;
    # Look for N = x^2 - y^2, for x >= 1, y >= 1.
    # Iterate over x and check y.
    for x from ceil(sqrt(N)) to infinity do
      ySquared := x^2 - N;
      y := isqrt(ySquared);
      if y*y=ySquared then        # If y is an integer, return factors.
        return x-y, x+y, iters;
      fi;
      if iters >= maxsteps then
        return 1, N, maxsteps;
      fi;
      iters := iters + 1;
    od;
  end;

> # Pollard rho factorization.  Same arguments as trial_factor.
> pollard_rho := proc(N, maxsteps)
    local xi, x2i, f, iters, p;
    # f(x) function iterated in Pollard rho method, we use f(x) = x^2+1.
    f := proc(x)
      return modp(x * x + 1, N);
    end;
    iters := 1;
    # Initial values for x(i) and x(2*i), where i=1.  We use x(1) = 2.
    xi := f(2);
    x2i := f(f(2));
    while true do
      # Compute p = gcd(x(i)-x(2*i), N).
      p := gcd(xi - x2i, N);
      # If p is a nontrivial factor, return factors.
      if p <> 1 and p <> N then
        return p, N/p, iters;
      fi;
      # Increase i by one.  Note we have to apply f twice to find
      # x(2*(i+1)) = f(f(x(2*i)).
      xi := f(xi);
      x2i := f(f(x2i));
```

```
      # Increment iteration counter.
      iters := iters + 1;
      if iters >= maxsteps then
        return 1, N, maxsteps;
      fi;
    od;
  end;

> # Pollard p-1 factorization.  Same arguments as trial_factor.
> pollard_p1 := proc(N, maxsteps)
    local two_k_fact, p, k, iters;
    two_k_fact := 2^(1);          # 2^(k!) for (initially) k = 1.
    iters := 1;                   # Number of iterations made through for loop.
    for k from 2 to infinity do
      # Compute p = gcd(2^(k!)-1, N) for current k value.
      p := gcd(two_k_fact - 1, N);
      # If p is a nontrivial factor, return factors.
      if p <> 1 and p <> N then
        return p, N/p, iters;
      fi;
      # Find 2^((k+1)!) = (2^(k!)) ^ (k+1).
      two_k_fact := two_k_fact &^ (k+1) mod N;
      # Increment number of iterations.
      iters := iters + 1;
      if iters >= maxsteps then
        return 1, N, maxsteps;
      fi;
    od;
  end;

> # Brent factorization.  Same arguments and return value as trial_factor.
> brent_factor := proc(N, maxsteps)
    local xi, x2i, f, iters, p;
    # f(x) function iterated in Pollard rho method, we use f(x) = x^2+1.
    f := proc(x)
      return modp(x * x + 1, N);
    end;
    iters := 1;
    # Initial values for x(i) and x(m), where i=1.
    xi := f(2);
    xm := 2;
    while true do
      # Compute p = gcd(x(i)-x(m), N).
      p := gcd(xi - xm, N);
      # If p is a nontrivial factor, return factors.
      if p <> 1 and p <> N then
        return p, N/p, iters;
      fi;
      # Increase i by one.  Update x(m) as needed.
      if 2^ilog2(iters) = iters then
        xm := xi;
      fi;
      xi := f(xi);
      # Increment iteration counter.
      iters := iters + 1;
      if iters >= maxsteps then
        return 1, N, maxsteps;
      fi;
    od;
  end;

> # Given 'algo', which should be one of the factorization functions
  # defined above, and k, returns the median time to factor the product
```

```
  # of two randomly selected k-digit primes, over 100 runs of the algorithm.
> median_steps_for_k_digit_prime := proc(algo, k)
    local i, times, p, q, p1, p2, iter;
    # Initially empty seqence of the number of steps made by the given algo
    # for each pair of random primes.
    times := seq(j, j=0..-1);
    # Run the algorithm 100 times on products of two random k-digit primes.
    for i from 1 to 100 do
      while 1=1 do
        p := rand(10^(k-1)..10^k-1)();
        q := rand(10^(k-1)..10^k-1)();
        if isprime(p) and isprime(q) and p <> q then
          break;
        fi;
      od;
      # Run the algorithm, but bail out after 1e6 steps.
      p1, p2, iter := algo(p*q, 1000000);
      times := times, iter;
    od;
    times := sort([times]);
    return times[1+floor(nops(times)/2)];
  end;

> # Reproduce the median number of steps for each algorithm when
> # given the products of two randomly selected 4-digit primes.
>
> # Vary the last argument to reproduce the data in Figure 1.
>
> median_steps_for_k_digit_prime(trial_factor, 4);
                                 3342
> median_steps_for_k_digit_prime(fermat_factor, 4);
                                 101
> median_steps_for_k_digit_prime(pollard_rho, 4);
                                 40
> median_steps_for_k_digit_prime(pollard_p1, 4);
                                 36
> median_steps_for_k_digit_prime(brent_factor, 4);
                                 97
```

Appendix B. References

[1]. Kalisky, Burt. "RSA Factoring Challenge." USENET newsgroup sci.crypto. March 18, 1991. Available: http://www.google.com/groups?selm=BURT.91Mar18092126%40chirality.rsa.com , Accessed November 17, 2004.

[2]. "General number field sieve." From Wikipedia, an online encyclopedia. November 13, 2004. Available: http://en.wikipedia.org/wiki/GNFS

[3]. Wesstein, Eric W. "RSA Encryption." From Mathworld, an online encyclopedia. April, 2001. Available: http://mathworld.wolfram.com/RSAEncryption.html

[4]. Junod, Pascal. "Cryptographic Secure Pseudo-Random Bits Generation: The Blum-Blum-Shub Generator." August 1999. Available: http://www.win.tue.nl/~henkvt/boneh-bbs.pdf

[5]. Housley et al. "RFC 2459: Internet X.509 Public Key Infrastructure Certificate and CRL Profile." January, 1999. Available: http://www.faqs.org/rfcs/rfc2459.html

[6]. "Integer factorization – Difficulty and complexity." From Wikipedia, an online encyclopedia. October 30, 2004. Available: http://en.wikipedia.org/wiki/Integer_factorization

[7]. Weisstein, Eric W. "Fermat, Pierre de." From MathWorld, an online encyclopedia. Available: http://scienceworld.wolfram.com/biography/Fermat.html

[8]. Weisstein, Eric W. "Pollard Rho Factorization." From MathWorld, an online encyclopedia. December 28, 2002. Available: http://mathworld.wolfram.com/PollardRhoFactorizationMethod.html

[9]. Weisstein, Eric W. "Brent's Factorization Method." From MathWorld, an online encyclopedia. December 28, 2002. Available: http://mathworld.wolfram.com/BrentsFactorizationMethod.html

[10]. Ohannessian, Robert J. "Bob's page of mildly useful but still pretty neat code snippets." February 18, 2003. Available: http://bob.allegronetwork.com/prog/tricks.html

[11]. Weisstein, Eric W. "Pollard Rho Factorization." From MathWorld, an online encyclopedia. December 28, 2002. Available: http://mathworld.wolfram.com/Pollardp-1FactorizationMethod.html

[12]. Campbell, Robert. "Computation – Exponentiation via the Russian Peasant Algorithm." March 29, 1998. Available: http://www.math.umbc.edu/%7Ecampbell/Math413Fall98/7-FermatThm.html

[13]. Lipson, John D. "Newton's method: a great algebraic algorithm." 1976. Available: http://portal.acm.org/citation.cfm?id=806344