

Runtime and Implementation of Factoring Algorithms: A Comparison

Justin Moore
CSC290 Cryptology
December 20, 2003

Abstract

Factoring composite numbers is not an “easy” task. It is classified as a “hard” algorithm, which is why the cryptosystems that are based upon its “hardness,” such as RSA, are very secure. Despite this, factoring algorithms are still not optimal and can still be optimized by a constant—and sometimes linear—factor. There are many ways to do this by utilizing different classes of algorithms. The most well known set of classes includes trial division, probabilistic methods, geometric methods, and number sieving. Each class attacks the problem of factorization a different way, producing at least one factor—sometimes non-deterministically—for each run of the algorithm. This paper outlines the methods used to implement each of five algorithms and determines their resulting runtime, which is then further compared to the ease of implementation. In the end a probabilistic algorithm—Pollard’s Rho—was deemed the best algorithm.

Background and Motivation

Factoring numbers is what is called a “hard” task, which means that any algorithm designed to factor will not run in polynomial time. In fact, most of the algorithms that exist today run on the order of e^n , where e is Euler’s number [6].

Since prime numbers can be found in polynomial time, and multiplication can also be done in polynomial time, the combination of these two creates a polynomial time algorithm that can generate large composite numbers with large prime factors. Because these large composite numbers can be generated “easily” and it is “hard” to factor them again, public key cryptosystems can be based on publishing the large composite number and having a private key that is somehow based upon the factorization of the public number. In particular, RSA has created a very popular version, where a user’s private key is Euler’s Phi function of the composite number.

Mainly due to the popularity of the RSA cryptosystem, many different methods for factoring composite numbers have been developed¹. It is useful to step back to take a look at the larger picture.

Project Goals

The comparison of factoring algorithms has already been done many times over and is a hotspot in research because of the popularity of the RSA cryptosystem [4].

¹ It is important to note that factoring large composite numbers is not necessarily an equivalent problem to breaking RSA. Mathematicians are divided on the open problem of whether or not the two are equivalent, but it is thought that, although it is still a “hard” problem, it is possible to break RSA without factorization. This is beyond the scope of this project, however, because there are no known methods for breaking RSA without factoring the modulus [3].

Therefore, although the main goal of this project is to compare factoring algorithms, this project also focuses on the implementation of algorithms.

An inverse relationship appears to exist between ease of implementation of any factoring algorithm and its respective runtime, although there are some exceptions. Thus, like any software engineering project, it is necessary to weigh the costs and the benefits of each side. Although some factorization algorithms are comparatively very fast, the time it takes to implement them may outweigh any speed-up in runtime. This comparison is a key factor in determining a good algorithm.

When runtime is the only factor being measured, there is a choice of whether to use an existing algorithm or to generate an implementation; either based on pseudo-code or from scratch. There is a tradeoff between the two choices. Using existing code guarantees a much faster result (since optimizations are built in) and—to a larger degree—a working solution. However, new implementations allow for more equality between each algorithm since they are being generated by the same person.

Since one of the goals of this project is to determine the tradeoffs between implementation and algorithm speed, the choice becomes moot. Thus, the project involves creating implementations of each algorithm for testing purposes instead of using existing algorithms.

Environment

All code was written in Java Standard Edition version 1.4.2. Programs were run on a machine running Linux with dual Pentium III 1133MHz processors.

Java is one of the best languages for this task because of its modularity and built in libraries. For each algorithm to be equal in structure, a class was defined for each that implemented a standardized interface. Thus, each implementation was a module that could be run and timed by a separate application; this is easy in Java because of its object-orientation and its type casting. Java also offers many libraries, in particular, the BigInteger class that handles infinite precision arithmetic, which is necessary for the large numbers that are involved in this project.

The only major negative aspect of using Java is its runtime overhead. Also, Java programs tend to run slower by a constant factor than, for example, C programs. These would be a problem if the purpose of this project were to measure absolute runtime, but since the different algorithms are being tested relative to each other the only difference that needs to be measured is the ratio of their results. Thus, since the measure of runtime by number does not count any overhead and taking a ratio removes constant factors from each term, the measurements are as accurate as they would be when written in a similar, faster language without overhead.

The hardware to run the programs was chosen because it was simply the most convenient, fastest computer available. Since all of the algorithms use integer operations, there is no disparity between integer-operations and floating-point operations that would make the results any different on any other processor (e.g. sparc). Also, Java automatically uses both processors to a large degree of efficiency, removing any disparity between program parallelisms.

Methods

For this project, four classes of factoring algorithms were chosen. These are: trial division, probabilistic, geometric, and number sieves. These are not strict categories; in fact, many of the algorithms are in multiple categories. Each class represents a different angle at which to attack the problem of factoring. Therefore, some algorithms will be better at factoring different numbers, depending on their prime decomposition. For each of the descriptions, the following terms will be used: modulus or composite (refers to the composite number to be factored; its name comes from RSA where the public key is the composite number, and a message is computed by exponentiating it and then reducing it modulo the composite); probabilistic (something that works with a probability less than one); deterministic (something that works with probability one).

Trial Division

Often called the naïve method of factoring, trial division works by dividing the modulus by numbers between two and its square root. If there is no remainder, the number is a factor of the modulus. This method is good for relatively small moduli, usually below ten thousand or up to one million on a very fast machine.

The simplest version of this algorithm is brute force trial division, where every number between two and the square root of the modulus is tried. However, this can be optimized easily with certain heuristics such as: only trying odd numbers, only using prime numbers, or using a smaller set of randomly chosen prime numbers. The last two heuristics are encompassed by the number sieving and probabilistic classes (respectively) as well, both of which will be discussed later.

The main deficiency of this algorithm is its speed. Without any heuristics, this algorithm requires the maximum number of operations for factorization and uses division, an operation that is very costly on any computer architecture. Even when optimized, division is still used, which means that the algorithm will still be slow because of its dependence on division.

Probabilistic Algorithms

Probabilistic algorithms work on the principal that not finding a factor with a probability greater than zero will take a fraction of the time it would take to find a factor with complete certainty [9]. Therefore, every time the algorithm is run, it multiplies the overall certainty of not finding a factor by the certainty of running it once, decreasing the certainty even more.

For some probability p of not finding a factor, if it takes n/m time to run—where n is the runtime of the deterministic version and m is some constant—then if the algorithm is run $m/2$ times, it will find a factor in $n/2$ time with probability $1-p^{m/2}$. If p is somewhat small and m is somewhat large, after $m/2$ operations or m/c operations where c is bigger than two, it is possible to get a speedup of some constant when the algorithm works with certainty that is very high.

There are various methods for speeding up algorithms using a probabilistic approach. One way is to pick a number of “witnesses” that verify the correctness of an algorithm, where each witness has a certain probability of being correct. Another method is to pick random numbers or a random number generator function and run tests using a random subset of the values used in the deterministic algorithm.

There are two main weaknesses to the probabilistic approach. First, it is not deterministic, so it will not work all of the time. This is acceptable because the probability of failure can be set to whatever tolerance is allowable. Also, it is almost impossible to determine what parameters are optimal for this algorithm. If there are multiple parts of the algorithm that are probabilistic, determining a good balance between them is very hard. Also, when comparing to deterministic algorithms, it is difficult to give a value to a probabilistic algorithm that fails to find a factor for a certain run.

Geometric Algorithms

This is a large class of algorithms based on a mixture of algebra, number theory, and geometry. The premise behind the algorithm is that some functions under the real numbers reduced modulo a number can form a group under an operation as long as there is closure, an identity element, and inverse elements for each member. Certain functions, such as the ray function and the function that measures elliptical curve arc-length are groups under certain operations [5]. If an element of these groups is found that generates a subgroup, the order of that element is the order of the subgroup, which divides the order of the entire group, the modulus. Thus, for every subgroup generator found, a factor is determined.

The biggest problem with geometric algorithms is that they are difficult to implement, especially in an optimal way. Since the group operation requires finding a point on the graph given two previous points, it is usually necessary to do some complicated calculations based upon high order functions. Eliminating code redundancy and optimizing memory access for calculations like this has been made into an entire field of research because of its difficulty. Despite this, optimized geometric algorithms are some of the fastest known factoring methods.

Number Sieving

Number sieving is based upon the elimination of multiple numbers for each factor that is tried. If a number is not a factor of the modulus, it implies that other numbers are not factors either; for each non-factor found, other non-factors can be eliminated in some way [10]. The sieve of Eratosthenes and the quadratic sieve do this using linear and quadratic functions of non-factors, respectively. This cuts down the number of trials by a linear—or even factorial—amount.

Number sieving has a lot of solutions that are only somewhat fast. It also contains the fastest known algorithm, the number field sieve. This algorithm works by taking the roots to different polynomials and their corresponding rings with integer coefficients and determines a set of numbers to remove because of their membership in the ring. It is very difficult to implement and has still not been proven to work for all numbers. Thus, the only truly efficient number sieve is not one that can be readily used.

Algorithms Implemented

Aside from the baseline trial division algorithm optimized by only trying odd numbers, four other algorithms were implemented: trial division by random prime numbers, Pollard's Rho method, Lenstra's elliptical curve algorithm, and the quadratic sieve algorithm. This set of algorithms represents at least one representative from each

class of algorithm; although most are in multiple classes, the overall set covers the entire range of modern factorization methods.

Random Prime Division

The random prime division method is a combination of probabilistic and trial division. It is basically an optimization on the standard trial division algorithm. It works by choosing a prime number at random between two and the square root of the modulus. This number is divided into the modulus and if the remainder is zero, it is saved as a factor. This factor and its corresponding factor (the result after the modulus is divided by it) are then factored using the same method. Thus, if there are p primes between two and the square root of the modulus, and the program is run p/c times for some constant c , it will work with probability $1/c$. A value of two for c was used in this project, giving a probability of .5 that a factor would be found.

Pollard's Rho Algorithm

Another probabilistic algorithm, Pollard's Rho, works by successive iterations of a random quadratic function. Random coefficients are chosen for a standard quadratic function that generates numbers, which are reduced modulo the composite number. Successive iterations of this function based upon a randomly chosen initial number generate a sequence that starts looping after a certain point². If two points are on the same position of the loop, they are congruent to each other modulo a factor of the composite. This is because the loop is actually a subgroup generated by the initial element as the identity and with the random function as the group operation. Thus, if two points are in the same equivalence class in the subgroup, they are equivalent to each other modulo the order of the subgroup, which divides the order of the entire group, which is the composite number. Subtraction yields a multiple of the order of the subgroup and taking the greatest common divisor of this and the modulus yields a factor of the modulus.

It is important to note that not every function and initial point pair will yield a subgroup. This algorithm is probabilistic and thus does not find a factor every time. However, bounds can be placed on it that forces a restart after a timeout period. If the algorithm works the first time, it is very fast; at worst case it runs at the same speed as trial division does.

Lenstra's Elliptical Curve Algorithm

Lenstra's elliptical curve algorithm is the most common geometric algorithm [1, 7]. It is based on the function $y^2=ax^3+b$, which is a function of the arc length of an ellipse. When the operation $*$ is defined on two points such that $(x_1, y_1)*(x_2, y_2) = (x_3, -y_3)$, where the point (x_3, y_3) is the third point on the line that goes through the points (x_1, y_1) and (x_2, y_2) ³. Inverses are defined as $(x, y)*(x, -y) = \infty$, where infinity is the point that all vertical lines converge and the identity of the group [8]. With a starting point and successive

² This is the reason that the algorithm has its name; the sequence has a tail and then starts to loop, much like the Greek Rho.

³ For every elliptical curve, any line will intersect the curve at three points, if it intersects at a tangent point, that point is counted twice.

iterations of the group operation, some point will eventually be a generator of a subgroup, and its order will be a factor of the modulus.

Thus, the optimal way to find a factor of the modulus is to take two initial points and successively operate on them in increasing numbers, so the k^{th} iteration will have k factorial iterations of the group operation. Thus, if the current term is a generator of a subgroup, the greatest common denominator of k factorial and the modulus will be the order of the element.

The worst part about this algorithm is that, like most other geometric algorithms, it is hard to implement, especially in an efficient manner. When fully optimized, elliptical curve algorithms are the second fastest known factoring algorithm, but these optimization techniques are very difficult and are often very obscure.

Quadratic Sieve

Although it is not the fastest number sieve, the quadratic sieve is the fastest sieve that has been proven to work. It works by the principle that, if two numbers are equivalent to each other modulo the composite number, their sum and difference are two factors of the modulus⁴. There are many different ways of choosing the numbers to compare; the best method is to find a set of B-smooth⁵ numbers for some small bound B. Finding a set of numbers that, when their squares are multiplied, have a prime decomposition with only even powers will give two square numbers: the prime decomposition and the product of the squared numbers. If there are k numbers that are prime less than B, with $k + 1$ values, it is possible to find a solution where the vector of powers of their primes sum to the zero vector mod two using Gaussian elimination.

The quadratic sieve will not always find a factor. The algorithm chooses random numbers that are close to, but greater than, the square root of the modulus. A failure occurs when either the matrix produced by the B-smooth numbers finds the trivial and improper factors, or it is irreducible. Also, the matrix reduction involved in this method is a very expensive operation that is difficult to optimize.

Methods

Each algorithm was run with numbers of differing complexity⁶ and length. All algorithms were run as a module of a Java application, which ran as a command line program that timed all classes that implemented the *Factorer* interface. Each algorithm was timed using the difference of the system time (in milliseconds) taken before and after execution. For each number, five runs were done and the average time was recorded.

The measure for ease of implementation was setting an implementation time bound. Once knowledge of an algorithm's purpose, function, and optimal implementation were known (including pseudo-code references), a time bound of six hours of design, coding, and testing per algorithm was allotted.

Results

Of the five algorithms, only three were compared. These were the naïve trial division algorithm, the random prime trial division algorithm, and Pollard's Rho

⁴ This is true because of a simple algebraic identity: $a^2 - b^2 = (a + b) * (a - b)$.

⁵ A number that is B-smooth only has prime factors less than or equal to B.

⁶ Complexity refers to the number of factors that are in a composite's decomposition.

algorithm. The other two, quadratic sieve and elliptical curve algorithms, were not compared because of insufficient optimization and incompleteness, respectively. The quadratic sieve was easy to implement in a few hours worth of work; however, sufficiently optimizing Gaussian elimination is a very difficult task. The overhead incurred by this expensive operation was reduced by 500ms, but that was only half the

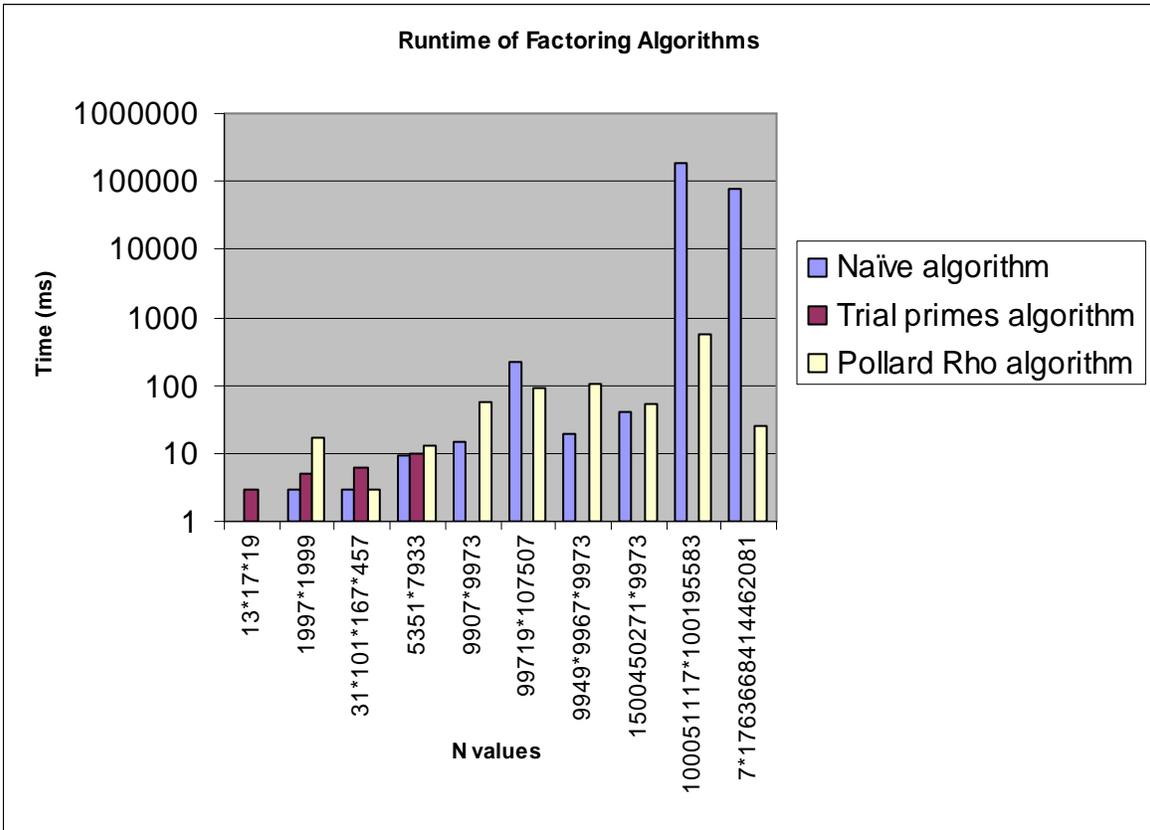


Figure 1

elliptical curve algorithm was a product of learning from the mistakes of the quadratic sieve. Instead of implementing and then optimizing, all the work was put into implementing an optimized version; this proved to be too difficult. The elliptical curve algorithm takes very advanced mathematics to understand, let alone to optimize. Even if a non-optimized version were implemented, the result would have been similar to the Quadratic Sieve. These two algorithms were compared to the other algorithms only on their difficulty of implementation, which was assigned an “infinite” value.

The remaining three algorithms were compared using numbers that ranged from 13*17*19 to 7*1763668414462081. Since the probabilistic trial division algorithm was only useful for comparison to the naïve algorithm, and its parameters allowed for only a 50% chance of success, when it showed a similar pattern to the naïve version, it was abandoned.

The two remaining algorithms—naïve trial division and Pollard’s Rho algorithm—took thirty minutes and two hours to implement, respectively⁷. Figure 1 shows the graphs of the runtimes of each algorithm as a function of the composite number each was given. The best measure of each algorithm is a linear regression, each of which were created using GraphPad Prizm, a statistical program. The graph was generated using Microsoft Excel. The two regressions had an inverse slope (number / milliseconds) of $8.9e10$ and $5.2e13$ which means that Pollard’s Rho algorithm is about 23% faster than the naïve method. This means that the difference in time of an hour and a half would be surpassed for numbers with approximately twenty digits, making the implementation of Pollard’s Rho worth it for these numbers.

Conclusions

In terms of difficulty, only three of five algorithms were easy enough to implement so that they were efficient enough to compete with naïve trial division. The speed that would have been gained, even if it were a factor of 300% or 400%--which is unlikely since Pollard’s Rho only worked at 130%--would make up for the implementation time with only huge numbers (200 to 250 digits).

Of the three remaining algorithms, naïve and probabilistic trial division are basically the same and are good for smaller numbers (less than sixteen digits). For any numbers over this, the runtime gets large very quickly. This leaves Pollard’s Rho algorithm, which is the best factoring algorithm in terms of ease of implementation and power. With a reasonable amount of implementation time, Pollard’s Rho factors a large range of composite numbers in a reasonable amount of time.

Acknowledgements

This project is supported by the cryptology and specifically RSA cryptosystems teachings offered by Professor Chris Brown’s cryptology course at the University of Rochester that was supplemented by the text *Making, Breaking Codes*, written by Paul Garrett. In addition, all works referenced in this paper played an integral part in the understanding and continuation of analysis of composite number factoring methods.

⁷ Times are approximate since there is very low accuracy in timing human efforts.

References

- [1] Atkin, A. O. L. and F. Morain. Elliptic curves and primality proving. *Math. Comp.*, 61(203):29--68, July 1993.
- [2] Bautista, Lopez. Factoring Integers. XXXI National Congress of the Mexican Mathematical Society: 243-269, 1999. (Translated from Spanish).
- [3] Boneh, Dan and Ramarathnam Venkatesan. Breaking RSA may not be equivalent to factoring. *Advances in cryptology---EUROCRYPT '98 (Espoo)*: 59-71, *Lecture Notes in Comput. Sci.*, 1403, Springer, Berlin, 1998.
- [4] Coutinho, S. C. The mathematics of ciphers. *Number theory and RSA cryptography*. A K Peters, Ltd., Natick, MA, 1999. (Translated from Portuguese)
- [5] de Vries, Andreas. The ray attack, an inefficient trial to break RSA cations cryptosystems. FH Sudwestfalen University of Applied Sciences, Haldener Straße: 182.
- [6] Diffie, W. and M. E. Hellman. New directions in cryptography. *IEEE Trans. Inform. Theory*, 22 (6): 644-654, 1976.
- [7] Lenstra, H. W.. Factoring integers with elliptic curves. *Annals of Math*, 126: 649-673, 1987.
- [8] Morain, F. Primality proving using elliptic curves: an update. In J. P. Buhler, editor, *Algorithmic Number Theory*, volume 1423 of *Lecture Notes in Comput. Sci.*, pages 111--127. Springer-Verlag, 1998. Third International Symposium, ANTS-III, Portland, Oregon, june 1998, Proceedings.
- [9] Bach E. and V. Shoup. "Factoring polynomials using fewer random bits," *Computer Sciences Technical Report No. 757*, University of Wisconsin--Madison; *Journal of Symbolic Computation*, to appear (1988).
- [10] Duntun, B., Jones, J., Sorenson, J. P.: A space-efficient fast prime number sieve. *Information Processing Letters* 59 (1996) 79—84.