|  |
| --- |
| RSA Cryptographic Algorithm and Integer Factorization Algorithms |

|  |
| --- |
| Yu Feng Chen  January 19th, 2017 |

**Research Question**

What are some of the methods or algorithms of integer factorization that currently exist? How well do they perform relative to each other, in term of time consummation or the size of the integer being factored?

**Introduction**

Over the past few decades, technology has been advancing in a very fast pace. Technological advancement leads to development of many electronic devices that take part in our daily life. Internet had been established and many things could be performing more easily online or by the devices. These devices not only are used to store information, but also used to message, to email, to do shopping, and even to perform payment online. However, had you ever wondered how the information is being kept safe? How do people prevent others from stealing your information?

There are many reasons why information or messages need to be kept in private, so we need a system to perform this protection. This led to the development of systems for transforming text to conceal its content, which is called cryptography. Today, one of the most widely used cryptographic systems for protection is the RSA cryptography. The idea is to have an encryption scheme in which the public knows how to encrypt messages but not how to decrypt them. This resembles a trapdoor, which is easy for one to go in one direction, but is difficult to go through in the opposite direction. The RSA encryption algorithm uses the mathematical trapdoor based on factorization. The key used to encrypt involve the use of a product of two very large prime numbers, while the key to decrypt involves the use of the two factors. So, one could say solving RSA is as easy as factoring.

However, just how “easy” is it to factor an integer? Suppose you were asked to multiply 53549 by 89561 using pencil and paper, you could get to the answer, 4795901989, in a few minutes. But what if you were given this product, and ask to factor instead. Certainly, it would take much longer to factor this number by hand.

When asking a person to factor a number, most of them would likely, including myself, just start thinking of numbers that divides into the given number, going from 2, 3, 5, 7… and on and on, until one that does divides is found. This would take a long time to perform when comes to factor huge numbers, such as those of 600 digits used in the RSA cryptography. So, are there other methods of integer factorization that we could use? This is basically how I came to my research question, from RSA cryptography into factorization, and the interest of wanting to know more about factorization lead to the decision on this topic. The goal of the investigation is to expand the knowledge on the field of integer factorization, how do the different methods work and how well do they perform; lastly, it might interests someone to make improvements upon those existed methods of factorization.

**History**

With the invention of the computer in the middle of the 20th century, both code making and code breaking became more complex. Encryption systems could now employ enormous arithmetical calculations. Not only the government and military, businesses also began to use encryption for many purposes. In the early 1970s, IBM developed the Data Encryption Standard (DES). It is a standard method of encryption that all companies could use. In 1976, DES officially became the national standard, and the National Security Agency (NSA) restricted keys to be less than 256. However, there was a problem when two parties arranging a key to use. The *key distribution problem* was solved in 1976 by Whitfield Diffie, Martin Hellman, and Ralph Merkle. The solution is known as the *Diffie-Hellman key exchange*.

In 1975, Diffie and Hellman published a paper explaining their idea for public key cryptography, which is a system that involves the use of asymmetric keys. However, their method was still missing an encryption function that could be used to perform. Later, three computer scientists at MIT, Ron Rivest, Adi Shamir, and Leonard Adleman, had figured out a function that could only be reversed by someone who knew the private key, which is now known as the RSA Encryption Algorithm. They later found a company call RSA Data Security, Inc., devoted to protecting online identities and digital assets.

There is another side to this history. Clifford Cocks, an English mathematician working for the UK intelligence agency Government Communications Headquarters (GCHQ), described an equivalent system in an internal document in 1973. However, given the relatively expensive computers needing to implement it at the time, it was mostly considered a curiosity and was never deployed. His discovery, however, was not revealed until 1997 due to its top-secret classification.

When it comes to integer factorization, many methods of integer factorization had been developed by mathematicians throughout the history. Besides the Trial Division method that is known by most people, another factorization algorithm is the Fermat's factorization method, which is named after Pierre de Fermat, a mathematician of the 17th century. Later in 1975, Daniel Shank made an improvement upon Fermat’s factorization method, now known as the Shanks' square forms factorization. John Pollard had invented the Pollard's *p* − 1 algorithm and Pollard's rho algorithm in 1974 and in 1975 respectively. In 1980, Richard Brent published a faster variant of the rho algorithm. Hugh C. Williams invented the Williams' *p* + 1 algorithm in 1982. There are many other more algorithms that are not mentioned here. But when it comes to factoring RSA numbers, a general-purpose factoring algorithm such as Quadratic sieve and General number field sieve is being used.

**Mathematical Background**

Solving RSA is as “easy” as factoring, this is because the RSA encryption algorithm is based on the mathematical trapdoor that it’s difficult to factor very large numbers. The RSA algorithm is an exponentiation cipher using a composite modulus, and the modulus used is a product of two large prime numbers. The RSA encryption algorithm consists of a setup, the key generalization, encryption and decryption.

First, choose two large prime numbers *p* and *q*, and let By Euler's totient function, or sometimes called Euler's phi function:

In number theory, Euler's totient function calculates the positive integers up to a given integer *n* that are relatively prime to *n*. It could be defined for any natural number *n* as follows:

For example, to . We must determine how many nonnegative integers less than 10 are relatively prime to 10. The numbers that satisfy this are 1, 3, 7, and 9. Hence, = 4.

If you try to find the totatives of a prime number *p*, then it will be one less than that prime number. Since *p* is prime, every nonzero element of is a reduced residue. These cases are defined by the following lemma:

*Let .*

Another lemma states that, *let p and q be distinct primes. Then*

To proof this, consider the set *S* of all nonnegative integers less than *pq*:

By definition, equals the number of elements of S that are relatively prime to *pq*. Every element of *S* that is not relatively prime to *pq* must be a multiple of *p* or a multiple of *q*. In the set *S*, the multiple of *q* are

While the multiples of *p* in the set *S* are

Notice that *S* contains exactly *p* multiples of *q* and *q* multiples of *p*. There is only 1 number, the number zero that is contained in both lists. Therefore, the total number of multiples of *p* or *q* in *S* is Since there are a total of *pq* elements of *S*, the number of elements of *S* that are relatively prime to *pq* is given by

Going back to the RSA algorithm, let , and Next, choose an encryption exponent, such that The decryption exponent, , is the multiplicative inverse of :

With .

The following chart will show the information involved with the private key and the public key generation:

|  |  |
| --- | --- |
| Private  (Known only by person who is decrypting) | Public  (Known by person who is encrypting) |
| Large prime numbers *p* and *q*  Decryption exponent *d* | The modulus *n* (where )  Encryption exponent *e* |

To encrypt, let be the plain text message we wish to encrypt and. The cipher text, *c*, is obtained by raising the message to the encryption exponent modulo *n*:

Modular exponentiation is a type of exponentiation performed over a modulus. The operation of modular exponentiation calculates the remainder when an integer *b* (the base) raised to the *e* th power (the exponent), , is divided by a positive integer *m* (the modulus). In symbols, given base *b*, exponent *e*, and modulus *m*, the modular exponentiation *c* is:

.

To decrypt, or to recover the plain text, raise the ciphertext to the decryption exponent *d* modulo *n*:

Here is an example of RSA encryption:

Encrypt the message FERMAT in two-letter blocks using the RSA algorithm, with modulus *n* = 3233 and encryption exponent . First, we need to separate the message into two-letter blocks and convert each two-letter block into a number. In this case, correspond the letters of the alphabet with numbers in order, such that , , and so on:

To encrypt the message, we raise each of these plaintext numbers to the power 7 modulo 3233. We then find

Thus, the ciphertext is 1663 2410 55.

Here is an example of RSA decryption:

Decrypt the message from the ciphertext that reads 1930 208 1910, which is a message that was encrypted in two-letter blocks with RSA using the values and To do this, you would raise each of the numbers in the ciphertext to the power *d* and reduce modulo 3901. However, you do not know the decryption exponent *d*, but you do know that *d* is the multiplicative inverse of Thus, before you find *d*, you need to calculate the value of . The best way to do this is to first factor 3901 as We then have

We now know that the decryption exponent *d* is the multiplicative inverse of 343 modulo 3772. Using the Euclidean Algorithm to find this inverse, we get

Now raise each of the ciphertext numbers to the power 11 modulo 3901, we find

Thus, the plaintext is 521 1205 1827. Covert this into letters reveals the decrypted message: EULER.

Note the step when you try to find *d* and try to calculate the value of , you had to factor the value *n* into factors. This step would be prohibitively time consuming if the number to be factored were large. After you solve for *d*, the rest of the decryption process will be easy. Thus, the hardest part, which is the trapdoor, is to factor the number *n*.

There are many methods and algorithms developed by mathematicians that involve integer factorization. One of the methods that most of us are familiar with is the Trial Division. The essential idea behind trial division tests is to see if an integer *n*, the integer to be factored, can be divided by each number in turn that is less than *n*. However, the trial factors need go no further than because, if n is divisible by some number *p*, then and if *q* were smaller than *p*, *n* would have earlier been detected as being divisible by *q* or a prime factor of *q*.

For example, for the integer *n* = 35, the only numbers that divide it are 1, 5, 7, 35. Using trial division, you would divide 35 by all integers up to , which is about 5.916, so you just need to divide 35 by 1, 2, 3, 4, 5. You will find out that 5 divides 35 and you use that number to find out the other factors.

Another integer factorization algorithm is the Pollard's rho algorithm. It is a probabilistic method for factoring a composite number *N* by iterating a polynomial modulo *N*. It is based on Floyd’s cycle-finding algorithm and on the observation that (as in the birthday problem) *t* random numbers , , ..., in the range [1, n] will contain a repetition with probability *P* > 0.5 if .

There are two aspects to the Pollard rho algorithm. The first is the idea of iterating a formula until it falls into a cycle. Let , where *n* is the number to be factored and *p* and *q* are its unknown prime factors. Iterating the formula

,

or almost any polynomial formula (an exception being ) for any initial value will produce a sequence of number that eventually fall into a cycle. The expected time until the become cyclic and the expected length of the cycle are both proportional to .

However, since with *p* and *q* relatively prime, the Chinese remainder theorem guarantees that each value of *x* (mod *n*) corresponds uniquely to the pair of values (*x* (mod *p*)), (*x* (mod *q*)). Furthermore, the sequence of follows the same formula modulo *p* and *q*, i.e.,

.

Therefore, the sequence (mod *p*) will fall into a much shorter cycle of length on the order of . It can be directly verified that two values and have the same value (mod *p*), by computing

,

which is equal to *p*.

The second part of Pollard's method concerns detection of the fact that a sequence has become periodic, which is based on Floyd’s cycle-finding algorithm. However, Brent later improved Pollard’s rho factorization method, by replacing with his cycle detection method.

Cycle detection is the algorithmic problem of finding a cycle in a sequence of iterated function values. Floyd's cycle-finding algorithm is a pointer algorithm that uses only two pointers, which move through the sequence at different speeds. It is also called the "tortoise and the hare algorithm", based on the fable of the race between the tortoise and the hare. Note that there are often two stages to a cycle finding algorithm; first, you detect a cycle (by finding two indices such that ), and then, perhaps, you may want to find minimal values such that for all natural numbers *k* and .

For the detection part, the strategy is to first set off the 'hare', which will run through the data quickly, and behind him set off the 'tortoise', who will examine the data more slowly. A cycle is detected if the tortoise and the hare ever agree.

Floyd's algorithm start the tortoise and the hare off at position zero, then at every iteration, let the tortoise advance one step, and the hare advance two steps. At each iteration, you compare and . If a cycle exists, then eventually you'll get to a value and the difference between hare and tortoise positions (which is also *i*) will be divisible by . At each step, you need to iterate the function three times; once for the tortoise, twice for the hare.

Brent's algorithm proceeds a little differently; this time, we let the tortoise sit at a power of 2, and the hare runs off to the next power of 2. At each inner iteration, the tortoise is at while the hare is at . Once the hare gets to , we reset the tortoise to that point. Again, if a cycle exists, we will eventually detect it. The benefits are this time, you only need to apply the function once at each step; furthermore, once , then the hare will find directly, rather than possibly a multiple. Even if you have to look further through the data, Brent's algorithm will require fewer functional evaluations than Floyd's.

Pollard's rho algorithm is also based on the observation made in the birthday problem, or the birthday paradox, that the probability of two persons having same birthday is unexpectedly high even for small set of people. In a set of *n* randomly chosen people, the probability that two people have the same birthday is 100% when the number of people reaches 367 (since there are only 366 possible birthdays, including February 29). However, probability is 99.9% with just 70 people, and 50% probability with 23 people. These conclusions are based on the assumption that each day of the year (except February 29) is equally probable for a birthday.

Now let *n* be a composite integer. Since *n* is composite, it has a non-trivial factor . Suppose we have to pick two numbers *x* and *y* from the range . The only time we get modulo *n* is when *x* and *y* are identical. However, based on the Birthday Paradox, since , there is a good chance modulo f even when *x* and *y* are not identical.

Another method of integer factorization is Fermat's factorization method. This method is based on the representation of an odd integer as the difference of two squares:

.

The difference could be factored into:

.

Each odd number has such a representation. If is a factorization of N, then:

.

Since *N* is odd, then *a* and *b* are also odd, so those halves are integers.

To find the factors, we had to try various values of *x*, such that . If is not a perfect square, then we had to increase the *x* value by 1 and try again.

For example, to factor , the first try for *x* is the square root of 5959, which rounded up to the next integer, 78. We then calculate:

Since 125 is not a perfect square, we then add one to *x*, and try for the new value of *x*.

In the second attempt, 282 is again not a perfect square, we then repeat the steps by increase the value *x* by 1 and try again.

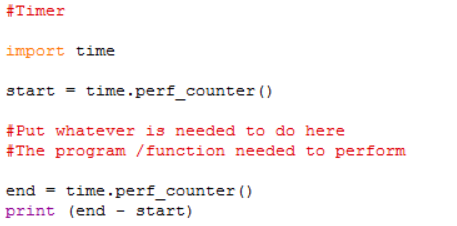
In the third try, we got a perfect square of 441. Thus, , and the factors of 5959 are and .

**Investigation**

In the mathematical background, we had gone over the details of four different integer factorization algorithms/methods, they are the Trial Division method, Pollard's rho algorithm, Brent’s improved version of Pollard rho factorization, and Fermat's factorization method. Thus, the investigation will be focusing on these four algorithms. But how do we compare the performance of these different methods to each other?

We could try to input a large number that is a product of two primes, and use each method to factor the number, the amount of time used to factor for each method could be the variable for the performance comparison. Of course, this could not be done by hands, thus, we need to write the algorithms into a program. The program will allow us to enter an input, the number being factored, and the program will output the factor(s) and the time consumed. In this case, all programs will be written in Python programing language (which is a widely used high-level programming language used for general-purpose programming, created by Guido van Rossum).

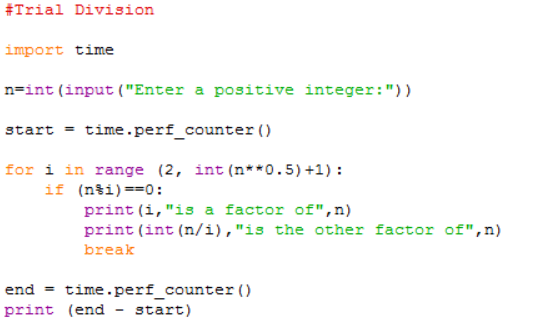
In order to know how much time it takes the algorithm to factor a given number, we first need to write a program for a timer. The program below is a timer program that could use to calculate the time a function performs. The timer will start right before the function, in this case the algorithms, and then stop once the performance is done, and it will output the time taken.



(A Python program for a timer)

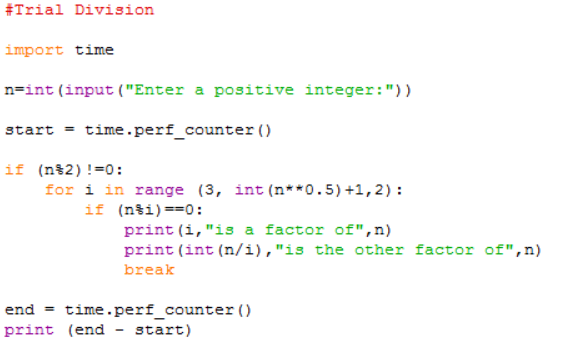
Trial Division:

The concept of this method is very simple, the program written will allow us to input a number, and then the program will divide every number, from 2, up until the square root of the inputted value. Once it found a number that could divide into the inputted value, it will stop and outputs the factors.



(A Python program for Trial Division method)

An improvement to the program could be made by excluding the factors of two, the evens. The program will first test if two divides the inputted value, if not, then it will test all the odd numbers up until the square root of the inputted value. The improved vision of the program is shown below:



(A Python program for improved vision of the Trial Division method)

The following table will show some of the value inputted and the amount of the time for each vision of the program to factor the number.

|  |  |  |  |
| --- | --- | --- | --- |
| Digits | Inputted/tested numbers | Time (sec.) – (Version 1, Version 2) | |
| 3 | 17\*43 | 0.017868627857 | 0.012984088506 |
| 6 | 571\*859 | 0.015147839883 | 0.01716936915 |
| 9 | 15817\*32257 | 0.019132429327 | 0.018518108825 |
| 12 | 443341\*753133 | 0.11229660263 | 0.061265334309 |
| 15 | 15485867\*32452843 | 3.06345673998 | 1.73147517513 |
| 18 | 373587883\*776531401 | 72.78719068666 | 35.8892010752 |
| 19 | 1611623773\*5112733757 | 569.9818520637 | 275.197504239 |
| 20 | 4666527007\*7367575799 | 1855.462699486 | 883.350360217 |
| 20 | 573259391\*24713066809 | 183.226902339 | 89.8261769275 |
| 21 | 17582163853\*38244924643 | 7068.657972418 | 3547.564940196 |
| 21 | 1611623773\*148780808401 | 534.6931834523 | 283.04478890288 |

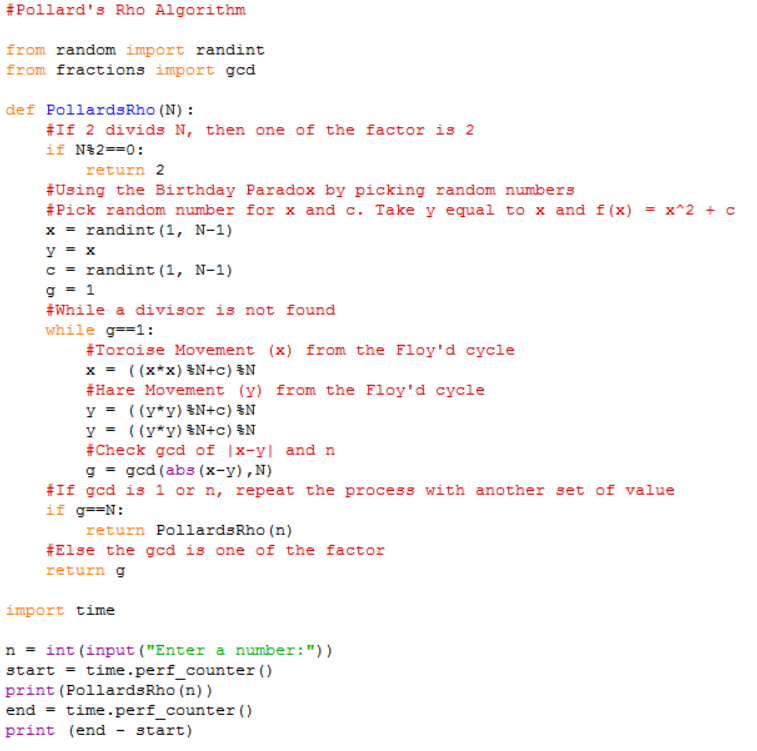
Base on the data that were collected in the table above, we could see that this method performs very fast for factoring small digit integers, in less than a second for a 12-digit number to be factored. As you increase the value, the run time for this method increases. When factoring a 15-digit number, it takes roughly 3 seconds to factor the number, but as you go further, factoring a 18-digit number takes a minute, a 19-digit number takes about 10 minutes, and when it comes to a 21-digit number, it takes about 2 hours to factor the number.

Even with the improved version, which on average takes half as much time to factor the numbers than the original version, it still takes a long time to factor the large integers. Thus, the Trial Division method is more suitable for factoring small integers, but will be slow to factor large integers.

Note that there are two different numbers tested for the 20-digit and 21-digit places, the only difference is that one number is a product of two primes that are close each other, while the number of the other trial is a product of two primes that is separated from each other by one or two digit places. This was done because it will help us see if the size of the factors affect the performance for the method. It turns out that it does make a difference. The number consists a smaller factor had a smaller run time. This makes sense since the program is running all the numbers from 2 up until the square root of the number being factored, so a number will be factored quicker if it has a smaller factor. Thus, if the number being factor had a small factor, then this method is suitable to be use, but we may not always know whether it consist a small factor or not.

Pollard's rho algorithm:

As a reminder, the concepts used in this algorithm include the observation of the birthday paradox and the use of Floyd’s cycle-finding algorithm. The program for this algorithm will be setup in the following way: First, setup a polynomial function for choosing random numbers (eg. Start with random *x* and *c*. Take *y* equal to *x* and ). Next, while a divisor isn’t obtained, we perform the cycle sequencing by moving *x* to and *y* to .Then we check the *gcd* for *(x – y)* and *n*. If *gcd* = 1 or *n*, return and repeat for another set of numbers, or else the *gcd* is one the factor. This factor then could help us find out the other factors. The program for Pollard's rho algorithm is shown below:



(A Python program for Pollard's rho algorithm)

The following table will show some of the value inputted and the amount of the time for each trial to factor the number.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Digits | Inputted/tested numbers | Time (sec.) – For 3 individual trials. | | |
| 3 | 17\*43 | 0.0033142 | 0.0047984 | 0.0038961 |
| 6 | 571\*859 | 0.004672 | 0.0044053 | 0.006044 |
| 9 | 15817\*32257 | 0.0062495 | 0.007256 | 0.0052081 |
| 12 | 443341\*753133 | 0.0109748 | 0.0076346 | 0.009538 |
| 15 | 15485867\*32452843 | 0.018718 | 0.0164567 | 0.029154 |
| 18 | 373587883\*776531401 | 0.140155 | 0.1049429 | 0.029688 |
| 19 | 1611623773\*5112733757 | 0.2038315 | 0.1057232 | 0.4090055 |
| 20 | 4666527007\*7367575799 | 0.3095235 | 0.2323143 | 0.706706 |
| 20 | 573259391\*24713066809 | 0.2051503 | 0.1043709 | 0.2317956 |
| 21 | 17582163853\*38244924643 | 1.0217849 | 0.8100343 | 0.907436 |
| 21 | 1611623773\*148780808401 | 0.1256805 | 0.6143912 | 0.343472 |
| 22 | 70107428687\*92714836259 | 2.3131691 | 2.6407481 | 0.7198288 |
| 22 | 8278737359\*375382617811 | 0.2079327 | 0.5439281 | 0.4043699 |
| 23 | 124934842639\*364195448411 | 1.1117743 | 2.2128393 | 1.0221009 |
| 23 | 55687146439\*729205648667 | 4.0178275 | 1.3108229 | 2.0091255 |
| 24 | 354084614573\*573461376017 | 5.5250967 | 0.9154797 | 6.2272982 |

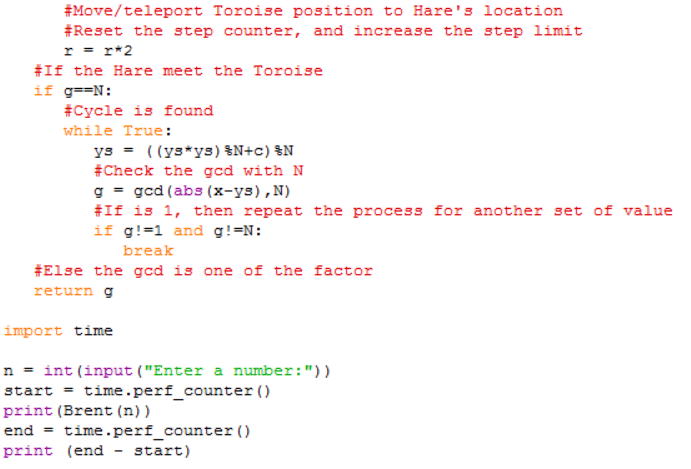
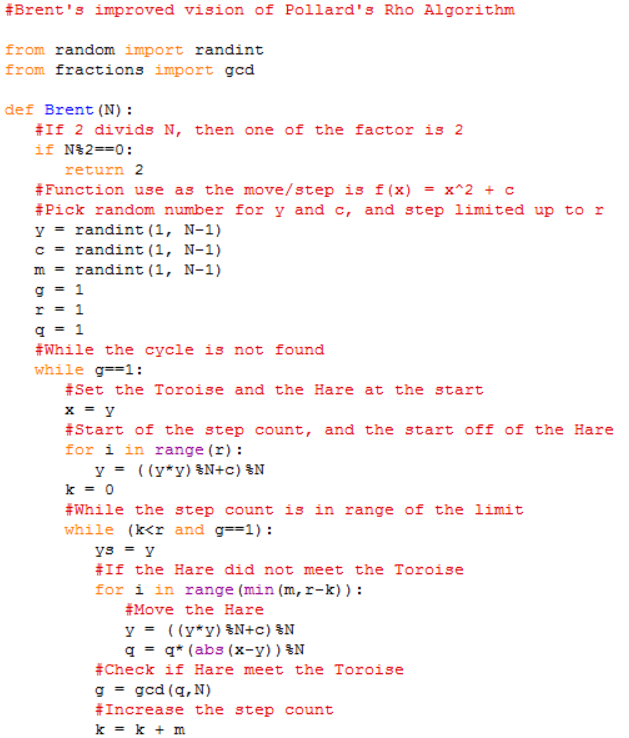
Base on the data that were collected in the table above, Pollard's rho algorithm could factor a number in a very short amount of time. It is faster than the Trial Division. For the 20-digit numbers, it could factor the number within a second. As the number increases in size, the time it takes to factor does increases, but in a small amount, a few extra seconds rather than minutes and hours than the Trial Division. Even a 24-digit number takes on average 6 seconds to factor.

Note that there were three different trials done for each of the number tested, recall that this algorithm starts by picking random number. This randomness factor could make a difference when factoring the inputted number. This is the reason why three trials are tested. As we could see in the table, some of the large numbers, numbers around 23-digits, could take a few seconds to be factored, but sometime it could take less than a second to be factored as well. Thus, the randomness does makes a difference, the run time for this algorithm is depended on the number being picked. We could say this was the “Lucky” algorithm, that if we are lucky and picked the “right” numbers, then factoring will be fast, but if it keeps picking the “wrong” numbers, it might take a longer time to factor the inputted value.

Based on table, when testing different numbers of same digit places but differ in factors difference, the size of the factor doesn’t seem to had an effect on the run time. Sometimes algorithm factored out the smaller factor very fast, while other times, it factored out the larger factor faster than the smaller one. Again, we could conclude that the performance mainly depended on randomness in picking the “right” numbers.

Brent’s improved version of Pollard rho factorization:

Brent’s version is only differed by the method of cycle detection used in the algorithm. The setup of algorithm and the program is similar to Pollard’s rho algorithm, except when we perform the cycle sequencing, we use Brent’s algorithm instead of Floyd’s algorithm. The program for Brent’s improved version is shown below:



(A Python program for Brent’s improved version of Pollard rho factorization)

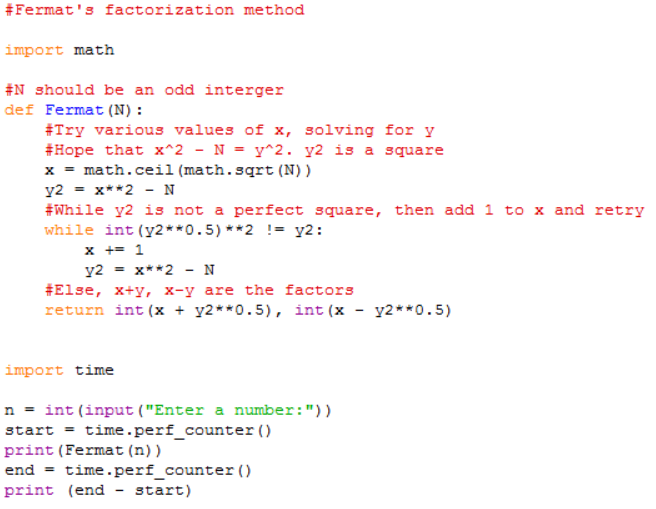
The following table will show some of the value inputted and the amount of the time for each trial to factor the number.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Digits | Inputted/tested numbers | Time (sec.) – For 3 individual trials | | |
| 3 | 17\*43 | 0.0041995 | 0.003533 | 0.0050722 |
| 6 | 571\*859 | 0.005836 | 0.004462 | 0.0033509 |
| 9 | 15817\*32257 | 0.004486 | 0.007909 | 0.0044796 |
| 12 | 443341\*753133 | 0.0094475 | 0.0054542 | 0.0087388 |
| 15 | 15485867\*32452843 | 0.00983861 | 0.0086803 | 0.0081217 |
| 18 | 373587883\*776531401 | 0.0151905 | 0.0168604 | 0.0410832 |
| 19 | 1611623773\*5112733757 | 0.14837203 | 0.103234 | 0.0432328 |
| 20 | 4666527007\*7367575799 | 0.0478360 | 0.0255589 | 0.1026678 |
| 20 | 573259391\*24713066809 | 0.025258 | 0.1054407 | 0.0185948 |
| 21 | 17582163853\*38244924643 | 0.3406954 | 0.3051636 | 0.7145309 |
| 21 | 1611623773\*148780808401 | 0.1039525 | 0.0281845 | 0.1056074 |
| 22 | 70107428687\*92714836259 | 0.6067665 | 0.1031715 | 0.2073997 |
| 22 | 8278737359\*375382617811 | 0.041645 | 0.2039603 | 0.104245 |
| 23 | 124934842639\*364195448411 | 0.3196505 | 0.4044903 | 0.8058277 |
| 23 | 55687146439\*729205648667 | 0.3045544 | 0.3080077 | 0.6487015 |
| 24 | 354084614573\*573461376017 | 0.7048322 | 0.1040173 | 0.6265947 |

Brent’s improved version of Pollard’s rho factorization algorithm uses a better cycle detection method, so it supposed to increases the speed for factoring integers. As shown in the table, it does speed up the run time for factoring. Such as the 24-digit number, it takes averagely 0.6 seconds to factor the number while Pollard’s rho algorithm takes averagely 6 seconds instead. This factorization algorithm performs quite well when factoring large numbers, at least for those numbers less than 25 digit places. Again, the run time for this algorithm also depended on the randomness on picking the “right” number, but overall it is faster than the Pollard’s rho algorithm.

Fermat's factorization method:

Fermat’s method is based on the representation of an odd integer as the difference of two squares. The setup of the program will be as follow: First, take the square root of the inputted value, *N*, and raise that number, *x*, to the ceiling function. We then square the number we get and subtract it to the inputted value, hoping to get a perfect square. If the outcome is not a perfect square, try another value of *x*, by adding one to it, and try again, until we found a perfect square, . Lastly, will each be a factor of the inputted value, *N*. The program for Fermat’s factorization method is shown below:



(A Python program for Fermat's factorization method)

The following table will show some of the value inputted and the amount of the time to factor the number.

|  |  |  |
| --- | --- | --- |
| Digits | Inputted/tested numbers | Time (sec.) |
| 3 | 17\*43 | 0.00414182388794162 |
| 6 | 571\*859 | 0.008275746547558284 |
| 9 | 15817\*32257 | 0.00815406763135397 |
| 12 | 443341\*753133 | 0.035098046342284375 |
| 15 | 15485867\*32452843 | 2.0106658681158573 |
| 15 | 3541541\*160481183 | 76.31415165301598 |
| 18 | 373587883\*776531401 | 47.581284676515814 |
| 18 | 24036583\*14059819243 | 9994.85627903352 |
| 19 | 1611623773\*5112733757 | 759.374826765569 |
| 20 | 4666527007\*7367575799 | 229.26090646051836 |
| 20 | 573259391\*24713066809 | 14088.679095957257 |

Base on the data that were collected in the table above, Fermat’s factorization method is an interesting one. This method performs quite well when factoring small numbers, and as the number size increases, the time it takes to factor also increases by many seconds or minutes, similar to the Trial division. However, if we look at the 19-digit trial and the 1st 20-digit trial, we could notice that it takes about roughly 9 more minutes to factor the 19-digit number than to factor the 20-digit number, but why is this?

If you look at the 2nd trials for the different digits: 15, 18, and 20. Those number are a product of two primes which are separated by two digit places away, and the time it takes to factor those number are larger, much larger, by hours. Now going back to the 19-digit and 20-digit cases, you could see that the difference between the factors of the 20-digit trial is smaller than the difference between the factors of the 19-digit trial. This shows that the amount of time that it takes this method to factor a number is not depended on the size of the number being factored, rather, it is depended on the differences in size between the factors of that number. If the difference between the factors is small, then this method could factor the number in a short amount of time, but if the difference between the factors is large, it will take a longer time to factor the number, it could even take longer than the Trial Division method in the worst case. Thus, Fermat’s factorization method is suitable for factoring number that consist factors that are close to each other, or else it could take a long time if the factors are separated far apart from each other.

**Conclusion**

The goal of this mathematical research is to expand our knowledge on the field of integer factorization, by investigating different methods and algorithms and how well do they perform. Through this investigation, we had analyzed the Trial Division, the Pollard's rho algorithm, Brent’s improved version of Pollard rho factorization, and Fermat's factorization method. Each of these methods/algorithms operates in its unique way. Some perform faster than others on factoring numbers, and each one had its strengths and weaknesses.

1. Trial Division: This method is the easiest one to understand, since it only requires to test all the numbers up to . However, when factoring large numbers, such as those of 20ish digit numbers, even with the modification of only testing the odd numbers after 2, the time it takes to factor the number are in minutes. As the number being factored become larger and larger, the run time could go into hours. But a number with a small factor could be factored in a short amount of time. Thus, this method is more suitable for factoring small digit numbers or numbers that have small factors.
2. Pollard's rho algorithm: This is the “Lucky” algorithm. Since the algorithm involves picking random numbers, the randomness factor cause the algorithm’s run time depended on the “luck”. This means that, if we are very lucky and the program picks the right number, the factorization could be done in very short amount of time, possibly less than a second. However, if we are unlucky and the program keep running numbers that doesn’t work, it could take a long time to factor the number. Even though it contains randomness, on average, it performs very well with an overall small amount of time, no more than a minute for numbers contains up to 24-digit places.
3. Brent’s improved version of Pollard rho factorization: In Brent’s improved version, Brent uses his algorithm to help detect cycle more quickly, which in turn, performs faster compare to Pollard’s original version. From this investigation, it does show that the average run time for the improved version is faster than the original one. It becomes more clear that this version performs better as the number gets larger and larger, it takes less than a second to factor a 24-digit number on average.
4. Fermat's factorization method: This method revolves more around the factors rather than the number being factored itself. As we increase the differences between the factors, the time it takes to factor increases in a large scale, by minutes or even hours. As shown in this investigation, when the two factors are separated by 2 digit places apart, their product may take up more time to factor compares to trial division. Therefore, this method is good to use when you know the factors of the number are close to each other, or else it could take up a long time to factor the number. This is also the reason why the RSA numbers are usually product of two very large prime, but they are separated by 2 to 3 digit places apart, to just avoid the efficiency of Fermat's factorization method.

Out of these four methods/algorithms involves integer factorization, Brent’s improved version of Pollard’s rho factorization performs the best, that it could factor larger number in a shorter amount of time. However, the numbers that was used to test on the different factorization methods are relatively small, when you compare to the 600-digits numbers that the RSA uses. Those run time will increase as the number being factored becomes larger. Nonetheless, these different integer factorization methods/algorithms are great to be known. Next time when you encounter a situation where you had to factor a large number (not too large), you may want to consider using one of these methods, of course not the Trial Division.

**Extensions and Applications**

The four different methods that we investigated are just few methods under integer factorization, there are still many more other methods/algorithms I had not mentioned, there are even faster ones, better ones out there. Such as the Lenstra elliptic curve factorization, which is useful to factor numbers that contain many small factors; or the Quadratic Sieve, the second fastest method known by far; or even the fastest method that is known so far, the General Number Field Sieve, which is used for factoring numbers that have more than 100-digit places; and many other methods. These are just a few other methods of integer factorization that you could look further into. You may even consider any possible improvement that could be made upon these methods, or to develop a more effective integer factorization method.

For applications, integer factorization is one of the basic blocks/pieces of a lot of computational number theory algorithms which is used, for example, in a lot of cryptography algorithms. Decrypting a RSA message without knowing the private key, for example, requires integer factorization. It also has applications in other fields, such as Combinatorics, and group theory. There is also the fast Fourier transform (FFT) algorithm that rely on integer factorization.

**Reference List**

1. James Pommersheim, Tim Marks, and Erica Flapan. Number Theory: A Lively Introduction with Proofs, Applications, and Stories. Wiley, 2010. Print.
2. “RSA (cryptosystem).” *Wikipedia*. N.p.: Wikimedia Foundation, 19 Dec. 2016. Web. 18 Jan. 2017.
3. “Modular exponentiation.” *Wikipedia*. N.p.: Wikimedia Foundation, 28 Dec. 2016. Web. 18 Jan. 2017.
4. “Integer factorization.” *Wikipedia*. N.p.: Wikimedia Foundation, 5 Dec. 2016. Web. 18 Jan. 2017.
5. “Fermat’s factorization method.” *Wikipedia*. N.p.: Wikimedia Foundation, 16 Nov. 2016. Web. 18 Jan. 2017.
6. “Pollard’s p − 1 algorithm.” *Wikipedia*. N.p.: Wikimedia Foundation, 14 June 2016. Web. 18 Jan. 2017.
7. “Pollard’s rho algorithm.” *Wikipedia*. N.p.: Wikimedia Foundation, 4 Jan. 2017. Web. 18 Jan. 2017.
8. “Shanks’ square forms factorization.” *Wikipedia*. N.p.: Wikimedia Foundation, 9 Jan. 2017. Web. 18 Jan. 2017.
9. “Trial division.” *Wikipedia*. N.p.: Wikimedia Foundation, 26 Apr. 2016. Web. 18 Jan. 2017.
10. Weisstein, Eric W. *Pollard rho Factorization method*. Wolfram Research, 28 Dec. 2002. Web. 18 Jan. 2017.
11. “Pollard’s Rho algorithm for prime Factorization.” *Mathematical*. GeeksforGeeks, 3 Jan. 2016. Web. 18 Jan. 2017.
12. Nash, Chris. “What is a good explanation of Floyd’s algorithm and Brent’s algorithm for cycle finding?” *Quora*. 20 June 2015. Web. 18 Jan. 2017.
13. “Pollard Rho Brent integer Factorization.” COME ON CODE ON, 18 Sept. 2010. Web. 18 Jan. 2017.
14. Weisstein, Eric W. *Fermat’s Factorization method*. Wolfram Research, Sept. 2017. Web. 18 Jan. 2017.
15. “Euler’s totient function.” *Wikipedia*. N.p.: Wikimedia Foundation, 13 Jan. 2017. Web. 18 Jan. 2017.
16. “Cycle detection.” *Wikipedia*. N.p.: Wikimedia Foundation, 29 Aug. 2016. Web. 18 Jan. 2017.