# Prime factorization

## Project 1

DD2440 Advanced algorithms

Viktor Collin
880316-0277
vcollin@kth.se

Anton Lindström
881015-4974
antlinds@kth.se

# *Abstract*

*This report discusses different algorithms for solving the problem of prime factorization of integer numbers. Four different algorithms is discussed: Trial division, Fermat's method, Pollard's rho and Pollard-Brent's method. All were implemented in C using the GMP library. The fastest of the evaluated algorithms was a combination of Pollard-Brent and trial division.*

# Table of contents

# Introduction

According to the fundamental theorem of arithmetic, every positive integer is either a prime number or a product of of prime numbers. Prime factorization is the task of finding the prime factors for an integer. When the prime factors are large this is a very hard task since there are no known algorithm for prime factorization that runs in polynomial time.
It's however easy to generate large prime numbers. There are lots of applications using these properties. For example, many algorithms for cryptography (such as RSA). The large prime numbers $p$ and $q$ and the composite $N = pq$ are easy to generate. The task of finding $N$ without knowing $p$ and $q$ is very hard and are in many cases not possible to do in reasonable time using modern computers.

# Problem description

Our problem was to create a computer program for factoring numbers. In the evaluation our program was given 100 integers and 15 seconds to factorize as many integers as possible. The integers were at most 100 bits long, thus less than $2^{100} = 1.27 \cdot 10^{30}$ .
The program was scored after how many integers it was able to factorize correctly.

# Background

There exists many algorithms for finding non-trivial divisors for an integer. We have examined four different variants for solving this problem: Trial division, Fermat's factorization method, Pollard's rho and Pollard-Brent's method.

## Trial division

Trial division is the most trivial algorithm for finding non-trivial factors.

Given an integer $N > 1$ , compute $\frac{N}{x}$ , for all $2 < x < n$ . If the remainder is 0, then $x$ is a non-trivial factor in $N$. If we can't find a number $x$ with the reminder 0, then $N$ is a prime.

This basic version of the algorithm is easy to improve. The largest factor in $N$ is $\sqrt{N}$ , since $\sqrt{N} \cdot \sqrt{N} = N$ . Therefore we only have test with $x$ between 2 and $\sqrt{N}$ .
A second improvement is to only test with prime numbers. These can be precomputed or generated using for example Sieve of Eratosthenes.

Trial division is very fast on small factors but when the factor size increase, the algorithm becomes very slow. The running time for trial division is $\Omega(\sqrt{N})$ [1].

## Fermat's factorization method

Fermat's factorization method uses the fact that every odd integer can be written as the differences of two squares. This is easy to prove. An odd integer can be written as $N = 2x - 1$ .

$2x-1=$
$2x-1+x^2-x^2=$
$\{x^2+2x-1=(x-1)^2\}=$
$x^2-(x-1)^2$

The odd integer $N$ can therefore be written as $N=a^2-b^2=(a+b)(a-b)=c\cdot d$. If $c>1$ and $d>1$, $c$ and $d$ are non-trivial factors in $N$. Fermat's factorization method simply tries to find a pair $a$ and $b$ fulfilling this condition. If we break out $a$ of the equation $a=\sqrt{N-b^2}$, we know that $a\geq\sqrt{N}$. Therefore we choose $\sqrt{N}$ as our starting point and then increment $a$ until $a^2-N$ is an even square.

```
FERMAT(N):
      a ← √N
      b2 ← a^2 − N
      while b2 isn't a square:
            a ← a + 1
            b2 ← a^2 − N
      return a − √(N)
```

The algorithm requires that N is odd so it's a good idea to first use trial division to remove all small factors. Fermat's factorization method find factors close to each other fast, but if the factors differs much in size the algorithm can be just as slow as trial division.

More advanced algorithms such as the Quadratic Sieve and Dixon's factorization method are based on Fermat's.

## Pollard's rho

The Pollard's rho algorithm was invented by John Pollard in 1975. It tries to find a non-trivial factor $d|N$. It's based on a heuristic that finds a factor $p$ after $O(\sqrt{p})$ iterations. This however has not been proved, but experiences shows that it behaves in this way. Pollards rho can be written as the following pseudo code:

```
POLLARD(N):
      x_0 ∈_U Z_N
      define the sequence {x_i}_i by x_i ≡ x_{i−1}^2 + 1 mod N
      for i ← 1,2, ... :
            d ← gcd(x_2i − x_i, N)
            if d > 1:
                  return d
```

It's easy to understand that when the algorithm finds a value $i$ such as $gcd(x_{2i}-x_i,N)>1$, this divisor is a non-trivial factor in $N$. The question is how many iterations is needed for this to happen.

If we study a sequence mod $N$ with a random starting point $x_0 \in_U \mathbb{Z}_N$ (FIG 1), we can see that after awhile, the sequence starts to loop. This is because $\mathbb{Z}_N$ is finite and $x_i$ only depends on $x_{i-1}$. According to [2] we can expect that it takes $O(\sqrt{N})$ steps before the sequence cycles.

Since $N$ is a composite, we can define $N = pq$, where $p$ and $q$ is relative prima and non-trivial. We also define the sequence $x'_i \equiv x_i \bmod p$ and our updating function $f_n(x) = x^2 + 1 \bmod n$. The next number in our new sequence $x'_{i+1}$ can be defined as the following [2]:

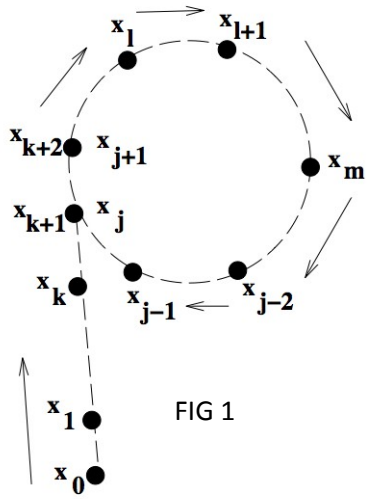


FIG 1

$$
\begin{aligned}
&x'_{i+1} = x_{x+1} \bmod p = \\
&f_N(x_i) \bmod p = \\
&((x_i^2 - 1) \bmod N) \bmod p = \\
&(x_i^2 - 1) \bmod p = \\
&((x_i \bmod p)^2 - 1) \bmod p = \\
&((x'_i)^2 - 1) \bmod p = \\
&f_p(x'_i)
\end{aligned}
$$

This shows that the $x'_i$ sequence obeys the same recurrence as the sequence $x_i$ [2] and in the same way as before, we can expect $O(\sqrt{p})$ iterations before this sequence cycles.

In 1980 Richard Brent published an optimized version of Pollards rho using another cycle detection algorithm called Brent's algorithm [3]. Brent claimed that the new version was around 24 % faster than John Pollards initial version.
The cycle detection in the initial version is called Floyd's cycle-detection where $x_{2i}$ "moves" twice as fast as $x_i$. Brent's algorithm uses a different approach where the first sequence pointer $y$, only "moves" when $i$ is a power of two:

```
POLLARD_BRENT(N):
      x ← 2
      y ← 2
      for i ← 1,2,3, ... :
            x ← x^2 + 1 mod N
            d ← gcd(x − y, N)

            if d != 1:
                  return d
            if isPowerOfTwo(i):
                  y ← x
```

# Method

In the beginning of the project we tried to implement these algorithms, evaluate them and then try to improve them to get better results.

## Our first attempt

First we tried to solve the problem by implementing Pollard's rho using Java and the BigInteger class. We choose Java because it is the language that we are most comfortable with but we was interested in using C and the GMP library as a new experience.

Because most algorithms just return a non-trivial factor (d) that is not guaranteed to be a prime we had to implement a way to divide the large number (N) into two smaller numbers (n, m). We implemented this by having a queue that when a d was found we added n = d and m = N/d to the queue. We then took the next element in the queue and if it was found to not be a prime we tried to factor it again. See pseudo code below.

```
FIND_FACTORS(N):
      f = new Vector
      q = new Queue
      q.add(N)
      while q.isNotEmpty:
            N = q.pop
            if isPrime(N):
                  f.add(N)
            else:
                  d = factorize(N)
                  q.add(d)
                  q.add(N/d)
      return f
```

Pollard's rho have problems detecting "perfect powers", i.e. when N = p^e. GMP includes a convenient function to easily detect this: mpz_perfect_power_p. Unfortunately Java's BigInteger class doesn't have a corresponding function and our attempt to implement one wasn't fast enough. This problem was the reason (besides for the fun of it!) we switched to C and GMP.

## Our second attempt

In the C version we implemented a recursive algorithm to factor a number down to its prime factors. This method was chosen to avoid queues and vectors in C. The base case of the recursion was when a prime was found.

```
FIND_FACTORS(N, exp):
      If IS_PRIME(N):
            ADD_FACTOR(N, exp)
      Else if IS_PERFECTPOWER(N):
            exp ← exp * FIND_PERFECTPOWER(N)
            FIND_FACTORS( N^(1/exp) , exp)
      Else:
            d ← FIND_NON_TRIVIAL_DIVISIOR(N)
            FIND_FACTORS(d, exp)
            FIND_FACTORS(N/d, exp)
```

Before FIND_FACTORS was called we used trial division with the first 1000 prime numbers to eliminate small factors. The function IS_PRIME is the Miller-Rabbin test implemented in GMP.

If N is a perfect power ( $N = p^{\exp}$ ) we need to keep track on the exponent exp for each factor and when the algorithm has found all factors, each factor is printed out exp times. The initial call to FIND_FACTORS uses exp = 1 since N^1 = N.

If N is a perfect power the function FIND_PERFECTPOWER is used to find the exponent. The largest exponent is $\log(N)$ since the lowest base is 2 and $2^{\log N} = N$ .

```
FIND_PERFECTPOWER(N)
      for exp = 2,3,4,...,log(N)
            if N^(1/exp) is integer
                  return exp
```

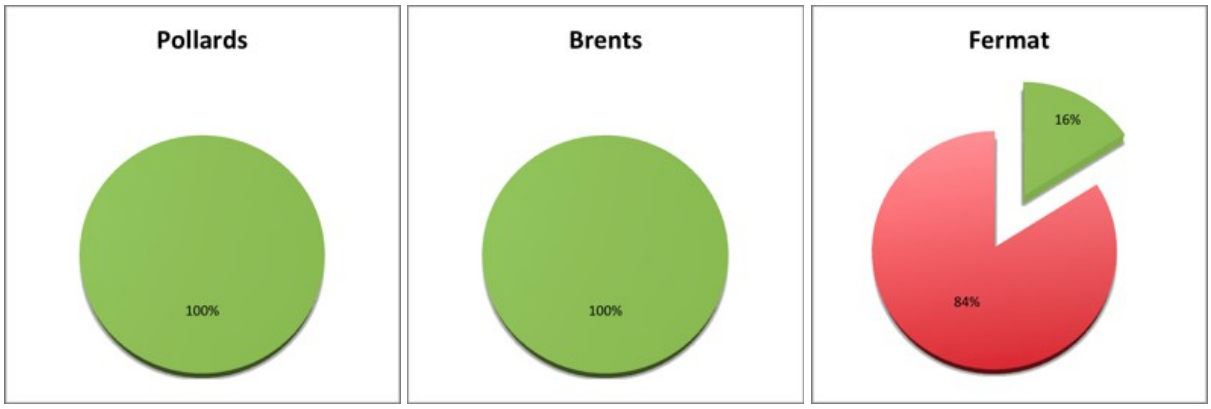
We implemented and evaluated three algorithms: Fermat's, Pollard's rho and Pollard-Brent. For the evaluation 100 random numbers up to $2^{100}$ was generated. Then each algorithm was given 15 seconds to factor each number. If the time expired the execution was terminated and counted as a failure. The correctness of each factorization was verified.
Since we wanted the result to depend more on the algorithm to be evaluated, the number of primes in the first trial division pass was reduced to the first 100 primes.

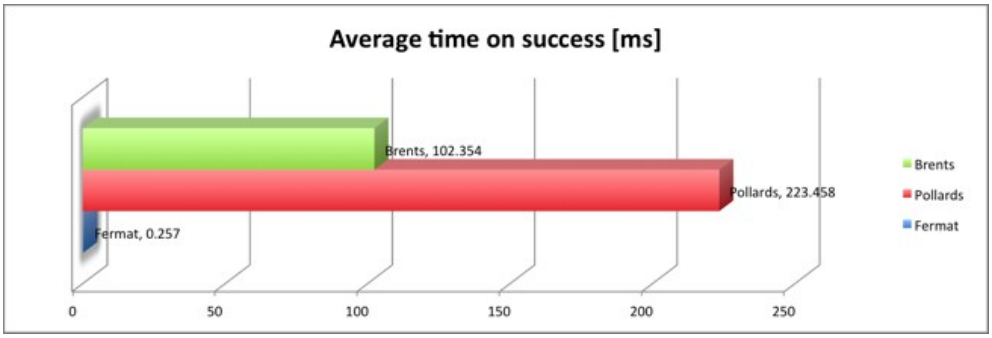
In the Kattis (KTH's evaluation system) version of each algorithm we used a simple counter that was decremented in well-chosen loops. If the counter went below 1 the factorization of the current number was aborted and the algorithm tried to factor the next given number. The initial timer value was different for each algorithm and was chosen by empirical methods (multiple submission to Kattis). In our evaluation this timer was disabled and the execution was aborted using the `ulimit` command.

# Results

Results from our test cases:

| Algorithm | Success Rate | Total Time | Average Time (on success) |
|---|---|---|---|
| Pollard-Brent | 100% | 10.24 s | 102.35 ms |
| Pollard-rho | 100% | 22.35 s | 223.46 ms |
| Fermat's | 16% | 1260 s | 0.257 ms |




Results from the Kattis evaluation:

| Algorithm | Score | Time | Kattis ID |
|---|---|---|---|
| Pollard-Brent | 81 | 14.07 s | 306418 |
| Pollard-rho | 74 | 14.55 s | 304559 |
| Fermat's | 19 | 14.81s | 306671 |

# Discussion

Our results contains no surprises. We can see that Fermat's method in the normal case is so slow that we had to terminate the execution but in the successful cases it succeeds very fast (average 0.257 ms), much faster than the other algorithms. This is probably because Fermat's is very good at finding factors if the factor is close to $\sqrt{N}$ where $N$ is the number that should be factored. Otherwise it's just as bad as trial division.

When Brent published his algorithm he claimed that it would be 24% faster than Pollard's rho but in our test case we get that it is twice as fast. The vast improvement can be explained by that Brent's algorithm does not use the expensive GCD function as much as the original version of Pollard's rho. There is a way to minimize the GCD usage in Pollard's rho by replacing them with some multiplications and a mod operation, this is however not implemented in our version.

The reason for that we get a 100% success rate for both Pollard's rho and Pollard-Brent in our evaluation but not in Kattis probably depends on two reasons. First, in our test we give the algorithm 15 seconds of run time and in Kattis we have to calculate all 100 numbers in only 15 seconds. The second reason is that the numbers in Kattis is probably more intelligently constructed with the purpose to be hard to factorize, rather than just random integers that is our case.

It would be interesting to do a deeper analysis of our implementation using for instance semi primes (large composites with only two factors) and integers with many large prime factors.

A possible improvement would be to replace the current counter timer with a more advanced function based on the actual time. Instead of having a fixed counter value for the maximum time per number, the allowed execution time could be calculated by the actual time left and the amount of numbers left to factorize. This would allow a number to be revisited if there is time left. Another improvement would also be to sort the numbers after expected factorization time e.g. based on the bit length of the numbers after a first pass with trial division. This wouldn't matter in our test case since time managing doesn't matter, but it may have improved the results in the Kattis evaluation.

An interesting next step would be to expand Fermat's factorization to the Quadratic Sieve method. This algorithm is significantly faster than Pollard-Brent and many implementation of this algorithm successfully factored all 100 numbers in the Kattis evaluation.

# References

[1] *Notes for the course advanced algorithms* by Johan Håstad

[2] *Introductions to algorithms*, Third edition, Thomas H. Cormen Charles E. Leiserson, Ronald L. Rivest, Clifford Stein

[3] *Wikipedia: Floyd's cycle-finding algorithm*
http://en.wikipedia.org/wiki/Floyd%27s_cycle-finding_algorithm#Brent.27s_algorithm