# Integer Factorization

**KTH - Royal Institute of Technology, CSC**
**DD2440 - Avalg**

Langesten, Daniel
911105-4038
langest@kth.se

Norberg, Jesper
900928-2196
jenor@kth.se

2013-11-07

## Abstract

The factorization of integers is a problem which has grown in importance as RSA encryptation has became more widespread. This report presents our approach to the factorization problem - the Brent-Pollard's $\rho$ method, in conjunction with Trial division and the Miller-Rabin probabilistic primality test.

In order to evaluate our program we have performance checked the various versions created, followed by analyzing the data. Finally we discuss possible improvements, in particular the implementation of the Quadratic Sieve.

## Contents

# 1 Introduction

*An overview of the problem at hand, as well as brief descriptions of the algorithms used in the implementations of the program.*

## 1.1 Problem

The task in the given assignment is to construct an algorithm that is able to factorize large integers. The algorithm need to be efficient since it has a limited amount of time. It should factorize 100 integers in 15 seconds, using a maximum of 32 MB of memory. Each integer is at most 100 bits long. The implementation needs to be in Java or in C/C++.

## 1.2 Background

The factorization of integers is a problem which has grown in importance as RSA encryptation has became more widespread. Integer factorization is a problem where a composite number is decomposed into non-trivial divisors. When an integer is very large it gets very hard to perform the factorization.

## 1.3 Approach

In order to complete the task presented in the project specification we initially needed to decide on which algorithms to use and in which programming language to implement them. The original algorithms of choice were Trial division, Pollard's $\rho$ method and Brent-Pollard's $\rho$ method. The reason for choosing these algorithms is that they are easy to implement compared to the best performing algorithms, whilst still producing adequate results. If we succeeded in the implementation of these algorithms we planed on implementing a more advanced algorithm, specifically the Quadratic Sieve. The programming language we initially decided upon was Java. This was because we both felt that Java was the language we had the most experience in, and therefore the language in which we would achieve the best results. After implementing the original algorithms however, we felt that our score was unexpectedly low, and concluded after some discussion that the fault likely laid in Java. As such, we ported our source code to C++, which procured much better results. This took quite some time however, which led to the abolishment of our plan to also implement the Quadratic Sieve.

# 2 Algorithms

*In this section we discuss different algorithms of interest. These are both implemented algorithms and algorithms that could be used in further development of the program.*

## 2.1 Miller-Rabin probabilistic primality test

In order to determine whether the number to factor is already prime, we used the Miller-Rabin probabilistic primality test, which is predefined within both BigIntegers for Java and GMP for C++.

## 2.2 Trial division

Trial division is one of the most laborious algorithms you can use to test if an integer is prime. The basic idea is to divide the integer with known primes smaller than the integer. If no prime divides the integer, it is prime. If you do not know all integers less than the integer, you can divide it by every other integer.

If utilized in the right way together with other algorithms, Trial division can be an important complement. Alone however, it can only factorize small numbers effectively.

## 2.3 Pollard's $\rho$

Pollard's $\rho$ method investigates a series of random numbers modulo $N$, where $N$ is the integer we want to factor. Since there are a limited amount of numbers that can appear in the series a number that has already been examined is bound to appear if we go through the series. This is also true for pseudo-random numbers which is of interest when programming. One commonly used pseudo-random number generator is $x_{i+1} \equiv x_i^2 + c(mod\ N)$. When checking for numbers that will repeat the visual representation will look like the Greek letter $\rho$ (rho) hence the name. First is a series of non-repeating numbers which then gets stuck in a loop where they start to repeat.

If $N$ is a product of 2 or more prime divisors and $P$ is one of these divisors. Then we can define a sequence of integers $y_i \equiv x_i(mod\ P)$, where $x_i$ is a number from our series.

We then get the following:

$$y_{i+1} \equiv x_i + 1(mod\ P)$$

$$y_{i+1} \equiv x_i^2 + c(mod\ P)$$

$$y_{i+1} \equiv y_i^2 + c(mod\ P)$$

where c is an arbitrary offset that differs from 0 and -2.

The $y_i$ sequence behaves just like the $x_i$ sequence but in $mod\ P$ instead of $mod\ N$. Even though we do not know $P$ we can detect when the $y_i$ sequence will fall into the earlier mentioned loop and start to repeat itself. So if $y_i \equiv y_j$ then $x_i(mod\ P) \equiv x_j(mod\ P)$ so $x_j - x_i$ is a multiple of $P$. Therefore $P$ divides the GCD of $x_j - x_i$ and $N$. So when $y_i \equiv y_j$ and $GCD(xi - xj, N) \neq 1$ we are done. [3]

## 2.4 Brent-Pollard's $\rho$

The same as Pollard's $\rho$, but using Brent's cycle finding algorithm instead [3]. This is an improvement, as the Brent-Pollard's $\rho$ is a faster algorithm, without losing in precision.

## 2.5 Quadratic Sieve

The Quadratic Sieve is second fastest integer factorization algorithm, and was invented by Carl Pomerance in 1981.[1] It builds upon combining Dixon's factorization method with sieving.

The Dixon factorization method builds upon the concept of having an exponent vector representing the prime-power factorization of a number. Providing an example; the prime-power factorization of 756 is: $2^2 \cdot 3^3 \cdot 5^0 \cdot 7^1$ , which can be represented with the exponent vector $(2, 3, 0, 1)$. Similarly, 84 would be represented with the exponent vector $(2, 1, 0, 1)$. Multiplying two numbers can be done by doing an element-wise addition of their exponent vectors, so $84 \cdot 756 = (2, 3, 0, 1) + (2, 1, 0, 1) = (4, 4, 0, 2) = 63504$. We can use this vector to identify squares. A number is a square if and only if each number in its vector is even. The square root will then be every element divided by two. The square root of $(4, 4, 0, 2)$ is therefore $(2, 2, 0, 1) = 252$. A clever way of simply identifying squares is to reduce the vector by modulo two. If all bits are 0, we have a square. If we don't have all bits at 0, multiplying with a vector reduced to an identical vector with modulo two will generate a square, as $(0, 0, 0, 1) + (0, 0, 0, 1) \equiv (0, 0, 0, 0)(mod 2)$. Applying this, identifying square roots is reduced to: Using a set of radix 2 vectors, find a subset which added (mod 2) becomes an empty vector. If we have more vectors than each vector has elements, linear dependence gives us that such a solution must exist.[2] An efficient way to identify it would be to perform a Gaussian elimination, after converting to matrix form.

Allowing all prime numbers in the exponent vectors would lead to far too big matrices however, which means a limitation is in order. As such, we only want "potential parts" of the main number where the all the prime numbers in their exponent vectors are low. These are called smooth numbers.

The method of efficiently finding these smooth numbers is called sieving, and from there the Quadratic Sieve gets its name.[1]

# 3 Program implementations

*Here descriptions of the program is presented.*

## 3.1 Version history

*Here the different implementations are explained. They are sorted chronologically from the time when the project started.*

### 3.1.1 Java with Trial division

The first implementation was written in Java and made use of trial division and Java's BigInteger, including its implementation of the Miller-Rabin probabilistic primality test. We did this as we presumed that the naive method of trial division would be insufficient by itself but we still wanted a simple factorizer which we could easilly test for correctness.

This implementation was able to factor large numbers to some extent but was quite slow. For the trial division the first 2001 prime numbers were pre-calculated and compiled into the program.

### 3.1.2 Java with Trial division and Pollard's $\rho$ method

This implementation is essentially *Java with trial division* extended with the Pollard's $\rho$ method. This significantly improved the performance of the program.

### 3.1.3 Java with Trial division and Brent-Pollard's $\rho$ method

This implementation is *Java with trial division* extended with the Brent-Pollard's $\rho$ method. The idea is to improve Pollard's $\rho$ method by using Brent's cycle detection method.[3]

### 3.1.4 C++ with Trial division and Pollard's $\rho$ method

This implementation is *Java with trial division and Pollard's $\rho$ method* rewritten i C++ to measure eventual performance gains over *C++ with Trial division and Brent-Pollard's $\rho$ method.*

### 3.1.5 C++ with Trial division and Brent-Pollard's $\rho$ method

In this version the programming language used is C++ rather than Java. The idea of this implementation was that C++ might perform better than Java on the given problem. This version is a rewritten version of *Java with Trial division and Pollard Brent method* in C++ instead of java. It proved to be a drastic improvement.

### 3.1.6 C++ with Trial Division and Brent-Pollard's $\rho$ method with check for big numbers

This is an extension of *C++ with Brent-Pollard's $\rho$ method* where a check is added after the Trial division. It checks for numbers that are to long to be factorized after the trivial primes are removed. The purpose with this check is to filter away number which we beforehand believe takes too long time to factorize.

## 3.2 Pseudo-code

*Here pseudo-code for the implementations described in version history is presented. It is used in all implementations except for Java with Trial division.*

**Algorithm 1** Factorize the integer N

```
procedure FACTORIZE($N$)
    $fQ$                                    ▷ A queue of integers to factor
    $fF$                                    ▷ A list of found factors
    $primes$                                ▷ A list of pre-calculated primes
    $fQ \leftarrow N$
    loop:
    while $!fQ.isEmpty()$ do
        $a \leftarrow fQ.poll()$
        if $a$ is prime then
            $fF.add(a)$
            $continue$ loop
        end if
        for $p$ in $primes$ do
            if if $p|a$ then
                $fF.add(p)$
                $fQ.offer(\frac{a}{p})$
                $continue$ loop
            end if
        end for
        $tmp \leftarrow PollardRho(a)$  ▷ If we get to here we will try Pollard's $\rho$ method. If we use Brent-Pollard's $\rho$ we use that instead here
        if $PollardRho(a)$ succeeded then
            $fQ.offer(\frac{a}{tmp})$
            $fQ.offer(tmp)$
            $continue$ loop
        end if
        return $fail$
    end while
    return $fF$
end procedure
```

## 4 Results

*Here the results of the different program implementations are presented.*

### 4.1 Java with Trial division

The best score on Kattis was 21, regardless of variations. This was probably due to the slow performance of the naive Trial division.

## 4.2 Java with Trial division and Pollard's $\rho$ method

The best score on Kattis was 44. A graph of the score variations is presented below; Figure 1.

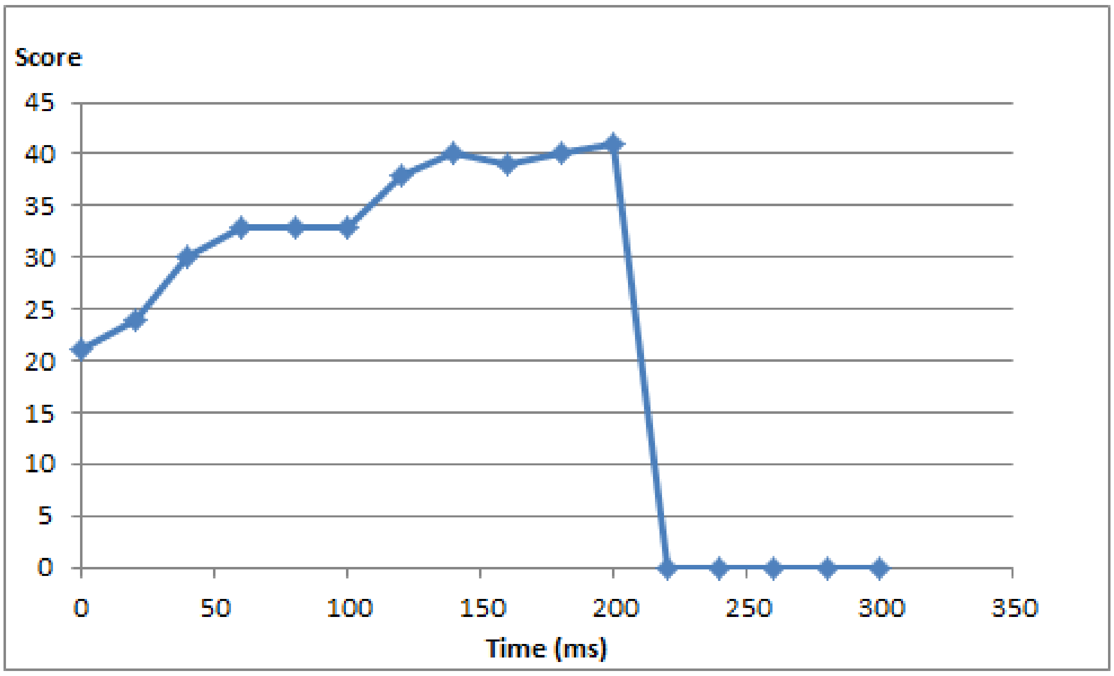

Figure 1: A graph describing how many primes we were able to factorize given a certain amount of time. The value 0 means it had Time Limit Exceeded.

## 4.3 Java with Trial division and Brent-Pollard's $\rho$ method

The best score on Kattis was 33. A graph of the score variations is presented below; Figure 2.

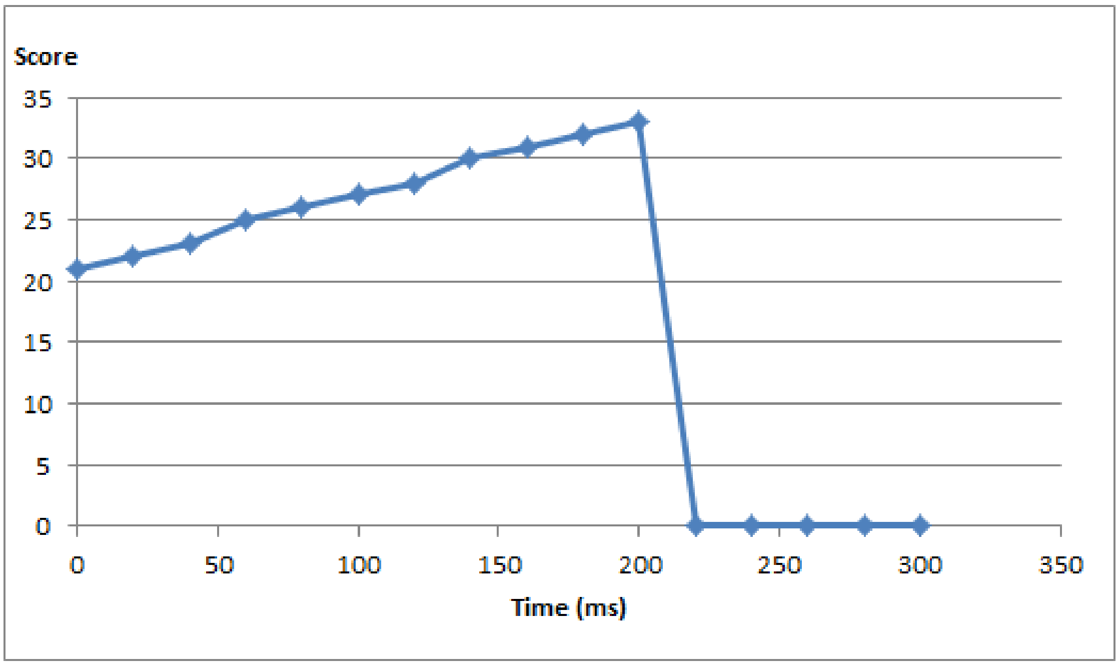

Figure 2: A graph describing how many primes we were able to factorize given a certain amount of time. The value 0 means it had Time Limit Exceeded.

## 4.4 C++ with Trial division and Pollard's $\rho$ method

The best score on Kattis was 58. A graph of the score variations is presented below; Figure 3.

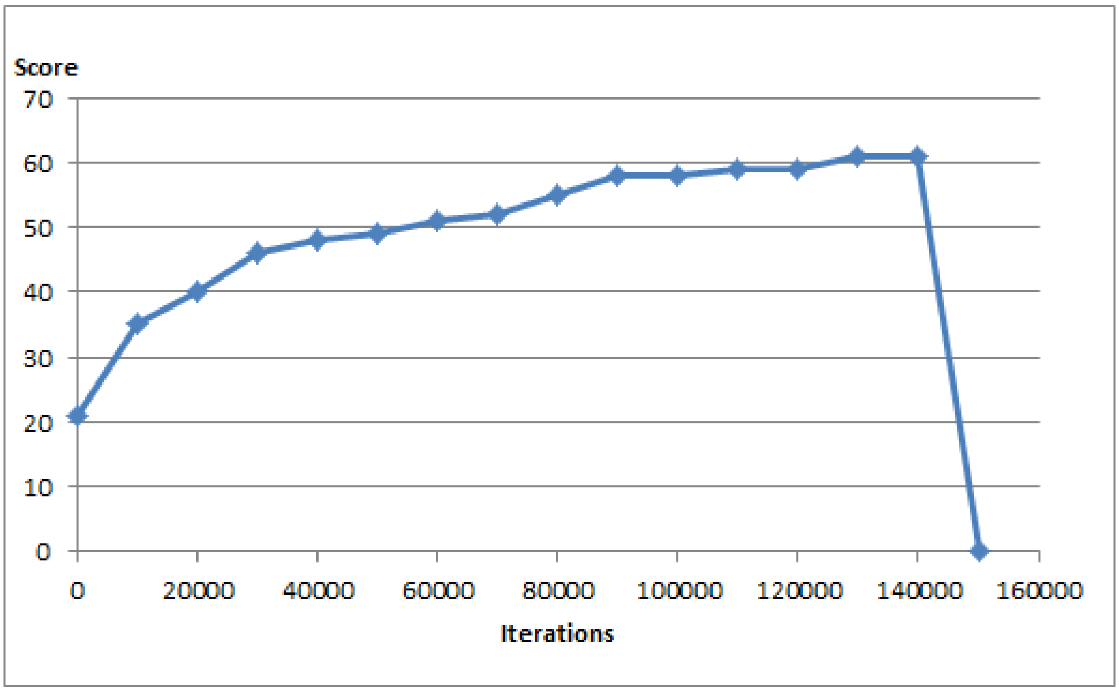

Figure 3: A graph describing how many primes we were able to factorize given a certain amount of iterations. The value 0 means it had Time Limit Exceeded.

## 4.5 C++ with Trial division and Brent-Pollard's $\rho$ method

The best score on Kattis was 80. A graph of the score variations is presented below; Figure 4.

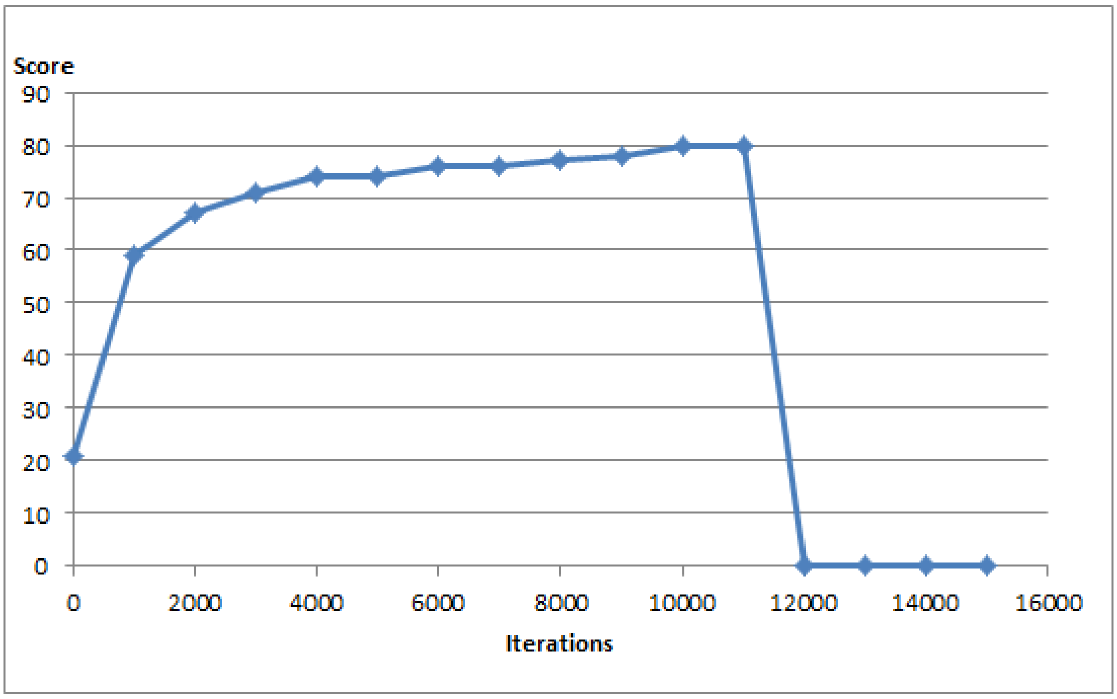

Figure 4: A graph describing how many primes we were able to factorize given a certain amount of iterations. The value 0 means it had Time Limit Exceeded.

## 4.6 C++ with Trial Division and Brent-Pollard's $\rho$ method with check for big numbers

The best score on Kattis was 80. A graph of the score variations is presented below; Figure 5.

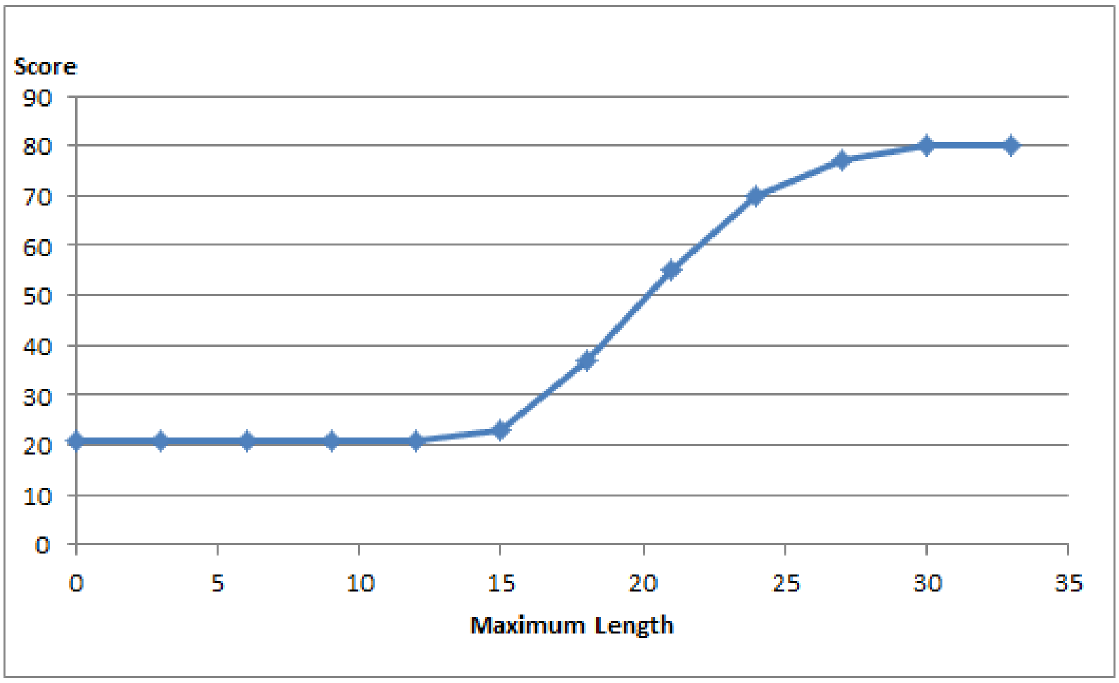

Figure 5: A graph describing how many primes we were able to factorize given a certain size of number as a limit. The value 0 means it had Time Limit Exceeded.

## 5 Discussion

*In this section we will use our results presented in 4 Results to discuss which parts of the project were successful, as well as which areas could be improved in order to heighten the performance of the program.*

### 5.1 Results

As earlier stated in our approach we would initially use Java as development language. But the results we got from it was not as good as we had expected. We deduced that this was most probably a performance issue since integer factorization is such a hard problem. Hence we decided to re-write the code in a more efficient language; C++. Indeed, as shown in the results we got almost twice as good results when using C++ instead of Java . A likely reason as to why Java was so much worse than C++ would be that the Java implementation ran out of time. The most obvious reason for this deducement is that the Java program needed a timer to run efficiently whilst the C++ program managed with just an iteration counter.

This would also be the reason why C++ is better than Java. It solves the time limited problem faster and because of that it has time to do better calculations. And the best of the C++ implementations is the ones with Brent-Pollard's $\rho$ method. This is very reasonable since it is an improved version of the regular Pollard's $\rho$ method.

That Java with only trivial division is worst is no suprise but that Java with Brent-Pollard's $\rho$ performs worse than Java with regular Pollard's $\rho$ is a bit supprising. We believe this result is because the Brent implementation need to make use of Javas BigInteger class more than the other implementation. Since BigInteger is immutable it will cost more to perform operate on it.

Our best implementation scored 80 on Kattis. The submission id is 451230.

### 5.2 What are we missing?

The main fault in our program is that we are using Pollard's $\rho$, an algorithm far slower than more advanced algorithms. As stated in the problem formulation, section 1.1, the numbers presented by Kattis will be at most 100 bits, which corresponds to 31 digits in base 10. For integers with decimal digit length between 20 and 120, the Quadratic Sieve is currently the fastest prime factorization algorithm.[1] As such, it is the optimal algorithm for the problem presented, and its implementation would be the natural next step if we were to continue pursuing the problem.

## 6 Conclusion

We are satisfied with our end result of 80 points, although given the opportunity, we would have liked to implement the Quadratic Sieve even though we did not have time to implement more advanced algorithms than we did.

## Bibliography

[1] Smooth numbers and the Quadratic Sieve - Carl Pomerance - Algorithmic Number Theory MSRI Publications Volume 44, 2008

[2] Linearly Dependent Functions - WolframMathWorld - http://mathworld.wolfram.com/LinearlyDependentFunctions.html (2013-11-07)

[3] Pollard rho Factorization Method - WolframMathWorld - http://mathworld.wolfram.com/PollardRhoFactorizationMethod.html (2013-11-07)