BACHELOR

Quasi-linear GCD computation and factoring RSA moduli

Cloostermans, B.

Award date:
2012

Eindhoven University of Technology
Department of Mathematics and Computer Science
Bachelor Mathematics

Quasi-linear GCD computation and factoring RSA moduli

Bachelor project - Version 1.0

Bouke Cloostermans
0640381

August 22, 2012

Abstract

This bachelor project consists of two parts. First, we describe two subquadratic GCD algorithms and test our implementation of these algorithms. Both algorithms work in a similar recursive way and have a running time of $O(n \log^2 n \log \log n)$ where n is the amount of digits of the input. Second, we describe and compare three algorithms which compute GCD's of more than two numbers. Two algorithms use a treelike structure to compute these GCD's and the other algorithm builds a coprime base from the input set. The algorithms which uses a treelike structure to compute these GCD's is faster than building a coprime base although all algorithms are quasilinear. The downside of these algorithm is that they only run well if few common factors exist in the input set. These algorithms have been used to factorize roughly 0.4% of a 11.1 million RSA keys dataset. However, this is only possible if the keys are badly generated.

Contents

1 Introduction

RSA is currently the most used internet encryption algorithm. It is still very time-consuming and in practice just impossible to break these keys if they are properly generated. In this bachelor project we will take a look at computing Greatest Common Divisor's of two large numbers in quasi-linear time and using these to possibly factorize RSA keys.

In the first part of this bachelor project we describe and compare two subquadratic algorithms to compute the Greatest Common Divisor of two n-bit integers. These algorithms are quite slow for small numbers but increasingly efficient as the numbers grow.

In the second part we describe and compare three algorithms which compute Greatest Common Divisor's of more than two numbers. The algorithms for this are focused on finding the Greatest Common Divisor of multiple RSA moduli. For this we can use the algorithms described in the first section to make them asymptotically fast. Using this algorithm and a huge database of RSA moduli we can possibly factorize RSA moduli. However, this only turns out to be possible if the RSA moduli are badly generated and therefore weak. In practice it seems that 0.4% of RSA moduli are in fact weak.

Notations: Unless specified explicitly, all logarithms are in base 2. $\ell(a)$ denotes the length of the binary representation of a. $\ell(a,b) = \max\{\ell(a), \ell(b)\}$ and $\underline{\ell(a,b)} = \min\{\ell(a), \ell(b)\}$.

2 Greatest Common Divisor

Since the purpose of this project is determining the Greatest Common Divisor (GCD) between two large numbers we will start off with a definition of a GCD.

Definition 2.1 The *Greatest Common Divisor* of two or more non-zero integers, is the largest positive integer that divides the numbers without a remainder.

For example, the GCD of 8 and 12 is 4.

Our next step is formulating how exactly we can find a GCD of two numbers. An algorithm that can compute the GCD of two numbers has long been known by mankind. The earliest description of the Euclidean GCD algorithm dates back to 300 B.C. A description of this algorithm is given in algorithm 1. This simple, yet elegant algorithm is still widely used to determine the GCD of two numbers.

Algorithm: EuclidGCD
Input: $a, b \in \mathbb{Z}, a > 0, b > 0$
Output: $\gcd(a, b)$
if $a = 0$ **then**
  return b
else
  return EuclidGCD(b, $a \mod b$)
end

Algorithm 1: The Euclidian GCD algorithm

Example 2.1 We use the Euclidean algorithm to determine the GCD of 73 and 38. The binary representation of each number is also shown.

i	$\mathbf{r_i}$	**binary**
0	73	1001001
1	38	100110
2	35	100011
3	3	11
4	2	10
5	1	1
6	0	0

We find the GCD of 73 and 38 is 1.

In example 2.1 we can now see a sequence of numbers which lead to the GCD of 73 and 38. This leads us to the next definition.

Definition 2.2 The sequence of $r_0 = a, r_1 = b, r_2, ...$ is the *sequence of remainders* that appear in the execution of the Euclidian GCD algorithm.

Computing the GCD of two n-bit numbers with this algorithm takes $O(n^2)$ time. Since we are interested in computing GCD's of large numbers, the running time of this algorithm can get out of hand. A faster version of this algorithm is given in algorithm 2.

Algorithm: BinaryGCD

Input: $a, b \in \mathbb{Z}$
Output: $\gcd(a, b)$

if $|b| > a$ **then**
  return BinaryGCD(b, a)
else if $b = 0$ **then**
  return a
else if $a \equiv 0 \mod 2$ **and** $b \equiv 0 \mod 2$ **then**
  return 2 BinaryGCD$(a/2, b/2)$
else if $a \equiv 0 \mod 2$ **and** $b \equiv 1 \mod 2$ **then**
  return BinaryGCD$(a/2, b)$
else if $a \equiv 1 \mod 2$ **and** $b \equiv 0 \mod 2$ **then**
  return BinaryGCD$(a, b/2)$
else
  return BinaryGCD$((|a| - |b|)/2, b)$
end

Algorithm 2: The binary GCD algorithm

This algorithm uses some elementary GCD properties to simplify the GCD computation: $\gcd(2a, 2b) = 2\gcd(a, b)$, $\gcd(a, 2b) = \gcd(a, b)$ if a odd, and $\gcd(a, b) = \gcd(a, b - a)$. BinaryGCD is generally faster than the Euclidean version since it doesn't require division, however the complexity is still $O(n^2)$. Therefore, we will formulate two asymptotically faster algorithms.

3 Stehlé's and Zimmermann's binary subquadratic GCD algorithm

This section will describe Stehlé's and Zimmermann's binary recursive GCD algorithm. The algorithm is completely described in [1].

3.1 Generalized Binary Division

We will start off this section with a new type of division: generalized binary (GB) division. To define GB division we first define the p-adic valuation of a number.

Definition 3.1.1 Let $p \in \mathbb{Z}$, $n \in \mathbb{Q}$.The *p-adic valuation* of n (notation: $\nu_p(n)$) is the amount of factors of p in n. $\nu_p(0) = \infty$ by definition.

Example 3.1.1 $60 = 2^2 \cdot 3^2 \cdot 5$ so $\nu_2(60) = 2$, $\nu_3(60) = 2$ and $\nu_5(60) = 1$.

For GB division we only need the 2-adic valuation of our number. Note that if we use the binary representation of n this simply means the amount of 0's on the right side of n. We can now define GB division.

Definition 3.1.2 $(q, r) = GB(a, b)$ with

$$\begin{aligned} r &= a + q\frac{b}{2^{\nu_2(b)-\nu_2(a)}} && (1)\\ |q| &< 2^{\nu_2(b)-\nu_2(a)} && (2)\\ \nu_2(r) &> \nu_2(b) && (3) \end{aligned}$$

$GB(a, b)$ always exists and is unique. This is proven in [1].

We name q the GB quotient and r the GB remainder of $GB(a, b)$. To compute the GB quotient and remainder of (a, b) we have algorithm 3 and 4. Both these algorithms use a function cmod.

Definition 3.1.3 *a cmod b* is the centered remainder of $a \mod b$. This means $-\frac{b}{2} < a \text{ cmod } b \leq \frac{b}{2}$.

Algorithm: ElementaryGB
Input: $a, b \in \mathbb{Z}$, $\nu_2(a) < \nu_2(b) < \infty$
Output: $(q, r) = \text{GB}(a, b)$

$q := 0$
$r := a$
while $\nu_2(r) \leq \nu_2(b)$ **do**
  $q := q - 2^{\nu_2(r)-\nu_2(a)}$
  $r := r - 2^{\nu_2(r)-\nu_2(b)}b$
end
$q := q \text{ cmod } 2^{\nu_2(b)-\nu_2(a)+1}$
$r := q\frac{b}{2^{\nu_2(b)-\nu_2(a)}} + a$
return (q, r)

Algorithm 3: Elementary GB division

Lemma 3.1.1 Algorithm 3 is correct.

Proof $\nu_2(r)$ increases by at least 1 each step and since $\nu_2(b)$ is finite the while loop ends. We find $r := a + \frac{q}{2^{\nu_2(b)-\nu_2(a)}}b$ satisfying equation (1) and $q := q \text{ cmod } 2^{\nu_2(b)-\nu_2(a)+1} \leq 2^{\nu_2(b)-\nu_2(a)}$ satisfying equation (2). Finally, the condition of the while loop ensures equation (3) is satisfied. Therefore, Algorithm 3 is correct. ■

Example 3.1.2 We use algorithm 3 to find $GB(17, 14)$.

$17 = 10001_2$, $14 = 1110_2$. We now keep subtracting $111_2 \cdot 2^i$ with i increasing by 1 each step. Since $\nu_2(17) = 1$ we stop when $\nu_2(r) = 1 + 1 = 2$.

```
10001
  111 -
-------
 1010
 1110 -
-------
 -100
```

We find $q = -11_2 = -3$, $-3 \bmod 2^2 = 1$ and $r = 2 \cdot 14 - 4$=24.

Note how each step of the while loop we subtract an increasing power of b to force 0's at the end of r.

Algorithm: FastGB
Input: $a, b \in \mathbb{Z}$, $\nu_2(a) < \nu_2(b) < \infty$
Output: $(q, r) = \mathrm{GB}(a, b)$
$A := -\frac{a}{2^{\nu_2(a)}}$
$B := \frac{b}{2^{\nu_2(b)}}$
$n := \nu_2(b) - \nu_2(a) + 1$
$q := 1$
for $i = 1 \to \lceil \log n \rceil$ **do**
  $q := q + q(1 - Bq) \mod 2^{2^i}$
end
$q := Aq \bmod 2^n$
$r := a + \frac{q}{2^{n-1}} b$
return (q, r)

Algorithm 4: Fast GB division

Lemma 3.1.2 Algorithm 4 is correct.

Proof In the final step of algorithm 4 we find $r := a + \frac{q}{2^{n-1}} b$. Since $n = \nu_2(b) - \nu_2(a) + 1$ this means equation (1) is satisfied. The second-to-last step we find $q := Aq \bmod 2^n \leq 2^{n-1}$ which means equation (2) is satisfied.

For equation (3) we first prove $1 - qB \equiv 0 \mod 2^n$ at the end of the while loop. As shown in algorithm 4, the algorithm defines q as $q_{i+1} := q_i + q_i(1 - Bq_i) \bmod 2^{2^i}$. We can use induction to prove our statement. Our induction hypothesis is $1 - q_i B \equiv 0 \mod 2^{2^i}$ for a given i. Thus, we need to prove $1 - q_{i+1} B \equiv 0 \bmod 2^{2^{i+1}}$.

The base case, $i = 1$ gives us $1 - Bq \mod 2 = 1 - \frac{b}{2^{\nu_2(b)}} q \mod 2 \equiv 0$ since $\frac{b}{2^{\nu_2(b)}}$ is always odd.

Since $1 - q_i B \equiv 0 \mod 2^{2^i}$ there exists a k such that $q_i B = 1 + k \cdot 2^{2^i}$.
$q_{i+1} = q_i + q_i(1 - Bq_i) \mod 2^{2^i}$ so $Bq_{i+1} = Bq_i + Bq_i(1 - Bq_i) \mod 2^{2^i}$.

This means there exists a k such that $Bq_{i+1} = 1 + k \cdot 2^{2^i} + (1 + k \cdot 2^{2^i})(1 - 1 + k \cdot 2^{2^i}) = 1 + k \cdot 2^{2^i} + k \cdot 2^{2^i} + k^2 \cdot (2^{2^i})^2 = 1 + (k^2 + k) \cdot 2^{2^{i+1}} \equiv 1 \mod 2^{2^{i+1}}$, so at the end of the while loop $1 - qB \equiv 0 \mod 2^n$.

Next, $q := Aq \bmod 2^n$ results in $A - qb \equiv 0 \mod 2^n$. From equation (1) we now get $r \equiv 0 \mod 2^{\nu_2(b)+1}$ which is the equivalent of equation (3). ■

We can now formulate an algorithm similar to algorithm 1 using GB division. This is algorithm 5. Note this algorithm requires an input of one odd and one even number to function correctly but this is not a real restriction on the functionality. We can simply divide a number a by $2^{\nu_2(a)}$ to force an odd number or multiply b by 2 to force an even number.

Algorithm: GBgcd
Input: $a, b \in \mathbb{Z}, 0 = \nu_2(a) < \nu_2(b) < \infty$
Output: $\gcd(a, b)$

```
if b = 0 then
    return a
else
    (q, r) := GB(a, b)
    return GBgcd(b, r)
end
```

Algorithm 5: The GB Euclidian GCD algorithm

Lemma 3.1.3 Algorithm 5 is correct.

Proof $\gcd(a, b) = \gcd(a, \frac{b}{2^{\nu_2(b)}}) = \gcd(a \mod \frac{b}{2^{\nu_2(b)}}, \frac{b}{2^{\nu_2(b)}}) = \gcd(\frac{b}{2^{\nu_2(b)}}, r)$ since $r = a \mod \frac{b}{2^{\nu_2(b)}}$. Therefore algorithm 5 returns the odd part of $\gcd(a, b)$ which is just $\gcd(a, b)$. ■

Example 3.1.3 We use the GBgcd algorithm to determine the GCD of 73 and 38. The binary representation of r_i is also shown.

i	$\mathbf{r_i}$	**binary**
0	73	1001001
1	38	0100110
2	92	1011100
3	-8	-1000
4	96	1100000
5	64	1000000
6	128	10000000
7	0	0

We find $\gcd(73, 38) = \frac{r_6}{2^{\nu_2(r_6)}} = 1$.

When we look at the binary representation of the remainder sequences of the basic Euclidian algorithm and the GB Euclidian algorithm we see the basic algorithm reduces the input by its most significant bits and the GB algorithm reduces the input by its least significant bits.

Lemma 3.1.4 Let r_0, r_1 be two non-zero integers with $0 = \nu_2(r_0) < \nu_2(r_1)$, and $r_0, r_1, ...$ be their GB remainder sequence, and $q_1, q_2, ...$ be the corresponding quotients.

$$\begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix} = \frac{1}{2^{\nu_2(r_i)}}[q_i]_{n_i}...[q_1]_{n_1} \cdot \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} \text{ where } [q_n] = \begin{pmatrix} 0 & 2^n \\ 2^n & q \end{pmatrix}, n_j = \nu_2(r_j) - \nu_2(r_{j-1})$$

Proof We will prove this claim using induction. First, we look at the base case $i = 1$.

$$\begin{aligned} \begin{pmatrix} r_1 \\ r_2 \end{pmatrix} &= \frac{1}{2^{\nu_2(r_1)}}[q_1]_{n_1} \cdot \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} \\ &= \frac{1}{2^{\nu_2(r_1)}} \begin{pmatrix} 0 & 2^{\nu_2(r_1)-\nu_2(r_0)} \\ 2^{\nu_2(r_1)-\nu_2(r_0)} & q_1 \end{pmatrix} \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} \\ &= \begin{pmatrix} 0 & 2^{-\nu_2(r_0)} \\ 2^{-\nu_2(r_0)} & 2^{-\nu_2(r_1)}q_1 \end{pmatrix} \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} \\ &= \begin{pmatrix} 0 & 1 \\ 1 & 2^{-\nu_2(r_1)}q_1 \end{pmatrix} \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} \end{aligned}$$

So we find

$$r_1 = r_1 \text{ and } r_2 = r_0 + \frac{q_1}{2^{\nu_2(r_1)}}r_1 = r_0 + \frac{q_1}{2^{\nu_2(r_1)-\nu_2(r_0)}}r_1.$$

This means the base case of our induction hypothesis is satisfied.

$$\begin{pmatrix} r_{i-1} \\ r_i \end{pmatrix} = \frac{1}{2^{\nu_2(r_{i-1})}}[q_{i-1}]_{n_{i-1}}...[q_1]_{n_1} \cdot \begin{pmatrix} r_0 \\ r_1 \end{pmatrix}$$

Since $r_{i+1} = r_{i-1} + \frac{q_i}{2^{\nu_2(r_i)-\nu_2(r_{i-1})}}$ we find

$$\begin{aligned}
\begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix} &= \begin{pmatrix} 0 & 1 \\ 1 & \frac{q_i}{2^{\nu_2(r_i)-\nu_2(r_{i-1})}} \end{pmatrix} \begin{pmatrix} r_{i-1} \\ r_i \end{pmatrix} \\
&= \begin{pmatrix} 0 & 1 \\ 1 & \frac{q_i}{2^{\nu_2(r_i)-\nu_2(r_{i-1})}} \end{pmatrix} \frac{1}{2^{\nu_2(r_{i-1})}} [q_{i-1}]_{n_{i-1}} ... [q_1]_{n_1} \cdot \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} \\
&= \frac{1}{2^{\nu_2(r_i)}} \begin{pmatrix} 0 & 2^{\nu_2(r_i)} \\ 2^{\nu_2(r_i)} & 2^{\nu_2(r_{i-1})} q_i \end{pmatrix} \frac{1}{2^{\nu_2(r_{i-1})}} [q_{i-1}]_{n_{i-1}} ... [q_1]_{n_1} \cdot \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} \\
&= \frac{1}{2^{\nu_2(r_i)}} \begin{pmatrix} 0 & 2^{\nu_2(r_i)-\nu_2(r_{i-1})} \\ 2^{\nu_2(r_i)-\nu_2(r_{i-1})} & q_i \end{pmatrix} [q_{i-1}]_{n_{i-1}} ... [q_1]_{n_1} \cdot \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} \\
&= \frac{1}{2^{\nu_2(r_i)}} [q_i]_{n_i} ... [q_1]_{n_1} \cdot \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} \quad \blacksquare
\end{aligned}$$

3.2 Recursion

The key insight in Stehlé's and Zimmermann's algorithm is specified in the following lemma.

Lemma 3.2.1 Let $\nu_2(a) < \nu_2(b) \leq k$, and let $k - \nu_2(b) \geq \nu_2(b) - \nu_2(a)$. Then $q_1, ..., q_j$, the sequence of GB quotients of a_0 and b_0 are the initial sequence of GB quotients of $a = 2^k a_1 + a_0$ and $b = 2^k b_1 + b_0$.

Proof From equation (1) we know

$$q = -\frac{a}{2^{\nu_2(a)}} \cdot \left(\frac{b}{2^{\nu_2(b)}} \right)^{-1} \mod 2^{\nu_2(b)-\nu_2(a)+1}.$$

We now choose $a = 2^k a_1 + a_0$ and $b = 2^k b_1 + b_0$ so we get

$$q = -\frac{2^k a_1 + a_0}{2^{\nu_2(a)}} \cdot \left(\frac{2^k b_1 + b_0}{2^{\nu_2(b)}} \right)^{-1} \mod 2^{\nu_2(b)-\nu_2(a)+1}.$$

Since $\nu_2(a)$ and $\nu_2(a)$ is stored in the least significant bits of a and b and $\nu_2(a) < \nu_2(b) \leq k$, we can replace $\nu_2(a)$ by $\nu_2(a_0)$ and $\nu_2(b)$ by $\nu_2(b_0)$.

$$\begin{aligned}
q &= -\frac{2^k a_1 + a_0}{2^{\nu_2(a_0)}} \cdot \left(\frac{2^k b_1 + b_0}{2^{\nu_2(b_0)}} \right)^{-1} \mod 2^{\nu_2(b_0)-\nu_2(a_0)+1} \\
\frac{2^k b_1 + b_0}{2^{\nu_2(b_0)}} \cdot q &= -\frac{2^k a_1 + a_0}{2^{\nu_2(a_0)}} \mod 2^{\nu_2(b_0)-\nu_2(a_0)+1}
\end{aligned}$$

Since $k - \nu_2(b) > \nu_2(b) - \nu_2(a)$ we get

$$\begin{aligned}
\frac{b_0}{2^{\nu_2(b_0)}} q &= -\frac{a_0}{2^{\nu_2(a_0)}} \mod 2^{\nu_2(b_0)-\nu_2(a_0)+1} \\
q &= -\frac{a_0}{2^{\nu_2(a_0)}} \cdot \left(\frac{b_0}{2^{\nu_2(b_0)}} \right)^{-1} \mod 2^{\nu_2(b_0)-\nu_2(a_0)+1} \quad \blacksquare
\end{aligned}$$

We can use lemma 3.2.1 to form a recursive algorithm HalfGBgcd, given in algorithm 6. The goal of this algorithm is to transform 2 n-bit numbers (a and b) to 2 $\frac{n}{2}$-bit numbers (c and d) with the same GCD and give the matrix corresponding to this transformation.

HalfGBgcd uses a recursive call with the lower $\frac{n}{2}$ bits of a and b (a_0 and b_0 to find 2 $\frac{n}{4}$-bit numbers (c_1 and d_1) with the same GCD as a_0 and b_0. If we multiply the corresponding matrix with the most significant bits of a and b (a_1 and b_1) and respectively add c and d we get 2 $\frac{3n}{4}$-bit numbers with the same GCD as a and b. Repeating this process gives us 2 $\frac{n}{2}$-bit numbers.

The structure of HalfGBgcd is shown in figure 1.

Algorithm: HalfGBgcd

Input: $a, b \in \mathbb{Z}, 0 = \nu_2(a) < \nu_2(b)$

Output: An integer j, an integer matrix R and two integers c and d with $0 = \nu_2(c) < \nu_2(d)$, such that $\begin{pmatrix} c \\ d \end{pmatrix} = 2^{-2j} R \cdot \begin{pmatrix} a \\ b \end{pmatrix}$, and $c^* = 2^j c$, $d^* = 2^j d$ are two consecutive remainders of the GB remainder sequence of (a, b) that satisfy $\nu_2(c^*) \leq \ell(a) \leq \nu_2(d^*)$.

$k := \ell(a)$ **if** $\nu_2(b) > k$ **then**
  return $0, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, a, b$
else
  $k_1 := \lfloor k/2 \rfloor$
  $a_1 := \lfloor \frac{a}{2^{k_1+1}} \rfloor$; $a_0 := a \mod 2^{k_1+1}$
  $b_1 := \lfloor \frac{b}{2^{k_1+1}} \rfloor$; $b_0 := b \mod 2^{k_1+1}$
  $j_1, R_1, c_1, d_1 := \text{HalfGBgcd}(a_0, b_0)$
  $\begin{pmatrix} a' \\ b' \end{pmatrix} := 2^{2k_1+1-2j_1} R_1 \cdot \begin{pmatrix} a_1 \\ b_1 \end{pmatrix} + \begin{pmatrix} c_1 \\ d_1 \end{pmatrix}$
  $j_0 := \nu_2(b')$
  if $j_0 + j_1 > k$ **then**
    return j_1, R_1, a', b'
  else
    $(q,r) := \text{FastGB}(a', b')$
    $b' := \lfloor \frac{b'}{2^{j_0}} \rfloor$; $r := \lfloor \frac{r}{2^{j_0}} \rfloor$
    $k_2 = k - (j_0 + j_1)$
    $b'_1 := \lfloor \frac{b'}{2^{k_1+1}} \rfloor$; $b'_0 := b' \mod 2^{k_1+1}$
    $r_1 := \lfloor \frac{r}{2^{k_1+1}} \rfloor$; $r_0 := r \mod 2^{k_1+1}$
    $j_2, R_2, c_2, d_2 := \text{HalfGBgcd}(b'_0, r_0)$
    $\begin{pmatrix} c \\ d \end{pmatrix} := 2^{2k_2+1-2j_2} R_2 \cdot \begin{pmatrix} b'_1 \\ r_1 \end{pmatrix} + \begin{pmatrix} c_2 \\ d_2 \end{pmatrix}$
    return $j_1 + j_0 + j_2, R_2 \cdot [q]_{j_0} \cdot R_1, c, d$
  end
end

Algorithm 6: The HalfGBgcd algorithm

Example 3.2.1 We now follow one step of algorithm 6 to reduce the GCD of 153 and 212 to the GCD of two smaller numbers.

$153 = 10011001_2$
$212 = 11010100_2$

We split the numbers as follows:

1001	1001
1101	0100

So, on the left side we have 9 and 13. On the right side we have 9 and 4. Executing one GB division on the right half of the numbers results in $2^{-2}\begin{pmatrix} 0 & 1 \\ 1 & -2^{-2} \end{pmatrix}\begin{pmatrix} 9 \\ 4 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$.

We can now use the matrix used in this calculation and the result to simplify the initial GCD problem.

$$2^4 \cdot 2^{-2}\begin{pmatrix} 0 & 1 \\ 1 & -2^{-2} \end{pmatrix}\begin{pmatrix} 9 \\ 13 \end{pmatrix} + \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 52 \\ 23 \end{pmatrix}$$

This tells us $\gcd(153, 212) = \gcd(52, 23)$.

We can now repeatedly use algorithm 6 to quickly reduce the size of a and b. This is done in algorithm 7 which will reduce $\gcd(a, b)$ repeatedly using the halfGBgcd algorithm.

Algorithm: FastGBgcd
Input: $a, b \in \mathbb{Z}, 0 = \nu_2(a) < \nu_2(b)$
Output: $\gcd(a, b)$

$j, R, a', b' :=$ HalfGBgcd(a, b)
if *b'=0* **then**
  return a'
else
  $(a, r) :=$ GB(a', b')
  return FastGBgcd$\left(\frac{b'}{2^{\nu_2(b')}}, \frac{r}{2^{\nu_2(b')}}\right)$
end

Algorithm 7: The FastGBgcd algorithm

4 Niels Möller's subquadratic GCD algorithm

The second algorithm we will discuss has been published by Niels Möller, and is completely described in [2]. This algorithm is very similar to the GCD algorithm proposed by Stehlé and Zimmermann. The main difference between the two algorithms is the type of division used. Stehlé and Zimmermann used GB division which is replaced by a function which very closely resembles standard division.

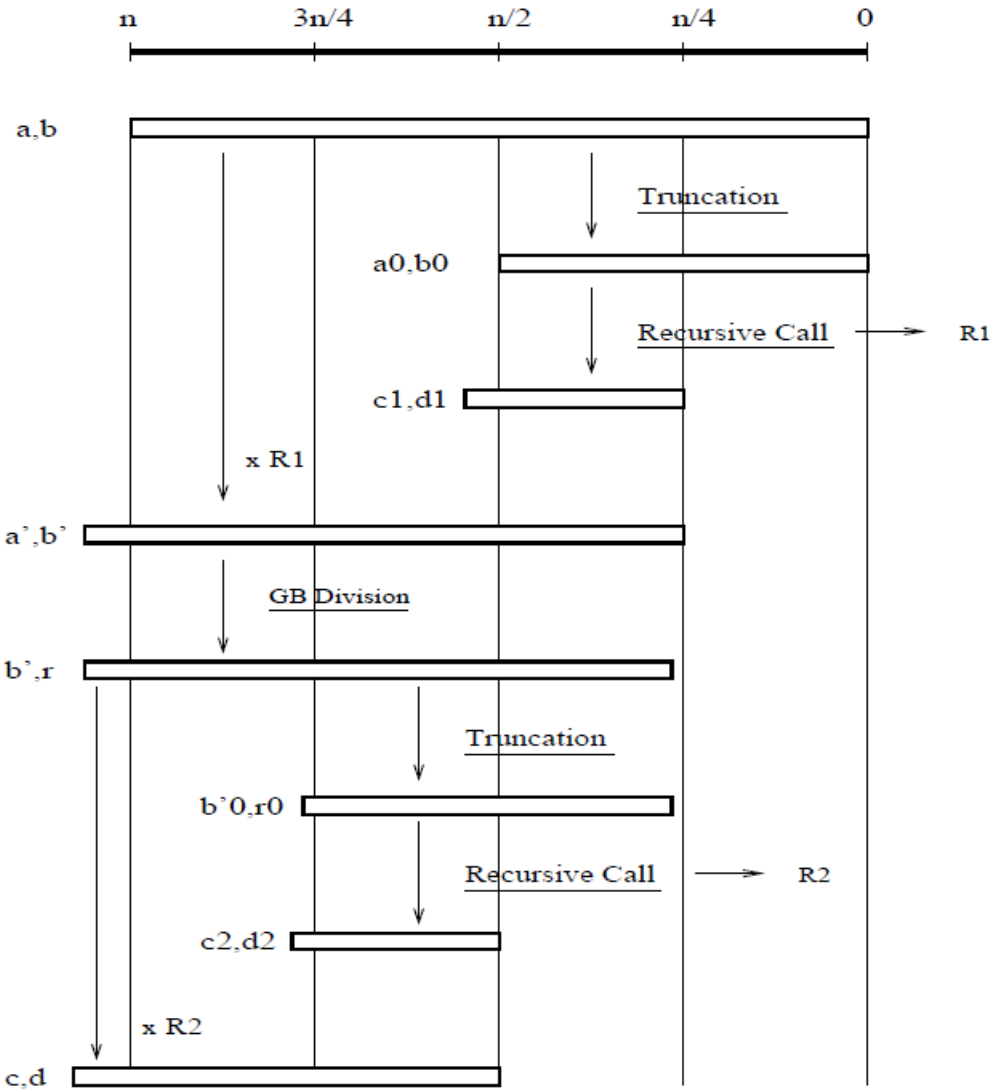

Figure 1: The recursive structure of algorithm HalfGBgcd. (source: [1])

Lemma 4.1 (Jebelean's criterion). Let $a > b > 0$, and let r_i be the remainder sequence. Let $n = \ell(a)$ and $m = \lceil n/2 \rceil$. If k is such that $\ell(r_{k+2}) > m$ then for any $p > 0$ and any a_1 and b_1, with $0 < a_1 < b_1 < 2^p$, the sequence $q_1, ...q_k$ is the initial quotient sequence for the numbers $a = a_1 2^p + a_0$ and $b = b_1 2^p + b_0$.

Proof of this lemma can be found in [3].

If we want our recursive algorithm to satisfy lemma 4.1 we need a function which never returns a remainder which is too small.

Definition 4.1 $sdiv(a, b, s)$ returns the largest q such that $qb < a$ and $\ell(a - qb) > s$.

Algorithm 8 correctly returns $\text{sdiv}(a, b, s)$.

Algorithm: sdiv
Input: $a, b \in \mathbb{N}$
Output: $\text{sdiv}(a, b, s)$

$q = \lfloor a/b \rfloor$
if $\ell(a - qb) \leq s$ **then**
  $q = q - 1$
end
return $q, a - qb$

Algorithm 8: The sdiv algorithm

Lemma 4.2 If a, b, c and d are integers, M is a 2×2 matrix, $\det M = 1$ and $\begin{pmatrix} a \\ b \end{pmatrix} = M \begin{pmatrix} c \\ d \end{pmatrix}$ then $\gcd(a, b) = \gcd(c, d)$.

Proof $\begin{pmatrix} a \\ b \end{pmatrix} = M \begin{pmatrix} c \\ d \end{pmatrix}$ implies $\gcd(c, d)$ divides both a and b and thus also $\gcd(a, b)$. Since $\det M = 1$, M^{-1} is also an integer matrix with $\det M^{-1} = 1$ and we can use the same argument to show $\gcd(a, b)$ divides $\gcd(c, d)$. ■

We can create a matrix which reduces a and b to two integers c and d with $\ell(a) > s$, $\ell(b) > s$ and $\ell(a - b) \leq s$ and $\gcd(a, b) = \gcd(c, d)$ by inverting the matrix M found using algorithm 9.

Algorithm: sgcd
Input: $a, b, s \in \mathbb{N}$
Output: Two integers c, d with $\gcd(c, d) = \gcd(a, b)$, $\ell(c) > s$, $\ell(d) > s$ and $\ell(c - d) \leq s$

$M = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
while $\ell(a - b) > s$ **do**
  if $a > b$ **then**
    $(q, a) = \text{sdiv}(a, b, s)$
    $M = M \begin{pmatrix} 1 & q \\ 0 & 1 \end{pmatrix}$
  else
    $(q, a) = \text{sdiv}(b, a, s)$
    $M = M \begin{pmatrix} 1 & 0 \\ q & 1 \end{pmatrix}$
  end
end
return a, b, M

Algorithm 9: The sgcd algorithm

We can now formulate algorithm 10. This algorithm is very similar to the GB division-variant using the exact same recursive structure.

Algorithm: halfGCD

Input: $a, b \in \mathbb{Z}$

Output: Two integers c, d with $\gcd(c,d) = \gcd(a,b)$ with $s = \lceil \ell(a,b) \rceil$, $\ell(c) > s$, $\ell(d) > s$ and $\ell(c-d) \leq s$ and an integer matrix M such that $\begin{pmatrix} a \\ b \end{pmatrix} = M \begin{pmatrix} c \\ d \end{pmatrix}$

$n := \lceil \ell(a,b) \rceil$; $s := \lceil n/2 \rceil$
if $\lceil \ell(a,b) \rceil > \lfloor 3n/4 \rfloor + 2$ **then**
  $p_1 := \lfloor n/2 \rfloor$; $n_1 := n - p_1$
  $a_1 := \lfloor \frac{a}{2^{p_1+1}} \rfloor$; $a_0 := a \mod 2^{p_1+1}$
  $b_1 := \lfloor \frac{b}{2^{p_1+1}} \rfloor$; $b_0 := b \mod 2^{p_1+1}$
  $c, d, M := \text{HalfGCD}(a_1, b_1)$
  $\begin{pmatrix} a \\ b \end{pmatrix} = 2^{p_1} \begin{pmatrix} c \\ d \end{pmatrix} + M^{-1} \begin{pmatrix} a_0 \\ b_0 \end{pmatrix}$
else
  $M = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
end
while $\lceil \ell(a,b) \rceil > \lceil 3n/4 \rceil$ **and** $\ell(a-b) > s$ **do**
  $a, b, M' := \text{sdiv}(a, b, s)$
  $M = M \cdot M'$
end
if $\lceil \ell(a,b) \rceil > s + 2$ **then**
  $n := \ell(a,b)$
  $p_2 := 2s - n + 1$; $n2 := n - p_2$
  $a_1 := \lfloor \frac{a}{2^{p_2+1}} \rfloor$; $a_0 := a \mod 2^{p_2+1}$
  $b_1 := \lfloor \frac{b}{2^{p_2+1}} \rfloor$; $b_0 := b \mod 2^{p_2+1}$
  $c, d, M' := \text{HalfGCD}(a_1, b_1)$
  $\begin{pmatrix} a \\ b \end{pmatrix} = 2^{p_1} \begin{pmatrix} c \\ d \end{pmatrix} + M'^{-1} \begin{pmatrix} a_0 \\ b_0 \end{pmatrix}$
  $M = M \cdot M'$
while $\ell(a-b) > s$ **do**
  $a, b, M' := \text{sdiv}(a, b, s)$
  $M = M \cdot M'$
  return a, b, M
end

Algorithm 10: The halfGCD algorithm

Example 4.1 We now follow one step of algorithm 10 to reduce the GCD of 212 and 153 to the GCD of two smaller numbers.

$212 = 11010100_2$
$153 = 10011001_2$

We split the numbers as follows:

$$\begin{array}{l|l} 1101 & 0100 \\ 1001 & 1001 \end{array}$$

So, on the left side we have 13 and 9. On the right side we have 4 and 9. Executing the extended Euclidian algorithm on the left half of the numbers gives us $2^{-2}\begin{pmatrix} 1 & -1 \\ -2 & 3 \end{pmatrix}\begin{pmatrix} 13 \\ 9 \end{pmatrix} = \begin{pmatrix} 4 \\ 1 \end{pmatrix}$.

We can now use the matrix used in this calculation and the result to simplify the initial GCD problem.

$$2^4\begin{pmatrix} 4 \\ 1 \end{pmatrix} + \begin{pmatrix} 1 & -1 \\ -2 & 3 \end{pmatrix}\begin{pmatrix} 4 \\ 9 \end{pmatrix} = \begin{pmatrix} 59 \\ 35 \end{pmatrix}$$

This tells us $\gcd(212, 153) = \gcd(59, 35)$.

Again, just like the GB division-variant this algorithm also works with an algorithm that keeps executing halfGCD until the numbers are small enough. This algorithm is given in algorithm 11.

Algorithm: fastGCD
Input: $a, b \in \mathbb{Z}$
Output: $\gcd(a, b)$

```
c, d, M = halfGCD(a, b)
if a = c and b = d then
 | gcd(a, b)
else
 | fastGCD(c, d)
end
```

Algorithm 11: The fastGCD algorithm

5 Running times

5.1 Analytic Approach

The calculation of the running time of the binary GCD algorithm and the sub-quadratic GCD algorithm so we will analyze them at the same time. Algorithms 6 and 10 both use two recursive calls and several multiplications which take $O(n \log n \log\log n)$ time with Schönhagen-Strassen multiplication. This means their running times both satisfy equation (4). In this equation k depends on the implementation.

$$T(n) = 2T\left(\frac{n}{2}\right) + kn \log n \log\log n \tag{4}$$

From this we can derive the following lemma.

Lemma 5.1.1 Algorithms 6 and 10 finish in $T(n) = O\left(n(\log n)^2 \log\log n\right)$ time.

Proof We can prove this lemma using induction. The base case is trivially satisfied since this case runs in constant time.

We can now assume there exists a c such that $T\left(\frac{n}{2}\right) \leq c \cdot \frac{n}{2}(\log \frac{n}{2})^2 \log\log \frac{n}{2}$ and we prove $T(n) \leq c \cdot n(\log n)^2 \log\log n$

$$\begin{aligned} T(n) &\leq 2c\frac{n}{2}(\log \frac{n}{2})^2 \log\log \frac{n}{2} + k \cdot n \log n \log\log n \\ &\leq c \cdot n\left((\log n)^2 - 2\log n + 1\right)\log\log n + k \cdot n \log n \log\log n \\ &= c \cdot n(\log n)^2 \log\log n + (k-2c)n \log n \log\log n + c \cdot n \log\log n \end{aligned}$$

Choosing $c = k$ results in

$$\begin{aligned} T(n) &\leq c \cdot n(\log n)^2 \log\log n - c \cdot n \log n \log\log n + c \cdot n \log\log n \\ &\leq c \cdot n(\log n)^2 \log\log n \quad \blacksquare \end{aligned}$$

Now we know the both halfGB algorithms run in $O\left(n(\log n)^2 \log\log n\right)$ time we can derive the running times of algorithms 7 and 11. The recursive formula for the running times is given by equation (5) which leads us to the following lemma.

$$T(n) = T\left(\frac{n}{2}\right) + kn \log n \log\log n \tag{5}$$

Lemma 5.1.2 Algorithms 7 and 11 finish in $T(n) = O\left(n(\log n)^2 \log\log n\right)$ time.

Proof Since $n^{\log_2 1+\epsilon} \leq n \log n \log\log n$ for ϵ small enough application of the master theorem [4] tells us $T(n) = O\left(n(\log n)^2 \log\log n\right)$ $\blacksquare$

5.2 Experimental approach

Here we will discuss the implementation and the testing of the algorithms. We will compare algorithms 2 (BinaryGCD), 7 (fastGBgcd) and 11 (fastGCD).

5.2.1 Implementation

We have implemented both recursive algorithms in Wolfram Mathematica 7.0 with a few slight modifications. The recursive calls on HalfGCD and HalfGBgcd were quite slow for relatively small numbers. Therefore the recursive call switches to a quadratic algorithm for numbers smaller than 100 bits. The choice for 100 bits is based purely on trial and error to see which boundary would be fastest. Testing of the algorithms has been done on a desktop computer.

In the implementation of Niels Möller's algorithm we made another slight alteration. As we see in algorithm 10 the algorithm creates a matrix M which is then inverted before using it for a multiplication. The matrix M is never

actually used without the inversion. Therefore in our implementation we didn't build a matrix M, but a matrix M^{-1}. This takes exactly as many operations as building a matrix M, but saves the time needed of inverting the matrix. The code for the implementation can be found in the appendix.

5.2.2 Stehlé's and Zimmermann's binary subquadratic GCD algorithm

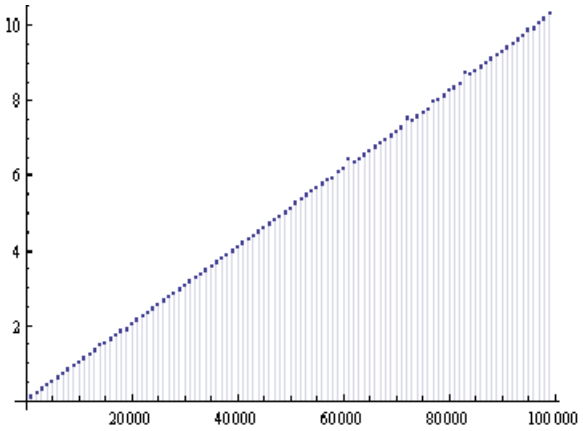

Figure 2: The running time of fastGBgcd for n-bit numbers

When we look at figure 2 we see the average running times of this algorithm computing the GCD's of increasingly large numbers. As predicted by the running time derivation at the beginning of this section, we have a nearly linear relation between the amount of bits and the running time. The algorithm takes about 1 second for every 10000 bits of input.

5.2.3 Niels Möller's subquadratic GCD algorithm

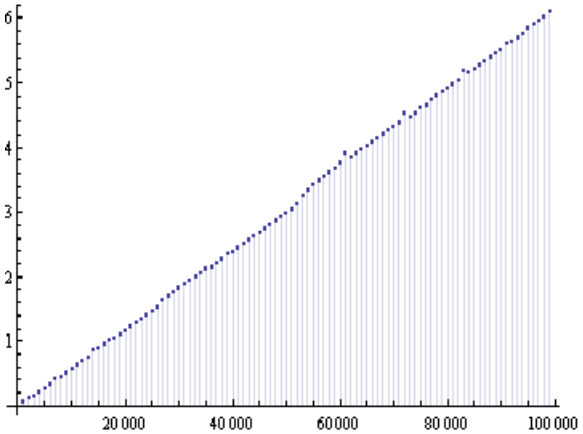

Figure 3: The running time of fastGCD for n-bit numbers

This implementation leads the the running times given by figure 3. Again, we see a nearly lineair relation between the amount of bits and the running time, just as predicted by the derivation. This algorithm takes about 0.6 seconds for every 10000 bits of input.

5.2.4 Comparison

When we plot the running times of binaryGCD, fastGBgcd and fastGCD in the same graph we get the graph given by figure 4. We see binaryGCD runs in quadratic time but is faster than the recursive algorithms for small numbers.

The recursive algorithms start winning for larger numbers though, as expected. We see FastGCD takes roughly 60% of the time fastGBgcd does in all situations. This corresponds with the the running time of fastGCD given in [2]. FastGCD starts overtaking binaryGCD at 10000-bit numbers, fastGBgcd starts overtaking binaryGCD at 80000-bit numbers.

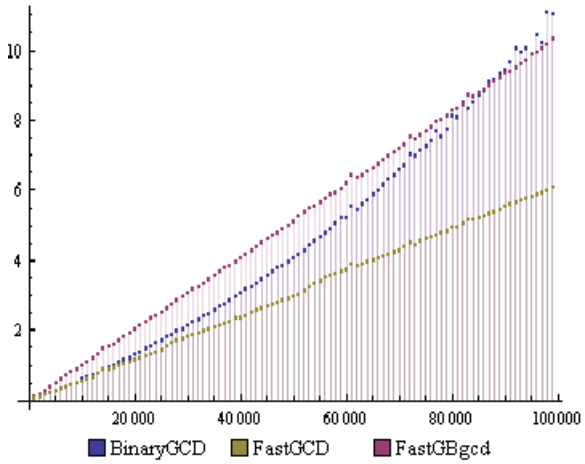

Figure 4: The running times of binaryGCD, fastGBgcd and fastGCD for n-bit numbers

6 Factorizing RSA moduli

We now describe three algorithms which we will use to try to factorize RSA moduli. Suppose we have two RSA moduli which share a prime number. If we compute the GCD of these two moduli we find this prime number and from there we can easily factor the key by executing one division. We now find as many RSA moduli as we can, and if two moduli share a GCD we can factorize these moduli.

The prime density function tells us there are roughly $\frac{2^{512}}{\log 2^{512}} - \frac{2^{511}}{\log 2^{511}} \approx 1.88531 \cdot 10^{151}$ 512-bit prime numbers. Therefore, even after collecting billions

of randomly generated RSA moduli the chances of finding a GCD unequal to 1 should be negligible. So why do we even bother computing these GCD's?

The answer lies in that the RSA moduli are not truly random. Modern random generation is based on software collecting information from physical sources (i.e. mouse movements or network events). If these inputs do not generate enough randomness the same input might occur somewhere else which in turn might result in a similar prime number being generated.

6.1 Building a GCD-tree

The first algorithm we will discuss is a basic algorithm which can be used to factorize RSA moduli. This is possibly a folklore algorithm of unknown origin suggested to me by Benne de Weger. The idea here is to compute the GCD of a pair of moduli and then replace them by the least common multiple of the moduli. This process is done in algorithm 12.

Algorithm: mergeGCD
Input: $n_1, n_2, ..., n_k \in \mathbb{Z}$
Output: all GCD's of n_i, n_j with $i \neq j$ such that $\gcd(n_i, n_j) \neq 1$ and a list of pairwise smallest common multiples

$s = \lfloor \frac{\ell(n)}{2} \rfloor$
$p = \{\}$
for $i = 1 \rightarrow s$ **do**
  $g = \gcd(n_{2i-1}, n_{2i})$
  if $g \neq 1$ **then**
    $p = p \cup (g, i)$
  end
  $m_i = \frac{n_{2i-1} \cdot n_{2i}}{g}$
end
if $\ell(n) \mod 2 \equiv 1$ **then**
  $m_{s+1} = n_{\ell(n)}$
end
return $\{m_i\}, p$

Algorithm 12: The mergeGCD algorithm

We do this for all moduli and keep repeating this process for the set of least common multiples we created. This way we get a tree of RSA moduli which we can use to find coprime factors between all RSA moduli. This is done in algorithm 13.

Algorithm: buildTree
Input: A set n
Output: A GCD-tree of n and all GCD's which are not equal to 1 found building the tree with their location
$t_1 = n$
for $i = 1 \rightarrow \lceil \log n \rceil$ **do**
  t_{i+1}, q=mergeGCD(t_i)
  $p = p \cup (i, q)$
end
return t, p

Algorithm 13: The buildTree algorithm

Example 6.1.1 Suppose we want to find common primefactors in $(2 \cdot 3, 5 \cdot 7, 11 \cdot 13, 2 \cdot 17)$

We find $\gcd(2 \cdot 3, 5 \cdot 7) = 1$ and $\gcd(11 \cdot 13, 2 \cdot 17) = 1$, so the next level of the GCD-tree will be $(2 \cdot 3 \cdot 5 \cdot 7, 11 \cdot 13 \cdot 2 \cdot 17)$.

We find $\gcd(2 \cdot 3 \cdot 5 \cdot 7, 11 \cdot 13 \cdot 2 \cdot 17) = 2$ so we now know there is a factor 2 in two of the numbers we started with.

This approach will usually work for finding GCD's of RSA moduli since the most RSA moduli won't share a prime number with another key. However, using this strategy we can find 'double' prime factors.

Example 6.1.2 Suppose we want to find common primefactors in $(2 \cdot 3, 5 \cdot 7, 2 \cdot 11, 3 \cdot 13)$

Again we find $\gcd(2 \cdot 3, 5 \cdot 7) = 1$ and $\gcd(2 \cdot 11, 3 \cdot 13) = 1$, so the next level of the GCD-tree will be $(2 \cdot 3 \cdot 5 \cdot 7, 2 \cdot 11 \cdot 3 \cdot 13)$.

We find $\gcd(2 \cdot 3 \cdot 5 \cdot 7, 2 \cdot 11 \cdot 3 \cdot 13) = 2 \cdot 3$ so we now know there appear factors $2 \cdot 3$ in the numbers we started with.

Sadly, this information is useless since we started with a key of the form $2 \cdot 3$ and we want to find 2 and 3, not $2 \cdot 3$. This problem will occur whenever we have a set of moduli of the form $(a \cdot b, a \cdot c, a \cdot d)$. Therefore we need a way to backtrack our way into the GCD-tree once we have found a common factor. The algorithm for this is given in algorithm 14.

Algorithm: checkTree
Input: A GCD-tree t, a GCD which has been found g and x, y representing respectively the width and height of g in the GCD tree
Output: a, b with $a \cdot b = g$
if $\ell(t_{y-1}) = 2x - 1$ **then**
  checkTree$(y - 1, 2x - 1, g)$
else
  $a = tree_{y-1,2x-1}/g$ $b = tree_{y-1,2x}/g$ **if** $a \neq 1$ **and** $b \neq 1$ **then**
    if $a \notin \mathbb{Z}$ **and** $b \notin \mathbb{Z}$ **then**
      return $\gcd(a, g), g/\gcd(a, g)$
    end
  else if $a \in \mathbb{Z}$ **and** $b \notin \mathbb{Z}$ **then**
    checkTree$(y - 1, 2x - 1, g)$
  else if $a \notin \mathbb{Z}$ **and** $b \in \mathbb{Z}$ **then**
    checkTree$(y - 1, 2x, g)$
  end
end

Algorithm 14: The checkTree algorithm

6.1.1 Running time

We will only discuss the time needed to build the GCD-tree, not the backtracking. This is because whenever a common factor will be found it will most likely be a prime number of a RSA modulus. Therefore, almost no backtracking will need to be done and the time consumed by backtracking will be negligible.

Merging two numbers of d digits using algorithm 12 takes one GCD computation and one multiplication. We already know a GCD computation of two d-bit numbers takes $O\left(d(\log d)^2 \log\log d\right)$ time and a multiplication of two d-bit numbers takes $O\left(d \log d \log\log d\right)$ time. Therefore merging two d-bit numbers takes $O\left(d(\log d)^2 \log\log d\right)$ time.

Since the mergeGCD merges two d-bit numbers into a single $2d$-bit number we end up with a single nd-bit number after $\log n$ levels. This is illustrated in figure 5.

Each level of our recursion tree we need $O\left(nd \log^2 d \log\log d\right)$ time. Therefore the recursion formula for algorithm 13 is given by equation (6).

$$T(n, d) = T\left(2n, \frac{d}{2}\right) + O\left(\log n \cdot nd \log^2 d \log\log d\right) \tag{6}$$

This leads us to the following lemma.

Lemma 6.1.1 Algorithm 13 runs in $O\left(n \log^3 n \log\log n\right)$ time for input of given bitsize.

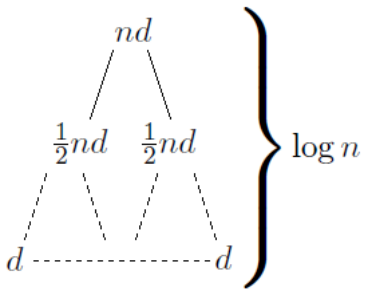

Figure 5: The recursion tree of algorithm mergeGCD

Proof

$$\begin{aligned}
T(n,d) &= \log n \cdot nd \log^2 d \log\log d + T\left(2n, \frac{d}{2}\right) \\
&\leq \log n \cdot nd \log^2 d \log\log d + c \log \frac{n}{2} \cdot nd \log^2 nd \log\log nd \\
&= \log n \cdot nd \log^2 d \log\log d + c\,(\log n - 1)\, nd \log^2 nd \log\log nd \\
&= cnd \log n \cdot nd \log^2 nd \log\log nd \\
&\quad + cnd\left(\log n \log^2 \log\log d - \log^2 nd \log\log nd\right) \\
&\leq cnd \log n \cdot nd \log^2 nd \log\log nd
\end{aligned}$$

Choosing d as a constant gives us $T(n) = O\left(n \log^3 n \log\log n\right)$

6.2 Nadia Heninger's Efficient all-pairs GCD's computation

A recent paper by Nadia Heninger, Zakir Durumeric, Eric Wustrow and J. Alex Halderman [5] has published a comparable algorithm for the same purpose. The idea in this algorithm is to first do one large multiplication to then quickly compute all GCD's. For this algorithm the following lemma is used.

Lemma 6.2.1 $\gcd(N_1, N_2 \cdot \ldots \cdot N_m) = \gcd(N_1, \frac{N_1 N_2 \cdot \ldots \cdot N_m \mod N_1^2}{N_1})$.
Proof

$$\begin{aligned}
\gcd(N_1, N_2 \cdot \ldots \cdot N_m) &= g \\
\gcd(N_1^2, N_1 N_2 \cdot \ldots \cdot N_m) &= N_1 \cdot g \\
\gcd(N_1^2, N_1 N_2 \cdot \ldots \cdot N_m + x N_1^2) &= N_1 \cdot g \ \forall x \in \mathbb{Z} \\
\gcd(N_1^2, N_1 N_2 \cdot \ldots \cdot N_m \mod N_1^2) &= N_1 \cdot g \\
\gcd\left(N_1, \frac{N_1 N_2 \cdot \ldots \cdot N_m \mod N_1^2}{N_1}\right) &= g \quad \blacksquare
\end{aligned}$$

The idea now is to compute the product of all N_i's and then find $\frac{\prod_{i \neq j} N_j \mod N_i^2}{N_i}$. For this we need a product tree.

Definition 6.2.1 A *product tree* of a set S is a tree with the set S at the bottom level. For all elements x in the tree, with children l and r, at levels above the lowest we have $x = l \cdot r$.

Algorithm 15 is a very straightforward algorithm to create a product tree.

Algorithm: productTree
Input: an array S
Output: T, a product tree of S
$T_1 = S$
for $i = 2 \rightarrow \lceil \log(\ell(S)) + 1 \rceil$ **do**
  $T_i = \{1, ..., 1\}$
  for $j = 1 \rightarrow \ell(T_{i-1,.})$ **do**
    $T_{i,\lceil j/2 \rceil} * = T_{i-1,j}$
  end
end

Algorithm 15: The productTree algorithm

Now we have a product tree we can use this tree to compute $z_i = \frac{\prod_{i \neq j} N_j \mod N_i^2}{N_i}$ for all i. From there we just need to compute $\gcd(z_i, N_i)$. This is done in algorithm 16.

Algorithm: computeGCDs
Input: an array S
Output: $\gcd(S_i, \prod_{i \neq j} S_j)$ for all i
$T =$productTree(S)
for $i = \lceil \log(\ell(S)) + 1 \rceil \rightarrow 1$ **do**
  for $j = 1 \rightarrow \ell(T_{i.}$ **do**
    $T_{ij} = T_{i+1,\lceil j/2 \rceil} \mod T_{ij}^2$
  end
end

Algorithm 16: The computeGCDs algorithm

The entire process of this algorithm is shown in figure 6.

Note that this algorithm only computes $\frac{\prod_{i \neq j} N_j \mod N_i^2}{N_i}$ which does not guarantee factorization. If a RSA modulus shares both prime factors with different moduli the GCD will be the key itself. Heninger's paper does not provide a real solution for this. In their research they simply used basic quadratic algorithms to factorize the moduli since this only occured a few times.

The running time of this algorithm is $n \log^2(n) \log \log n$, which is shown in [5].

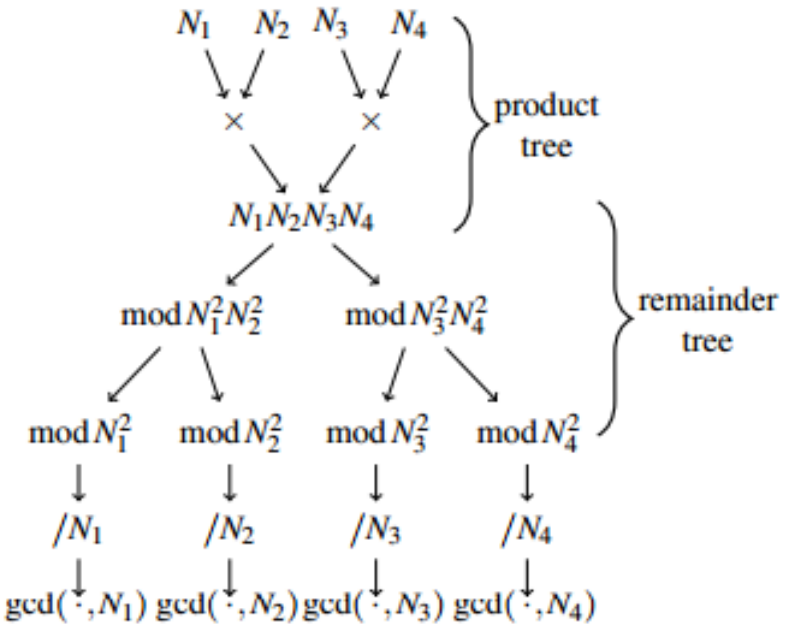

Figure 6: The process of Nadia Heninger's algorithm. (Source: [5])

6.3 Bernstein's Coprime Base algorithm

This algorithm builds a coprime base of a given input. The definition of a coprime base is given below.

Definition 6.3.1 A set P is a *coprime base* for S if $\gcd(P_i, P_j) = 1$ for all P_i, P_j with $i \neq j$ and each element in S is a product of powers of elements of P.

We first describe an algorithm to determine the coprime base of two integers which is given in algorithm 17. This is a fairly straightforward algorithm. It factors out the GCD of the input in both integers and returns the GCD and the two integers without a factor of their GCD in it. The reason the algorithm looks somewhat long is that we aren't interested in returning 1 or double numbers in our coprime base.

Algorithm: coprimeBase
Input: 2 integers a and b
Output: the coprime base of a and b
$g = \gcd(a, b)$
if $g = 1$ **then**
  if $a = 1$ **then**
    return $\{b\}$
  else
    if $b = 1$ **then**
      return $\{a\}$
    else
      return $\{a, b\}$
    end
  end
else
  while $a/g \in \mathbb{N}$ **do**
    $a = a/g$
  end
  while $b/g \in \mathbb{N}$ **do**
    $b = b/g$
  end
  if $a = 1$ **and** b=*1* **then**
    return $\{g\}$
  else if $a = 1$ **then**
    return $\{b, g\}$
  else if $b = 1$ **then**
    return $\{a, g\}$
  else
    return $\{a, b, g\}$
  end
end

Algorithm 17: The coprimeBase algorithm

Next we describe an algorithm to find the powers of coprimes of b in a, algorithm 18. A more intuitive notation of this is $\gcd(a, b^\infty)$. This algorithm is also straightforward. We determine $\gcd(a, b)$ and keep factoring out powers of b in a until there are none left.

Algorithm: coprimeInfo
Input: 2 integers $a \neq 1$ and $b \neq 1$
Output: $\gcd(a, b)$ and powers of coprimes of b in a, and powers of coprimes of b not in a
$gcd = x = \gcd(a, b)$
$y = a/x$
while $g \neq 1$ **do**
  $g = \gcd(x, y)$
  $x = x \cdot g$
  $y = y/g$
end
return gcd, x, y

Algorithm 18: The coprimeInfo algorithm

We could use this algorithm with an element in a linear fashion on a coprime base to find all factors the element has in common with the coprime base. However, a faster version exists. This algorithm for this is given in algorithm 19. The idea is fairly straightforward. We multiply all elements in the coprime base and check if the result has a factor in common with the integer which is being tested. If there is no factor in common, we are done. If there is, we do two recursive calls on the same algorithm. Once with the first $n/2$ elements, once with the last $n/2$ elements.

Algorithm: split
Input: An integer a and a coprime base P with n elements
Output: The powers of primes of P_i in a for all i
if $n = 0$ **then**
  return 1
else
  $g, b, c =$coprimeInfo$(a, \prod P)$
  if $n = 1$ **then**
    return P_1, b
  else
    $k = \lfloor n/2 \rfloor$
    $x =$split$(b, \{P_1, ..., P_k\})$
    $y =$split$(b, \{P_{k+1}, ..., P_n\})$
    return $x \cup y$
  end
end

Algorithm 19: The split algorithm

Now we have an algorithm to find common factors of an element with a coprime base, we can use this to extend the coprime base. We find all common factors of elements in the coprime base and the new element and then individually determine their coprime base and build a new coprime base from there.

This is done in algorithm 20.

Algorithm: extendBase
Input: An integer a and a coprime base P
Output: a coprime base of $P \cup \{a\}$
if $P = \{\}$ **then**
  return b
else
  $x = \prod P$
  $g, a, r =$coprimeInfo(b, x)
  if $r \neq 1$ **then**
    return $P \cup \{r\}$
  else
    $S =$split(a, P)
    $P =$
    for $i = 1 \rightarrow \ell(S)$ **do**
      $P = P \cup$ coprimeBase(split(S_i))
    end
  end
end

Algorithm 20: The extendBase algorithm

We now have a way to extend a coprime base by one element, but we want to do more than that. We want to merge two coprime bases. For this, Daniel Bernstein came with a new idea. We will look at individual bits of the location of elements in their array. The formal definition of this is given below.

Definition 6.3.2 The i^{th} bit of j in its binary notation, starting from 0 at the least significant bit, is named $bit_i(j)$.

Lemma 6.3.1 Let $q_0, q_1, ..., q_{n-1} \in \mathbb{R}$ and unique and q_i coprime to q_j for $i \neq j$. Let $b \geq 1$ be an integer with $2^b \geq n$. Define $x(e, i) = \prod_{\text{bit}_i(k)=e} q_j$. If a coprime set P is a base for $\{x(0,0), x(0,1), ..., x(0,b-1), x(1,0), x(1,1), ..., x(1,b-1)\}$ then it is also a base for $\{q_0, ..., q_{n-1}\}$.

Proof of this lemma can be found in [7].

We now describe an algorithm to create a coprime base for $\{x(0,0), x(0,1), ..., x(0,b-1), x(1,0), x(1,1), ..., x(1,b-1)\}$ which, according to lemma 6.2.1 will be a coprime base for $\{q_0, ..., q_{n-1}\}$. The algorithm that does this is described by algorithm 21.

Algorithm: mergeBases
Input: 2 coprime bases P and Q
Output: the coprime base of $P \cup Q$
$S = P$ **for** $i = 1 \to \lceil \log(\ell(Q)) \rceil$ **do**
  $x = \prod_{\text{bit}_i(j)=1} Q_j$
  $S =$extendBase(S, x)
  $x = \prod_{\text{bit}_i(j)=0} Q_j$
  $S =$extendBase(S, x)
end
return S

Algorithm 21: The mergeBases algorithm

We now examine how algorithm 21 merges two coprime bases.

Example 6.3.1 Suppose we want to merge bases P and Q with $P = \{2, 3, 5, 7\}$ and $Q = \{2, 11, 13, 17\}$.

We find $\prod_{\text{bit}_0(j)=1} Q_j = 11 \cdot 17$ so we find the coprime base of $\{2, 3, 5, 7\}$ and $11 \cdot 17$ which is $\{2, 3, 5, 7, 11 \cdot 17\}$.

Next, we find the coprime base of $\{2, 3, 5, 7, 11 \cdot 17\}$ and $\prod_{\text{bit}_0(j)=0} Q_j = 2 \cdot 13$ which is $\{2, 3, 5, 7, 11 \cdot 17, 13\}$.

We continue this process for bit 1 of Q_j and we find the coprime base of $\{2, 3, 5, 7, 11 \cdot 17, 13\}$ and $\prod_{\text{bit}_1(j)=1} Q_j = 13 \cdot 17$ which is $\{2, 3, 5, 7, 11, 13, 17\}$.

Finally, the coprime base of $\{2, 3, 5, 7, 11, 13, 17\}$ and $\prod_{\text{bit}_1(j)=0} Q_j = 2 \cdot 11$ is $\{2, 3, 5, 7, 11, 13, 17\}$.

Now the initial goal we had, building a coprime base from a set of integers becomes straightforward. Algorithm 22 does this in a divide-and-conquer fashion.

Algorithm: createBase
Input: a set S with n elements
Output: the coprime base of S
if $\ell(S) \leq 1$ **then**
  return S
else
  $k = \lfloor n/2 \rfloor$
  P =createBase($\{S_1, ..., S_k\}$)
  Q =createBase($\{S_{k+1}, ..., S_n\}$)
  return mergeBases(P, Q)
end

Algorithm 22: The createBase algorithm

This algorithm runs in $O(n \log n \mu(\log n))$ time where $\mu(n)$ is the amount of time needed for a multiplication. This is $n \log n \log \log n$ with Shönhagen-Strassen multiplication. Proof for this can be found in [7].

6.4 Implementation

We have implemented these algorithms in Wolfram Mathematica 7.0 and tested them on a desktop computer. The code for this can be found in the appendix.

6.4.1 Building a GCD tree

Figure 7 shows the running time of the mergeGCD algorithm with increasingly large inputs. The algorithm takes roughly 8 seconds for every 1000 moduli to be examined. What stands out is the small peaks just before (2^i)-bit numbers. These can be explained by the structure of the algorithm. The length of the array to be examined at the next level of the algorithm is $\lceil \frac{n}{2} \rceil$. For $(2^i - 1)$-bit numbers this means every level the length of the array is rounded up. For 2^i-bit numbers there is no rounding up at all.

Building the GCD tree will take the mergeGCD algorithm roughly the same amount of time regardless of how many common prime factors are found. However, filtering out the prime factors from the results can become increasingly difficult as the amount of common prime factors grow. Therefore, this algorithm is only really suited to factorize if there are few factors to be found.

6.4.2 Nadia Heninger's Efficient all-pairs GCD's computation

Figure 8 shows the running time of Nadia Heninger's algorithm with increasingly large inputs. The running times seem to increase nearly linear which coincides with the running time given in the paper.

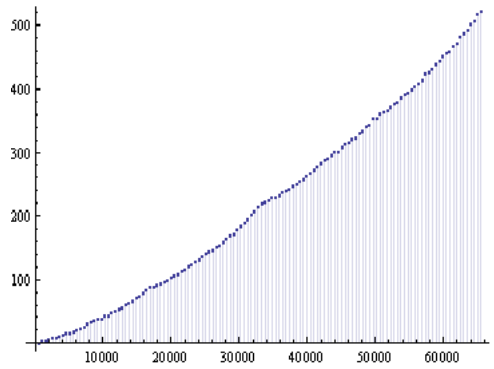

Figure 7: The running time of algorithm mergeGCD with n 1024-bit RSA moduli

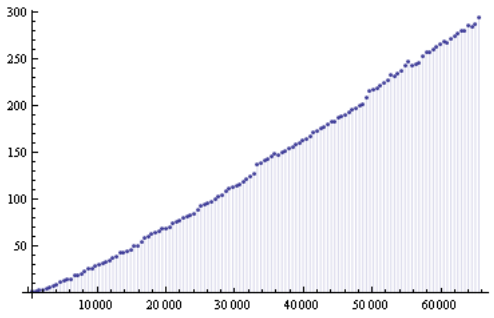

Figure 8: The running time of algorithm computeGCDs with n 1024-bit RSA moduli

The algorithm takes roughly 4.5 seconds for every 1000 moduli to be examined.

6.4.3 Bernstein's coprime base algorithm

Figure 9 shows the running time of Dan Bernstein's algorithm with increasingly large inputs. The running times seem to be increasing a bit faster than linear, but this can be attributed to the relatively small inputs.

6.4.4 Comparison

It's easy to see the order of the speed of the algorithms here. Heninger's algorithm is faster than building a GCD tree and Bernstein's algorithm is slower than both of them. All algorithms are quasi-linear but Heninger's algorithm seems to be roughly 1.7 times as fast as building a GCD tree and Bernstein's algorithm doesn't appear to be in the same league in our implementation.

Aside from the speed, building a GCD tree can get very messy when coprime factors are less scarce and it can become hard to actually find the common factors. Heninger's algorithm isn't perfect on this matter either but it seems to

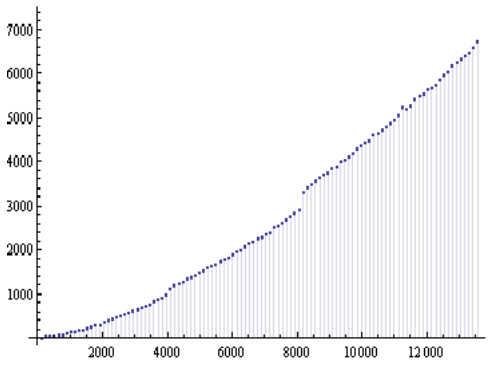

Figure 9: The running time of algorithm Bernstein's coprime base algorithm with n 1024-bit RSA moduli

be easier to filter out the actual moduli when the factorization didn't work out perfectly. Bernstein's algorithm is a clear winner on this issue. The resulting coprime base does not need any additional work to find the coprime factors. However, since we are working with RSA moduli this might not be too important. After all, if the moduli are properly generated no coprime factors should be found.

Arjen K. Lenstra, James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung and Christophe Watcher have used coprime factorization to factorize RSA moduli, see [6]. They have managed to factorize 0.2% of a set of 11.7 RSA moduli, but they do not describe their GCD-method in detail. Nadia Heninger and Zakir Durumeric and Eric Wustrow and J. Alex Halderman have done similar research on this matter and their team managed to factorize 0.4% of a set of 11.1 million RSA moduli.

7 References

[1] Damien Stehlé and Paul Zimmermann. A Binary Recursive Gcd Algorithm. In Ducan A. Buell's Algorithmic Number Theory, 6th International Symposium, ANTS-VI, Burlington, VT, USA, June 13-18, 2004, Proceedings. pp. 411-425.

[2] Niels Möller. On Schönhages Algorithm and Subquadratic Integer Gcd Computation. Math. Comput. vol 77, no 261, 2008. pp. 589-607.

[3] Tudor Jebelean. A double-digit Lehmer-Euclid algorithm for finding the GCD of long integers. Journal of Symbolic Computation 19 (1995), no 1-3, pp. 145-157.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, Second Edition. MIT Press and McGraw-Hill, 2001. pp. 73-90.

[5] Nadia Heninger and Zakir Durumeric and Eric Wustrow and J. Alex Halderman. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices. 2012. Proceedings of the 21st USENIX Security Symposium.

[6] Arjen K. Lenstra, James P. Hughes, Maxime Augier, Joppe W. Bos, Thorsten Kleinjung and Christophe Watcher. Ron was wrong, Whit was right. 2012. Cryptology ePrint Archive, Report 2012/064. `http://eprint.iacr.org/`

[7] Daniel J. Bernstein. Factoring into coprimes in essentially linear time. 2005. Journal of Algorithms 54 (1), 1–30

8 Appendix

8.1 Implementation of Stehlé and Zimmermann's algorithm

```
splitnumber = Function[{a, k}, {Floor[a/2^k], Mod[a, 2^k]}];

split = Function[{a}, Block[{k = IntegerExponent[a, 2]}, {a/2^k, k}]];

cmod = Function[{a, b}, If[2 a == b, a, Mod[a, b, -b/2]]];

fastGB = Function[{a, b},
   Block[{A, B, n, q, i, expa, expb},
    {A, expa} = split[a]; {B, expb} = split[b]; n = expb - expa + 1;
    q = 1; For[i = 2, i < 2 n, i *= 2, q = Mod[q + q (1 - B q), 2^i]];
    q = cmod[-A q, 2^n];
    {q, a + q b / 2^(n - 1)}
    ]];

gbgcd = Function[{a, b},
   If[b == 0, a/2^IntegerExponent[a, 2], gbgcd[b, fastGB[a, b][[2]]]]];

gbgcd2 := Function[{a, b},
  Block[{M = {{1, 0}, {0, 1}}, j = 0, r0 = a, r1 = b, r2, q},

   While[IntegerExponent[r1, 2] <= IntegerLength[a, 2]/2,
    j += IntegerExponent[r1, 2] - IntegerExponent[r0, 2];
    {q, r2} = fastGB[r0, r1];
    M = {{0, 1}, {1,
        q/2^(IntegerExponent[r1, 2] - IntegerExponent[r0, 2])}}.M;
    {r0, r1} = {r1, r2}
    ];
   {j, 2^j M, r0/2^j, r1/2^j}
   ]
  ]

halfGBgcd = Function[{a, b},
   (*Input: a=r_ 0 (odd), b=r_ 1 (even)*)
   (*Output: Number of digits 'won', Multiplication matrix, r_i,
   r_(i+1)*)
   Block[{k = Floor[IntegerLength[a, 2]/2], k1, a0, a1, b0, b1, j1,
     R1, c1, d1, A, B, j0, q, r, k2, B0, B1, r0, r1, j2, R2, c2, d2,
     c, d},

    If[IntegerExponent[b, 2] > k, {0, {{1, 0}, {0, 1}}, a, b},

     k1 = Floor[k/2];
     {a1, a0} = splitnumber[a, 2 k1 + 1];
```

```
     {b1, b0} = splitnumber[b, 2 k1 + 1];
     {j1, R1, c1, d1} =
      If[b0 > 2^100, halfGBgcd[a0, b0], gbgcd2[a0, b0]];
     {A, B} = 2^(2 k1 + 1 - 2 j1) R1.{a1, b1} + {c1, d1};
     j0 = IntegerExponent[B, 2];
     If[j0 + j1 > k, {j1, R1, A, B},
      {q, r} = fastGB[A, B];
      k2 = k - (j0 + j1);
      r = r/2^j0;
      B = B/2^j0;
      {B1, B0} = splitnumber[B, 2 k2 + 1];
      {r1, r0} = splitnumber[r, 2 k2 + 1];
      {j2, R2, c2, d2} =
       If[B0 > 2^100, halfGBgcd[B0, r0], gbgcd2[B0, r0]];
      {c, d} = 2^(2 k2 + 1 - 2 j2) R2.{B1, r1} + {c2, d2};
      {j1 + j0 + j2, R2.{{0, 2^j0}, {2^j0, q}}.R1, c, d}
      ]
     ]
    ]
   ];

fastGBgcd = Function[{a, b},
   Block[{j, R, A, B, q, r},
    {j, R, A, B} = halfGBgcd[a, b];
    If[B == 0, Abs[A],
     {q, r} = fastGB[A, B];
     fastGBgcd[B/2^IntegerExponent[B, 2], r/2^IntegerExponent[B, 2]]
     ]
    ]
   ];

startupGCD[a_Integer?OddQ, b_Integer?EvenQ] := fastGBgcd[a, b];
startupGCD[a_Integer?EvenQ, b_Integer?OddQ] := fastGBgcd[b, a];
startupGCD[a_Integer?OddQ, b_Integer?OddQ] := fastGBgcd[a, 2 b];
startupGCD[a_Integer?EvenQ, b_Integer?EvenQ] :=
  Block[{am, ae, bm, be},
   {am, ae} = split[a]; {bm, be} = split[b];
   If[ae > be, fastGBgcd[bm, am 2] 2^Min[ae - 1, be],
    fastGBgcd[am, bm 2] 2^Min[be - 1, ae]]];
```

8.2 Implementation of Niels Möller's algorithm

```
splitnumber = Function[{a, k}, {Floor[a/2^k], Mod[a, 2^k]}];

sdiv = Function[{a, b, s},
   Block[{q},
```

```
    If[Abs[a] > Abs[b],
     q = IntegerPart[a/b];
     If[IntegerLength[a - q b, 2] <= s, q = q - 1];
     {q, a - q b, {{1, -q}, {0, 1}}},

     q = IntegerPart[b/a];
     If[IntegerLength[b - q a, 2] <= s, q = q - 1];
     {q, b - q a, {{1, 0}, {-q, 1}}}
     ]
    ]
   ];

sgcd = Function[{a, b, s},
   Block[{c = a, d = b, m = {{1, 0}, {0, 1}}, q, m2},
    While[IntegerLength[Abs[c] - Abs[d], 2] > s,
     If[Abs[c] > Abs[d],
      {q, c, m2} = sdiv[c, d, s],
      {q, d, m2} = sdiv[c, d, s]
      ];
     m = m2.m
     ];
    {c, d, m}
    ]
   ];

halfGCD = Function[{r0, r1},
   Block[{n, s, p1, p2, n1, n2, a1, a0, b1, b0, c, d, m, a = r0,
     b = r1, m2, q, r},
    n = Max[IntegerLength[a, 2], IntegerLength[b, 2]];
    s = Ceiling[n/2];

    If[Min[IntegerLength[a, 2], IntegerLength[b, 2]] >
      Floor[3 n/4] + 2,
     p1 = Floor[n/2];
     n1 = n - p1;
     {a1, a0} = splitnumber[a, p1];
     {b1, b0} = splitnumber[b, p1];
     {c, d, m} =
      If[Abs[a1] > 2^100, halfGCD[a1, b1],
       sgcd[a1, b1, Floor[IntegerLength[Max[a1, b1], 2]/2]]];
     {a, b} = 2^p1 {c, d} + m.{a0, b0},

     m = {{1, 0}, {0, 1}}
     ];

    While[
     Max[IntegerLength[a, 2], IntegerLength[b, 2]] > Ceiling[3 n/4] &&
```

```
      IntegerLength[Abs[a] - Abs[b], 2] > s,
     {q, r, m2} = sdiv[a, b, s];
     If[Abs[a] > Abs[b], a = r, b = r];
     m = m2.m;
     ];

    If[Min[IntegerLength[a, 2], IntegerLength[b, 2]] > s + 2,
     n = Max[IntegerLength[a], IntegerLength[b, 2]];
     p2 = 2 s - n + 1;
     n2 = n - p2;
     {a1, a0} = splitnumber[a, p2];
     {b1, b0} = splitnumber[b, p2];
     {c, d, m2} =
      If[Abs[a1] > 2^100, halfGCD[a1, b1],
       sgcd[a1, b1, Floor[IntegerLength[Max[a1, b1], 2]/2]]];
     {a, b} = 2^p2 {c, d} + m2.{a0, b0};
     m = m2.m
     ];

    While[IntegerLength[Abs[a] - Abs[b], 2] > s,
     {q, r, m2} = sdiv[a, b, s];
     If[Abs[a] > Abs[b], a = r, b = r];
     m = m2.m;
     ];
    {a, b, m}
    ]
   ];

fastGCD = Function[{a, b},
   Block[{c, d, m, q, r},
    {c, d, m} = halfGCD[a, b];
    If[a == c && b == d, Abs[sgcd[c, d, 0][[1]]],
     fastGCD[c, d]
     ]
    ]
   ];
```

8.3 Implementation of the MergeGCD algorithm

```
mergeGCD = Function[{n},
   Block[{g, i, s = Floor[Length[n]/2], n2},
    n2 = Array[0 &, Ceiling[Length[n]/2]];
    For[i = 1, i <= s, i++,
     g = GCD[n[[2 i - 1]], n[[2 i]]];
     If[g != 1, output = Join[output, {{level, i, g}}]
      ];
     n2[[i]] = (n[[2 i - 1]]/g) n[[2 i]]
```

```
     ];
    If[Mod[Length[n], 2] == 1, n2[[s + 1]] = n[[Length[n]]]];
    n2
    ]
   ];

output = {};
buildTree = Function[{n},
   Block[{level, tree = Array[0 &, 1 + Ceiling[Log[2, Length[n]]]]},
    tree[[1]] = n;
    For[level = 1, level <= Ceiling[Log[2, Length[n]]], level++,
     tree[[level + 1]] = mergeGCD[tree[[level]]]
     ];
    output
    ]
   ];

treecheck = Function[{level, i, g},
   Block[{a, b, c},
    If[Length[tree[[level - 1]]] == 2 i - 1,
     treecheck[level - 1, 2 i - 1, g],
     a = tree[[level - 1]][[2 i - 1]]/g;
     b = tree[[level - 1]][[2 i]]/g;
     If[a != 1 && b != 1,
      If[Not[IntegerQ[a]] && Not[IntegerQ[b]],
       c = GCD[a, g];
       {c, g/c}
       ];
      If[IntegerQ[a] && Not[IntegerQ[b]],
       treecheck[level - 1, 2 i - 1, g]
       ];
      If[Not[IntegerQ[a]] && IntegerQ[b],
       treecheck[level - 1, 2 i, g]
       ]
      ]
     ]
    ]
   ];
```

8.4 Implementation Nadia Heninger's Efficient all-pairs GCD's computation

```
productTree = Function[{S},
   Block[{T, x, y, i, j},
    T = Array[0 &, Ceiling[Log[2, Length[S]] + 1]];
    T[[1]] = S;
    For[i = 2, i <= Length[T], i++,
```

```
     T[[i]] = Array[1 &, Ceiling[Length[T[[i - 1]]]/2]];
     For[j = 1, j <= Length[T[[i - 1]]], j++,
      T[[i]][[Ceiling[j/2]]] *= T[[i - 1]][[j]]
      ]
     ];
    T
    ]
   ];

computeGCD = Function[{A},
   Block[{T, i, j},
    T = productTree[A];
    For[i = Length[T] - 1, i >= 1, i--,
     For[j = 1, j <= Length[T[[i]]], j++,
      T[[i]][[j]] = Mod[T[[i + 1]][[Ceiling[j/2]]], T[[i]][[j]]^2]
      ]
     ];
    GCD[T[[1]]/A, A]
    ]
   ];
```

8.5 Implementation of Dan Bernstein's coprime base algorithm

```
product = Function[{S},
   Block[{x, y},
    If[Length[S] == 0, 1,
     If[Length[S] == 1, S[[1]],
      x = product[Take[S, Floor[Length[S]/2]]];
      y = product[Take[S, -Ceiling[Length[S]/2]]];
      x y
      ]
     ]
    ]
   ];

coprimeInfo = Function[{a, c},
   Block[{gcd, x, y, n, g},
    x = GCD[a, c];
    gcd = x;
    y = a/x;
    For[n = 1, g != 1, n++,
     g = GCD[x, y];
     x = x g;
     y = y/g
     ];
```

```
    {gcd, x, y}
    ]
   ];

split = Function[{a, Q},
   Block[{P = Q, g, b, c, x, y},
    If[Length[P] == 0, 1,
     {g, b, c} = coprimeInfo[a, product[P]];
     If[Length[P] == 1, {{P[[1]], b}},
      x = split[b, Take[P, Floor[Length[P]/2]]];
      y = split[b, Take[P, -Ceiling[Length[P]/2]]];
      Join[x, y]
      ]
     ]
    ]
   ];

addToCoprimeBase = Function[{Q, b},
   Block[{P = Q, i, p, x, g, a, r, S},
    If[P == {}, If[b != 1, b],
     x = product[P];
     {g, a, r} = coprimeInfo[b, x];
     If[r != 1, Append[P, r],
      S = split[a, P];
      P = {};
      For[i = 1, i <= Length[S], i++,
       P = Union[P, coprimeBase @@ S[[i]]];
       ];
      P
      ]
     ]
    ]
   ];

coprimeBase = Function[{A, B},
   Block[{a = A, b = B, g},
    g = GCD[a, b];
    If[g == 1,
     If[a == 1, {b},
      If[b == 1, {a}, {a, b}]],
     While[IntegerQ[a/g], a = a/g];
     While[IntegerQ[b/g], b = b/g];
     If[a == 1 && b == 1, {g},
      If[a == 1, {b, g},
       If[b == 1, {a, g},
        {a, b, g}]
       ]
```

```
      ]
     ]
    ]
   ];

mergeBases = Function[{P, Q},
   Block[{i, s, t, x, S = P},
    For[i = 1, i <= Length[Q], i *= 2,
     s = Array[Mod[Floor[#/i], 2] &, Length[Q]];
     t = Pick[Q, s, 1];
     x = product[t];
     S = addToCoprimeBase[S, x];
     t = Pick[Q, s, 0];
     x = product[t];
     S = addToCoprimeBase[S, x];
     ];
    S
    ]
   ];

arrayCoprimeBase = Function[{S},
   Block[{P, Q},
    If[Length[S] <= 1, S,
     P = arrayCoprimeBase[Take[S, Floor[Length[S]/2]]];
     Q = arrayCoprimeBase[Take[S, -Ceiling[Length[S]/2]]];
     mergeBases[P, Q]
     ]
    ]
   ];

MultiplicationGCD = Function[{n},
   Block[{output = {}, t, i, s, x, y, g, p,
     l = Ceiling[Log[2, Length[n]]]},
    p = product[n];
    For[i = 1, i <= Length[n], i *= 2,
     s = Array[Mod[Floor[#/i], 2] &, Length[n]];
     t = Pick[n, s, 1];
     x = product[t];
     y = Mod[p, x^2]/x;
     g = GCD[x, y];
     If[g != 1, output = Union[output, {g}]]
     ];
    output
    ]
   ];
```