Universiteit Utrecht

Extending Stein's GCD algorithm

and a comparison to Euclid's GCD algorithm

Joris Barkema

Bachelor's Thesis

Utrecht University
Bachelor of Science (BSc) Mathematics

Utrecht, 14-6-2019

Supervisor - dr. G. Tel

Abstract

Euclid's algorithm and Stein's binary GCD algorithm are the two most well-known GCD algorithms. It has already been proven that Euclid's algorithm is $\mathcal{O}(n^2)$, but we aim to provide a more intuitive and thorough proof using an in-depth look at the cost of the modulo operation. An extended version of Stein's algorithm is also provided with proof of correctness. While there exist other extended versions of Stein's algorithm, this version distinguishes itself through its relative simplicity and elegant use of properties of Bézout's identity. Finally, a comparison of the performance of both algorithms with their extended versions is provided.

Contents

1 Introduction

The Greatest Common Divisor (GCD) of two numbers p and q is the greatest number which is a divisor of both of the numbers. The oldest known algorithm to calculate the GCD is Euclid's algorithm, but faster algorithms are still being developed. In this paper we will discuss Euclid's algorithm and Stein's binary GCD algorithm.

Some GCD algorithms, among them Euclid's algorithm and Stein's algorithm, can be extended to return the Bézout coëfficients along with the GCD. The Bézout coëfficients are used to find multiplicative inverse elements in modular arithmetic. An important application is calculating the private key in the public-key cryptosystem RSA [Rivest et al.1978].

In this paper we will first perform an extensive analysis of the modulo operation in C# in section 2. We will look at the cost of this operation on numbers of different sizes to get a stricter algorithmic complexity, which we will need in section 3 where we will provide our proof that Euclid's algorithm is $\mathcal{O}(n^2)$.

In section 4 Bézout's identity is discussed along with the two general strategies to calculate the coëfficients. Then in section 5 we give the proof of correctness of our extended version of Stein's binary GCD algorithm, with finally in section 6 a comparison of the performance of the discussed algorithms, along with a built-in GCD algorithm of the *BigInteger* class.

2 Modulo operation experiments

Before we can begin analyzing the complexity of Euclid's algorithm, we need to further inspect the complexity of the modulo operation in C#. The modulo operation is strongly related to the integer division operation, the difference being that instead of the quotient, the remainder is returned. This could lead to the assumption that both operations are implemented in the same manner, most well-known being long division. It is well known that integer division using long division is at most $\mathcal{O}(n^2)$, where n is the bitsize of the largest of the two numbers. However, with some extra work we will find a stronger limit.

As the actual implementation of the modulo operation is not accessible, the reasoning in this section is based on experimental results. Based on these results an explanation of how this operation could be implemented is given. The program with which the experiments have been conducted can be found in appendix A.

2.1 Equal bitsizes

The first experiment calculated the modulus of two numbers, represented as C# *BigIntegers*, of an equal amount of random bits. Each value in the graph represents the average of ten test runs made out of 100 pairs of two numbers of the indicated amount of bits. Within each test run the calculations are repeated

10.000 times to prevent inaccuracies in the time measurement. The result of this experiment can be seen in figure 1.

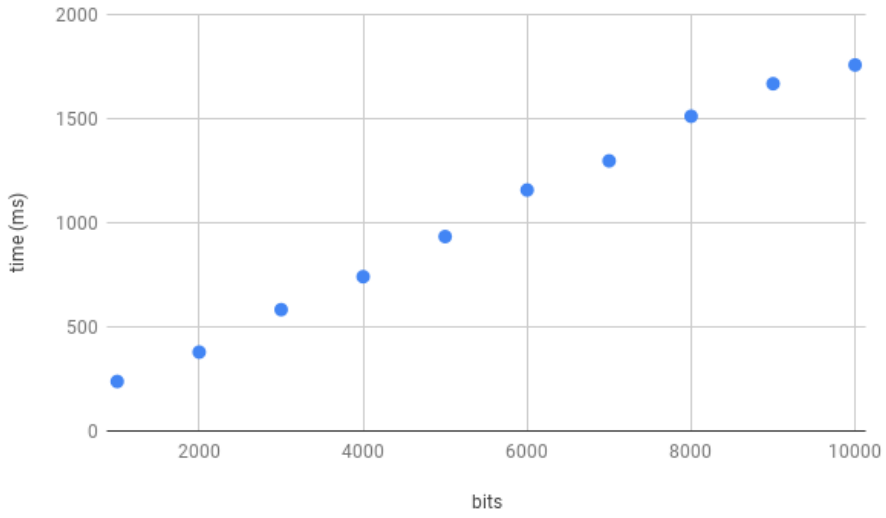

Figure 1: Modulo operation with indicated bitsizes

In this figure we can clearly see a roughly linear trend. However, we can not simply assume the modulo operation is $\mathcal{O}(n)$. This is because although the bits are randomly generated, the bitsize of the numbers will not be random. The bitsize of a number is the left-most bit that is set to 1. To illustrate: The odds that a number generated from 1000 random bits will have a bitsize smaller than 990 bits is 1 in $2^{10} = 1024$. This means that in the previous experiment the two numbers of which the modulus was calculated were almost always very close to each other in bitsize. Consequently, we can not safely conclude that the growth in computational cost as the bitsizes of the two numbers increase is linear in all cases, even though this figure suggests so.

2.2 Different bitsizes

To compare the cost when using numbers of different bitsizes another experiment was conducted. In this experiment one number was always created from 10.000 random bits, while the amount of bits for the second number was varied. This time the calculations within each test run were only repeated 100 times as this was sufficient to prevent inaccuracies in the time measurement. The result of this experiment can be seen in figure 2.

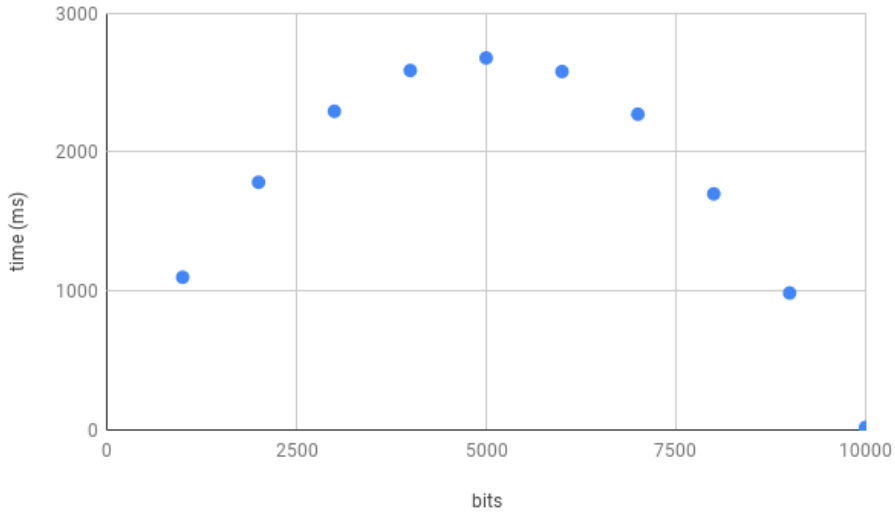

Figure 2: Modulo operation with 10.000 bits and the indicated bitsize

In this figure we see a parabolic shape with a maximum around 5000 bits, half the bitsize of the first number. This gives a first indication that the modulo operation may indeed be implemented using long division. The cost of the long division is proportional to the bitsize of the second number, the cost of one step, multiplied by the bitsize of the quotient, the maximum amount of steps. Note that the bitsize of the quotient is roughly equal to the difference in bitsize between two numbers. This would then indeed have a maximum when the bitsize of the second number is half the bitsize of the first number.

2.3 Half bitsize

If the modulo operation indeed uses long division, we would expect the modulo operation of a number with a certain bitsize and another number with half its bitsize to be quadratic. Another experiment was performed to test this, performing the modulo operation on a number with the indicated bitsize and a second number with half the indicated bitsize in the same way as the previous experiments, repeated 100 times within each test run. The result of this experiment can be seen in figure 3.

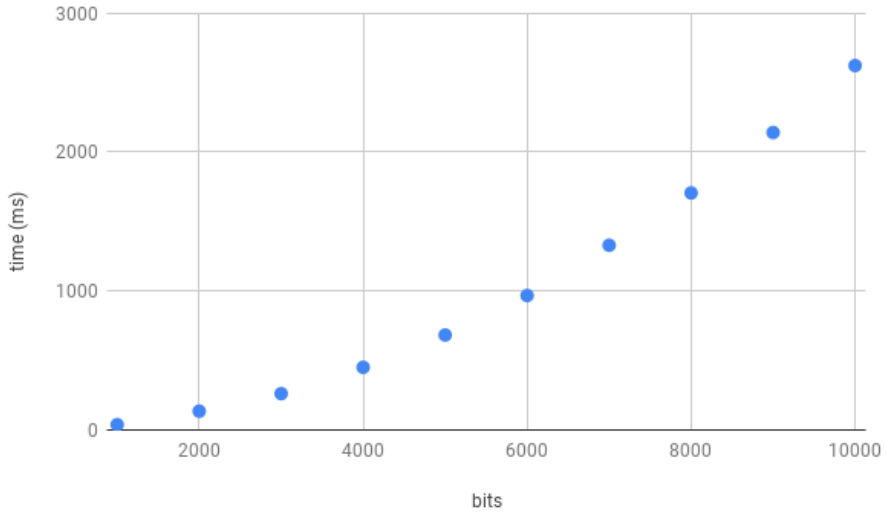

Figure 3: Modulo operation with the indicated bitsize and half of the indicated bitsize

This figure further supports the idea that the modulo operation uses long division, because we find a quadratic trend as expected. More precisely, the conducted experiments lead to the following observation, which we will need when proving the algorithmic complexity of Euclid's algorithm is $\mathcal{O}(n^2)$.

Let $D(n_1, n_2)$ be the computational cost of the modulo operation for a number with n_1 bits and n_2 bits, then

Observation 2.1. $\exists d : D(n_1, n_2) \leq d \cdot n_2 \cdot (n_1 - n_2) + \mathcal{O}(n)$

The $\mathcal{O}(n)$ term is needed because it is possible that $n_1 = n_2$, in which case $n_1 - n_2 = 0$, nullifying d. It is not possible for n_2 to be zero, because that would mean dividing by zero.

3 Complexity of Euclid's algorithm

As Euclid's algorithm is a widely known and used algorithm, proof of the correctness of this algorithm will be omitted, and only an analysis of the algorithmic complexity will be provided. Although it has already been proven that Euclid's algorithm is $\mathcal{O}(n^2)$, for example by Eric Bach and Jeffrey Shallit [Bach et al.1996, Chapter 4], this section aims to provide a more complete and intuitive proof. In order to do this we will first expand on observation 2.1.

3.1 Double modulo operation

Let $D(n)$ be $D(n_1, n_2)$ plus $D(n_2, n_3)$, where $n = n_1 \geq n_2$ and n_3 is the modulus of n_1 and n_2. Then $D(n)$ represents the cost of the modulo operation on two numbers followed by the modulo operation on the smaller number of the two and the modulus of the first operation, which is two steps in Euclid's algorithm. With observation 2.1 we will prove the following lemma:

Lemma 3.1. $\exists d : D(n) \leq d \cdot n_1 \cdot (n_1 - n_3)$

Proof.

$$D(n) = D(n_1, n_2) + D(n_2, n_3)$$

With observation 2.1 $\exists d'$:

$$\begin{aligned} D(n) &= d' \cdot n_2 \cdot (n_1 - n_2) + d' \cdot n_3 \cdot (n_2 - n_3) + \mathcal{O}(n_1) \\ &\leq d' \cdot n_1 \cdot (n_1 - n_3) + d' \cdot n_1 \cdot (n_1 - n_3) + \mathcal{O}(n_1) \\ &= 2 \cdot d' \cdot n_1 \cdot (n_1 - n_3) + \mathcal{O}(n_1) \end{aligned}$$

In this case we can be sure that $(n_1 - n_3) \geq 1$ because if $n_1 > n_2$, then also $n_1 > n_3$, and if $n_1 = n_2$ then the two numbers are of an equal amount of bits. This means the modulus, the remainder after division, will be at least 1 bit smaller, so $n_2 > n_3$ which implies $n_1 > n_3$. Now we know that d' will never be completely nullified, we can also absorb the $\mathcal{O}(n_1)$ in d'.

$$\begin{aligned} \exists C : D(n) &\leq 2 \cdot d' \cdot n_1 \cdot (n_1 - n_3) + C \cdot n_1 \\ &\leq 2 \cdot d' \cdot n_1 \cdot (n_1 - n_3) + C \cdot n_1 \cdot (n_1 - n_3) \\ &= (2 \cdot d' + C) \cdot n_1 \cdot (n_1 - n_3) \end{aligned}$$

so let $d = 2 \cdot d' + C$ to get:

$$\exists d : D(n) \leq d \cdot n_1 \cdot (n_1 - n_3)$$

□

This lemma is necessary to prevent issues caused by the $\mathcal{O}(n)$ term, and to guarantee that after each step the size in bits of the largest number decreases.

3.2 Complexity

Let $E(n)$ be the computational cost of Euclid's algorithm on two numbers of at most n bits. We want to prove Euclid's algorithm is quadratic, $E(n) = \mathcal{O}(n^2)$, formally:

Theorem 3.2. $\exists C: \ \forall n : E(n) \leq C \cdot n^2$

To prove this we will use the substitution method. The induction hypothesis is:

$$\forall n' < n : E(n') \leq C \cdot (n')^2$$

Proof.

$$E(n) \leq D(n) + E(n_3) \tag{1}$$

With lemmma 3.1,

recall that this is actually two steps of the algorithm.

$$= d \cdot n \cdot (n - n_3) + E(n_3) \tag{2}$$

With the induction hypothesis

$$\leq d \cdot n \cdot (n - n_3) + C \cdot (n_3)^2 \tag{3}$$

$$\leq d \cdot n \cdot (n - n_3) + d \cdot n_3 \cdot (n - n_3) + C \cdot (n_3)^2 \tag{4}$$

If $C \geq d$

$$\leq C \cdot n \cdot (n - n_3) + C \cdot n_3 \cdot (n - n_3) + C \cdot (n_3)^2 \tag{5}$$

$$= C \cdot n \cdot (n - n_3) + C \cdot n_3 \cdot n \tag{6}$$

$$= C \cdot n^2 \tag{7}$$

□

We can visualize this proof in an image:

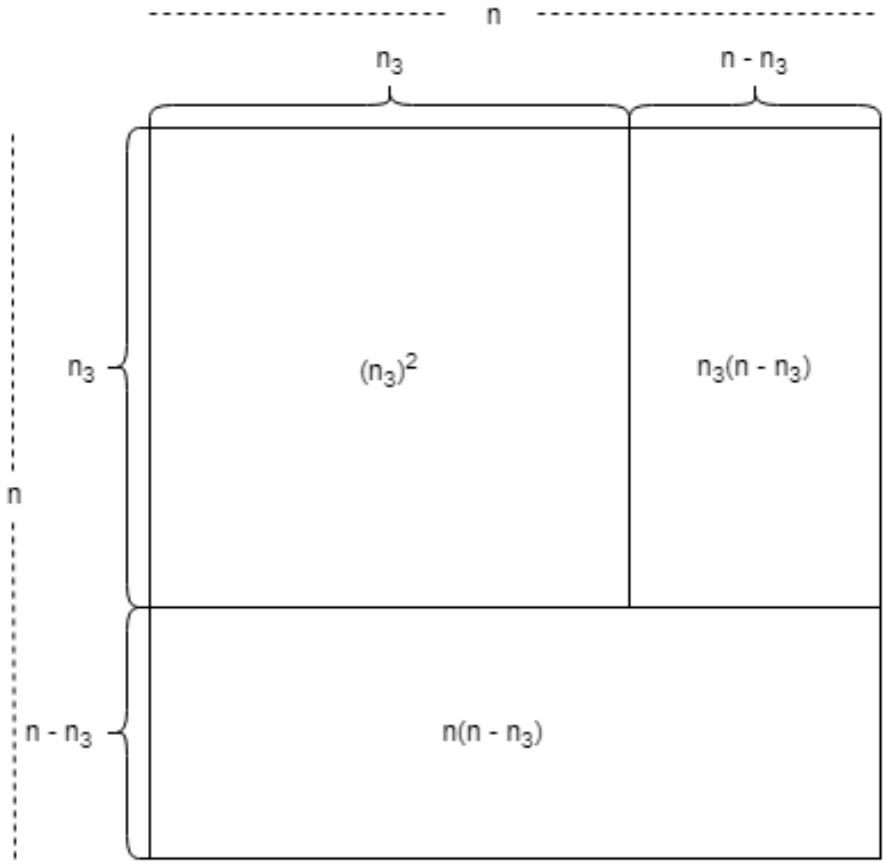

Figure 4: Visual representation of the proof

In figure 4 the situation after step 5 is illustrated. Note that the constant term C is not included since it has no effect on the complexity. This illustrates why the extra term was added in step 4. Because of this extra term, the square with surface n^2 fills up perfectly.

4 Bézout's identity

Before discussing the extended version of Stein's GCD algorithm which calculates the Bézout coëfficients along with the GCD, first we need to understand what exactly the Bézout coëfficients are. Bézout's identity [Bézout1779] states the following:

Theorem 4.1 (Bézout's identity)**.** *Let a and b be integers with GCD(a, b) = d. Then there exist integers x and y such that ax + by = d.*

The integers x and y are called the Bézout coëfficients. There are two general strategies to calculate the Bézout coëfficients: top-down and bottom-up. The top-down strategy keeps (a, b) constant and adapts (x, y) and d as the algorithm progresses, while the bottom-up strategy first calculates the GCD and then working its way back up, adapting (a, b) along the way until the original (a, b) is reached. To illustrate the differences in the two strategies two examples are given using Euclid's algorithm, with input $a = 28$ and $b = 13$.

4.1 Bottom-up

For the bottom-up strategy, first begin by calculating the GCD according to Euclid's algorithm, while remembering the quotiont q and remainder r of each step

n	a	b	q	r
0	28	13	2	2
1	13	2	6	1
2	2	1	2	0

Now we know that in the last row where $r \neq 0$, r is the GCD. We also know here that $a_n - q_n \cdot b_n = r_n$, in this example:

$$1 = 13 - 6 \cdot 2$$

Continue by replacing b in the equation working upwards:

$$b_n = a_{n-1} - q_{n-1} \cdot b_{n-1}$$

until the first row is reached, at which point we have our answer.

$$1 = 13 - 6 \cdot (28 - 2 \cdot 13)$$
$$1 = 13 \cdot 13 - 6 \cdot 28$$

4.2 Top-down

Initialise the first two rows as illustrated in the table below. It is clear to see that in the first two rows $a \cdot x + b \cdot y = r$. Calculate the x and y of the next row:

$$\begin{aligned} q_{n+1} &= r_{n-1}/r_n \\ r_{n+1} &= r_{n-1} - r_n \cdot q_{n+1} \\ x_{n+1} &= x_{n-1} - x_n \cdot q_{n+1} \\ y_{n+1} &= y_{n-1} - y_n \cdot q_{n+1} \end{aligned}$$

Now from the equations it is easy to see that if in each row $a \cdot x_n + b \cdot y_n = r_n$ and $a \cdot x_{n-1} + b \cdot y_{n-1} = r_{n-1}$, then also $a \cdot x_{n+1} + b \cdot y_{n+1} = r_{n+1}$ so this is true for each row, and after calculating the GCD, again the last $r \neq 0$, has been calculated, we also know the Bézout coëfficients, which are simply the x and y in the corresponding row.

q	r	x	y
-	28	1	0
-	13	0	1
2	2	1	-2
6	1	-6	13
2	0	13	-28

5 Extended version of Stein's GCD algorithm

Stein's binary GCD algorithm [Stein1967] is also a very widely known and used algorithm, so a proof of correctness will not be provided, nor an analysis of the algorithmic complexity which is $\mathcal{O}(n^2)$ [Knuth1968] like Euclid's algorithm. Instead we will provide a proof of correctness of the proposed extended algorithm, which uses the top-down strategy.

5.1 Proof of correctness

In this section we will prove the correctness of our extended algorithm, which can be found in appendix A.4. Let p^* and q^* be two *BigIntegers*, the input of our algorithm. We will call the GCD of the numbers r, and the Bézout coëfficients sp and sq, such that $p^* \cdot sp + q^* \cdot sq = r$.

We first need to check there will be no problems if either one of the numbers is equal to zero. Without loss of generality say $p^* = 0$. Then $r = q^*$, and $0 \cdot 0 + 1 \cdot q^* = r$. In the next step common factors of two are removed, later we will see this does not affect the Bézout coëfficients in any way. We will prove the correctness of the remainder of the algorithm with an invariant.

Proof. Let p_0 and q_0 be the current p and q, so p^* and q^* without their common factors of two. We will declare new numbers $sp = 1$, $sq = 0$, $tp = 0$ and $tq = 1$, similar to the top-down strategy for Euclid's algorithm. Our eventual Bézout coëfficients will be sp and sq, with GCD p, but since p and q may be swapped we also need tp and tq. The invariant will be:

$$sp \cdot p_0 + sq \cdot q_0 = p$$
$$tp \cdot p_0 + tq \cdot q_0 = q$$

In the initialization the invariant is clearly true. In the first while loop p is divided by two until it is odd. To maintain the invariant sp and sq also need to be divided by two. This is only possible without causing problems if both sp and sq are even. We can guarantee this by checking if they are both even, and if they are not we will subtract q_0 from sp and add p_0 to sq, making them both even.

Bézout's identity We can view theorem 4.1 as d being the dot product of (a, b) and (x, y). This is the reason we can subtract q_0 from sp and add p_0 to sq without breaking the invariant. Our vector $(a, b) = (p_0, q_0)$ and $(x, y) = (sp, sq)$. Since $(-q_0, p_0)$ is perpendicular to (p_0, q_0), adding it to (sp, sq) does not change the dot product of the two vectors. We can see that when we work out the dot product as well:

$$(sp - q_0) \cdot p_0 + (sq + p_0) \cdot q_0 = sp \cdot p_0 - p_0 \cdot q_0 + sq \cdot q_0 + p_0 \cdot q_0 = sp \cdot p_0 + sq \cdot q_0$$

We will prove this guarantees that both sp and sq are even with case analysis. We know p is even since we are in the $while(p.IsEven)$ loop and that q is odd, because common factors were removed.

Case 1: sp is odd, sq is even If sq is even, then $sq \cdot q_0$ will also be even. Since p is even, $sp \cdot p_0$ must also be even. Then p_0 must be even, because sp is odd. This means q_0 is odd because p_0 and q_0 do not have common factors of two. Then $sp - q_0$ will become even, and $sq + p_0$ will stay even.

Case 2: sp is even, sq is odd The reasoning of this case is symmetrical to the first case, with the only difference being that $sp - q_0$ will stay even instead of become even, and $sq + p_0$ will become even instead of stay even.

Case 3: sp and sq are both odd We know p_0 and q_0 do not have any common factors of two, so at least one of them must be odd. If both sp and sq are odd, then at least one of $sp \cdot p_0$ or $sq \cdot q_0$ must be odd. Because p is even, the other must be odd as well. This means that p_0 and q_0 must also both be odd, in which case $sp - q_0$ and $sq + p_0$ will both become even.

Now we have guaranteed that both sp and sq are even and we can divide them

by two. Note that we did not use the knowledge that $p = p_0$ in this part of the code, because this way the given proof is symmetrical to the next loop which performs the same operation while q is even and p is odd. We know p is odd there because at first we removed all factors of two from p, and in the loop while $p \neq 0$ the value of p is only changed if p and q are swapped, which only happens after the factors of two are removed from q.

After the loop p and q are swapped if $p > q$, to maintain the invariant sp and tp are swapped, as well as sq and tq. This simply swaps the two equations of the invariant, so if they were true before they will be after.

The last step is then subtracting p from q. There are no complications like what we found when dividing, we can simply subtract sp from tp and sq from tq to maintain the invariant:

$$\begin{aligned}(tp \cdot p_0 + tq \cdot q_0) - (sp \cdot p_0 + sq \cdot q_0) &= q - p \\ (tp - sp) \cdot p_0 + (tp - sq) \cdot q_0 &= q - p\end{aligned}$$

As mentioned before, the algorithm does not return p but p multiplied with the common factors of $p*$ and $q*$ removed at the beginning. To complete the proof we will show that the values sp and sq are still valid. From the invariant we know $sp \cdot p_0 + sq \cdot q_0 = r'$, where r' is the GCD r without the removed factors of two. Let k be the removed factors of two. Then:

$$\begin{aligned}sp \cdot p_0 + sq \cdot q_0 &= r' \\ (sp \cdot p_0 + sq \cdot q_0) \cdot k &= r' \cdot k \\ sp \cdot p_0 \cdot k + sq \cdot q_0 \cdot k &= r \\ sp \cdot p^* + sq \cdot q^* &= r\end{aligned}$$

□

6 Performance Comparison

We compared the performance of Euclid's and Stein's algorithm, along with their extended versions. There is a built-in *BigInteger.GreatestCommonDivisor* method as well in C#, which has been included in the comparison.

6.1 Implementation details

Every algorithm is tested in the same way, and has been implemented as subclasses of a shared superclass *GCD*. The *Test* class is responsible for performing the tests, where a test consists of *size* pairs of *BigIntegers* with *bitSize* bits. After every test, the results calculated by the GCD algorithms is verified, this time is not counted for the performance of the algorithm.

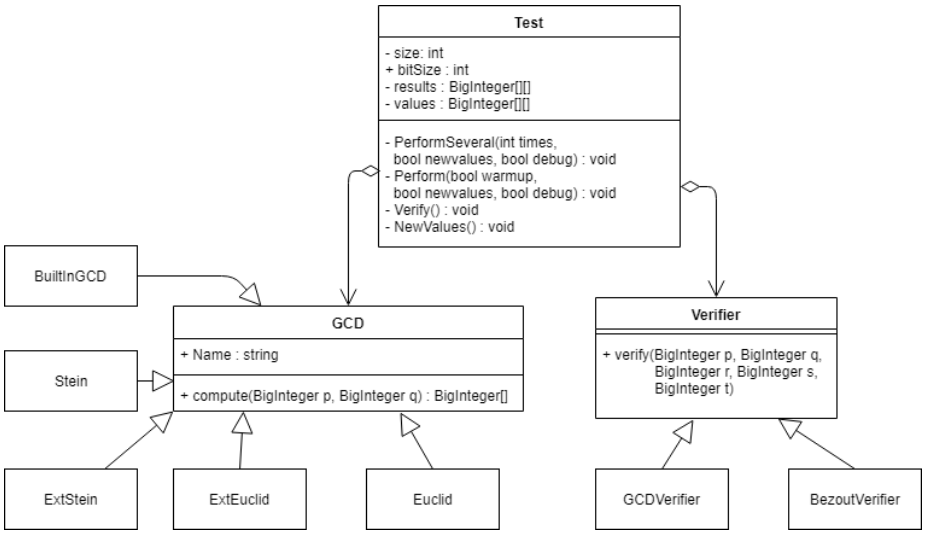

Figure 5: UML class diagram of the testing setup

GCD algorithm implementations The GCD algorithm implementations can all be found in the appendix. The *compute* method returns an array because the extended versions also return the Bézout coëfficients.

Verifier implementations The GCD verifier first checks if the GCD, r, is zero when it should not be, then checks whether the GCD calculated by the built-in GCD algorithm is equal to r. This of course only works if we trust that the built-in GCD algorithm is correct. We have chosen to trust this is the case. The Bézout verifier checks Bézout's identity (Theorem 4.1):

Testing implementation The tests were performed with the *PerformSeveral* method. This method first runs an untimed test as a warm-up. This makes sure the cache is in a good state, which can improve performance [Zhang et al.2013]. Then it calls the *Perform* method for the indicated amount of times. This method begins by making sure the priority of the thread is set to the highest possible. Then after one more small warm up where the GCD of the first pair is calculated, it will call the garbage collector to collect to prevent this from happening while the stopwatch is ticking which could skew the results, before finally starting the stopwatch and performing the test. The complete implementation can be found in appendix A.

6.2 Results

All tests were run while all other processes which could be stopped were stopped in another attempt to prevent minimize disruptions, on an Intel® Core™ i5-

7300HQ processor @2.50 Ghz with 8.00 GB RAM. As mentioned, the algorithms were written in C#.

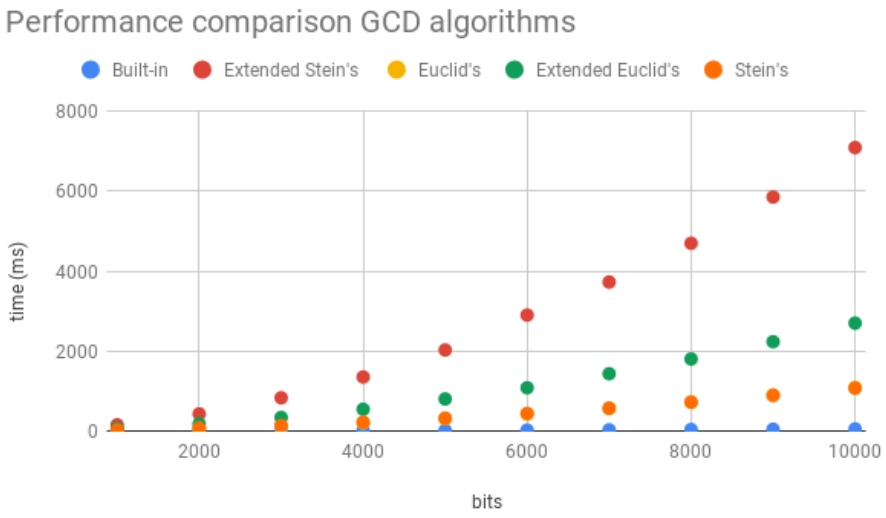

Figure 6: Performance results of all GCD algorithms

Extended versions As was expected, both of the extended versions of the algorithms are slower than their non-extended counterparts. Extended Euclid's stays relatively close to its counterpart, but Extended Stein's is much slower. This is most likely because in the extended version each step is more expensive. Because Euclid's strength is that it has fewer steps, this effect is much smaller there compared to Stein's algorithm, its strength being that although it has more steps, the efficiency of the steps is much better. It is possible that the performance of both extended algorithms could be improved if they were to be implemented in a lower-level programming language with built-in support for swapping variables, like *std::swap* in C++. However, in this implementation swapping the two numbers is still only swapping the pointers, so the difference could also be negligible. The advantage would be to prevent having to create a temporary variable to swap the variables.

The Built-in method In figure 6 we can see that the built-in method is the fastest by a large margin. It must be said, however, that we do not know how this method is implemented. Still, this raises an interesting question about the algorithmic complexity of this method. To test whether or not this method is also $\mathcal{O}(n^2)$, we performed a new test with 100 pairs per test. In the table, *ratio* is the ratio between the duration of the indicated bitsize and the duration of the previous bitsize.

bitsize	duration (ms)	ratio
10000	54.9	-
20000	183.3	3.34
40000	613.8	3.35
80000	2277.3	3.71
160000	8832.4	3.88
320000	34611.3	3.92
640000	136755.9	3.95

We can see that the ratio approaches 4 as the bitsize of the numbers grows. This is what we would expect it to be if the algorithm were quadratic. An explanation for it not being 4 when comparing the smaller bitsizes is that the algorithm has some start-up cost independent, or at least sub-quadratic, of the bitsize of the numbers. This would lead to the ratio being smaller for smaller numbers, since a large part of the cost in that case is the start-up cost which does not or barely grow when the numbers grow. The influence of the start-up cost as the cost of the actual calculation grows would become smaller, causing the ratio to approach 4.

Euclid's and Stein's algorithm At first glance it may seem like Euclid's algorithm is not plotted in figure 6, but that is simply because the results of Euclid's and Stein's algorithm are too close to each other to distinguish on this scale. When zooming in we can see that there is a difference:

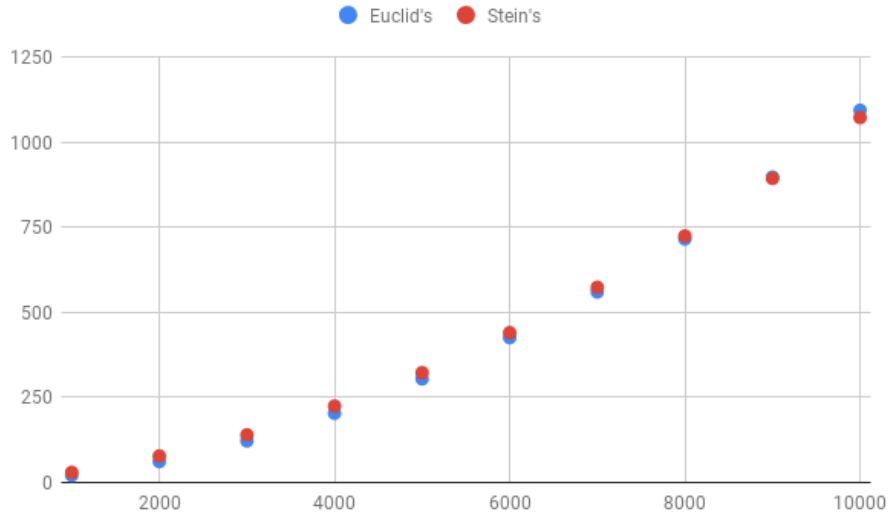

Figure 7: Euclid's vs Stein's GCD algorithm

We can see that at first Euclid's is faster, but at a bitsize of around 8000

Stein's overtakes it. When continuing with even larger numbers we will see that Stein's stays faster with a growing difference. It is important to note that it may be possible to further optimize both algorithms, which could lead to very different results.

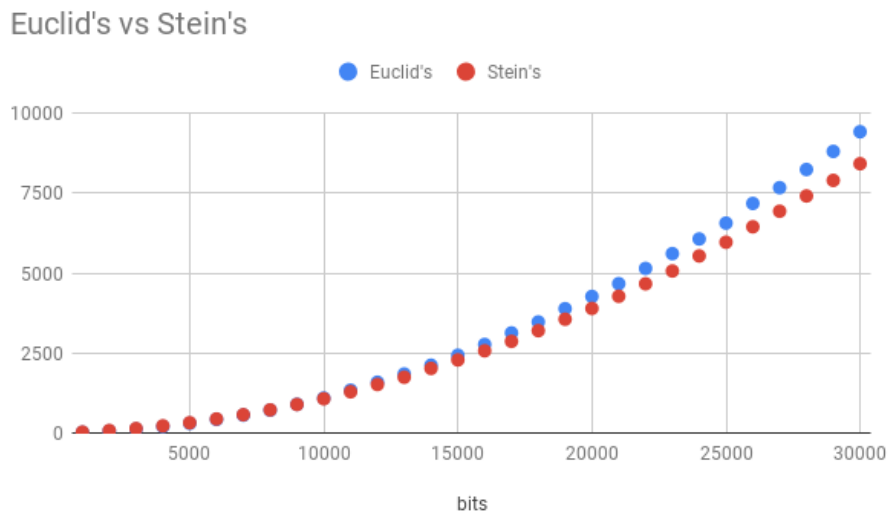

Figure 8: Euclid's vs Stein's GCD algorithm

The results are not what we expected them to be. Although we can see that Stein's algorithm does indeed become faster than Euclid's, We expected it to be much faster based on other literature [Knuth1968] [Shallit and Sorenson1994] [Sorenson1994]. The explanation for Stein's algorithm being faster is that it was developed specifically to be used by computers and only uses cheap operations such as bitshifts and subtractions, while Euclid's algorithm was meant to be used by humans and uses the more expensive modulo operation. The advantage Euclid's algorithm has is that it needs fewer steps to find an answer. A possible explanation for the unexpected speed of Euclid's algorithm can be found when looking at the ratio of p and q combined with a closer look at the cost of the modulo operation.

6.3 p, q-ratio Euclid's algorithm

When looking at the ratio between p and q in each step of Euclid's algorithm, the first thing to notice is that the ratio is almost always very low. We can see in figure 9 that in over 41% of cases it is 1, in over 77% of the cases it is ≤ 5, and in over 90% of cases it is ≤ 13. Note that we use the ratio as an integer here, a ratio of 1 means that $q \leq p < 2q$.

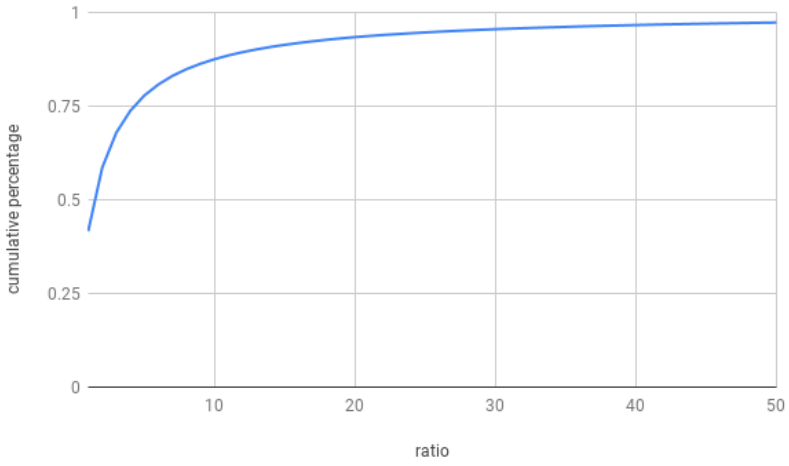

Figure 9: Cumulative percentage of ratio between p and q in Euclid's algorithm

This becomes interesting when we remember our earlier observation 2.1, stating that the cost of the modulo operation scales with the difference in ratio. Before we can draw conclusions, however, we need to look at the cost of the modulo operation in more detail when the difference in bitsizes between the two numbers is relatively small.

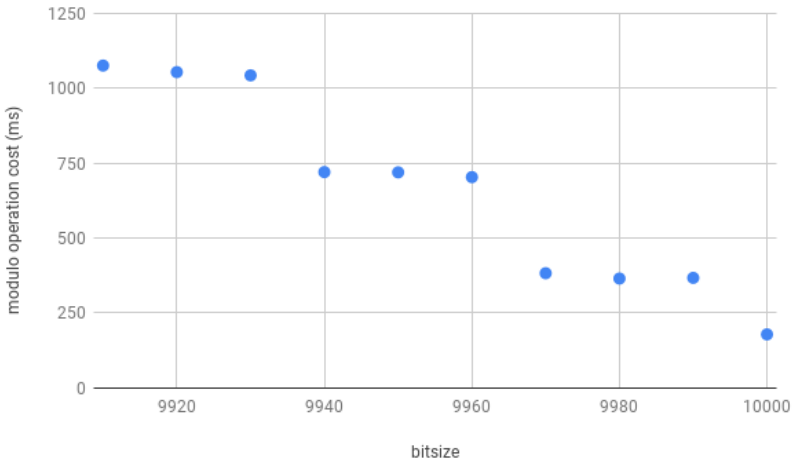

Figure 10: Duration of the modulo operation of a 10.000 bit number and a number around 10.000 bits

We can see that when the numbers are of an equal size, the cost is very low.

Then, around every 30 bits there is a jump in cost. A possible explanation for this is that *BigIntegers* are most likely stored in blocks, or words, of 32 bits, the size of an integer. Once the difference in bitsize gets larger than one word, every step in the operation requires the use of two words instead of one, complicating the operation. Since the ratio of p and q is virtually always smaller than the maximum value of a word, the modulo operation will virtually always be quite fast, and over 40% of the time even faster, when the ratio is 1. This could be an explanation for why Euclid's algorithm is faster than we expected.

7 Conclusion

To conclude, experimenting with the modulo operation lead to the observation that the cost of the operation scales with the bitsize of the smaller number multiplied by the bitsize of the quotient. This observation was used in a concise and intuitive proof that Euclid's algorithm is $\mathcal{O}(n^2)$. An extended version of Stein's binary GCD algorithm was provided using a top-down strategy to calculate the Bézout coëfficients, with a proof of correctness using Bézout's identity as a basis representation.

The performance of both algorithms with their extended versions was compared. In this comparison we saw that Stein's algorithm was faster than Euclid's algorithm, but not as much as we expected. A possible explanation for this was found in another experiment with the modulo operation, combined with the insight that the ratio between two numbers used in Euclid's algorithm is almost always very low. Euclid's extended algorithm was much faster than Stein's extended algorithm, but both were significantly slower than their non-extended counterparts. The C# built-in GCD method was much faster than every other algorithm, but still $\mathcal{O}(n^2)$. The disadvantage of this method is that it has no extended version. We had hoped that our version of Stein's extended algorithm would be faster than the extended version of Euclid's algorithm, but because of the extra cost per step in the extended versions and the fact that Stein's needs more steps than Euclid's it was slower by roughly a factor of two.

References

[Bach et al.1996] Bach, E., Shallit, J. O., Jeffrey, S., and Shallit, J. (1996). *Algorithmic Number Theory: Efficient Algorithms*, volume 1. MIT press.

[Bézout1779] Bézout, E. (1779). *Théorie générale des équations algébriques; par M. Bézout...* de l'imprimerie de Ph.-D. Pierres, rue S. Jacques.

[Knuth1968] Knuth, D. (1968). The art of computer programming 2: Seminumerical algorithms. *MA: Addison-Wesley*, 30.

[Rivest et al.1978] Rivest, R. L., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126.

[Shallit and Sorenson1994] Shallit, J. and Sorenson, J. (1994). Analysis of a left-shift binary gcd algorithm. *Journal of Symbolic Computation*, 17(6):473–486.

[Sorenson1994] Sorenson, J. (1994). Two fast gcd algorithms. *Journal of Algorithms*, 16(1):110–144.

[Stein1967] Stein, J. (1967). Computational problems associated with racah algebra. *Journal of Computational Physics*, 1(3):397–405.

[Zhang et al.2013] Zhang, Y., Soundararajan, G., Storer, M. W., Bairavasundaram, L. N., Subbiah, S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2013). Warming up storage-level caches with bonfire. In *Presented as part of the 11th {USENIX} Conference on File and Storage Technologies ({FAST} 13)*, pages 59–72.

A Source code

The complete source code with which the tests have been performed can be found on https://github.com/jorisBarkema/GCD. The GCD algorithm implementations are also included explicitly below.

A.1 Euclid's algorithm

Implementation of Euclid's algorithm in C#.

```
public override BigInteger[]
    compute(BigInteger p, BigInteger q, bool debug = false)
{
    while (p != q)
    {
        if (p == 0) return new BigInteger[] { q };
        if (q == 0) return new BigInteger[] { p };
        if (p > q) p = p % q;
        else { q = q % p; }
    }

    return new BigInteger[] { p };
}
```

A.2 Extended Euclid's algorithm

Implementation of the extended version of Euclid's algorithm in C#.

```
public override BigInteger[]
    compute(BigInteger p, BigInteger q, bool debug = false)
{
    BigInteger oldR = p, newR = q,
               oldS = 1, newS = 0,
               oldT = 0, newT = 1;

    while (newR != 0)
    {
        BigInteger quotient = oldR / newR;

        BigInteger temp = newR;
        newR = oldR - quotient * temp;
        oldR = temp;

        temp = newS;
        newS = oldS - quotient * temp;
        oldS = temp;
```

```
        temp = newT;
        newT = oldT - quotient * temp;
        oldT = temp;
    }

    // Bezout coefficients oldS and oldT
    return new BigInteger[] { oldR, oldS, oldT } ;
}
```

A.3 Stein's binary GCD algorithm

Implementation of Stein's binary GCD algorithm in C#.

```
public override BigInteger[]
    compute(BigInteger p, BigInteger q, bool debug = false)
{
    int shift = 0;

    if (p == 0) return new BigInteger[] { q };
    if (q == 0) return new BigInteger[] { p };

    while (p.IsEven && q.IsEven)
    {
        p >>= 1;
        q >>= 1;
        shift++;
    }

    while (p.IsEven) p >>= 1;

    while (!q.IsZero)
    {
        while (q.IsEven) q >>= 1;

        if (p > q)
        {
            BigInteger swap = p;
            p = q;
            q = swap;
        }

        q = (q - p) >> 1;
    }

    return new BigInteger[] { p << shift };
}
```

A.4 Extended Stein's binary GCD algorithm

Implementation of the proposed extended version of Stein's binary GCD algorithm in C#.

```
public override BigInteger[]
    compute(BigInteger p, BigInteger q, bool debug = false)
{
    int shift = 0;

    if (p == 0) return new BigInteger[] { q, 0, 1 };
    if (q == 0) return new BigInteger[] { p, 1, 0 };

    while (p.IsEven && q.IsEven)
    {
        p >>= 1;
        q >>= 1;
        shift++;
    }

    BigInteger p0 = p, q0 = q;

    BigInteger sp = 1, sq = 0;
    BigInteger tp = 0, tq = 1;

    while (p.IsEven)
    {
        if (!(sp.IsEven && sq.IsEven))
        {
            sp -= q0;
            sq += p0;
        }
        p >>= 1;
        sp >>= 1;
        sq >>= 1;
    }

    while (!q.IsZero)
    {
        while (q.IsEven)
        {
            if (!(tp.IsEven && tq.IsEven))
            {
                tp -= q0;
                tq += p0;
            }
```

```
            q >>= 1;
            tp >>= 1;
            tq >>= 1;
        }

        if (p > q)
        {
            BigInteger t = p;
            p = q;
            q = t;

            t = sp;
            sp = tp;
            tp = t;

            t = sq;
            sq = tq;
            tq = t;
        }

        q = q - p;
        tp = tp - sp;
        tq = tq - sq;
    }

    return new BigInteger[] { p << shift, sp, sq };
}
```

A.5 Built-in Greatest Common Divisor method

```
public override BigInteger[]
    compute(BigInteger p, BigInteger q, bool debug = false)
{
    return new BigInteger[]
        { BigInteger.GreatestCommonDivisor(p, q) };
}
```