1 Heuristisch binaire-GCD-achtig algoritme voor kwadratische getallenlichamen

1.1 Inleiding

Het algoritme bouwt verder op het binaire GCD-idee zoals beschreven in *Binary GCD Like Algorithms for Some Complex Quadratic Rings* van Saurabh Agarwal en Gudmund Skovbjerg Frandsen en is feitelijk een implementatie van de ideeën in *Two fast GCD algorithms* van Jonathan Sorenson.

Het hoofddoel is om grootste gemene delers te kunnen berekenen voor de ring van gehelen in algemene kwadratische getallenlichamen. Hoewel het bekend is dat veel van deze ringen klassegetal groter dan 1 hebben, kunnen we dan alsnog proberen een GCD uit te rekenen – die dan niet uniek is.

In verband met een gelimiteerd aantal tijd heb ik me beperkt tot de (ring van gehelen van) imaginair-kwadratische getallenlichamen die een uniekfactorisatiedomein zijn. In het vervolg van dit artikel zullen we $\mathcal{O}_d$ noteren voor de ring van gehelen in $\mathbb{Q}(\sqrt{d})$. De unieke factorisatiedomeinen voor $d < 0$ zijn:

$$d \in \{-1, -2, -3, -7, -11, -19, -43, -67, -163\}$$

Eveneens is bekend dat voor $d \in \{-19, -43, -67, -163\}$ de ring $\mathcal{O}_d$ *niet* euclidisch is.

Nu zijn er voor $d \in \{-1, -3\}$ (gehelen van Gauss resp. Eisenstein) al effectieve algoritmes bekend, en hebben Agarwal en Skovbjerg een grondige analyse gegeven van hoe hun binaire-GCD-algoritme te werk gaat voor $d \in \{-2, -7, -11, -19\}$. In dit artikel zullen we ons focussen op de niet-euclidische ringen, dat wil zeggen: $d \in \{-19, -43, -67, -163\}$. In het vervolg van dit artikel zullen we altijd aannemen dat d een van deze genoemde getallen is. We noteren $\omega_d = \frac{1+\sqrt{d}}{2}$ als $d = 1 \bmod 4$, en $\omega_d = \sqrt{d}$ als $d = 2, 3 \bmod 4$. Dan kunnen we opmerken dat we een aantal speciale priemen hebben in de ruimtes $\mathcal{O}_d$: $\pi_d := \omega_d$. We weten dat deze getallen priem zijn voor $d \in \{-19, -43, -67, -163\}$ omdat:

$$\pi_d \overline{\pi_d} = (\omega_d)(1 - \omega_d) = \frac{(1 + \sqrt{d})(1 - \sqrt{d})}{4} = \frac{1 - d}{4} \in \{5, 11, 17, 41\}$$

De normen van $\pi_d = \omega_d$ zijn dus gelijk aan priemgetallen, wat betekent dat ze irreducibel i.e. priem (want we leven in een hoofdideaalring) zijn.

In imaginair-kwadratische lichaam is de norm $N : \mathcal{O}_d \to \mathbb{N}$, $a + b\omega \mapsto (a+b\omega)(a+b\overline{\omega})$ altijd een element van de natuurlijke getallen en in zekere zin een maat voor de grootte van het getal. Dit wordt gebruikt in het algoritme.

1.2 Het idee van het algoritme

We weten dat we een priemelement ω_d hebben in $\mathcal{O}_d$. Stelt u zich voor dat we twee getallen $\alpha, \beta \in \mathcal{O}_d$ hebben zo dat ω_d geen deler is van deze twee getallen.

De belangrijke stap in het algoritme is zoeken naar relatief kleine $k, l \in \mathbb{Z}$ zodanig dat:

$$k\alpha + l\beta \equiv 0 \bmod \omega_d$$

in de hoop dat $N\left(\frac{k\alpha+l\beta}{\omega_d}\right) < N(\alpha)$.

Waarom doen we dit? Omdat in de juiste omstandigheden geldt:

$$(\alpha, \beta) = \left(\frac{k\alpha + l\beta}{\omega_d}, \beta\right)$$

Hier is $(.,.)$ de notatie voor de grootste gemene deler van twee elementen. Bovenstaande uitspraak kunnen we bewijzen in het volgende lemma:

Lemma: 1. *Laat $\alpha, \beta \in \mathcal{O}_d$ en $k, l \in \mathbb{Z}$ zodanig dat ω_d, l, k de getallen α, β niet delen. Bovendien $k\alpha + l\beta \equiv 0$ mod ω_d. Dan geldt:*

$$(\alpha, \beta) = \left(\frac{k\alpha + l\beta}{\omega_d}, \beta\right)$$

Bewijs. We weten dat k geen deler is van β, dus:

$$(\alpha, \beta) = (k\alpha, \beta) = (k\alpha + l\beta, \beta)$$

Verder is het gegeven dat ω_d geen deler is van β. ω_d is echter wel een deler van $k\alpha + l\beta$, dus we kunnen het laatstgenoemde getal net zo goed door ω_d delen. QED □

Het tweede hoofdidee is – om bovengenoemde k en l te verkrijgen – we α en β modulo ω_d kunnen berekenen. Omdat ω_d een priemelement is met norm $p \in \{5, 11, 17, 41\}$ geldt dat we α mod ω_d kunnen representeren met een getal in $\mathbb{F}_p$; kort gezegd:

$$\mathcal{O}_d/(\omega_d) \simeq \mathbb{F}_p$$

We zullen $a, b \in \mathbb{F}_p$ de representanten van de restklasse modulo ω_d noemen van α respectievelijk β. Voor deze representanten geldt: $0 \leq a, b < p$. Nu is het evident:

$$ka + lb \equiv 0 \bmod p \iff k\alpha + l\beta \equiv 0 \bmod \omega_d$$

Dus de vraag om k, l te vinden voor α, β is gereduceerd tot k, l vinden voor $a, b \in \mathbb{F}_p$.

Het derde hoofdidee is de opmerking dat de k, l die we moeten zoeken relatief klein moeten zijn. Dit ligt aan het volgende:

$$N(k\alpha + l\beta) = k^2 N(\alpha) + l^2 N(\beta) + 2kl < \alpha \cdot \beta > \tag{1}$$

Schrijven we $\alpha = a_1 + ia_2$, $\beta = b_1 + ib_2$, dan is het inproduct $< \alpha \cdot \beta >= a_1 b_1 + a_2 b_2$. Omdat we weten:

$$N\left(\frac{k\alpha + l\beta}{\omega_d}\right) = \frac{k^2 N(\alpha) + l^2 N(\beta) + 2kl < \alpha \cdot \beta >}{p}$$

zouden we op zijn minst willen hebben:

$$k^2 + l^2 \leq p$$

willen we goede kans hebben dat $N\left(\frac{k\alpha+l\beta}{\omega_d}\right) < N(\alpha)$. Dus de vraag is: voor welke p priem geldt dat voor alle $a, b \in \mathbb{F}_p$ dat er $k, l \in \mathbb{F}$ zijn met $ka + lb \equiv 0$ en $k^2 + l^2 \leq p$?

Vermoeden 1. *Een lijst van alle priemen p waarvoor geldt:*

$$\forall a, b \in \mathbb{F}_p \ \exists \ k, l \in \mathbb{F}_p \ \textit{zodat} \ ka + lb \equiv 0 \ \textit{en} \ k^2 + l^2 \leq p$$

is: $\{2, 3, \underline{5}, 7, \underline{11}, 13, \underline{17}, 19, 29, 31, 37, \underline{41}, 43, 67, 89\}$

Misschien heeft u het al gezien, maar de ondergestreepte getallen hierboven zijn precies de normen van de ω_d in onze gevallen. Merk op dat ook de getallen $1 + \omega_d$ in onze gevallen $d \in \{-19, -43, -67, -163\}$ irreducibel zijn en $N(1 + \omega_d)$ komen *ook* in deze lijst voor.

Het eindidee van het algoritme is dat we lijsten maken van (a, b) met de bijbehorende (k, l) zodanig dat $ka + lb \equiv 0 \bmod p$ en $k^2 + l^2 \leq p$.

We gaan verder met vergelijking (1), waarbij we zonder verlies van algemeenheid aannemen: $N(\alpha) \geq N(\beta)$.

$$N(k\alpha + l\beta) = k^2 N(\alpha) + l^2 N(\beta) + 2kl < \alpha \cdot \beta >$$

$$\leq (k^2 + l^2) N(\alpha) + 2kl < \alpha \cdot \beta > \leq (|k| + |l|)^2 N(\alpha) \tag{2}$$

De laatste ongelijkheid is gebaseerd op de Cauchy-Schwarz-ongelijkheid. Dus alleen als $(|k| + |l|)^2 < p$ hebben we zekerheid over dit algoritme. Helaas gebeurt dit slechts in ongeveer de helft van de gevallen (a, b).

Definitie 1. *We noemen* (a, b) *met* $a, b \in \mathbb{F}_p$ *goede moduli als er* $k, l \in \mathbb{Z}$ *zijn met* $ka + lb \equiv 0$ *en* $(|k| + |l|)^2 < p$. *Anders heten* (a, b) *slechte moduli.*

In het geval dat α, β slechte moduli hebben, gaan we het feit exploiteren dat we de tekens van k, l kunnen manipuleren. Dus: als $< \alpha \cdot \beta >> 0$ dan willen we dat de tekens van k en l ongelijk zijn, en als $< \alpha \cdot \beta >< 0$ dan willen we dat de tekens van k en l gelijk zijn. In deze gevallen geldt dan:

$$N(k\alpha + l\beta) = k^2 N(\alpha) + l^2 N(\beta) + 2kl < \alpha \cdot \beta >\leq (k^2 + l^2)N(\alpha)$$

Dus: Voor de goede moduli (a, b) houden we een lijst van (k, l) bij, van 'goede coefficienten'. Voor de slechte moduli (a, b) houden we een 'positieve lijst' van (k, l) en een 'negatieve lijst' van (k, l) bij.

Voorbeeld 1. *Laten we* $(1, 4)$ *bekijken in* $\mathbb{F}_{11}$. *Zoals u uit kunt puzzelen, zijn dit slechte moduli. Echter wel:* $3 * 1 + 2 * 4 \equiv 0$ *en* $-1 * 1 + 3 * 4 \equiv 0$, *dus de positieve coefficienten zijn:* $(3, 2)$ *en de negatieve* $(-1, 3)$. *Merk op dat voor de positieve coefficienten* niet *geldt:* $k^2 + l^2 \leq p$.

Voorbeeld 2. *Laten we* $(1, 9)$ *bekijken in* $\mathbb{F}_{17}$. *Dit zijn goede moduli, want:* $-1 * 1 + 2 * 9 \equiv 0$ *en* $(| - 1| + |2|)^2 = 9 < 17$.

1.3 Het algoritme

We zullen eerst het algoritme beschrijven voor α, β waarvoor geldt dat $2, 3, 5, 7, \omega_d, \overline{\omega_d}$ deze twee getallen niet delen.

```
    // Input: alpha, beta
while (N(α) != 0)
    a := α mod ω_d
    b := β mod ω_d
    a' := α mod ω̄_d
    b' := β mod ω̄_d
    if (a,b) goede moduli, then
        k,l := getCoefficients(a,b)
        α = (kα+lβ)/ω_d
        continue
    else if (a',b') goede moduli, then
        k,l := getCoefficients(a',b')
        α = (kα+lβ)/ω̄_d
        continue
```

```
    else
        inprod := < α · β >
        if (inprod < 0) then
            k',l' = getPositiveCoefficients(a',b')
            k,l = getPositiveCoefficients(a,b)
        else
            k',l' = getNegativeCoefficients(a',b')
            k,l = getNegativeCoefficients(a,b)
        end if;
        if k'^2 + l'^2 > k^2 + l^2 then
            α = (k'α+l'β)/ω̄_d
            continue
        else
            α = (kα+lβ)/ω_d
            continue
        end if;
    end if;
end while;
```

1.4 De code

Hoewel het hierboven erg kort, bondig en eenvoudig lijkt, komt het bij een werkelijke implementatie vaak neer op iets meer technisch gepruts. In de implementatie representeer ik het getal $x + y\omega_d$ met $[x, y]$. Ook hebben we een heleboel lokale functies, die ik hieronder even kort laat zien met hun eveneens korte beschrijving:

```
////////////////////////////////
// Declaring local functions ///
////////////////////////////////

// Calculates conjugate
function conj(alpha)
...
// multiplication
function mult(alpha,beta)
...
// norm
function normOf(alpha)
...
// returns alpha/prime
```

```
function DivideBy(alpha,prime,p);
...
// returns true iff prime is a divisor of alpha
function isDivisor(alpha,prime,p)
...
// extracts powers of i from alpha, i.e.
// suppose  alpha = alpha' * i^n, then this returns alpha' and
//n.
function getIntegerPowerFromNum(alpha,i)
...
// divides alpha by (one of) the coefficients if possible.
// coefficients is a sequence with two elements.
function ifPossibleDivideBy(alpha,coef)
...
// calculates inproduct of alpha and beta
function inprod(alpha,beta)
...
// Extracts the primepowers of 'prime' and 'cprime' out of
// alpha. suppose alpha =  alpha' * (prime)^n * (cprime)^m,
// this returns: alpha', n,m
function getPowerFromNum(alpha,prime,p)
...
function kAlpha_plus_lBeta(alpha,beta,coef)

function coefficientNorm(coef)
        return (coef[1]^2 + coef[2]^2);
end function;

// returns a = alpha mod prime
function ModPrime(alpha,prime,p)

// extracts all small primes <= Ceiling(Sqrt(p)) from alpha
function extractSmallPrimes(alpha,p)
```

Naast deze functies zijn er nog twee andere lokale functies, die eigenlijk de 'motor' zijn van dit algoritme. Dit zijn `CFunAll` en `speedingUp`. De eerste hiervan beslist welke coefficienten (k, l) er uitgezocht moeten worden. De tweede moet het GCD-proces versnellen in de situatie dat $N(\alpha) >> N(\beta)$.

Ik moet even opmerken dat we hier uitgaan van een lijst van goede moduli in combinatie met goede coefficienten, en een lijst van slechte moduli in combinatie met positieve- en negatieve coefficienten. Deze zijn al vooraf berekend en bevinden zich in de tekstfiles `data`,`data_11`,`data_17`,`data_43`.

We gaan een goeie blik werpen op `CFunAll`:

```
function CFunAll(alpha,beta,d)
        prime := [0,1]; cprime := [1,-1];
        p := (-d + 1) div 4;
        index := Index(UFDdisc,d);
```

```
        goodList := goodLists[index];
        badList := badLists[index];
        goodCoeff := goodCoefs[index];
        posCoeff := badCoefPos[index];
        negCoeff := badCoefNeg[index];
        // goodList bestaat uit de goede moduli,
        // de coefficienten daarvan liggen in
        // goodCoeff.
        // badList bestaat uit de slechte moduli,
        // de coefficienten daarvan liggen in
        // posCoeff en negCoeff.

        modulus1 := [ModPrime(alpha,prime,p),
                ModPrime(beta,prime,p)]; // [a,b]
        modulus2 := [ModPrime(alpha,cprime,p),
                ModPrime(beta,cprime,p)]; // [a',b']

        if (modulus1 in goodList) then
                coef := goodCoeff[Index(goodList,modulus1)];
                return kAlpha_plus_lBeta(alpha,beta,coef),coef;
        end if;
        if (modulus2 in goodList) then
                coef := goodCoeff[Index(goodList,modulus2)];
                return kAlpha_plus_lBeta(alpha,beta,coef),coef;
        end if;

        // Here there are no good 'moduli'.
        ind1 := Index(badList,modulus1);
        ind2 := Index(badList,modulus2);
        if (inprod(alpha,beta) lt 0) then
            // negative inproduct, use positive coefficients
                coef1 := posCoeff[ind1];
                coef2 := posCoeff[ind2];
        else
                coef1 := negCoeff[ind1];
                coef2 := negCoeff[ind2];
        end if;

        if (coefficientNorm(coef1) gt coefficientNorm(coef2))
        then
          return kAlpha_plus_lBeta(alpha,beta,coef2),coef2;
        else
          return kAlpha_plus_lBeta(alpha,beta,coef1),coef1;
        end if;

end function;
```

De bovenstaande procedure beslist dus de k, l. Echter, als $N(\alpha) > p^2 N(\beta)$, dan vinden we dat het wat sneller kan, en gebruiken we de functie `speedingUp`. Omdat α dan zo groot is, is het onverstandig om α of β te vervangen door $k\alpha + l\beta$, tijdens het algoritme. Het is slimmer om dan te vervangen door $\alpha + l\beta$, wat neerkomt op $k = 1$. Dat doen we als volgt:

We berekenen weer $a = \alpha \bmod \omega_d$ en $b = \beta \bmod \omega_d$. Dan berekenen we het getal $q = ab^{-1}$ in $\mathbb{F}_p$. Nu is eenvoudig te checken dat $1 * a + (-q) * b \equiv 0$. Deze procedure kunnen we dan ook voor $\overline{\omega_d}$ doen. We nemen dan de kleinste q (van de twee), en returnen $\alpha + (-q)\beta$. In code ziet dat er zo uit:

```
function speedingUp(alpha,beta,d)
        prime := [0,1]; cprime := [1,-1];
        p := (-d + 1) div 4;
        modulus1 := [ModPrime(alpha,prime,p),
                ModPrime(beta,prime,p)];
        modulus2 := [ModPrime(alpha,cprime,p),
                ModPrime(beta,cprime,p)];

        q := (modulus1[1]*InverseMod(modulus1[2],p)) mod p;
        //q = a*b^-1
        r := (modulus2[1]*InverseMod(modulus2[2],p)) mod p;
        //r =  a'*b'^-1
        if (q gt ((p-1) div 2)) then q := -(p-q); end if;
        if (r gt ((p-1) div 2)) then r := -(p-r); end if;

        if (Abs(q) gt Abs(r)) then
           return kAlpha_plus_lBeta(alpha,beta,[1,-r]),[1,-1];
        else
           return kAlpha_plus_lBeta(alpha,beta,[1,-q]),[1,-1];
        end if;
end function;
```

We gaan nu nog een klein stukje voorbereidende code bestuderen, die ervoor zorgt dat α en β voldoen aan de eisen, zoals niet deelbaar zijn door ω_d, en door kleine priemen:

```
  max_increasing := Log(10,Sqrt(normOf(alpha)+normOf(beta)));
  incr := 0; // counts how many times the norm increases
  steps := 0; // counts how many loopsteps there've been
  p := normOf(prime);
  cprime := conj(prime);
  alpha,i1,j1 := getPowerFromNum(alpha,prime,p);
  beta,i2,j2 := getPowerFromNum(beta,prime,p);
  // alpha and beta are now 'prime'- and 'cprime'free
  alpha,powa := extractSmallPrimes(alpha,p);
  beta,powb := extractSmallPrimes(beta,p);
  pows := [ Min(powa[i],powb[i]) : i in [1..#powa] ];
  // alpha and beta are now free of all primes < Sqrt(p)
```

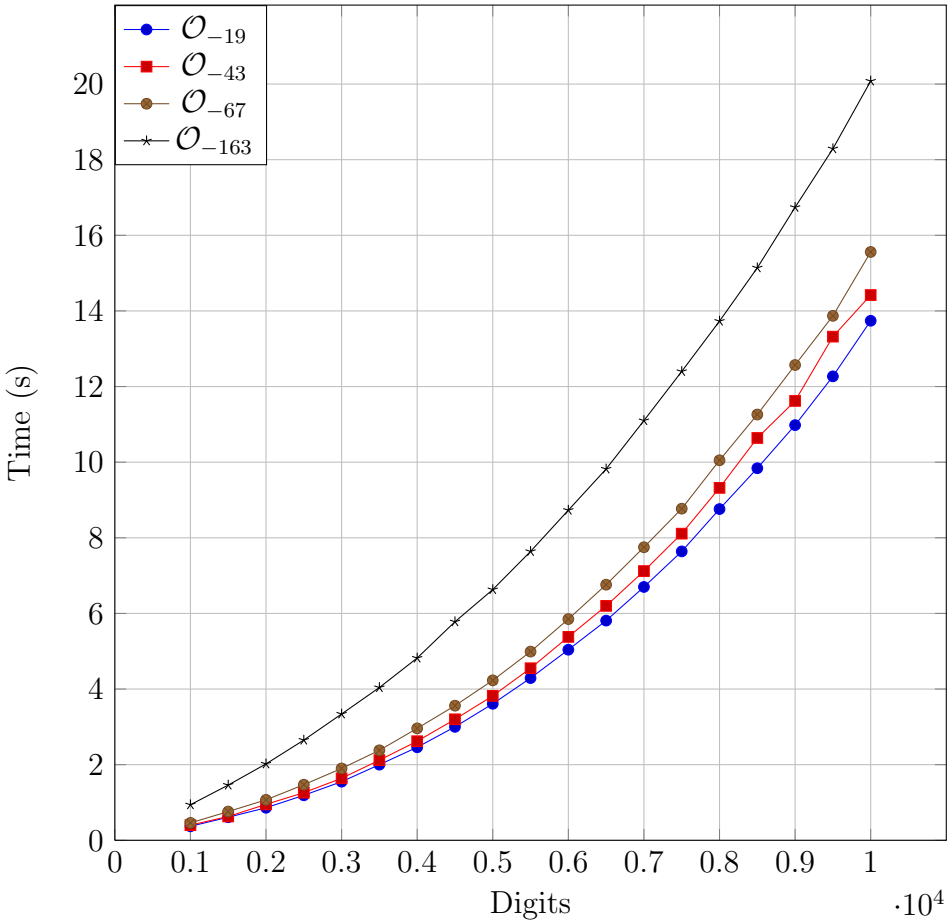
```
  if (normOf(alpha) lt normOf(beta) ) then
        bufalpha := alpha; alpha := beta; beta := bufalpha;
  end if;
  // After this, alpha has always the biggest norm.
  nalpha := normOf(alpha); nbeta := normOf(beta);
```

Na deze voorbereiding geldt: $N(\alpha) \geq N(\beta)$, $\overline{\omega_d}, \omega_d \nmid \alpha$, $\overline{\omega_d}, \omega_d \nmid \beta$. En ook $q \nmid \alpha, \beta$ voor kleine priemen q.

```
while true do
        steps := steps + 1;
        if ((nalpha div p^2) gt nbeta) then
                gamma,coef := speedingUp(alpha,beta,d);
        else
                gamma,coef := CFunAll(alpha,beta,d);
        end if;
        if (gamma eq [0,0]) then break; end if;

        eta := getPowerFromNum(gamma,prime,p);
        // this divides eta by prime or cprime
        eta := ifPossibleDivideBy(eta,coef);
        // this is to avoid that little primes
        // will pop up as common divisor
        neta := normOf(eta);

        if (neta gt nalpha) then incr := incr + 1;  end if;
        if (incr gt max_increasing) then
                printf "Increased more than %o times.\n",incr;
                return [0,0],0,0,[0,0];
        end if; // if the norm increases too much, abort.

        if (neta gt nbeta) then
                alpha := eta; nalpha := neta;
        else    alpha := beta; beta := eta;
                nalpha := nbeta; nbeta := neta;
        end if;
end while;

alpha := extractSmallPrimes(alpha,p);
printf "%o of the %o steps made the norm increase.\n",incr,steps;
return alpha,Min(i1,i2),Min(j1,j2),pows;
```

1.5 Tests

Een rij typische outputs van het algoritme kun je zien in figuur 2. Een figuur met de rekentijden tegen het aantal digits uitgezet kun je zien in figuur 1.



Figuur 1: Grafiek van de berekentijden van het Binaire Gcd-algoritme voor oplopend aantal digits, voor verschillende kwadratisch-imaginaire ringen van gehelen

1.6 Nabeschouwing

Hoewel het lijkt alsof dit algoritme alleen gebruikt kan worden voor kwadratisch-imaginaire getallenringen, is het ook heel goed mogelijk om hem te gebruiken voor reëelkwadratische ringen. Je moet dan wel eerst een priem π vinden met norm p, waarbij p een van de priemen is in lemma 1. Voor willekeurige reëelkwadratische ringen is dit een lastig probleem.

Merk op dat in de ringen die we hier bespreken, ook $1 + \omega_d$ een priemelement was. We kunnen het algoritme dus verbeteren door ook deze elementen mee te nemen als 'testpriemen'. Misschien is het programma na zo'n implementatie wel langzamer, maar kan het misschien ook makkelijker bewezen worden dat het algoritme termineert.

```
> BinGcd([1242,3132],[332,3334],[0,1],-19);
[ 4, -1 ]
0 0 [ 1, 0 ]
> BinGcd([1242,3132],[332,3334],[0,1],-43);
[ 1, 0 ]
0 0 [ 1, 0 ]
> BinGcd([1242,3132],[332,3334],[0,1],-67);
[ 1, 0 ]
0 0 [ 1, 0, 0 ]
> BinGcd([1242,3132],[332,3334],[0,1],-163);
[ -1, 0 ]
0 0 [ 1, 0, 0, 0 ]
```

Figuur 2: Output van `BinGcd`

Omdat de helft van de 'moduli' goed is, geldt (omdat we met ω_d en $\overline{\omega_d}$ werken) dat de kans dat de norm daalt sowieso groter is dan $75\% = 1 - 1/4$. Als we ook nog de priemen $1 + \omega_d$ er in implementeren wordt de kans dat de norm daalt groter dan $93.75\% = 1 - 1/16$.

Het lijkt er overigens op dat het algoritme complexiteit $O(n^2)$ of $O(n^2 \log(n))$ heeft.