Die Implementierung des 'Number-Field-Sieve'

Diplomarbeit

von

Jürgen Loho

in Informatik

Mai 1991

Die Arbeit wurde nach einem Thema von *Prof. Dr. Johannes Buchmann* am Fachbereich 14 der Informatik an der Universität des Saarlandes in Saarbrücken verfasst.

Besonders danken möchte ich

Stefan Düllmann für die Implementierung der langen Arithmetik und der LLL-Reduktion,

Jörg Zayer für seine Hilfsbereitschaft und Geduld in Fragen aus der Zahlentheorie,

sowie

Ralf Roth, der dafür gesorgt hat, daß die benötigte Hard- und Software immer verfügbar war.

Hiermit versichere ich an Eides Statt, daß ich diese Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen benutzt habe.

Saarbrücken, 30.März 1991

Inhaltsverzeichnis

Kapitel 1

Einleitung

Das *Number-Field-Sieve* ist eines der neuesten und nach der theoretischen Laufzeitabschätzung das asymptotisch schnellste Verfahren um eine große Zahl $n \in \mathbb{Z}$ zu faktorisieren.

Gesucht ist, wie auch bei anderen Faktorisierungsalgorithmen eine Kongruenz der Form

$$x^2 \equiv y^2 \; mod \; n \quad \text{mit} \quad x \not\equiv y \; mod \; n$$
$$\Rightarrow \gcd(x-y, n) \text{ ist ein nichttrivialer Teiler von } n.$$

Um solche x, y zu finden, suchen wir bei vorgegebenen Basen $B = \{b_1, \cdots, b_s\}$ und $C = \{c_1, \cdots, c_t\}$, wobei $B, C \subset \mathbb{Z}$ sind, Relationen der Form:

$$\prod_{i=1}^{s} b_i^{e_{ij}} \equiv \prod_{l=1}^{t} c_l^{f_{lj}} \; mod \; n \quad j = 1, \cdots, v. \tag{1.1}$$

Diese versuchen wir durch geschicktes aufmultiplizieren in die gewünschten Quatrate zu verwandeln.

Allerdings ist es für eine bestimmte Zahl n sehr schwierig die Voraussetzungen zu erfüllen, die für eine erfolgreiche Anwendung des Algorithmus erforderlich sind. Diese Aufgabe überlasse ich dem Anwender und beschreibe nur die Bedingungen, unter denen mein Programm erfolgversprechend läuft.

Diese Arbeit soll vor allem meine Implementierung des *Number-Field-Sieve* kommentieren. Ich werde deshalb weniger auf die Theorie, als auf die genaue Beschreibung der Algorithmen eingehen. Aus diesem Grunde wird immer nur eine Idee vermittelt, welche Hintergründe die Verfahren haben. Ich hoffe, daß dies ausreichen wird, um einzusehen, daß das Programm die gewünschten Ergebnisse liefert.

Dem, der sich für die Theorie des *Number-Field-Sieve* interressiert, empfehle ich besonders die Diplomarbeit von Jörg Zayer ([**8**]) und desweiteren [**4**], [**6**] und [**7**].

Einige der programmierten Algorithmen weichen von denen in der mir vorliegenden Literatur zum Teil ab. Hier erkläre ich die verschiedenen Möglichkeiten und versuche klar zu machen, warum ich den von mir eingeschlagenen Weg gegangen bin.

Es wurden auch einige Schritte getan, die zur Verallgemeinerung des Algorithmus beitragen sollen. Diese werden ausführlich erläutert und die dabei aufgetretenen Probleme aufgezeigt und teilweise Ideen zur Lösung geliefert.

Kapitel 2

Die Programme

2.1 Arbeitsweise

Das Number Field Sieve setzt sich bei der Implementierung aus 5 Programmen zusammen, die teilweise nacheinander, teilweise parallel nebeneinander laufen können.

Zuerst muß das Programm **pcp** gestartet werden. Dieses sucht zu den Primzahlen p, die kleiner als eine Schranke c sind, die zugehörigen cp's, für die $f(cp) \equiv 0 \ mod \ p$ ist. Die Paare (p, cp) werden in einem File abgespeichert.

pimake tritt als nächstes in Aktion. Es benutzt unter anderem das Ausgabefile von **pcp** als Eingabefile und sucht jetzt die Erzeugenden der zu den Paaren (p, cp) korrespondierenden Primideale. Diese werden wiederum in einem File gespeichert.

Die nächsten drei Programme können theoretisch gleichzeitig gestartet werden.
sieb versucht a, b so zu finden, daß $a + bm$ über der Primzahlbasis und $a + b\rho$ über der Basis aus Erzeugenden von Primidealen zerfallen. Bei beiden Zerlegungen ist je eine große Primzahl (Largeprime) bzw. das Erzeugende eines Primideals großer primer Norm (LargePI), die nicht in der entsprechenden Basis enthalten ist, erlaubt. Auch darf bei $a + b\rho$ noch eine beliebige Einheit auftreten. Die so erhaltenen Relationen werden in ein File geschrieben.

Dort schaut **gleichungen** nach, ob eine neue Kongruenz gefunden wurde. Ist dies der Fall, testet das Programm, ob aus dieser die 'Largeprimes',die 'LargePI' und die Einheit herrauszudividieren sind. Scheitert der Test, merkt sich **gleichungen** die Relation und versucht es zu einem späteren Zeitpunkt mit Hilfe anderer Kongruenzen noch einmal. Ist er aber erfolgreich, haben wir jetzt eine Gleichung ohne 'Largeprime', 'LargePI' und mit der 1 als einziger Einheit. Diese wird zu den Relationen geschrieben, aus denen wir unsere Quadrate kombinieren wollen.

Das ist die Aufgabe von **prglens**, welches auch nur in Aktion tritt, wenn neue brauchbare Relationen gefunden wurden. Das Programm dient eigentlich nur als Schnittstelle zu einer schnellen Gaußelimination, die von A.K. Lenstra geschrieben wurde. Aus diesen Lösungen werden dann die Teiler von n berechnet und ausgegeben.

2.2 Die Eingabefiles

Das File **ein_nfs** muß die wichtigsten Informationen, die zum Start der Programme nötig sind, enthalten. Hier definiert der Benutzer den zu betrachtenden Zahlkörper und legt die Grösse der Faktorbasis, der Siebintervalle und der Largeprimes fest. Er muß selbst darauf achten, daß seine Eingaben korrekt und sinnvoll sind, und daß das Polynom f, welches den Zahlkörper beschreibt, folgenden Bedingungen standhält:
Sei n die zu faktorisierende Zahl;

(1) $\exists\, m \in \mathbb{N}$ mit $f(m) = n$;

(2) f muß irreduzibel über $\mathbb{Q}$ sein;

(3) f muß mindestens eine reelle Nullstelle ρ besitzen;

(4) $\mathbb{Z}[\rho]$ muß ein ZPE-Ring sein.

Das File muß in allen Directories stehen, in denen Teilprogramme gestartet werden sollen. Die Eingabe erfolgt zeilenweise. In jeder Zeile steht nur ein Wert:

- Grad D des Polynoms, das den Zahlkörper beschreibt.
- Koeffizienten des Polynoms, aufsteigend von Grad 0 bis Grad D.
- Größe der Primzahlbasis.
- Alle Nullstellen des Polynoms in komplexer Darstellung, d.h.
  Realteil
  Imaginärteil
  mit 12-stelliger Genauigkeit.
  (Die erste muß Imaginärteil 0 haben!!)
- Obergrenze der Largeprimes, die berücksichtigt werden sollen.
- Anzahl der Siebintervalle.
- Länge eines Siebintervalls.
- Untergrenze der a's, für die $a + bm$ und $a + b\rho$ untersucht werden sollen.
- Name des Ausgabefiles von **sieb**.
- Höhe der Koeffizienten, beim Untersuchen von Linearkombinationen der Basen von Primidealen.

primzahlen ist eine sequentielle Datei mit den ersten 50000 Primzahlen, die zur Erstellung der Faktorbasis B und der Largeprime-Menge benutzt wird. Auch sie muß in allen benutzten Directories stehen.

Um das Suchen der *cp* zu parallelisieren, kann man die Datei **ein_pcp** anlegen. In ihr stehen die Grenzen $1 \leq r_1 < r_2 \leq s$ wobei $s = |B|$ ist. Das Programm **pcp** bearbeitet dann nur die Primzahlen vom r_1-ten bis zum r_2-ten Element der Faktorbasis B. In der dritten Zeile muß dann noch der Name der Datei stehen, in die die Ergebnisse abgespeichert werden sollen.

Ähnlich verhält es sich beim Sieben der $a + bm$ bzw. der $a + b\rho$. Die Grenzen für a wurden ja schon im File **ein_nfs** bestimmt. Da nur über a gesiebt wird, kann man die zu untersuchenden b's in mehrere Teilmengen aufteilen und jeden Aufruf von **sieb** eine davon bearbeiten lassen. Dazu legt man das File **ein_sieb** an. In ihm stehen die Grenzen $b_1, b_2 \in \mathbb{N}$ für b. **sieb** kümmert sich nur um die $a + bm$ bzw. $a + b\rho$, für die $b_1 \leq b \leq b_2$ ist.

Als letztes Eingabefile brauchen die Programme **sieb** und **prglens** die Datei **m**. In ihr muß die Zahl m mit $f(m) = n$ stehen. Die Programme müssen mit '$\langle name \rangle$ < m' aufgerufen werden.

2.3 Das Header-File nfs.h

Das Header-File **nfs.h** sorgt für die Anbindung der Include Dateien *stdio.h, math.h* und *memory.h.* Außerdem enthält es die Konstantendefinitionen, Typdeklarationen und die Definition der globalen Variablen.

1. String-Konstanten:
   - **EINGABE**
EINSIEB
EINPCP
PRIMDATEI } sind die Konstanten für die Namen der eben beschriebenen Eingabefiles.
   - **MATRIX** bezeichnet den Namen des Files, in das die Relationen der Form (2.1) geschrieben werden.
   - In **NEWREL** steht der Name für eines der Eingabefiles der Gaußelimination.
   - **PCPS**
PRIMIDEALE } bezeichnen die Ausgabefiles von **pcp** bzw. **pimake**.
   - **LARGEP**
LARGEPI
EINHEITEN } Dies sind sequentielle Dateien, in denen die Relationen stehen, die Largeprimes oder Einheiten enthalten, die nicht beim ersten Versuch wegdividiert werden konnten.

2. Numerische Konstanten:

   - **LDIM** wird in der langen Arithmetik gebraucht. Es bestimmt die Vorbesetzung der Länge der langen Zahlen.
   - **ASC_1** bezeichnet den ASCII-Code des Characters '1'
   - **BL** ist die Anfangsgröße des Feldes *b_teiler*
   - **MULT** ist der Multiplikator bei der Umwandlung von *double*-Werten in *short*-Werte für die 'Logarithmen der Primidealerzeuger' [1]
   - **EPS** ist die Fehlerschranke, innerhalb derer zwei Logarithmen als gleich gelten.
   - **MULTSHORT** ist der Multiplikator bei der Umwandlung von *double*-Werten in *short*-Werte, für den Logarithmus der $a + bm$ bzw. der Norm von $a + b\rho$.
   - **MAXINT**, **MAXINTBEREICH** } sind Konstanten, die das Ende des Integerbereiches als *int* bzw. als *double* enthalten.
   - **LIST** wird beim Deklarieren und Vorbesetzen von Listen benötigt.
   - **SCHLAF** enthält die Anzahl der Sekunden, die **gleichungen** bzw. **prglens** ruhen sollen, wenn sie nichts zu tun haben, bevor sie wieder nachschauen, ob es Arbeit für sie gibt.
   - **TRUE**, **FALSE** } sind auf 1 bzw. 0 gesetzt.

3. Typedeklarationen:

   - **LZ** ist eine Abkürzung für *unsigned short*, den Type der langen Zahlen.
   - **RING** steht für *int*, welches der Type der Komponenten eines Elementes aus $\mathbb{Z}[\rho]$ ist.
   - **KOMPLEX** ist der Type *double*. In ihm werden die Wurzeln des Polynoms abgespeichert.
   - **TEILER** ist ein Struct der Form:{

     ```
     int nummer;
     int haeufigkeit;
     struct teiler *next;
     }
     ```

     Hierin werden die Zerlegungen von $a + bm$ und $a + b\rho$ abgespeichert. In 'nummer' steht die Stelle des Teilers in der entsprechenden Faktorbasis; in 'haeufigkeit' der dazugehörende Exponent und 'next' verweist auf den nächsten Teiler.

[1]siehe 4.3.1 *Rechnen mit Logarithmen*

4. globale Variablen:

   - **treffer:** Die Variable *char* *treffer ist ein Feld mit der Länge eines Siebintervalles und dient dazu zu markieren, bei welchen Elementen der Siebmenge nach den ersten Teilsieben überhaupt noch Chancen bestehen, vollständig zerlegt zu werden. An der Stelle dieser Elemente steht eine '1' im Trefferfeld, sonst eine '0'.
   - **lgsiebfeld:** Das Feld *short* *lgsiebfeld hat die selbe Länge wie das Trefferfeld. In ihm stehen statt der zu zerlegenden Elemente ihre Logarithmen bzw. die Logarithmen ihrer Normen.

2.4 Die lange Arithmetik

Das Programmpaket zur langen Arithmetik stammt von *Stefan Duellmann* und basiert im wesentlichen auf den *Classical Algorithms* in [4].
Eine lange Zahl wird als Array aus kurzen Zahlen dargestellt. LDIM dient, wie schon erwähnt, zum Festlegen der Länge dieses Arrays, wenn eine lange Zahl zum ersten Mal deklariert wird. Übersteigt die Zahl irgendwann die Länge des Feldes, wird dieses automatisch erweitert.
In jeder Komponente der langen Zahl kann ein Wert zwischen 0 und einer oberen Grenze, die vom Datentyp *short* abhängt, dargestellt werden. Sei s die Anzahl der Bytes einer *short*-Variablen, so ist diese Grenze $2^{8 \cdot s} - 1$, d.h. die Basis der langen Arithmetik ist $2^{8 \cdot s}$. Bei einer Vordimensionierung des Feldes von LDIM = 50 und der Länge einer *short*-Variablen von s = 2, ergibt sich für die größte, ohne Erweiterung des Feldes darstellbare Zahl: $2^{8 \cdot 2(50-1)} - 1 = 2^{784} - 1 \approx 10^{236}$.
Die nullte Komponente spielt bei der Darstellung einer langen Zahl eine besondere Rolle. Sie enthält in den niederwertigen Bits die Länge der Zahl bezüglich der Basis $2^{8 \cdot s}$, also die Anzahl der tatsächlich belegten Stellen des Feldes. Im höchstwertigen Bit ist das Vorzeichen abgespeichert (0 für positiv und 1 für negativ). In den folgenden Komponenten stehen die Stellen der Zahl in aufsteigender Reihenfolge.
Beispiel:

$$x = 4294968530 = 1234 \cdot (2^{16})^0 + 0 \cdot (2^{16})^1 + 1 \cdot (2^{16})^2 \ \Rightarrow$$

$$x[0] = 3;\ x[1] = 1234;\ x[2] = 0;\ x[3] = 1;$$

Die nicht belegten Stellen sind undefiniert, also nicht notwendig gleich Null!

Kapitel 3

Die Module

3.1 pcp

3.1.1 Der Algorithmus

- **Eingabe**
  - $f(x) \in \mathbb{Z}[x]$;
  - $d = deg(f)$;
  - B die Primzahlbasis;
  - $|B|$ = Größe der Primzahlbasis;
  - $0 \leq start, ende \leq |B|$ geben den Bereich der Basis an, der bearbeitet werden soll;
- **Ausgabe**
  - $\{(p, cp) \mid p \in B \wedge f(cp) \equiv 0 \; mod \; p\}$;
  - Anzahl der Paare (p, cp);

Der triviale Algorithmus ist in einem Satz erklärt:
Für alle $p \in B$ teste für alle $0 \leq cp < p$, ob $f(cp) \equiv 0 \; mod \; p$ ist.
Durch ein paar einfache Überlegungen nach Cantor-Zassenhaus, kann man dieses Verfahren erheblich schneller machen:

Bezeichne im weiteren $\mathrm{gcd}_p()$ bzw. $/_p$ die ggT-Berechnung bzw. die Division in $\mathbb{Z}/p\mathbb{Z}[x]$.
Es ist $\prod_{i=0}^{p-1}(x-i) \equiv (x^p - x) \; mod \; p \;\Rightarrow\; \mathrm{gcd}_p(x^p - x, f) = \prod_{j=0}^{e}(x - cp_j) =: g(x)$

wobei die cp_j die Nullstellen von $f \; mod \; p$ sind; und f hat $e = deg(g)$ Nullstellen $mod \; p$.

O.E. sei $p \neq 2$ und 0 keine Nullstelle von $f \ mod \ p$ d. h. der konstante Anteil von $f: \ \ const(f) \not\equiv 0 \ mod \ p \ \Rightarrow \gcd_p(x^p - x, f) = \gcd_p(x^{p-1} - 1, f)$.

$$x^{p-1} - 1 = \left(x^{\frac{p-1}{2}} - 1\right)\left(x^{\frac{p-1}{2}} + 1\right) \ \Rightarrow$$

$$g = \gcd_p\left(x^{\frac{p-1}{2}} - 1, f\right) \cdot \gcd_p\left(x^{\frac{p-1}{2}} + 1, f\right)$$

Aus diesen Betrachtungen entstand folgender verbesserter Algorithmus:

Für alle $p \in B$:
  Falls $f(0) \equiv 0 \ mod \ p$
    speichere $(p, 0)$;

  $g = \gcd_p(x^{p-1} - 1, f)$;
  Falls $deg(g) = 0$:
    $\Rightarrow$ es gibt kein weiteres cp, sodaß $f(cp) \equiv 0 \ mod \ p$
    gehe zum nächsten p.

  Falls $deg(g) = 1$:
    $-const(g) = cp$ ist das letzte cp mit $f(cp) \equiv 0 \ mod \ p$
    speichere (p, cp);
    gehe zum nächsten p.

  berechne $g_0 = \gcd_p(x^{\frac{p-1}{2}} - 1, g)$;
  Falls $deg(g_0) = 1$:
    speichere $(p, -const(g_0))$ ab;
    setze $g \leftarrow g/_p \, g_0$;
    setze $k \leftarrow 1$;

  sonst: Falls $deg(g) - deg(g_0) = 1$:
      speichere $(p, const(g/_p \, g_0))$ ab;
      setze $g \leftarrow g_0$;
      setze $k \leftarrow 1$;

    sonst: setze $g_1 \leftarrow g/_p \, g_0$;
      setze $k \leftarrow 2$;

    $\vdots$

```
⋮
Falls k = 1 :
    Falls deg(g) = 1 :
        speichere (p, −const(g));
        gehe zum nächsten p.

    setze i ← 1;
    solange deg(g) > 1 :
          Falls g(i) ≡ 0 mod p:
              speichere (p, i);
              setze g ← g /_p (x − i);

          setze i ← i + 1;

    speichere (p, const(g));

sonst (* k = 2 *) :
    setze i ← 1;
    setze d ← deg(g);
    solange d > 0 :
          Falls g0(i) ≡ 0 mod p :
              speichere (p, i);
              setze g0 ← g0 /_p (x − i);
              setze d ← d − 1;
              Falls deg(g0) = 1 :
                  speichere (p, −const(g0));
                  setze g0 ← 1;
                  setze d ← d − 1;

          sonst: Falls g1(i) ≡ 0 mod p :
                     speichere (p, i);
                     setze g1 ← g1 /_p (x − i);
                     setze d ← d − 1;
                     Falls deg(g1) = 1 :
                         speichere
                         (p, −const(g1));
                         setze g1 ← 1;
                         setze d ← d − 1;
```

Die schlechtesten Laufzeiten beider Verfahren sind zwar gleich, aber in der Realität zeigt sich für die meisten Polynome f, daß es nur verhältnismäßig wenige p zu f gibt, sodaß f mehr als eine Nullstelle $mod\ p$ hat.

Bei sehr großen Primzahlbasen oder 'schlechten' Polynomen lohnt es sich wahrscheinlich, die Grundidee des Algorithmus zu verallgemeinern:
berechne für einige $0 \leq a_i < p$:

$$\mathrm{gcd}_p\left((x-a_i)^{\frac{p-1}{2}}-1, g(x)\right) = g_{i1}$$
$$g /_p \, g_{i1} = g_{i2}$$

Versuche jetzt durch Bildung von $\mathrm{gcd}_p(g_{ij}, g_{kl}) \quad i \neq k \quad j,l \in \{1,2\}$ die Linearfaktoren *mod p* von g und damit die cp zu finden.

Zur Polynomarithmetik siehe [**2**].

3.1.2 Die Realisierung

- Hauptroutine: *pcp_make()*;
- allgemeine Unterroutinen [1] : *k_primzahlen(),make_RINGfeld();*

Aus dem Eingabefile *ein_nfs* werden der Grad D des Polynoms f, dessen Koeffizienten und die Größe der Faktorbasis B eingelesen.
Anfang und Ende des von diesem Aufruf zu bearbeitenen Primzahlbereichs wird aus *ein_pcp* gelesen.
Jetzt werden mit Hilfe der Prozedur *k_primzahlen* alle Primzahlen bis zur größten zu bearbeitenden aus dem File *primzahlen* eingelesen.
Nachdem das Programm alle nötigen Eingaben erhalten hat, startet die Hauptroutine *pcp_make*.

pcp_make()

Parameter:
int *primzahlen, start, ende, *polynom, *pcplaenge, D, B;

- spezielle Unterroutinen: *gvonxmodp(), polyggt(), teile();*
- allgemeine Unterroutinen [1]: *make_intfeld(), make_intpointerfeld(), setnullint();*

Die Datenstrukturen Polynome werden in *int*-Arrays der Länge $D+1$ abgespeichert. Eine Ausnahme bildet das Feld für die Polynome der Form $x^{p-1}-1$. Dieses hat die Länge der größten zu bearbeitenden Primzahl. Auf seinen Anfang zeigt ein fester Zeiger.
Die Grade der Polynome werden in *int* Variablen geschrieben.
In dem 2-dimensionalen Feld *pcp* der Länge: *Anzahl der zu bearbeitenden Primzahlen*D* werden die gefundenen Paare (p, cp) zwischengespeichert.

[1] Die allgemeinen Unterroutinen werden erst im Kapitel *Allgemeine Routinen* erklärt.

Arbeitsweise Da $p = 2$ in dem eben beschriebenen Verfahren ausgenommen wurde, wird zuerst getestet, ob $f(i) \equiv 0 \ mod \ 2$ für $i = \{0, 1\}$. Jetzt folgt die Prozedur obigem Algorithmus.

Laufzeit Eine grobe Abschätzung für die Laufzeit des Verfahrens im schlechtesten Fall, d.h. wir müssen immer das triviale Verfahren anwenden, ist
$O(D \cdot \frac{L_n[1/3, 2(2/3)^{2/3}]}{(2(2/3)^{2/3}+o(1))(\log n)^{1/3}(\log\log n)^{2/3}})$. Sie resultiert aus der Abschätzung für die Größe der Faktorbasis B (siehe Kapitel 5), aus der Abschätzung für die Anzahl der Primzahlen $<$ x $\approx$ x/ log x und aus der relativ 'schlechten' Annahme, daß das Mittel aller Primzahlen bis x $\approx$ x/2 ist.

polyggt()

Parameter:
int ∗poly1, ∗d1, ∗poly2, ∗d2, p;

- spezielle Unterroutinen: *euklrest(), invmodp();*

Die Prozedur berechnet den ggT der Polynome poly1 und poly2 mod p mittels des Euklidischen Algorithmus. In ∗d1 und ∗d2 stehen die Grade. Das Ergebnis steht in dem Feld, in dem vorher das größere der beiden Polynome stand. Die eingegebenen Polynome werden im Laufe der Rechnung überschrieben. Dies wurde aus Speicherplatzgründen in Kauf genommen.

teile()

Parameter:
int ∗poly, ∗d, ∗poly2, d2, ∗erg, ∗derg, p;

- spezielle Unterroutinen: *invmodp();*

Die Prozedur teilt poly durch poly2 mod p mittels Polynomdivision:

d2 ← deg(poly2);
d ← deg(poly);
derg ← d-d2;
berechne multinv ← das multiplikative Inverse von poly2[d2];
solange d $\leq$ d2:
    inv ← (multinv·poly[d]) mod p;
    z ← d-d2;
    erg[z] ← inv;
    poly ← (poly-inv·poly2) mod p;
    d ← deg(poly);

Das Ergebnis steht jetzt in erg, der Rest in poly. Die Grade werden in *d, d2 und *derg übergeben.

int **gvonxmodp()**

Parameter:
int *g, x, p, d;

Die Funktion berechnet $g(x)\ mod\ p$ und gibt das Ergebnis als *int* Funktionswert zurück. In d steht der Grad von g.

euklrest()

Parameter:
int *poly, *d, *poly2, d2, p;

- spezielle Unterroutinen: *invmodp();*

Die Prozedur gibt in poly den Rest der Division poly durch poly2 mod p zurück. Die entsprechenden Grade sind wieder *d und d2.

int **invmodp()**

Parameter:
int x, p;

Die Funktion liefert als *int* das multiplikative Inverse von x mod p zurück. Hierzu wird der erweiterte Euklidische Algorithmus benutzt.
Denn es gilt: $p \in \mathbb{P}, x \not\equiv 0\ mod\ p \Rightarrow \gcd(x,p) = 1 \Rightarrow 1 = ax + bp \Rightarrow 1 \equiv ax\ mod\ p$
d.h. a ist das multiplikative Inverse modulo p von x.

$y \leftarrow p;$
$rest \leftarrow x;$
$l_0 \leftarrow 0;$
$l_1 \leftarrow 1;$
solange $rest \neq 1:$
    $teiler \leftarrow y\ div\ x;$
    $rest \leftarrow y\ mod\ x;$
    $l_2 \leftarrow l_0 - teiler \cdot l_1;$
    $l_0 \leftarrow l_1;$
    $l_1 \leftarrow l_2;$
    $y \leftarrow x;$
    $x \leftarrow rest;$

$a \leftarrow l_1\ mod\ p;$

3.2 pimake

3.2.1 Der Algorithmus

- **Eingabe:**
  - Die Paare (p, cp) die von pcp gefunden wurden.
  - Anzahl der Paare (p, cp)
  - $f(x) \in \mathbb{Z}[x]$
  - $D = deg(f)$
  - Die Primzahlbasis B
  - $|B|$ die Größe der Faktorbasis
  - Die Nullstellen von f
- **Ausgabe**
  - $\{(p, cp, \alpha = a_0 + a_1\rho + \cdots + a_{D-1}\rho^{D-1}) \mid p \in B \ \wedge f(cp) \equiv 0 \ mod \ p \ \wedge$ $a_0 + a_1 cp + \cdots + a_{D-1} cp^{D-1} \equiv 0 \ mod \ p \wedge N(\alpha) = p\}$
  - Anzahl der Trippel (p, cp, α)

1. Idee

Vor: Man kann in $\mathbb{Z}[\rho]$ schnell dividieren.
Sei $B = \{p_1, \ldots, p_{|B|}\}$ mit $p_1 < p_2 < \cdots < p_{|B|}$;
Solange $\exists$ (p, cp), für das noch kein (p, cp, α, k) gefunden wurde:
  Für viele $\alpha = a_0 + a_1\rho + \cdots + a_{D-1}\rho^{D-1} \in \mathbb{Z}[\rho]$ mit kleinen Koeffizienten:
    Falls $|N(\alpha)| = k \cdot p$ mit $p \in B, k < p$ und k quadratfrei:
      teste für welches cp $\quad a_0 + a_1 cp + \cdots + a_{D-1} cp^{D-1} \equiv 0 \ mod \ p$ ist;
      Falls schon ein (p, cp, β, k_1) gefunden wurde:
        Falls $k_1 > k$:
          streiche (p, cp, β, k_1);
          speichere (p, cp, α, k);

      sonst: speichere (p, cp, α, k);

Für $i = 1$ bis $|B|$:
  Für alle $(p = p_i, cp, \alpha, k)$:
    solange $\exists\, j < i$ mit $p_j | k \wedge a_0 + a_1 cp_{p_j} + \cdots + a_{D-1} cp_{p_j}^{D-1} \equiv 0 \; mod \; p_j$:
      $\alpha \leftarrow \alpha / \alpha_j$;
      $k \leftarrow k / p_j$;

    speichere (p, cp, α);

Im Allgemeinen ist es so, daß es keinen schnellen Algorithmus zum dividieren in $\mathbb{Z}[\rho]$ gibt. Deshalb habe ich mir zuerst mit folgender vereinfachter Methode beholfen:

Für sehr viele $\alpha = a_0 + a_1\rho + \cdots + a_{D-1}\rho^{D-1} \in \mathbb{Z}[\rho]$ mit kleinen Koeffizienten:
  Falls $N(\alpha) = p \in B$:
    teste für welches cp $a_0 + a_1 cp + \cdots + a_{D-1} cp^{D-1} \equiv 0 \; mod \; p$ ist;
    speichere (p, cp, α);

Auch das wird solange gemacht, bis zu jedem (p, cp) ein α gefunden wurde.

2. Idee

Im Gegensatz zum obigen Ansatz, wird bei diesem Verfahren konkret ein Erzeuger des Primideals $\mathcal{P}$, das zu einem bestimmten (p, cp) korrespondiert, gesucht.

$$f(x) = f_0 + f_1 x + \cdots + f_{D-1} x^{D-1} + x^D$$

$$\mathcal{P} = \langle p, \rho - cp \rangle \Rightarrow$$

$$\begin{pmatrix} p & 0 & \cdots & \cdots & 0 & -cp & 0 & \cdots & 0 & -f_0 \\ 0 & p & \ddots & & \vdots & 1 & -cp & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots & 0 & 1 & \ddots & 0 & -f_{D-3} \\ \vdots & & \ddots & p & 0 & \vdots & \ddots & \ddots & -cp & -f_{D-2} \\ 0 & \cdots & \cdots & 0 & p & 0 & \cdots & 0 & 1 & -cp - f_{D-1} \end{pmatrix} \tag{3.1}$$

ist ein Erzeugendensystem des Ideals. Dieses kann man auf Hermite Normalform bringen und erhält dann eine Basis des Primideals:

$$\begin{pmatrix} 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \cdots & 0 & 1 & 0 \\ e_0 & e_1 & \cdots & e_{D-2} & p \end{pmatrix} \tag{3.2}$$

Wendet man darauf jetzt die LLL-Reduktion an, so bekommt man eine Basis aus kurzen Vektoren. Nun wird getestet, ob einer der Basisvektoren Norm p hat. Ist dies der Fall, speichern wir diesen zusammen mit p und cp ab.
Ansonsten testen wir kleine Linearkombinationen der Basisvektoren auf ihre Norm, bis wir eine mit Norm p gefunden haben.

Der entsprechende Algorithmus lautet also wie folgt:

Für alle Paare (p, cp):
  erstelle Matrix der Form (3.1);
  berechne die Hermite Normalform dieser Matrix;
  führe LLL-Reduktion auf der Matrix durch;
  interpretiere die Spalten der Matrix $(\alpha_0 \cdots \alpha_{D-1})$ als Elemente von $\mathbb{Z}[\rho]$;
  Falls $\exists \alpha_i$ mit $N(\alpha_i) = p$:
    speichere (p, cp, α_i);

  sonst: Für viele $c_0\alpha_0 + \cdots + c_{D-1}\alpha^{D-1} = \beta$ mit c_i 'klein':
    Falls
    $N(\beta) = p$:
    speichere (p, cp, β);
    stop;

3.2.2 Die Realisierung

- Hauptroutine: *primidealbestimmung();*
- allgemeine Unterroutinen: *wurzelpot(), k_primzahlen(), make_RINGfeld(), make_wurzeln();*

Aus dem Eingabefile *ein_nfs* werden der Grad D des Polynoms f, dessen Koeffizienten, die Größe der Faktorbasis, die Nullstellen von f und MAXHOEHE eingelesen. Anschließend werden von *wurzelpot* die Potenzen der Nullstellen mit komplexer Arithmetik berechnet und gespeichert. *k_primzahlen* liest dann die Faktorbasis ein. Jetzt sind alle nötigen Eingaben verfügbar und *primidealbestimmung* startet.

primidealbestimmung()

Parameter:
RING $*$polynom;
KOMPLEX $***$wurzeln;

int D, $*$primzahlen, B, MAXHOEHE;

- spezielle Unterroutinen: *hnf(), Norm_k(), polyvonnull(), mat_besetz(), linkomb(), hochzaehlen2();*
- allgemeine Unterroutinen: *make_intfeld(), make_intpointerfeld(), make_RINGfeld(), lange_make(), umwandle(), ring_zuweisung(), ring_neg(), erstellePCPS(), pcp_einlese();*

Datenstrukturen Die Paare (p, cp) sind in dem zweidimensionalen Feld *int* $**$pcp abgespeichert. In pcp[0][i] stehen die $p \in B$ und in pcp[1][i] die zugehörigen cp's. Die in dem Verfahren benötigte Matrix ist als *int* $**$mat deklariert. Allerdings werden die Einträge nicht wie gewohnt zeilenweise sondern spaltenweise gespeichert, d.h. mat[0] zeigt auf den ersten Spaltenvektor, also mat[i][j] ist der Eintrag der j-ten Zeile in der i-ten Spalte. Dadurch werden viele Operationen in *hnf()* einfacher.

int $*$erzeuger, $*$vek, $*$testelement, $*$zaehler sind Vektoren der Länge D, in denen Elemente aus $\mathbb{Z}[\rho]$ gespeichert werden.
Auch *int* $*$ggtfeld ist ein Vektor der Länge D. Seine Funktion wird hoffentlich bei der Beschreibung von *hochzaehlen2()* deutlich.

Arbeitsweise Im wesentlichen folgt die Prozedur dem Algorithmus der 2. Idee. Interessant ist hierbei vielleicht die Realisierung der *Für viele* $c_0\alpha_0 + \ldots$ Schleife:
$\vec{c} = (c_0, \ldots, c_{D-1}) \leftarrow (-1, \ldots, -1, 1)$;
$hoehe \leftarrow 1$;
solange noch kein Element β mit $N(\beta) = p$ gefunden wurde und hoehe $<$ MAXHOEHE
Falls $N(c_0\alpha_0 + \cdots + c_{D-1}\alpha_{D-1}) = p$:
speichere $(p, cp, \beta = c_0\alpha_0 + \cdots + c_{D-1}\alpha_{D-1})$;

sonst :
zähle $\vec{c}$ hoch;

Den letzten Befehl erledigt die Prozedur *hochzaehlen2()*, und der Algorithmus wird dort erklärt. Zusätzlich zu (p, cp, β) speichern wir für später ersichtliche Zwecke die Vorkommastellen von **MULTSHORT**$\cdot\log(\beta)$ ab.

hnf()

Parameter:
int $**$mat, D, p;

- spezielle Unterroutinen: *findemin(), spaltentausch(), negspalte(), redzeile(), redmatrix(), [mlll_init(), mlll(), mlll_frei];*

Diese Prozedur bringt die $D \times 2D$ Matrix mat in Hermite Normalform [2] nach folgendem Algorithmus:
Für Zeile $i = 0$ bis $D - 1$ von mat:
    finde Spalte $j \geq i$ mit minimalem i-ten Zeileneintrag;
    solange $j \neq i$:
        tausche die Spalten i und j;
        Falls $mat[i][i] < 0$:
            negiere Spalte i;

        Für alle Spalten $k > i$:
            solange $|mat[k][i]| \geq mat[i][i]$:
                Spalte k $\leftarrow$ Spalte $k - sign(mat[k][i]) \cdot$ Spalte i;

        finde Spalte $j \geq i$ mit minimalem i-ten Zeileneintrag;

Für Spalte $i = 1$ bis $D - 1$:
    Für Spalte $j = 0$ bis i:
        solange $mat[j][i] < mat[i][i]$:
            Spalte j $\leftarrow$ Spalte j + Spalte i;

        solange $mat[j][i] \geq mat[i][i]$:
            Spalte j $\leftarrow$ Spalte j − Spalte i;

Danach wird die LLL-Reduktion durchgeführt.

Norm_k()

Parameter:
Ring ∗ringelement;
int D, ∗norm;
KOMPLEX ∗ ∗ ∗wurzeln;
double ∗Norm;

- spezielle Unterroutinen: *ringvonrho();*

- allgemeine Unterroutinen: *komplex_mult();*

Die Norm von ringelement wird mit Hilfe der Wurzeln berechnet:

[2] siehe auch **[1]**

Seien $\rho_{(1)}, \rho_{(2)}, \cdots, \rho_{(D)}$ die Wurzeln von $f(x)$
und $ringelement = a_{0,0} + a_{1,0}\rho + \cdots + a_{D-1,0}\rho^{D-1}$
Sei $g(x) = a_{0,0} + a_{1,0}x + \cdots + a_{D-1,0}x^{D-1}$
Es gilt:

$$Norm(ringelement) = \prod_{i=1}^{D} g(\rho_{(i)})$$

Falls das Ergebnis den *int*-Bereich überschreitet, liefert die Prozedur 0 und das Ergebnis steht nur in Norm, sonst liefert sie 1 und das Ergebnis steht zusätzlich noch in norm.

int **polyvonnull()**

Parameter:
Ring ∗ringelement;
int erg, D;
KOMPLEX ∗∗NS;

- spezielle Unterroutinen: *ringvonrho();*
- allgemeine Unterroutinen: *reel();*

$ringelement[0] + ringelement[1] \cdot NS + \cdots + ringelement[D-1] \cdot NS^{D-1}$
wird berechnet. Dies ist im allgemeinen eine komplexe Zahl. Von dieser wird der Logarithmus des Betrags berechnet und mit **MULT** multipliziert. Die Vorkommastellen werden in erg zurückgegeben.

mat_besetz()

Parameter:
int ∗∗matrix, D, ∗f, p, cp;

- allgemeine Unterroutinen: *multrho();*

In diesem Unterprogramm wird eine Matrix der Form (3.1) erstellt, die dann **hnf()** übergeben werden kann.

linkomb()

Parameter:
int ∗∗mat, ∗zaehler, D, vek;

- allgemeine Unterroutinen: *setnullint(), ring_add();*

mat wird mit zaehler multipliziert, wobei mat eine $D \times D$ Matrix ist und zaehler ein D-dimensionaler Vektor, genau wie vek, in dem das Ergebnis zurückgegeben wird.

hochzaehlen2()

Parameter:
int $*$feld, $*$ggtfeld, cnt, $*$hoch, D;

Wie wir schon gesehen haben, stellen wir an den Algorithmus folgende Anforderungen:

(i) Er soll uns einen D-dimensionalen Vektor mit betragsmäßig kleinen Koeffizienten liefern.

(ii) Es sollen keine Wiederholungen vorkommen.

(iii) Es sollen keine Vektoren $\alpha = (a_0, \ldots, a_{D-1})$ mit $\gcd(a_0, \ldots, a_{D-1}) \neq 1$ vorkommen.

(iv) Der Algorithmus soll alle Möglichkeiten an Vektoren, unter Berücksichtigung von (i)-(iii), ausschöpfen.

Die Prozedur *hochzaehlen2* realisiert eine Funktion von $\mathbb{Z}^D \to \mathbb{Z}^D$, für die gilt:
Die Menge $\{hochzaehlen2^*((-1, \ldots, -1, 1))\}$ erfüllt die Bedingungen (ii)-(iv) und sei
$hochzaehlen2^{(i)}((-1, \ldots, -1, 1)) = (a_0, \ldots, a_{D-1})$
$hochzaehlen2^{(j)}((-1, \ldots, -1, 1)) = (b_0, \ldots, b_{D-1})$
Für alle $i, j \in \mathbb{N}$ gilt:

$$i < j \quad \Rightarrow \quad max\{|a_0|, \ldots, |a_{D-1}|\} \leq max\{|b_0|, \ldots, |b_{D-1}|\}$$

Dies besagt, daß, benutzt man die Ausgabe eines Aufrufs der Prozedur als Eingabe des nächsten Aufrufs, Vektoren mit kleinen Einträgen vor Vektoren mit großen Einträgen als Ausgabe vorkommen.

Definition der Funktion $nachfolger(\alpha)$:
$\alpha = (a_0, \ldots, a_{D-1}) \in \mathbb{Z}^D$;
$Max = max\{a_0, \ldots, a_{D-1}\}$;
$i = max\{j \mid a_j = Max\}$;
$k = min\{j \mid a_j \neq Max$ für $j < i$ oder $a_j \neq Max - 1$ für $j > i\}$;

$$k < i: \; nachfolger((Max, \ldots, Max, a_k, \ldots, \underbrace{Max}_{i-te\ Stelle}, \ldots, a_{D-1})) := (-Max, \ldots, -Max, a_k + 1, \ldots, \underbrace{Max}_{i-te\ Stelle}, \ldots, a_{D-1});$$

$$k > i: \; nachfolger((Max, \ldots, Max, \underbrace{Max}_{i-te\ Stelle}, Max - 1, \ldots, Max - 1, a_k, \ldots, a_{D-1})) := (-Max, \ldots, -Max, \underbrace{Max}_{i-te\ Stelle}, -(Max - 1), \ldots, -(Max - 1), a_k + 1, \ldots, a_{D-1});$$

$$k = D; i > 0: \; nachfolger((Max, \ldots, Max, \underbrace{Max}_{i-te\ Stelle}, Max - 1, \ldots, Max - 1)) := (-Max, \ldots, -Max, \underbrace{Max}_{(i-1)-te}, \underbrace{-(Max - 1)}_{i-te\ Stelle}, \ldots, -(Max - 1));$$

$$\Rightarrow \quad i \leftarrow i-1;$$

$$k = D; i = 0: \ nachfolger((\underbrace{Max}_{i-te\ Stelle}, Max-1, \ldots, Max-1)) :=$$
$$(-(Max+1), \ldots, -(Max+1), Max+1)$$
$$\Rightarrow \quad i \leftarrow D-1; \quad Max \leftarrow Max+1;$$

hochzaehlen2(α) benutzt diese Funktion:

$\beta \leftarrow nachfolger(\alpha) = (b_0, \ldots, b_{D-1});$
solange $\gcd(b_0, \ldots, b_{D-1}) \neq 1$:
$\beta \leftarrow nachfolger(\beta);$

Zur Berechnung von $\gcd(b_0, \ldots, b_{D-1})$ wird ein Feld $ggtfeld$ so angelegt, daß:
$ggtfeld[D-1] = |b_{D-1}|;$
Für $i = D-2$ bis 0:
$ggtfeld[i] = \gcd(ggtfeld[i+1], |b_i|)$

D.h. der gesuchte ggT steht in $ggtfeld[0]$.

Um diese Schleife nicht jedes mal durchlaufen zu müssen, benutzen wir, daß die Änderungen in *ggtfeld* nach der Anwendung der $nachfolger$-Funktion von den Änderungen in β abhängen.

Seien k, i wie eben bei Austritt aus der Funktion $nachfolger()$:

Falls $k < D$:
besetze alle Einträge von $ggtfeld[j]$ mit $0 \leq j \leq k$ neu;

sonst: Falls $i < D-1$:
besetze alle Einträge von $ggtfeld[j]$ (mit $i+1 \leq j \leq D-1$) mit Max-1;
besetze alle Einträge von $ggtfeld[j]$ (mit $0 \leq j \leq i$) mit 1;
(* da gcd(Max,Max-1) = 1 $\forall$ Max$\in \mathbb{N}$ *)

sonst: (* $i = D-1$ *)
besetze alle Einträge von $ggtfeld[j]$ $(0 \leq j \leq D-1)$ mit Max+1;

int **findemin()**

Parameter:
int **mat, i, D, *minstelle;

Die Funktion sucht in der $D \times 2D$ Matrix mat den minimalen Eintrag der i-ten Zeile und gibt in minstelle die Spaltennummer des Eintrags zurück. Sie liefert als Funktionswert 0, wenn die i-te Zeile eine 0-Zeile ist, 1 sonst.

spaltentausch()

Parameter:
int ∗sp1, ∗sp2, zeilen;

Die Einträge der Vektoren sp1 und sp2, die beide die Länge zeilen haben, werden getauscht.

neg_spalte()

Parameter:
int ∗sp, n;

Die Prozedur negiert die Einträge der Spalte sp. n ist die Länge von sp.

redzeile()

Parameter:
int ∗∗mat, i, D, p;

- spezielle Unterroutinen: *lspaltenadd()*;

Die Einträge in der i-ten Zeile aller Spalten > i, werden auf Einträge, die betragsmäßig kleiner sind als mat[i][i], durch geschickte Additionen der i-ten Spalte reduziert.

redmatrix()

Parameter:
int ∗∗mat, D, p;

- spezielle Unterroutinen: *lspaltenadd()*;

Bei Eintritt in die Prozedur ist mat bereits in unterer Dreiecksgestalt. *redmatrix()* verkleinert jetzt wiederum durch geschickte Addition von Spalten die Einträge unterhalb der Diagonalen solange, bis sie größer oder gleich 0 aber kleiner als das Diagonalelement der zugehörigen Zeile sind.

lspaltenadd()

Parameter:
int a, ∗spalte1, ∗spalte2, D, p;

Die Prozedur berechnet (a·spalte1+spalte2) mod p.

ringvonrho()

Parameter:
RING ∗ringelement;
int D;
KOMPLEX ∗∗wurzel, ∗erg;

Sei $g(x) = ringelement[0] + ringelement[1]\,x + \cdots + ringelement[D-1]\,x^{D-1}$. Die Prozedur berechnet g(Wurzel) und gibt das Ergebnis in erg zurück. Um hier nicht immer die Potenzen der Wurzel berechnen zu müssen, wurde dies im Voraus getan und in dem zweidimensionalen Feld wurzel gespeichert.

mlll

Die hier benutzte LLL-Reduktion wurde von *Stefan Düllmann* implementiert; ich habe das Prozedurenpacket *mlll.c* übernommen, und für meine Zwecke benutzt. Der Algorithmus folgt im Wesentlichen den Ideen aus:

A.K. Lenstra, H.W. Lenstra Jr. und L. Lovasz *Factoring polynomials with rational coefficients*, Math. Ann. **261** (1982), 515–534.

M. Pohst *A modification of the LLL-algorithm*, J. Symbolic Computation **4** (1987), 123–127.

3.3 sieb

3.3.1 Der Algorithmus

- **Eingabe**
  - Grad D des Polynoms;
  - Die Koeffizienten des Polynoms;
  - $|B|$ Größe der Faktorbasis;
  - Die Wurzeln des Polynoms;
  - Die Größenordnung der größten Largeprime;
  - Anzahl der Teilintervalle;
  - Länge eines Teilintervalls;
  - Name der Relationendatei;
  - Die Trippel (p, cp, α);
  - Startwert des Siebes für a;
  - Startwert für b;
  - Endwert für b;
- **Ausgabe**
  - Relationen, in der Form:
    - 0 : keine Largeprime; 1 : Largeprime vorhanden;
    - Largeprime, falls vorhanden;
    - 0 : Einheit; 1 : Element mit Norm = Largeprime;
    - Logarithmus der Einheit bzw. des Elementes;
    - Zerlegung von $a + bm$:
      - Stelle der Primzahl in B;
      - Exponent der Primzahl;
    - -1;
    - Zerlegung von $a + b\rho$:
      - Stelle des Elementes in C;
      - Exponent des Elementes;

Die Aufgabe von **sieb** besteht darin, zu testen, ob einerseits
(1) $a + bm$ über der Faktorbasis B und andererseits
(2) $a + b\rho \in \mathbb{Z}[\rho]$ über der Basis C aus Erzeugenden von Primidealen zerfällt,
wobei $a, b \in \mathbb{Z}$ mit
(3) $\gcd(a, b) = 1$.

Denn sei $\gcd(a,b) = d \neq 1 \quad \Rightarrow \quad a + bm = d(a' + b'm)$ und $a + b\rho = d(a' + b'\rho)$ und dies würde zu der selben Relation wie die Zerlegungen von $a' + b'm$ und $a' + b'\rho$ führen.

Sei $p \in B$:

(i) $p|\gcd(a,b) \quad \Rightarrow \quad p|\gcd(a+p,b)$;

(ii) $p|a+bm \quad \Rightarrow \quad p|(a+p)+bm$;

(iii) $p|Norm(a+b\rho) \;\Rightarrow\; \exists$ ein Primideal $\mathcal{P} = \langle\alpha\rangle$ mit Norm p, sodaß $\alpha|a+b\rho$. Zu $\mathcal{P}$ korrespondiert ein Paar $(p, cp) \;\Rightarrow\; a + b\,cp \equiv 0 \; mod \; p \;\Rightarrow$ $(a+p) + b\,cp \equiv 0 \; mod \; p \;\Rightarrow\; p|Norm((a+p)+b\rho)$;

Aus diesen Feststellungen resultiert folgende grundsätzliche Vorgehensweise:
Die Intervalle für a, b, in denen $a + bm$ bzw. $a + b\rho$ untersucht werden sollen, seien $\{a_{unten}, \ldots, a_{oben}\}$ bzw. $\{b_{unten}, \ldots, b_{oben}\}$, wobei $b_{unten} > 0$ ist. Da für kleine b die meisten Relationen gefunden werden, wählt man am besten $b_{unten} = 1$.
Jetzt legen wir ein $trefferfeld$ für a an; d.h. ein Feld der Länge $a_{lang} = a_{oben} - a_{unten} + 1$. Die Einträge von $trefferfeld$ liegen in $\{0,1\}$. Für ein festes b bedeutet $trefferfeld[i] = 0$, daß das Paar $(a_{unten} + i, b)$ eine der Forderungen (1)-(3) nicht erfüllt, und damit unbrauchbar für unsere Zwecke ist.

Die folgenden Schritte werden für alle $b_{unten} \leq b \leq b_{oben}$ durchgeführt.

<u>1. Schritt:</u> $\gcd(a,b)$
berechne alle Teiler $q_1, \ldots, q_l \in \mathbb{P}$ von b;
Für $i = 1$ bis l:
 berechne kleinstes a mit $a \geq a_{unten}$ und $q_i|a$;
 $j \leftarrow a - a_{unten}$;
 solange $j < a_{lang}$:
  $trefferfeld[j] \leftarrow 0$;
  $j \leftarrow j + q_i$;

Nach diesem Schritt steht nur noch in den $trefferfeld[i]$ eine 1, für die $\gcd(a_{unten} + i, b) = 1$ ist.

<u>2. Schritt:</u> Test, ob $a + bm$ vollständig über B zerfällt.
Hierzu legen wir ein weiteres Feld $siebfeld$ der Länge a_{lang} an und belegen es mit 1.
Sei jetzt $B = \{p_1, \ldots, p_{|B|}\}$;
Für $i = 1$ bis $|B|$:
    berechne kleinstes a mit $a \geq a_{unten}$ und $p_i|(a + bm)$;
    $j \leftarrow a - a_{unten}$;
    solange $j < a_{lang}$:
        Falls $trefferfeld[j] = 1$:
            $siebfeld[j] \leftarrow siebfeld[j] \cdot p_i$;
            $P \leftarrow p_i^2$;
            solange $P|((a_{unten} + j) + bm)$:
                $siebfeld[j] \leftarrow siebfeld[j] \cdot p_i$;
                $P \leftarrow P \cdot p_i$;

        $j \leftarrow j + p_i$;

Für $j = 0$ bis $a_{lang} - 1$:
    Falls $trefferfeld[j] = 1$:
        Falls $siebfeld[j] \neq ((a_{unten} + j) + bm)$:
            $trefferfeld[j] \leftarrow 0$;

Jetzt steht in $trefferfeld[i]$ nur noch dann eine 1, wenn $\gcd(a_{unten} + i, b) = 1$ ist, und $(a_{unten} + i) + bm$ über B zerfällt.

<u>3. Schritt:</u> Test, ob $a + b\rho$ vollständig über C zerfällt.
Wir besetzen $siebfeld$ wieder mit 1 vor.
Sei $C' = \{\alpha_1, \ldots, \alpha_{|C|}\}$ und (p_i, cp_i) das zu α_i korrespondierende Paar.
Für $i = 1$ bis $|C|$:
    berechne kleinstes a mit $a \geq a_{unten}$ und $a + b\,cp \equiv 0 \; mod \; p$;
    $j \leftarrow a - a_{unten}$;
    solange $j < a_{lang}$:
        Falls $trefferfeld[j] = 1$:
            $siebfeld[j] \leftarrow siebfeld[j] \cdot p_i$;
            $P \leftarrow p_i^2$;
            solange $P|Norm((a_{unten} + j) + b\rho)$:
                $siebfeld[j] \leftarrow siebfeld[j] \cdot p_i$;
                $P \leftarrow P \cdot p_i$;

        $j \leftarrow j + p_i$;

Für $j = 0$ bis $a_{lang} - 1$:
  Falls $trefferfeld[j] = 1$:
    Falls $siebfeld[j] \neq Norm((a_{unten} + j) + b\rho)$:
      $trefferfeld[j] \leftarrow 0$;

$trefferfeld$ sagt uns, für welche a sich $a + bm$ über B und $a + b\rho$ über C zerlegen lassen.

Nun folgt die Zerlegung selbst:
Für $j = 0$ bis $a_{lang} - 1$:
  Falls $trefferfeld[j] = 1$:
    zerlege mit Probedivision $(a_{unten} + j) + bm = ({p_1}^{e_1} \cdots {p_{|B|}}^{e_{|B|}})$ über B;
    Für alle i mit $e_i \neq 0$:
      speichere i, e_i;

    zerlege mit Probedivision $Norm((a_{unten}+j)+b\rho) = ({p_1}^{e_1} \cdots {p_{|B|}}^{e_{|B|}})$ über B;
    Für alle i mit $e_i \neq 0$:
      suche das $(p_i = p_k, cp_k)$ mit $(a_{unten} + j) + b\,cp_k \equiv 0 \; mod \; p_k$;
      speichere k, e_i;

Idee der Largeprimevariante:

Sollte das bisherige Verfahren zu wenig Relationen finden, was im allgemeinen auch der Fall ist, so kann man folgendes versuchen:
Man läßt zu, daß bei der Zerlegung von $a + bm$ bzw. $Norm(a + b\rho)$ noch eine Primzahl p_L aus $p_{|B|} < p_L \leq MAXLARGE \leq {p_{|B|}}^2$ mit Exponent 1 auftreten darf. Auch solche Relationen speichern wir und versuchen dann später, mit Hilfe anderer, gleichartiger Relationen, die großen Primzahlen wieder herauszudividieren.
In den Sieben müssen dafür nur in den letzten Schleifen die Zeilen

Falls $siebfeld[j] \neq (a_{unten} + j) + bm$
bzw.
Falls $siebfeld[j] \neq Norm((a_{unten} + j) + b\rho)$
in
Falls $siebfeld[j] > \frac{(a_{unten}+j)+bm}{MAXLARGE}$
bzw.
Falls $siebfeld[j] > \frac{Norm((a_{unten}+j)+b\rho)}{MAXLARGE}$

geändert werden.
Bei der Zerlegung muß dann das p_L bzw. das Element, welches mit einem (p_L, cp_L) korrespondiert, gesondert abgespeichert werden.
Für diesen Schritt ist es vorher natürlich nötig, **pcp** und **pimake** bis $MAXLARGE$ laufen zu lassen.

Sieben über Primzahlpotenzen:

Beim Sieben machen wir nach dem derzeitigen Verfahren sehr viele Operationen umsonst. Wir berechnen die Potenzen von p_i jedes mal neu z.B. wird ${p_i}^2$ $a_{lang}\, div\, p_i$ mal berechnet.
Auch die Probedivisionen für den Test $P|((a+j)+bm)$ werden für alle $a_{lang}\, div\, p_i$ in Frage kommenden j und den hier auftretenden Potenzen von p_i gemacht.
Dies läßt sich vermeiden, indem man zum ersten die Potenzen der $p_i \in B$ bis zu einer bestimmten Größe, vorher einmal berechnet und abspeichert; und zum zweiten auch über diese siebt.
Sei e_i die Potenz der i-ten Primzahl so, daß ${p_i}^{e_i} < a_{lang} \leq {p_i}^{e_i+1}$

$$\begin{aligned}\text{Sei } B' &= \{(p_1,p_1),({p_1}^2,p_1),\ldots,({p_1}^{e_1},p_1),(p_2,p_2),\ldots,({p_2}^{e_2},p_2),\ldots,({p_{|B|}}^{e_{|B|}},p_{|B|})\}\\ &= \{(q_1,r_1),(q_2,r_2),\ldots,(q_{|B'|},r_{|B'|})\} \text{ und } |B'| = \sum_{i=1}^{|B|} e_i;\end{aligned}$$

Für $i = 1$ bis $|B'|$:
- berechne kleinstes a mit $a \geq a_{unten}$ und $q_i|(a+bm)$;
- $j \leftarrow a - a_{unten}$;
- solange $j < a_{lang}$:
  - Falls $trefferfeld[j] = 1$:
    - $siebfeld[j] \leftarrow siebfeld[j] \cdot r_i$;
    - Falls $i = |B|$ oder $r_{i+1} \neq r_i$:
      - (* q_i ist höchste gespeicherte Potenz von r_i *)
      - $P \leftarrow q_i \cdot r_i$;
      - solange $P|((a+j)+bm)$:
        - $siebfeld[j] \leftarrow siebfeld[j] \cdot r_i$;
        - $P \leftarrow P \cdot r_i$;
  - $j \leftarrow j + q_i$;

Die letzte Schleife ist wieder wie oben.
Es werden jetzt nur noch für $a_{lang}\, div\, p_i^{e_i}$ in Frage kommende j vergebliche Rechnungen gemacht.

Rechnen mit Logarithmen:

Da Multiplikationen und Divisionen mehr Zeit brauchen als Additionen und Subtraktionen, bringt es einen erheblichen Zeitgewinn, wenn man beim Sieben mit den Logarithmen der Primzahlen bzw. von a+bm und der $Norm(a+b\rho)$ arbeitet. Dazu berechnet man zuerst die Logarithmen aller Primzahlen und speichert sie zu den Paaren aus B'. Allerdings würde es uns viel Speicherplatz kosten, und zu hohe Anforderungen an die Rechengenauigkeit der Maschine stellen, wenn wir die Logarithmen im *double*-Format speichern würden. Eine andere Möglichkeit ist: wir multiplizieren die Logarithmen mit einer gewissen 10-er Potenz **MULTSHORT**$= 10^L$ und streichen die Nachkommastellen. **MULTSHORT** ist so gewählt, das die erhaltenen Werte in *short*-Variablen passen.

$Log()$ sei die Funktion, die dies tut.
Für das Sieb von $a + bm$ heißt das:

Sei $B' = \{(p_1, p_1, Log(p_1)), ({p_1}^2, p_1, Log(p_1)), \ldots, ({p_{|B|}}^{e_{|B|}}, p_{|B|}, Log(p_{|B|}))\}$
$= \{(q_1, r_1, s_1), \ldots, (q_{|B'|}, r_{|B'|}, s_{|B'|})\};$

ersetze jetzt die Zeilen

$$siebfeld[j] \leftarrow siebfeld[j] \cdot r_i;$$
durch
$$siebfeld[j] \leftarrow siebfeld[j] + s_i;$$

Die letzte Schleife muß dann so aussehen:
$grenze \leftarrow Log(b \cdot m) - Log(MAXLARGE);$
Für $j \leftarrow \ldots$
$\vdots \quad \vdots$
Falls $siebfeld[j] < grenze$:
$trefferfeld[j] = 0;$

Durch die Festsetzung von $grenze$ gehen wir nicht auf die genauen Werte aller $(a_{unten} + j) + bm$ ein. Da $b > 0$ und m im Verhältnis zu den auftretenden a im allgemeinen sehr groß ist, verzichten wir durch diese Pauschalierung auf Nachkommastellen im Logarithmus, die bei oben erwähnter Verkürzung meistens sowieso nicht ins Gewicht fallen. Wir sparen uns aber sehr viele Rechnungen, selbst wenn wir jetzt eventuell bei der wirklichen Zerlegung ab und zu einen Kandidaten haben, bei dem diese mißlingt.

Für das Sieb von $a + b\rho$ müssen wir etwas mehr Aufwand treiben:
Die Basis C' wird analog zu eben erweitert.
Allerdings wird $siebfeld$ nicht mehr mit 1 vorbesetzt, sondern:

Für $i = 0$ bis $a_{lang} - 1$:
Falls $trefferfeld[i] = 1$;(∗ dies sind nach dem ersten Sieb nicht mehr viele ∗)
$siebfeld[i] = Log(|Norm((a_{unten} + i) + b\rho)|);$

jetzt ersetze die Zeilen

$$siebfeld[j] \leftarrow siebfeld[j] \cdot r_i;$$
durch
$$siebfeld[j] \leftarrow siebfeld[j] - s_i;$$

Die letzte Schleife muß dann so aussehen:
$grenze \leftarrow Log(MAXLARGE)$;
Für $j \leftarrow \ldots$
$\vdots \qquad \vdots$
Falls $siebfeld[j] > grenze$:
$trefferfeld[j] = 0$;

Das Arbeiten mit Logarithmen, auch bei der Zerlegung von $a + b\rho$, bringt uns die Lösung eines Problems, das ich bisher bewußt unterschlagen habe: Wie erkennt man die in der Zerlegung auftretenden Einheiten?
Wendet man den obigen Algorithmus auf $a + b\rho$ an, so erhält man meist:
$a + b\rho \neq \prod_{i=1}^{|C|} \alpha_i^{e_i} = \beta$ mit $Norm(a + b\rho) = Norm(\beta)$
d.h. $a + b\rho = \beta \cdot \gamma$ mit $Norm(\gamma) = 1$ also γ ist aus der Einheitengruppe von $\mathbb{Z}[\rho]$.
Da Divisionen in $\mathbb{Z}[\rho]$ im allgemeinen sehr umständlich und langsam sind, suchte ich einen Weg, die Einheiten γ, wenn schon nicht explizit zu berechnen, so doch wenigstens zu erkennen und vielleicht sogar zu spezifizieren.
Sei $\alpha = a_0 + a_1\rho + \cdots + a_{D-1}\rho^{D-1} \neq 0$
Da $\rho \in \mathbb{R}$, ist die Abbildung $\alpha \mapsto \log(|\alpha|)$ wohldefiniert und hat folgende Eigenschaften:

(i) $\log(|\alpha \cdot \beta|) = \log(|\alpha|) + \log(|\beta|)$;

(ii) $\log(|\alpha|) = 0 \;\Rightarrow\; \alpha = 1 \;\vee\; \alpha = -1$;

(iii) $\alpha \cdot \beta = \pm 1 \;\Rightarrow\; \log(|\alpha \cdot \beta|) = \log(|\alpha|) + \log(|\beta|) = 0 \;\Rightarrow$
$\beta = \pm\alpha^{-1} \;\Leftrightarrow\; \log(|\beta|) = -\log(|\alpha|)$;

Jetzt zerlegen wir $a + b\rho$ wie folgt:

Für $j = 0$ bis a_{lang}:
Falls $trefferfeld[j] = 1$:
$\vdots$
zerlege mit Probedivision $Norm(a_{unten} + j + b\rho) = (p_1^{e_1} \cdot \cdots \cdot p_{|B|}^{e_{|B|}}) \cdot p_L$
über $B \cup M = \{p_L \mid p_{|B|} < p_L \leq MAXLARGE\}$;
$LOG = \log(|(a_{unten} + j) + b\rho|)$;
Für $i = 1$ bis $|B|$:
Falls $e_i \neq 0$:
suche $(p_i = p_k, cp_k, \alpha_k)$ mit $(a_{unten} + j) + b\,cp_k \equiv 0 \; mod \; p_k$;
$LOG \leftarrow LOG - \log(|\alpha_k|)$;
speichere k, e_i;

$(*)$ suche (p_L, cp_L, α_L) mit $a_{unten} + j + b\,cp_L \equiv 0 \; mod \; p_L, \;\; p_L \in M$;
$(*)$ $LOG \leftarrow LOG - \log(|\alpha_L|)$;
$(*)$ speichere (p_L, cp_L);
speichere LOG; (* dies ist jetzt der Log der Einheit *)

Wie im weiteren mit den Einheiten verfahren wird, beschreibe ich in **gleichungen**.

Um das bei der Largeprime-Variante nötige Erweitern von **pcp** und **pimake** zu vermeiden, behandeln wir die Erzeuger von Primidealen mit großer Norm ebenso wie die Einheiten. Das heißt, wir versuchen nicht die zu einem p_L gehörenden cp und α zu finden, sondern begnügen uns damit, daß wir die Norm von α also p_L kennen und den Logarithmus, der bei der Zerlegung von $a + b\rho$ übrigbleibt. Dieser wird mit p_L abgespeichert. In der Zerlegung fallen also die Zeilen $(*)$ weg und **pcp** und **pimake** laufen wieder nur bis $p_{|B|}$.

3.3.2 Die Realisierung

- Hauptroutine: *siebphase();*
- allgemeine Unterroutinen: *make_RINGfeld(), make_wurzeln(), wurzelpot();*

Aus dem Eingabefile *ein_nfs* wird der Grad D des Polynoms f, dessen Koeffizienten, die Größe der Faktorbasis $|B|$, die Nullstellen von f , die Größe der größten Largprime $MAXLARGE$, die Anzahl der Intervalle, die Intervallänge, a_{unten} und der Name der Ausgabedatei für die Relationen gelesen. Jetzt wird Speicherplatz für die Felder $treffer$ und $lgsiebfeld$ reserviert. Anschließend werden von *wurzelpot* die Potenzen der Nullstellen mit komplexer Arithmetik berechnet und gespeichert; nun startet *siebphase.*

siebphase()

Parameter:
int $*$ f, D, B, MAXLARGE, INTERVALLANZ, INTERVALLAENGE, STARTSIEB;
double $***$ wurzeln;
char $*$ godot;

- spezielle Unterroutinen: *gcd(), sieben(), teilervon(), norm_abnull(), gvonxmodp()*[3], *polyvonnull()* [4] *;*
- allgemeine Unterroutinen: *make_intfeld(), make_intpointerfeld(), make_RINGfeld(), primeinlesen(), primideale(), wartenauf(), schreibfreigabe();*

Datenstrukturen In *int* *prim steht wieder die Faktorbasis B. Das zweidimensionale Feld *int* **pcp haben wir auch schon kennengelernt: (pcp[0][i],pcp[1][i]) sind die Paare (p_i, cp_i) und in pcp[2][i] steht jetzt der Logarithmus des korrespondierenden α_i.
Das Feld *int* **g_ enthält in g_[i] die Koeffizienten von α_i.
Die Felder *int* *plog, *primlog, *startmodp, *mmodp, *intermodp, *restmodp, *schrittweite werden in der Initialisierungsphase einmal vorberechnet um uns später das unnötige mehrfache Wiederholen von Rechnungen zu ersparen. Zum Beispiel wird hier das Feld für das Sieben über Primzahlpotenzen berechnet.
Die Länge von plog ist $laenge = |C|$. plog[i]= $Log(p_i)$ mit $(p_i, cp_i, \alpha_i) \in C'$. Dies benötigen wir für die Zerlegung von $a + b\rho$ über C.
Die Länge der anderen Felder ist $tiefe = |B'|$. Ihre Belegung wird in **Arbeitsweise** erklärt.
In *int* *b_teiler stehen nach Aufruf von *teilervon()* die Teiler des jeweils aktuellen b, wobei in b_teiler[0] die Anzahl der Teiler steht.
Nach *sieben()* stehen in *int* *zerl die Stellen im $treffer$-Feld, deren entsprechende Paare (a, b) alle Siebe erfolgreich durchlaufen haben. Wieder steht in zerl[0] die Anzahl dieser Stellen.
In *int* *zerl0, *zerl1 werden für diese Paare nacheinander die jeweiligen Zerlegungen aufgebaut, bevor sie in ein File gespeichert werden. zerl0[i] bzw. zerl1[i] enthalten den Exponenten des i-ten Elementes aus B bzw. C mit dem es an der Zerlegung beteiligt ist.

[3]siehe 4.1.2
[4]siehe 4.2.2

Arbeitsweise Da unser zu bearbeitendes Intervall $[a_{unten}, a_{oben}]$ sehr groß sein kann, wird dieses durch die Parameter INTERVALLAENGE und INTERVALLANZ in INTERVALLANZ Teilintervalle der Länge INTERVALLAENGE aufgeteilt. Der Bereich von a ist damit
$[a_{unten}$ =STARTSIEB,STARTSIEB+INTERVALLANZ·INTERVALLAENGE= $a_{oben}]$.
Die eben erwähnten Felder werden zuerst durch diese Schleife belegt:

$tiefe \leftarrow 0$;
Für $i = 0$ bis $|B|$:
    pot ← prim[i];
    LOG ← *Log*(prim[i]);
    tue primlog[*tiefe*]← *LOG*;
        schrittweite[*tiefe*]← *pot*;
        mmodp[*tiefe*]← *m mod pot*;
        startmodp[*tiefe*]←STARTSIEB *mod pot*;
        intermodp[*tiefe*]←INTERVALLAENGE *mod pot*;
        $tiefe \leftarrow tiefe + 1$;
        pot ← *pot*·prim[i];

    solange *pot* ≤INTERVALLAENGE;

Weiterhin haben wir für jedes Teilintervall eine unterschiedliche erste Stelle an der die einzelnen Primzahlen teilen. Diese errechnen wir, bevor wir ein neues Teilintervall bearbeiten, und speichern sie in restmodp.
runde $\in \{1 \cdots$INTERVALLANZ$\}$ gibt an über welchem Teil gerade gesiebt wird.
So lautet der weitere Algorithmus also:

Für $b = b_{unten}$ bis b_{oben}:

berechne die Teiler von b;

Für $runde = 1$ bis INTERVALLANZ:

Für $i = 0$ bis $tiefe$:

restmodp[i]= (startmodp[i]+intermodp[i]$\cdot$($runde$ − 1)+mmodp[i]$\cdot b)mod$ schrittweite[i];

$startwert$ =STARTSIEB+($runde - 1$)$\cdot$INTERVALLAENGE;

gcdsieb für Teilintervall $runde$;

erstes Sieb für Teilintervall $runde$;

zweites Sieb für Teilintervall $runde$;

(∗ In zerl stehen die Stellen, die wahrscheinlich zerlegt werden können ∗)

Für $k = 1$ bis zerl[0]:

j =zerl[k];

$test = Norm(startwert + j + b\rho)$;

$logwert2 = Log(startwert + j + b\rho)$;

zerlege $test$;

finde die zu den p_i und $startwert + j + b\rho$ gehörenden cp_i und α_i;

speichere die Potenzen in zerl1[i];

addiere die Logarithmen der α_i, also die pcp[2][i] in $logwert$;

$logwert \leftarrow logwert2 - logwert$;

Falls sich die Norm bis auf eine Largeprime zerlegen ließ:

speichere $(1, logwert)$; (∗ in $logwert$ steht der Logarithmus eines Elementes mit großer Norm ∗)

Falls sich die Norm ganz zerlegen ließ:

speichere $(0, logwert)$; (∗ in $logwert$ steht der Logarithmus einer Einheit ∗)

Falls die Norm sich nicht zerlegen ließ:

gehe zum nächsten k;

sonst: $test = startwert + j + bm$;

zerlege $test$;

speichere die Potenzen der p_i in zerl0[i];

Falls $test$ bis auf höchstens eine Largeprime zerlegt wurde:

speichere die Zerlegungen in der in Ausgabe beschriebenen Form ab.

(nächstes k)

(nächste $runde$)

(nächstes b)

sieben()

Parameter:
int b, mlog, maxlarge, **pcp, laenge, *rest, *schrittweite, tiefe, startwert, D, INTERVALLAENGE, *f;
short *primlog, *plog, *zerl;
double NS;

- spezielle Unterroutinen: *norm_abnull(), gvonxmodp();*
- allgemeine Unterroutinen: *make_Ringfeld(), setnullint(), setnullshort();*

sieben führt die beiden Siebe über dem Teilintervall [startwert...startwert+INTERVALLAENGE] nach dem oben beschriebenen Algorithmus mit Hilfe der eben berechneten Felder und Werte durch.

gcd()

Parameter:
int b, startwert, *b_teiler, INTERVALLAENGE;

- allgemeine Unterroutinen: *setxchar();*

Falls b < INTERVALLAENGE ist, wendet *gcd* das in Schritt 1. erwähnte Verfahren nur für [startwert...startwert+b] an. D.h. es belegt zuerst nur $treffer[0], \ldots, treffer[b-1]$ richtig. Dieser Teil wird jetzt einfach nach hinten kopiert,da sich ab hier die ggT's wiederholen, denn $\gcd(a,b) = \gcd(a+b,b)$. Also ist jetzt $treffer[0], \ldots, treffer[2b-1]$ richtig belegt. So kopiert die Prozedur immer die neu entstandenen Teile hintereinander, bis das gesammte $treffer$-Feld der Länge INTERVALLAENGE richtig belegt ist. Dies bringt vor allem für kleine b's einen erheblichen Zeitgewinn in dieser Prozedur.

teilervon()

Parameter:
int n, *prim, *teilerfeld;

Die Prozedur zerlegt mit Probedivision n über der Faktorbasis B und speichert die Teiler in teilerfeld ab. In teilerfeld[0] speichert sie die Anzahl der Teiler. Zahlen bis mindestens ${p_{|B|}}^2$ werden richtig zerlegt; dies reicht für unsere Wahl von b auch bei kleinen Faktorbasen aus.

norm_abnull()

Parameter:
int ∗abnull, ∗funktion, D;

Sei $\alpha \in \mathbb{Z}[\rho]$:

$$\begin{aligned}
\alpha &= a_{0,0} + a_{1,0}\rho + \cdots + a_{D-1,0}\rho^{D-1}\\
\alpha\rho &= a_{0,1} + a_{1,1}\rho + \cdots + a_{D-1,1}\rho^{D-1}\\
\vdots \quad & \quad \vdots \qquad\qquad \vdots\\
\alpha\rho^{D-1} &= a_{0,D-1} + a_{1,D-1}\rho + \cdots + a_{D-1,D-1}\rho^{D-1}
\end{aligned}$$

Es gilt:

$$Norm(\alpha) = Det \left(\quad a_{i,j} \quad \right)_{\substack{i=0..D-1\\ j=0..D-1}}$$

hieraus folgt für abnull= $a + b\rho$ und funktion= $f_0 + f_1 x + \cdots + f_D x^D$:

$$Norm(abnull) = a^D + \sum_{i=1}^{D} (-1)^i a^{D-i} b^i f_{D-i}$$

Nach dieser Formel berechnet die Prozedur die Norm.

3.4 gleichungen

3.4.1 Der Algorithmus

- **<u>Eingabe</u>**
  - Der Grad D des Polynoms;
  - Die Größe $s = |B|$ der Faktorbasis;
  - MAXLARGE;
  - Die Nullstellen der Funktion;
  - Die Faktorbasis B;
  - Die Größe $t = |C|$;
  - Die Trippel (p, cp, α);
- **<u>Ausgabe</u>**
  - Relationen der Form (1.1)

Wie wir uns erinnern, werden Relationen

$$\prod_{i=1}^{s} b_i^{e_{ij}} \equiv \prod_{l=1}^{t} c_l^{f_{lj}} \; mod \; n$$

gesucht, wobei $b_i \in B \wedge c_l \in C$ sind. Bisher haben wir aber erst solche gefunden, bei denen auf der linken Seite noch Largeprimes und auf der rechten $\varphi(\alpha)$, α hat große Norm und $\varphi(\mu)$, μ ist Einheit $\neq \pm 1$, vorkommen können. Diese störenden Faktoren wollen wir jetzt beseitigen.

<u>1. Fall:</u>
Wir haben eine Relation, in der nur links eine Largeprime vorkommt, d.h.

$$\prod_{i=1}^{s} b_i^{e_{ij}} \cdot p_L \equiv \prod_{l=1}^{t} c_l^{f_{lj}} \; mod \; n$$

Jetzt schauen wir in einer Liste (Large-Liste) nach, ob wir schon einmal eine Relation mit dem gleichen p_L gefunden hatten.
Ist dies <u>nicht</u> der Fall, wird die neue in die Large-Liste aufgenommen.
sonst :

$$\prod_{i=1}^{s} b_i^{e_{ik}} \cdot p_L \equiv \prod_{l=1}^{t} c_l^{f_{lk}} \; mod \; n$$

sei die erste, die mit dieser p_L gefunden wurde:
Wir dividieren jetzt die neue Relation durch die erste.
Dies dürfen wir unter folgender Voraussetzung tun:

$$\gcd(b_i, n) = \gcd(c_l, n) = 1 \ \ \forall\ b_i \in B \wedge c_l \in C \tag{3.3}$$

Um diese Bedingung kümmere ich mich erst beim Berechnen der Teiler; man behalte sie sich im Kopf, nehme sie hier aber erst einmal als erfüllt an.

$$\frac{\prod_{i=1}^{s} b_i^{e_{ij}} \cdot \cancel{p}_L}{\prod_{i=1}^{s} b_i^{e_{ik}} \cdot \cancel{p}_L} \equiv \frac{\prod_{l=1}^{t} c_l^{f_{lj}}}{\prod_{l=1}^{t} c_l^{f_{lk}}} \ mod\ n \ \Rightarrow$$

$$\prod_{i=1}^{s} b_i^{e_{ij}-e_{ik}} \equiv \prod_{l=1}^{t} c_l^{f_{lj}-f_{lk}} \ mod\ n$$

Dies ist eine Relation der gewünschten Form, die wir jetzt im Ausgabefile abspeichern.

2. Fall:

Wir haben eine Relation, in der nur rechts ein $\varphi(\alpha)$ vorkommt, mit $Norm(\alpha)$ ist eine Largeprime.

$$\prod_{i=1}^{s} b_i^{e_{ij}} \equiv \prod_{l=1}^{t} c_l^{f_{lj}} \cdot \varphi(\alpha) \ mod\ n$$

Wir verfahren im Prinzip wie eben, da wir aber weder das α noch das $\varphi(\alpha)$ kennen, vergleichen wir die Logarithmen der Elemente, die wir ja abgespeichert haben. Dadurch ergibt sich auch eine weitere Möglichkeit zwei Relationen zu kombinieren.

Falls schon einmal eine Relation mit $|Log(|\alpha|)| = |Log(|\beta|)|$ gefunden wurde:

Falls $Log(|\alpha|) = Log(|\beta|)$ ($\Rightarrow\ \alpha = \pm\beta$):

$$\frac{\prod_{i=1}^{s} b_i^{e_{ij}}}{\prod_{i=1}^{s} b_i^{e_{ik}}} \equiv \frac{\prod_{l=1}^{t} c_l^{f_{lj}} \cdot \cancel{\varphi}(\alpha)}{\pm \prod_{l=1}^{t} c_l^{f_{lk}} \cdot \cancel{\varphi}(\alpha)} \ mod\ n \ \Rightarrow$$

$$\prod_{i=1}^{s} b_i^{e_{ij}-e_{ik}} \equiv \pm \prod_{l=1}^{t} c_l^{f_{lj}-f_{lk}} \ mod\ n$$

sonst ($Log(|\alpha|) = -Log(|\beta|) \ \Rightarrow\ \pm\beta = \alpha^{-1} \ \Rightarrow\ \pm\varphi(\beta) \equiv \varphi(\alpha)^{-1}\ mod\ n$):

$$\prod_{i=1}^{s} b_i^{e_{ij}} \cdot \prod_{i=1}^{s} b_i^{e_{ik}} \equiv \prod_{l=1}^{t} c_l^{f_{lj}} \cdot \varphi(\alpha) \cdot \prod_{l=1}^{t} c_l^{f_{lk}} \cdot (\pm\varphi(\alpha)^{-1}) \ mod\ n \ \Rightarrow$$

$$\prod_{i=1}^{s} b_i^{e_{ij}+e_{ik}} \equiv \pm \prod_{l=1}^{t} c_l^{f_{lj}+f_{lk}} \ mod\ n$$

sonst: speichere die neue Relation in $\varphi(\alpha)$-Liste.

<u>3. Fall:</u>
Wir haben eine Relation, in der nur rechts ein $\varphi(\mu)$ vorkommt, mit μ Einheit. Hier gehen wir wie in Fall 2 vor und legen dabei $\varphi(\mu)$-Liste an. Falls allerdings $Log(\mu) = 0$ d.h. $\mu = \pm 1$, speichere die Relation sofort ins Ausgabefile.

Ein wenig anders müssen wir die gemischten Fälle behandeln:
Sei $a = \varphi(\alpha)$ bzw. $\varphi(\mu)$

$$\prod_{i=1}^{s} b_i^{e_{ij}} \cdot p_L \equiv \prod_{l=1}^{t} c_l^{f_{lj}} \cdot a$$

Zuerst schauen wir in der Large-Liste nach, ob schon eine Relation mit p_L gefunden wurde.
Ist dies der Fall, dividieren wir durch diese und erhalten :

$$\prod_{i=1}^{s} b_i^{e_{ij}-e_{ik}} \equiv \prod_{l=1}^{t} c_l^{f_{lj}-f_{lk}} \cdot a \mod n$$

Dies ist eine Relation für Fall 2 bzw. Fall 3
sonst, suchen wir in $\varphi(\alpha)$-Liste bzw. $\varphi(\mu)$-Liste, ob schon eine Relation mit a (eigentlich $Log(|\alpha|)$ bzw. $Log(|\mu|)$) gefunden wurde.
Ist dies der Fall, dividieren wir durch diese und erhalten:

$$\prod_{i=1}^{s} b_i^{e_{ij}\pm e_{ik}} \cdot p_L \equiv \prod_{l=1}^{t} c_l^{f_{lj}\pm f_{lk}} \mod n$$

Dies ist eine Relation für Fall 1.
Sonst (weder p_L noch a kamen bisher vor) speichere die Relation in eine Hilfsdatei und bearbeite sie später noch einmal.

<u>Die trivialen Relationen:</u>
Finden wir durch **pimake** für ein p, D Trippel (p, cp_i, α_i) $1 \leq i \leq D$, so bedeutet dies, daß $p \in \mathbb{Z}[\rho]$ in D Linearfaktoren aus $\mathbb{Z}[\rho]$ zerfällt und damit bis evtl. auf eine Einheit über unserer Faktorbasis C zerlegbar ist. D.h.:

$$p \equiv \prod_{i=1}^{D} \varphi(\alpha_i) \cdot \varphi(\mu) \mod n$$

ist eine Relation für Fall 3.

3.4.2 Die Realisierung

- Hauptroutine: *rel_komb();*
- allgemeine Unterroutinen: *make_RINGfeld;*

Die benötigten Eingaben werden wie immer aus *ein_nfs* gelesen und *rel_komb* wird aufgerufen.

rel_komb()

Parameter:
int D, B, MAXLARGE;
char *reldatei;
KOMPLEX **wurzeln;

- spezielle Unterroutinen:*polyvonnull()* [5] , *kombiniere(),rel_einlese();*
- allgemeine Unterroutinen: *make_intfeld(), make_intpointerfeld(), make_liste(), primeinlesen(), primideale();*

Datenstrukturen Belegung und Zweck der Felder *int* *prim, *zerl0, *zerl1, **pcp, **g_ wurden schon einmal erklärt.
int **leere-liste ist ein *NULL*-Zeiger, der eine leere Liste repräsentiert.
Die Felder *int* **einheits_liste, **lp_liste und **lpi_liste entsprechen den Listen: $\varphi(\mu)$-Liste, Large-Liste und $\varphi(\alpha)$-Liste. Um nicht die gesamten Relationen im Hauptspeicher zu haben, und trotzdem schnell suchen zu können, sind die implementierten Listen zweidimensionale Felder. Im Feld ..._liste[0], stehen die Largeprimes bzw. die Logarithmen der α und μ nach der Größe sortiert. In diesem Feld wird binär gesucht. In ..._liste[1][i], steht die Stelle, an der im entsprechenden Hilfsfile die gesamte Relation zu finden ist, die zu ..._liste[0][i] gehört. Dies bedeutet, daß wir einerseits keine sehr aufwendige Fileorganisation benötigen, sondern neue Relationen einfach an das bestehende File anhängen; wir aber andererseits die Datei nicht linear zu durchsuchen brauchen, sondern direkt an die richtige Stelle springen können.

Arbeitsweise Zuerst liest die Prozedur B und die Trippel (p, cp, α) ein. Dann untersucht sie diese auf triviale Relationen und ruft dementsprechend *kombiniere()* mit modus 0 oder 1 auf.
modus ist die Variable, die dem Unterprogramm *kombiniere* sagt, welcher der oben erwähnten Fälle auf die übergebene Relation zutrifft:
modus = 0: Relation hat Form (1.1), also nur abspeichern;
= 1: entspricht dem 3. Fall;
= 2: entspricht dem 1. Fall, allerdings i.a. mit $\varphi(\mu)$;
= 3: entspricht dem 2. Fall;
= 4: entspricht dem Fall: Largeprime und $\varphi(\alpha)$;

Nachdem die trivialen Relationen alle gespeichert wurden, liest die Prozedur nacheinander die von **sieb** gefundenen Relationen in zerl0 und zerl1 ein und übergibt sie unter Angabe des entsprechenden Modus und der passenden Listen *kombiniere()*.

[5] siehe 4.2.2

Laufzeit Für jede Relation, die **sieb** findet, durchsuchen wir eine der Listen binär und fügen entweder in diese Liste ein neues Element ein oder kombinieren zwei Relationen miteinander.
Sei $F = |B| + |C| + 1$
Unser Sieb muß $O(F)$ Relationen mit Largeprimes und Einheiten finden, damit wir daraus F Kongruenzen der Form (1.1) berechnen können. Die binäre Suche hat die Laufzeit $O(\log_2 F)$, wenn wir als schlechteste Listenlänge F annehmen. Auch für das Kombinieren zweier Relationen ist $O(\log_2 F)$ eine obere Abschätzung. Das einfügen in eine Liste läuft, da wir die Listen mit Arrays realisiert haben, linear zur Listenlänge. so kommen wir auf eine ungefähre obere Abschätzung von $O(F^2 \log_2 F)$ mit
$F = \frac{2L_n[1/3,(2/3)^{2/3}]}{((2/3)^{2/3}+o(1))(\log n)^{1/3}(\log\log n)^{2/3}}$

kombiniere()

Parameter:
int ∗zerl0, ∗zerl1, modus, ∗∗liste, ∗∗liste1, B, laenge;
FILE ∗fp, ∗pt;

- spezielle Unterroutinen: *speich_in_Mat(), inliste(), rel_sub(), rel_add;*

- allgemeine Unterroutinen: *binsuche();*

kombiniere arbeitet, je nach modus, die möglichen Fälle ab und verwaltet mit Hilfe von *inliste* die Listen. Ist modus = 0, sind beide Listen leer. Bei 1 oder 3 ist liste die einheits_ bzw. lpi_liste und liste1 die leere_liste. In den beiden anderen Fällen ist liste die lp_liste und liste1 die einheits_ bzw. lpi_liste. fp ist der Zeiger auf das zu liste gehörende File, ft, der auf das zu liste1 gehörende.

rel_einlesen()

Parameter:
int ∗zerl0, ∗zerl1, modus, ∗stelle, B, laenge;
char ∗reldatei;

- allgemeine Unterroutinen: *setnullint();*

Die Prozedur schaut nach, ob schon eine weitere Relation von **sieb** gefunden wurde. Ist dies der Fall, liest sie diese in zerl0 und zerl1 ein, berechnet den Modus der Relation und gibt 1 zurück. Sonst gibt sie 0 zurück.

inliste()

Parameter: *int* x, $**$liste, $*$zerl0, $*$zerl1, B, laenge, merkmal, genauwert;
FILE $*$fd;

- allgemeine Unterroutinen: *feldeinträge();*

Die Prozedur fügt x in die Liste liste an die richtige Stelle liste[0][i] ein, schreibt die Zerlegung ans Ende des Files fd und merkt sich in liste[1][i] diese Stelle. genauwert gibt den Bereich an, in dem zwei Logarithmen als gleich angesehen werden sollen. merkmal ist bei dem gemischten Fall 'Largeprime und $\varphi(\mu)$' der Logarithmus von μ.

rel_add()

Parameter:
int $*$zerl0, $*$zerl1, B, laenge, $*$merkmal;
FILE $*$fd;

Die Prozedur addiert zu den Exponenten der Zerlegung in zerl0 und zerl1 die Exponenten der Relation, auf deren Anfang der Filepointer von fd zeigt. Sie liefert in merkmal den Logarithmus einer Einheit, falls in der gelesenen Relation eine solche existiert.

rel_sub()

Parameter:
int $*$zerl0, $*$zerl1, B, laenge, $*$merkmal;
FILE $*$fd;

Die Prozedur subtrahiert von den Exponenten der Zerlegung in zerl0 und zerl1 die Exponenten der Relation, auf deren Anfang der Filepointer von fd zeigt. Sie liefert in merkmal den Logarithmus einer Einheit, falls in der gelesenen Relation eine solche existiert.

speich_in_Mat()

Parameter:
int ∗zerl0, ∗zerl1, B, laenge;

- allgemeine Unterroutinen: *feldeinträge();*

Die Prozedur speichert die Zerlegung ins Ausgabefile in folgender Form:
(∗ elemanz=Anzahl der von Null verschiedenen Einträge in zerl0 und zerl1; ∗)
–elemanz·2;
–0;
–zerl0:
-von Null verschiedener Eintrag zerl0[i];
-Stelle des Eintrags i;
–zerl1:
-von Null verschiedener Eintrag zerl1[i];
-Stelle des Eintrags i+B Größe der Faktorbasis B.

3.5 prglens

3.5.1 Der Algorithmus

- **Eingabe**
  - Grad D des Polynoms;
  - die Koeffizienten von f;
  - $|B|$ Größe der Faktorbasis B;
  - die Nullstellen des Polynoms;
  - MAXLARGE;
  - m;
  - die Faktorbasis B;
  - die Trippel (p, cp, α);
- **Ausgabe**
  - Teiler von n;

Hier wird zuerst einmal dafür gesorgt, daß die in **gleichungen** erwähnte Bedingung (3.3) erfüllt ist:

$$\prod_{i=1}^{s} b_i^{e_{ij}} \equiv \prod_{l=1}^{t} c_l^{f_{lj}} \; mod\, n \text{ und } n = v \cdot w \quad \Rightarrow$$

$$\prod_{i=1}^{s} b_i^{e_{ij}} \equiv \prod_{l=1}^{t} c_l^{f_{lj}} \quad \begin{matrix} mod\, v \\ mod\, w \end{matrix} \tag{3.4}$$

d.h. wenn ich alle Teiler, die in unserer Faktorbasis liegen aus n herausdividiere, sodaß:

$$v = \prod_{i=1}^{s} b_i^{e_i} \cdot \prod_{l=1}^{t} c_l^{f_l} \text{ mit } v \cdot w = n \text{ und}$$
$$\gcd(w, b_i) = \gcd(w, c_l) = 1 \;\; \forall\, b_i \in B \wedge c_l \in C$$

so gilt für w unsere Bedingung und unsere Relationen sind für w korrekt; außerdem ist v über den Faktorbasen leicht zu bestimmen.
Jetzt stellt sich natürlich die Frage, warum man das w nicht einfach vorher berechnet und darauf den gesamten Algorithmus anwendet.
Ein schwieriges Problem ist, nach eigener und auch der Erfahrung von anderen, ein geeignetes Polynom zu finden. Dazu kann es für eine bestimmte Zahl x nützlich sein, die

Zahl mit kleinen Faktoren zu multiplizieren.

Wir berechnen also w. Dann bringen wir das Ausgabefile von **gleichungen**: MATRIX in die von Programm *lenstra* benötigte Form und rufen dieses auf. *lenstra* ist wie schon erwähnt eine schnelle Gaußelimination. Bringt uns diese Lösungen, so berechnen wir hieraus Teiler von w. Diesen Algorithmus beschreibe ich in *teilerberech()*.

3.5.2 Die Realisierung

- Hauptroutinen: *lenstra, LGStest();*
- spezielle Unterroutinen: *phi(), n_durchkleineTeiler(), umspeichern();*
- allgemeine Unterroutinen: *make_intfeld(), make_intpointerfeld(),*
  make_RINGfeld(), primeinlesen(), primideale();

Zuerst werden die Eingaben aus *ein_nfs* gelesen. Jetzt werden Files gelöscht, die von einem vorangehenden Aufruf von *lenstra* noch existieren können; dies sind Hilfsdateien, die *lenstra* anlegt und auf die immer wieder zugegriffen wird. Die Faktorbasis B und die Trippel (p, cp, α) werden eingelesen. Ein Feld fiwerte von langen Zahlen wird hierzu korrespondierend alloziert, welches die eigentliche Faktorbasis C enthalten soll. Dessen Belegung erledigt die Prozedur *phi()*, die, mit Hilfe des Feldes g_, welches ja die α's enthält, $\varphi(\alpha)$ berechnet, für alle α, und in fiwerte abspeichert. Als nächstes wird n durch die Teiler, die in einer der Faktorbasen liegen, dividiert.

$(*)$ schaue, ob in MATRIX schon neue Relationen stehen;

> **Wenn ja** : speichere diese in das von *lenstra* benötigte Format um;
> rufe *lenstra* auf;
>
> **Falls** Lösungen gefunden wurden:
> berechne daraus Teiler von n (*LGStest()*);
>
> **sonst** : schlafe eine gewisse Zeit;
>
> gehe zu $(*)$;

lenstra

lenstra ist ein eigenständiges Programm, welches nur über Eingabefiles gesteuert wird. Möchte man, daß das Programm eine ganz neue Matrix bearbeitet, so muß man die oben erwähnten Hilfsfiles ALLSOLUTIONS, ELIMINATED, HOWFAR und RELATIONS löschen. In RESTART schreibt man die Größe der Faktorbasis; in unserem Fall also $|B|+|C|+1$. Die $+1$ wird gebraucht, da wir die (-1) in unserer Faktorbasis aufnehmen

müssen, weil wir sehr oft mit Beträgen gerechnet haben und dies zu einer Umkehrung des Vorzeichens geführt haben kann. Jede Negierung haben wir uns deshalb gemerkt.
Als weiteres Eingabefile benötigt *lenstra* NEWRELATIONS, in der die bisher gefundenen Relationen in einer bestimmten Form, die ich in *umspeichern()* erwähnen werde, stehen müssen. Wurden nach dem Abarbeiten dieser Kongruenzen neue gefunden, so werden die alten aus NEWRELATIONS gestrichen, die neuen hineingeschrieben und *lenstra* erneut aufgerufen. In den Hilfsdateien hat sich das Programm den derzeitigen Stand der Dinge gemerkt und bearbeitet jetzt die neuen Spalten als Erweiterung der alten Matrix.

LGStest()

Parameter:
FILE *fd;
int *lg_pos, *linksteiler, *rechtsteiler, B, laenge, *prim;
LZ *n, **fiwerte;

- spezielle Unterroutinen: *teilerberech();*
- allgemeine Unterroutinen: *setnullint();*

Die Lösung, die *LGStest* einliest, besteht aus den Nummern der Relationen, die der Reihenfolge ihres Auftreten in MATRIX entsprechen. Zwei Lösungen sind durch die Zahl 7654321 getrennt. Zu jeder Zahl wird jetzt die entsprechende Kongruenz gelesen und die Exponenten der Elemente der Faktorbasen werden in den Feldern linksteiler (für B) und rechtsteiler (für C) aufaddiert. Taucht die 7654321 auf, so wird *teilerberech()* aufgerufen und danach linksteiler und rechtsteiler wieder auf Null gesetzt.

phi()

Parameter:
RING *element;
int D;
LZ *m,*erg;

Die Prozedur berechnet φ(element) =element[0]+element[1]$\cdot$m+ $\ldots$ +element[D-1]$\cdot$m^{D-1} und gibt das Ergebnis in erg zurück.

n_durchkleineTeiler()

Parameter:
LZ *n, **fiwerte, *erg;
int *prim, B, laenge;

Die Prozedur berechnet zuerst:

$$\prod_{p \in B} p \cdot \prod_{\varphi(\alpha) \in C} \varphi(\alpha) \ mod\ n \text{ also:}$$

$hilf \leftarrow \prod_{i=0}^{B-1} \text{prim[i]} \cdot \prod_{j=0}^{laenge-1} \text{fiwerte[j] mod n};$
$ggt \leftarrow \text{gcd}(hilf\text{,n});$
$hilf \leftarrow ggt;$
solange $hilf \neq 1$:
    $n \leftarrow n/hilf;$
    $hilf \leftarrow \text{gcd}(ggt, n);$
    $ggt \leftarrow ggt \cdot hilf;$

Am Ende dieser Schleife steht in ggt das Produkt der kleinen Teiler von n, die in unseren Faktorbasen liegen und in n steht n/ggt.

umspeichern()

Parameter:
int ∗rel_pos;

Die Prozedur hat sich in rel_pos das Ende der Datei MATRIX bei ihrem letzten Aufruf gemerkt und schaut jetzt nach, ob inzwischen neue Kongruenzen gefunden wurden.
Ist dies der Fall:
    $i \leftarrow 0;$
    solange noch nicht das Fileende erreicht ist:
        lies nächste Relation ein;
        $i \leftarrow i + 1;$
        schreibe i in NEWRELATIONS;
        schreibe die eingelesene Kongruenz ohne die Vorzeichen in NEWRELATIONS;
        schreibe 0 in NEWRELATIONS;

    rel_pos ← neues Ende von MATRIX;

teilerberech()

Parameter:
int ∗zerl0, ∗zerl1, mp, laenge, ∗p;
LZ ∗n, ∗∗fiwerte;

- allgemeine Unterroutinen: *lange_make(), potenz();*

In zerl0 und zerl1 sind jetzt die Zerlegungen unserer Quadrate von x und y gespeichert; mp= $|B| + 1$ und laenge= $|C|$. In p steht die Faktorbasis B und in fiwerte C.

$$\prod_{i=0}^{mp-1} p[i]^{zerl0[i]} \equiv \prod_{j=0}^{laenge-1} fiwerte[j]^{zerl1[j]} \bmod n \text{ wobei } \begin{matrix} zerl0[i] \\ zerl1[j] \end{matrix} \text{ gerade}$$
$$0 \leq i \leq B$$
$$0 \leq j < laenge$$

allerdings kann es sein, daß einige zerl0[i] bzw. zerl1[j] kleiner als 0 sind. Deshalb berechnen wir:

$K = \{i \in \{0, \ldots, mp-1\} \mid zerl0[i] > 0\};$
$\bar{K} = \{i \in \{0, \ldots, mp-1\} \mid zerl0[i] < 0\};$
$L = \{j \in \{0, \ldots, laenge-1\} \mid zerl1[j] > 0\};$
$\bar{L} = \{j \in \{0, \ldots, laenge-1\} \mid zerl1[j] < 0\};$

$$x = \prod_{k \in K} p[k]^{zerl0[k]/2} \cdot \prod_{\bar{l} \in \bar{L}} fiwerte[\bar{l}]^{-zerl1[\bar{l}]/2};$$
$$y = \prod_{\bar{k} \in \bar{K}} p[\bar{k}]^{-zerl0[\bar{k}]/2} \cdot \prod_{l \in L} fiwerte[l]^{zerl1[l]/2};$$

$teiler1 \leftarrow \gcd(x-y, n);$
$teiler2 \leftarrow n/teiler1;$
$teiler1$ und $teiler2$ werden ausgegeben.

Kapitel 4

Allgemeine Routinen

In diesem Kapitel werden die Prozeduren beschrieben, die nicht direkt etwas mit dem Problem zu tun hatten.

4.1 Speicherplatzoperationen

int ***make_intfeld()***

Parameter:
int n;

Die Prozedur allociert Speicherplatz für ein *int*-Feld der Länge n und gibt als Funktionswert den Pointer auf die erste Speicherstelle zurück.

int ****make_intpointerfeld()***

Parameter:
int n;

Die Prozedur allociert Speicherplatz für ein Feld der Länge n von *int*-Pointern und gibt den Pointer auf die erste Speicherstelle zurück.

RING ***make_RINGfeld()***

Parameter:
int n;

Die Prozedur allociert Speicherplatz für ein Feld der Länge n von Werten des Datentyps *RING* und gibt den Pointer auf die erste Speicherstelle zurück.

LZ $*$**lange_make()**

Parameter:
int n;

Die Prozedur allociert Speicherplatz für eine lange Zahl. n ist im allgemeinen LDIM. Der Pointer auf die erste Speicherstelle der langen Zahl wird zurück gegeben.

KOMPLEX $***$**make_wurzeln()**

Parameter:
int D;

Die Prozedur allociert Speicherplatz für die Datenstruktur, in der die D Nullstellen unseres Polynoms und deren ersten D-1 Potenzen abgespeichert werden. Sie entspricht einem dreidimensionalen Feld der Größe D$\times$(D-1)$\times$2 mit Einträgen des Datentyps *KOMPLEX*. Zurück gegeben wird der Pointer auf die erste Speicherzelle der Datenstruktur.

int $**$**make_liste()**

keine Parameter.

Die Prozedur allociert ein zweidimensionales *int*-Feld der Größe 2$\times$LIST, das für unsere Suchlisten gebraucht wird.
liste[0][0] wird mit 0 vorbelegt; in ihr steht später immer die Nummer der letzten belegten Speicherstelle der Liste.
liste[1][0] wird mit LIST-1 vorbelegt und gibt im weiteren den letzten allocierten Speicherplatz der Liste an.
Der Pointer auf die erste Speicherstelle des Feldes wird zurückgegeben.

4.2 Speicherbelegung

setnullshort()

Parameter:
short $*$feld;
int laenge;

Die Prozedur besetzt das *short*-Feld feld der Länge laenge mit 0.

setnullint()

Parameter:
int $*$feld, laenge;

Die Prozedur besetzt das *int*-Feld feld der Länge laenge mit 0.

setxchar()

Parameter:
char *feld;
int laenge,x;

Die Prozedur besetzt das *char*-Feld feld der Länge laenge mit dem Character mit ASCII-Code x.

4.3 Einleseoperationen

int ***k_primzahlen()**

Parameter:
int k;

Die Prozedur liest aus PRIMDATEI die ersten k Primzahlen in ein Feld ein und gibt den Pointer auf dieses Feld zurück.

erstellePCPS()

Parameter:
int B;

- allgemeine Unterroutinen: *file_copy();*

Das erste Modul **pcp** kann ja, durch Änderung der Eingabedatei *ein_pcp*, mehrmals nebeneinander gestartet werden. Ist dies der Fall, so wird die Ausgabe eines jeden Starts in ein anderes File geschrieben. *erstellePCPS* linkt diese Files zusammen und erstellt die Datei, auf die **pimake** zugreift.

!!!!! VORSICHT !!!!!

Die Prozedur arbeitet nur richtig, wenn die **pcp**-Aufrufe alle im selben Directory schreiben und lesen!

int **file_copy()**

Parameter:
char *quelle, *ziel, *art;

Diese Prozedur stammt von *Stefan Düllmann.*
'Der Inhalt der Datei mit Namen quelle wird in die Datei mit Namen ziel kopiert (art='w') bzw. an die Datei mit Namen ziel angehängt (art='a').'

int ∗∗**pcp_einlese()**

Parameter:
int ∗anzahl, D, B;

- allgemeine Unterroutinen: *make_intfeld(), make_intpointerfeld();*

Die Prozedur liest aus dem Ausgabefile von **pcp** die Paare (p, cp) in ein zweidimensionales Feld und deren Anzahl in anzahl. Das Feld hat die Größe 2×anzahl. Der Pointer darauf wird als Funktionswert zurück gegeben.

int ∗**primeinlesen()**

Parameter:
int ∗anzahl,MAXLARGE;

- allgemeine Unterroutinen: *make_intfeld;*

Die Prozedur liest aus PRIMDATEI die Primzahlen ≤ MAXLARGE ein.

primideale()

Parameter:
int n, ∗∗g_, ∗p, ∗cp, ∗loga, D;

Die Prozedur liest aus PRIMIDEALE die ersten n Trippel (p, cp, α) in (p[i],cp[i],g_[i]) und den Log(g_[i]) in loga[i].

4.4 Operationen auf Elementen aus $\mathbb{Z}[\rho]$

ring_zuweisung()

Parameter:
RING ∗ringelement1, ∗ringelement2;
int D;

Die Prozedur weist ringelement1 die Einträge von ringelement2 zu. D ist die Länge der beiden Felder.

ring_neg()

Parameter:
RING *ringelement;
int D;

Die Prozedur negiert die Einträge von ringelement. D ist die Länge des Feldes.

int **ring_vgl()**

Parameter:
RING *ringelement1, *ringelement2;
int D;

Die Prozedur vergleicht ringelement1 und ringelement2, und gibt 1 zurück, falls alle Einträge gleich sind, 0 sonst.

ring_add()

Parameter:
RING *summand1, *summand2, *summe;
int faktor, D;

Die Prozedur multipliziert zuerst die Einträge von summand2 mit faktor und addiert diese dann komponentenweise mit den Einträgen von summand1. Das Ergebnis wird in summe zurückgegeben. Die Länge der Felder ist wieder D. Faktor macht eine spezielle Prozedur zum komponentenweisen Subtrahieren natürlich überflüssig.

multrho()

Parameter:
RING ∗ringelement, ∗produkt, ∗f;
int D;

Die Prozedur multipliziert ringelement mit ρ, reduziert das Ergebnis mit Hilfe von f und gibt es in produkt zurück. D ist die Länge der Felder.
Sei f= $f_0 + f_1 x + \ldots + f_{D-1} x^{D-1} + x^D \Rightarrow$
$(a_0 + a_1\rho + \ldots + a_{D-1}\rho^{D-1}) \cdot \rho =$
$a_0\rho + a_1\rho^2 + \ldots + a_{D-1}\rho^D =$
$-a_{D-1}f_0 + (a_0 - a_{D-1}f_1)\rho + \ldots + (a_{D-2} - a_{D-1}f_{D-1})\rho^{D-1}$

4.5 Operationen auf komplexen Zahlen

komplex_mult()

Parameter:
KOMPLEX ∗c1, ∗c2, ∗cerg;

Die Prozedur führt die komplexe Multiplikation auf den Zahlen c1, c2 durch und gibt das Ergebnis in cerg zurück.

komplex_add()

Parameter:
KOMPLEX ∗c1, ∗c2, ∗cerg;

Die Prozedur addiert die beiden komplexen Zahlen c1 und c2 und gibt das Ergebnis in cerg zurück.

double **reel()**

Parameter:
KOMPLEX ∗compl;

Als Funktionswert wird der Betrag der komlexen Zahl compl zurückgegeben.
$|compl| \leftarrow \sqrt{\Re(compl)^2 + \Im(compl)^2}$;

wurzelpot()

Parameter:
KOMPLEX $***$wurz;
int D;

wurz ist das dreidimensionale Feld, in das die Nullstellen des Polynoms f gelesen werden; genauer: wurz hat die Größe D$\times$(D-1)$\times$2 und in wurz[i][0] steht die i-te Nullstelle in komplexer Darstellung für 0$\leq$i<D.
Diese Prozedur hat jetzt die Aufgabe, das Feld ganz aufzufüllen; d.h. in wurz[i][j] die (j+1)-te Potenz der i-ten Nullstelle zu schreiben.

Für $i = 0$ bis $D - 1$:

Für $j = 0$ bis $D - 2$:
wurz[i][j] =wurz[i][$j-1$]$\cdot$wurz[i][0];

wobei die Multiplikation eine komplexe Multiplikation ist.

4.6 Allgemeine Operationen

umwandle()

Parameter:
double d;
LZ $*$l;
FILE $*$tmp;

Der *double*-Wert d, der keine Nachkommastellen besitzen darf, wird in eine lange Zahl verwandelt.

int **feldeintraege()**

Parameter:
int $*$feld, feldlaenge;

Die von Null verschiedenen Einträge von feld werden gezählt und die Anzahl als Funktionswert zurückgegeben.

int **binsuche()**

Parameter:
int x, $*$liste, listenlaenge, genauwert;

Die sortierte Liste liste wird mittels binärer Suche nach einem Wert aus [x$-$genauwert,x+genauwert] durchsucht. Wurde ein Eintrag aus diesem Intervall gefunden, so wird dessen Stelle in liste als Funktionswert zurückgegeben, sonst -1.

potenz()

Parameter:
LZ ∗bas, ∗erg, ∗n;
int exponent;

Dies ist eine schnelle Exponentiation langer Zahlen modulo n. Berechnet wird $erg = bas^{exponent} \quad mod\ n$ nach folgendem Prinzip:
stelle exponent binär dar: exponent= $e_1 e_2 \ldots e_x \quad e_i \in \{0, 1\}$;

Falls $e_1 = 1$:
erg ← bas;

sonst :
erg ← 1;

Für $i = 1$ bis x:
bas ← bas^2;

Falls $e_i = 1$:
erg ← erg·bas;

Hieraus sieht man: statt exponent lange Multiplikationen machen wir nur $2 \cdot \log_2(exponent)$ lange Multiplikationen.

4.7 Dateiverwaltungsoperationen

Da ich bei der Parallelisierung von **sieb** alle gefundenen Relationen der parallel laufenden Programme in eine Datei schreiben will, muß ich darauf achten, daß keine zwei Programme gleichzeitig schreiben. Dazu verwende ich das S_IEXEC Bit des Filestatus der entsprechenden Datei.
Normalerweise gibt dieses Bit an, ob das File ein ausführbares Programm enthält; dies ist natürlich bei Textdateien nicht der Fall. Das Bit ist also immer auf 0 gesetzt.
Sobald aber ein Programm in diese Datei schreiben will, setze ich das Bit auf 1. Dies ist für die anderen Programme ein Zeichen, daß sie das File jetzt nicht öffnen dürfen.

wartenauf()

Parameter:
char ∗filename;

Es wird überprüft, ob das S_IEXEC-Bit der Datei mit Namen filename gesetzt ist. Ist dies der Fall, so wird eine Warteschleife solange durchlaufen, bis das Bit wieder den Wert 0 hat.

Nun wird das Bit auf 1 gesetzt und die Prozedur endet
Dies bedeutet, daß das die Prozedur aufrufende Programm jetzt in die Datei schreiben kann.

schreibfreigabe()

Parameter:
char *filename;

Das S_IEXEC-Bit der Datei mit Namen filename wird auf 0 gesetzt.

Kapitel 5

Beispiele und Laufzeiten

5.1 Beispiele

Die folgenden Abschätzungen stammen aus der Abhandlung [**4**].

Sei $L_x[\nu, \gamma] = exp\,((\gamma + o(1))(\log x)^{\nu}(\log\log x)^{1-\nu})$;
Die heuristisch erwartete Zeit des *number field sieve* ist:
$L_n\left[1/3, 2(2/3)^{2/3}\right] \approx exp\left((1.526 + o(1))(\log n)^{1/3}(\log\log n)^{2/3}\right)$.

In dem Artikel sind auch Werte für die Größenordnug der Eingabedaten D, m, $max(B)$, a_{oben}, b_{oben} angegeben:
$D \approx \left(\frac{3\log n}{2\log\log n}\right)^{1/3}$;
$m \approx L_n\left[2/3, (2/3)^{1/3}\right]$;
$max(B) \approx a_{oben} \approx b_{oben} \approx L_n\left[1/3, (2/3)^{2/3}\right]$;
Diese Werte habe ich in den Beispielen in eckigen Klammern hinter die von mir gewählten Werte geschrieben.

Die Berechnungen wurden auf einer oder zwei *Sun 4/60 Sparc Station 1* durchgeführt.

5.1.1 Beispiel i)

$$n = 1331003 = 110^3 + 3$$

<u>Eingaben:</u>

ein_nfs: $D = 3$; [1.99]
$f(x) = x^3 + 3$;
$|B| = 25$; [12]
$\rho = \rho_0 = -1.4424957031$;
$\rho_1 = 0.721124785154 - 1.24902476648I$;
$\rho_2 = \bar{\rho}_1$;
MAXLARGE= 150;
INTERVALLANZ= 1;
INTERVALLAENGE= 50; [68]
$a_{unten} = -25$; [-34]
Ausgabe von **sieb**: RELATIONEN;
MAXHOEHE= 6;

ein_sieb: $b_{unten} = 1$;
$b_{oben} = 30$; [34]
Ausgabe der Laufzeit in 'aussieb'.

m: $m = 110$; [1156]

<u>Ergebnisse:</u>

pcp: 0.06 sec 23 Paare (p, cp);

pimake: 0.13 sec für die zugehörigen α.

sieb: gcd: —
Siebe: 0.22 sec
Zerlegung: 0.30 sec
———————————
Gesamt: 0.53 sec
Es wurden 116 Relationen gefunden.

gleichungen: kombinierte daraus 88 Relationen der Form (2.1).

prglens: Teiler: $257 \cdot 5179$

5.1.2 Beispiel ii)

$$n = 1006012013 = 1002^3 + 5;$$

Eingaben:

ein_nfs: $D = 3$; [2.17]
$f(x) = x^3 + 5$;
$|B| = 75$; [23]
$\rho = \rho_0 = -1.70997594668$;
$\rho_1 = 0.854987973338 - 1.48088260968I$;
$\rho_2 = \bar{\rho}_1$;
MAXLARGE= 600;
INTERVALLANZ= 1;
INTERVALLAENGE= 50; [162]
$a_{unten} = -25$; [−81]
Ausgabe von **sieb**: RELATIONEN;
MAXHOEHE= 6;

ein_sieb: $b_{unten} = 1$;
$b_{oben} = 50$; [81]
Ausgabe der Laufzeit in 'aussieb'.

m: $m = 1002$; [13914.7]

Ergebnisse:

pcp: 0.49 sec 74 Paare (p, cp);

pimake: 0.86 sec für die zugehörigen α.

sieb: gcd: —
Siebe: 0.47 sec
Zerlegung: 0.69 sec
————————
Gesamt: 1.21 sec
Es wurden 250 Relationen gefunden.

gleichungen: kombinierte daraus 186 Relationen der Form (2.1).

prglens: Teiler: $8731 \cdot 115223$

5.1.3 Beispiel iii)

$$n = 2326306724947 = 1235^4 - 3 \cdot 1235^2 - 3;$$

<u>Eingaben:</u>

ein_nfs: $D = 4$; [2.34]
$f(x) = x^4 - 3x^2 - 3$;
$|B| = 100$; [43]
$\rho = \rho_0 = 1.94712296671$;
$\rho_1 = -\rho_0$;
$\rho_2 = 0.889543617524I$;
$\rho_3 = \bar{\rho}_2$;
MAXLARGE= 1000;
INTERVALLANZ= 1;
INTERVALLAENGE= 200; [368]
$a_{unten} = -100$; [−184]
Ausgabe von **sieb**: RELATIONEN;
MAXHOEHE= 8;

ein_sieb: $b_{unten} = 1$;
$b_{oben} = 100$; [184]
Ausgabe der Laufzeit in 'aussieb'.

m: $m = 1235$; [196359]

<u>Ergebnisse:</u>

pcp: 1.95 sec 89 Paare (p, cp);

pimake: 1.04 sec für die zugehörigen α.

sieb: gcd: 0.01 sec
Siebe: 3.35 sec
Zerlegung: 1.40 sec
————————————
Gesamt: 5.04 sec
Es wurden 387 Relationen gefunden.

gleichungen: kombinierte daraus 208 Relationen der Form (2.1).

prglens: Teiler: 8741 · 266137367

5.1.4 Beispiel iv)

$$F7 = 2^{(2^7)} + 1 = 340282366920938463463374607431768211457$$
$$n = 2 \cdot F7 = 8796093022208^3 + 2$$

Eingaben:

ein_nfs: $D = 3$; [3.1]
$f(x) = x^3 + 2$;
$|B| = 500$; [1319]
$\rho = \rho_0 = 1.25992104989$;
$\rho_1 = 0.629960524947 - 1.09112363597I$;
$\rho_2 = \bar{\rho}1$;
MAXLARGE= 10000;
INTERVALLANZ= 1;
INTERVALLAENGE= 12001; [21706]
$a_{unten} = -6000$; [−10853]
Ausgabe von **sieb**: RELATIONEN;
MAXHOEHE= 6;

ein_sieb: $b_{unten} = 1$;
$b_{oben} = 2000$; [10853]
Ausgabe der Laufzeit in 'aussieb'.

m: $m = 8796093022208$; [$3.3 * 10^{12}$]

Ergebnisse:

pcp: 29.86 sec 479 Paare (p, cp);

pimake: 1.98 sec für die zugehörigen α.

sieb: gcd: 6.35 sec
Siebe: 263.38 sec
Zerlegung: 55.28 sec
———————————
Gesamt: 339.92 sec = 5 min 39.9 sec
Es wurden 2659 Relationen gefunden.

gleichungen: kombinierte daraus 1066 Relationen der Form (2.1).

prglens: Teiler: 114093784013702581094 42 · 59649589127497217
und 1192991785249944 34 · 5704689200685129054721

5.1.5 Beispiel v)

$$n = 751700570744044127192616748103510078478028806446946172523097520012 7$$
$$n = 52361396397820127^4 - 4 \cdot 52361396397820127^2 + 2$$

Eingaben:

ein_nfs: $D = 4$; [3.58]
$f(x) = x^4 - 4x^2 + 2$;
$|B| = 6000$; [15169]
$\rho = \rho_0 = -0.76536686473$;
$\rho_1 = -\rho_0$;
$\rho_2 = -1.84775906502$;
$\rho_3 = -\rho_2$;
MAXLARGE= 100000;
INTERVALLANZ= 3;
INTERVALLAENGE= 20000; [77273]
$a_{unten} = -30000$; [-165909]
Ausgabe von **sieb**: RELATIONEN;
MAXHOEHE= 8;

ein_sieb: $b_{unten} = 1$;
$b_{oben} = 20000$; [165909]
Ausgabe der Laufzeit in 'aussieb'.

m: $m = 52361396397820127$; [$4.8 * 10^{18}$]

Ergebnisse:

pcp: 14681.65 sec = 4 h 4 min 41 sec 5957 Paare (p, cp);

pimake: 58.47 sec für die zugehörigen α.

sieb: gcd: 506.34 sec
Siebe: 27825.22 sec
Zerlegung: 7694.04 sec
———————————
Gesamt: 42179.14 sec = 11 h 42 min 59 sec
Es wurden 27349 Relationen gefunden.

gleichungen: kombinierte daraus 17358 Relationen der Form (2.1).

prglens: Teiler:
186812208641 · 31683700840009422215077673792088546988180935197325696 1

Teiler :
$6237955583180309986727\,2961 \cdot 94885288183235521648625405876130411\,2641$

Teiler :
$5079180256659675431743744001 \cdot 1165326259898370899563555579802985\,6001$

Teiler :
$6237955583180309986727\,2961 \cdot 94885288183235521648625405876130411\,2641$

5.1.6 Beispiel vi)

$$n = (3^{29})^5 + 3$$

$$n = 1522586358169246802159262479225089070726226750574991661790882326344646$$

Eingaben:

ein_nfs: $D = 5$; [3.61]
$f(x) = x^5 + 3$;
$|B| = 6000$; [18048]
$\rho = \rho_0 = -1.24573093962$;
$\rho_1 = -0.384952203076 + 1.18476052767I$;
$\rho_2 = \bar{\rho}_1$;
$\rho_3 = 1.00781750057 - 0.73222227463I$;
$\rho_4 = \bar{\rho}_3$;
MAXLARGE= 100000;
INTERVALLANZ= 1;
INTERVALLAENGE= 50000; [401698]
$a_{unten} = -25000$; [−200849]
Ausgabe von **sieb**: RELATIONEN;
MAXHOEHE= 10;

ein_sieb: $b_{unten} = 1$;
$b_{oben} = 15000$; [200849]
Ausgabe der Laufzeit in 'aussieb'.

m: $m = 68630377364883$; $[1.4 * 10^{19}]$

Ergebnisse:

pcp: 4709.30 sec =1 h 18 min 29 sec 5881 Paare (p, cp);

pimake: 1173.90 sec für die zugehörigen α.

sieb: gcd: 247.74 sec
Siebe: 18929.30 sec
Zerlegung: 2688.42 sec
———————————
Gesamt: 23996.52 sec = 6 h 39 min 56 sec
Es wurden 13155 Relationen gefunden.
Das sind zu wenige um Teiler zu finden also ändern wir den Siebbereich.

b: 15001 bis 25000
a: -25000 bis 25000
gcd: 103.81 sec
Siebe: 6078.29 sec
Zerlegung: 306.36 sec
———————————
Gesamt: 7264.17 sec = 2 h 1 min 4 sec
Es wurden 1319 Relationen gefunden.

b: 20001 bis 25000
a: -15000 bis 15000
gcd: 64.21 sec
Siebe: 3519.82 sec
Zerlegung: 124.67 sec
———————————
Gesamt: 4113.32 sec = 1 h 8 min 33 sec
Es wurden 511 Relationen gefunden.

b: 1 bis 25000
a: 25001 bis 55000
gcd: 264.67 sec
Siebe: 18222.61 sec
Zerlegung: 532.66 sec
———————————
Gesamt: 20869.60 sec = 5 h 47 min 49 sec
Es wurden 2304 Relationen gefunden.

b: 1 bis 15000
a: -55001 bis -25001
gcd: 146.80 sec
Siebe: 11736.61 sec
Zerlegung: 365.47 sec
————————————
Gesamt: 13413.90 sec = 3 h 43 min 33 sec
Es wurden 1505 Relationen gefunden.

b: 25000 bis 32766
a: -15000 bis 15000
Für diese Siebgrenzen fehlen mir leider die Zeiten, aber aus den vorhergehenden kann man auf etwa 2 h schließen.
Hier wurden noch 589 Relationen gefunden.
Das macht insgesamt 19381 Relationen bei einer Gesamtzeit von etwa 21h 20 min.

gleichungen: kombinierte daraus 12047 Relationen der Form (2.1).

prglens: Teiler:
$35862819601960638726 \cdot 424558463352392466567042384407189420575011115653921$

Teiler :
$771888412770875135090278086 \cdot 19725472399612331297469869339049297206664961$

Teiler :
$6 \cdot 2537643930282078003598770798708481784543711250958319436318137210574 41$

Teiler :
$18777577903171610288815041 \cdot 8108534370197317577661985375045177511087184 06$

Teiler :
$129140166 \cdot 117901843038458445388653324111809572145526493327799475013132 81$

Teiler :
$24249395274935136790990517803680 6 \cdot 62788632083664176005952984256231926 41$

Teiler :
$3988322153843625493168 1 \cdot 381761126468156541508484939765727441502821177 66$

Teiler :
$1653015648161 \cdot 921096155298434489566437864963812232748478483309848634886$

Teiler :
$466957716721926 \cdot 32606514543927093824815422164856554689581431833367867 21$

Teiler :
$11118120874830726 \cdot 13694637567901313745664343202735011918452671208702192 1$

Teiler :
$9918093888966 \cdot 153516025883072414927739644160635372124746413884974772481$

Teiler :
$4041565879155856131831752967280 1 \cdot 37673179250198505603571790553739155846$

Teiler :
$81754966841140318493187568203841 \cdot 18623778065102934406848627578404737606$

Teiler :
$5977136600326773121 \cdot 2547350780114354799402254306443136523450066939235 26$

Teiler :
$1005049950874174991328 6 \cdot 1514936005762626655976983435108325074767896697 61$

Teiler :
$460806 \cdot 33041808443667113756315292752808971036102540995017245040014286 41$

5.2 Beurteilung

Schauen wir uns nun die Werte für die heuristisch erwarteten Laufzeiten l unserer Beispiele an:

Bsp. i) : $l = 1158.3$
Bsp. ii) : $l = 6517.6$
Bsp. iii) : $l = 33959$
Bsp. iv) : $l = 1.178 * 10^8$
Bsp. v) : $l = 2.753 * 10^{10}$
Bsp. vi) : $l = 4.034 * 10^{10}$

Wählen wir das letzte Beispiel als Bezugspunkt, d.h. wir nehmen an, daß die Laufzeit in diesem Fall nahe an die erwartete herankommt.
Dies ergibt folgenden Vergleich zwischen realer und theoretischer Zeit (es werden nur die Zeiten für die Siebe genommen):

Bsp.	reale Zeit	theoretische Zeit
vi)	64490 sec	64490 sec (dies ist unsere Annahme)
v)	27825 sec	44011 sec
iv)	263.38 sec	188.32 sec
iii)	3.35 sec	0.05 sec
ii)	0.47 sec	0.01 sec
i)	0.22 sec	0.002 sec

Für unsere ersten Beispiele ist der Vergleich sinnlos, da hier die $o(1)$ in der Laufzeitfunktion jeden Bezug unmöglich macht.
Je näher wir allerdings an unseren Bezugspunkt kommen, um so mehr Aussagekraft müßte der Vergleich haben.
Aus den beiden Werten des Beispiels v) sieht man, daß entweder die Eingabedaten des

Beispiels vi) sehr schlecht gewählt waren, oder die von v) sehr gut. Vergleicht man nun noch die von mir gewählten Eingaben des Beispiels vi) mit den theoretischen Abschätzungen hierfür, so sieht man, daß der Grad des Polynoms zu hoch und die Faktorbasis B zu klein gewählt war. Hier ließe sich die Laufzeit vielleicht noch etwas drücken.

Auffällig ist beim Beispiel v) die extrem hohe Zeit für **pcp**. Diese wird durch das Polynom $x^4 - 4x^2 + 2$ verursacht, da dieses für alle $p \in \mathbb{P} \setminus \{2\}$ entweder 4 Nullstellen modulo p oder keine Nullstelle modulo p hat. D.h., daß wir im ersten Fall sehr viele Werte aus $\{0, \ldots, p-1\}$ durchprobieren müssen. Hier würde sich wohl die erwähnte Verallgemeinerung lohnen.

Ich habe in meinen Beispielen die Laufzeiten von **gleichungen** und **prglens** nicht erwähnt, da beide Programme immer gleichzeitig zum Sieb liefen und ihre Arbeitszeit im Verhältnis zu diesem unwesentlich war.

Anhang A

Literaturverzeichnis

[1] **P. Domich, R. Kannan, L. Trotter**
Hermite normal form computation using modulo determinant arithmetic, Mathematics of operations research Vol.12, Februar 1987

[2] **D. Knuth**
The art of computer programming vol.2, second edition, Addison-Wesley

[3] **A.K. Lenstra, H.W. Lenstra Jr.**
Algorithms in number theorie, J.van Leeuwen's Hanbook of theoretical computer science, North-Holland, Amsterdam

[4] **A.K. Lenstra, H.W. Lenstra Jr., M.S. Manasse, J.M. Pollard**
The number field sieve, Symposium on Theory of Computing 1990 S.564 ff.

[5] **Pohst & Zassenhaus**
Algorithmic algebraic number theorie, Encyclopedia of mathematics and it,s applications, Cambridge University Press

[6] **J.M. Pollard**
Factoring with cubic integers, Manuskript

[7] **I.N. Stewart, D.O. Tall**
Algebraic number theory, second edition, Chapman and Hall

[8] **Jörg Zayer**
Die Theorie des Number-Field-Sieve, Diplomarbeit an der Universität des Saarlandes 1991