Kapitel 15
Zahlendarstellungen

Das Thema des letzten Kapitels – ASN.1–kodierte Datenströme – wird meist in Verbindung mit „gesicherter“ Datenübertragung benötigt, wobei mit „gesichert“ nicht nur „verschlüsselt“ gemeint ist. Die Sicherung von Daten ist in vielen Fällen mit der Handhabung großer ganzer Zahlen verbunden, wobei „groß“ für „nicht systemunterstützt“ steht.[1] Grund genug, sich einmal mit der Software für solche Zahlen zu beschäftigen. Da neben den ganze Zahlen in der Technik die reellen Zahlen (*die komplexen Zahlen betrachte ich mal als Ableger der reellen, obwohl das technisch natürlich nicht ganz so trivial ist*) eine noch größere Rolle spielen, nehmen wir diese beziehungsweise ihre Pendants auf den Rechnern gleich mit in unsere Betrachtungen auf.

Für die Durchführung von Berechnungen stellt ein Rechnersystem in der Regel zwei Zahlenklassen zur Verfügung:

- Ganzzahlige Typen unterschiedlicher Breite, typischerweise mit 8, 16 oder 32 (*gegebenenfalls auch 64*) Bit mit oder ohne Vorzeichen
- Fließkommazahlen mit 32, 64, 80 oder 128 Bit Breite.

Ganzzahlige Datentypen erlauben grundsätzlich exaktes fehlerfreies Rechnen, während Fließkommazahlen dies meist nicht erlauben, und das nicht nur, weil reelle Werte mit einer endlichen Länge dargestellt werden müssen, sondern aus viel elementareren Gründen, wie wir sehen werden.

---

[1] Einige Java–Spezies werden an dieser Stelle möglicherweise widersprechen, da Java doch Zahlen beliebiger Länge bereitstellt (was dann obendrein noch als Argument verwendet wird, Java als die technisch einzig sinnvolle Sprache darzustellen, da nur sie alles beinhaltet). Dem kann ich zwei Sachen entgegenhalten: (A) Eine sehr große Zahl von derzeitigen und zukünftigen Bibliotheksfunktionen existiert nur, weil einige der verspotteten „Dummköpfe“ die entsprechenden Anwendungen in C oder etwas ähnlichem programmiert und in das Java–System eingebunden haben. (B) Wie ich in den letzten Kapiteln zeigen werde, sind viele Rechenergebnisse unbrauchbar, wenn der Anwender die Technik einfach nur nutzt, ohne ihre Eigenschaften zu verstehen. Gerade da muss man einigen Java–Argumentatoren entgegenhalten, dass sie nicht verstanden haben, worum es eigentlich geht. In dem Sinne werden Sie das Rechnen mit Zahlen nach den nächsten beiden Kapiteln vermutlich etwas anders betrachten.

G. Brands, *Das C++ Kompendium*, eXamen.press, 2nd ed.,
DOI 10.1007/978-3-642-04787-9_15, © Springer-Verlag Berlin Heidelberg 2010

Neben diesen vom System angebotenen Datentypen werden aber oft weitere benötigt:

- Lange ganze Zahlen mit mehreren tausend Bit Breite werden in Verschlüsselungsalgorithmen benötigt.
- Für exakte Rechnungen mit nicht ganzzahligen Werten können unter bestimmten Randbedingungen echte rationale Datentypen eingesetzt werden, die aus zwei ganzen Zahlen bestehen (*Zähler und Nenner eines echten Bruchs*).
  Die Randbedingungen sind dabei schnell aufzuzählen: Die Ausgangsdaten müssen echte rationale Zahlen sein, und während der Rechnung dürfen nur Funktionen zum Einsatz kommen, deren Funktionswerte ebenfalls rational sind. Damit scheiden Wurzeln, trigonometrische Funktionen, Logarithmen usw. aus. Die Einsatzfelder solcher Zahlen sind also recht beschränkt.
- Längere Fließkommazahlen erlauben die Verkleinerung von Rechenfehlern. Sie sind sinnvoll einsetzbar,
  - wenn die Rechenfehler mit den Standard–Fließkommazahlen die erforderliche Genauigkeit übersteigen (*zum Thema „Rechenfehler“ werden wir im nächsten Kapitel umfangreiche Untersuchungen anstellen*) und gleichzeitig der Rechenaufwand begrenzt bleibt, weil beim Übergang von mäßig genauen hardwareunterstützten Zahlenmodellen zu hochgenauen softwaregetriebenen Modellen natürlich mit größeren Zeiteinbußen zu rechnen ist;
  - wenn durch Intervallrechnung der Vertrauensbereich von Ergebnissen geprüft oder Ergebnisse auf ein grundsätzlich andere Art genauer ermittelt werden sollen (*siehe Kap. 13*);
  - wenn spezielle Rechenalgorithmen für die Verrechnung ganzer Zahlen hochgenaue Fließkommazwischendaten erfordern.
- Spezielle mehrkomponentige Typen, beispielsweise komplexe Zahlen, Quaternionen, Polynome usw.

Für die Lösung einer gestellten Aufgabe genügt es nicht, einen (*anscheinend*) passenden Standarddatentyp auszuwählen und ein so berechnetes Ergebnis dem Auftraggeber in die Hand zu drücken. Jeder einigermaßen seriöse Auftraggeber wird verlangen, dass gewissen Garantien für die Qualität des Ergebnisses eingehalten werden; eine Aufgabe, die man nur dann erfüllen kann, wenn die Eigenschaften der Zahlensysteme bekannt sind.

Bei den Eigenschaften der Zahlentypen kann zwischen algebraischen (*durch die mathematische Klassifizierung festgelegten*) und systemabhängigen Eigenschaften differenziert werden. Die systemabhängigen Eigenschaften werden durch die spezielle Implementation bestimmt, und zur Ermittlung der Eigenschaften zur Laufzeit stellt die Standardbibliothek eine Vorlagenklasse zur Verfügung, die eine Tabelle der wichtigsten Kenndaten enthält und vom Programmierer neuer Datentypen zu spezialisieren ist:

```
template <class T> numeric_limits { ... };
```

Sie enthält beispielsweise Angaben über den Wertebereich oder die Genauigkeit eines Datentyps. Der größte darstellbare Wert des Typs `int` lässt sich beispielsweise durch die Abfrage

```
numeric_limits<int>::max()
```

ermitteln. Für eigene Erweiterungen implementieren wir die Klasse

```
template <class T>
class algebra: public numeric_limits<T> { ... };
```

Die Grundprinzipien wurden bereits in Kap. 6.2.3 dargestellt, so dass wir uns hier auf weitere spezielle Fälle und Details konzentrieren können.

15.1 Ganze Zahlen

Software für den Umgang mit langen ganzen Zahlen ist Voraussetzung für viele heutige Verschlüsselungssysteme. Allerdings sollte diese Feststellung nicht überstrapaziert werden: In den meisten Sicherheitsprotokollen dienen Verfahren auf der Basis langer ganzer Zahlen nur für die Initialisierung, während der verschlüsselte Datenverkehr während des Großteils einer Sitzung mit schnellen anderen Verfahren abgewickelt wird. Natürlich ist höchste Effizienz auf diesem Gebiet nur mit Assembler–Treibern zu erreichen, aber auch gute Routinen in Hochsprache zeigen schon beeindruckende Leistungen.

Wir werden im weiteren für jede der vier Grundrechenarten bis zu drei Algorithmen präsentieren. Ein Grund hierfür ist, dass in verschiedenen Programmierumgebungen nicht unbedingt alle Rechenoptionen zur Verfügung stehen, ein anderer Grund ist der Aufwand, den man bei einer Implementation bereit ist zu erbringen. Das lässt sich am Thema „Assembler–Programmierung“ gut verdeutlichen, trifft aber genauso auf die komplexeren vorgestellten Algorithmen zu:

Assembler–Routinen sind meist in der Lage, die Operationen erheblich schneller auszuführen, da weniger Befehle zum Einsatz kommen und das Hin- und Herschieben von Daten zwischen der CPU und dem Speicher vermindert wird. Der Effizienzgewinn hat aber einen Haken: Zunächst einmal sind zusätzlich gute Hardwarekenntnisse über die verwendete Maschine notwendig. Welche Register bietet der Prozessor an, welche Eigenschaften haben sie und wie werden sie bedient? Mit einer Assembler–Routine lässt man sich damit zunächst auf einen bestimmten Maschinentyp ein und die entwickelten Routinen müssen auf einem anderen Maschinentyp grundsätzlich neu überdacht werden.[2]

[2] In den meisten Fällen wird vermutlich eine PC-ähnliche Maschine mit einem Intel-Pentium-Prozessor oder einem kompatiblen Prozessor zum Einsatz kommen, was die Auswahl natürlich einschränkt. Andere Maschinen mit RISC-Prozessoren sind unter Umständen anders anzufassen.

Mit der Festlegung auf einen bestimmten Maschinentyp ist aber noch nicht alles erledigt. Der Kern der Implementierung liegt zwar nun fest, aber die Routine soll ja in eine Hochsprachenumgebung und ein Betriebssystem eingebunden werden. Zusätzlich ist daher noch zu klären, wie die Schnittstellendefinitionen des verwendeten Systems aussehen (*das heißt wo und wie man sich die Adressen der Variablen abholen kann*) und in welcher Form das Entwicklungssystem die Einbindung von Assembler–Befehlen erlaubt.

Fazit. Assembler–Implementationen müssen auf das jeweilige System angepasst werden, was schlimmstenfalls ebenso viele verschiedene Implementation bedeutet, wie Systeme mit der Software bedient werden sollen. Da damit immer noch nicht auszuschließen ist, dass ein Einsatz auf einem noch nicht getesteten System erfolgen soll,[3] bilden die Hochsprachenversionen einen Methodenkern, auf den man sich im Zweifelsfall immer zurückziehen kann.[4]

15.1.1 Basisalgorithmen

Für die Implementation langer ganzer Zahlen legen wir folgendes fest:

(a) Die Zahlen werden ziffernweise auf Feldern eines ganzzahligen Grunddatentyps gespeichert. Die Felder haben jeweils die Kapazität n Bit, auf jedem Feld wird eine Ziffer der Zahl zur Basis

$$B = 2^m \quad , \quad m \leq n$$

gespeichert. Um das nicht so trocken stehen zu lassen, hier ein Beispiel: Denken Sie an die Darstellung einer Zahl in hexadezimaler Schreibweise, also mit Hilfe der Ziffern `0..F` . Wenn der Datentyp `char` verwendet werden, gilt $m = 4$, $sn = 8$. Wir nutzen also nur die Hälfte der Bitkapazität für die Speicherung einer Ziffer. Wollen wir die volle Kapazität nutzen, so benötigen wir 256 Ziffern, beispielsweise `0x00..0xFF`.

(b) Gespeichert werden grundsätzlich positive Werte. Eine negative Zahl wird als positiver Betrag und Vorzeichen gespeichert.

(c) Die Länge der Zahlen ist offen, die Darstellung selbst ist längennormiert, das heißt

$$laenge = k \quad \Leftrightarrow \quad z[k] \neq 0, \forall\, r > k : [r] = 0$$

---

[3] Das kann beispielsweise schon eine weiterentwickelte Version eines getesteten System sein.

[4] In diesem Zusammenhang werden dem Leser sicher auch die „Test suites“ hochperformanter Bibliotheken in den Sinn kommen, die nach Erstellen der Systembibliothek Tests gegen bekannte Ergebnisse fahren. Kommt nicht das Erwartete heraus, so sind entweder Assembler-Routinen in der Umgebung nicht brauchbar oder, noch schlimmer, die verwendeten „Sprachstandarts“ gelten auf der untersuchten Maschine nicht.

Die Festlegungen (b) und (c) ergänzen einander: bei der Darstellung negativer Zahlen durch ein Äquivalent zum Zweierkomplement der Binärdarstellung müsste die Darstellungslänge auf einen bestimmten Wert festgelegt werden (*und alle negativen Zahlen hätten genau die maximale Darstellungslänge*). Für den Grunddatentyp bleiben verschiedene Möglichkeiten offen, die in den Algorithmen berücksichtigt werden müssen:

- Der Grunddatentyp kann vorzeichenbehaftet oder vorzeichenlos sein.
- Die Basis kann mit $n = m$ oder $m < n$ imlpementiert werden.
  Im ersten Fall sind alle Bits eines Datenwortes vollständig genutzt, die Zahl wird also so kompakt wie möglich gespeichert. Für diesen Speichertyp sind nur vorzeichenlose Grunddatentypen zulässig, die Algorithmen können unter Umständen nicht in jedem Fall implementiert werden.[5]
  Im zweiten Fall hat mehr als eine Ziffer auf einem Datenspeicher platz. Diese Reserve kann für die Rechenalgorithmen genutzt werden.

Wir werden im weiteren für die Rechnung nur vorzeichenlose Datentypen verwenden. Für die Speicherung einer Zahl definieren wir die Datenstruktur

```
struct b_feld{
    uchar * buf;        // unsigned char
    int     len;
    int     reserved;
    bool    negativ;
    ...
};//end struct
```

Der Pufferbereich wird jeweils in bestimmten Inkrementen vergrößert, sobald er für die Aufnahme einer Zahl unzureichend ist, das heißt bei der Abspeicherung eines Ergebnisses der Fall *len* > *reserved* eintritt. Für die Rechnung mit ganzen Zahlen definieren wir die Klasse `GanzeZahl`, die ein Referenzobjekt auf einen solchen Datenpuffer hält. Das Prinzip ist in Kap. 8.5 ausführlich diskutiert worden und kann hier übernommen werden. Wir halten lediglich den Grundaufbau einer solchen Klasse noch einmal fest:

```
class GanzeZahl {
public:
    GanzeZahl();
    GanzeZahl(const GanzeZahl& gz);
    GanzeZahl& operator=(const GanzeZahl& gz);
```

[5]Diese Bemerkung trifft nicht auf C-Implementationen zu, die immer die notwendige Freiheit gewähren. In Pascal oder Fortran ist das jedoch nicht unbedingt so, da die Sprachsyntax weiter von der Maschinenebene entfernt ist als die von C und Überlaufsicherungen einschließt.

```
    GanzeZahl& operator=(int gz);
    GanzeZahl& operator+=(const GanzeZahl& gz);
    GanzeZahl  operator+(const GanzeZahl& gz) const;
    ...
    bool operator==(const GanzeZahl& gz) const;
    bool operator< (const GanzeZahl& gz) const;
    ...
    GanzeZahl operator-()const;
    ...
protected:
    b_feld * data;
};//end class
```

Das interne Datenobjekt `b_feld * data` wird mit einer Referenzzählung gemäß Kap. 9.3 verwaltet. Auf Details der Implementation der Klasse werden wir im weiteren nicht eingehen, sondern uns auf die reinen Rechenalgorithmen beschränken, da die Prinzipien bereits mehrfach zum Einsatz gekommen sind. Denken Sie bei der Implementation daran, dass Sie viele Operatoren mit bestimmten Kombinationen von Operanden mit Hilfe einfacher `inline`- Funktionen auf wenige Grundfunktionen zurückführen können (*im Kap. 4.1 finden Sie die Vereinbarungen für Vergleichsoperatoren*).

> **Aufgabe.** Stellen Sie die komplette Schnittstelle der Klasse bereit. Rechnungen sollten zwischen zwei großen Zahlen oder zwischen einer Maschinen–Integer und einer großen Zahl (*in beliebiger Reihenfolge*) möglich sein. Die Methoden können auch weitgehend als `inline` deklariert werden, um Rechnungen mit kleinen Zahlen zu beschleunigen.

Die Rechenalgorithmen erfordern auch jeweils einige Vorbereitungen. Sehen wir uns dazu die Aufgabe $a + b$ an. Für die Fälle

$$(a \geq 0 \wedge b \leq 0) \vee (a \leq 0 \wedge b \leq 0)$$

handelt es sich um eine Addition und die Aufgabe ist klar. Haben die Zahlen jedoch unterschiedliches Vorzeichen, so müssen wir eine Subtraktion durchführen, wobei die größere Zahl von der kleineren abgezogen werden muss. Dazu verwenden wir zweckmäßigerweise die implementierte normale Subtraktionsmethode. Bei der müssen wir aber ähnliche Prüfungen durchführen und eventuell Sogar die Additionsmethode aufrufen. Das darf natürlich nicht zu einem zyklischen Verhalten führen.

> **Aufgabe.** Stellen Sie die Regeln für die Addition/Subtraktion mit unterschiedlichen Vorzeichen zusammen und entwerfen Sie Algorithmen dazu.

Die Methoden müssen natürlich robust gegen Eigenbezüge sein, das heißt bei der Implementation müssen wir sicherstellen, dass auch Anweisungen wie

$$a \circ = a \ , \quad \circ := +, -, ^*, /, \ldots$$

korrekt bearbeitet werden.

Nach diesen allgemeinen Vorbemerkungen zu Konstruktion der Klasse kommen wir nun zu den Algorithmen.0Wenig Kommentierung bedürfen folgende Operationen:

- Arithmetische UND– oder ODER–Operationen (*Ausführung Wort für Wort*),
- Schiebeoperationen (*nach Links schieben beginnt beim höchsten Wort, nach Rechts schieben beim ersten Wort*),
- Vergleiche (*bei gleicher Länge und gleichem Vorzeichen beginnen beim höchsten Wort*).

In einem früheren Kapitel haben wir bereits allgemein nutzbare Vorlagenklassen eingeführt, die solche Operationen durchführen können. Noch fehlende Methoden können bei Bedarf ergänzt werden. Für die meisten Zwecke wird eine Anpassung der Methoden `length(..)` und `data(..)` genügen; für Schiebeoperationen ist je nach Schieberichtung eine Vor- oder Nachbearbeitung der Länge notwendig. Linksschieben innerhalb einer Methode kann damit folgendermaßen realisiert werden:

```
zref = RefAllocator<GanzeZahl>::
                          allocator.Copy(zref);
zref->Resize(neue_laenge);
bit_shl(anzahl_bits);

// mit

int length(const GanzeZahl& gz){
    return gz.zref->len;
}//end function

char * data(const GanzeZahl& gz){
    return gz.zref->buf;
}//end function
```

Für die Ein- und Ausgabe werden Algorithmen für die Umwandlung in Strings benötigt. Trivial ist hier die Umwandlung in das Hexadezimalformat (*beginnen beim höchsten Wort*) oder das Lesen vom Hexadezimalformat (*um vier Bit nach Links schieben und die gewandelte nächste Ziffer an die unterste Position schreiben*). Ähnlich lässt sich auch das Einlesen von einem Dezimalstring erledigen: Das Lesen erfolgt von Links nach Rechts. Vor jedem Lesen wird das Zwischenergebnis mit Zehn multipliziert und anschließend die in einen Binärwert gewandelte Ziffer addiert.

```
template <>
bool  fromString(GanzeZahl& gz, string s){
```

```
    ...
    neg = s.find("-") == 0;
    if (s[0]=='+' || s[0]=='-')
        s.erase(0,1);
    string help = "0123456789";
    for (i=0 ; i<s.length() ; i++) {
        ziffer= help.find(s.substr(i,1));
        if (ziffer!=-1){
            gz *= 10;
            gz +=ziffer;
        } else {
            gz=0;
            return false;
        };//endif
    }; /* endfor */
    if(neg)
        gz=-gz;
    return true;
}//end function
```

Für die Umwandlung einer Zahl in einen Ziffernstring lässt sich leicht der Algorithmus ermitteln, wenn der Ziffernstring als Polynom formuliert wird:

$$z_n \ldots z_2 z_1 z_0 \quad \Leftrightarrow \quad z_n * 10^n + z_{n-1} * 10^{n-1} + \ldots + z_1 * 10 + z_0$$

Offenbar erhält man $z_0$als Divisionsrest bei Teilen der Zahl durch 10, z_1 als Divisionsrest bei Teilen des ganzzahligen Divisionsergebnisses der Zahl mit 10 durch 10 usw.:

```
template <>
string toString(GanzeZahl const& gz){
    ...
    do{
        ch=(char)Abs((gz%10).Long());
        ch+='0';
        s=ch+s;
        gz/=10;
    }while(b_not_equal(gz,0));
    if(is_neg)
        s="-"+s;
    return s;
};//end function
```

15.1.1.1 Additionsalgorithmen

Die Ein-/Ausgabeoperationen lassen sich also mit Hilfe der Rechenalgorithmen erledigen, die wir nun genauer untersuchen[6] (*das kann auch negativ formuliert werden: Um Ein- und Ausgaben längerer Zahlen machen zu können, müssen die Rechenroutinen funktionieren. Das hat einigen Einfluss auf die Testsystematik, da verschiedene Methoden gleichzeitig getestet werden*). Die technisch gesehen einfachste Rechenoperation ist die Addition zweier Zahlen, womit die echte vorzeichenunabhängige Addition der Inhalte der Datenpuffer gemeint. Eine echte Addition findet bei

```
(a.negativ && b.negativ) ||
(!a.negativ && !b.negativ)
```

statt, das heißt wenn beide Operanden das gleiche Vorzeichen besitzen. Das Vorzeichen des Ergebnisses ändert sich bei der Operation nicht. Wir können nun nach Speichertyp unterscheiden:

Ziffern mit Reserve

Bei einer Addition können Ziffern–Überläufe auftreten, die auf die nächste Position zu übertragen sind. Das maximale Ergebnis einer Ziffernaddition mit davor stattgefundenem Überlauf bei der Addition zweier Zahlen ist

$$(B-1)+(B-1)+1 < 2*B$$

Es kann also nur eine Eins als Übertrag auftreten. Werden r Zahlen gleichzeitig addiert, so kann für jeden Summanden ein Übertrag auftreten, sofern die Anzahl der Summanden kleiner als die Basis bleibt. Zwischen Basis, Anzahl der Summanden und Reserve bei der Zifferndarstellung gilt dann

$$B < \frac{INT\_MAX+1}{r}$$

und wir erhalten den Algorithmus (*Speicherung auf der Variablen „res“, Summanden in den Variablen z_1..z_r von Zahlen, Basis B*)

```
s=0;
for(i=0;i<res.len;++i){
    res.buf[i]=s+z_1.buf[i]+z_2.buf[i]+ ... + z_r.buf[i];
    s=res.buf[i]/B;
    res.buf[i]%=B;
```

[6] Im weiteren Fallen kaum konkrete Aufgaben an, da ansonsten meiner Meinung nach die Diskussion doch zu sehr zerrissen wird. Arbeiten Sie den Code deshalb bitte ohne ausdrückliche Aufforderung parallel zur Theorie/zu den Beispielen auf.

```
}//endfor
if(s!=0){
   res.buf[i]=s;
   ++res.len;
}//endif
```

Ziffern ohne Reserve

Ohne Reserve ist natürlich auch kein Speicherplatz für den Übertrag vorhanden (*wir setzen die Verwendung des größten zur Verfügung stehenden Datentyps voraus*). Vorausgesetzt, das System reagiert nicht ungehalten durch eine Programmabbruch auf einen Überlauf und wir verwenden vorzeichenlose Grunddatentypen, ist das Vorliegen eines Übertrags aber leicht festzustellen. Einschränken müssen wir die Anzahl der Überläufe auf einen möglichen beschränken, weshalb die gleichzeitige Addition von mehr als zwei Zahlen nicht mehr möglich ist.

Für die Durchführung verwenden wir soweit möglich den größten zur Verfügung stehenden ganzzahligen Standarddatentyp ohne Vorzeichen. Ist die Länge der Datenblöcke beispielsweise 26 Byte, so können sechs Additionen mit dem Datentyp `ulong` durchgeführt werden, zwei weitere mit dem Datentyp `uchar` schließen die Aktion ab.[7] Einschließlich Übertragskontrolle berechnen wir (*der Ergebnispuffer* `a.buf`*ist mit den Werten eines Operanden vorinitialisiert, was der Operation* **+=***entspricht.Der Ergebnispuffer muss mindestens die Länge des Puffers des zweiten Operanden aufweisen. Dazu wird gegebenenfalls der Überhang vorab kopiert, da die Addition der einzelnen Worte der Puffer nur bis zur kleineren der beiden Längen durchgeführt wird.*)

```
for(i=0,ov=0;i<b.len;++i){
    ov=(temp=b.buf[i]+ov)<ov;
    ov=((a.buf[i]+=temp)<temp)||ov;
}//endfor
```

Addieren wir nämlich zwei vorzeichenlose ganze Zahlen fester Breite und findet dabei ein Überlauf statt, so ist das Ergebnis kleiner als die beiden Summanden, weil das überlaufende Bit fortfällt (*zweite Rechenzeile ohne das logische ODER*). Hat in der davor liegenden Operation bereits ein Überlauf stattgefunden, so müssen wir das Überlaufbit dazu addieren, wobei bei dieser Operation natürlich auch ein Überlauf stattfinden kann und das Ergebnis anschließend Null ist (*erste Rechenzeile*). Für die korrekte Berücksichtigung beider möglicher Überläufe sind zwei Vergleiche notwendig (*vergewissern Sie sich durch konstruierte Beispiele, dass nichts eingespart werden kann*). Die Zusammenfassung aus beiden Operationen (*logisches ODER der zweiten Rechenzeile*) ist der Vortrag für die nächste Addition, wobei insgesamt

[7] Man kann natürlich immer mit der Breite *ulong* arbeiten und gegebenenfalls einige Nullbytes am Zahlenende in Kauf nehmen. Die byteweise Vorgehensweise hängt mit den Algorithmen für die Multiplikation zusammen.

maximal ein Bit überlaufen kann. Bleibt am Ende ein Überlaufbit übrig, so müssen weitere Bytes im Ergebnis inkrementiert werden:

```
while(ov)
    ov=++a.buf[i++]==0;
```

Gegebenenfalls ist die verwendete Länge des Datenpuffers unter Auffüllen mit Nullwerten anzupassen. In ähnlicher Weise sind die Inkrementoperatoren implementierbar.

Assembler–Implementierung

Zumindest bei einer Operation sei einmal der Vergleich mit einer Assemblerprogrammierung durchgeführt, um Effizienzvergleiche anstellen zu können Bei einer Assembler–Imlementation ist ebenfalls $m = n$, das Vorzeichen spielt keine Rolle, da die CPU sich nicht besonders darum kümmert, aber mit dem Carry–Bit eine automatische Überlaufkennung bereitstellt, die komplexere Prüfungen obsolet mach. Mit einem hypothetischen Assembler können wir die Addition so formulieren:

```
        mov     R1,[z1]     // Adresse z1
        mov     R2,[z2]
        mov     R3,l1       // Anzahl worte
        clc                 // Carry löschen
loop:   mov     A,[R1]      // Wort 1 laden
        adc     A,[R2]      // Wort 2 mit Carry
                            // addieren
        mov     [R1],A      // speichern
        inc     R1          // Zählregister
                            // bearbeiten
        inc     R2
        dec     R3
        jnz     loop
        adc     A,0         // überzähliges Carry
                            // sichern
        mov     [R1],A
```

Das entspricht der oben notierten Hochsprachenversion des Algorithmus, das heißt die Kontrolle der Puffergrößen, gegebenenfalls die Erweiterung und Kopieren/Auffüllen mit Werten kommt natürlich noch separat hinzu. Im Vergleich zum reinen Assemblerkode liefert die Hochsprachenversion den Code (*nicht optimiert*):

```
for(i=0;i<ll;i++){
   mov          dword ptr [ebp-14 h],0
   jmp          _add+3Fh (00448d7f)
   mov          eax,dword ptr [ebp-14 h]
   add          eax,1
   mov          dword ptr [ebp-14 h],eax
```

```
   mov          ecx,dword ptr [ebp-14 h]
   cmp          ecx,dword ptr [ebp-18 h]
   jae          _add+0BCh (00448dfc)

ov=(temp=ul1[i]+ov)<ov;
   mov          edx,dword ptr [ebp-1Ch]
   and          edx,0FFh
   mov          eax,dword ptr [ebp-14 h]
   mov          ecx,dword ptr [ebp-8]
   mov          eax,dword ptr [ecx+eax*4]
   add          eax,edx
   mov          dword ptr [ebp-0Ch],eax
   mov          ecx,dword ptr [ebp-1Ch]
   and          ecx,0FFh
   cmp          dword ptr [ebp-0Ch],ecx
   sbb          edx,edx
   neg          edx
   mov          byte ptr [ebp-1Ch],dl

ov=((uld[i]+=temp)<temp)||ov;
   mov          eax,dword ptr [ebp-14 h]
   mov          ecx,dword ptr [ebp-4]
   mov          edx,dword ptr [ecx+eax*4]
   add          edx,dword ptr [ebp-0Ch]
   mov          eax,dword ptr [ebp-14 h]
   mov          ecx,dword ptr [ebp-4]
   mov          dword ptr [ecx+eax*4],edx
   mov          edx,dword ptr [ebp-14 h]
   mov          eax,dword ptr [ebp-4]
   mov          ecx,dword ptr [eax+edx*4]
   cmp          ecx,dword ptr [ebp-0Ch]
   jb           _add+0AAh (00448dea)
   mov          edx,dword ptr [ebp-1Ch]
   and          edx,0FFh
   test         edx,edx
   jne          _add+0AAh (00448dea)
   mov          dword ptr [ebp-20 h],0
   jmp          _add+0B1h (00448df1)
   mov          dword ptr [ebp-20 h],1
   mov          al,byte ptr [ebp-20 h]
   mov          byte ptr [ebp-1Ch],al

}//endfor
   jmp          _add+36 h (00448d76)
```

So kompakt der Hochsprachenkode auch aussehen mag – im Vergleich mit einer Assemblersequenz ist er sehr teuer, aber dafür eben universell einsetzbar. Ähnliches

gilt natürlich auch für die Vorbereitung oder die anderen Algorithmen, die in Assembler ebenfalls wesentlich kompakter werden.

15.1.1.2 Subtraktionsalgorithmen

Die Implementierung der Subtraktion (*womit eine echte Subtraktion gemeint ist, also zum Beispiel auch die Addition einer positiven und einer negativen Zahl*) erfolgt ähnlich.

Ziffern mit Reserve

Hier können wir wieder mehrere Zahlen simultan bearbeiten, falls der Grunddatentyp vorzeichenbehaftet ist (*andernfalls können wir aber mit einer temporären Variablen gleicher Wortbreite rechnen*).

```
s=0;
for(i=0;i<a.len;++i){
    a.buf[i]-=(b.buf[i]+s);
    if(a.buf[i]<0){
        a.buf[i]+=B;
        s=1;
    }//endif
}//endfor
```

Der Algorithmus setzt in dieser Form $a > b$ voraus, das heißt die Zahlen sind vor Durchführen der Subtraktion zu sortieren. Bei unterschiedlichen Längen ist dies kein Problem, bei gleichen Längen kann der Prüfaufwand aber schlimmstenfalls genauso aufwendig sein wie die Subtraktion selbst.

Ziffern ohne Reserve

Hierbei gilt es diesmal, einen Unterlauf zu erkennen. Ohne vorhergehendes Unterlaufbit ist das Ergebnis größer ist als der Ausgangswert. Das Unterlaufbit der vorhergehenden Operation berücksichtigen wir separat:

```
for(i=0,ov=false;i<b.len;i++){
    ovh=(temp=a.buf[i]-ov)>a.buf[i];
    ov=((a.buf[i]=temp-b.buf[i])>temp)||ovh;
}//endfor
```

Ist `a.len>b.len` (*wir stellen vor Durchführung des Algorithmus sicher, dass zumindest bei ungleichen Pufferlängen diese Sortierung vorliegt*) und bleibt am Ende der Subtraktionsfolge ein Unterlauf übrig, so müssen die weiteren Datenworte des Ergebnisses dekrementiert werden, bis kein Unterlauf mehr eingetreten ist (*vergleiche Inkrementieren*). Bei gleich langen Zahlen ist das natürlich nur dann möglich, wenn die betragskleinere von der betragsgrößeren Zahl abgezogen wird, also eine

Vorsortierung erfolgt, wie wir sie bereits angesprochen haben. Erfolgt keine Vorsortierung, so kann nach Abschluss der Subtraktion eine negative Zahlenkodierung vorliegen, das heißt es bleibt ein nicht beseitigtes Unterlaufbit übrig. Es nützt in diesem Fall nichts, die Zahlenlänge zu vergrößern, da jede vorne angefügte Ziffer den Wert $(B-1)$ (*das heißt 0xFF für Bytes*) aufweist. Eine negative Zahl kann nur durch Bildung des Zweierkomplements wieder in eine positive Zahl überführen werden. Auf Byteebene lautet der Algorithmus:

```
for(i=0;i<b.len;++i)
    a.buf[i] ^= 0xFF;
i=0;
do
    ov=++a.buf[i++]==0;
while(ov);
```

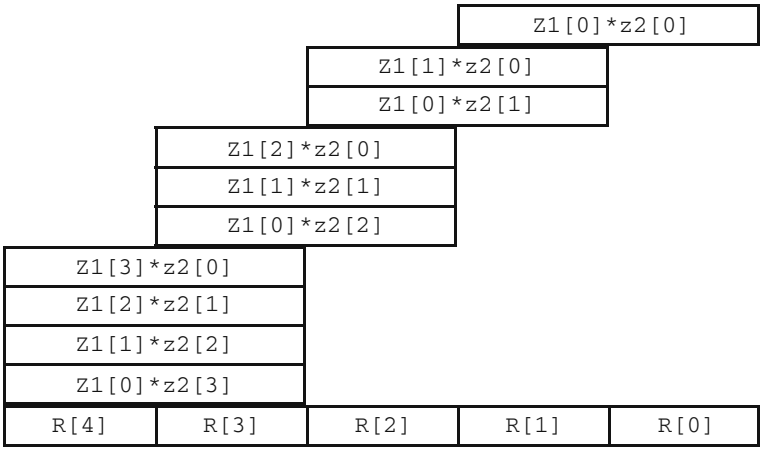
Aufgabe. Welche der beiden Implementierungen – Vorsortierung oder Zweierkomplementbildung – ist vorzuziehen?

Assembler–Implementierung

Den Code erhält man aus der Addition durch Austausch des Befehls `adc` durch `sbc`.

15.1.1.3 Multiplikationsalgorithmen

Addition und Subtraktion sind hinsichtlich des Rechenaufwands bei steigender Zahlengröße unkritische Anwendungen, da der Aufwand nur mit der Ordnung $O(n)$ wächst. Kritischer ist der Aufwand bei der Multiplikation, die wir zunächst „schulmäßig“ mit dem Aufwand $O(n^2)$ erledigen, wobei jede Ziffer der einen Zahl mit jeder Ziffer der anderen Zahl multipliziert wird und die Ergebnisse addiert werden.



Dabei ist jedes Produkt zweier Ziffern maximale zwei Ziffern groß. Das folgende Schema gibt einen Überblick über die entstehenden Faktoren und ihre Addition. Die Puffergröße des Ergebnisses, die bei der Addition/Subtraktion den Wert ($res.len \leq max(a.len,b.len)+1$) aufweist, liegt nun bei dem Wert ($res.len = a.len+b.len$), und Rechnungen ohne Hilfspuffer, wie sie bei **a+=b** möglich sind, sind nun ebenfalls nicht mehr durchführbar.

Ziffern mit Reserve

Hier kann man durch geschickte Zusammenfassung der Ziffernprodukte alle für eine Ergebnisposition auftretenden Überträge in einem Durchlauf mit berechnen (*der Gesamtaufwand $O(n^2)$ bleibt davon allerdings unberührt*). Anstatt die Schleifenvariablen beider Zahlen unabhängig laufen zu lassen, berechnen wir jeweils alle Summanden für eine Ergebnisposition und übernehmen den Überlauf in die nächste Position. Dazu muss allerdings ein genügend großer Datentyp für die Zwischenergebnisse zur Verfügung stehen. Wie Sie anhand des Schemas nachvollziehen können, kann einschließlich des Übertrags ein Zwischenergebnis die Größe

$$r * (B-1)^2 + (B-1) = r * B^2 - (r-1) * B + (r-1)$$

erreichen, wobei $r = min(a.len,b.len)$ die Anzahl der Summanden pro Ergebnisziffer ist. Bei 32–Bit Grunddatentypen stehen somit im Prinzip nur 8–Bit–Ziffern zur Verfügung, um ausreichende Reserven für die Addition zu haben. Der Algorithmus besitzt zwei Schleifen. Die äußere durchläuft alle Ziffernpositionen des Ergebnisses, die innere umfasst alle zu dieser Position vorhandenen Ziffern–Positionssummen der Faktoren:

```
s=0;
for(i=0;i<a.len+b.len;++){
    temp=0;
    for(k=max(0,i-b.len);k<=max(i,a.len);++k)
        temp=temp+a.buf[k]*b.buf[i-k];
    temp=temp+s;
    res[i]=temp%B;
    s=temp/B;
}//endfor
```

Ziffern ohne Reserve

Hier verzichten wir auf die Sortierung und bilden Produkte mit vorzeichenlosen Grunddatentypen der halben maximalen Breite. Die entstehenden Produkte der maximalen Breite werden auf die entsprechenden Ergebnispositionen addiert und die Überträge fortgeschrieben. Das folgende Implementationsbeispiel verwendet Zeiger anstelle der Feldvariablen mit Indizes (*vergleiche die Anmerkungen in Kap. 5.1*)

```
ushort *u1, *u2, *ue1, *ue2, *ud, *udd;
ulong *ll,temp;
bool ov;

ud=(ushort*)zd;
u1=(ushort*)z1;
u2=(ushort*)z2;
ue1=&u1[(l1)/sizeof(ushort)];
ue2=&u2[(l2)/sizeof(ushort)];

for(;u1!=ue1;++u1,++ud){
    if(*u1!=0){
        for(u2=(ushort*)z2,udd=ud;u2!=ue2;++u2,++udd){
            if(*u2!=0){
                temp=(ulong)*u1 * (ulong)*u2;
                ll=(ulong*)udd;
                ov=(*ll+=temp)<temp;
                while(ov)
                    ov=++*++ll==0;
            }//endif
        }//endfor
    }//endif
}//endfor
```

Assembler–Implementierung

Diese kann auf eine ähnliche Weise erstellt werden. Auf einem n–Bit–Prozessor benötigen wir dazu zunächst zwei ($n/2$)–Bitregister für die Faktoren und ein n–Bitregister für das Ergebnis. In den meisten Prozessoren ist es möglich, zwei Halbregister für die Multiplikation zusammen zu schalten, so dass nur ein Register belegt wird. Dazu kommen drei Adressregister und zwei Zählregister, also insgesamt sechs Register. Der C–Code für „Ziffern ohne Reserve“ kann für den Assembler–Code übernommen werden, wobei die Inkrement–Schleife für den Übertrag auf höhere Ziffernpositionen des Ergebnisses durch Zwischensichern der Zieladresse auf dem Stack realisiert werden kann:

```
    push    res             // res = Register mit der
lp: add     res,WORDLEN     // Zieladresse
    inc    [res]
    jnc    lp
    pop    res
```

Quadratbildung

Es ist sinnvoll, die Quadratbildung$a = a * a$ bei der Multiplikation als Spezialfall zu behandeln. Zunächst einmal taucht sie nämlich sehr häufig in Anwendungen auf:

In der Verschlüsselungstechnik müssen höhere Potenzen großer Zahlen ermittelt werden, worunter man sich etwa folgendes vorstellen kann.

$$872.93829.73239.84785.73813^{3349.83283.89798.27238.94894.72834}$$

Die Zahlen sind zwar noch eine gute Größenordnung unter dem, mit was man es wirklich zu tun hat, aber es ist wohl unzweifelhaft, das das 3349.83283.89798. 27238.94894.72834–malige Muliplizieren der Zahl 872.93829.73239.84785.73813 mit sich selbst wenig Aussicht auf Erfolg hat.[8] Durch fortgesetzte Quadrierung lässt sich die Aufgabe allerdings problemlos lösen:

```
// Berechne:  result = a ^ b;
result=1;
sq=a;
for(i=0;i<bit_high(a);i++){
    if( (bit_test(a,i))
        result*=sq;
    sq*=sq;
}//endfor
```

Die Schleife läuft über die Anzahl der Bits der Zahl `b` . In jedem Schritt wird quadriert, so dass `sq` die Werte

$$a,\ a^2,\ a^4,\ a^8, \ldots$$

annimmt. Ist das k-te Bit in `b` gesetzt, wird die aufgelaufene Zweierpotenz zum Ergebnis multipliziert. Falls Ihnen die Methode noch nicht bekennt gewesen sein sollte, verifizieren Sie, dass der Exponent auf diese Weise als Polynom in Potenzen von Zwei rekonstruiert wird und jede beliebige Potenz mit dem Aufwand $O(ld(2))$an Multiplikationen zu ermitteln ist.

Zurück zum Quadrieren: Hier taucht im Produkt jedes Ziffernprodukt

$$z_i * z_k \ ,\ i \neq k$$

aufgrund der Symmetrie zweimal auf. In den Algorithmen kann daher etwa die Hälfte der Schleifendurchläufe eingespart werden, symbolisch:

```
for(i=0;i<len;i++){
    sq[2*i]+=b[i]*b[i];
    for(j=0;j<i;j++)
        sq[i+j]+=2*b[i]*b[j];
}//endfor
```

[8] Auch die Darstellung in den gebräuchlichen 3er-Schritten ist sinnlos, da hierfür keine Begriffe wie „super-Mega-Giga-Protz-...“ vorhanden sind. Zum Abzählen sind dann 5er- oder 10er-Unterteilungen sinnvoller.

Aufgabe. Dabei ist jedoch Vorsicht geboten, zumindest bei Ziffernspeicherung ohne Reserve! Der Term `b[i]*b[j]` nimmt bereits die ganze zur Verfügung stehende Datenwortbreite ein, kann also nicht noch mit Zwei multipliziert werden (*was einem Schieben nach Links um eine Position entspricht*). Bei der zweimaligen Addition zum Ergebnis können aber auch zwei Überläufe anstelle eines Überlaufs eintreten. Implementieren Sie einen Algorithmus, der dies berücksichtigt.

15.1.1.4 Divisionsalgorithmen

Die Division ist mit Abstand die aufwendigste Operation. Sie ist stets eine Division mit Rest, das heißt die Ausführung liefert gleichzeitig den Faktor und den Divisionsrest:

$$b = q * a + r \quad , \quad 0 \leq |r| < |a|$$

Die Definition ist so noch nicht ganz eindeutig. Beispielsweise können wir

$$-17 = -3 * 5 + (-2) \quad \textit{oder} \quad -17 = -4 * 5 + 3$$

als Lösung im Negativen ansehen. Im Rahmen einer Modulrechnung sind beide Lösungen äquivalent, für die Praxis muss aber klar sein, was die Rechenfunktion liefert. Mathematisch gilt die Konvention: Das Produkt ist die betragsmäßig größte Zahl, die kleiner als der Betrag des Dividenden bleibt, der Rest ist der betragsmäßig kleinste mögliche Rest (*das heißt die Rechenmethoden liefern die linke Lösung*).

Ziffern mit Reserve

Die Division führen wir ähnlich einer Schuldivision durch: wir dividieren die zwei höchsten Ziffern des Dividenden durch die höchste Ziffer des Divisors und erhalten ein bis zwei Ziffern des Ergebnisses an der dem Abstand der Ziffernpositionen (Dividend–Divisor) entsprechenden Stelle. Das Ergebnis wird zu den vorhandenen Ziffern addiert. Der Divisor wird mit den Ergebnisziffern multipliziert und das Produkt vom Dividenden abgezogen. Je nach Strategie kann die Ziffer des Divisors auch erhöht werden, um negative Zwischenergebnisse zu vermeiden. Ein einfacher Algorithmus unter Ausnutzung der bereits entwickelten Methoden ist

```
// a.len >= 2 !
res=0;
while(a>b){
    ls=(a.buf[a.len-1]*B+
                a.buf[a.len-2])/b.buf[b.len-1];
    pos=a.len-b.len-2;
    sa=0;
```

```
    sa[p]=ls%B;
    sa[p+1]=ls/B;
    res=res+sa;
    a=a-b*sa;
}//endwhile
```

Ziffern ohne Reserve

Hier kommt ein vergleichbarer Algorithmus zum Einsatz: aus dem Dividenden werden so viele Bits verwendet, dass der größte ganzzahlige Datentyp des Systems gefüllt ist; für den Divisor wird der zweitgrößte Datentyp verwendet.

```
ulong *ul; ushort * us, divisor;
us=(ushort*)&b.buf[b.len-sizeof(ushort)];
divisor=*us+1;

ul=(ulong*)&a.buf[a.len-sizeof(ulong)];
while(a>b){
```

Das Prinzip lässt sich am besten an einem Beispiel verstehen: Auf das Dezimalsystem übertragen, repräsentieren wir den Dividenden und den Divisor durch

```
172819274923789347928793874982738947 :
             12890992839109192839010     ====>

172819270000000000000000000000000000 :
             12900000000000000000000     ====>

17281927 : 1290  ( * 1000000000 )
```

Das Ergebnis der Division der Repräsentatoren, die mit Maschinenmitteln durchgeführt werden kann, verwenden wir als Multiplikator für den Divisor und ziehen das Ergebnis vom Dividenden ab:

```
17281927 : 1290 = 13396

13396 * 1000000000 *  12890992839109192839010 =
          172687740072706747271377960000000000

172819274923789347928793874982738947 -
172687740072706747271377960000000000 =
   131534851082600657415914982738947
```

Für diese Operationen stehen uns die notwendigen Algorithmen bereits zur Verfügung. Der Rest ist nun maximal um die Länge (*in Zehnerpotenzen*) des Divisorrepräsentanten verkürzt. Bei der Subtraktion sind negative Ergebnisse zu vermeiden. Dies erreichen wir durch Addition einer Eins zum Divisorrepräsentanten. Der Repräsentator des Dividenden ist nicht größer als der Dividend selbst, der Repräsentator

des Divisors größer, der Quotient aus beiden damit so beschaffen, dass negative Ergebnisse nicht eintreten können.

Im nächsten Schritt wird nun der Dividendrepräsentant neu bestimmt und der neue Quotient(*enrepräsentant*) an die korrespondierende kleinere Position im Ergebnis addiert. Nach Abschluss der Division (*der Dividend ist nach einer Subtraktion kleiner als der Divisor*) enthält der Dividend den Divisionsrest.

Die Implementation ist im Grunde relativ einfach, da die Algorithmen für Addition, Subtraktion und Multiplikation genutzt werden und lediglich die korrekte Position im Ergebnis für die Addition des nächsten Quotienten berechnet werden muss. Der Code wird jedoch durch die Berücksichtigung der Sonderfälle

- die Zahlen sind nicht (*wesentlich*) größer als die Repräsentatoren, so dass auf Datenverkürzungen und Inkremtentieren verzichtet werden kann (*gegebenenfalls kann eine komplette Rechnung mit Standardtypen durchgeführt werden*),
- der Dividend ist kleiner als der Divisorrepräsentant, aber größer oder gleich dem eigentlichen Divisor (*es kann eine zusätzliche Subtraktion durchgeführt werden*)

etwas länger.

Aufgabe. Implementieren Sie den Divisionsalgorithmus. Testfälle können Sie mittels der Multiplikation und Addition konstruieren.

15.1.2 Anmerkungen zur Implementation

Wie Sie vermutlich schon längst bemerkt haben, können Sie Algorithmen für das Rechnen mit langen ganzen Zahlen recht schnell gewinnen, wenn Sie Algorithmen für Polynome implementiert haben und diese ein wenig erweitern. Mathematisch ist das zu erwarten, denn für Polynome mit Koeffizienten aus den rationalen Zahlen (*Kurzschreibweise* $\mathbb{Q}[x]$) gelten die gleichen Rechenregeln wie für die ganzen Zahlen $\mathbf{Z}$ selbst. Betrachten wir die Einschränkung $\mathbf{Z}[x] \subset \mathbb{Q}[x]$ und setzen $x \to z$ mit irgendeiner ganzen Zahl z, so werden die Polynome auf ganze Zahlen abgebildet. Die Abbildung ist surjektiv – verschiedene Polynome erzeugen bei dieser Einsetzung die gleiche ganze Zahl – und mathematisch daher ein Homomorphismus.

Die Erweiterungsregeln sind recht einfach:

(a) Führe die Rechnung mittels der für Polynome zur Verfügung stehenden Algorithmen durch.
(b) Lege das Vorzeichen anhand des Vorzeichens des höchsten Koeffizienten fest.
(c) Führe eine Ziffernreduktion unter Beachtung des ermittelten Gesamtvorzeichens durch. Die Ziffernreduktion besteht aus

   - einer modulo-Division durch z und Übertrag des Divisionsergebnisses auf die nächste Stelle,
   - ggf. Addition oder Subtraktion von z zum Einstellen des richtigen Vorzeichens und erneuten Übertrag $\pm$ einer Einheit.

Lediglich die Division erfordert eine genauere Untersuchung. Wie man leicht überprüft, gilt der kleine, aber entscheidende Unterschied bei der division mit Rest zwischen den verschiedenen Ringen

$$\begin{array}{ll} \mathbb{Z}, \mathbb{Q}[x]: & a = k * b + r \ , \ |r| < |b| \quad , \quad grad(b) \\ \mathbb{Z}[x]: & a = k * b + r \ , \ grad(r) \leq grad(a) \end{array}$$

Auf Polynomen mit ganzzahligen Koeffizienten sind die höchsten Ziffern somit nicht in jedem Fall erfolgreich so durcheinander teilbar, dass die höchste Potenz verschwindet. Das Problem ist aber auf eine einfache Art zu beseitigen, die oben bereits beschrieben wurde und die wir auch bei der Division auf dem Papier verwenden: wir nutzen die Surjektivität von $v{:}\mathbb{Z}[x] \rightarrow \mathbb{Z}$ aus und führen die Division statt mit dem vorgelegten Polynom mit einem anderen aus, das einen um 1 verminderten Grad aufweist:

$$P_n(x) \rightarrow Q_{n-1}(x) \ , \ q_{n-1} = p_{n-1} + z * p_n \ , \ q_k = p_k$$

Das ändert zwar nichts am grundsätzlichen Unterschied der Division mit Rest, sorgt jedoch dafür, dass der Divisionsrest $< z$ist und daher kein Ziffernübertrag mehr stattfindet. Damit ist in einem Schritt tatsächlich die höchste Stelle des Dividendenpolynoms beseitigt. Die weitere Aufarbeitung erfolgt wie beschrieben.

15.1.3 Verbesserung der Effizienz

Angesichts des Aufwands bei der Multiplikation stellt sich die Frage, ob nicht Algorithmen mit einer geringeren Abhängigkeit als der quadratischen von der Faktorengröße zu finden sind. Die Antworten (*es sind sogar zwei bekannt*) lauten „ja“. Die Antworten auf die Folgefragen, ob das anwendungstechnische Auswirkungen hat, sind allerdings „derzeit wohl nur bescheidene“ beziehungsweise „vielleicht in Zukunft“. Da die Algorithmen technisch recht interessant und lehrreich sind, schauen wir sie uns trotzdem an:

15.1.3.1 Karatsuba – Multiplikationsalgorithmus

Für die erste Möglichkeit spalten wir große Zahlen in zwei Teile auf (*wir zerlegen sie sozusagen in jeweils zwei anfangs sehr große Ziffern*) und multiplizieren die Teile aus:

$$\begin{array}{l} z_1 * z_2 = (a + b * K) * (c + d * K) = \\ \quad a * c + (a * d + b * c) * K + b * d * K^2 \end{array}$$

Formal haben wir damit zunächst nichts gewonnen. Der mittlere Term lässt sich aber durch einen anderen substituieren:

$$a * d + b * c = a * c + b * d + (a - b) * (d - c)$$

Setzen wir dies in die vorhergehende Gleichung ein und zählen die Operationen ab, so stellen wir (*möglicherweise überrascht*) fest, dass eine der aufwendigen Multiplikationen durch drei relativ preiswerte Additionen ersetzt worden sind. Ab einer bestimmten Zahlengröße ist das sicherlich günstiger als die reine Multiplikation, aber wirklich gewonnen haben wir offenbar eigentlich noch nichts. Interessant wird der Algorithmus erst bei rekursiver Anwendung, indem die drei verbleibenden Multiplikationen dem gleichen Rechenschema unterworfen werden, nun aber mit halber Wortbreite. Die Aufwandsabschätzung bei Rekursion lautet unter Berücksichtigung der Additionsteile

$$O(MUL[m]) = 3 * O(MUL[m/2]) + k_{add} * m$$

Durch Einsetzen und unter Vernachlässigung der linearen Terme erhält man daraus den folgenden Zusammenhang zwischen Aufwand und Faktorengröße:

$$O(MUL[m]) = O(m^{ld(3)})$$

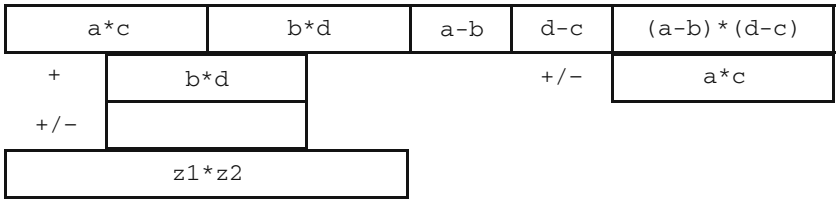
Anstelle der quadratisch mit der Zahlenlänge steigenden Ordnung führt die auf den ersten Blick unscheinbare Modifikation nun auf den Exponenten 1,58. Ab irgendeiner Zahlengröße ist das Rechnen nach diesem Algorithmus dann keine Frage des Snobismus, sondern ein Muss, auch wenn er bei kleineren Zahlen vielleicht noch ungünstig ist. Wo der Übergang zu einem günstigeren Laufzeitverhalten liegt, kann durch Versuche festgestellt werden.

Bei einer Implementation wird man vielleicht zunächst auf den Gedanken kommen, die Länge einer Zahl (*in Bit oder Byte*) zunächst auf die nächste Potenz der Zahl Zwei zu erweitern, um rekursiv in jedem Schritt die Länge bequem halbieren zu können. Eine Zahl mit 51 Byte Länge wird also auf 64 Byte erweitert, allerdings genauso eine Zahl von 34 Byte. Im Laufe der Rekursion wird man dann zwar feststellen, dass in einigen Bereichen nichts mehr zu tun ist, weil ohnehin Null als Ergebnis herauskommt, und dadurch die Verluste aus der Aufblähung im Rahmen halten können; besonders befriedigend finde ich das allerdings nicht. Wir diskutieren deshalb hier einen Algorithmus, der auf solche Erweiterungen verzichtet.

Um das System nicht laufend mit Speicheranforderungen während der Rekursion zu belasten, reservieren wir zunächst für die Speicherung der Zwischenergebnisse einen statischen Puffer der zehnfachen Länge der Faktoren (*3 Produkte der Länge m, 2 Summen der Länge m/2, in den weiteren Rekursionsschritten davon jeweils ½, ¼, usw., was insgesamt den Faktor Zwei ausmacht, zuzüglich ein wenig Reserve. Die Bruchteile der ursprünglichen Länge genügen jeweils, da der für die Berechnung des ersten Produktes während der Rekursion notwendige Speicher anschließend nicht mehr benötigt wird und für die Berechnung des zweiten Produkts zur Verfügung gestellt werden kann*). An die Rekursionsfunktion übergeben wir jeweils Zeiger auf die beiden Faktoren, ihre Längen, den Startindex im Puffer für die Zwischenrechnung sowie die Länge des erwarteten Ergebnisses:

```
void karatsuba(ulong st,          //Puffer Start
           ulong len,             //Länge Ergebnis
           const uchar* z1, ulong l1,  //1. Faktor
           const uchar* z2, ulong l2)  //2. Faktor
```

Der Pufferbereich wird folgendermaßen organisiert:



Die Produkte $a * c$ und $b * d$ werden hintereinander gespeichert, so dass durch Addition/Subtraktion der drei weiteren Werte direkt das Ergebnis entsteht. Nach Berechnung des dritten Produkts wird

$$a * c + (a - b) * (d - c)$$

berechnet und diese Summe nach Addition von $b * d$ in der Mitte des Ergebniswerte ebenfalls dort addiert. Dabei müssen die Vorzeichen bei den Subtraktionen berücksichtigt werden. Sind beide Differenzen positiv oder negativ, so können alle Zahlen addiert werden. Besitzt das dritte Produkt ein negatives Vorzeichen, so sind die Produkte $a * c$ und $b * d$ zunächst von diesem zu subtrahieren und dieses Zwischenergebnis je nach dabei auftretendem Vorzeichen vom Zentralbereich des Gesamtergebnisses zu addieren oder zu subtrahieren.
Im Detail: Der Algorithmus arbeitet mit beliebigen Wortlängen. In jedem Schritt wird die Wortlänge der Teilung durch

```
nlen = max(l1,l2);
nlen=(nlen+1)/2;
```

berechnet. Ist die Länge der Zahlen kein vielfaches von Zwei, so ist die Teilungslänge etwas größer als die halbe Zahlenlänge (*weshalb in den Überlegungen zur Pufferdimensionierung eine großzügige Reserve berücksichtigt wurde*). Für die Teilprodukte werden die Startbereiche und Längen im Hilfspuffer festgelegt und die Rekursion aufgerufen:

```
// Berechnung a*c
ac=st;
karatsuba(ac,nlen*2,z1,min(l1,nlen),
          z2,min(l2,nlen));

// Berechnung b*d
bd=ac+2*nlen;
```

```
if(l1>nlen && l2>nlen){
    karatsuba(bd,nlen*2,&z1[nlen],l1-
                        nlen,&z2[nlen],l2-nlen);
}else{
    memset(&kfeld[bd],0,2*nlen);
}//endif
...
```

Da auf dem letzten Produkt noch Additionen auszuführen sind, bei denen Überläufe entstehen können, wird das Differenzenprodukt mit einem breiteren Speicher ausgestattet (-> *Reserve*). Die Längen sind dabei von den arithmetischen Funktionen genau einzuhalten, weshalb wir oben bei der Addition die Unterteilung in Bereiche, in denen mit dem Typ `ulong` gearbeitet wird und solche, in denen `uchar` verwendet wird, vorgenommen haben. Abschließend wird das Ergebnis ermittelt (`vzxy` *ist* `true`*, wenn die zugehörende Zahl negativ ist; die Subtraktionsfunktion gibt bei negativen Ergebnissen das positive Zweierkomplement aus*):

```
// Berechnung der Summen
if((vzab&&vzdc) || !(vzab||vzdc)){
    _add(&kfeld[rs],2*nlen+1,&kfeld[ac],2*nlen);
    _add(&kfeld[rs],2*nlen+1,&kfeld[bd],2*nlen);
    _add(&kfeld[ac+nlen],3*nlen,
                          &kfeld[rs],2*nlen+1);
}else{
    vzab=_sub(&kfeld[rs],2*nlen,&kfeld[ac],2*nlen);
    if(!vzab){
        vzab=_sub(&kfeld[rs],2*nlen,
                          &kfeld[bd],2*nlen);
        if(!vzab){
            _sub(&kfeld[ac+nlen],3*nlen,
                          &kfeld[rs],2*nlen);
        }else{
            _add(&kfeld[ac+nlen],3*nlen,
                          &kfeld[rs],2*nlen);
        }//endif
    }else{
        _add(&kfeld[rs],2*nlen+1,
                          &kfeld[bd],2*nlen);
        _add(&kfeld[ac+nlen],3*nlen,
                          &kfeld[rs],2*nlen+1 );
    }//endif
}//endif
```

Kommen wir nun zur Abschätzung der praktischen Bedeutung: Im Gegenzug zur geringeren Ordnung ist das Verfahren mit einigem Verwaltungsaufwand und

zusätzlichen Additionen verbunden. Es sollte daher erst ab einer bestimmten Schwellenlänge der Zahlen wirklich effizienter als die Schulmultiplikation sein. Vergleichsmessungen zeigen, dass erst ab einer Zahlenlänge von 80 Byte (= 640 Bit) ein Gleichstand zwischen beiden Verfahren (*in dieser Implementation*) erreicht wird, wobei unterhalb von 40 Byte die Rekursion beendet und die Zwischenergebnisse auf herkömmliche Art berechnet werden sollten. Der Unterschied zwischen beiden Methoden wächst nur langsam: Erst bei einer Zahlenlänge von etwa 6.500 Byte wird ein Laufzeitverhältnis von 5:1 erreicht.

Verschlüsselungsverfahren auf der Basis großer Zahlen werden heute mit Zahlenlängen zwischen 1.024 Bit beim RSA–Verfahren und 2.048 Bit beim Diffie–Hellman–Verfahren abgewickelt, entsprechend 128 beziehungsweise 256 Byte. Auf einem 700 MHz–Pentium–Prozessor lassen sich mit den Hochsprachenversionen ohne Reserve 33.000/37.000 beziehungsweise 10.200/14.600 Multiplikationen durchführen, entsprechend einer Durchsatzsteigerung von 12% beziehungsweise 43%. Der Gewinn ist praktisch zwar bescheiden, trotzdem sollte für Rechnungen mit größeren Zahlen der Algorithmus zur Standardausrüstung einer Bibliothek gehören, wenn auf reinen Assemblerkode oder spezielle Hardware verzichtet wird.

Auch von diesem Algorithmus lässt sich eine spezielle Version für Quadrierungen angeben. Aus der allgemeinen Gleichung wird in diesem Fall

$$(a + b * K) * (a + b * K) = a^2 + b^2 * K^2 + 2ab * K$$

Zu vereinfachen ist an diesem Ausdruck nichts. Erst bei rekursiver Ausführung ergeben sich Vorteile, da der dritte Term durch eine gewöhnliche Karatsuba–Multiplikation ausgewertet werden kann (*die ersten beiden sind wiederum Quadrate, die jeweils in zwei weitere kleinere Quadrate und ein normales Produkt zerfallen*).

Aufgabe. Implementieren Sie alles und führen Sie Effizienzmessungen für die Quadrierung durch.

15.1.3.2 Multiplikation mittels Fouriertransformation

Wie oben angedeutet, existiert noch eine weitere Methode mit geringerer Aufwandsordnung. Dazu muss allerdings etwas tiefer in die mathematische Trickkiste gegriffen werden.

Wir betrachten noch einmal das Multiplikationsschema. Die Berechnung einer bestimmten Ziffer des Ergebnisses können wir folgendermaßen formulieren:

$$Z_N = \sum_{k=0}^{N} Z_{a,k} * Z_{b,N-k} \tag{15.1}$$

Hierbei handelt es sich natürlich zunächst um ein Zwischen- und Rohergebnis, denn für das endgültige Ergebnis ist die Berechnung aller Ziffern sowie ein Ziffernübertrag notwendig. Wir wollen (15.1) denn auch nur benutzen, um einem weiteren Verfahren auf die Spur zu kommen.

Diese Art der Summierung in (15.1) wird in der Mathematik als diskrete Faltung (*der beiden Ziffernreihen* Z_a, Z_b) bezeichnet. Der Begriff Faltung wiederum ist häufig mit den Begriffen Fouriertransformation und Reihenentwicklung verbunden. Wir gehen diesen Begriffen nun ein wenig auf den Grund.

Funktionen werden bekanntermaßen auf Rechnern gerne in genäherter Form als interpolierende Polynome implementiert. Sind Stützpunkte im Intervall $[0, 2\pi]$ gegeben – was man durch eine einfache Koordinatentransformation immer erreichen kann, kann das interpolierende Polynom auch als trigonometrisches Polynom formuliert werden, wobei wir hier der Bequemlichkeit und Klarheit halber gleich die komplexe Schreibweise verwenden:[9]

$$f(x) = \sum_{k=0}^{N-1} \alpha_k \omega^k \ , \ \ \omega = e^{ix} \tag{15.2}$$

Für die Stützpunkte $0 \leq x_0 < x_1 < \ldots < x_{n-1} < 2\pi$ wählen wir aus Gründen, die gleich klar werden, die spezielle Bedingung

$$x_k = 2\pi k/N \ , \ k = 0, \ldots N-1 \ , \ \omega_k = e^{ix_k} \tag{15.3}$$

Mit (15.3) gilt nämlich

$$\omega_k^I = \omega_l^k \ , \ \sum_{k=0}^{N-1} \omega_k^I \omega_k^{-j} = N\delta_{kj} \tag{15.4}$$

Die erste Beziehung ist trivial, für die Prüfung der zweiten ist zu beachten, dass jedes ω_s Nullstelle des Polynoms $(\omega^N - 1)$ ist.[10] Wegen

$$\omega^N - 1 = (\omega - 1)(\omega^{N-1} + \omega^{N-2} + \ldots + 1) \tag{15.5}$$

ist aber $\omega_s = \omega_{1-j} = 1$, was für $l = j$ erfüllt ist, oder

$$\sum_{k=0}^{N-1} \omega_{l-j}^k = \sum_{k=0}^{N-1} \omega_k^{l-j} = \sum_{k=0}^{N-1} \omega_k^l \omega_m^{-j} = 0 \tag{15.6}$$

---

[9] Häufig schrecken Leute vor komplexen Zahlen zuück wie der Teufel vor dem Weihwasser. Machen Sie sich aber einfach einmal klar, dass komplexe Zahlen wie rationale Zahlen einfach nur aus zwei Komponenten bestehen. Außer den anderen Rechenvorschriften sind sie also nicht komplizierter als rationale Zahlen.

[10] Die Stützstellen (15.3) sind ja nichts anderes als die N-ten komplexen Einheitswurzeln, wie man sich leicht überzeugt.

In der Sprache der linearen Algebra bilden die ω_k^l also eine Matrix aus orthogonalen Spaltenvektoren. Dies werden wir bei der Formulierung der Faltung (15.1) nach der Transformation noch benötigen.

Wir betrachten nun die Ziffern unserer Zahlen als Funktionswerte, die wir zur Berechnung der Koeffizienten α_k verwenden. Diese erhalten wir einfach durch

$$\alpha_k = \frac{1}{N} \sum_{j=0}^{N-1} Z_j \omega_j^{-k} \tag{15.7}$$

Setzen wir nämlich (15.2) mit den zugehörenden Indizes in die rechte Seite von (15.7) ein und beachten die Orthogonalitätsbeziehungen (15.6), so bleibt die linke Seite übrig. (15.7) und (15.2) sind somit unsere Basisformeln für die Fouriertransformation und Rücktransformation von Ziffern. Fassen wir das etwas übersichtlicher zusammen. Im Besitz der Matrix

$$\Omega = \begin{pmatrix} \omega_0^0 & \omega_0^1 & \dots \\ \omega_1^0 & \omega_1^2 & \dots \\ \dots & & \omega_{N-1}^{N-1} \end{pmatrix} \tag{15.8}$$

gilt

$$\vec{Z} = \begin{pmatrix} z_1 \\ \dots \\ Z_{N-1} \end{pmatrix} = \Omega \vec{\alpha} \ , \ \vec{\alpha} = \begin{pmatrix} \alpha_1 \\ \dots \\ \alpha_{N-1} \end{pmatrix} = \Omega^{-1} \vec{Z} \tag{15.9}$$

mit

$$\Omega - 1 = \overline{\Omega^T} \tag{15.10}$$

Wir können für die Praxis also zunächst auf Bekanntes aus Kap. 6 zurückgreifen.

Um nun wieder die Verbindung zur Multilpikation herzustellen, setzen wir die Transformierten in die Schulmultiplikation ein:

$$\begin{aligned} Z &= \sum_{i=0}^{N-1} \sum_{k=0}^{N-1} Z_{a,i}{}^* Z_{b,k} \\ &= \sum_{i,k} \left(\sum_{j=0}^{N-1} \alpha_j \omega_i^j \right) \left(\sum_{l=0}^{N-1} \beta_j \omega_k^l \right) \\ &= \sum_{j,l} \alpha_j \beta_l \sum_{i,k} \omega_i^j \omega_k^l = \sum_j \alpha_j \beta_j \end{aligned} \tag{15.11}$$

Dabei haben wir wieder die Bedingungen (15.4) sowie die Zyklizität $\omega_k = \omega_{k+z*N}$ verwendet,[11] um die Summe über die verschiedenen ω-Terme auszuwerten. Anstatt N^2 Multiplikationen durchführen zu müssen, genügen nach der Transformation N Multiplikationen, um das Ergebnis zu erhalten.

Bleibt die Frage nach dem Aufwand für die (*diskrete*) Fouriertransformation. Hier stellen wir aber zunächst fest: wir haben noch nichts gewonnen, denn die Transformation benötigt als Matrixmultiplikation wieder N^2 Multiplikationen, d.h. die Ordnung des Algorithmus hat sich noch nicht geändert. Also alles umsonst? Nein, denn die schnelle diskreten Fouriertransformation erlaubt die Transformation von Zahlen mit $N{=}2^n$ Ziffern mit dem Aufwand

$$O(FFT) \approx O(N * ld(N) * ld(ld(N)))$$

Dies ist die asymptotische Führungsgröße, da der Multiplikationsaufwand selbst linear mit der Anzahl der Ziffern steigt, und sie ist nochmals besser als diejenige für die Karatsuba–Multiplikation. Die Proportionalitätsfaktoren sind bei (*zwei*) Fouriertransformationen allerdings recht hoch, so dass wir auch hier erst ab recht großen Zahlen oder bei Einsatz spezieller Hardware mit einem echten Vorteil rechnen können. Hinzu kommt als weiterer Effekt aus dem Transformationsalgorithmus, dass die Anzahl der Ziffern in der Rechnung Potenzen von Zwei seien sollten, das heißt die optimale Zahlengröße ändert sich in recht großen Sprüngen bzw. bei nur geringem Überschreiten einer Zweipotenz gelangt man schnell wieder in einen Bereich, in dem einer der anderen Algorithmen günstiger läuft. Alles das trägt dazu bei, dass sinnvolle technische Anwendungen dieses Algorithmus derzeit wohl kaum vorhanden sind. Algorithmen für die technisch noch nicht realisierten Quantencomputer nutzen jedoch dieses Verfahren, so dass sich die Bedeutung in Zukunft durchaus auch anders darstellen könnte.
Sehen wir uns nochmals (15.6) und (15.7) an. Mit $M = N/2$[12] lassen sich aus der Symmetrie der trigonometrischen Funktion zunächst folgende Beziehungen zwischen den Exponentialtermen ableiten:

$$\omega_N^{jk} = \omega_N^{j(k+M)} \ , \ \left(\omega_N^{jk}\right)^2 = \omega_M^{jk} \ , \ \omega_N^M = -1 \tag{15.12}$$

Mit Hilfe dieser Eigenschaften lassen sich die geraden und die ungeraden Terme des Polynoms durch Summen mit halber Länge ausdrücken:

---

[11] Für den Nachweis kann man wie in (15.6) vorgehen und bei festgehaltenem Index i nur über k summieren. Der dabei auftretende Doppelindex ij wird aufgrund der Zyklizität eindeutig auf einen anderen Index abgebildet, so dass man letztendlich $\sum \omega_i^j \omega_k^l = c * \delta_{i,k}\delta_{j,l}$ erhält.

[12] Wegen N=2^n sind die Werte immer ganzahlig.

$$\begin{aligned} N\beta_{2\mathrm{h}} &= \sum_{k=0}^{N-1} z_k \omega_N^{2hk} \sum_{k=0}^{M-1} (z_k + z_{k+M})\omega_M^{hk} \\ N\beta_{2\mathrm{h}+1} &= \sum_{k=0}^{N-1} z_k \omega_N^{(2\mathrm{h}+1)k} = \sum_{k=0}^{M-1} \left((z_k - z_{k+M})\,\omega_N^k\right)\omega_M^{hk} \end{aligned} \tag{15.13}$$

Die beiden reduzierten Summenterme besitzen wiederum die gleichen Eigenschaften wie die ursprüngliche Summe, so dass der Prozess der Zerlegung in kürzere Summen rekursiv fortgesetzt werden kann. Verwenden wir die Indizes

- $m = n, n-1, n-2, \ldots 0$ zur Kennzeichnung des Reduktionsschrittes,
- $M = 2^{m-1}$ für die Anzahl der im betreffenden Reduktionsschritt pro Summe vorhandenen Summanden,
- $r = 0, 1, 2, .. R-1, R = 2^{n-m}$ für die Anzahl und Kennzeichnung der Summen,
- $j = 0,1, \ldots 2\mathrm{M}-1$ zur Indizierung der verschiedenen Gleichungen einer Gruppe,

so lassen sich die Summen allgemein durch folgenden Ausdruck angeben

$$N\beta_{jR+r} = \sum_{k=0}^{2M-1} z_{r,k}^{(m)} * \omega_M^{jk} \tag{15.14}$$

wobei die Terme $z_{r,k}^{(m)}$ Summen jeweils zweier Koeffizienten der vorhergehenden Summenformel darstellen. Ausgehend von den Startwerten

$$z_{0,k}^{(n)} = z_k \tag{15.15}$$

erhalten wir über die Iteration

$$\begin{aligned} z_{r,k}^{(m-1)} &= z_{r,k}^{(m)} + z_{r,k+M}^{(m)} \\ z_{r+R,k}^{(m-1)} &= \left(z_{r,k}^{(m)} - z_{r,k+M}^{(m)}\right)\omega_m^k \end{aligned} \tag{15.16}$$

schließlich die Lösungen

$$\beta_r = \frac{1}{N} z_{r,0}^{(0)} \tag{15.17}$$

Während für die Auswertung der ursprünglichen Summen $O(N^2)$ Rechenschritte notwendig waren, sind dies jetzt nur noch $O(N \log N)$ Schritte. Da pro Schritt je zwei Werte zu zwei neuen Werten verdichtet werden, die die vorhandenen ersetzen, wird während der Rechnung kein zusätzlicher Speicherplatz benötigt, sondern die Daten können auf in der weiteren Rechnung nicht mehr benötigten Positionen abgelegt werden. Wir ersetzen nach der Summation jeweils

$$z_{r,k}^{(m)} \quad \Rightarrow \quad z_{r,k}^{(m-1)} \quad ; \quad z_{r,k+M}^{(m)} \quad \Rightarrow \quad z_{r+R,k}^{(m-1)} \tag{15.18}$$

Dies führt zu einer etwas verwickelten Indexabbildung, das heißt die der Ursprungsposition i entsprechende Zielposition ist eine andere Zahl k . Die geraden Terme eines Reduktionsschrittes werden im vorderen Bereich des Feldes angeordnet, die ungeraden im hinteren. Bei rekursiver Durchführung des Algorithmus ergibt sich daraus folgende Transposition der Indizes, an einem Beispiel mit acht Ziffern dargestellt:

0	1	2	3	4	5	6	7
0	2	4	6	1	3	5	7
0	4	2	6	1	5	3	7
0	4	2	6	1	5	3	7

Das Transpositionsmuster wird leicht verständlich, wenn man das Bitmuster der Indexpositionen betrachtet. Die Schlusspositionen bestehen aus dem inversen Bitmuster der Startpositionen:

$$a_{n-k-1}2^k \Rightarrow a_k 2^k \quad ; \quad k = 0..n-1 \tag{15.19}$$

Da bei der Rücktransformation auf diese Indizes zugegriffen werden muss, empfiehlt sich die Anlage einer Tabelle. Die Invertierung erfolgt durch versetztes Schieben auf zwei Variablen, das in Hochsprache relativ mühsam aussieht und in Assembler deutlich eleganter lösbar ist.

```
for(i=0;i<N;i++){
    ii=i;
    res=0;
    for(j=0;j<n;j++){
        res<<=1;
        if(ii&1)
            res|=1;
        ii»=1;
    }//endfor;
    index[i]=res;
}//endfor
```

```
// Assemblersequenz
    mov    R1,N
    mov    R2,i
    mov    R3,0
lp: rrc    R2
    rlc    R3
    dec    R1
    jnz    lp
    mov    [index+i],R3
```

So weit wir den Algorithmus bis hier bearbeitet haben, müssen wir drei Transformationen durchführen, nämlich die Transformationen der beiden Faktoren und die Rücktransformation des Ergebnisses. Von diesen drei Transformationen kann eine eingespart werden, wenn wir von vornherein mit komplexen Größen arbeiten und weitere Symmetrieeigenschaften der Fouriertransformation berücksichtigt werden. Da wir auf jeden Fall mit komplexen Größen arbeiten müssen, können wir nämlich auch den Startvektor komplex machen, wobei folgende Symmetrie gilt:

Ausgangswerte	Spektrum
Reell	Konjugiert komplex
Reell und ungerade	Imaginär und ungerade
Reell und gerade	Reell und gerade
Imaginär und gerade	Imaginär und gerade
Imaginär und ungerade	Reell und ungerade
Konjugiert komplex	Reell

Die Begriffe „gerade“ und „ungerade“ beziehen sich dabei auf die Symmetrie der Daten. Die Daten mit den Indizes 0 und $N/2$ werden isoliert von den anderen betrachtet, für die restlichen Indizes folgt

$$\textit{gerade}\colon x[k] = x[N-k]\ ; \quad \textit{ungerade}\colon x[k] = -x[N-k]$$

Ein beliebiges Signal lässt sich durch

$$\begin{aligned} x_g[k] &= \tfrac{1}{2} * (x[k] + x[N-k]) \\ x_u[k] &= \tfrac{1}{2} * (x[k] - x[N-k]) \end{aligned} \tag{15.20}$$

in einen geraden und einen ungeraden Anteil zerlegen. Die Rücktransformation, wenn gerade und ungerade Anteile bekannt sind, ist damit auch geklärt. Wir nutzen diese Eigenschaften nun zur Einsparung einer Transformation aus, in dem wir eine Variable (*den ersten Faktor unserer Multiplikationsaufgabe*) auf den Realteil des komplexen Puffers laden, die andere auf den Imaginärteil:

$$\begin{aligned} z[k] &= zl[k] + i * z2[k] \\ &= (z1_g[k] + z1_u[k]) + i * (z2_g[k] + z2_u[k]) \end{aligned} \tag{15.21}$$

Mit Hilfe der Symmetrietabelle erhalten wir die Transformierten von *z1* und *z2* durch

$$\begin{aligned} z1_F[k] &= z_{F,\,real,\,g}[k] + i * z_{F,\,imag,\,u}[k] \\ z2_F[k] &= z_{F,\,imag,\,g}[k] + i * Z_{F,\,real,\,u}[k] \end{aligned} \tag{15.22}$$

Die geraden und ungeraden Anteilen berechnen wir wie oben beschrieben. Fassen wir nun alles zu einem Algorithmus zusammen. Die Länge des Produkts der beiden

Faktoren ist maximal die Summe der Längen der Faktoren. Der Transformationspuffer muss mindestens diese Größe aufweisen, das heißt die obere Hälfte des Feldes $z[k]$ ist normalerweise Null. Da die schnelle Fouriertransformation in der diskutierten Version[13] eine Pufferlänge von einer Potenz von zwei verlangt, vergrößern wir N auf den nächsten in Frage kommenden Wert und stellen die Anzahl der Indexbits fest:

```
l=z1.len+z2.len;
N=1;
while(N<l)
    N<<=1;
for(n=0;n<32;n++)
    if((N&((long)1<<n))!=0)
        break;
```

Welches Zahlenmodell ist bei der Transformation zu verwenden? Für die Fouriertransformation wird ein Feld komplexer Zahlen bereitgestellt, das im Realteil die Ziffern eines Faktors aufnimmt, im Imaginärteil die Ziffern des anderen.[14] Nach Ablauf des Algorithmus enthalten Realteile die Ziffern des Ergebnisses, und zwar zunächst noch einschließlich der Überläufe, die in einem weiteren Rechenschritt auf die Folgepositionen übertragen werden müssen. Im Laufe der Rechnung müssen wir mit Rundungsfehlern rechnen, die das Ergebnis natürlich nicht beeinflussen dürfen. Haben die ganzen Zahlen Ziffern mit N_z Bit Größe und bestehen die Zahlen aus n_z Ziffern, so muss eine Fließkommavariable

$$N_f = 2 * ld(n_2) * N_2 + N_{rund} \tag{15.23}$$

Bit Genauigkeit aufweisen. Verwenden wir den Datentyp `double`, so können nur Ziffern mit zwei Byte Breite verarbeitet werden, bei ganzen Zahlen mit mehr als zwei KB Speichergröße muss die Verarbeitung sogar auf ein Byte reduziert werden.[15]
Die Fouriertransformation wird nun wie beschrieben umgesetzt. Für die Hintransformation erhalten wir den Algorithmus

```
Complex<T> e,u,v;
vector<Complex<T> >::iterator it1;
```

[13] Die Vergrößerung des Rechenpuffers verändert natürlich die Aufwandsabschätzungen etwas und die Verdopplung der Puffergröße führt zu deutlichen Sprüngen in der Ausführungszeit. Es existiert auch eine Version des FFT–Algorithmus, die die Verdopplung der Puffergröße und damit die Sprünge vermeidet; wir gehen hierauf jedoch nicht weiter ein.

[14] Die Implementation von komplexen Zahlen kann schon fast als „Fingerübung“ betrachtet werden. In der C++ – Standardbibliothek ist ebenfalls eine Vorlagenklasse vorhanden, auf die auch zurückgegriffen werden kann.

[15] Auch das führt natürlich dazu, dass die Multiplikation mittels der Fouriertransformation nicht ganz so schnell zur klassischen oder Karatsuba–Multiplikation aufschließt.

```
FFT_Map<T> fm;
fm.BitSize(BigFloat().precision());
T tmp; int m,k,r,M;

for(m=n;m>0;m--){
    M=(1<<m);
    for(k=0;k<M/2;k++){
        e.real=fm.fcos(M,k);
        e.imag=-fm.fsin(M,k);
        for(r=0;r<N;r+=M){
            u=dat[r+k]; v=dat[r+k+M/2];
            dat[r+k]=u+v;
            dat[r+k+M/2]=(u-v)*e;
        }//endfor
    }//endfor
}//endfor
convert(tmp,N);
for(it1=dat.begin();it1!=dat.end();++it1)
    *it1/=tmp;
```

Der Algorithmus ist hier in einer Version für lange Fließkommatypen angegeben, die wir jedoch erst im nächsten Kapitel entwickeln werden. Für das Verständnis hier ist das Objekt `fm` zur Berechnung der Sinus- und Kosinuswerte zu erläutern. Die Funktionswerte werden sowohl für die Hin- als auch für die Rücktransformation benötigt, und wegen der Vergrößerung der Pufferbreite durch Verdopplung sind die Parameter meist für eine ganze Reihe von Multiplikationen unverändert einsetzbar. Wenn die Berechnung der Funktionswerte einen nicht zu vernachlässigenden Aufwand darstellt – was bei `double` meist nicht der Fall ist, aber auf jeden Fall bei anderen softwaregestützten Typen – ist es sinnvoll, berechnete Werte zu speichern und wieder zu verwenden. Wir deklarieren dazu einen Container vom Typ `map` mit den beiden Schleifenvariablen als Schlüssel (*die Variablen werden auf einen String kopiert*).

```
typedef map<string,BigFloat> BFCosMap;
static BFCosMap msin, mcos;

BigFloat FFT_cos(BFCosMap& mp, int M, int k,
                 bool cosinus){
    BFCosMap::iterator it; BigFloat bf;
    if(M%k==0){ M/=k; k=1; }//endif
    _ms.resize(2*sizeof(int));
    memcpy(_ms.begin(),&M,sizeof(int));
    memcpy(_ms.begin()+sizeof(int),&k,sizeof(int));
    it=mp.find(_ms);
    if(it!=mp.end()){
        return it->second;
```

```
    }else{
        bf=algebra<BigFloat>::pi();
        bf*=k; bf/=M; bf*=2;
        bf=cos(bf);
        mp.insert(pair<string,BigFloat>(_ms,bf));
        return bf;
    }//endif
}//end function
```

In einer Gesamtstrategie ist noch zu berücksichtigen, welche Genauigkeit die gespeicherten Daten aufweisen müssen (*hochgenaue Werte sind natürlich bei Verminderung der Genauigkeit weiterhin nutzbar, während der Übergang zu genaueren Werten eine Neuberechnung erfordert*) und ob Teile der Berechnung bei Änderung der Parameter weiter verwendbar sind (*bei einer Verdopplung der Stützstellen können die bereits berechneten Werte weiter verwendet werden, was die Schlüsselanpassung zu Beginn der Containerbefragung erfordert*).

Mit Hilfe der zuvor berechneten Indextranspositionen werden die transformierten Anteile beiden Zahlen getrennt und das Ergebnis durch komponentenweise Multiplikation berechnet:

```
for(i=1;i<N/2;i++){
    tmp1.real=hlb*(dat[index[i]].imag
                    + dat[index[N-i]].imag);
    tmp1.imag=-hlb*(dat[index[i]].real
                    - dat[index[N-i]].real);
    tmp2.real=hlb*(dat[index[i]].real
                    + dat[index[N-i]].real);
    tmp2.imag=hlb*(dat[index[i]].imag
                    - dat[index[N-i]].imag);
    res[index[i]]=tmp1*tmp2;
    tmp1.imag=-tmp1.imag;
    tmp2.imag=-tmp2.imag;
    res[index[N-i]]=tmp1*tmp2;
}//endfor
res[index[0]]=dat[index[0]].imag*
                dat[index[0]].real;
res[index[N/2]]=dat[index[N/2]].imag*
                dat[index[N/2]].real;
```

Die anschließende Rücktransformation unterscheidet sich kaum von der Hintransformation, berücksichtigt aber ebenfalls die Indextransposition:

```
for(m=n;m>0;m--){
    M=(1<<m);
    for(k=0;k<M/2;k++){
```

```
        e.real=fm.fcos(M,k);
        e.imag=fm.fsin(M,k);
        for(r=0;r<N;r+=M){
            i1=index[r+k];
            i2=index[r+k+M/2];
            u=dat[i1]; v=dat[i2];
            dat[i1]=u+v;
            dat[i2]=(u-v)*e;
        }//endfor
    }//endfor
}//endfor
```

Abschließend werden die ganzzahligen Anteile der rücktransformierten Werte auf die Ziffern der ganzen Zahl rückübertragen und dabei die Überläufe übertragen. Im folgenden Beispielkode ist mit der Ziffernbreite „Byte“ gearbeitet worden, so dass für die korrekte Übertragung der Überläufe mit dem Datentyp `unsigned long` gearbeitet werden kann. Haben Sie andere Ziffernbreiten verwendet, so müssen Sie den Algorithmus entsprechend anpassen.

```
ulong ll, *pl;
for(i=0;i<l;i++){
    ll=floor(dat[i].real()+0.5);
    pl=(ulong*)&buf[i];
    *pl+=ll;
}//endfor
```

Damit haben wir die Multiplikation mittels einer Fouriertransformation abgeschlossen. Der Galopp durch die Theorie war natürlich extrem schnell, und selbst sehr komprimiert geschriebene Bücher über numerische Mathematik benötigen mehr Raum, um die Theorie hinter den Algorithmen zu klären. Ich halte es aber für gut möglich, dass der verfolgte Weg – mehr oder weniger Komprimierung der Theorie auf Ausgangsbedingungen und Ergebnis und Umsetzung in einen Algorithmus und optional Nachbearbeiten der Theorie mit einem Buch der numerischen Mathematik – Ihnen sogar einen leichteren Zugang verschafft, da schneller Ergebnisse zu sehen sind.

Welche Vorteile bringt nun diese Version der Multiplikation? Einige Probeläufe mit dieser Implementation überzeugen davon, dass ein Gewinn bei technisch interessanten Zahlengrößen nicht zu erwarten ist. Möglicherweise sieht das anders aus, wenn eine Beschleunigung der Transformation durch spezielle FFT–Hardware möglich ist; allerdings kann ich da aus eigener Erfahrung nichts beisteuern. Wie weit Zahlentheoretiker in Bereiche vorstoßen, in denen der Algorithmus grundsätzlich interessant ist, kann auch nur eine Analyse der Spezialliteratur beantworten. Projektiert sind Anwendungen bei den derzeit noch nicht existierenden Quantencomputern, beispielsweise in Algorithmen zur Faktorisierung großer Zahlen.

Weitere Beispiele – allerdings wohl auch mehr von theoretischem Interesse – diskutieren wir bei den großen Fließkommazahlen.

15.2 Quotientenkörper

Die Algebra kennt im wesentlichen zwei Methoden, (*bestimmte*) Ringe in Körper[16] zu überführen. Bei unendlichen Ringen sind dies Quotientenkörper, bei endlichen Ringen Restklassenkörper. Für Details zu diesen Begriffen verweise ich auf einführende Lehrbücher über Algebra.

Das bekannteste Beispiel eines Quotientenkörpers ist sicherlich der Körper der rationalen Zahlen, der aus dem Ring der ganzen Zahlen entsteht, wenn die dort bestehenden Probleme der Umkehrung der Multiplikation beseitigt werden. Eine rationale Zahl lässt sich durch ein Paar ganzer Zahlen darstellen, und das Rechnen mit rationalen Zahlen hat gegenüber Fließkommazahlen den Vorteil, exakt durchgeführt werden zu können (*solange auf den Aufruf von Funktionen mit irrationalen Ergebnissen verzichtet werden kann*). Vorzugsweise zur Lösung theoretisch begründeter Probleme, bei denen es schon einen wesentlichen Unterschied bedeuten kann, ob ein Ergebnis exakt 1/3 oder irgendwie 0,33333333...??? ist, wird man sich rationaler Zahlen als Rechengrößen bedienen.

Ein in technischen Anwendungen häufiger auftretender Quotientenkörper ist der der rationalen Funktionen, dessen Elemente sich ähnlich wie bei den rationalen Zahlen als Quotienten von Polynomen darstellen lassen.
Eine einfache, alle Fälle umfassende Implementation ist

```
template <class T> class Rational {
   ...
protected:
   T zaehler, nenner;
};//end class
```

Die Implementation von Rechen- und Vergleichsoperatoren entspricht den gewöhnlichen Rechenregeln mit rationalen Zahlen, also

$$q_1 * q_2 = \frac{z_1 * z_2}{n_1 * n_2}\ ,\ q_1 + q_2 = \frac{z_1 * n_2 + z_2 * n_1}{n_1 * n_2}$$

Bei allen Rechenvorgängen werden Zähler und Nenner dabei recht schnell sehr groß, so dass regelmäßig gekürzt werden sollte. Hierfür ist der größte gemeinsame

[16] Kurzerläuterung: „Ringe“ sind algebraische Gebilde, in denen Addition, Subtraktion und Multiplikation vollständig erklärt sind, die Division aber nicht immer lösbar ist. In „Körpern“ ist auch die Divisionsaufgabe eindeutig lösbar. Als Beispiel betrachte man die Aufgabe 5/2. In $\mathbb{Z}$ findet man keinen Faktor, der mit 2 multipliziert 5 ergibt, in $\mathbb{R}$ ist der Faktor durch 2,5 wohldefiniert.

Teiler von Zähler und Nenner festzustellen, was mit Hilfe des euklidischer Algorithmus bewerkstelligt wird. Dazu stellen wir den Bruch *Z/N* als Division mit Rest dar, wobei der Rest immer positiv ist:

$$Z = N * f + r_1 \;\; ; \;\; r_1 < |N|$$
$$5/2 \rightarrow 5 = 2 * 2 + 1 \;\; , \;\; -5/2 \rightarrow -5 = (-3) * 2 + 1$$

Dieses „Ordnungsschema“ einer Division mit Rest ist Voraussetzung dafür, dass wir den Ring, dem Zähler und Nenner entstammen, überhaupt zu einem Quotientenkörper erweitern dürfen. Für die ganzen Zahlen ist dies durch Einführung des Betrages einer Zahl unmittelbar klar, für Polynome ist für die Ordnungsrelation anstelle des Betrages der Grad des Polynoms heranzuziehen, das heißt die höchste Potenz, deren Koeffizient von Null verschieden ist (*vergleiche Kap. 9.5 für weitere Details*):

$$r_1(x) < N(x) \Leftrightarrow grad(r_1) < grad(N)$$

Ein gemeinsamer Teiler von *Z* und *N* ist sicher auch Teiler des Restes, so dass wir den Prozess iterativ fortsetzen können:

$$n = r_1 * f' + r_2$$
$$r_1 = r_2 * f'' + r_3 \ldots$$

Da die Reste dem diskreten Ring entstammen und immer kleiner werden, aber die Null nicht unterschreiten können bricht die Iteration irgendwann ab mit $r_{n+1} = 0$, und $r_n = ggT(z,n)$ ist der gesuchte Teiler, durch den Zähler und Nenner gekürzt werden können.

Aufgabe. Implementieren Sie, falls in Kap. 4.1 oder 9.5 noch nicht erfolgt, anhand der Beschreibung den `ggT`–Algorithmus.

Das Kürzen eines „Bruches“ kann nun nach beliebigen Kriterien vorgenommen werden:

(a) Manuell, was aber eine permanente Kontrolle in der Anwendung bedeutet;
(b) Am Ende einer Anweisungszeile, bei `a=b+c+d` also beim Abspeichern der Summe auf der Variablen `a` ;
(c) Bei Überschreiten einer vorgegebenen Größe der Datenpuffer für Zähler oder Nenner.

Sie können nun die Vorlagenklasse `Rational` vollständig mit Kürzung implementieren und für die Arbeit mit ganzen Zahlen oder Polynomen einsetzen. Im allgemeinen Fall sind wir damit bereits am Ende der Betrachtungen angelangt. Wir betrachten als Ergänzung speziell für die rationalen Zahlen aber noch eine Methode, bei Rechnungen mit Näherungswerten anstelle der exakten Werte die Fehlergröße der Rechnung zu kontrollieren. Überschreitet die Länge der Zahlen trotz Kürzens eine vorgegebene Schwelle (*oder sind irrationale Zahlen zu verarbeiten, zum Beispiel Wurzeln, Logarithmen, usw*), so kann eine Reduktion auf eine Zahl vorgegebener

Länge und minimalem Abstand zum echten Wert durchgeführt werden, das heißt gesucht wird die Zahl, für die

$$R_M = \{r{:}max(Z.len,\, N.len) \leq M\}$$
$$r_{opt} \in R_M{:}\forall r_M \in R_M \neq r_{opt}{:}\left|r - r_{opt}\right| < |r - r_M|$$

Für eine vorgegebene maximale Zahlenlänge M ist dies durch eine Kettenbruchentwicklung von Zähler und Nenner zu erreichen. Dazu entwickeln wir

$$\frac{a}{b} = v_0 + \frac{r_1}{b} = v_0 + \frac{1}{\frac{b}{r_1}} = v_0 + \frac{1}{v_\cdot + \frac{r_2}{r_1}} = \ldots$$
$$= v_0 + \cfrac{1}{v_1 + \cfrac{1}{v_2 + \cfrac{1}{\ldots + v_{n-2} + \cfrac{1}{v_{n-1} + \frac{1}{v_n}}}}}$$

Bei endlichen Brüchen bricht diese Entwicklung wie beim `ggT` schließlich ab.[17] Ein Bruch lässt sich dann auch durch die Teiler v_k darstellen, beispielsweise

$$\frac{1355}{946} = [1{,}2{,}3{,}5{,}8{,}3]$$

Lassen wir die hinteren Glieder fort und setzen aus dem verbleibenden Kettenbruch wieder einen normalen Bruch zusammen, so erhalten wir kürzere Brüche in der Nähe des Originals:

$$[1{,}2{,}3{,}5] = \frac{53}{37} \;;\; \left|\frac{1355}{346} - \frac{53}{37}\right| < \frac{1}{10{,}000}$$

Es lässt sich zeigen, dass Kettenbrüche die besten Näherungen für die Originalbrüche sind und in der k-ten Näherung die Qualität

$$\left|\frac{a}{b} - \frac{a_k}{b_k}\right| < \frac{1}{b_k^2}$$

besitzen. Es lässt sich somit sowohl eine Optimierung zu bestimmten Längen als auch zu bestimmten Genauigkeiten durchführen. Für eine vorgewählte Genauigkeit lautet der Algorithmus

```
GanzeZahl zz(rz.zaehler),nn(rz.nenner);
GanzeZahl A0, A1(DivideR(zz,nn)), A2(1);
GanzeZahl B0, B1(1), B2, bk;
_swap(zz,nn);
RationaleZahl res(A1,B1),t;

while ((zz!=0) && (_abs((res-rz)) > diff)){
```

[17] Wurzeln lassen sich auf unendliche periodische Kettenbrüche zurückführen. Kettenbrüche lassen sich daher auch zur optimalen Approximation von Wurzeln einsetzen.

```
    bk = DivideR(zz,nn);
    _swap(zz,nn);
    A0=bk*A1+A2;
    B0=bk*B1+B2;
    _swap(A0,A1); _swap(A0,A2);
    _swap(B0,B1); _swap(B0,B2);
    res.zaehler=A1;
    res.nenner=B1;
};/*endwhile*/
```

`DivideR` ist eine Kombination aus **/=** und **%=** und liefert als Rückgabewert den Faktor sowie im ersten Parameter den Divisionsrest.

Die gleiche Entwicklung lässt sich natürlich auch für rationale Funktionen durchführen. Allerdings muss man hierbei genau wissen, was man damit bezwecken will. Betrachten wir beispielsweise die Funktion

$$q(x) = \frac{(x-0.5)*(x-0.6)*(x-0.7)}{(x-0.4)*(x-0.65)*(x-0.75)*(x-0.85)}$$

so erhalten wir nach einer Kettenbruchentwicklung und Abbruch nach drei Glieder die Näherung

$$q_3(x) = \frac{(x-0.491\ldots)*(x-0636\ldots)}{(x-0.398\ldots)*(x-0.727\ldots)*(x-852\ldots)}$$

Ohne dass man hier prophetische Gaben besitzen muss, lässt sich feststellen, dass die Funktionen in der Nähe der (*meisten*) Nullstellen von Zähler und Nenner wenig miteinander zu tun haben. Eine generelle Näherung wie bei rationalen Zahlen kommt also bei rationalen Funktionen nicht in Frage.

Alles in allem wird deutlich, dass Rechnungen in Quotientenkörpern mit einem recht großen Aufwand verbunden sind. Multiplikationen und Divisionen bedürfen je zweier ganzzahliger Multiplikationen, Additionen und Subtraktionen jeweils dreier Multiplikationen und einer Addition, von mehr oder minder häufigen Kürzungsversuchen einmal abgesehen. Da selbst Additionen zu einem schnellen Anwachsen der Zahlengröße führen, die auch mit der Kettenbruchtechnik nur wieder sehr mühsam unter Verlust der Genauigkeit gezähmt werden kann, sind rationale Zahlen kein geeignetes Mittel für hochgenaue (*aber nicht absolut genaue*) technische Rechnungen.

15.3 Restklassenkörper

15.3.1 Theoretische Grundlagen

Die Konstruktion eines Körpers aus einem endlichen Ring führt zu einem Restklassenkörper. Die folgenden theoretischen Ausführungen sind noch knapper gehalten

als an anderen Stellen. Ich gehe davon aus, dass Sie sich im Gegensatz zu den anderen Algorithmen an diesen Problemkreis erst dann begeben, wenn Sie sich mit den möglichen Anwendungen und ihrer Theorie vertraut gemacht haben (*bei den anderen Algorithmen ist bezüglich der Anwendungen meist sehr viel mehr allgemein bekannt*). Wenn auch mehr als Rekapitulation gedacht, bieten die folgenden Ausführungen natürlich auch wieder eine Basis für einen experimentellen Einstieg mit nachfolgender theoretischer Auseinandersetzung.

Ausgangspunkt sind wieder euklidische Ringe (*eindeutige Division mit Rest*), nur fixieren wir nun N auf eine Primzahl p und fassen alle Z, die den gleichen Rest ergeben, zu einer Menge, der Restklasse r zusammen. Zu jeder Zahl N existieren dann N verschiedene Restklassen, und mathematisch kennzeichnet man die Zugehörigkeit einer Zahl zu einer bestimmten Restklasse durch

$$z \equiv r \, mod \, N$$

und nennt N das Modul einer Restklassenmenge. Einfache Überlegungen und Versuche zeigen, dass unter der Nebenbedingung $N = p$ mit beliebigen Vertretern von Restklassen gerechnet werden darf:

$$Z_1 + Z_2 \equiv r_1 + r_2 \equiv r \, mod \, p$$
$$Z_1 * Z_2 \equiv r_1 * r_2 \equiv r \, mod \, p$$

Stellt man sich eine Multiplikationstabelle zusammen, so wird auch schnell klar, dass die Division immer möglich ist, das heißt zu gegebenen a,b immer ein x existiert, so dass $a \equiv b * x \, mod \, p$ gilt. Der euklidische Algorithmus für den ggT lässt sich sogar zu einer Form erweitern, die eine direkte Berechnung des x erlaubt. Dazu halten wir zunächst fest, dass für alle Restklassen $\neq$ Null $ggT(r,p) = 1$ gilt. Die letzte Gleichung im ggT–Algorithmus können wir nun auch so formulieren:

$$1 = (1) * r_n + q_n * r_{n-1}$$

Setzen wir r_n rekursiv in den `ggT`–Algorithmus ein, bis die erste Gleichung erreicht ist, so erhalten wir

$$1 = u * r + v * p \quad \Rightarrow$$
$$1 \equiv u * r \, mod \, p \quad wegen \quad v * p \, mod \, P = 0$$

Dieser Algorithmus liefert uns somit das Inverse zu einer Restklasse r und die Division ist nun durch

$$a/r \equiv a * u \, mod \, p$$

ohne Hilfe einer Tabelle lösbar.

Aufgabe. Führen Sie den rekursiven Beweis vollständig durch.

Aus der Rekursion lässt sich folgender Algorithmus für die Berechnung des Zahlenpaars u,v konstruieren: Wir deklarieren zwei Vektoren

$$\vec{W}_0 = \begin{pmatrix} b \\ 1 \\ 0 \end{pmatrix} \quad , \quad \vec{W}_1 = \begin{pmatrix} a \\ 0 \\ 1 \end{pmatrix} \quad , \quad b > a$$

und berechnen iterativ

$$q_k = [W_{k-1,1}/W_{k,1}] \; ; \; \vec{W}_{k+1} = \vec{W}_{k-1} - q_k * \vec{W}_k$$

Bei $q_k = 1$ (*der größte gemeinsame Teiler*) ist das Ende erreicht und es gilt (*allgemein, das heißt auch bei* $ggT(a,b) \neq 1$)

$$ggT(a,b) = W_{n,1} = b * W_{n,2} + a * W_{n,3}$$

Aufgabe. Setzen Sie den Algorithmus in ein Programm um.

15.3.2 Implementation der Restklasse

Damit haben wir nun alle Requisiten für die Implementation einer Vorlagenklasse für Restklassenrechnungen bereitgestellt. Innerhalb eines Problemkreises beziehen sich alle Rechnungen immer auf ein Modul, jedoch sind in Anwendungen die Problemkreise häufiger zu wechseln. Meist ist das mit einem Funktionsaufruf verbunden, wobei innerhalb der Funktion ein anderes Modul verwendet wird als in der rufenden Funktion, beim Rücksprung das ursprüngliche Modul jedoch wieder herzustellen ist. Wir tragen dieser Systematik Rechnung und legen das Modul deshalb in Form einer statischen Stackvariablen an. Dabei machen wir Gebrauch von unseren Untersuchungsergebnissen zu den Ankerobjekten interner Referenzen und implementieren den Stack so, dass externe statische Variable nicht deklariert werden müssen.

```
template <class T> class Restklasse {
    ...
    inline const T& Modul()const {
        if(MStack().empty())
            return algebra<T>::eins();
        return MStack().top();
    };//end function

    typedef stack<T> MODUL;

    static inline MODUL& MStack(){
        static MODUL modul;
```

```
        return modul;
    };//end function

    T value;
};//end class
```

Der restliche Aufbau ist ähnlich trivial wie bei der Klasse `Rational`, so dass ich mich hier auf das Beispiel „Division“ beschränke

```
...
inline Restklasse<T>& operator/=(const T& m)   {
    T op1,op2;
    if(Euklid(Modul(),m,op1,op2)!=1){
        *this=algebra<T>::null();
    }else{
        value*=op2;
        reduce();
    }//endif
    return *this;
};//end function

inline void reduce(){
    if(modul.empty())
        return;
    value%=modul.top();
    if(value<0){
        value+=modul.top();
    }//endif
};//end function
```

Aufgabe. Implementieren Sie die Vorlagenklasse vollständig.

Als Besonderheit gegenüber den anderen Klassen ist anzumerken, dass die Implementation der Vergleichsoperators `bool operator<(..)` keinen Sinn macht. Denken Sie darüber nach, warum!

In der gleichen Weise wie auf dem Ring der ganzen Zahlen lässt sich die Restklassenrechnung auch auf einem Ring von Polynomen durchführen. Während der Begriff „Primzahl“ im allgemeinen Verständnis gut verankert ist (*wenn es auch nicht ganz einfach ist, große Primzahlen zu finden*), ist der äquivalente Begriff „Primpolynom“ etwas diffiziler. Wir wollen darauf jedoch nicht weiter eingehen: Wenn in Rechnungen ausschließlich Multiplikationen verwendet werden, genügt auch die Einschränkung, dass nur zum Modul teilerfremde Restklassen eingesetzt werden, um alle Rechnungen eindeutig zu machen. Für tiefergehende Detail verweise ich Sie auf die Theorie.

15.4 Fliesskommazahlen

15.4.1 Grundlagen

Da die Grundkonstruktion der Fließkommazahlen oder „floating point values“ Gegenstand der einführenden Veranstaltungen in der Informatik ist, frische ich hier nur kurz das Gedächtnis aus: Fließkommazahlen besitzen eine fest eingestellte Genauigkeit von m Bit und werden symbolisch durch das Polynom

$$Z = vz * \sum_{k=1}^{m} z_k * B^{E-k}$$

mit negativen Potenzen dargestellt. Der Faktor B^E schiebt bei Zahlen beliebiger Größe das Komma der Ziffernfolge auf eine beliebige Position, wobei der Exponent E eine vorzeichenbehaftete ganze Zahl ist. z_k sind die Ziffern der Potenzreihe, vz ist das Vorzeichen der Zahl. Der Wertebereich des Exponenten E gibt den Zahlenbereich an, der mit dem Zahlentyp darstellbar ist, die Anzahl der Ziffern (*die Mantisse*) die Genauigkeit, mit der eine Zahl gespeichert werden kann.

Der Standarddatentyp double besteht beispielsweise aus 64 Bit, von denen 53 auf die Mantisse, 1 auf das Vorzeichen und 11 auf den Exponenten entfallen.[18] Damit lassen sich Zahlen im Bereich

$$10^{-308} \leq |z| \leq 10^{308}$$

mit etwa 16 Stellen Genauigkeit darstellen.

Die Potenzreihe enthält je nach darzustellender Zahl positive und negative Potenzen. Ganze Zahlen lassen sich unter der Voraussetzung, dass die Anzahl der zur Verfügung stehenden Bits groß genug ist, auch mit diesem Datentyp exakt darstellen. Das gilt nicht mehr für Zahlen mit gebrochenen Anteilen: Im Gegensatz zu den rationalen Zahlen lassen sich nämlich bereits die meisten Eingabewerte nicht mehr exakt darstellen, egal wieviele Stellen verwendet werden. Soll beispielsweise eine Zahl $0{,}xxx_A$ zur Basis A in eine Zahl zu Basis B umgewandelt werden, so lässt sich dies mit dem (*ganzzahlig zu betreibenden!*) Algorithmus

```
vector<ulong> k,Z;
k.push_back(xxx) ;
do {
    temp=k.back()*B;
    Z.push_back(temp/A^3);
```

[18] Das ist kein Schreibfehler: Die Zahlen werden mittels des Exponenten immer so normiert, dass das höchste Bit=1 ist. Wenn man das weiss, braucht man dieses Bit aber nicht mehr zu speichern. Auf diesen Trick ist man auch bei der Definition des Typs *double* gekommen, so dass auf 64 Bit insgesamt 65 Bit dargestellt werden..

```
    k.push_back(temp%A^3);
}while(k.back()!=0 &&
         find(k.begin(),k.back(),k.back()!=k.back());
```

durchführen. Das Feld Z enthält die Ziffernfolge in der neuen Basis (*pro Feldposition eine Ziffer*), die sich meist als periodisch erweist, das heißt das letzte Element von k wird nicht Null, sondern nimmt nach einiger Zeit einen bereits angenommenen Wert wieder an.

Aufgabe. Alle Größen im Algorithmus müssen dabei ganze Zahlen sein. Begründen Sie, warum dies so sein muss und nicht mit Fließkommazahlen gerechnet werden darf.

Die Perioden können erstaunlich lang ausfallen, zum Beispiel

$$0{,}234_{10} = 0{,}18\,\overline{560507534121727024368}$$

Da wir normalerweise im Dezimalsystem rechnen und Zahlen in dieser Form in den Rechner eingeben, Zahlen in Rechnersystemen aber im Binärformat verarbeitet werden, ist bereits der Eingabeschritt in den meisten Fällen nicht fehlerfrei. Die meisten Zahlen müssen bei der Speicherung auf die Maximalzahl der Ziffern gekürzt werden (*der o.g. Algorithmus eignet sich mit Begrenzung auf eine bestimmte Stellenanzahl zum Einlesen von Zahlenwerten*). Bei der Begrenzung ist ein Rundungsverfahren festzulegen, das auch bei den Rechenoperationen zum Einsatz kommt. Für die meisten Rundungsverfahren ist eine Besetzungsanalyse von „Zusatzbits“, die bei der Operation abgeschnitten werden, durchzuführen. Wir können folgende Arten der Rundung unterscheiden:

(a) Betragskleinster Wert: unabhängig vom Inhalt werden die Zusatzbits abgeschnitten;
(b) Betragsgrößter Wert: sind Positionen in den Zusatzbits gesetzt, wird grundsätzlich auf die nächste darstellbare Größe inkrementiert;
(c) Rundung nach $+\infty$: Entspricht im Positiven der Rundungsart „betragsgrößter Wert“, im Negativen der Rundungsart „betragskleinster Wert“;
(d) Rundung nach $-\infty$: Entspricht im Negativen der Rundungsart „betragsgrößter Wert“, im Positiven der Rundungsart „betragskleinster Wert“;
(e) kaufmännische Rundung (4/5–Rundung): Ist das erste wegfallende Bit gesetzt, wird nach $+\infty$ gerundet, ist es unbesetzt, nach $-\infty$.

Je nach Problemstellung kann die Rundungsvorschrift auch zwischen einzelnen Rechenvorgängen gewechselt werden (*wir werden dies im nächsten Kapitel untersuchen*).

15.4.2 Klassenkonstruktion

Damit können wir zur Konstruktion einer Klasse `BigFloat` kommen. Da in Rechnungen die Exponenten weitgehend getrennt von den Mantissen behandelt werden können, können wir sowohl hinsichtlich der Datenstrukturen als auch der Algorithmen kräftige Anleihen bei den ganzen Zahl tätigen. Für den Datenteil folgt somit.

```
struct __bfloat: Z_buf {
    int exponent;
    int precision;
    int rd_method;
    ...
};

class BigFloat{
    ...
    __bloat * dat;
};//end class
```

Das Attribut `precision` enthält die maximale Pufferlänge des Datenteils. Nach Rechenoperationen wird die Mantisse jeweils auf diese Größe unter Durchführung eines Rundungsvorgangs verkürzt, sofern sie überschritten wird. Um die Rundung implementieren zu können, sind noch einige Festlegungen notwendig:

- Das Attribut `exponent` legt fest, an welcher Bitposition relativ zum Pufferbeginn der Nullpunkt der Zahl liegt. Der Wert acht gibt dann beispielsweise an, dass die ersten acht Bit Nachkommastellen kodieren, die höheren Bit einen ganzzahligen Anteil; ein negativer Wert kennzeichnet die Kodierung einer großen Zahl, deren ganzzahliger Anteil bereits nicht mehr durch die Mantissenbits vollständig kodiert werden kann.
- Es gilt $\equiv 0 \, mod \, 8$, das heißt Zahlen werden byteweise und nicht bitweise kodiert. Kommt am größeren Ende ein Bit hinzu, so fallen bei der Rundung am unteren Ende acht Bits fort. Es werden somit nicht alle Bits zur Kodierung genutzt wie beim maschineninternen Typ `double`, jedoch spielt der verlorene Anteil bei der Gesamtgröße der Zahlen nur eine untergeordnete Rolle und wir sparen einigen Rechenaufwand, der bei bitweiser Verschiebung entstehen würde.
- Die Zahlen werden mit voller Länge gespeichert, das heißt kürzere Kodierungen werden jeweils wieder auf die Gesamtlänge aufgefüllt.[19]

[19] Das ist nicht unbedingt notwendig. Es kann auch mit der Bitanzahl gearbeitet werden, die jeweils gültig definiert ist. Ob die Verkürzung zu besseren Laufzeiten führt, hängt davon ab, welcher Anteil von Zahlen in einer längeren Rechnung tatsächlich weniger gültige Bits als die maximale Anzahl aufweisen. Voraussichtlich reicht das meist nicht, aber wenn Sie gegenteilige Meinung sind, müssen Sie in den Algorithmen entsprechende Anpassungen vornehmen.

Bei der Rundung muss der Exponent entsprechend angepasst werden. Die verschiedenen Rundungsmethoden erfordern eine Reihe von Fallunterscheidungen, der Rundungsvorgang muss bei einer Aufrundung auch gegebenenfalls auftretende Überläufe berücksichtigen. Die Methode wird mit dem ersten fortfallenden Byte (*oder Null, falls keines vorliegt*) aufgerufen. Ergänzen Sie die folgende Methode auch für Vergrößerung des Puffers.

```
inline void Round(uchar val){
    ...
    if(dat->len > dat->precision){
        li=dat->len - dat->precision;
        val=dat->buf[li-1];
        memmove(dat->buf,&dat->buf[li],
                          dat->precision);
        dat->len-=li;
        dat->exponent -= (8*li);
    }//endif
    ...
    if(dat->rd_method==0 && val>=0x80){
        do{
            _inc(static_cast<Z_buf*>(dat));
            if(dat->len <= dat->precision){
                return;
            }//end
            val=dat->buf[0];
            memmove(dat->buf,&dat->buf[1],
                              dat->precision);
            --dat->len;
           dat->exponent+=8;
        }while(val>=0x80);
        return;
    }//endif
    ...
};//end function
```

Die Rundungsmethode ist zum Abschluss jeder arithmetischen Operation aufzurufen. Für die arithmetischen Operationen nutzen wir die bei den ganzen Zahlen entwickelten Algorithmen. Am leichtesten ist wohl die Multiplikation umzusetzen, bei der die Mantissen wie ganze Zahlen miteinander multipliziert und die Exponenten addiert werden. Die bei der Multiplikation entstehende Ergebnismantisse besitzt die doppelte Breite einer normalen Mantisse, und nach Durchführung der Rundung wird die hintere Hälfte einfach entfernt.

Aufgabe. Implementieren Sie eine Methode zur Multiplikation zweier Zahlen.

15.4.3 Addition und Subtraktion

Bei Additionen oder Subtraktionen sind die Operanden zunächst auf den gleichen Exponenten zu normieren (*in dieser Darstellung stehen links die Bits mit kleinen negativen Exponenten, rechts die mit dem größten*):

```
xxxxxxxxxxxxxxxxxxxx E +e(1)    +
yyyyyyyyyyyyyyyyyyyy E +e(1)-3)   -->

xxxxxxxxxxxxxxxxxxxx E +e(1)    +
000yyyyyyyyyyyyyyyyyYYY E +e(1)
```

Die betragsmäßig kleinere Zahl wird so weit nach rechts verschoben (*und der Exponent dabei vergrößert*), bis die Exponenten übereinstimmen. Dabei rutschen eine Anzahl von Bits in den abzuschneidenden Bereich und werden nach Rundung auf die letzte verbleibende Stelle gelöscht. Kritisch hinsichtlich der Anzahl der gültigen Bits ist die Subtraktionen gleich großer Zahlen:

```
abcdefghjiklxxx  E +e   -
abcdefghijklyyy  E +e      -->

000000000000zzz  E +e      -->
zzz000000000000  E +e-12
```

Auch dieses Zahl wird nach der Berechnung wieder normiert, das heißt so weit nach links verschoben, dass die ersten Bitpositionen besetzt sind. Die hinteren sind jetzt natürlich Null, wobei man den Zahlen natürlich nicht ansieht, dass die Nullen „erdichtet“ sind. Wären die Daten in einer Darstellung mit größerer Mantisse beispielsweise

```
abcdefghjiklxxx111101011001  E +e   -
abcdefghijklyyy000100001000  E +e      -->

zzz111001010001 E +e
```

gewesen, so überzeugt das erste Ergebnis natürlich nicht sonderlich. Je länger die Ziffernfolgen einer Zahl sind, desto weniger Probleme wird man mit solchen Effekten haben, da gleiche Ziffernfolgen (*außer wenn sie theoretisch gefordert sind, aber dann sind rationale Zahlen geeigneter*) mit steigender Länge weniger wahrscheinlich werden und der Rechner dadurch natürlich weniger Gelegenheit bekommt, sich den größten Teil einer Zahl selbst auszudenken. Wie man mit solchen Fehlern quantitativ umgehen kann, untersuchen wir im nächsten Kapitel.

Aufgabe. Implementieren Sie nun auch Addition und Subtraktion. Unterscheiden Sie die verschiedenen Vorzeichenfälle und die Exponentenfälle $e_1 < e_2, e_1 = e_2, e_1 > e_2$. Eine Verschiebung der Mantissen ist nicht notwendig, wenn Sie Hilfsobjekte deklarieren, die übergebene Adressen und Längen auf `Z_buf` abbilden. Definieren Sie dazu eine Struktur `Z_temp`, die von `Z_buf` erbt und auf die Erzeugung eigenen Speichers verzichtet, sowie statische Variablen für die Arbeit.

15.4.4 Division i

Es bleibt noch die Division zu implementieren, und hierfür wollen wir drei Algorithmen angeben. Zunächst modifizieren wir den Algorithmus für die ganzzahlige Division für den Einsatz bei Fließkommazahlen: Indem wir den Dividenden (*genauer die Mantisse; die Exponenten müssen ähnlich wie bei der Multiplikation nur voneinander subtrahiert werden*) vor der Durchführung der Division um eine volle Pufferbreite nach links schieben, erhalten wir ein Ergebnis einer ganzzahligen Division, das die volle Mantissenbreite unserer Zahlen aufweist. Gilt $rest > divisor/2$, so ist das Ergebnis noch aufzurunden:

```
1234 : 4321  -->
1234 0000 : 4321 =  2855 Rest 3545 = 2856
```

15.4.5 Division ii

Eine Division ist somit mit einem relativ hohen Aufwand verbunden. Einen alternativen Algorithmus für sehr große Mantissen erhalten wir, wenn wir ähnlich der Karatsuba–Multiplikation die Zahl in zwei Hälften zerlegen und den Quotienten umformen, um die Aufgabe auf Zahlen mit weniger vielen signifikanten Ziffern zu reduzieren:

$$\frac{u_h + B * u_l}{v_h + B * v_l} = \frac{u_h + B * u_l}{v_h} \left(\frac{1}{1 + B * (v_l / v_h)} \right)$$

Der Index h gibt das betragsmäßig höhere Halbwort an, das heißt die Koeffizienten von 2^{-1},2^{-2},... $B = 2^{-WORDLEN/2}$ ist der Faktor für die halbe Wortbreite. Entwickelt man den Klammerausdruck in eine Tailorreihe nach $B * (v_l/v_h)$, so erhält man

$$\frac{u}{v} = \frac{u_h + B * u_l}{v_h} \left(1 - B * (v_l/v_h) + B^2 * (v_l/v_h)^2 - \ldots \right)$$

Aufgrund unserer Vereinbarung über die Darstellung der Zahlen gilt $(v_l/v_h) < 2^8$, und die Terme in der Reihenentwicklung werden schnell kleiner, so dass nur der erste ausgewertet werden muss. Sehen wir uns den Aufwand für diesen Algorithmus an (*die Additionen lassen wir als unerheblich fort*):

- Der Divisor besitzt nur noch die halbe Mantissengröße der Zahlen.
- Es sind zwei Divisionen durchzuführen, bei denen der Dividend die volle Mantissengröße besitzt.
- Es ist eine Division durchzuführen, in der der Dividend die 1,5–fache Mantissengröße besitzt.
- Es ist eine normale Multiplikation durchzuführen.

Es müssen zwar mehr Einzeloperationen ausgeführt werden, aber der Aufwand steigt bei zunehmender Mantissengröße deutlich weniger stark an, so dass dieser Algorithmus ab einer bestimmten Mantissengröße günstiger wird als der einfache Algorithmus. Wie bei der Karatsuba–Multiplikation kann die Division wieder rekursiv durchgeführt werden, bis die Grenze für die Durchführung mit dem normalen ganzzahligen Divisionsalgorithmus erreicht ist. Da wir es immer mit der gleichen Mantissengröße während einer Rechnung zu tun haben, kann der Wert fest eingestellt werden. Der Ablauf sei an einem Zahlenbeispiel demonstriert. Bei vierstelliger Dezimalrechnung ohne Berücksichtigung der Exponenten erhalten wir:

```
0. direkte Rechnung:
22 31 : 15 29 = 14 59

1. Term:
22 00 : 15 = 14 67
   31 : 15 =    21 ,  Summe  14 88

2. Klammer mit einem Entwicklungsglied
10 00 - 29 : 15 = 98 06

3. Produkt und Ergebnis
14 88 * 98 06 = 14 59
```

Aufgabe. Implementieren Sie die Algorithmen. Die Rekursion im zweiten Algorithmus lässt sich bis zur Ausnutzung der (*ganzzahligen*) Hardwaredivision der Maschine ausnutzen, so dass auf die Algorithmen für die ganzzahlige Division sogar verzichtet werden kann. Bestimmen Sie die Ordnungen der Algorithmen und legen Sie für verschiedene Rechengenauigkeiten jeweils die optimalen Algorithmen fest.

Hinweis. Hier können Sie sich großzügig auf einige Fälle beschränken. Der „Standard“ für Berechnungen mit Fließkommazahlen liegt bei acht Byte Zahlenbreite „über alles“, also einschließlich Exponent und Vorzeichen. Von vielen Maschinen oder Sprachen unterstützt wird noch `real*16`, womit Sie anfangen können. Wenn Sie dann noch `real*32, real*64` und `real*128` untersuchen, haben Sie genügend Datenpunkte für die Lösung der Aufgabe und vermutlich schon größere Zahlen, als man je brauchen kann. Kleinere Stufungen sind allerdings relativ sinnlos. Wenn man wirklich mit einem Format an die Grenze der Rechnungen stößt, ist weniger als eine Verdoppelung der Mantissengröße meist wenig sinnvoll.

15.4.6 Division iii

In einem dritten Algorithmus vermeiden wir die Division vollständig, indem wir sie durch eine Multiplikation mit dem Inversen ersetzen und dieses iterativ ermitteln.

$$a/b \to a * b^{-1} \ ; \ b * b^{-1} = 1$$

Für die Iterationsfunktion für die Berechnung des Inversen nach dem Newtonverfahren findet man nach einigen Versuchen:

$$f(x) = \frac{1}{x*b} - 1$$
$$\Phi(x) = x - \frac{f(x)}{f'(x)} = 2 * x - b * x^2$$

Sie enthält wunschgemäß keine Division mehr und konvergiert, wenn man sich von kleineren Werten her dem Inversen nähert:

```
b = 2.345   -->  x[0] = 0.2345

x[1]=0.3400478637     x[2]=0.4089373986
x[3]=0.4257209256
x[4]=0.4264380225     x[5]=0.4264392325
x[6]=0.4264392324

==>  x[∞]=x[5]
```

Die Iteration ist beendet, wenn der neue Wert nicht größer als der alte ist. Bei jedem Iterationsschritt verdoppelt sich, wie beim Newton–Verfahren bei einfachen Nullstellen üblich, die Anzahl der signifikanten Stellen, so dass bei guten Startwerten nur wenige Schritte ausreichen.

> **Aufgabe.** Implementieren und Testen Sie diesen Algorithmus. Achten Sie bei den Startwerten darauf, dass dieser stets kleiner sein muss als der Fixpunkt! Testen Sie das Verfahren im Vergleich mit den anderen Algorithmen. Da der Algorithmus bei Überschreiten des Fixpunktes fortläuft, kann bei Instabilitäten auch das Rundungsmodell geändert werden.

15.4.7 Relationen

Bei Vergleichen treten eine ganze Reihe von Fallunterscheidungen auf, im Fall der Prüfung auf Gleichheit

```
inline bool operator==(const BigFloat& b1,
                       const BigFloat& b2){
    if(b1.dat->neg != b2.dat->neg)
        return false;
```

```
    if(b1.dat->exponent !=b2.dat->exponent)
        return false;
    if(b1.dat->len != b2.dat->len)
        return false;
    if(b1.dat->len>0)
        return memcmp(b1.dat->buf,b2.dat->buf,
                              b1.dat->len)==0;
    else
        return true;
}//end function
```

Implementieren Sie entsprechend `bool operator<(..)`. Auch die Implementierung der Methoden `bool is_less(..), bool is_null(..)` nebst den notwendigen Konstanten überlasse ich Ihnen.

Die Verrechnung der langen Fließkommazahlen mit ganzen Zahlen ist problemlos zu lösen, da beide Zahlenmodelle die gleichen Datenstrukturen verwenden. Werte des Maschinentyps `double` oder anderer Maschinentypen können mit Hilfe zweier Funktionen der C–Bibliothek auf lange Fließkommawerte übertragen werden. Der folgende Algorithmus überträgt die Daten bitweise und ist deshalb nicht unbedingt innerhalb von Schleifen einzusetzen.[20]

```
BigFloat& BigFloat::operator=(const double& d){
    uchar mant; int exp; double r(d);
    ...              // Initialisierung
    r=frexp(r,&dat->exponent);
    exp=0;
    while(r!=0 && dat->len < dat->precision){
        exp++;
        r=ldexp(r,1);
        mant=static_cast<uchar>(r);
        r-=mant;
        if(dat->len>0)
            _shl(static_cast<Z_buf*>(dat),1);
        else{ ... }//endif
        dat->buf[0]|=mant;
    }//end
    dat->exponent=exp - dat->exponent;
    ...    // Normierung auf volle Mantissenlänge
    return *this;
}//end function
```

[20] Man kann natürlich auch mehr als ein Bit pro Durchlauf anfordern, muss dann aber kontrollieren, wie viele tatsächlich ankommen. Bei genauer Kenntnis der Maschinenimplementation ist natürlich auch ein direkte möglich, aber dann für jeden Maschinentyp getrennt.

Für die Konvertierung eines Strings in eine lange Fließkommazahl wird zunächst die Ziffernfolge in eine ganze Zahl übersetzt und diese auf eine Fließkommavariable übertragen. Mit dem Absolutbetrag des Exponenten oder der Position des Kommas wird anschließend $10^{|e|}$ berechnet und je nach Vorzeichen die gespeicherte ganze Zahl mit dem Wert multipliziert oder durch den Wert dividiert. Die Ausführung überlasse ich Ihnen.
Bei der Konvertierung einer langen Fließkommazahl in einen String ist zwischen dem ganzzahligen und dem gebrochenen Anteil zu unterscheiden. Der ganzzahlige Anteil lässt sich nach Verschieben der Mantissenbits auf eine ganze Zahl und gegebenenfalls Linksschieben entsprechend des Exponenten durch den Konvertierungsalgorithmus der ganzen Zahlen berechnen. Auch für den gebrochenen Anteil können ganze Zahlen zum Einsatz kommen, in dem Potenzen von fünf berechnet und nach Linksschieben der kleineren Potenzen addiert werden:

```
template <> string toString(const BigFloat& n){
    GanzeZahl z5,z;
    z=1; z5=1;
    ...
    for(i=n.dat->exponent-1;i>=0;--i){
        z=z*10;
        z5*=5;
        if(i >= 8*n.dat->len)
            continue;
        if(bit_test(*n.dat->buf,i)){
            z+=z5;
        };
    }//endfor
    u=val2str(z);
    ...
    return s;
}//endfunction string
```

Die Verknüpfung mit dem ganzzahligen Anteil sowie die Schönungsarbeiten für die Ausgabe seien wieder Ihnen überlassen.

15.4.8 Reelle Konstanten und Funktionen

Für die Arbeit mit dem neuen Zahlentyp sind nun noch einige Konstante sowie Funktionen notwendig, die üblicherweise in numerischen Anwendungen benötigt werden. Diese müssen in der vollen Mantissenbreite bereitgestellt und können nicht einfach von den Maschinentypen übernommen werden.[21] Unverzichtbar sind

[21] Sofern es sich um iterative Algorithmen wie beispielsweise die Berechnung einer Wurzel handelt, können die maschinengenauen Daten als Startwerte für die Iteration verwendet werden. Die Iterationen sind dann nach wenigen Schritten zu Ende.

zunächst Werte für und e . Für lautet der aktuell wohl beste Algorithmus von Bailey, Borwein und Pouffle:

$$\pi = \sum_{k=0}^{\infty} \left(\frac{4}{8*k+1} - \frac{2}{8*k+4} - \frac{1}{8*k+5} - \frac{1}{8*k+6} \right) * \left(\frac{1}{16} \right)^k$$

Für e wird zweckmäßigerweise die Taylorreihe von $exp(x)$ verwendet:

$$e = \sum_{k=0}^{\infty} \frac{1}{k!}$$

Beide Reihen konvergieren recht schnell. Die Daten werden beim Start der Anwendung oder bei Vergrößerung der Mantisse berechnet und als „Konstante“ in statischen Variablen der Klasse `Constant>BigFloat>` gespeichert. Die Algorithmen summieren gliedweise auf und brechen ab, wenn sich nach Hinzufügen eines weiteren Summengliedes nichts mehr geändert hat (*Sie können als Übung mathematisch Überprüfen, wie groß der maximale Fehler bei dieser Vorgehensweise ist*).

Wichtige Funktionen sind $\sin(x), \cos(x), \ln(x), \exp(x)$ und weitere damit verwandte Funktionen. Aufgrund der wechselnden Mantissengröße bleiben für die Berechnung von Funktionswerten im Grunde nur die Taylorreihen, die nur dann schnell konvergieren, wenn die Ordinate des gesuchten Punktes in der Nähe der Entwicklungsstelle liegt. Das lässt sich durch Ausnutzung von Symmetrieeigenschaften bei den trigonometrischen Funktionen leicht erreichen. Die Berechnung kann nämlich reduziert werden auf

$$\sin(x) = \sin(f_1(x \bmod \pi/2, x \bmod \pi)) * sign(x) * f_2(x \bmod 2*\pi)$$
$$\cos(x) = \cos(f_1(x \bmod \pi/2, x \bmod \pi)) * f_3(x \bmod 2*\pi)$$

Die Sinusfunktion ist eine ungerade Funktion ($\sin(x) = -\sin(-x)$), die Kosinusfunktion eine gerade Funktion. Werte modulo einer reellen Zahl werden ähnlich wie bei den ganzen Zahlen durch den Algorithmus

```
BigFloat fmod2(BigFloat& f, const BigFloat& modul){
    BigFloat r;
    ...
    r=(f/=mod);
    if(f.dat->exponent >= 8*f.dat->len){
        f=0;
    }else if(f.dat->exponent >= 0){
        f.dat=LFAlloc::allocator.Copy(f.dat);
        memset(f.dat->buf,0,f.dat->exponent/8);
    }//endif
    r-=f;
    r*=mod;     /**/
    ...
```

```
    return r;
}//end function
```

berechnet, wobei die Anweisung /**/ vom gebrochene Teil der Division auf einen „Divisionsrest“ transformiert. Für die Funktionen $f_1(a,b)$,$f_2(a)$,$f_3(a)$ verifiziert man unschwer anhand der Funktionsgraphen

$$f_1(a,b) = \begin{cases} a & , \ b \in (0,\pi/2) \\ \pi/2 - a & , b \in (\pi/2,\pi) \end{cases}$$

$$f_2(a) = \begin{cases} 1 & , \quad a \in (0,\pi) \\ -1 & , \quad a \in (\pi,2*\pi) \end{cases}$$

$$f_3(a) = \begin{cases} 1 & , a \in (0,\pi/2) \cup (3*\pi/2,2*pi) \\ -1 & , \qquad a \in (\pi/2,3*\pi/2) \end{cases}$$

Aufgabe. Implementieren Sie die trigonometrischen Funktionen mit Hilfe der Taylorreihenentwicklungen

$$\sin(x) = \sum_{k=0}^{\infty} (-1)^k * \frac{x^{2k+1}}{(2k+1)!}$$
$$\cos(x) = \sum_{k=0}^{\infty} (-1)^k * \frac{x^{2k}}{(2k)!}$$

Die Iteration kann wie bei der Berechnung von und e fortgesetzt werden, bis der Summenterm konstant bleibt. Weitere Funktionen können Sie nach Bedarf implementieren. Von den Umkehrfunktionen sind `arcsin(x)` und `arccos(x)` ebenfalls mit Hilfe von Taylorreihenentwicklungen implementierbar (*Definitionsbereich* [-1,1]), `arctan(x)` aber nicht mehr. Hier hilft eine Newtoniteration weiter, wobei der Startwert mit der Standard–math–Bibliothek ermittelt wird.

Die zweite auf der Konstanten e basierende Funktionengruppe mit `ln(x)` und `exp(x)` lässt sich auf ähnliche Art wir die trigonometrischen Funktionen berechnen. Die Taylorreihen lauten

$$e^x = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$
$$\ln(x) = \sum_{k=0}^{\infty} \frac{(x-1)^{2k}}{2k*(x+1)^{2k}}, x < 1$$

Die Exponentialfunktion zerlegen wir in

$$e^x = e^{[x]} * e^{x \ mod \ 1}$$

wobei mit [x] die nächstkleinere ganze Zahl ist (*Vorzeichen beachten!*). Der erste Term wird mit Hilfe der Funktion `power(..)` ausgewertet, der zweite konvergiert wegen `0<=x<1` bei der Reihenauswertung sehr schnell. Die gleiche Art der Auswertung kann auf den Logarithmus angewandt werden: Mit Hilfe des Algorithmus

```
if(algebra<BigFloat>::e()<=x){
    sf.push(algebra<BigFloat>::e());
    xp=1;
    while(sf.top()<x){
        sf.push(sf.top()*sf.top());
        xp*=zwei;
    }//endwhile
    sf.pop();
    xp/=zwei;
    while(algebra<BigFloat>::e()<=x &&
                          !sf.empty()){
        x/=sf.top();
        add+=xp;
        sf.pop();
        xp/=zwei;
    }//endwhile
}else if(x<=inverse(algebra<BigFloat>::e())){
    ...
```

werden die ganzzahligen Potenzen von e im Argument entfernt (*der zugehörende Anteil von* `ln(x)` *befindet sind in der Variablen* `add`, *setzen Sie den Algorithmen für kleine x fort*). Das verbleibende Restargument erfüllt die Konvergenzbedingungen und der Logarithmus kann durch Iteration ermittelt werden.

15.4.9 Interpolation von Werten

Die Auswertung von Reihen zur Berechnung von Funktionswerten kann für die praktische Anwendung schnell zu zeitaufwendig werden. Auf hinreichend kleinen Intervallen lassen sich aber auch Polynome mit kleinen Graden verwenden. Für eine Reihe von Mantissengrößen kann man sich Sätze von Polynomen erzeugen, die in den Anwendungen dann nur noch geladen werden müssen und eine schnellere Auswertung der Standardfunktionen erlauben.

Um einen Satz von Näherungspolynomen zu erzeugen, ist zunächst eine mathematische Untersuchung der auszuwertenden Funktion sinnvoll. Der Polynomgrad kann nämlich von Intervall zu Intervall variieren: Die Sinusfunktion ist beispielsweise in der Nähe der Null annähernd linear und wird am Maximum näherungsweise durch eine Parabel beschrieben. Man kann also je nach Lage des Teilintervalls unterschiedliche Polynomansätze machen.

Nach Festlegen des Polynomgrades für einen bestimmten Bereich wird ein Intervall `[a,b)` für die Gültigkeit eines Polynoms festgelegt und mittels der Reihen hochgenaue Punkte im Intervall berechnet.

$$\begin{aligned} &\textit{Polynom}: \quad P_n(x) = \sum_{k=0}^{n} a_k * x^k \\ &\textit{Stützstellen}: a \le x_0 < x_1 < \ldots < x_{n-1} < x_n \le b \\ &\textit{Punkte}: \quad (x_0,y_0),(x_1,y_1),\ldots(x_n,y_n) \end{aligned}$$

Die Koeffizienten des Polynoms $P_n(x)$ erhält man am einfachsten durch Auswertung von Lagrange–Polynomen:

$$P_n(x) = \sum_{k=0}^{n} y_k \prod_{\substack{j=0 \\ j\neq k}}^{n} \frac{(x - x_j)}{(x_k - x_j)}$$

Das so berechnete Polynom kann nun auf seine Qualität überprüft werden, in dem mit ihm ermittelte Funktionswerte mit hochgenauen Zwischenwerten verglichen werden. Wird die erforderliche Qualität nicht erreicht, können die Stützstellen verschoben, das Intervall verkleinert oder der Polynomgrad (*zusammen mit der Anzahl der Stützstellen*) erhöht werden, bis man mit dem Ergebnis zufrieden ist. Zweckmäßigerweise hangelt man sich sukzessive von einem Intervallrand zum anderen und speichert die gefundenen Polynome in einer `map` ab.

```
typedef map<BigFloat,Polynom<BigFloat> > IMap;
IMap imap;
BigFloat a,b; Polynom<BigFloat> p;
...
imap.insert(IMap::value_type(a,p));
```

Speicherung und Laden der Daten überlasse ist Ihnen. In der Anwendung ist die Berechnung eines Funktionswertes dann recht einfach:

```
BigFloat function(const BigFloat& x){
    ...
    return imap.lower_bound(x)();
}//end function
```

15.5 Die Körper F_{2^m}, F_{p^m}

In der Verschlüsselungstechnik werden neben Restklassenkörpern des Typs $F_p = \mathbb{Z}/p*\mathbb{Z}$ auch solche des Typs F_{p^m} eingesetzt. Ist mit F_p ein Körper der Charakteristik $Char(F_p) = p$ (*das heißt* $p*1 = 0$) und $\#F_p^* = p-1$ gemeint, so gilt $Char(F_{p^m}) = p$ und $\#F_{p^m}^* = p^m - 1$. Die ausführliche Theorie müssen wir Lehrbüchern der Algebra

überlassen; wir werden uns hier nur um die Darstellung der Elemente solcher Körper auf Rechnern annehmen.

Körper mit den gewünschten Eigenschaften lassen sich als Vektorräume über dem Grundkörper F_p konstruieren. Dazu wählen wir ein überF_p irreduzibles Polynom $P_m(x)$vom Grad n. Irreduzibel bedeutet, dass $P_m(x)$ keine Nullstelle in F_p besitzt. Wir können nun eine iterierte Restklasse definieren, die die geforderten Eigenschaften besitzt:

```
typedef Restklasse<GanzeZahl > Fp;
typedef Restklasse<Polynom<Fp> > Fpm;
...
Fp::ModulMonitor mp;
Fpm::ModulMonitor mpm;
...
mp.set( /* p */ );
mpm.set( /* P_m(x) */ );
```

Aufgabe. Verifizieren Sie: mit dieser Definition erhalten wir einen Körper mit den gewünschten Eigenschaften.

Eigentlich könnte man nun meinen, das Kapitel sei an dieser Stelle bereits beendet. Es existieren jedoch einige Sonderfälle, für die spezielle Implementationen sinnvoll sind:

(a) Der Körper F_{2^m} wäre mit dieser Implementation völlig überfrachtet, da sein Grundkörper nur aus $F_2 = \{0,1\}$ besteht. Für einen Koeffizienten des Polynoms genügt daher ein Bit. Körper mit dieser Charakteristik kommen darübre hinaus in der Praxis überaus oft zum Einsatz.
(b) Es existieren Darstellungen mit zyklischen Basen $\left(\beta,\beta^2,\beta^3,\ldots\beta^m\right)$, bei denen auf die Restklassendarstellung verzichtet werden kann.β ist dabei eine m-te (*komplexe*) Einheitswurzel.

Beginnen wir mit b) im allgemeinen Fall $p \neq 2$. Wegen

$$\beta^k * \beta^l = \beta^{k+l \bmod m+1}$$

kann die Multiplikation auf einem Feld der Größe m durchgeführt werden (*anstatt 2m im normalen Restklassenfall, bei dem ja erst das Produkte ermittelt werden muss, bevor die Division mit Rest durchgeführt werden kann*).

Aufgabe. Ermitteln Sie den Algorithmus, nach dem die Multiplikation durchgeführt wird. Die Implementation für einen solchen Körper führen Sie bitte anhand der ausführlichen Darstellung im nächsten Kap. 12.6 durch. Sie brauchen nur $D = 1$ zu setzen. Für große m wird die Implementation mit `for`-Schleifen, wie im theoretischen Teil von Kap. 12.6 beschrieben, durchgeführt, für nicht zu große m wird im nächsten Kapitel auch eine effizientere Implementationsmethode vorgestellt.

15.6 Metaprogramme und Körpererweiterungen

15.6.1 Theoretische Vorbemerkungen

Zufällig ausgewählte Polynome besitzen bekanntlich nur in Ausnahmefällen Nullstellen im Körper der rationalen Zahlen, und man ist im allgemeinen Fall gezwungen, auf den Körper der reellen Zahlen mit seiner Darstellung durch Fließkommazahlen auf Rechnern auszuweichen. In einigen Anwendungen ist es aber notwendig, weiterhin exakt rechnen zu können, wobei im Gegenzug nicht alle, sondern nur einige ausgewählte Polynome Nullstellen im untersuchten Körper aufweisen müssen. In der Algebra wird dieses Problem dadurch gelöst, dass die Nullstellen nur formal an den Körper $\mathbb{Q}$ der rationalen Zahlen adjungiert werden, aber keine näherungsweise Darstellung verwendet wird.

Betrachten wir als Beispiel die Nullstellen des Polynoms

$$P(x) = x^3 - 2$$

Es hat eine reelle und zwei (*konjugiert*) komplexe Nullstellen. Um $\mathbb{Q}$ zu einem Körper zu erweitern, der die reelle Wurzel enthält, können wir folgendermaßen die Zahlen dieses Körpers definieren:

$$\mathbb{Q}\left(\sqrt[3]{2}\right) := \left\{ z = \left(a + b\sqrt[3]{2} + c\sqrt[3]{4} \right) \ , \ \ a, b, c \in \mathbb{Q} \right\}$$

Sollen alle Wurzeln des Polynoms im Erweiterungskörper enthalten sein, so ist der Körper $\mathbb{Q}(\sqrt[3]{2}, i)$ zu bilden, in dem jede Zahl insgesamt 6 Komponenten aufweist (*drei Komponenten wie oben sowie der gleiche Satz noch einmal, jedoch mit der imaginären Größe i als zusätzlichem Faktor*).

> **Aufgabe.** Weisen Sie durch direkte Rechnung nach, dass $\mathbb{Q}(\sqrt[3]{2})$ ein Körper ist, d.h. im Wesentlichen die vier Grundrechenarten definiert sind und die Ergebnisse wieder in der Menge $\mathbb{Q}(\sqrt[3]{2})$ liegen. Die Division ist dabei etwas aufwändig, und wir gehen weiter unten auf dieses Poblem genauer ein.
>
> Prüfen Sie ebenfalls direkt, dass $\mathbb{Q}(\sqrt[3]{2}, i)$ alle drei Wurzeln des Polynoms enthält.

Ohne dass wir uns nun weiter auf die algebraischen Zusammenhänge oder Algorithmen einlassen, in denen solche Zahlenmodelle benötigt werden, untersuchen wir nun deren Implementationsmöglichkeiten. Anhand des Beispiels können wir folgende Konstrutkionsprimzipien feststellen:

(a) Wenn eine Wurzel $\sqrt[n]{D}$ irgendeiner rationalen Zahl D im Erweiterungskörper vorhanden sein soll, so benötigt das Zahlenmodell n Komponenten, nämlich eine rationale Komponente sowie Komponenten mit den Wurzeln $\sqrt[n]{D}, \sqrt[n]{D^2}, \ldots \sqrt[n]{D^{n-1}}$, die bei einer Multiplikation von Zahlen entstehen.

(b) Wenn zwei verschiedene Wurzeln $\sqrt[n]{D}, \sqrt[m]{E}$ enthalten sein sollen, wird zunächst eine Zahl mit n Komponenten wie in (a) definiert, die wiederum Darstellungsbasis einer Zahl mit m Komponenten ist. Insgesamt enthält die Zahl $n * m$ Komponenten.

Unsere Zahlenkonstruktion ist mithin

```
template <class T, int n>
class ZahlKoerper { ...
```

mit der Implementation

```
Zahlkoerper<
    Zahlkoerper<
        Rational<GanzeZahl>,3>,3>
```

für das Beispiel $\sqrt[n]{D} = \sqrt[3]{3}, \sqrt[m]{E} = \sqrt[3]{5}$ (*wobei wir die Wurzelgrade als Integer-Template-Parameter eingeführt haben und die Wurzelargumente selbst noch setzen müssen*).

Den Wurzelparameter setzen wir mit einem ähnlichen Trick wie die Module bei den Restklassen, nur dass wir hier keinen Stack einführen (*wenn solche Körperdarstellungen benötigt werden, dann im Allgemeinen mit konstanten Parametern. Falls komplexere Aufgaben auftauchen, können Sie ja den Stack-Mechanismus nachträglich implementieren*).

> **Aufgabe.** Für den Wurzelparameter wird eine Klassenmethode des Typs `static inline` mit einer Variable des Typs `T static` benötigt. Der Rückgabewert ist `T&`, damit Werte geladen und gelesen werden können. Implementieren Sie dies und initialisieren Sie das obige Beispiel mit beiden Wurzelparametern.

Als Datenstruktur benötigen wir ein Feld von Werten des `template`-Parametertyps `T`. Wie in den Zahlenklassen zuvor implementieren wir dies wieder mit internen Datenträgern, die leicht austauschbar sind.

```
RefAllocClass(Komp)
       T a[stufe];
       inline void init(){...}
       inline Komp& operator=(Komp const& c){...}
   }* komp;
```

Die Berücksichtigung der weiteren aus dieser Definition erwachsenden Konsequenzen überlasse ich Ihnen.

Die arithmetischen Operationen haben Sie ja bereits in der vorletzten Aufgabe untersucht. Ich stelle die Ergebniss hier noch einmal dar:

Addition. Die beiden Zahlen werden komponentenweise addiert.

$$(c_0,c_{1,\ldots}c_{n-1}) = (a_0,a_{1,\ldots}a_{n-1}) + (b_0,b_{1,\ldots}b_{n-1}) \quad , \quad c_1 = a_i + b_i$$

Sonst ist hier weiter nichts zu bemerken.
Multiplikation. Wie üblich wird jede Komponenten der einen Zahl mit jeder der anderen multipliziert. Unter Berücksichtigung der Wurzelargumente finden wir:

$$c_k = \sum_{j=0}^{k} a_j * b_{k-j} + D * \sum_{j=k+1}^{n-1} a_j * b_{n+k+1-j}$$

wegen $\sqrt[n]{D^k} * \sqrt[n]{D^l} = D\sqrt[n]{D^{k+1}}$ für $k + l < n$ und
$\sqrt[n]{D^k} * \sqrt[n]{D^l} = D\sqrt[n]{D^{(k+l) \bmod n}}$ für $k + l \geq n$.
Division. Die Division ist für $n = 2$ mit Hilfe der konjugierten Erweiterung zu lösen:

$$\frac{(a_0,a_1)}{(b_0,b_1)} = \frac{(a_0,a_1)}{(b_0,b_1)} * \frac{(b_0, -b_1)}{(b_0, -b_1)} = \frac{(a_0,a_1) * (b_0,b_1)}{b_0^2 - Db_1^2}$$

Das Prinzip ist aus der Arithmetik der komplexen Zahlen bekannt und bedarf daher keiner weiteren Erläuterung.

Bei Körpern mit höherer Stufe ist eine geschlossene Lösung in dieser Form nicht möglich, sondern man muss auf die Definition des inversen Elementes zurückgreifen:

$$a/b = a * b^{-1} \quad mit \quad b * b^{-1} = 1$$

Aus der 2. Formel erhalten wir für $n = 4$ das lineare Gleichungssystem (*Sie können dies sicher leicht auf den allgemeinen Fall verallgemeinern*):

$$\begin{pmatrix} b_0 & b_3 * D & b_2 * D & b_1 * D \\ b_1 & b_0 & b_3 * D & b_2 * D \\ b_2 & b_1 & b_0 & b_3 * D \\ b_3 & b_2 & b_1 & b_0 \end{pmatrix} * \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Außer für $b = 0$ hat dieses System immer eine Lösung, die mittels des Gauss-Verfahrens zu ermitteln ist.

15.6.2 Implementation der Algorithmen

Da unsere Konstruktion den allgemeinen Fall mit unbekanntem n abdecken soll, würde eine schulmäßige Implementation mit `for`-Schleifen arbeiten und den in Kap. 6 implementierten Gauss-Algorithmus einsetzen. Da wir es aber mit relativ kleinen n zu tun haben, ist diese Lösung nicht besonders effizient.

```
// Standardimplementation:
for(i=0;i<3;i++)
   c[i]=a[i]*b[i];

// alternative Implementation
(*(c++)=*(a++))*=*(b++);
(*(c++)=*(a++))*=*(b++);
(*c=*a)*=*b;
```

Die durch geschickte Registernutzung oder durch Vermeidung temporärer Variabler gegebenenfalls mögliche Kodeoptimierung einmal Außen vorgelassen (*auch der Optimierer des Compilers soll ja etwas leisten, und wir haben ja schon einmal festgestellt, dass es oft nicht sinnvoll ist, ihn überlisten zu wollen*), haben wir zumindest die Schleifenvariable vermieden, bei drei Rechenschritten rein formal 25% des Gesamtaufwands (*das stimmt natürlich meist nicht, wenn in den Schleifenkörpern eigene Datentypen zum Einsatz kommen*).

Aufgrund des „freien“ Parameters n in der `template`-Parameterliste können solche effizienten Implementationen nur durch zur Compilezeit ablaufende Algorithmen erzeugt werden. Solche Algorithmen, die zur Compilezeit ähnliches bearbeiten wie Algorithmen zur Laufzeit, heißen auch Metaprogramme. Die Konstruktion solcher Programme wird aus der alternativen Implementation bereits ersichtlich: eine Template-Funktion mit dem Parameter n erzeugt eine Zeile des Alternativprogramms und ruft sich anschließend mit dem Parameter $n-1$ selbst wieder auf. Damit dies nicht zu einer Endlosschleife führt, existiert eine Spezialisierung für $n = 0$, die keinen weiteren Aufruf hat. In der Praxis sieht das für eine Funktion, die alle Komponenten eines Feldes mit einem bestimmten Wert initialisiert, folgendermaßen aus:

```
template <class T, int n>
struct FastOperation {
    static inline T& set(T* t, T const& value) {
        return *t=FastOperation<T,n-1>::
                                    set(t+1,value);
    }//end function
};//end struct

template <class T>
struct FastOperation<T,1> {
    static inline T& set(T* t, T const& value) {
        return *t=value;
    }//end function
};//end class

template <int n, class T> inline
void fo_clear(T* t)
  { FastOperation<T,n>::set(t,null<T>()); }
```

```
template <int n, class T> inline
void fo_set(T* t, T const& value)
  { FastOperation<T,n>::set(t,value);}
```

Im Programm aufgerufen wird eine der beiden Funktionen, z.B. durch `fo_set<5=(a,val)`. Rekursiv wird hierdurch die Zuweisungskette

```
t=*(t+1)=*(t+2)=*(t+3)=*(t+4)=value
```

erzeugt (*aufgrund der besonderen Zuweisungsstruktur müssen wir diese Kette bei* $n = 1$ *abbrechen lassen*). Die Hilfsfunktionen erleichtern die Aufrufsyntax und halten Offsetberechnungen zu Beginn des Algorithmus aus dem Hauptprogramm heraus (*siehe unten*).

> **Aufgabe.** Implementieren Sie auf ähnliche Weise Mate-Programme für die komponentenweise Addition von Feldern, das Kopieren der Inhalte eines Feldes auf ein anderes sowie die Multiplikation mit einem konstanten Faktor. Die statischen Methoden haben in diesem Fall natürlich keinen Rückgabewert:
>
> ```
> template <class T, int n>
> struct FastOperation {
>     static inline void aradd(T* a, T const* b) {
>         *a+=*b;
>         FastOperation<T,n-1>::aradd(++a,++b);
>     }//end function
>     ...
> ```

Etwas komplizierter ist die Multiplikation (*siehe oben*). Sie erfordert zwei „Skalarprodukte“ sowie eine Multiplikation. Auf die Komponenten einer temporären Variablen werden zunächst die mit dem Wurzelfaktor D behafteten Skalarprodukte summiert, diese mit D multipliziert und abschließend die unteren Skalarprodukte hinzugefügt. Die ergibt eine minimale Anzahl von notwendigen temporären Speicherplätzen.

```
template <class T, int n> inline ZahlKoerper<T,n>&
ZahlKoerper<T,n>::operator*=(ZahlKoerper<T,n>
                                        const& z){
    ZahlKoerper<T,n> tmp;
    fo_upsum<stufe>(tmp.komp->a,komp->a,z.komp->a);
    fo_mul<stufe>(tmp.komp->a,wurzel());
    fo_lowsum<stufe>(tmp.komp->a,komp->a,
                                  z.komp->a,false);
    *this=tmp;
    return *this;
}//end function
```

Damit dies funktioniert, müssen wir bei den Skalarprodukten als zusätzliche Parametrierung vorsehen, ob die Summationsvariable initialisiert wird oder nicht. Wir erhalten dann die Implementation

```
// Rekursive Klasse
template <class T, int n>
struct FastOperation {
    static inline
    T& ipsum(T& c, T const* a, T const* b){
        return FastOperation<T,n-1>::
                    ipsum(c,a+1,b-1)+=(*a * *b);
    }//end function

    static inline
    void lowsum(T* c, T const* a, T const* b){
        FastOperation<T,n>::ipsum(*c,a,b);
        FastOperation<T,n-1>::lowsum(--c,a,--b);
    }//end funcion
};//end struct

// Abbruchspezialisierung
template <class T>
struct FastOperation<T,1> {
    static inline
    T& ipsum(T& c, T const* a, T const* b){
        return c+=(*a * *b);
    }//end function

    static inline
    void lowsum(T* c, T const* a, T const* b){
        FastOperation<T,1>::ipsum(*c,a,b);
    }//end funcion
};//end class

// keine-Operation-Spezialisierung
template <class T>
struct FastOperation<T,0> {
    static inline
    T& ipsum(T& c, T const*, T const*){return c;}
    static inline
    void lowsum(T*, T const*, T const*){}
};//end class

// Einsatzmethode
template <int n, class T> inline
void fo_lowsum(T* c,T const* a,T const* b,
                                    bool clr=true){
    if(clr){
        fo_clear<n>(c);
    }//end funcion
    FastOperation<T,n>::lowsum(c+n-1,a,b+n-1);
}//end function
```

In der Funktion `fo_lowsum(..)` wird die Initialisierung des Zieles bei Bedarf durchgeführt und die notwendigen Offsets der Zeiger berechnet. Die Klassenmethode `lowsum(..)` übernimmt die Berechnung der Skalarprodukte für alle Felder, die Klassenmethode `ipsum(..)` die Berechnung der einzelnen Skalarprodukte mit gegenläufigen Indizes. Durch die Rekursion werden alle Summationen als aufeinander folgende Einzelanweisungen implementiert und zwei Schleifenvariablen eingespart. Durch Abbruchspezialisierung und „keine-Operation"-Spezialisierung muss bei der Implementation keine Nebenbedingung der Art ($n > 1$) beachtet werden.

> **Aufgabe.** Die Methoden sind sehr eng am beobachteten Problem ausgelegt, Sie können die Klasse `FastOperation` aber leicht auch durch weitere Methoden für andere Probleme erweitern. Implementieren Sie nun zunächst die Klasse `ZahlKoerper` einschließlich der Multiplikation.

Bliebe abschließend noch die Division zu implementieren, die eine Implementation des Gauss-Algorithmus verlangt. Die Matrix legen wir als Struktur an, so dass ein doppelter Zeiger entsteht und kein intern strukturiertes Feld:

```
template <class T, int n>
struct ZL { T a[n+1]; };

template <class T, int n>
struct FastGauss {
    ZL<T,n> zl[n];
    ...
```

Das Feld enthält den Zielvektor, d.h. der gesamt Algorithmus ist entsprechend seinem vorgesehenen Verwendungszweck darauf ausgelegt, eine Lösung für ein System zu liefern. Für das Setzen des Zielvektors und der Matrixelemente konstruieren wir wieder rekursive Klassenmethoden, die über die Zeilen iterieren:

```
// Setzen des Zielvektors
template <class T, int n, int i> struct vc {
    static inline void cp(ZL<T,n>* a, T const* b){
        a->a[n]=*b;
        vc<T,n,i-1>::cp(++a,++b);
    }//end function
};//end struct

template <class T, int n> struct vc<T,n,1> {
    static inline void cp(ZL<T,n>* a, T const* b){
        a->a[n]=*b;
    }//end function
};//end struct
```

Aus das Setzen der Matrixelemente besteht im Wesentlichen aus dem Kopieren der Komponenten der zu invertierenden Zahl in umgekehrter Reihenfolge, wobei

in jeder Zeile eine abnehmende Anzahl von Elementen mit der Wurzelbasis zu multiplizieren ist (*siehe oben*). Ist n die Anzahl der Zeilen und s die aktuell zu setzende Zeile, so besteht die Kunst lediglich darin, die Indizes der verschiedenen Kopier- und Multiplizieralgorithmen geschickt zu setzen.

```
// Setzen der Matrixelemente
template <class T, int n, int s> struct mv_fg{
    static inline void cp(T* a, FastGauss<T,n>& fg,
                                T const& d){
        fo_inv_copy-s>(fg.zl[s-1].a,a);
        fo_inv_copy<n-s>(fg.zl[s-1].a+s,a+s);
        fo_mul<n-s>(fg.zl[s-1].a+s,d);
        mv_fg<T,n,s-1>::cp(a,fg,d);
    }//end function
};//end struct

template <class T, int n> struct mv_fg<T,n,0>{
    static inline void cp(T const*,
                    FastGauss<T,n>&, T const&){}
};//end struct
```

Sind die Matrixelemente einmal gesetzt, kann der Gausssche Algorithmus, bestehend aus Pivot-Sortierung und Spaltenelimination, durchgeführt werden. Hierzu benötigen wir einen weiteren `int-template`-Parameter, der die aktuell bearbeitete Zeile mitzählt:

```
template <class T, int n, int m> struct _elim {
    static inline void f(ZL<T,n>* zl, T& tmp){
        _pivot<T,n,n-m,m-1>::f(zl);
        _line_elim<T,n,n-m,m-1>::f(zl,tmp);
        _elim<T,n,m-1>::f(zl,tmp);
    }//end function
};//end class

template <class T, int n> struct _elim<T,n,1> {
    static inline void f(ZL<T,n>*,T&){}
};//end class
```

Die Spaltenelimination umfasst die Zeilen $0 \leq m \leq n-2$, die jeweils zur Eliminierung der Spalten $m+1 \leq s \leq n-1$ verwendet werden. Wir lassen den Arbeitszähler m rückwärts laufen, um die Rekursion sauber abbrechen zu können, und setzen intern auf die normalen Zeilennummern $n-m$ um, um nicht das Gausssschema neu berechnen zu müssen. Programmiertechnisch macht dies nichts aus, da Ausdrücke wie $n-m+1$ zur Compilezeit ausgewertet werden und keinen Einfluss auf die Laufzeit haben.

Die Pivot-Sortierung muss nur dafür sorgen, dass in der Arbeitzeile im Diagonalelement keine Null steht. Die Größe der Werte ist unerheblich, da wie bereits erwähnt mit absolut genauen Zahlen gearbeitet wird. Die Rekursion wird

in diesem Sinne normal aufgebaut (*wieder mit Rückwärtszählung*), enthält aber laufzeitbedingte vorzeitige Abbrüche. Wir benötigen nun noch einen weiteren `int-template`-Parameter, um die Zeilen bis zum Matrixende zählen zu können. Da sich der Offset in den Zeilen jeweils ändert, wenn die nächste Arbeitszeile bearbeitet wird, können wir auf dieses aufwändige Indexsystem nicht verzichten.

```
template <class T, int n, int m, int k>
struct _pivot {
    static inline void f(ZL<T,n>* zl){
        if(zl[m].a[m]==null<T>()){
            swap(zl[m],zl[n-k]);
            _pivot<T,n,m,k-1>::f(zl);
        }//endif
    }//end function
};//end struct

template <class T, int n, int m>
struct _pivot<T,n,m,0> {
    static inline void f(ZL<T,n>* zl){}
};//end struct
```

Der Austausch der Zeilen fällt durch das Doppelzeiger-System sehr leicht, da nur Zeiger auf Felder getauscht werden.[22]
Zum Abschluss wird die Spaltenelimination nach dem gleichen Schema umgesetzt:

```
template <class T, int n, int m, int k>
struct _line_elim {
    static inline void f(ZL<T,n>* zl, T& tmp){
        if(zl[n-k].a[m]!=null<T>()){
            (tmp=zl[m].a[m])/=zl[n-k].a[m];
            fo_mul<n-m+1>((zl[n-k].a)+m,tmp);
            fo_ar_sub<n-m+1>((zl[n-k].a)+m,
                                    (zl[m].a)+m);
        }//endif
        _line_elim<T,n,m,k-1>::f(zl,tmp);
    }//end function
};//end struct

template <class T, int n, int m>
struct _line_elim<T,n,m,0> {
```

[22] Man hätte natürlich auch die Inhalte kopieren können und dafür ein innenstrukturiertes Feld (*Feld mit Doppelindex*) verwenden können, müsste dann aber auch die interne Indizierung bei der auswahl der Algorithmen beachten. Mit dem Doppelzeigersystem erledigt sich diese Problem, und aufgrund des Aufbaus der Algorithmen entsteht auch kein Mehraufwand gegenüber einer Indexberechnung.

```
    static inline void f(ZL<T,n>*,T&){}
};//end struct
```

Auch diese Aufgabe besteht wieder aus einer geschickten Berechnung der Indizes.

Aufgabe. Damit Sie auch einmal mit den Indizes herumexperimentieren können, erhalten Sie nun die Aufgabe, die Rückwärtselimination durchzuführen. Auch hierfür genügt wieder ein weiter Zählindex (Abb. 15.1).

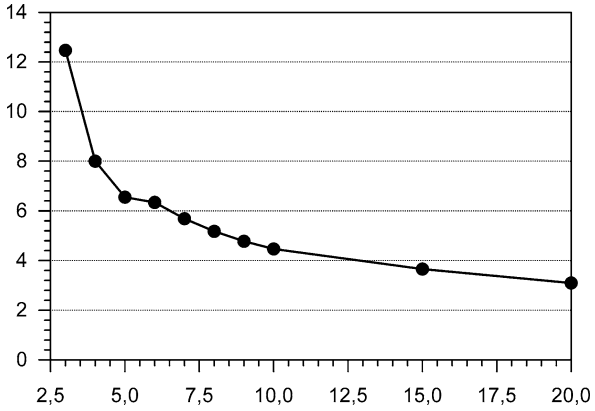


Abb. 15.1 Laufzeitverhältnis Standardalgorithmus/ Metaprogramm als Funktion des Matrizengrades

Damit haben wir alle wesentlichen Implementationsteile diskutiert, und Sie können die Klassendefinition nun abschließen und das Ganze in Betrieb nehmen. Was der Compiler hier erzeugt, kann recht umfangreich werden, weil entsprechend den drei geschachtelten `for`-Schleifen der schulmäßigen Implementationdes Gauss-Algorithmus nun drei geschachtelte rekursive Compiler-Algorithmen zum Einsatz kommen. Im Back-Engineering des fertigen Kodes haben wir eine Version vor uns, die jemand schreiben würde, der nie etwas von Schleifen gehört hätte. Im Gegenzug ist dieser Code allerdings so effizient wie möglich geworden. Ein Vergleich mit einer Standard-Gauss-Implementation weist folgende Verhältnisse auf:

Bei kleinen Graden – und gerade die sind ja im Zusammenhang mit Zahlkörpern interessant – ist eine Effizienzverbesserung um den Faktor 10 erreichbar. Bei zunehmendem Grad wird die Effizienzsteigerung erwartungsgemäß geringer, weil die Schleifen- und Adressverwaltung zunehmend einen verhältnismäßig geringeren Anteil an der Gesamtbelastung aufweist.