Piologie

Eine exakte arithmetische Bibliothek in C++

Version 1.3.2

Sebastian Wedeniwski

21. Oktober 2001

Zusammenfassung. PIOLOGIE ist eine Bibliothek für eine beliebig genaue Arithmetik, die mit natürlichen und ganzen Zahlen operiert.
Die Effizienz der Algorithmen in Theorie und Praxis und die Implementierung in ANSI-C++ nach neuestem Stand sind die grundlegenden Forderungen dieser Bibliothek. Dabei ist die Arithmetik auf alle Systeme portabel und vom Compiler unabhängig, da sie nur einen einzigen Grunddatentyp verwendet und dadurch ohne Laufzeiteinbußen Assembler vermeiden kann.
Übersichtlichkeit und leichte Handhabung der Bibliothek gewährleisten schließlich die einfachen Funktionsaufrufe, die für sich selbsterklärend sind, und die einschränkungsfreie Argumentübergabe.

Die vorliegende Arbeit ist auf den Prozessoren

80486, Pentium, PentiumPro, Pentium II,
DEC alpha,
HP PA,
IBM RS6000, IBM PowerPC, IBM S/390
MIPS,
SuperSPARC, UltraSPARC

mit den Compilern

Apogee C++ 3.0,
Borland C++ 5.02,
Digital C++ 5.0,
Edison Design Group C++ front end 2.33,
GNU C++ 2.6.1 - 2.8.1,
HP C++ A.10.22, HP aC++ A.01.00,
IBM CSet++ for AIX 3.1.1, IBM Visual Age C++ 3.0, IBM C++ 3.6
KAI C++ 3.2,
Microsoft Visual C++ 6.0,
SGI MIPSpro C++ 7.1,
SUN WorkShop C++ 4.2,
Watcom C++ 11.0

unter den Systemen

SunOS, Solaris, Irix, HP-UX, AIX, Digital Unix, Linux, OS/2, OS/390, Windows NT/95, DOS32

getestet worden.

Resultat. Moderne Architekturen machen Assembler hinfällig.

Inhaltsverzeichnis

Kapitel 1

Grundlagen

1.1 Grundlegende Implementierungsrichtlinien

Im folgenden stellen wir sieben Richtlinien zur Implementierung einer Arithmetik vor, auf denen die PIOLOGIE-Bibliothek beruht.

1. **Effizienz.** Die Effizienz der Algorithmen in Theorie und Praxis ist die grundlegende Forderung an diese Arbeit.

2. **Implementierung.** Bei der Implementierung muß der ANSI-C++ Standard eingehalten werden, um Portabilität für jedes System, das sich auch an den Standard hält, gewähren zu können. Daher sollten integrierte Assembler-Sequenzen prinzipiell vermieden werden und in zeitkritischen Situationen eine konkrete Analyse über den dadurch möglichen Zeitgewinn erfolgen, um zu entscheiden, ob überhaupt Assembler in Betracht gezogen werden kann.

3. **Einschränkungsfrei.** Die Bibliothek sollte so geschrieben sein, daß sie ohne Änderungen in den Quell-Codes auf jedem Rechnersystem, welches einen ANSI-C++ Compiler besitzt, compiliert und dort ausgeführt werden kann. Daher muß alles verkapselt programmiert sein, das heißt es dürfen keine globalen Variablen verwendet werden, und selbst globale Konstanten sollten sich selber einstellen können. Die voreingestellten Konstanten dürfen höchstens ein Geschwindigkeitsdefizit mit sich bringen, aber auf keinen Fall die Benutzung einschränken. Aus diesem Grund darf zum Beispiel eine Konstante `MAXARRAY`, die die feste Allokationsgröße in vielen Arithmetikimplementationen angibt und so die maximal möglichen Stellen einschränkt, nicht verwendet werden, denn damit wäre das Überlaufproblem lediglich auf eine höhere Ebene verschoben und keineswegs gelöst.

4. **Benutzerforderungen.** Es ist eine stabile Schnittstelle zum Anwender hin notwendig, weil dem Benutzer nicht vorgeschrieben werden darf, unter welchen Bedingungen, in welcher Umgebung und mit welchen Argumenten er die Funktion aufruft. Genausowenig sollte er mit dem verbindlichen Aufrufen von Konstruktoren und Destruktoren unnötig beansprucht werden.

5. **Einheitlichkeit.** Aus Stabilitäts- und Effizienzgründen darf es nur einen einzigen Grunddatentyp für alle elementaren arithmetischen Operationen geben; es soll also kein größerer Datentyp als Auffangbecken für die bei diesen Operationen entstandenen Überträge beziehungsweise Überläufe verwendet werden. Dieser Grunddatentyp muß komplett ausgenutzt werden, wobei die Basis für die Zahlendarstellung vom Computer und dem gewählten Datentyp abhängt.

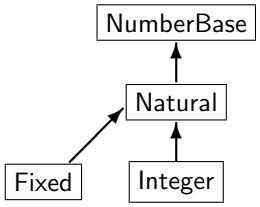
6. **Selbsterklärend.** Die Arithmetik sollte weitgehend durch sich selbst erklärend sein, weswegen es zum Beispiel auf jeden Fall überladene Operatoren geben muß. Diese Operatoren sollten so effizient

implementiert sein, daß sie Funktionen für arithmetische Operationen überflüssig machen. Das Überladen der Operatoren hat einen so hohen Stellenwert, daß es den alten Standard wie "`iadd`" für die Addition ganzer Zahlen beziehungsweise "`radd`" für die Addition rationaler Zahlen ersetzen soll. Spätestens nämlich, wenn wir Kombinationen bilden wollen, werden die Ausdrücke sonst unflexibel und ufern aus, wie etwa "`rpadd`" für die Polynomaddition über den rationalen Zahlen oder "`ipadd`" für die Polynomaddition über den ganzen Zahlen. Und das Spiel geht endlos weiter, wenn wir Polynome über Matrizen und Matrizen mit Polynomeinträgen und so fort betrachen, die beispielsweise bei der Lösung von gewöhnlichen Differentialgleichungen entstehen können.
Darüberhinaus sind Operatoren intuitiv standardisiert und verlangen deshalb nicht notwendig eine Spezifikation der Argumentübergabe.
Der Anwender sollte also nur mit den Sprachelementen vertraut sein müssen, die zur Lösung seiner Probleme unbedingt erforderlich sind.

7. Progressivität. Von großer Bedeutung sind die neueren Features der Programmiersprache C++, die durch die STL (siehe [39]) unumgänglich geworden sind. Durch die Verwendung dieser neuen Sprachmittel können oft nur aktuelle Compiler mithalten, wodurch die Portabilität einer Bibliothek gefährdet ist. In diesem Punkt werden wir unsere einzige Einschränkung machen und Compiler mit der STL voraussetzen, um eine Bibliothek zu haben, die (zumindest bis zur endgültigen Festlegung des Standards) auf neuestem Stand ist.

Nun gilt es zu untersuchen, ob eine Implementierung möglich ist, die all diesen Forderungen gerecht wird.

1.2 Zahlenhierarchie/-systeme



1.2.1 Verwaltung der Arithmetik

In NumberBase ist die grundlegende Verwaltung der Arithmetik verankert, unter anderem die Digitoperationen und die Fehlerbehandlung:

"digit.h" 2 ≡

```
////////////////////////////////
//
// Piologie V 1.3
// multi-precision arithmetic
// Digit / NumberBase
//
// (c) 1996-1999 HiPiLib
//               www.hipilib.de
//
// Sebastian Wedeniwski
```

```
// 12/13/1999
//

#ifndef _Include_Digit_H_
#define _Include_Digit_H_

⟨ included files of the file "digit.h" 395 ⟩
⟨ type declaration of SignDigit 185 ⟩
⟨ type declaration of Digit 3a ⟩
⟨ macro definitions of the file "digit.h" 396a ⟩
⟨ declarations of error-handling 396b ⟩
⟨ function prototypes of the file "digit.h" 396c ⟩
⟨ constants of PIOLOGIE 396d ⟩
⟨ class prototype of NumberBase 6a ⟩
⟨ inline implementation of the file "digit.h" 397d ⟩

#endif
◇
```

1.2.2 Implementationskonstanten

Unser Basisdatentyp Digit ist ein elementarer vorzeichenloser Typ, der mit den elementaren Speichereinheiten eines Rechners und Grundoperationen korrespondiert. In der Regel wird hierfür der elementare Typ `unsigned int` oder `unsigned long` eingesetzt:

⟨ type declaration of Digit 3a ⟩ ≡
```
typedef unsigned long Digit;
◇
```
Macro referenced in 2.

Wenn man allerdings den GNU-C++Compiler verwendet, dann ist auch der Typ `unsigned long long int` möglich, der genau die doppelte Datenlänge von `long int` hat.
Durch die Wahl des Basisdatentyps Digit steht gleichzeitig die erste wichtige Konstante β fest, die die Anzahl der Bits angibt:

⟨ constant β 3b ⟩ ≡
```
const size_t BETA        = sizeof(Digit)*CHAR_BIT;
◇
```
Macro referenced in 396d.

wobei `CHAR_BIT` die Anzahl der Bits pro Byte angibt, und zwar in der Standard-Headerdatei:

⟨ standard system include file <limits.h> 3c ⟩ ≡
```
#include <limits.h>
◇
```
Macro referenced in 395.

Eine weitere Konstante γ repräsentiert unsere maximal mögliche Zahl in Digit, das heißt

⟨ constant γ 3d ⟩ ≡
```
const Digit  GAMMA       = ~Digit(0);                 // 2^BETA - 1
◇
```
Macro referenced in 396d.

Um nicht bei jedem internen Überlauf die Langzahl um nur einen Digit zu erweitern, führen wir die Konstante δ ein, um die die Langzahl erweitert wird.

⟨ constant δ 4a ⟩ ≡
```
const size_t DELTA       = 8;                          // >= 2!
```
◇
Macro referenced in 396d.

Zur Konvertierung in das Dezimalsystem benötigen wir noch die Konstante $\alpha := 10^{\lfloor \log_{10}(2^\beta) \rfloor}$, die die größte mögliche Zehnerpotenz in Digit darstellt:

⟨ constant α 4b ⟩ ≡
```
const size_t ALPHA_WIDTH = size_t(0.301029995664*BETA);
const Digit  ALPHA       = pow10(ALPHA_WIDTH);
```
◇
Macro referenced in 396d.

Die Länge unseres Naturals wird durch den Typ `size_t` aus der Standard-Headerdatei

⟨ standard system include file `<stdlib.h>` 4c ⟩ ≡
```
#include <stdlib.h>
```
◇
Macro referenced in 395.

repräsentiert, der folgendermaßen belegt ist:
Sei $x \in \mathsf{Natural} \subset \mathbb{N}$, dann gilt für die Länge von x

$$\mathcal{L}(x) := \begin{cases} \lfloor \log_{2^\beta} x \rfloor + 1\,, & x > 0 \\ 1\,, & x = 0 \end{cases} .$$

1.2.3 Die Klasse NumberBase und grundlegende Notationen aus C++

Alle Implementierungen erfolgen ausschließlich in C++. An dieser Stelle ist zwar keine Einführung in C++ vorgesehen, aber eine knappe Erläuterung einiger grundlegender Begriffe und Vorgehensweisen am Beispiel der Klasse NumberBase. Zum besseren Verständnis des folgenden ist eine Einführung in C++ wie zum Beispiel mit den Werken von B. Stroustrup [50] und S. B. Lippman [33] empfehlenswert.

Einige Schlüsselwörter

Eine **Klasse** ist eine grundlegende Struktur, die sowohl Daten als auch Funktionen zusammenfaßt und sie als gleichwertige Elemente enthält. Diese Elemente werden in drei Zugangsebenen aufgeteilt:

1. Auf die privat deklarierten Elemente im **private**-Abschnitt können nur Elemente derselben Klasse zugreifen, weshalb ein Zugriff von "außen" nicht möglich ist.
2. Die im **protected**-Abschnitt geschützten Elemente können dagegen nicht nur von der eigenen Klasse, sondern auch von den abgeleiteten Klassen erreicht werden.
3. Die öffentlichen Elemente schließlich sind im **public**-Abschnitt allgemein zugänglich.

Damit können wir nun Strukturen erzeugen, in denen ein Anwender nur gewisse Zugangsrechte erteilt bekommt.

Bei einer Anweisung wie `NumberBase x` wird Speicherplatz für die Variable `x` (unser erzeugtes **Objekt**) reserviert, das auch als **Instanz** der Klasse NumberBase bezeichnet wird.

Das Schlüsselwort `inline` vor der Funktionsdefinition bedeutet ein direktes Einsetzen ins Programm, ähnlich einer textuellen Ersetzung eines Makros mit zusätzlicher Typüberprüfung und bedingungsloser Argumentübergabe. Der Vorteil besteht darin, daß keine Sprünge mehr in die Funktion hinein auftreten, wie zum Beispiel beim Vertauschen zweier Digits:

⟨ the function swap for Digits 5 ⟩ ≡

```
inline void swap(Digit& a, Digit& b)
// Algorithm:  swap(a, b)
//             Let t in Digit.
// Input:      a,b in Digit.
// Output:     a,b in Digit such that t := a, a := b, b := t ||
{
  Digit t = a; a = b; b = t;
}
```

◇

Macro referenced in 397d.

`inline`-Funktionen sind aber nur für kleine und schnelle Funktionen sinnvoll, um Effizienzeinbußen bei den relativ zeitaufwendigen Funktionsaufrufen einzusparen; gleichzeitig gewährt man hiermit dem Compiler bessere Optimierungsmöglichkeiten. Bei Funktionen, die einen größeren Rechenaufwand benötigen, ist der Zeitgewinn durch eine `inline`-Funktion überhaupt nicht meßbar und daher auch nicht notwendig:

```
inline void copy(Digit* to, Digit* from)
{
  for (size_t sz = 1000; sz; --sz) *to++ = *from++;
}
```

Hier ist die Laufzeit für die Schleifendurchläufe sogar um ein Vielfaches höher als ein einzelner Sprungbefehl.
`inline`-Funktionen können auch geschickt zur Vermeidung der `goto`-Anweisung eingesetzt werden:

```
inline bool found(const int z, const int* x, const int n, const int m)
{
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < m; ++j)
      if (x[i][j] == z) return true;
  return false;
}

void search(const int z, const int* x, const int n, const int m)
{
  if (found(z, x, n, m)) //...
  else //...
}
```

Konstruktor/Destruktor

Eine Elementfunktion, die den gleichen Namen wie die Klasse selbst trägt, ist ein **Konstruktor**. Diese Funktion wird jedesmal aufgerufen, wenn ein neues Objekt angelegt werden soll. Um dieses Objekt wieder zu entfernen, dient das Gegenstück zum Konstruktor, der **Destruktor**. Dieser wird mit dem Namen der Klasse und einer vorangestellten Tilde (~) bezeichnet. Konstruktoren und Funktionen können auch **überladen** werden. Das bedeutet, daß mehrere Funktionen denselben Namen zugewiesen bekommen können, wobei sie sich aber in den Argumenten unterscheiden müssen. Jede Klasse besitzt aber nur eine Destruktor-Funktion, die argumentlos ist. Dieser Destruktor wird automatisch für jedes Objekt aufgerufen, wenn sein definierter **Gültigkeitsbereich** verlassen wird.

Die Klasse NumberBase

Die Deklaration unserer Klasse hat die folgende Gestalt:

⟨ class prototype of NumberBase 6a ⟩ ≡

```
class NumberBase {
protected:
  NumberBase();                                  // default constructor
  ~NumberBase();                                 // destructor

  ⟨ protected function prototypes 397b ⟩
public:
  ⟨ protected attributes 397a ⟩
  ⟨ function prototypes for Digitoperations in NumberBase 397c ⟩

  void errmsg(const int, const char*) const;
};
```

◇

Macro referenced in 2.

Weitere Konstruktoren werden wir beim Betrachten der Klasse Natural im Kapitel 2.2.1 auf der Seite 24 kennenlernen.

1.3 Korrekte Software

Wir verwenden mehrere Diagnosemakros, die die Aufgabe haben, die Sicherheit während und nach dem Abschluß der Bibliotheksentwicklung zu testen. Durch Diagnoseroutinen im Quelltext lassen sich dabei Fehler einfacher erkennen, lokalisieren und beheben. Als Grundlage für die folgenden Makros

⟨ macro definitions for internal conditions 6b ⟩ ≡

```
#ifdef _Piologie_Debug_
# define CONDITION(expr)     assert(expr)

⟨ condition of a Natural 20a ⟩
⟨ condition of an Integer 187a ⟩
#else
# define CONDITION(__ignore)        ((void)0)
# define NATURALCONDITION(__ignore) ((void)0)
# define INTEGERCONDITION(__ignore) ((void)0)
# define NATURAL_FOR_CHECK(__ignore1, __ignore2) ((void)0)
# define INTEGER_FOR_CHECK(__ignore1, __ignore2) ((void)0)
#endif
```

◇

Macro referenced in 398b, 409, 426, 437.

dient die ANSI-Funktion `assert`, die eine Bedingung prüft und das Programm abbricht, wenn sie nicht zutrifft. Aus Effizienzgründen wird die Indirektion `_Piologie_Debug_` eingeführt, um die zusätzlichen Überprüfungen im Quelltext gegebenenfalls abschalten zu können.

1.4 Ordnung

Definition. Sei $f : \mathbb{N}^k \to \mathbb{R}$ eine Funktion für $k \in \mathbb{N}$, dann ist

$$\mathcal{O}(f) := \{g : \mathrm{Im}(f) \to \mathbb{R} \mid \exists_{a,b \in \mathbb{R}_{>0}} \forall_{x \in \mathrm{Im}(f)} : a|g(x)| \leq |f(x)| \leq b|g(x)|\}$$

die **Äquivalenzklasse** von f. Dadurch sind zwei Funktionen $f, g : \mathbb{N}^k \to \mathbb{R}$ zueinander **äquivalent** ($f \sim g$), falls $f \in \mathcal{O}(g)$ gilt. Alle Funktionen dieser Klasse benötigen also – mit Ausnahme einer Konstante – denselben zeitlichen Aufwand. Dabei nennt man $\mathcal{O}(f)$ auch die **Ordnung** von f.
Der Einfachheit halber schreiben wir ab jetzt $\mathcal{O}(\mathrm{Im}(f))$ statt $\mathcal{O}(f)$.

Beispiel (Multiplikation zweier Naturals). Rekursiv betrachtet, sah lange Zeit die konventionelle Methode, zwei Naturals miteinander zu multiplizieren, folgendermaßen aus:

Seien $x, y \in \mathsf{Natural}$ mit $\mathcal{L}(x) = \mathcal{L}(y) = n$, so gilt
$x \cdot y = 2^{2m\beta} x_1 y_1 + 2^{m\beta}(x_1 y_0 + x_0 y_1) + x_0 y_0$
mit $m := \lfloor n/2 \rfloor, x := 2^{m\beta} x_1 + x_0$ und $y := 2^{m\beta} y_1 + y_0$.

A. Karatsuba jedoch konnte die Anzahl der notwendigen Operationen von $\mathcal{O}(n^2)$ auf $\mathcal{O}(n^{\log_2(3)})$ reduzieren[1], indem er die folgende Rekursion aufstellte:

$$a := x_1 y_1, b := (x_1 - x_0)(y_0 - y_1), c := x_0 y_0 \text{ und somit } xy = 2^{2m\beta} a + 2^{m\beta}(b + a + c) + c.$$

Genaugenommen müßten wir

$$\texttt{mul} \in \mathcal{O}((\mathcal{L}(x), \mathcal{L}(y)) \mapsto \mathcal{L}(x) \cdot \mathcal{L}(y)) = \mathcal{O}(\mathcal{L}(x) \cdot \mathcal{L}(y))$$

mit $\texttt{mul} : \mathsf{Natural}^2 \to \mathsf{Natural} : (x, y) \mapsto x \cdot y$ für die konventionelle Multiplikation zweier Zahlen schreiben. Man wählt jedoch oft der Einfachheit und Übersichtlichkeit wegen einen möglichst einfachen **Repräsentanten**, zum Beispiel $\texttt{mul} \in \mathcal{O}(n^2)$, wenn es nicht so sehr auf die genaue Anzahl der Operationen ankommt.

1.5 Programmtechnische Optimierungsrichtlinien

In diesem Abschnitt sind zwei Richtlinien von besonderem Interesse:

1. **Klare Struktur.** Ein wichtiger Punkt ist die klare Struktur und elegante Integrierung der Algorithmen. Daher dürfen die längst überholten `goto`-Anweisungen[2] nicht mehr verwendet werden und sollten die Sprachelemente `break` und `continue` möglichst vermieden werden. Bei `for`-Konstruktionen kann man geteilter Meinung sein, wie das folgende Beispiel zeigt, das in Bezug auf die Code-Generierung in beiden Fällen äquivalent ist:

```
for (i = 0; i < j; ++i)                                   // mit break-Anweisung
  if (i == k) break;

for (i = 0; i < j && i != k; ++i);                        // ohne break-Anweisung
```

   Hierbei kann es dem Programmierer überlassen bleiben, welche Konstruktion für ihn ansprechender ist und er lieber verwenden möchte. Wir werden jedoch bei der Implementierung der Langzahlbibliothek die zweite Variante (ohne `break`-Anweisung) bevorzugen.

[1] Eine genaue Analyse findet man im Kapitel 2.11.4 auf der Seite 95.
[2] Diese sind eher rudimentäre Sprachelemente aus C-Zeiten als übersichtliche Strukturhilfen.

2. Schleifen aufrollen und komprimieren. Unser Anliegen besteht nicht darin, das Programm durch aufrollen und komprimieren von Schleifen mit aller Gewalt zu optimieren, was meist nämlich nur zu unübersichtlichen Befehlsfolgen führt, wie zum Beispiel bei dieser "kompakten" Primzahlermittlung:

```
void primes(int n){for(int a=n;--a;n%a?a:--a?a=--n:printf("%d,",n));}
```

Warum sollte jemand so etwas schreiben? Für einen Blumentopf?

Bemerkung. Weitaus effizienter und übersichtlicher arbeiten die Siebalgorithmen, die wir später im Kapitel 5.7.1 noch betrachten werden.

Ähnliches liegt auch bei folgendem "aufgerollten" Algorithmus vor:

```
void send(register* to, register* from, register count)
// Duff's Device. Hilfreiche Kommentare wurden bewußt entfernt.
{
  register n = (count+7)/8;
  switch (count%8) {
    case 0: do { *to++ = *from++;
    case 7:      *to++ = *from++;
    case 6:      *to++ = *from++;
    case 5:      *to++ = *from++;
    case 4:      *to++ = *from++;
    case 3:      *to++ = *from++;
    case 2:      *to++ = *from++;
    case 1:      *to++ = *from++;
               } while (--n > 0);
  }
}
```

Dasselbe Ergebnis kann nämlich auch anders erzielt werden, wie die drei nächsten Varianten zeigen:

```
inline void copy(Digit* result, const Digit* first, const size_t count)
{
  for (const Digit* last = first+count; first != last; ++first, ++result)
    *result = *first;
}

inline void copy2(Digit* result, const Digit* first, size_t count)
{
  while (count) {
    --count;
    result[count] = first[count];
  }
}

inline void memcopy(Digit* to, const Digit* from, size_t count)
{
  memcpy(to, from, count*sizeof(Digit));
}
```

Alle vier Versionen sind zwar etwa gleich schnell[3],

Prozessor/ Compiler	`copy` (Basis)	`copy2`	`memcopy`	`send`
Pentium:				
Watcom C++ 10.6	100%	97,8%	99,7%	99,7%
Visual C++ 4.0	100%	87,2%	88,9%	100%
Borland C++ 5.0	100%	87,3%	87,4%	87,3%
GNU-C 2.7.2:				
Pentium	100%	100%	99,1%	98,6%
80486	100%	98,6%	97,8%	98,6%
SPARCstation10	100%	100,9%	99,4%	92,3%
UltraSPARC	100%	152,1%	38,3%	100,0%
IBM Power PC	100%	105,2%	85,7%	90,5%
HP PA 7100	100%	104,9%	99,4%	99,6%
Silicon Graphics C++ 5.3:				
MIPS R4000	100%	132,7%	74,9%	120,1%
DEC C++:				
DEC Alpha	100%	98,8%	86,7%	98,8%

aber erstens ist der `send`-Code ungefähr um einen Faktor fünf umfangreicher als die drei anderen Lösungen,

Prozessor/ Compiler	`copy`	`copy2`	`memcopy`	`send`
Pentium:				
Watcom C++ 10.6	39 Bytes	39 Bytes	33 Bytes	153 Bytes
Visual C++ 4.0	36 Bytes	40 Bytes	25 Bytes	249 Bytes

und zweitens scheidet eine solche typenlose Datenübergabe aus Sicherheitsgründen für uns aus. Um die drei kürzeren Varianten voneinander abzugrenzen, seien die folgenden Gedanken erwähnt: Bei dieser Arithmetik muß man bedenken, daß sich der Einsatz der Kopierfunktion meist auf wenige Ziffern beschränkt, weshalb sich zur Steigerung der Effizienz eigentlich ein `inline` anbietet. Bei den meisten Compilern ist auch die ANSI-Funktion `memcpy` als `inline` gelöst und bietet in der Regel die optimalen Prozessoranweisungen, doch ist ihr Verhalten bei Überlappung undefiniert. Damit ist diese Funktion für eine unserer häufigsten Tätigkeiten, die interne Datenverschiebung, in der Regel nicht zu gebrauchen. Welche der beiden Funktionen `copy` und `copy2` nun schneller ist, hängt hauptsächlich vom verwendeten Prozessor ab.
Wir werden bei der Implementierung der Algorithmen überwiegend die zeigerorientierte `copy`-Funktion vorziehen und der Übersichtlichkeit wegen, die folgenden sechs Makros zur Speicherverarbeitung verwenden:

⟨ macro definitions for internal memory manipulations 9 ⟩ ≡

```
#define FILL_ZERO(a, b)                                    \
  do *a++ = 0; while (a != b)

#define FILL_DELTA(a)                                      \
  a[0] = a[1] = a[2] = a[3] = a[4] = a[5] = a[6] = a[7] = 0;

#define COPY(a, b, c, d)                                   \
  do *a++ = *b++; while (c != d);
```

[3]Die Laufzeiten sind mit der Funktion `copy` als Basisgröße prozentual ermittelt, das heißt, daß zum Beispiel beim UltraSPARC die Funktion `copy2` um 52,1% langsamer ist und die `memcopy` Funktion nur 38,3% der von `copy` benötigten Zeit braucht.

```
#define COPY_BACKWARD(a, b, c, d)                       \
  do *--a = *--b; while (c != d);

#define MOVE(a, b, c, d)                                \
  do { *a++ = *b; *b++ = 0; } while (c != d);

#define MOVE_BACKWARD(a, b, c, d)                       \
  do { *--a = *--b; *b = 0; } while (c != d);
◇
```

Macro referenced in 398b, 409.

1.6 Additionsalgorithmus

In diesem Abschnitt werden wir aus zwei Gründen einen neuen Additionsalgorithmus entwerfen:

1. Es ist sinnvoll, einen einzigen Grunddatentyp für alle elementaren arithmetischen Operationen zu verwenden, der vollständig ausgenutzt wird (siehe Kapitel 1.1).

2. Assembler-Sequenzen können mit wenigen Ausnahmen ohne Laufzeiteinbußen vermieden werden.

1.6.1 Ansätze

Dezimaler Additionsalgorithmus

A. Binstock und J. Rex führen in [5] auf Seite 422 einen dezimalen Ansatz für einen Additionsalgorithmus ein:

```
for (int i = start; i >= stop; i--) {
  sum->term[i] += t1->term[i] + t2->term[i];
  if (sum->term[i] > 9) {
    sum->term[i] -= 10;
    sum->term[i-1] += 1;
  }
}
```

Wir werden in den nächsten Abschnitten noch sehen, daß wir durch diese Darstellung sehr viel Rechenzeit verschenken, auch wenn wir als Basis die größtmögliche Zehnerpotzenz wählen.

Additionsalgorithmus über Bytes

W. H. Press, S. A. Teukolsky, W. T. Vetterling und B. P. Flannery nutzen mit ihrem Ansatz in [40] auf Seite 916 zwar ein komplettes Byte aus, benutzen jedoch für die elementaren Operationen zwei verschiedene Maschinentypen, und das, um lediglich eine überlaufende Eins im höheren Datenbereich abzufangen. Da aber auf den heutigen Maschinen eine 32-Bit-Addition – genauso wie die 8-Bit-Addition[4] – nur einen Taktzyklus in Anspruch nimmt, ist hier mindestens ein Faktor vier geopfert worden.

[4] Genaugenommen darf man im allgemeinem nicht davon ausgehen, daß der Maschinentyp `char` aus 8 Bit besteht.

```
unsigned short ireg = 0;
for (int j = n; j >= 1; j--) {
  ireg = u[j] + v[j] + (unsigned char)(ireg & 0xff);
  w[j+1] = (unsigned char)(ireg >> 8 & 0xff);
}
w[1] = (unsigned char)(ireg >> 8 & 0xff);
```

(Trotzdem ist dieser Ansatz recht effizient im Hinblick auf die Assembler-Code-Erzeugung. Die Maschinenunabhängigkeit wird dadurch erzielt, daß die Relation

$$\texttt{sizeof(char)} \leq 2 \cdot \texttt{sizeof(short)}$$

für die internen Maschinentypen vom ANSI-Komitee festgelegt wurde.)

1.6.2 Additionsalgorithmus, der die vorgegeben Kriterien erfüllt

Um uns an die ANSI-Festlegungen zu halten, dürfen wir bei der Addition keinen Zugriff auf die Maschinen-Flags vornehmen, was die Implementierung erheblich erschwert. Also müssen wir für den Übertrag zuersteinmal ein Register verwenden.
Eine effiziente Möglichkeit wird durch die folgende Gleichung realisiert:

Seien $a, b \in \mathsf{Digit}$, so gilt $\quad a + b = c \cdot 2^\beta + d$ mit $a + b \equiv d \pmod{2^\beta}$ und $c = \begin{cases} 1\,, & d < a \\ 0\,, & d \geq a \end{cases}$,

was in ANSI-C++ die folgende Gestalt hat:

```
Digit d = a+b;
Digit c = (d < a);
```

weil `(d < a)`$\in \{0, 1\} \subseteq$ `int` gilt.

Mit dieser Vorüberlegung zur Ermittlung des Übertrages werden wir unseren Additionsalgorithmus in zwei Bereiche aufteilen: Einen zur überlaufsfreien Addition und einen zur Addition inklusive Inkrementierung.
Diese beiden Bereiche sind stellvertretend für eine Übertragsvariable, die damit eingespart werden kann.

⟨ memberfunction Natural.add_no_inc with 3 arguments 11 ⟩ ≡

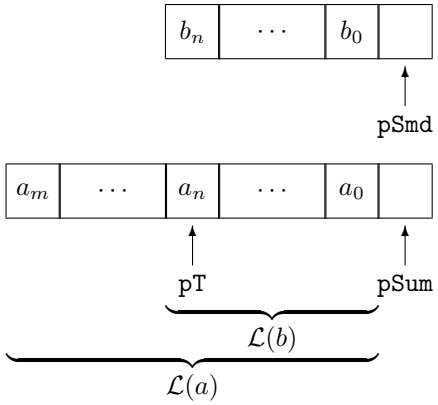
```
bool Natural::add_no_inc(const Digit* pT, Digit* pSum, const Digit* pSmd) const
// Algorithm:  d := x.add_no_inc(r, s, t)
//             Let a,b in Natural.
// Input:      x in Natural,
//             r,s in [a.root, a.p+L(a)] where r < s,
//             t in [b.root, b.p+L(b)] where t-(s-r) also in [b.root, b.p+L(b)].
// Output:     d in bool such that [d] x [r, s[ := [r, s[ + [t-(s-r), t[ ||
{
  CONDITION(pT < pSum);

  Digit c,d;
  do {
    c = *--pSmd;                          // non carry addition
    d = *--pSum;
    *pSum = d += c;
    if (c > d)
      do {
```

```
      if (pSum == pT) return true;
      c = *--pSmd;
      d = *--pSum;
      *pSum = d += c+1;                  // addition with carry
    } while (c >= d);
  } while (pSum != pT);
  return false;
}
```
◇

Macro referenced in 409.

Hierbei werden die Zeiger (`pT`, `pSum`, `pSmd`) für die Anweisung `a += b` folgendermaßen übergeben:

Seien $a, b \in \mathsf{Natural}$ mit $\mathcal{L}(a) - 1 = m \geq n = \mathcal{L}(b) - 1$, dann



Der Additionsalgorithmus bleibt auch weitgehend gleich, wenn wir eine Addition mit zwei verschiedenen Summanden durchführen, die dem `operator +` entspricht.

⟨ memberfunction Natural.add_no_inc with 4 arguments 12 ⟩ ≡

```
bool Natural::add_no_inc(const Digit* pT, Digit* pSum,
                         const Digit* pSmd1, const Digit* pSmd2) const
// Algorithm:  d := x.add_no_inc(r, s, u, v)
//             Let a,b,c in Natural.
// Input:      x in Natural,
//             r,s in [a.root, a.p+L(a)] where r < s,
//             u in [b.root, b.p+L(b)] where u-(s-r) also in [b.root, b.p+L(b)].
//             v in [c.root, c.p+L(c)] where v-(s-r) also in [c.root, c.p+L(c)].
// Output:     d in bool such that [d] x [r, s[ := [u-(s-r), u[ + [v-(s-r), v[ ||
{
  CONDITION(pT < pSum);

  do {
    Digit c = *--pSmd1;                 // non carry addition
    Digit d = *--pSmd2;
    *--pSum = c += d;
    if (d > c)
      do {
        if (pSum == pT) return true;    // addition with carry
        c = *--pSmd1;
        d = *--pSmd2;
```

```
        *--pSum = c += d+1;
      } while (d >= c);
    } while (pSum != pT);
    return false;
  }
  ◇
```

Macro referenced in 409.

1.6.3 Praxisdemonstration an den Fibonacci-Zahlen

Die Fibonacci-Zahlen sind definiert durch die Rekursion:

$$F_0 = 0, F_1 = 1 \text{ und } F_n = F_{n-1} + F_{n-2} \quad \text{für } n \geq 2.$$

Wir verwenden den folgenden Algorithmus, in dem die Addition die entscheidende Rolle spielt:

⟨ initialization of Fibonacci algorithm 13a ⟩ ≡

```
Natural k,i,j(1);
◇
```

Macro referenced in 22a.

⟨ additive Fibonacci algorithm 13b ⟩ ≡

```
while (--n) {
  k = i+j;
  if (--n == 0) return k;
  i = j+k;
  if (--n == 0) return i;
  j = k+i;
}◇
```

Macro referenced in 22a.

Wir werden im Kapitel 5.1 noch einen wesentlich schnelleren Algorithmus kennenlernen; momentan liegt jedoch unser Hauptinteresse in der Analyse des Additionsalgorithmus und nicht in der schnellstmöglichen Berechnung der Fibonacci-Zahlen.
Bei der Analyse betrachten wir verschiedene Compiler auf verschiedenen Betriebssystemen, um die Portabilität zu demonstrieren. Das Programm ist jeweils unter optimalen Bedingungen compiliert und ausgeführt worden, um dadurch vergleichbare Ergebnisse zu erreichen.
Für einen Pentium-Rechner mit 100 MHz ergaben sich die folgende Zeiten:

	Watcom C++ 10.6 (OS/2) (OS/2)	Visual C++ 4.0 (Win32)	Borland C++ 5.0 (Win32)	GNU-C 2.7.2 (Linux)
`fib(10000)`	0.16 s	0.16 s	0.15 s	0.15 s
`fib(100000)`	18.4 s	13.5 s	15.2 s	14.9 s

Auf anderen Rechnersystemen unter dem Betriebssystem Unix und mit dem populären GNU-Compiler mit der gleichen Parameterübergabe (`-O2`) sind folgende Zeiten entstanden:

	SUN-SPARCstation10 (40 MHz)	SUN UltraSPARC (134 MHz)	IBM Power PC (80 MHz)	HP PA 7100
`fib(10000)`	0.20 s	0.08 s	0.13 s	0.17 s
`fib(100000)`	25.5 s	8.4 s	12.5 s	15.7 s

Und mit anderen Compileren:

Prozessor	Compiler	`fib(10000)`	`fib(100000)`
MIPS R4000 (100 MHz)	Silicon Graphics C++ 5.3	0.18 s	23.15 s
DEC Alpha (150 MHz)	DEC C++	0.07 s	7.55 s

Kapitel 2

Die Arithmetik natürlicher Zahlen

2.1 Spezifikation

2.1.1 Natürliche Zahlen

Die Menge der natürlichen Zahlen bezeichnet man mit $\mathbb{N} := \{0,1,2,3,4,...\}$. Unsere Klasse Natural ist eine echte Teilmenge von $\mathbb{N}$ mit den folgenden Eigenschaften:

- Es existiert eine größte natürliche Zahl ω, die speicherabhängig ist,
  das heißt $\max(\mathsf{Natural}) < \omega$.
- Jede Zahl $x \in \mathsf{Natural} - \{\omega\}$ besitzt einen Nachfolger (`operator++`).
- Jede Zahl $x \in \mathsf{Natural} - \{0\}$ besitzt einen Vorgänger (`operator--`).
- Verschiedene natürliche Zahlen haben verschiedene Nachfolger.

Rechnungen:

- Addition, Multiplikation und Potenzierung in Natural sind abgeschlossen (Speicherüberlauf ausgenommen), das heißt für $n, m \in \mathsf{Natural}$ gilt:

$$n + m \in \mathsf{Natural}, n \cdot m \in \mathsf{Natural}, n^m \in \mathsf{Natural}.$$

- Die Subtraktion hingegen ist in Natural nicht abgeschlossen.
  Zum Beispiel $1 - 2 \notin \mathsf{Natural}$ (Fehlermeldung: “result negative!”).
- Die Division ist ebenfalls in Natural nicht abgeschlossen.
  Zum Beispiel $\frac{3}{2} \notin \mathsf{Natural}$, weil $3 \equiv 1 \pmod 2$.
  Stattdessen wird eine ganzzahlige Division durchgeführt: Für $n, m \in \mathsf{Natural}$ gilt somit:
  $\left\lfloor \frac{n}{m} \right\rfloor := \sup\{x \in \mathsf{Natural} : x \leq \frac{n}{m}\} \in \mathsf{Natural}$. Den Divisionsrest von $\frac{n}{m}$ erhalten wir dabei durch $n - m \cdot \left\lfloor \frac{n}{m} \right\rfloor$.

2.1.2 Datenstruktur

Eine entscheidende Rolle für die Effizienz einer Arithmetik spielt die gewählte Datenstruktur, denn hierdurch wird auch die Leistungsfähigkeit der Algorithmen festgelegt. Um nun Zahlenketten mit einer vorgegebenen Struktur im Speicher abzulegen, gibt es mehrere Varianten, von denen zwei vorgestellt werden sollen:

- Die linear verketteten Listen haben, wie Sedgewick in [44] auf den Seiten 591-594 zeigt, den Vorteil, daß der Speicher optimal genutzt werden kann und somit redundante Nullen gar nicht auftreten können. Der Vorteil ist, daß das Anhängen von neuen Gliedern (insbesondere Digits) in einer verketteten List nur eine konstante Zeit benötigt. Doch gerade hierin liegen auch drei entscheidene Nachteile:

  1. Durch die vielen Allokationen entstehen beträchtliche Geschwindigkeitseinbußen.
  2. Es wird wegen der vielen Zeiger auf den jeweiligen Nachfolger doppelt soviel Speicher verbraucht.
  3. Der Zugriff auf ein beliebiges Glied unserer Kette ist nicht mehr in konstanten Zeit möglich.

- Eine andere Möglichkeit bieten homogene Felder (Arrays beziehungsweise Vektoren), die nur einmal alloziiert werden und danach linear im Speicher zur Verfügung stehen. Der große Nachteil hierbei ist aber die einmalige Allokation, da dadurch von vornherein die Länge feststeht und somit zwangsläufig entweder nicht verwendeter Speicher mitoperiert oder ein unerwünschter Überlauf entsteht. Viele schlecht programmierte Langzahl-Arithmetiksysteme setzen daher einfach eine Konstante fest, die je nach Aufgabenstellung vergrößert beziehungsweise verkleinert wird. Oftmals kann diese Konstante jedoch nicht einmal während des Programmablaufs verändert werden und zwingt dadurch den Benutzer, vor dem Compilieren die entsprechenden Konstanten zu setzen. Daran denken zu müssen, ist nicht nur lästig für den Benutzer, sondern auch längst nicht mehr nötig nach dem heutigen Stand der Technik. Hier könnte man zum Beispiel abhelfen, indem ein Überlauf intern abgefangen und mittels einer neuen Allokation das Array durch ein größeres substituiert wird. Dies kann man oftmals so geschickt mit Operationen verbinden, daß keine zusätzliche Laufzeit für das notwendige Kopieren entsteht und dadurch nur eine konstante Zeit zur Allokation das Arrays hinzukommt. Allgemein betrachtet, benötigen wir jedoch für das Anhängen einer neuen Komponente an unser Array einen linearen Zeitaufwand, was deutlich schlechter ist als der konstante Zeitaufwand bei einer linear verketteten Liste. Doch darauf werden wir weiter unten im Kapitel 2.1.4 noch genauer eingehen.

2.1.3 Zahlendarstellung

Ein Natural wird bei uns durch den Zeiger `root` für den Arrayanfang, `p` für die Anfangsposition der dargestellten Zahl (nur aus Effizienzgründen, da zwischen `root` und `p` nach Konvention nur Nullen enthalten sind) und schließlich durch die Anzahl der notwendigen Digits `size` repräsentiert. Das Grundgerüst bilden Arrays, die sich während der Laufzeit vergrößern können, um dadurch einen unerwüschten Überlauf zu vermeiden.

⟨ variables for representation of a Natural 16 ⟩ ≡

```
size_t   size;
Digit*   root;
Digit*   p;
```

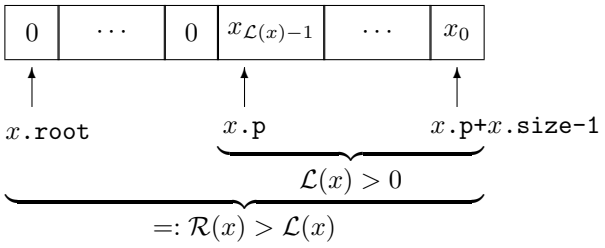
◇

Macro referenced in 402b.

Die **Zahlendarstellung**: Sei $x \in \mathsf{Natural}$, dann gilt

$$x = \sum_{k=0}^{\mathcal{L}(x)-1} x_k 2^{\beta k} \quad \text{mit } 0 \leq x_k \leq \gamma \text{ für alle } 0 \leq k < \mathcal{L}(x),$$

und für die interne Darstellung:



Hier bietet sich oftmals die kürzere Intervallschreibweise an, wie sie in [39] verwendet wird:

$$\mathsf{Natural} \ni x = [x\texttt{.root}, x\texttt{.p} + x\texttt{.size}[\quad = (0)_{k=0}^{\mathcal{R}(x)-\mathcal{L}(x)} \times \ (x_k)_0^{k=\mathcal{L}(x)-1} \in \mathsf{Digit}^{\mathcal{R}(x)},$$

wobei die folgende Definition gilt:

Definition.

$$\begin{aligned}
(x_0, x_1) \times (x_2, x_3) &:= (x_0, x_1, x_2, x_3) \\
[x_0, x_4[&:= (x_0, x_1, x_2, x_3) \\
(x_k)_0^{k=n} &:= (x_n, \ldots, x_0) \\
(x_k)_{k=0}^{n} &:= (x_0, \ldots, x_n) \\
0^n &:= \underbrace{(0, \ldots, 0)}_{n} \qquad \text{für ein } n \in \mathbb{N}.
\end{aligned}$$

Der vollständig verfügbare Speicher zur Zahlendarstellung eines Naturals x wird durch $\mathcal{R}(x)$ dargestellt und in der Klasse Natural repräsentiert durch die Funktion

⟨ memberfunction Natural.rootsize 17 ⟩ ≡
```
inline size_t Natural::rootsize() const
// Algorithm:  c := a.rootsize()
// Input:      a in Natural.
// Output:     c in size_t such that c = R(a) ||
{
  return (p+size) - root;
}
```
◇
Macro referenced in 402b.

Laufzeit. Sei $x \in \mathsf{Natural}$, dann $x\texttt{.rootsize()} \in \mathcal{O}(3)$.

Auf die Länge $\mathcal{L}(x)$ eines Naturals x kann man über die Funktion

⟨ memberfunction Natural.length 18a ⟩ ≡

```
inline size_t Natural::length() const
// Algorithm:  c := a.length()
// Input:      a in Natural.
// Output:     c in size_t such that c = L(a) ||
{
  return size;
}
◇
```

Macro referenced in 402b.

öffentlich zugreifen, weil die Datenelemente – insbesondere `size` – geschützt sind.

Laufzeit. Sei $x \in$ Natural, dann x`.length()`$\in \mathcal{O}(1)$.

Aus demselben Grund existieren die beiden folgenden Funktionen, die einen öffentlichen Zugriff auf die Elemente x_0 und $x_{\mathcal{L}(x)-1}$ eines Naturals x gewähren.

1. ⟨ memberfunction Natural.highest 18b ⟩ ≡

```
inline Digit Natural::highest() const
// Algorithm:  c := a.highest()
// Input:      a in Natural.
// Output:     c in Digit such that c = [a/2^(BETA*(L(a)-1))] ||
{
  return *p;
}
◇
```

   Macro referenced in 402b.

   Laufzeit. Sei $x \in$ Natural, dann x`.highest()`$\in \mathcal{O}(1)$.

2. ⟨ memberfunction Natural.lowest 18c ⟩ ≡

```
inline Digit Natural::lowest() const
// Algorithm:  c := a.lowest()
// Input:      a in Natural.
// Output:     c in Digit such that c = a and GAMMA ||
{
  return p[size-1];
}
◇
```

   Macro referenced in 402b.

   Laufzeit. Sei $x \in$ Natural, dann x`.lowest()`$\in \mathcal{O}(1)$.

Damit haben wir nichts anderes als ein Stellenwertsystem, in dem die Zahl durch ihre Ziffern dargestellt wird, wobei der Wert der Ziffer von der Stelle abhängt, an welcher sie innerhalb der Zahl geschrieben ist.

Beispiel. Wenn wir $\beta = 32$ haben und unsere zu repäsentierende Zahl $x = 10^{10}$ ist, so erhalten wir die interne Darstellung $x = (2, 1410065408) = 2 \cdot 2^{\beta} + 1410065408$ (2^{β}-*System*).

Konvention. Sei $x \in$ Natural, so verlangen wir aus Effizienzgründen und der Eindeutigkeit wegen, daß für $[x.\texttt{root}, x.\texttt{p}[= 0^{\mathcal{R}(x)-\mathcal{L}(x)}$ ist. Deshalb benötigen wir nach den meisten internen durchgeführten Operationen die folgende Memberfunktion[1]:

⟨ memberfunction Natural.normalize 19a ⟩ ≡

```
inline void Natural::normalize()
// Algorithm:  a.normalize()
// Input:      a in Natural.
// Output:     a in Natural such that not (a.p) = (0) or a.size = 1 ||
{
  Digit* pT = p;
  if (*pT == 0) {
    size_t sT = size;
    if (sT > 2) {
      do { ++pT; --sT; } while (*pT == 0 && sT > 1);
      p = pT; size = sT;
    } else if (sT == 2) { p = ++pT; size = --sT; }
  }
}
```

◇

Macro referenced in 402b.

die darauf achtet, daß keine unnötigen Nullen entstehen, und den Zeiger p wieder auf die korrekte Position setzt. Bei dieser Durchführung sprechen wir von **Normalisieren** eines Naturals.

Laufzeit. Sei $x \in$ Natural, dann $x.\texttt{normalize()} \in \mathcal{O}(1)$.

Bemerkung. Diese Normalisierungsfunktion ist äquivalent zum Ausdruck

```
while (*p == 0 && size > 1) { ++p; --size; },
```

aber ein wenig schneller.

Durch die folgende Funktion erfahren wir, wie viele Nullen am Ende eines Naturals sind.

⟨ memberfunction Natural.trailing_zeros 19b ⟩ ≡

```
size_t Natural::trailing_zeros(Digit*& a) const
// Algorithm:  c := b.trailing_zeros(a)
// Input:      b in Natural where not b = 0.
// Output:     a in [b.p, b.p+b.size[, c in size_t
//             such that not (a) = (0), [a+1, a+c] = 0^c
//             and a+c = b.p+b.size-1 ||
{
  NATURALCONDITION(*this);
  CONDITION(*this != 0);

  Digit* pT = p+size-1;
  if (*pT) { a = pT; return 0; }
  else {
    size_t c = 1;
    while (*--pT == 0) ++c;
```

[1] Bei dieser Impementation wird zum Beispiel vom Borland-Compiler (Version ≤ 5.0) eine Warnung ausgegeben, weil er keine Schleifen in einer `inline`-Funktion verarbeiten kann. Solche Warnungen sollte man als eine Schwäche des Compilers ansehen.

```
    a = pT;
    return c;
  }
}
◇
```

Macro referenced in 409.

Laufzeit. Sei $x \in$ Natural, dann x`.trailing_zeros()`$\in \mathcal{O}(1)$.

Diagnosemakro

Wir benötigen das folgende Diagnosemakro, um die Bedingung an die interne Darstellung eines Naturals zu überprüfen:

⟨ condition of a Natural 20a ⟩ ≡

```
# define NATURALCONDITION(a)                          \
  if ((a).root) {                                     \
      CONDITION(*(a).root == 0 &&                     \
                *((a).p-1) == 0 &&                    \
                (a).root < (a).p &&                   \
                (a).size > 0 &&                       \
                ((a).size == 1 || *(a).p != 0));      \
      const Digit* _pE = (a).p;                       \
      for (Digit* _pA = (a).root; _pA != _pE; ++_pA)  \
        CONDITION(*_pA == 0);                         \
  }

# define NATURAL_FOR_CHECK(a, b)                      \
  Natural (a) = (b);
◇
```

Macro referenced in 6b.

2.1.4 Speicher- und Strukturverwaltung

In unserer Klasse Natural besitzt jede Instanz eine eigene Kopie der Datenelemente. Nur die Datenelemente `NaturalSize` und `NaturalSizeOld` werden von allen Instanzen gemeinsam genutzt.

⟨ static variables for memory management of Naturals 20b ⟩ ≡

```
static size_t NaturalSize;
static size_t NaturalSizeOld;
◇
```

Macro referenced in 402b.

Die `static`-Deklaration der Elemente sorgt dafür, daß von ihnen jeweils nur ein Exemplar existiert. Dennoch sind die Elemente Bestandteil der Klasse Natural und können wegen der geschützten Deklaration nur von abgeleiteten Klassen beziehungsweise der eigenen Klasse Natural angesprochen werden. Solch eine `static`-Deklaration ist auch für Funktionen sinnvoll, weil so globale Variablen vermieden werden können.
`NaturalSize` gibt die Anzahl der Digits an, die ein öffentlicher Natural-Konstruktor alloziiert. Um den ursprüglichen Zustand wiederherzustellen, existiert die Variable `NaturalSizeOld`. In der Voreinstellung weisen wir einem Natural die Größe von δ Digits zu:

⟨ definition of static variables for memory management of Naturals 21a ⟩ ≡
```
size_t Natural::NaturalSize    = DELTA;
size_t Natural::NaturalSizeOld = DELTA;
```
◇
Macro referenced in 409.

Der Benutzer kann jedoch wegen der geschützten Deklaration dieser beiden Variablen nicht direkt darauf zugreifen. Deshalb werden die folgenden drei Funktionen unterstützt:

1. Die öffentliche Funktion

   ⟨ memberfunction Natural.NumberOfDecimals 21b ⟩ ≡
   ```
   inline size_t Natural::NumberOfDecimals(const size_t sz)
   {
     NaturalSizeOld = NaturalSize;
     return NaturalSize =
       min(size_t(sz/(BETA*0.301029995664))+1, size_t(GAMMA/BETA));
   }
   ```
   ◇
   Macro referenced in 402b.

   dient zur Festlegung der Grundgröße in Dezimalstellen für alle nach diesem Aufruf folgenden Natural-Instanzen.

   Laufzeit. Sei $t \in$ `size_t`, dann `Natural::NumberOfDecimals(`t`)`$\in \mathcal{O}(1)$.

2. Die öffentliche Funktion

   ⟨ memberfunction Natural.NumberOfDigits 21c ⟩ ≡
   ```
   inline size_t Natural::NumberOfDigits(const size_t sz)
   {
     NaturalSizeOld = NaturalSize;
     return NaturalSize = min(max(sz, size_t(1)), size_t(GAMMA/BETA));
   }
   ```
   ◇
   Macro referenced in 402b.

   verhält sich analog zu `NumberOfDecimals`; die Grundgröße wird jedoch in Digits angegeben.

   Laufzeit. Sei $t \in$ `size_t`, dann `Natural::NumberOfDecimals(`t`)`$\in \mathcal{O}(1)$.

3. Die öffentliche Funktion

   ⟨ memberfunction Natural.RestoreSize 21d ⟩ ≡
   ```
   inline void Natural::RestoreSize()
   {
     NaturalSize = NaturalSizeOld;
   }
   ```
   ◇
   Macro referenced in 402b.

   stellt die ursprüngliche Arraygröße wieder her.

   Laufzeit. `Natural::RestoreSize()`$\in \mathcal{O}(1)$.

Beispiel. Diese Funktionen sind dann sinnvoll, wenn wir mit großen Zahlen rechnen und den internen Überlauf vermeiden wollen. Wenn wir zum Beispiel eine effiziente Funktion zur Berechnung der Fibonacci-Zahlen programmieren, so können wir Überläufen entgehen, indem wir die folgende Abschätzung durch die Formel von Binet[2] anwenden:

$$\begin{aligned}\log_{10}(F_n) &= \log_{10}\left(\frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)\right)\\ &< n\cdot\log_{10}\left(\frac{1+\sqrt{5}}{2}\right) \quad < \quad n\cdot 0.209 \quad .\end{aligned}$$

Mit dieser Abschätzung reservieren wir schließlich nur so viel Speicher, wie wir tatsächlich für das Resultat benötigen.

⟨ calculates the n-th Fibonacci number 22a ⟩ ≡

```
Natural fibonacci(Digit n)
// Algorithm:  c := fibonacci(n)
// Input:      n in Digit.
// Output:     c in Natural such that c = F_n
//             where F_0 = 0, F_1 = 1, F_k = F_(k-1)+F_(k-2) for k >= 2 ||
{
  if (n <= 1) return n;
  Natural::NumberOfDecimals(size_t((n*209)/1000));
  ⟨ initialization of Fibonacci algorithm 13a ⟩
  if (n >= 50) {
    Natural t;
    Natural::RestoreSize();
    ⟨ fast Fibonacci algorithm 266 ⟩
  } else {
    Natural::RestoreSize();
    ⟨ additive Fibonacci algorithm 13b ⟩
  }
  return j;
}
```

◇

Macro referenced in 433.

Laufzeit. Sei $a \in \mathsf{Digit}$, dann

$$\texttt{fibonacci}(a) \in \begin{cases} \log_2(a)\cdot\mathcal{O}(\texttt{sqr} : \mathsf{Natural} \to \mathsf{Natural}) , & a \geq 50 \\ a\cdot\mathcal{O}(\mathsf{add} : \mathsf{Natural}^2 \to \mathsf{Natural}) , & a < 50 \end{cases} .$$

Die Speicherverwaltung ist vom Prinzip her sehr einfach. Wir verwenden ein homogenes Feld, das im Konstruktor alloziiert wird. Falls nun mittels einer Operation ein interner Überlauf entsteht, so wird im allgemeinen eine der beiden folgenden geschützten Funktionen aufgerufen:

1. Der interne Aufruf der Funktion

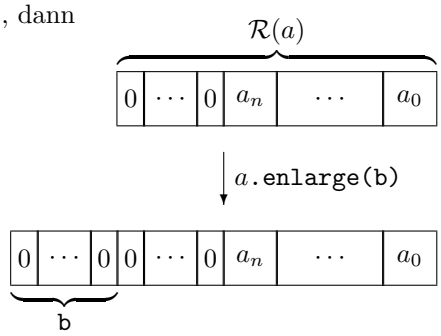
   ⟨ memberfunction Natural.enlarge 22b ⟩ ≡

[2] Diese Formel veröffentlichte Leonhard Euler 1765; danach geriet sie in Vergessenheit, bis sie 1843 von Jacques Binet wiederentdeckt wurde.

```
void Natural::enlarge(const size_t b)
// Algorithm:  a.enlarge(b)
// Input:      a in Natural, b in size_t where b >= 1.
// Output:     a in Natural such that R(a) := b+R(a) ||
{
  CONDITION(b >= 1);

  const size_t sA = p-root+b;
  const size_t sT = size;
  Digit* pT = NOTHROW_NEW Digit[sA+sT];
  if (!pT) errmsg(2, "(enlarge)");
  Digit* pA = pT+sA;
  FILL_ZERO(pT, pA);
  pA = p; p = pT;
  const Digit* pE = pA+sT;
  COPY(pT, pA, pA, pE);
  delete[] root;
  root = pT-sT-sA;
}
◇
```

Macro referenced in 409.

vergrößert die gesamte Arraygröße der Klasse um das übergebene Argument `b`.
Sei $a \in$ Natural mit $n := \mathcal{L}(a)$, dann



Laufzeit. Sei $x \in$ Natural und $t \in$ `size_t`, dann x`.enlarge(`t`)`$\in \mathcal{O}(\mathcal{R}(x) + t)$.

2. Die Funktion `enlarge` wird immer dann eingesetzt, wenn ein Überlauf entsteht oder der ursprüngliche Arrayinhalt nach einer Vergrößerung unverändert bleiben soll. Dies bedarf einer zusätzlichen Kopie, weil wir uns für die arrayorientierte Datenstruktur entschieden haben, und bringt offensichtliche Laufzeiteinbußen mit sich. Doch oftmals kann man den Überlauf noch vor der durchzuführenden Operation abfangen und damit die Laufzeiteinbußen gering halten beziehungsweise komplett eliminieren, wie dies zum Beispiel der Additionsalgorithmus im Kapitel 2.8.2 auf Seite 54 macht. Hierzu benötigen wir jedoch die Funktion

⟨ memberfunction Natural.setsize 23 ⟩ ≡

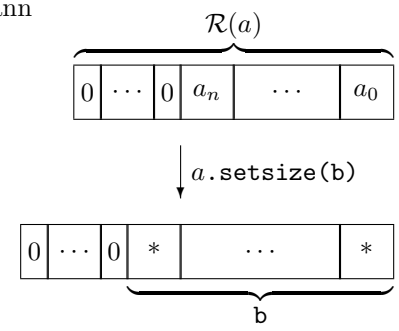
```
Digit* Natural::setsize(const size_t b)
// Algorithm:  r := a.setsize(b)
// Input:      a in Natural, b in size_t where b >= 1.
// Output:     a in Natural, r in [a.root, a.p+L(a)[
//             such that R(a) >= b+DELTA, L(a) = b, r = a.p ||
//
// Note:       a is not normalize!
{
  CONDITION(b >= 1);
```

```
  Digit* pT = p;
  const size_t sT = size;
  Digit* rT = root;
  if (pT+sT < rT+b+DELTA) {                // (rootsize() < b+DELTA)?
    delete[] rT;
    root = pT = NOTHROW_NEW Digit[b+DELTA];
    if (!pT) errmsg(2, "(setsize)");
    FILL_DELTA(pT);
    pT += DELTA;
  } else if (sT > b) {
    const Digit* pE = pT+sT-b;
    FILL_ZERO(pT, pE);
  } else pT -= b-sT;
  p = pT; size = b;
  return pT;
}
◇
```

Macro referenced in 409.

die die Länge der Klasse Natural auf das übergebene Argument `sz` setzt. Hierbei wird zwar darauf achtgegeben, daß der Bereich $[\texttt{root}, \texttt{p}[= 0^{\texttt{p}-\texttt{root}}$ korrekt ist; es wird jedoch nicht gefordert, daß die Klasse nach dem Aufruf `setsize` normalisiert ist und der Inhalt erhalten bleibt.

Sei $a \in$ Natural mit $n := \mathcal{L}(a)$, dann



Laufzeit. Sei $x \in$ Natural und $t \in$ `size_t`, dann x`.setsize(`t`)` $\in \begin{cases} \mathcal{O}(\mathcal{L}(x)-t)\,, & \mathcal{L}(x) > t \\ \mathcal{O}(1)\,, & \mathcal{L}(x) \le t \end{cases}$.

Bemerkung. Wenn wir zum Beispiel den additiven Fibonacci-Algorithmus auf der Seite 13 ohne die Funktionen `NumberOfDecimals` und `RestoreSize` anwenden, so entstehen während der Berechnung von `fibonacci(100000)` intern 924 Überläufe. Obwohl die Anzahl der Überläufe recht hoch ist, liegen unsere Laufzeiteinbußen lediglich bei etwa 4%.

2.2 Konstruktor und Destruktor der Klasse Natural

2.2.1 Die öffentlichen Konstruktoren und der Destruktor

Die Klasse Natural unterstützt drei öffentliche Konstruktoren und einen Destruktor:

⟨ public constructors and destructor of Natural 25a ⟩ ≡

```
Natural(const Digit = 0);                    // default constructor
Natural(const Natural&);                     // copy constructor
#ifndef _Old_STD_
explicit
#endif
Natural(const char*, const Digit = 10);      // constructor for string conversion
~Natural();                                  // destructor
```

◇

Macro referenced in 402b.

Bei zwei Konstruktoren ist in der Argument-Deklaration ein Ausdruck spezifiziert worden, was bedeutet, daß dieser Ausdruck als **Default-Argument** aufgenommen wurde. Dieses Argument wird also nur optional übergeben, und falls keine Übergabe stattfindet, wird die Vorgabe verwendet. Zusätzlich wird bei diesen Argumenten gefordert, daß alle nachfolgenden Argumente ebenfalls als Default-Argumente übergeben werden.

Die wichtigsten Konstruktoren einer Klasse sind der **Kopierkonstruktor**, der eine Referenz auf seine eigene Klasse enthält, und der **Standardkonstruktor**, an den kein Argument übergeben wird. Somit können wir nun unsere Klasse Natural auf die folgenden vier Arten initialisieren:

Beispiel.

```
int main()
{
  Natural c;                                          // Standardkonstruktor wird aufgerufen
  Natural a = 2000;                                   // Standardkonstruktor wird aufgerufen
  Natural b = prim;                                     // Kopierkonstruktor wird aufgerufen
  Natural prim("1234567891");              // Überladener Konstruktor zur ASCII-String-Konvertierung

  return 0;
}
```

Die Konstruktoren können aber auch funktional aufgerufen werden:

```
int main()
{
  Natural c(0);                                       // Standardkonstruktor wird aufgerufen
  Natural a(2000);                                    // Standardkonstruktor wird aufgerufen
  Natural b(prim);                                      // Kopierkonstruktor wird aufgerufen
  Natural prim("1234567891", 10);          // Überladener Konstruktor zur ASCII-String-Konvertierung

  return 0;
}
```

Die drei Konstruktoren, die für die interne Zahlendarstellung den benötigten Speicher anfordern, sind folgendermaßen implementiert:

1. Der Standardkonstruktor:

   ⟨ default constructor Natural 25b ⟩ ≡

   ```
   Natural::Natural(const Digit a)
   // Algorithm:  c := Natural(a)
   // Input:      a in Digit.
   // Output:     c in Natural such that c = a ||
   ```

```
{
  const size_t sT = NaturalSize+DELTA;
  Digit* pT = NOTHROW_NEW Digit[sT];
  root = pT; size = 1;
  if (!pT) errmsg(2, "(default constructor)");
  Digit* pE = pT+sT;
  *--pE = a; p = pE;
  FILL_ZERO(pT, pE);

  NATURALCONDITION(*this);
}
◇
```

Macro referenced in 409.

Laufzeit. Sei $a \in$ Digit, dann `Natural(`a`)`$\in \mathcal{O}$(`Natural::NaturalSize`).

2. Der Kopierkonstruktor:

⟨ copy constructor Natural 26a ⟩ ≡

```
Natural::Natural(const Natural& a)
// Algorithm:  c := Natural(a)
// Input:      a in Natural.
// Output:     c in Natural such that c = a ||
{
  NATURALCONDITION(a);

  const size_t sA = a.size;
  Digit* pT = NOTHROW_NEW Digit[sA+DELTA];
  root = pT; size = sA;
  if (!pT) errmsg(2, "(copy constructor)");
  FILL_DELTA(pT);
  p = pT += DELTA;
  const Digit* pA = a.p;
  const Digit* pE = pA+sA;
  COPY(pT, pA, pA, pE);

  NATURALCONDITION(*this);
}
◇
```

Macro referenced in 409.

Laufzeit. Sei $x \in$ Natural, dann `Natural(`x`)` $\sim$ `copy(0,`x`)`.

3. Der überladene Konstruktor zur ASCII-String-Konvertierung:

⟨ overloaded constructor Natural for ASCII-string conversion 26b ⟩ ≡

```
Natural::Natural(const char* a, const Digit b)
// Algorithm:  c := Natural(a, b)
// Input:      a in String, b in Digit.
// Output:     c in Natural such that c = a ||
{
  Digit* pT = NOTHROW_NEW Digit[DELTA];
  p = root = pT; size = DELTA;
  if (!pT) errmsg(2, "(constructor, string conversion)");
  atoN(a, b);
}
◇
```

Macro referenced in 409.

Hierbei liefert das Schlüsselwort `this` einen Zeiger auf das eigene Objekt.

Laufzeit. Sei $s \in$ String und $a \in$ Digit, dann `Natural(`s`,`a`)`$\in \mathcal{O}(n^2)$.

Der Destruktor räumt lediglich den reservierten Speicher mit `delete[]` wieder frei:

⟨ destructor Natural 27a ⟩ ≡
```
Natural::~Natural()
{
//  NATURALCONDITION(*this);

  delete[] root;
}
```
◇

Macro referenced in 409.

Laufzeit. `~Natural()`$\in \mathcal{O}(1)$.

2.2.2 Die geschützten Konstruktoren

Für die internen Algorithmen werden wir eher auf die folgenden vier geschützten Konstruktoren zugreifen, um Natural-Instanzen zu erhalten, mit denen überwiegend Überläufe vermieden werden:

1. Der Konstruktor

   ⟨ protected constructor Natural without the initialization of the elements 27b ⟩ ≡
   ```
   inline Natural::Natural(const size_t a, char)
   // Algorithm:  c := Natural(a, b)
   // Input:      b in char, a in size_t where a >= 1.
   // Output:     c in Natural such that L(c) = R(c) = a ||
   //
   // Note:       This constructor don't fulfill the conditions for Naturals.
   //
   // internal constructor without the initialization of the elements.
   {
     get_memory(a);
   }
   ```
   ◇

   Macro referenced in 402b.

   verwendet die folgende Funktion

   ⟨ memberfunction Natural.get_memory 27c ⟩ ≡
   ```
   void Natural::get_memory(const size_t a)
   // Algorithm:  c.get_memory(a)
   // Input:      a in size_t where a >= 1.
   // Output:     c in Natural such that L(c) = R(c) = a ||
   //
   // internal allocation without the initialization of the elements.
   {
     CONDITION(a >= 1);

     Digit* pT = NOTHROW_NEW Digit[size = a];
   ```

```
  p = root = pT;
  if (!pT) errmsg(2, "(get_memory)");
}
◇
```

Macro referenced in 409.

um lediglich Speicher zu alloziieren und läßt den Inhalt unbestimmt. Im allgemeinem ist das Objekt, das durch diesen Konstruktor erzeugt wurde, inkorrekt, weil es nicht normalisiert ist. Das zweite **ungenutzte Argument** wird lediglich aus Gründen der Eindeutigkeit verwendet.

Laufzeit. Sei $t \in$ `size_t` und $c \in$ `char`, dann `Natural(`t`,`c`)`$\sim$ `get_memory(`t`)`
.

Laufzeit. Sei $x \in$ Natural und $t \in$ `size_t`, dann x`.get_memory(`t`)`$\in \mathcal{O}(1)$.

2. Sei $x \in$ Natural, $a \in$ Digit und x durch den Konstruktor

⟨ protected constructors Natural for the optimal allocation 28a ⟩ ≡

```
Natural::Natural(const Digit a, size_t b)
// Algorithm:  c := Natural(a, b)
// Input:      a in Digit, b in size_t where b >= 2.
// Output:     c in Natural such that c = a*2^{BETA*(b-1)}, R(c) = b+DELTA ||
//
// Note:       This constructor don't fulfill the conditions for Naturals
//             e.g. a = 0.
{
  CONDITION(b >= 2);

  Digit* pT = NOTHROW_NEW Digit[b+DELTA];
  root = pT; size = b;
  if (!pT) errmsg(2, "(internal constructor)");
  FILL_DELTA(pT);
  p = pT += DELTA;
  const Digit* pE = pT+b;
  *pT++ = a;
  FILL_ZERO(pT, pE);
}
◇
```

Macro defined by 28ab, 29.
Macro referenced in 409.

erzeugt, dann hat x die Darstellung $[a] + 0^{\mathtt{sz}-1}$.

Laufzeit. Sei $a \in$ Digit und $t \in$ `size_t`, dann `Natural(`a`,`t`)`$\in \mathcal{O}(t)$.

3. Sei $x \in$ Natural durch den Konstruktor

⟨ protected constructors Natural for the optimal allocation 28b ⟩ ≡

```
Natural::Natural(size_t a, const Natural& b)
// Algorithm:  c := Natural(a, b)
// Input:      a in size_t, b in Natural where a >= 1.
// Output:     c in Natural such that c = b, R(c) = L(b)+a ||
{
  NATURALCONDITION(b);
  CONDITION(a >= 1);
```

```
  const size_t sB = b.size;
  Digit* pT = NOTHROW_NEW Digit[sB+a];
  root = pT; size = sB;
  if (!pT) errmsg(2, "(internal constructor)");
  const Digit* pE = pT+a;
  FILL_ZERO(pT, pE);
  pE += sB; p = pT;
  const Digit* pB = b.p;
  COPY(pT, pB, pT, pE);

  NATURALCONDITION(*this);
}
◇
```

Macro defined by 28ab, 29.
Macro referenced in 409.

erzeugt, dann hat x die Darstellung $0^{\mathtt{sz}} + (\mathtt{a}_k)_0^{k=\mathcal{L}(\mathtt{a})}$.

Laufzeit. Sei $t \in$ `size_t` und $x \in$ Natural, dann `Natural(`t`,`x`)`$\in \mathcal{O}(t + \mathcal{L}(x))$.

4. Sei $x \in$ Natural durch den Konstruktor

⟨ protected constructors Natural for the optimal allocation 29 ⟩ ≡

```
Natural::Natural(const Natural& a, size_t b)
// Algorithm:  c := Natural(a, b)
// Input:      a in Natural, b in size_t where b >= 1.
// Output:     c in Natural such that c = a*2^{BETA*b},
//             R(c) = L(a)+b+DELTA ||
//
// Note:       This constructor don't fulfill the conditions for Naturals (a=0).
{
  NATURALCONDITION(a);
  CONDITION(b >= 1);

  const size_t sA = a.size;
  const size_t sT = sA+b;
  Digit* pT = NOTHROW_NEW Digit[sT+DELTA];
  root = pT; size = sT;
  if (!pT) errmsg(2, "(internal constructor)");
  FILL_DELTA(pT);
  p = pT += DELTA;
  const Digit* pE = pT+sA;
  const Digit* pA = a.p;
  COPY(pT, pA, pT, pE);
  pE += b;
  FILL_ZERO(pT, pE);
}
◇
```

Macro defined by 28ab, 29.
Macro referenced in 409.

erzeugt, dann hat x die Darstellung $(\mathtt{b}_k)_0^{k=\mathcal{L}(\mathtt{b})} + 0^{\mathtt{sz}}$.

Laufzeit. Sei $x \in$ Natural und $t \in$ `size_t`, dann `Natural(`x`,`t`)`$\in \mathcal{O}(\mathcal{L}(x) + t)$.

2.3 Vertausch- und Kopierfunktion

2.3.1 Vertauschen zweier Naturals

Eine häufig wichtige Funktion ist das Vertauschen der Werte zweier Variablen. Die triviale Lösung ist, eine temporäre Variable dreimal zu kopieren. Eleganter und schneller ist es aber, die Zeiger zu vertauschen:

⟨ function swap for Naturals 30a ⟩ ≡

```
void swap(Natural& a, Natural& b)
// Algorithm:  swap(a, b)
// Input:      a,b in Natural.
// Output:     a,b in Natural such that t := a, a := b, b := t
//             where t in Natural ||
{
  NATURALCONDITION(a);
  NATURALCONDITION(b);

  Digit* pA = a.p; a.p = b.p; b.p = pA;
  pA = a.root; a.root = b.root; b.root = pA;
  size_t sz = a.size; a.size = b.size; b.size = sz;

  NATURALCONDITION(a);
  NATURALCONDITION(b);
}
```

◇

Macro referenced in 409.

Laufzeit. Seien $x, y \in$ Natural, dann swap$(x,y) \in \mathcal{O}(12)$.

2.3.2 Kopieren eines Naturals

Die Kopierfunktion ist einerseits eine der einfachsten Langzahloperationen, andererseits aber auch eine der wichtigsten, grundlegendsten und häufigsten Tätigkeiten unseres Systems. Daher will diese Funktion ganz besonders gut überlegt sein, weshalb wir diesen Algorithmus auch sehr detailliert betrachten werden. Ein weiterer Grund für unsere Ausführlichkeit an dieser Stelle ist, daß sich der Aufbau des Kopieralgorithmus und insbesondere die Fallunterscheidungen in der Implementation immer wieder in anderen elementar programmierten Algorithmen in der Langzahlarithmetik wiederholen werden.

Bei der Implementation der Kopierfunktion wird zusätzlich zur eigentlichen Kopiertätigkeit des übergebenen Naturals a noch ein Argument b verlangt, das die Anzahl der zu kopierenden Elemente angibt.

⟨ memberfunction Natural.copy 30b ⟩ ≡

```
Natural& Natural::copy(const Natural& a, const size_t b)
// Algorithm:  c.copy(a, b)
// Input:      a in Natural, b in size_t where 1 <= b <= L(a).
// Output:     c in Natural such that
//             c = [a/2^(BETA*(L(a)-b))]*2^(BETA*(L(a)-b))
//                 + c mod 2^(BETA*(L(a)-b)) ||
{
  NATURALCONDITION(a);
  CONDITION(b >= 1 && b <= a.size);
```

```
  const size_t sA = a.size;
  const size_t sT = size;
  Digit* rT = root;
  Digit* pT = p;
  ⟨ case 1 of Natural.copy 31 ⟩
  ⟨ case 2 of Natural.copy 32a ⟩ ⟨ case 3 of Natural.copy 32b ⟩
  p = pT; size = sA;
  const Digit* pA = a.p;
  const Digit* pE = pA+b;
  COPY(pT, pA, pA, pE);

  NATURALCONDITION(*this);

  return *this;
}
◇
```

Macro referenced in 409.

Laufzeit. Seien $x, y \in$ Natural und $t \in$ `size_t`, dann x`.copy(y, t)`$\in \mathcal{O}(\mathcal{L}(y))$.

Diese Funktion teilt sich in drei Fälle auf:

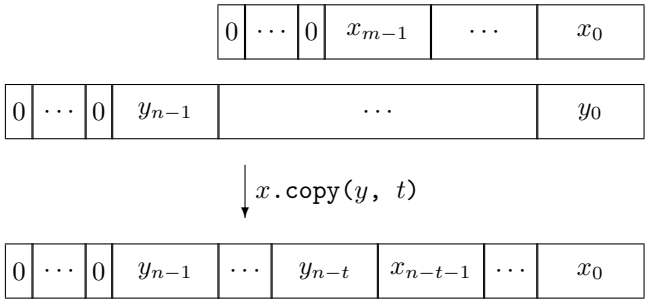
1. $\mathcal{R}($`*this`$) < \mathcal{L}($`b`$)$:

   ⟨ case 1 of Natural.copy 31 ⟩ ≡

   ```
   if (sA >= sT+size_t(pT-rT)) {        // (a.size >= rootsize())?
     Digit* pX = NOTHROW_NEW Digit[sA+DELTA];
     if (!pX) errmsg(2, "(copy)");
     root = pX;
     FILL_DELTA(pX);
     pX += DELTA;
     const size_t sz = sA-b;
     if (sz) {
       pX += b; pT += sT;
       const Digit* pE = pT;
       pT -= sz;
       COPY(pX, pT, pT, pE);
       pX -= sA;
     }
     pT = pX;
     delete[] rT;◇
   ```

   Macro referenced in 30b.

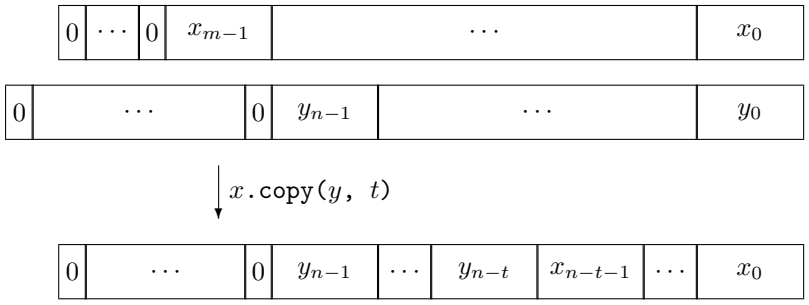
   Sei $x, y \in$ Natural und $t \in$ `size_t` mit $m := \mathcal{L}(x), n := \mathcal{L}(y) > \mathcal{R}(x)$, dann



2. $\mathcal{L}(\texttt{*this}) > \mathcal{L}(\texttt{b})$:

⟨ case 2 of Natural.copy 32a ⟩ ≡
```
} else if (sT > sA) {
  const Digit* pE = pT+sT-sA;
  FILL_ZERO(pT, pE);
}◇
```
Macro referenced in 30b.

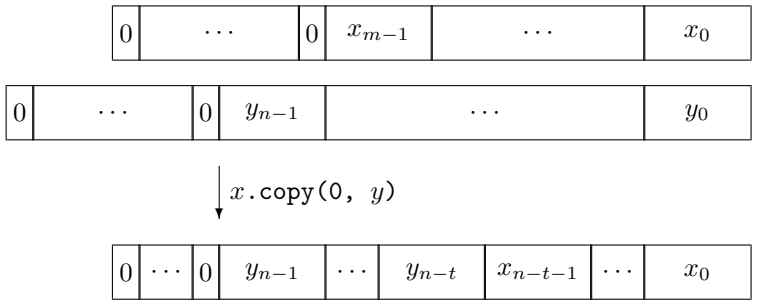
Sei $x, y \in$ Natural und $t \in$ size_t mit $m := \mathcal{L}(x) > \mathcal{L}(y) := n$, dann



3. $\mathcal{L}(\texttt{*this}) \leq \mathcal{L}(\texttt{b})$:

⟨ case 3 of Natural.copy 32b ⟩ ≡
```
else pT -= sA-sT;◇
```
Macro referenced in 30b.

Sei $x, y \in$ Natural und $t \in$ size_t mit $m := \mathcal{L}(x) \leq \mathcal{L}(y) =: n$, dann



Somit wird der operator= einfach an die copy-Funktion weitergeleitet:

⟨ assign operator= for Naturals 32c ⟩ ≡
```
inline Natural& Natural::operator=(const Natural& a)
// Algorithm:  c := c = a
// Input:      a,c in Natural.
// Output:     c in Natural such that c = a ||
{
  return copy(a, a.size);
}
◇
```
Macro referenced in 402b.

Laufzeit. Seien $x, y \in$ Natural, dann x=$y \sim x$.copy(0,y).

Es existiert auch noch ein operator= zur Zuweisung eines Digits. Der Rückgabewert dieser Funktion ist wieder ein Digit, weil ein Digit immer schneller als ein Natural verarbeitet werden kann.

⟨ assign operator= of a Natural with a Digit 33 ⟩ ≡

```
Digit Natural::operator=(const Digit a)
// Algorithm:  c := b = a
// Input:      a in Digit, b in Natural.
// Output:     b in Natural, c in Digit such that b = a, c = a ||
{
  Digit* pT = p;
  size_t sT = size;
  if (sT == 1) *pT = a;
  else {
    Digit* pE = pT+sT;
    *--pE = a; size = 1; p = pE;
    FILL_ZERO(pT, pE);
  }

  NATURALCONDITION(*this);

  return a;
}
```

◇

Macro referenced in 409.

Laufzeit. Sei $x \in$ Natural und $a \in$ Digit, dann x=$a \in \mathcal{O}(\mathcal{L}(x))$.

2.4 Effiziente Operatoren

Eine gewöhnliche und einfache Implementation der Operatoren erfolgt über den Kopierkonstruktor. Daher kann man beispielsweise den operator+ folgendermaßen realisieren:

```
inline Natural operator+(const Natural& a, const Natural& b)
{
  return Natural(a) += b;
}
```

Bei dieser Implementation wird allerdings für jede Plus-Operation der Kopierkonstruktor zwei Mal aufgerufen: Einmal um a zu kopieren und ein weiteres Mal, um das Ergebnis zurückzugeben. Diese zusätzlichen Aufrufe des Kopierkonstruktors verlangsamen die Addition etwa um den Faktor drei, weil der Kopierkonstruktor fast so aufwendig wie der Additionsalgorithmus ist. Wegen dieser deutlichen Laufzeiteinbuße sind Operatoren überflüssig, die auf solche Weise implementiert werden.

Daher werden wir nun mit Hilfe von Templates eine Methode erarbeiten, bei denen die Operatoren für einfache Ausdrüecke ohne den Kopierkonstruktor auskommen.
Die Idee ist recht simpel: Wir werden einfach die Argumente an die darauf folgende Operation weiterleiten.

Dafür benötigen wir zunächst einmal einen Binder, der die Argumente “temporär” festhält:

⟨ binder for the arguments of an operator 34a ⟩ ≡

```
template <class Arg1, class Arg2, class Tag>
struct binder_arguments {
  const Arg1& x;
  const Arg2& y;

  binder_arguments(const Arg1& a, const Arg2& b) : x(a), y(b) {}
};
```

◇

Macro referenced in 397d.

Zur Identifikation unserer Operation verwenden wir eine leere Klasse, wie zum Beispiel für die Addition die folgende:

```
struct Natural_plus_tag {};
```

Durch die Definition dieser beiden Klassen `binder_arguments` und `Natural_plus_tag`, brauchen wir nun im `operator+` lediglich den Binder mit den Eingabeargumenten zurückzugeben:

⟨ additive `operator+` for Naturals 34b ⟩ ≡

```
⟨ additive constructor and assignment 34c ⟩
inline binder_arguments<Natural, Natural, Natural_plus_tag>
 operator+(const Natural& a, const Natural& b)
// Algorithm:  c := a+b
// Input:      a,b in Natural.
// Output:     c in Natural such that c = a+b ||
{
  return binder_arguments<Natural, Natural, Natural_plus_tag>(a, b);
}
```

◇

Macro referenced in 402b.

Laufzeit. Seien $x, y, z \in$ Natural, dann $z{=}x{+}y \sim z.\mathtt{add}(x,y)$.

Die eigentliche Operation erfolgt dann entweder im Konstruktor oder bei der Zuweisung:

⟨ additive constructor and assignment 34c ⟩ ≡

```
inline Natural::Natural(const binder_arguments<Natural, Natural,
                                               Natural_plus_tag>& a)
{
  get_memory(max(a.x.size, a.y.size)+DELTA);
  add(a.x, a.y);
}

inline Natural& Natural::operator=(const binder_arguments<Natural, Natural,
                                                          Natural_plus_tag>& a)
{
  if (this == &a.x) return *this += a.y;
  else if (this == &a.y) return *this += a.x;
  else { add(a.x, a.y); return *this; }
}
```

◇

Macro referenced in 34b.

Der Zuweisungsoperator gewährt eine optimale Realisierung des folgenden Ausdrucks:

```
c = a+b;
```

Der Konstruktor ist aber auch erforderlich, um größere Ausdrücke zu ermöglichen. Größere Ausdrücke, wie zum Beispiel `d = a+b+c`, erzeugen allerdings bei dieser Methode temporäre Objekte, was die Laufzeit erhöht, aber im Endeffekt schneller ist als wenn, wie bei der einfachen Methode vier Mal der Kopierkonstruktor, aufgerufen wird.

Ein großer Vorteil bei dieser Argumentübergabe ist, daß ein gut optimierender Compiler die Argumente zur Compilezeit auswertet, wodurch keine zusätzliche Laufzeit benötigt wird und die Methode bei zusätzlichen Tests auf Spezialfälle dem einfachen Funktionsaufruf sogar überlegen ist; denn wir müssen nun nicht mehr innerhalb unseres Algorithmus die Argumente über Zeiger vergleichen.

2.5 Fehlerbehandlung

Jeder intern aufgetretene Fehler hat eine Identifikationsnummer und einen zusätzlichen Ausgabetext:

⟨ definitions of error-handling 35 ⟩ ≡

```
void default_piologie_error_handler(const int IDerr, const char* msg)
{
  switch(IDerr) {
    case 0: break;
//    case 1: cerr << "Overflow!"; break;
    case 2: cerr << "Out of memory!"; break;
    case 3: cerr << "Result negative (No Natural solution)!"; break;
    case 4: cerr << "Division by zero!"; break;
    case 5: cerr << "Same location error!"; break;
    case 6: cerr << "Discriminant negative (No Real solution)!"; break;

    default: cerr << "Internal error!";
  }
  cerr << msg << endl;
  exit(1);
}

static piologie_error_handler_t
 piologie_error_handler = default_piologie_error_handler;

piologie_error_handler_t set_piologie_error_handler(piologie_error_handler_t a)
{
  piologie_error_handler_t b = piologie_error_handler;
  piologie_error_handler = a;
  return b;
}

void NumberBase::errmsg(const int a, const char* b) const
{
  (*piologie_error_handler)(a, b);
}
```

◇

Macro referenced in 398b.

Hierbei wurde eine ähnliche Konstruktion wie `set_new_handler` aus der Standard-Headerdatei `<new.h>` umgesetzt. Somit kann der Benutzer den aufgetretenen Fehler in einer eigenen Funktion aufnehmen und Intelligenteres anschließen als einfach das Programm zu beenden.

Bemerkung. Diese Lösungsmethode ist veraltet und wird in einer späteren Version geändert werden, da C++ heutzutage Ausnahmebehandlungen unterstützt, die dem Benutzer bessere Möglichkeiten bieten, auf Fehler zu reagieren.

2.6 Ein- und Ausgabe und die Konvertierung

2.6.1 Streamausgabe

Dezimalsystem

Um die interne Darstellung eines Naturals in das Dezimalsystem zu transferieren, benötigen wir die Abbildung

$$\mathsf{Natural} \ni x = \sum_{k=0}^{\mathcal{L}(x)-1} x_k 2^{\beta k} \mapsto \sum_{k=0}^{s} d_k \alpha^k = x'$$

mit $0 \leq x_k \leq \gamma, 0 \leq d_k < \alpha$ für alle $0 \leq k \leq s$ und $s := \left\lfloor \frac{\beta \cdot \mathcal{L}(x)}{\log_2 \alpha} \right\rfloor$, wobei s nur eine obere Schranke bildet und demzufolge x' noch normalisiert werden muß.

Der Algorithmus ist sehr einfach:
Eingabe: $a \in \mathsf{Natural}$.

1. Ausgabe von a mod α Digitweise von rechts nach links,
   a `/=` α.
2. Führe Schritt 1 solange durch, bis $a = 0$ ist.

⟨ puts a Natural on output stream 36 ⟩ ≡

```
static void output_stream(OSTREAM& out, const Natural& a)
{
  const size_t s = 1 + a.length()*BETA/size_t(log2(ALPHA));
  const size_t t = out.width();
  if (s < 17000) {
    Digit* pC = NOTHROW_NEW Digit[s];
    if (!pC) a.errmsg(2, "(stream operator<<)");
    Digit* pE = pC + s;
    Digit* pOut = pE;
    size_t sz = 0;
    Natural b = a;
    while (b.length() != 1) {
      div(b, ALPHA, b, *--pOut);
      sz += ALPHA_WIDTH;
    }
    out.width((sz < t)? t-sz : 0);
    out << b.highest();
    while (pOut != pE) {
      out.width(ALPHA_WIDTH);
      out.fill('0');
```

```
      out << *pOut;
      ++pOut;
    }
    out.width(0);
    delete[] pC;
    return;
  }

  Natural c = pow(Natural(ALPHA), s/2);
  Natural q,r;
  div(a, c, q, r);

  const size_t sz = ALPHA_WIDTH*(s/2);
  out.width((sz < t)? t-sz : 0);
  output_stream(out, q);
  out.width(sz);
  out.fill('0');
  output_stream(out, r);
  out.width(0);
}

OSTREAM& operator<<(OSTREAM& out, const Natural& a)
// Algorithm:  o := o << a
// Input:      o in ostream, a in Natural.
// Output:     o in ostream ||
//
// Note:       Only decimal output is supported.
{
  output_stream(out, a);
  out.fill(' ');
  return out;
}
◇
```

Macro referenced in 409.

Laufzeit. Sei $o \in$`ostream` und $x \in$ Natural, dann o<<$x \in \mathcal{O}\left(\frac{\mathcal{L}(x)}{2} \cdot \left\lfloor \frac{\beta \mathcal{L}(x)}{\log_2 \alpha} \right\rfloor\right)$.

Interne Darstellung

Zusätzlich besteht noch die Möglichkeit, ein Natural in seiner internen Darstellung auszugeben:

⟨ puts internal representation of a Natural on output stream 37 ⟩ ≡

```
OSTREAM& operator<<(OSTREAM& out, const Natural::rep& a)
// Note: puts internal representation of Natural a on an output stream.
{
  const size_t sA = a.size;
  const Digit* pA = a.p;
  const Digit* pE = pA+sA;
  out << CHAR_BIT << '*' << sizeof(Digit) << '*' << sA;
  if (a.bin) {
    out << 'B';
    do {
      const Digit d = *--pE;
      for (size_t i = 0; i < sizeof(Digit); ++i) out << char((d >> (CHAR_BIT*i))&0xff);
```

```
    } while (pA != pE);
  } else {
    out << '(' << *--pE;
    while (pA != pE) out << ',' << *--pE;
  }
  return out << ')';
}
```
◇

Macro referenced in 409.

Laufzeit. Sei $o \in$`ostream` und $r \in$ Natural`::rep`, dann o`<<`$r \in \mathcal{O}(\mathcal{L}(r))$.

Bemerkung. Aus Portabilitätsgründen werden die Zahlen nur im Textformat ausgegeben, da die Ausgabe der beiden Funktionen `put(char)` und `get(char)` der Stream-Bibliothek von der Plattform abhängig sind und vor allem bei dem *newline*-Zeichen (`'\n'`) Probleme bereiten. Ein Nachteil der Verwendung des Textformates ist allerdings, daß unsere Ausgabedatei doppelt so groß wird.

Hierzu wird die interne Struktur

⟨ output variables for representation of a Natural 38a ⟩ ≡
```
struct rep {
  const size_t size;
  const Digit* p;
  const bool   bin;
  rep(const size_t a, const Digit* b, const bool c = false)
    : size(a), p(b), bin(c) {}
};
```
◇

Macro referenced in 402b.

verwendet, die dann durch die öffentliche Funktion `print` angesprochen werden kann.

⟨ function `print` of a Natural 38b ⟩ ≡
```
inline Natural::rep print(const Natural& a, bool b)
// Algorithm:  o := o << print(a, b)
// Input:      o in ostream, a in Natural, b in bool.
// Output:     o in ostream ||
//
// Note:       puts internal representation of Natural a on output stream.
{
  return Natural::rep(a.size, a.p, b);
}
```
◇

Macro referenced in 402b.

Beispiel. `cout << print(a) << endl;`.

Ein mit `print` ausgegebenes Natural kann mit der öffentlichen Memberfunktion `scan` wieder eingelesen werden. Der Rückgabewert dieser Funktion `scan` ist `true`, falls das Einlesen erfolgreich war. Falls das Einlesen nicht erfolgreich durchgeführt werden konnte ist das Eingabe-Natural undefiniert.

⟨ gets internal representation of a Natural from input stream 38c ⟩ ≡
```
bool Natural::scan(ISTREAM& in)
// Algorithm:  b := a.scan(i)
```

```
// Input:      a in Natural, i in istream.
// Output:     a in Natural, i in istream, b in bool ||
//
// Note:       gets Natural a as an internal representation from input stream
//             if b is true.
{
  if (!in.good()) return false;
  char c = 0;
  size_t l = 0;
  if (!(in >> l) || l != CHAR_BIT || !in.get(c) || c != '*') return false;
  l = 0;
  if (!(in >> l) || l < 1 || (l%sizeof(Digit) && sizeof(Digit)%l)
      || !in.get(c) || c != '*')  return false;
  size_t sT = 0;
  if (!(in >> sT) || sT < 1) return false;
  sT *= l;
  if (sT % sizeof(Digit)) sT += sizeof(Digit);
  sT /= sizeof(Digit);
  Digit* pT = setsize(sT);
  Digit* pE = pT+sT;
  if (in.get(c)) {
    if (c == 'B') {
      if (l <= sizeof(Digit)) {
        do {
          Digit d = 0;
          for (size_t i = 0; i < sizeof(Digit); ++i) {
            if (!in.get(c)) return false;
            d |= Digit((unsigned char)c) << (CHAR_BIT*i);
          }
          *--pE = d;
        } while (--sT);
        if (!in.get(c) || c != ')') ++pE;
      } else {
        Natural x,y;
        do {
          size_t i;
          x = 0;
          for (i = 0; i < sizeof(Digit); ++i) {
            if (!in.get(c)) return false;
            y = Digit((unsigned char)c); y <<= CHAR_BIT*i;
            x |= y;
          }
          for (i = 0; i < l; i += sizeof(Digit)) {
            *--pE = x & GAMMA; x >>= BETA;
          }
        } while (--sT);
        if (!in.get(c) || c != ')') ++pE;
      }
    } else if (c == '(') {
      if (l == sizeof(Digit)) {
        do
          if (!(in >> *--pE)) break;
        while (in.get(c) && c == ',' && --sT);
        if (c != ')') ++pE;
      } else if (l < sizeof(Digit)) {
        do {
```

```
            Digit x = 0;
            for (size_t i = 0; i < sizeof(Digit); i += l) {
              Digit y;
              in >> y;
              x |= y << (CHAR_BIT*i);
              if (!in.get(c) || c != ',') break;
            }
            *--pE = x;
          } while (c == ',' && --sT);
        } else {
          Natural x;
          do {
            in >> x;
            for (size_t i = 0; i < l; i += sizeof(Digit)) {
              *--pE = x & GAMMA; x >>= BETA;
            }
          } while (in.get(c) && c == ',' && --sT);
        }
      }
    }
    normalize();

    NATURALCONDITION(*this);

    return (pT == pE && in.good() && c == ')');
  }
  ◇
```

Macro referenced in 409.

Laufzeit. Sei $i \in$`ostream`, $x \in$ Natural, dann x`.scan(`i`)`$\in \mathcal{O}(n)$.

Bemerkung. Der Einfachheit wegen wird in der Funktion `scan` vorausgesetzt, daß `CHAR_BIT` denselben konstanten Wert enthält wie die Maschine, die die interne Darstellung des Naturals durch die Funktion `print` geschrieben hat.

2.6.2 Streameingabe

Bemerkung. In der Implementation wird noch zusätzlich die Konvention getroffen, daß das *newline*-Zeichen (`'\n'`) am Ende des übergebenen Eingabestreams entfernt wird.

⟨ gets a Natural from input stream 40 ⟩ ≡

```
ISTREAM& operator>>(ISTREAM& in, Natural& a)
// Algorithm:  i := i >> a
// Input:      i in istream.
// Output:     i in istream, a in Natural ||
//
// Note:       Only decimal input is supported.
{
  if (!in.good()) return in;
  Digit  d = 0;
  size_t i = 0;
  char ch  = 0;
  a = 0;
  while (!in.eof() && in.get(ch) && isdigit(ch)) {
```

```
    d *= 10; d += ch - '0';
    if (++i == ALPHA_WIDTH) {
      a *= ALPHA; a += d;
      d = i = 0;
    }
  }
  if (i) { a *= pow10(i); a += d; }
  if (in.good() && ch != '\n') in.putback(ch);
  return in;
}
◇
```

Macro referenced in 409.

Laufzeit. Sei $i \in$`istream` und $x \in$ Natural, dann i`>>`$x \in \mathcal{O}(n^2)$.

2.6.3 Konvertierung in ein beliebiges Stellenwertsystem

Sei b die gewählte Basis, so gilt nun die Abbildung:

$$\mathsf{Natural} \ni x = \sum_{k=0}^{\mathcal{L}(x)-1} x_k 2^{\beta k} \mapsto \sum_{k=0}^{s} d_k b^k = x'$$

mit $0 \leq x_k \leq \gamma, 0 \leq d_k < b$ für alle $0 \leq k \leq s$ und $s := \left\lfloor \frac{\beta \cdot \mathcal{L}(x)}{\log_2 b} \right\rfloor$.

Falls eine Basis $b > 10$ vorgegeben ist, so verwenden wir für die gebräuchliche Darstellung der Zahlen $10 \ldots 35$ den Buchstaben $A \ldots Z$. Deswegen ist nur die Basis $2 \leq b \leq 36$ in unserer Implementierung möglich, und es erfolgt bei einer korrekten Parameterübergabe die Konvertierung in einen Nullterminierenden String.

Wenn die Basis in $\{2, 4, 8, 16, 32\}$ liegt, dann können wir den String einfach aus der internen binären Darstellung erzeugen.

⟨ converts a Natural to a string 41 ⟩ ≡

```
static void Ntoa2(const Natural& a, char* c, const Digit b, size_t width, bool active)
// Algorithm:  c := Ntoa2(a, c, b)
// Input:      a in Natural, b in Digit, c in String
//             where b = 2^x for x in N, sizeof(c) > BETA*L(a)/x.
// Output:     c in String such that c = a ||
//
// Note:       conversion Natural to string.
{
  const Digit j = log2(b);
  Digit k = log2(a)+1;
  Digit k2 = (k/j)*j;
  char* str = c;
  if (active && width) {
    char* str2 = str + width-k/j - (k != k2);
    do *str = '0'; while (++str != str2);
  }
  Digit d;
  if (k != k2) {
    d = 0;
    do {
```

```
      d <<= 1; d |= Digit(a.testbit(--k));
    } while (k != k2);
    *str++ = (d <= 9)? char(d + '0') : char(d + 'A' - 10);
  }
  while (k > 0) {
    d = 0;
    k2 = k-j;
    do {
      d <<= 1; d |= Digit(a.testbit(--k));
    } while (k != k2);
    *str++ = (d <= 9)? char(d + '0') : char(d + 'A' - 10);
  }
  *str = 0;
}
◇
```

Macro defined by 41, 42b, 43.
Macro referenced in 409.

Für größere Zahlen nutzen wir den schnellen Multiplikationsalgorithmus, indem wir die zu konvertierende Zahl zerlegen.

⟨ trade off point for the Natural to string conversion 42a ⟩ ≡

```
const size_t NTOA_MARK = 17000;
◇
```

Macro referenced in 409.

⟨ converts a Natural to a string 42b ⟩ ≡

```
static void Ntoa(Natural a, char* c, const Digit b, size_t width, bool active)
// Algorithm:  c := Ntoa(a, c, b)
// Input:      a in Natural, b in Digit, c in String
//             where 2 <= b <= 36, sizeof(c) > BETA*L(a)/log2(b).
// Output:     c in String such that c = a ||
//
// Note:       conversion Natural to string.
{
  char* str = c;
  Digit  b2 = b;
  const Digit w = GAMMA/b;
  size_t j = 1;
  while (b2 <= w) { b2 *= b; ++j; }
  const size_t s = 1 + a.length()*BETA/size_t(log2(b2));
  if (s < NTOA_MARK) {
    Digit d;
    while (true) {
      div(a, b2, a, d);
      if (a == 0) break;
      for (size_t i = 0; i < j; ++i) {
        const char c = char(d%b);
        *str++ = (c <= 9)? char(c + '0') : char(c + 'A' - 10);
        d /= b;
      }
      width -= j;
    }
    while (d > 0) {
      const char c = char(d%b);
      *str++ = (c <= 9)? char(c + '0') : char(c + 'A' - 10);
```

```
      d /= b;
      --width;
    }
    if (active && width) {
      char* str2 = str+width;
      do *str = '0'; while (++str != str2);
    }

    *str = 0;
    for (char* str2 = c; str2 < --str; ++str2) {
      const char c = *str2;
      *str2 = *str;
      *str = c;
    }
  } else {
    Natural c = pow(Natural(b2), s/2);
    Natural q,r;
    div(a, c, q, r);

    const size_t sz = j*(s/2);
    Ntoa(q, str, b, (sz < width)? width-sz : 0, active);
    Ntoa(r, str+strlen(str), b, sz, true);
  }
}
◇
```

Macro defined by 41, 42b, 43.
Macro referenced in 409.

Die öffentliche Konvertierungsfunktion überprüft, ob die Voraussetzungen erfüllt sind und wählt je nach vorgegebener Basis einen der beiden vorherigen Algorithmen aus.

⟨ converts a Natural to a string 43 ⟩ ≡

```
char* Ntoa(const Natural& a, char* c, const Digit b)
// Algorithm:  c := Ntoa(a, c, b)
// Input:      a in Natural, b in Digit, c in String
//             where 2 <= b <= 36, sizeof(c) > BETA*L(a)/log2(b).
// Output:     c in String such that c = a ||
//
// Note:       conversion Natural to string.
{
  CONDITION(b >= 2 && b <= 36);

  *c = 0;
  if (b < 2 || b > 36) return c;
  if ((b & (~b+1)) == b) Ntoa2(a, c, b, 0, false);
  else Ntoa(a, c, b, 0, false);
  return c;
}
◇
```

Macro defined by 41, 42b, 43.
Macro referenced in 409.

Laufzeit. Sei $x \in$ Natural, $s \in$ String und $a \in$ Digit, dann `Ntoa(`x`,`s`,`a`)` $\sim$ x`.mul` : Natural$^2 \to$ Natural.

2.6.4 Konvertierung aus einem beliebigen Stellenwertsystem

Die größtmögliche Potenz der Basis, die in ein Digit paßt, wird zur Konvertierung eingesetzt, um die Laufzeit möglichst gering zu halten:

⟨ converts a string to a Natural 44 ⟩ ≡

```
const char* Natural::atoN(const char* a, const Digit b)
// Algorithm:  c := d.atoN(a, b)
// Input:      d in Natural, a in String, b in Digit where 2 <= b <= 36.
// Output:     d in Natural, c in String such that d = a ||
//
// Note: Returns a pointer to the first occurrence of a non-digit character
//       in a.
{
  CONDITION(b >= 2 && b <= 36);

  *this = 0;
  if (b < 2 || b > 36) return a;
  const char* a2 = a;
  if (b < 10) {
    while (true) {
      const char d = *a2;
      if (d == 0 || !isdigit(d) || d-'0' >= b) break;
      ++a2;
    }
  } else {
    while (true) {
      const char d = *a2;
      if (d == 0 || !isdigit(d) && toupper(d)-'A'+10 >= b) break;
      ++a2;
    }
  }
  if ((b & (~b+1)) == b) {
    const Digit j = log2(b);
    Digit l = (a2-a)*j;
    while (a != a2) {
      const Digit d = Digit((isdigit(*a))? *a - '0' : toupper(*a)-'A'+10);
      Digit i = Digit(1) << j;
      do {
        i >>= 1; --l;
        if (d&i) setbit(l);
      } while (i != 1);
      ++a;
    }
    return a;
  }

  Digit  b2 = b;
  const Digit w = GAMMA/b;
  size_t j = 1;
  while (b2 <= w) { b2 *= b; ++j; }
  size_t i = 0;
  size_t l = (a2-a)/j;
  Digit d = 0;
  if (l < ATON_MARK) {
    while (a != a2) {
```

```
      d *= b; d += Digit((isdigit(*a))? *a - '0' : toupper(*a)-'A'+10);
      if (++i == j) {
        *this *= b2; *this += d;
        d = i = 0;
      }
      ++a;
    }
  } else {
    ⟨ fast conversion of a string to a Natural 45b ⟩
  }
  if (i) {
    b2 = 1;
    do b2 *= b; while (--i);
    *this *= b2; *this += d;
  }

  NATURALCONDITION(*this);

  return a;
}
◇
```

Macro referenced in 409.

Laufzeit. Sei $x \in \mathsf{Natural}$, $s \in \mathsf{String}$ und $a \in \mathsf{Digit}$, dann $x\texttt{.atoN}(s,a) \sim x\texttt{.mul} : \mathsf{Natural}^2 \to \mathsf{Natural}$.

Für größere Zahlen nutzen wir den schnellen Multiplikationsalgorithmus, indem wir den Eingabestring in Teile zerlegen und hierarchisch zusammenlegen:

⟨ trade off point for the string to Natural conversion 45a ⟩ ≡

```
const size_t ATON_MARK = 20000;
◇
```

Macro referenced in 409.

⟨ fast conversion of a string to a Natural 45b ⟩ ≡

```
Digit* v = new Digit[l];
Digit* v2 = v;
while (a != a2) {
  d *= b; d += Digit((isdigit(*a))? *a - '0' : toupper(*a)-'A'+10);
  if (++i == j) {
    *v2 = d; ++v2;
    d = i = 0;
  }
  ++a;
}

CONDITION(l > 1);
size_t l2 = l >> 1;
Natural* c = new Natural[l2];
size_t k2 = l2;
size_t k = l-1;
while (true) {
  if (--k2 == 0) {
    if (k == 2) { c[0] = v[k-2]; c[0] *= b2; c[0] += v[k-1]; }
    else c[0] = v[k-1];
    c[0] *= b2; c[0] += v[k];
    break;
```

```
  }
  c[k2] = v[k-1]; c[k2] *= b2; c[k2] += v[k];
  k -= 2;
}
delete[] v;
Natural b3 = b2;
while (l > 3) {
  l = l2;
  l2 >>= 1;
  Natural* c2 = new Natural[l2];
  b3 *= b3;
  k2 = l2;
  k = l-1;
  while (true) {
    if (--k2 == 0) {
      if (k == 2) { c2[0] = c[k-2]; c2[0] *= b3; c2[0] += c[k-1]; }
      else c2[0] = c[k-1];
      c2[0] *= b3; c2[0] += c[k];
      break;
    }
    c2[k2] = c[k-1]; c2[k2] *= b3; c2[k2] += c[k];
    k -= 2;
  }
  delete[] c;
  c = c2;
}
*this = c[0];
delete[] c;
◇
```

Macro referenced in 44.

⟨ converts a string to a Natural by function call 46 ⟩ ≡

```
inline Natural atoN(const char* a, const Digit b)
// Algorithm:  c := atoN(a, b)
// Input:      a in String, b in Digit where 2 <= b <= 36.
// Output:     c in Natural such that c = a ||
//
// Note:       conversion string to Natural; return 0 by conversion error.
{
  Natural result;
  result.atoN(a, b);
  return result;
}
◇
```

Macro referenced in 402b.

Laufzeit. Sei $x \in$ Natural, $s \in$ String und $a \in$ Digit, dann `atoN(`s`,`a`)` $\sim$ x`.atoN(`s`,`a`)`.

2.7 Vergleichsoperatoren

Das Grundgerüst bildet die Funktion `compare`, die gleichbedeutend ist mit der mathematischen Funktion

$$\texttt{compare}(a,b) = \text{sign}(a-b) = \begin{cases} 1\,, & a > b \\ 0\,, & a = b \\ -1\,, & a < b \end{cases}$$

mit $a, b \in$ Natural.

⟨ compares two Naturals 47a ⟩ ≡

```
int Natural::compare(const Natural& b) const
// Algorithm:  c := a.compare(b)
// Input:      a,b in Natural.
// Output:     c in int such that c = sign(a - b) ||
{
  NATURALCONDITION(*this);
  NATURALCONDITION(b);

  const size_t sT = size;
  const size_t sB = b.size;
  if (sT > sB) return 1;
  else if (sT < sB) return -1;

  Digit* pT = p;
  Digit* pB = b.p;
  for (const Digit* pE = pT+sT; *pT == *pB; ++pB)
    if (++pT == pE) return 0;
  return (*pT < *pB)? -1 : 1;
}
◇
```

Macro referenced in 409.

Laufzeit. Seien $x, y \in$ Natural, dann x`.compare(`y`)` $\in \begin{cases} \mathcal{O}(\mathcal{L}(x))\,, & x = y \\ \mathcal{O}(1)\,, & x \neq y \end{cases}$

2.7.1 Gleichheit

⟨ comparison `operator==` for Naturals 47b ⟩ ≡

```
inline bool operator==(const Natural& a, const Natural& b)
// Algorithm:  c := a == b
// Input:      a,b in Natural.
// Output:     c in bool such that if a = b then c = true else c = false ||
{
  return (a.compare(b) == 0);
}
◇
```

Macro referenced in 402b.

Laufzeit. Seien $x, y \in$ Natural, dann x`==`$y \sim$ `compare(`x`,`y`)`.

⟨ comparison operator!= for Naturals 48a ⟩ ≡
```
inline bool operator!=(const Natural& a, const Natural& b)
// Algorithm:  c := a != b
// Input:      a,b in Natural.
// Output:     c in bool such that if a = b then c = false else c = true ||
{
  return (a.compare(b) != 0);
}
```
◇
Macro referenced in 402b.

Laufzeit. Seien $x, y \in$ Natural, dann x!=$y \sim$ compare(x,y).

2.7.2 Ordnungs Relationen

⟨ comparison operator< for Naturals 48b ⟩ ≡
```
inline bool operator<(const Natural& a, const Natural& b)
// Algorithm:  c := a < b
// Input:      a,b in Natural.
// Output:     c in bool such that if a < b then c = true else c = false ||
{
  return (a.compare(b) < 0);
}
```
◇
Macro referenced in 402b.

Laufzeit. Seien $x, y \in$ Natural, dann x<$y \sim$ compare(x,y).

⟨ comparison operator<= for Naturals 48c ⟩ ≡
```
inline bool operator<=(const Natural& a, const Natural& b)
// Algorithm:  c := a <= b
// Input:      a,b in Natural.
// Output:     c in bool such that if a <= b then c = true else c = false ||
{
  return (a.compare(b) <= 0);
}
```
◇
Macro referenced in 402b.

Laufzeit. Seien $x, y \in$ Natural, dann x<=$y \sim$ compare(x,y).

⟨ comparison operator> for Naturals 48d ⟩ ≡
```
inline bool operator>(const Natural& a, const Natural& b)
// Algorithm:  c := a > b
// Input:      a,b in Natural.
// Output:     c in bool such that if a > b then c = true else c = false ||
{
  return (a.compare(b) > 0);
}
```
◇
Macro referenced in 402b.

Laufzeit. Seien $x, y \in$ Natural, dann x>$y \sim$ compare(x,y).

⟨ comparison operator>= for Naturals 49a ⟩ ≡
```
inline bool operator>=(const Natural& a, const Natural& b)
// Algorithm:  c := a >= b
// Input:      a,b in Natural.
// Output:     c in bool such that if a >= b then c = true else c = false ||
{
  return (a.compare(b) >= 0);
}
◇
```
Macro referenced in 402b.

Laufzeit. Seien $x, y \in$ Natural, dann x>=$y \sim$ compare(x,y).

Bemerkung. Wir implementieren alle sechs Vergleichsoperatoren (==, !=, <, <=, >, >=), obwohl lediglich die Gleichheits- und Kleiner-Relation bei der Verwendung der STL (siehe [48]) relevant sind. Der Grund für diesen Aufwand ist, daß Templates keine Konstruktoren implizit aufrufen und wir sie deshalb nicht verwenden können. Der Compiler würde sonst zum Beispiel für den Ausdruck (a/b != c) für a,b,c∈ Natural wegen der im Kapitel 2.4 auf der Seite 33 eingeführten Technik zur Erzeugung effizienter Operatoren keinen passenden Vergleichsoperator finden.

Bemerkung. Es wäre durchaus möglich, die Vergleichsoperatoren auch als Elemente der Klasse Natural zu definieren:

```
inline bool Natural::operator<(const Natural& a) const
{
   return (compare(a) < 0);
}
```

Dann wäre allerdings die folgende Konstruktion zum Beispiel nicht möglich:

```
Natural a = 5;
if (3 < a) //...
```

2.7.3 Digit-Vergleiche

Oft wird ein Natural nur mit einer Konstanten ($\leq \gamma$) oder einem Digit verglichen.

Beispiel. `for (Natural b = a; b > 100; b = sqrt(b)) ++i;`

Deshalb sind die folgenden sechs Vergleichsoperatoren schnell und sehr nützlich:

⟨ comparison operator== of a Natural with a Digit 49b ⟩ ≡
```
inline bool operator==(const Natural& a, const Digit b)
// Algorithm:  c := a == b
// Input:      a in Natural, b in Digit.
// Output:     c in bool such that if a = b then c = true else c = false ||
{
  return (a.size == 1 && *a.p == b);
}
◇
```
Macro referenced in 402b.

Laufzeit. Sei $x \in$ Natural und $a \in$ Digit, dann x==$a \in \mathcal{O}(2)$.

⟨ comparison operator!= of a Natural with a Digit 50a ⟩ ≡

```
inline bool operator!=(const Natural& a, const Digit b)
// Algorithm:  c := a != b
// Input:      a in Natural, b in Digit.
// Output:     c in bool such that if not a = b then c = true else c = false ||
{
  return (a.size != 1 || *a.p != b);
}
```

◇

Macro referenced in 402b.

Laufzeit. Sei $x \in$ Natural und $a \in$ Digit, dann x!=$a \in \mathcal{O}(2)$.

⟨ comparison operator< of a Natural with a Digit 50b ⟩ ≡

```
inline bool operator<(const Natural& a, const Digit b)
// Algorithm:  c := a < b
// Input:      a in Natural, b in Digit.
// Output:     c in bool such that if a < b then c = true else c = false ||
{
  return (a.size == 1 && *a.p < b);
}
```

◇

Macro referenced in 402b.

Laufzeit. Sei $x \in$ Natural und $a \in$ Digit, dann x<$a \in \mathcal{O}(2)$.

⟨ comparison operator<= of a Natural with a Digit 50c ⟩ ≡

```
inline bool operator<=(const Natural& a, const Digit b)
// Algorithm:  c := a <= b
// Input:      a in Natural, b in Digit.
// Output:     c in bool such that if a <= b then c = true else c = false ||
{
  return (a.size == 1 && *a.p <= b);
}
```

◇

Macro referenced in 402b.

Laufzeit. Sei $x \in$ Natural und $a \in$ Digit, dann x<=$a \in \mathcal{O}(2)$.

⟨ comparison operator> of a Natural with a Digit 50d ⟩ ≡

```
inline bool operator>(const Natural& a, const Digit b)
// Algorithm:  c := a > b
// Input:      a in Natural, b in Digit.
// Output:     c in bool such that if a > b then c = true else c = false ||
{
  return (a.size > 1 || *a.p > b);
}
```

◇

Macro referenced in 402b.

Laufzeit. Sei $x \in$ Natural und $a \in$ Digit, dann x>$a \in \mathcal{O}(2)$.

⟨ comparison operator>= of a Natural with a Digit 51a ⟩ ≡
```
inline bool operator>=(const Natural& a, const Digit b)
// Algorithm:  c := a >= b
// Input:      a in Natural, b in Digit.
// Output:     c in bool such that if a >= b then c = true else c = false ||
{
  return (a.size > 1 || *a.p >= b);
}
```
◇
Macro referenced in 402b.

Laufzeit. Sei $x \in$ Natural und $a \in$ Digit, dann x>=$a \in \mathcal{O}(2)$.

2.8 Additive Operationen

2.8.1 Inkrementierung und Dekrementierung

Der Inkrement-`operator++` beziehungsweise Dekrement-`operator--` ist ein spezieller Zuweisungsoperator. Dieser operatorerhöht den Variableninhalt um eins (Inkrementieren) beziehungsweise vermindert ihn um eins (Dekrementieren). Diese Operatoren können sowohl in Präfix- als auch in Postfix-Schreibweise verwendet weden. Doch dies ist nur dann relevant, wenn wir zum Beispiel noch zusätzliche Zuweisungen durchführen, da `++x` den neuen (erhöhten) Wert von `x` liefert und `x++` den alten (nicht erhöhten) Wert liefert. Demzufolge gilt:

	`y = ++x`	äquivalent zu	`x += 1, y = x`
und	`y = x++`	äquivalent zu	`y = x, x += 1.`

Bei der Implementierung muß man lediglich darauf achtgeben, daß beim Inkrementieren ein Überlauf und beim Dekrementieren ein negatives Ergebnis $\notin$ Natural entstehen kann.
Die Realisierung ist der Funktionalität entsprechend sehr effizient und knapp:

⟨ memberfunction Natural.inc 51b ⟩ ≡
```
void Natural::inc(Digit* b)
// Algorithm:  a.inc(b)
// Input:      a in Natural, b in [a.p, a.p+L(a)].
// Output:     a in Natural such that a := a + 2^(BETA*(a.p+L(a)-b)) ||
{
  NATURALCONDITION(*this);
  CONDITION(b >= p && b <= p+size);

  while (++(*--b) == 0);
  if (b < p) {
    ++size; p = b;
    if (b == root) enlarge(DELTA);
  }

  NATURALCONDITION(*this);
}
```
◇
Macro referenced in 409.

Laufzeit. Sei $x \in$ Natural und $q \in [x.\mathrm{p}, x.\mathrm{p}+\mathcal{L}(x)]$, dann x`.inc(`q`)`$\in \mathcal{O}(1)$.

Beim Dekrementieren ist das Laufzeitverhalten besser, es kann jedoch nun im Gegensatz zum Inkrementieren eine Fehlermeldung entstehen.

⟨ memberfunction Natural.dec 52a ⟩ ≡

```
void Natural::dec(Digit* b)
// Algorithm:  a.dec(b)
// Input:      a in Natural, b in [a.p, a.p+L(a)].
// Output:     a in Natural such that a := a - 2^(BETA*(a.p+L(a)-b)) ||
{
  NATURALCONDITION(*this);
  CONDITION(b >= p && b <= p+size);

  Digit c;
  Digit* pT = p;
  while (b != pT) {
    c = --(*--b);
    if (c != GAMMA) {
      if (c == 0 && b == pT) {
        size_t sT = size;
        if (sT > 2) {       // is possible, e.g. after sub!
          do { ++pT; --sT; } while (*pT == 0 && sT > 1);
          p = pT; size = sT;
        } else if (sT == 2) { p = ++pT; size = --sT; }
      }

      NATURALCONDITION(*this);

      return;
    }
  }
  errmsg(3, "(dec)");
}
```

◇

Macro referenced in 409.

Laufzeit. Sei $x \in$ Natural und $q \in [x.\mathrm{p}, x.\mathrm{p}+\mathcal{L}(x)]$, dann x`.dec(`q`)`$\in \mathcal{O}(1)$.

Präfix-Operatoren

Die Präfix-Operatoren sind nur eine Abbildung auf eine der vorherigen Funktionen:

⟨ prefix incrementation of a Natural 52b ⟩ ≡

```
inline Natural& Natural::operator++()
// Algorithm:  c := ++a
// Input:      a in Natural.
// Output:     a,c in Natural such that a := a+1, c := a ||
{
  inc(p+size);
  return *this;
}
```

◇

Macro referenced in 402b.

Laufzeit. Sei $x \in$ Natural, dann ++x $\sim$ x`.inc(`x`.p+`$\mathcal{L}(x)$`)`.

⟨ prefix decrementation of a Natural 53a ⟩ ≡
```
inline Natural& Natural::operator--()
// Algorithm:  c := --a
// Input:      a in Natural.
// Output:     a,c in Natural such that a := a-1, c := a ||
{
  dec(p+size);
  return *this;
}
```
◇
Macro referenced in 402b.

Laufzeit. Sei $x \in$ Natural, dann `--`$x \sim x$`.dec(`x`.p+`$\mathcal{L}(x)$`)`.

Postfix-Operatoren

Um die Postfix- von den Präfix-Operatoren unterscheiden zu können, wird ein ungenutztes `int`-Argument zur Markierung der Postfix-Version verwendet.
Der Rückgabewert kann nach Definition des Operators nicht mehr über die Referenzierung Natural& übergeben werden, sondern muß in eine neue Instanz kopiert werden. Hinzu kommt noch, daß der Rückgabewert Natural als `const` zurückgegeben werden muß, weil sonst der Ausdruck

```
Natural a; a++++;
```

zulässig ist und in `a.operator++(0).operator++(0)` umgewandelt wird.

⟨ postfix incrementation of a Natural 53b ⟩ ≡
```
inline const Natural Natural::operator++(int)
// Algorithm:  c := a++
// Input:      a in Natural.
// Output:     a,c in Natural such that c := a, a := a+1 ||
{
  const Natural a(*this);
  inc(p+size);
  return a;
}
```
◇
Macro referenced in 402b.

Laufzeit. Sei $x \in$ Natural, dann x`++`$\in \mathcal{O}(\mathcal{L}(x))$.

⟨ postfix decrementation of a Natural 53c ⟩ ≡
```
inline const Natural Natural::operator--(int)
// Algorithm:  c := a--
// Input:      a in Natural.
// Output:     a,c in Natural such that c := a, a := a-1 ||
{
  const Natural a(*this);
  dec(p+size);
  return a;
}
```
◇
Macro referenced in 402b.

Laufzeit. Sei $x \in$ Natural, dann x`--`$\in \mathcal{O}(\mathcal{L}(x))$.

2.8.2 Addition

Die Addition läßt sich jetzt mit der geleisteten Vorarbeit recht einfach realisieren, wobei die tatsächliche Addition auch nur auf der Länge $\min(\mathcal{L}(a), \mathcal{L}(b))$ durchgeführt wird:

⟨ addition of two Naturals 54a ⟩ ≡

```
void Natural::add(const Natural& a, const Natural& b)
// Algorithm:  c.add(a, b)
// Input:      a,b in Natural where not a.p = c.p and not b.p = c.p.
// Output:     c in Natural such that c = a+b ||
{
  NATURALCONDITION(a);
  NATURALCONDITION(b);
  CONDITION(a.p != p && b.p != p);

  const size_t sA = a.size;
  const size_t sB = b.size;
  const Digit* pA = a.p;
  const Digit* pB = b.p;
  if (sA == sB) {
    Digit* pT = setsize(sA);
    add_with_inc(pT, pT+sA, pA+sA, pB+sA);
  } else if (sA > sB) {
    Digit* pT = setsize(sA);
    const Digit* pE = pA+sA-sB;
    COPY(pT, pA, pA, pE);
    if (pB+sB > b.root+sA) add_with_inc(pT, pT+sB, pA+sB, pB+sB);
    else if (add_no_inc(pT, pT+sB, pA+sB, pB+sB)) inc(pT);
  } else {
    Digit* pT = setsize(sB);
    const Digit* pE = pB+sB-sA;
    COPY(pT, pB, pB, pE);
    if (pA+sA > a.root+sB) add_with_inc(pT, pT+sA, pA+sA, pB+sA);
    else if (add_no_inc(pT, pT+sA, pA+sA, pB+sA)) inc(pT);
  }

  NATURALCONDITION(*this);
}
```

◇

Macro referenced in 409.

Laufzeit. Seien $x, y, z \in$ Natural, dann z.add(x,y)$\in \mathcal{O}(\max\{\mathcal{L}(x), \mathcal{L}(y)\})$.

Zuweisungsoperator

Auf den ersten Blick wirkt der Zuweisungsoperator+= etwas aufwendiger, was jedoch nur an der elementaren Programmierung liegt. Hier treten im Gegensatz zur vorherigen Funktion add mehrere Fallunterscheidungen auf, weil wir nur die notwendigen Argumente miteinander addieren wollen:

⟨ assign operator+= for Naturals 54b ⟩ ≡

```
Natural& Natural::operator+=(const Natural& a)
// Algorithm:  c := c += a
// Input:      a,c in Natural.
// Output:     c in Natural such that c := c+a ||
```

```
{
  NATURALCONDITION(*this);
  NATURALCONDITION(a);

  const size_t sT = size;
  const size_t sA = a.size;
  const Digit* pA = a.p;
  Digit* rT = root;
  Digit* pT = p;
  if (sA == sT) {
    if (pT-rT == 1) {
      Digit* pC = NOTHROW_NEW Digit[sT+DELTA];
      root = pC;
      if (!pC) errmsg(2, "(operator+=)");
      FILL_DELTA(pC);
      p = pC += DELTA;
      add_with_inc(pC, pC+sT, pT+sT, pA+sT);
      delete[] rT;
    } else add_with_inc(pT, pT+sT, pA+sT);
  } else if (sA < sT) {
    if (pT-rT == 1) {
      Digit* pC = NOTHROW_NEW Digit[sT+DELTA];
      root = pC;
      if (!pC) errmsg(2, "(operator+=)");
      FILL_DELTA(pC);
      p = pC += DELTA;
      const Digit* pE = pT+sT-sA;
      COPY(pC, pT, pT, pE);
      if (pA+sA > a.root+sT) add_with_inc(pC, pC+sA, pT+sA, pA+sA);
      else if (add_no_inc(pC, pC+sA, pT+sA, pA+sA)) inc(pC);
      delete[] rT;
    } else {
      pT += sT;
      if (add_no_inc(pT-sA, pT, pA+sA)) inc(pT-sA);
    }
  } else {      // sA > sT
    if (pT+sT <= rT+sA+1) {                // (rootsize() <= a.size+1)?
      Digit* pC = NOTHROW_NEW Digit[sA+DELTA];
      root = pC;
      if (!pC) errmsg(2, "(operator+=)");
      FILL_DELTA(pC);
      p = pC += DELTA; size = sA;
      const Digit* pE = pA+sA-sT;
      COPY(pC, pA, pA, pE);
      if (add_no_inc(pC, pC+sT, pT+sT, pA+sT)) inc(pC);
      delete[] rT;
    } else {
      Digit* pE = pT+sT-sA;
      p = pE; size = sA;
      COPY(pE, pA, pE, pT);
      if (add_no_inc(pE, pE+sT, pA+sT)) inc(pE);    // because COPY
    }
  }

  NATURALCONDITION(*this);
```

```
  return *this;
}
```
◇

Macro referenced in 409.

Laufzeit. Seien $x, y \in$ Natural, dann x+=$y \in \mathcal{O}(\max\{\mathcal{L}(x), \mathcal{L}(y)\})$.

Addition mit einem Digit

Die Addition mit einem Digit ist in Schleifen oft sehr nützlich und zeitsparender als die allgemeine Addition:

⟨ addition of a Natural with a Digit 56a ⟩ ≡

```
void Natural::add(const Natural& a, const Digit b)
// Algorithm:  c.add(a, b)
// Input:      a,c in Natural, b in Digit where not a.p = c.p.
// Output:     c in Natural such that c = a+b ||
{
  NATURALCONDITION(a);
  CONDITION(a.p != p);

  const Digit* pA = a.p;
  const size_t sA = a.size;
  if (sA == 1) {
    Digit* pT = setsize(1);
    Digit x = *pA;
    *pT = x += b;
    if (x < b) { *--pT = 1; p = pT; size = 2; }
  } else {
    Digit* pT = setsize(sA);
    const Digit* pE = pT+sA-1;
    COPY(pT, pA, pT, pE);
    Digit x = *pA;
    *pT = x += b;
    if (x < b) {
      while (++(*--pT) == 0);
      const Digit* pC = p;
      if (pT < pC) { p = pT; size = sA+1; }
    }
  }

  NATURALCONDITION(*this);
}
```
◇

Macro referenced in 409.

Laufzeit. Sei $x, y \in$ Natural und $a \in$ Digit, dann y.add(x,a)$\in \mathcal{O}(\mathcal{L}(x))$.

⟨ assign operator+= of a Natural with a Digit 56b ⟩ ≡

```
Natural& Natural::operator+=(const Digit a)
// Algorithm:  c := c += a
// Input:      a in Digit, c in Natural.
// Output:     c in Natural such that c := c+a ||
{
```

```
  NATURALCONDITION(*this);

  Digit* pT = p+size-1;
  Digit b = *pT;
  *pT = b += a;
  if (b < a) inc(pT);

  NATURALCONDITION(*this);

  return *this;
}
◇
```

Macro referenced in 409.

Laufzeit. Sei $x \in$ Natural und $a \in$ Digit, dann x+=$a \sim x$.inc(x.p+$\mathcal{L}(x)-1$).

⟨ additive operator+ of a Natural with a Digit 57 ⟩ ≡

```
inline Natural::Natural(const binder_arguments<Natural, Digit,
                                                Natural_plus_tag>& a)
{
  get_memory(a.x.size+DELTA);
  add(a.x, a.y);
}

inline Natural& Natural::operator=(const binder_arguments<Natural, Digit,
                                                          Natural_plus_tag>& a)
{
  if (this == &a.x) return *this += a.y;
  else { add(a.x, a.y); return *this; }
}

inline binder_arguments<Natural, Digit, Natural_plus_tag>
 operator+(const Natural& a, const Digit& b)
// Algorithm:  c := a + b
// Input:      a in Natural, b in Digit.
// Output:     c in Natural such that c = a+b ||
{
  return binder_arguments<Natural, Digit, Natural_plus_tag>(a, b);
}

inline binder_arguments<Natural, Digit, Natural_plus_tag>
 operator+(const Digit& a, const Natural& b)
// Algorithm:  c := a + b
// Input:      a in Digit, b in Natural.
// Output:     c in Natural such that c = a+b ||
{
  return binder_arguments<Natural, Digit, Natural_plus_tag>(b, a);
}
◇
```

Macro referenced in 402b.

Laufzeit. Seien $x, y \in$ Natural und $a \in$ Digit, dann x+$a \sim y$.add(x,a).

Unterprogramme für die Addition

Die folgenden zwei Funktionen sind für die Programmierung und Übersichtlichkeit des Codes sehr hilfreich, weil man sich dadurch bei der Addition nicht jedesmal um den verbleibenden Übertrag kümmern muß:

⟨ memberfunction Natural.add_with_inc with 3 arguments 58a ⟩ ≡

```
void Natural::add_with_inc(const Digit* pT, Digit* pSum, const Digit* pSmd)
// Algorithm:  a.add_with_inc(r, s, t)
//             Let b in Natural.
// Input:      a in Natural where R(b) > L(a) >= L(b) or R(a) > L(b) >= L(a),
//             r,s in [a.p, a.p+L(a)] where r < s,
//             t in [b.p, b.p+L(b)] where t-(s-r) also in [b.p, b.p+L(b)].
// Output:     a in Natural such that [r, s[ := [r, s[ + [t-(s-r), t[ ||
//
// Note:       R(b) > L(a) >= L(b) or R(a) > L(b) >= L(a)!
{
  CONDITION(pT < pSum && pT >= p && pT <= p+size);
  CONDITION(pSum >= p && pSum <= p+size);

  Digit c,d;
  do {
    c = *--pSmd;                          // non carry addition
    d = *--pSum;
    *pSum = d += c;
    if (c > d)
      do {
        c = *--pSmd;
        d = *--pSum;
        *pSum = d += c+1;                 // addition with carry
      } while (c >= d);
  } while (pSum > pT);
  pT = p;
  if (pSum < pT) { p = pSum; ++size; }
}
◇
```

Macro referenced in 409.

⟨ memberfunction Natural.add_with_inc with 4 arguments 58b ⟩ ≡

```
void Natural::add_with_inc(const Digit* pT, Digit* pSum,
                           const Digit* pSmd1, const Digit* pSmd2)
// Algorithm:  a.add_with_inc(r, s, u, v)
//             Let b,c in Natural.
// Input:      a in Natural where L(a) >= L(b) and L(a) >= L(c)
//             and (R(b) > L(c) >= L(b) or R(c) > L(b) >= L(c)),
//             r,s in [a.p, a.p+L(a)] where r < s,
//             u in [b.p, b.p+L(b)] where u-(s-r) also in [b.p, b.p+L(b)].
//             v in [c.p, c.p+L(c)] where v-(s-r) also in [c.p, c.p+L(c)].
// Output:     a in Natural such that [r, s[ := [u-(s-r), u[ + [v-(s-r), v[ ||
//
// Note:       R(b) > L(c) >= L(b) or R(c) > L(b) >= L(c)!
{
  CONDITION(pT < pSum && pT >= p && pT <= p+size);
  CONDITION(pSum >= p && pSum <= p+size);
```

```
  Digit c,d;
  do {
    c = *--pSmd1;                  // non carry addition
    d = *--pSmd2;
    *--pSum = c += d;
    if (d > c)
      do {
        c = *--pSmd1;
        d = *--pSmd2;
        *--pSum = c += d+1;
      } while (d >= c);
  } while (pSum > pT);
  pT = p;
  if (pSum < pT) { p = pSum; ++size; }
}
◇
```

Macro referenced in 409.

2.8.3 Subtraktion

Die Subtraktion hat eine ähnliche Gestalt wie die Addition, wobei die Differenz nur kleiner oder gleich dem Subtrahenden ist und somit intern kein Überlauf entstehen kann. Es muß allerdings bei der Natural-Arithmetik zusätzlich kontrolliert werden, ob das Ergebnis nicht negativ ist.

⟨ subtraction of two Naturals 59 ⟩ ≡

```
void Natural::sub(const Natural& a, const Natural& b)
// Algorithm:  c.sub(a, b)
// Input:      a,b in Natural.
// Output:     c in Natural such that c = a-b ||
{
  NATURALCONDITION(a);
  NATURALCONDITION(b);

  size_t sA = a.size;
  const size_t sB = b.size;
  const Digit* pA = a.p;
  const Digit* pB = b.p;
  Digit* pT = p;
  if (pA == pT) *this -= b;
  else if (pB == pT) {
    ⟨ special case of subtraction 60a ⟩
  } else if (sA == sB) {
    pT = setsize(sA);
    if (abs(pT, pA, pB, sA) == -1) errmsg(3, "(sub)");
    normalize();
  } else if (sA > sB) {
    pT = setsize(sA);
    const Digit* pE = pA+sA-sB;
    COPY(pT, pA, pA, pE);
    sub(pT, pT+sB, pA+sB, b.p+sB);
    normalize();
  } else errmsg(3, "(sub)");

  NATURALCONDITION(*this);
}
◇
```

Macro referenced in 409.

Die Addition hat den Vorteil, daß die beiden Eingabeargumente kommutieren. Diese Möglichkeit besteht bei der Subtraktion nicht und erfordert daher eine gesonderte Betrachtung des Falles $c = a - c$ für $a, c \in \mathsf{Natural}$.

⟨ special case of subtraction 60a ⟩ ≡

```
if (sB == sA) {
  if (abs(pT, pA, pT, sB) == -1) errmsg(3, "(sub)");
  normalize();
} else if (sB < sA) {
  Digit* rT = root;
  if (pT+sB <= rT+sA) {
    pT = NOTHROW_NEW Digit[sA+DELTA];
    root = pT; size = sA;
    if (!pT) errmsg(2, "(sub)");
    FILL_DELTA(pT);
    p = pT += DELTA;
    const Digit* pE = pA+sA-sB;
    COPY(pT, pA, pA, pE);
    sub(pT, pT+sB, pA+sB, pB+sB);
    normalize();
    delete[] rT;
  } else {
    size = sA;
    sA -= sB;
    p = pT -= sA;
    const Digit* pE = pA+sA;
    COPY(pT, pA, pA, pE);
    sub(pT, pT+sB, pA+sB, pT+sB);
    normalize();
  }
} else errmsg(3, "(sub)");◇
```

Macro referenced in 59.

Laufzeit. Seien $x, y, z \in \mathsf{Natural}$, dann $z\texttt{.sub(}x\texttt{,}y\texttt{)} \in \mathcal{O}(\max\{\mathcal{L}(x), \mathcal{L}(y)\})$.

Der additive `operator-` ist ähnlich wie die Addition eine Abbildung auf die `sub`-Funktion:

⟨ additive `operator-` for Naturals 60b ⟩ ≡

```
inline Natural::Natural(const binder_arguments<Natural, Natural,
                                               Natural_minus_tag>& a)
{
  get_memory(max(a.x.size, a.y.size)+DELTA);
  sub(a.x, a.y);
}

inline Natural& Natural::operator=(const binder_arguments<Natural, Natural,
                                                          Natural_minus_tag>& a)
{
  sub(a.x, a.y);
  return *this;
}
```

```
inline binder_arguments<Natural, Natural, Natural_minus_tag>
 operator-(const Natural& a, const Natural& b)
// Algorithm:  c := a-b
// Input:      a,b in Natural.
// Output:     c in Natural such that c = a-b ||
{
  return binder_arguments<Natural, Natural, Natural_minus_tag>(a, b);
}
◇
```

Macro referenced in 402b.

Laufzeit. Seien $x, y, z \in$ Natural, dann z=x-$y \sim z$.sub(x,y).

Zuweisungsoperator

Der Zuweisungsoperator-= kommt nun mit weniger Operationen aus als die sub-Funktion, wie man in der folgenden Implementation sehen kann:

⟨ assign operator-= for Naturals 61a ⟩ ≡

```
Natural& Natural::operator-=(const Natural& a)
// Algorithm:  c := c -= a
// Input:      a,c in Natural.
// Output:     c in Natural such that c := c-a ||
{
  NATURALCONDITION(*this);
  NATURALCONDITION(a);

  const size_t sT = size;
  const size_t sA = a.size;
  const Digit* pA = a.p;
  Digit* pT = p;
  if (sA <= sT) sub(pT+sT-sA, pT+sT, pA+sA);
  else errmsg(3, "(sub)");
  normalize();

  NATURALCONDITION(*this);

  return *this;
}
◇
```

Macro referenced in 409.

Laufzeit. Seien $x, y \in$ Natural, dann x-=$y \in \mathcal{O}(\min\{\mathcal{L}(x), \mathcal{L}(y)\})$.

Subtraktion mit einem Digit

⟨ subtraction of a Natural with a Digit 61b ⟩ ≡

```
void Natural::sub(const Natural& a, const Digit b)
// Algorithm:  c.sub(a, b)
// Input:      a,c in Natural, b in Digit where not a.p = c.p.
// Output:     c in Natural such that c = a-b ||
{
  NATURALCONDITION(a);
```

```
  CONDITION(a.p != p);

  const Digit* pA = a.p;
  const size_t sA = a.size;
  if (sA == 1) {
    Digit* pT = setsize(1);
    Digit x = *pA;
    if (x < b) errmsg(3, "(sub)");
    *pT = x-b;
  } else {
    Digit* pT = setsize(sA);
    Digit* pE = pT+sA-1;
    COPY(pT, pA, pT, pE);
    Digit x = *pA;
    if (x < b) {
      Digit y;
      do y = --(*--pE); while (y == GAMMA);
      if (y == 0 && pE == p) { p = pE+1; size = sA-1; }
    }
    *pT = x-b;
  }

  NATURALCONDITION(*this);
}
◇
```

Macro referenced in 409.

Laufzeit. Sei $x, y \in$ Natural und $a \in$ Digit, dann y`.sub(`x`,`a`)`$\in \mathcal{O}(\mathcal{L}(x))$.

⟨ assign `operator-=` of a Natural with a Digit 62a ⟩ ≡

```
Natural& Natural::operator-=(const Digit a)
// Algorithm:  c := c -= a
// Input:      a in Digit, c in Natural.
// Output:     c in Natural such that c := c-a ||
{
  NATURALCONDITION(*this);

  Digit* pT = p+size;
  Digit b = *--pT;
  if (b < a) dec(pT);
  *pT = b-a;              // no normalize important!

  NATURALCONDITION(*this);

  return *this;
}
◇
```

Macro referenced in 409.

Laufzeit. Sei $x \in$ Natural und $a \in$ Digit, dann x`-=`$a \sim x$`.dec(`x`.p+`$\mathcal{L}(x) - 1$`)`.

⟨ additive `operator-` of a Natural with a Digit 62b ⟩ ≡

```
inline Natural::Natural(const binder_arguments<Natural, Digit,
                                                Natural_minus_tag>& a)
{
  get_memory(a.x.size+DELTA);
  sub(a.x, a.y);
}

inline Natural& Natural::operator=(const binder_arguments<Natural, Digit,
                                                          Natural_minus_tag>& a)
{
  if (this == &a.x) return *this -= a.y;
  else { sub(a.x, a.y); return *this; }
}

inline binder_arguments<Natural, Digit, Natural_minus_tag>
 operator-(const Natural& a, const Digit& b)
// Algorithm:  c := a - b
// Input:      a in Natural, b in Digit.
// Output:     c in Natural such that c = a-b ||
{
  return binder_arguments<Natural, Digit, Natural_minus_tag>(a, b);
}
◇
```

Macro referenced in 402b.

Laufzeit. Seien $x, y \in$ Natural und $a \in$ Digit, dann x-$a \sim y$.sub(x,a).

Unterprogramme für die Subtraktion

⟨ memberfunction Natural.sub with 3 arguments 63a ⟩ ≡

```
inline void Natural::sub(Digit* pT, Digit* pDif, const Digit* pSub)
// Subtraktion *pDif -= *pSub until pDif = pT,
// pDif-pT <= pSub.rootsize().
{
  if (sub_no_dec(pT, pDif, pSub)) dec(pT);
}
◇
```

Macro referenced in 402b.

⟨ memberfunction Natural.sub_no_dec with 3 arguments 63b ⟩ ≡

```
bool Natural::sub_no_dec(const Digit* pT, Digit* pDif, const Digit* pSub) const
// Algorithm:  d := x.sub_no_inc(r, s, t)
//             Let a,b in Natural.
// Input:      x in Natural,
//             r,s in [a.p a.p+L(a)] where r < s,
//             t in [b.p b.p+L(b)] where t-(s-r) also in [b.p b.p+L(b)].
// Output:     d in bool such that [d] x [r, s[ := [r, s[ - [t-(s-r), t[ ||
{
  CONDITION(pT < pDif);

  Digit c,d;
  do {
    c = *--pSub;
    d = *--pDif;
    if (d >= c) *pDif = d -= c;
```

```
    else {
      *pDif = d -= c;
      do {
        if (pT == pDif) return true;
        c = ~(*--pSub);
        d = *--pDif;
        *pDif = d += c;
      } while (d >= c);
    }
  } while (pT != pDif);
  return false;
}
```
◇

Macro referenced in 409.

⟨ memberfunction Natural.sub with 4 arguments 64a ⟩ ≡

```
void Natural::sub(const Digit* pT, Digit* pDif,
                  const Digit* pMin, const Digit* pSub)
// Algorithm:  a.sub(r, s, u, v)
//             Let b,c in Natural.
// Input:      a in Natural,
//             r,s in [a.root, a.p+L(a)] where r < s,
//             u in [b.root, b.p+L(b)] where u-(s-r) also in [b.root, b.p+L(b)].
//             v in [c.root, c.p+L(c)] where v-(s-r) also in [c.root, c.p+L(c)].
// Output:     [r, s[ := [u-(s-r), u[ - [v-(s-r), v[ ||
{
  CONDITION(pT < pDif && pT >= p && pT <= p+size);
  CONDITION(pDif >= p && pDif <= p+size);

  Digit c,d;
  do {
    c = *--pSub;
    d = *--pMin;
    if (d >= c) *--pDif = d - c;
    else {
      *--pDif = d - c;
      do {
        if (pT == pDif) { dec(pDif); return; }
        c = ~(*--pSub);
        d = *--pMin;
        *--pDif = d += c;
      } while (d >= c);
    }
  } while (pT != pDif);
}
```
◇

Macro referenced in 409.

Die für den Anwender nicht sichtbare Funktion `subpos`, die zusätzlich die Bedingungen stellt, daß der Minuend kleiner oder gleich dem Subtrahend ist und die Speicherallokation vom Minuend größer oder gleich der des Subtrahenden ist, werden wir intern oft benötigen, weil sie ein wenig schneller als `sub` ist.

⟨ internal function subpos with 3 arguments 64b ⟩ ≡

```
inline void subpos(const Digit* pT, Digit* pDif, const Digit* pSub)
// Algorithm:  subpos(r, s, t)
//             Let a,b in Natural.
// Input:      r,s in [a.p, a.p+L(a)] where r < s,
//             t in [b.p, b.p+L(b)] where t-(s-r) also in [b.p, b.p+L(b)],
//             where R(a) > L(b), [r, s[ >= [t-(s-r), t[.
// Output:     [r, s[ := [r, s[ - [t-(s-r), t[ ||
//
// Note:       R(a) > L(b)!
{
  CONDITION(pT < pDif);

  Digit c,d;
  do {
    c = *--pSub;
    d = *--pDif;
    if (d >= c) *pDif = d -= c;
    else {
      *pDif = d -= c;
      do {
        c = ~(*--pSub);
        d = *--pDif;
        *pDif = d += c;
      } while (d >= c);
    }
  } while (pT < pDif);
}
◇
```

Macro referenced in 409.

⟨ internal function subpos with 4 arguments 65 ⟩ ≡

```
inline void subpos(const Digit* pT, Digit* pDif,
                   const Digit* pMin, const Digit* pSub)
// Algorithm:  subpos(r, s, u, v)
//             Let b,c in Natural.
// Input:      a in Natural,
//             r,s in [a.root, a.p+L(a)] where r < s,
//             u in [b.root, b.p+L(b)] where u-(s-r) also in [b.root, b.p+L(b)].
//             v in [c.root, c.p+L(c)] where v-(s-r) also in [c.root, c.p+L(c)].
//             where R(a) > L(b), [u-(s-r), u[ >= [v-(s-r), v[.
// Output:     [r, s[ := [u-(s-r), u[ - [v-(s-r), v[ ||
//
// Note:       R(a) > L(b)!
{
  CONDITION(pT < pDif);

  Digit c,d;
  do {
    c = *--pSub;
    d = *--pMin;
    if (d >= c) *--pDif = d - c;
    else {
      *--pDif = d - c;
      do {
        c = ~(*--pSub);
        d = *--pMin;
```

```
        *--pDif = d += c;
      } while (d >= c);
    }
  } while (pT < pDif);
}
◇
```

Macro referenced in 409.

Durch diese Zusatzinformation sparen wir eine Vergleichsoperation ein, was in einer Assemblerlösung, die die Übertragsinformation über Flags ermittelt, zum Beispiel nicht möglich wäre.

2.8.4 Abstand zweier Naturals

Unter dem Abstand zweier Naturals verstehen wir die folgende Abbildung:

$$\text{abs} : \mathsf{Natural}^2 \to \mathsf{Natural} \ : (a,b) \mapsto \begin{cases} a-b\,, & a > b \\ b-a\,, & a < b \\ 0\,, & a = b \end{cases} .$$

⟨ distance of two Naturals 66 ⟩ ≡

```
Natural abs(const Natural& a, const Natural& b)
// Algorithm:  c := abs(a, b)
// Input:      a,b in Natural.
// Output:     c in Natural such that c = |a-b| ||
{
  NATURALCONDITION(a);
  NATURALCONDITION(b);

  const size_t sA = a.size;
  const size_t sB = b.size;
  Natural c(max(sA, sB)+DELTA, ' ');

  if (sA > sB) c = a-b;
  else if (sA < sB) c = b-a;
  else {
    const Digit* pA = a.p;
    const Digit* pB = b.p;
    Digit* pC = c.p;
    if (pA != pC && pB != pC) pC = c.setsize(sA);
    c.abs(pC, pA, pB, sA);
    c.normalize();
  }

  NATURALCONDITION(c);

  return c;
}
◇
```

Macro defined by 66, 67a.
Macro referenced in 409.

Laufzeit. Seien $x, y \in$ Natural, dann abs$(x,y) \in \mathcal{O}(\max\{\mathcal{L}(x), \mathcal{L}(y)\})$.

⟨ distance of two Naturals 67a ⟩ ≡

```
int abs(const Natural& a, const Natural& b, Natural& c)
// Algorithm:  d := abs(a, b, c)
// Input:      a,b in Natural.
// Output:     d in int, c in Natural such that c = |a-b|, d*c = a-b ||
{
  NATURALCONDITION(a);
  NATURALCONDITION(b);

  const size_t sA = a.size;
  const size_t sB = b.size;

  if (sA > sB) { c = a-b; return 1; }
  else if (sA < sB) { c = b-a; return -1; }
  const Digit* pA = a.p;
  const Digit* pB = b.p;
  Digit* pC = c.p;
  if (pA != pC && pB != pC) pC = c.setsize(sA);
  const int i = c.abs(pC, pA, pB, sA);
  c.normalize();

  NATURALCONDITION(c);

  return i;
}
```

◇

Macro defined by 66, 67a.
Macro referenced in 409.

Laufzeit. Seien $x, y, z \in$ Natural, dann abs$(x, y, z) \in \mathcal{O}(\max\{\mathcal{L}(x), \mathcal{L}(y)\})$.

Intern werden wir die zeigerorientierte Funktion abs oft verwenden.

⟨ memberfunction Natural.abs 67b ⟩ ≡

```
int Natural::abs(Digit* pC, const Digit* pA, const Digit* pB, size_t n) const
// Algorithm:  d := x.abs(r, s, t, n)
//             Let a,b,c in Natural.
// Input:      x in Natural,
//             n in size_t where n > 0,
//             r,r+n in [a.p, a.p+L(a)], s,s+n in [b.p, b.p+L(b)],
//             t,t+n in [c.p, c.p+L(c)].
// Output:     d in int such that d = sign([s, s+n[ - [t, t+n[),
//             [r, r+n[ = |[s, s+n[ - [t, t+n[| ||
{
  CONDITION(n > 0);

  do {
    const Digit a = *pA;
    const Digit b = *pB;
    if (a > b) {
      subpos(pC, pC+n, pA+n, pB+n);
      return 1;
    } else if (a < b) {
      subpos(pC, pC+n, pB+n, pA+n);
      return -1;
```

```
    }
    *pC = 0; ++pA; ++pB; ++pC;
  } while (--n);
  return 0;
}
```
◇

Macro referenced in 409.

2.9 Schiebeoperationen

In diesem Abschnitt stellen wir die einzige Bedigung an unsere Rechnersysteme. Eine Linksverschiebung (`operator<<`) soll der Multiplikation mit zwei und dementsprechend eine Rechtsverschiebung (`operator>>`) der ganzzahligen Division durch zwei entsprechen. Deshalb sind im NumberBase-Konstruktor die folgenden Zeilen eingebaut:

⟨ testing the bits for low ending 68a ⟩ ≡
```
const Digit d = 1 << 1;
if (d != 2) errmsg(0, "Bitproblem!");
```
◇

Macro referenced in 398b.

Daher wird beim Compilieren eine Warnung ausgegeben, daß der Ausdruck `(d != 2)` nicht erfüllbar ist.

2.9.1 Linksverschiebung

Man muß bei der Linksverschiebung darauf achtgeben, daß die interne Digit-Verschiebung auf vielen Rechnersystemen modulo β durchgeführt wird und dadurch das Verschieben um 0 (`b` $= 0$) abgefangen werden muß. Desweiteren ist es möglich, daß ein interner Überlauf bei der Linksverschiebung auftritt.

⟨ multiplication of a Natural by a power of 2 68b ⟩ ≡
```
void Natural::lshift(const Natural& a, size_t b)
// Algorithm:  c.lshift(a, b)
// Input:      a in Natural, b in size_t where not a.p = c.p.
// Output:     c in Natural such that c = a*2^b ||
{
  NATURALCONDITION(a);
  CONDITION(a.p != p);

  const Digit* pA = a.p;
  Digit d = *pA;
  if (d == 0) *this = 0;
  else {
    const size_t sA = a.size;
    const size_t b2 = b/BETA;
    size_t sT = sA+b2+1;
    Digit* pT = setsize(sT);
    if (b2) {
      const Digit* pE = pT+sT;
      pT += sA+1;
      FILL_ZERO(pT, pE);
      pT -= sT;
```

```
    }
    b %= BETA;
    if (b) {
      const Digit b2 = BETA-b;
      const Digit* pE = pA+sA;
      Digit e = d >> b2;
      *pT++ = e;
      if (e == 0) { size = --sT; p = pT; }
      while (++pA != pE) {
        e = *pA;
        *pT++ = (d << b) | (e >> b2);
        if (++pA == pE) { d = e; break; }
        d = *pA;
        *pT++ = (e << b) | (d >> b2);
      }
      *pT = d << b;
    } else {
      *pT = 0;
      size = --sT; p = ++pT;
      const Digit* pE = pA+sA;
      COPY(pT, pA, pA, pE);
    }
  }

  NATURALCONDITION(*this);
}
◇
```

Macro referenced in 409.

Laufzeit. Seien $x, y \in$ Natural und $t \in$ `size_t`, dann y`.lshift(`x`,`t`)` $\in \mathcal{O}\left(\mathcal{L}(x) + \left\lfloor \frac{t}{\beta} \right\rfloor\right)$.
Eine Linksverschiebung um mehrere Digits ist eine Linksverschiebung um den Faktor β.

⟨ multiplication of a Natural by a power of 2^β 69 ⟩ ≡

```
void Natural::lmove(size_t b)
// Algorithm:  a.lmove(b)
// Input:      a in Natural, b in size_t.
// Output:     a in Natural such that a := a*2^(BETA*b) ||
{
  NATURALCONDITION(*this);

  Digit* pT = p;
  size_t sT = size;
  Digit c = *pT;
  if (c) {
    ++b;
    Digit* rT = root;
    size_t sC = sT+b;
    Digit* pC = pT-b;
    if (rT+b >= pT) {
      root = pC = NOTHROW_NEW Digit[sC+DELTA];
      if (!pC) errmsg(2, "(lmove)");
      FILL_DELTA(pC);
      pC += DELTA;
    }
    *pC = 0;
```

```
    size = --sC; p = ++pC;
    const Digit* pE = pT+sT;
    COPY(pC, pT, pT, pE);
    if (--b) {
      const Digit* pE = pC+b;
      FILL_ZERO(pC, pE);
    }
    if (root != rT) delete[] rT;
  }

  NATURALCONDITION(*this);
}
◇
```

Macro referenced in 409.

Laufzeit. Seien $x \in$ Natural und $t \in$ `size_t`, dann x`.lmove(`t`)`$\in \mathcal{O}\left(\mathcal{L}(x) + t\right)$.

Verschiebungsoperator

⟨ shift `operator<<` of a Natural 70a ⟩ ≡

```
inline Natural::Natural(const binder_arguments<Natural, size_t,
                                               Natural_lshift_tag>& a)
{
  get_memory(a.x.size+a.y%BETA+DELTA);
  lshift(a.x, a.y);
}

inline Natural& Natural::operator=(const binder_arguments<Natural, size_t,
                                                          Natural_lshift_tag>& a)
{
  if (this == &a.x) return *this <<= a.y;
  else { lshift(a.x, a.y); return *this; }
}

inline binder_arguments<Natural, size_t, Natural_lshift_tag>
 operator<<(const Natural& a, const size_t& b)
// Algorithm:  c := a << b
// Input:      a in Natural, b in size_t.
// Output:     c in Natural such that c = a*2^b ||
{
  return binder_arguments<Natural, size_t, Natural_lshift_tag>(a, b);
}
◇
```

Macro referenced in 402b.

Laufzeit. Seien $x, y \in$ Natural und $t \in$ `size_t`, dann x`<<`$t \sim y$`.lshift(`x`,`t`)`.

Zuweisungsoperator

Wir werden auch den `operator<<=` auf der unteren Ebene verwirklichen und nicht auf die Funktion `lshift` abbilden, weil wir so zusätzlich Laufzeit einsparen können. Dadurch wird jedoch der Code unübersichtlicher.

⟨ assign `operator<<=` of a Natural 70b ⟩ ≡

```
Natural& Natural::operator<<=(size_t a)
// Algorithm:  c := c <<= a
// Input:      a in size_t, c in Natural.
// Output:     c in Natural such that c := c*2^a ||
{
  NATURALCONDITION(*this);

  if (a >> (CHAR_BIT*sizeof(size_t)-1)) errmsg(2, "(operator<<=)");
  Digit* pT = p;
  size_t sT = size;
  Digit c = *pT;
  if (c) {
    size_t b = a/BETA+1;
    a %= BETA;
    Digit* rT = root;
    size_t sC = sT+b;
    Digit* pC = pT-b;
    if (rT+b >= pT) {
      root = pC = NOTHROW_NEW Digit[sC+DELTA];
      if (!pC) errmsg(2, "(operator<<=)");
      FILL_DELTA(pC);
      pC += DELTA;
    }
    p = pC; size = sC;
    if (a) {
      const Digit a2 = BETA-a;
      const Digit* pE = pT+sT;
      Digit d = c >> a2;
      *pC++ = d;
      if (d == 0) { size = --sC; p = pC; }
      while (++pT != pE) {
        d = *pT;
        *pC++ = (c << a) | (d >> a2);
        if (++pT == pE) { c = d; break; }
        c = *pT;
        *pC++ = (d << a) | (c >> a2);
      }
      *pC++ = c << a;
    } else {
      *pC = 0;
      size = --sC; p = ++pC;
      const Digit* pE = pT+sT;
      COPY(pC, pT, pT, pE);
    }
    if (--b) {
      const Digit* pE = pC+b;
      FILL_ZERO(pC, pE);
    }
    if (root != rT) delete[] rT;
  }

  NATURALCONDITION(*this);

  return *this;
}
◇
```

Macro referenced in 409.

Laufzeit. Sei $x \in$ Natural und $t \in$ size_t, dann x<<=$t \in \mathcal{O}\left(\mathcal{L}(x) + \left\lfloor \frac{t}{\beta} \right\rfloor\right)$.

2.9.2 Rechtsverschiebung

Die Rechtsverschiebung läßt sich im Gegensatz zur Linksverschiebung einfacher implementieren, weil hier kein Überlauf entstehen kann. Ansonsten ist die Implementierung weitgehend äquivalent zur Linksverschiebung.

⟨ division of a Natural by a power of 2 72a ⟩ ≡

```
void Natural::rshift(const Natural& a, size_t b)
// Algorithm:  c.rshift(a, b)
// Input:      a in Natural, b in size_t where not a.p = c.p.
// Output:     c in Natural such that c = [a/2^b] ||
{
  NATURALCONDITION(a);
  CONDITION(a.p != p);

  const Digit* pA = a.p;
  const size_t sA = a.size;
  const size_t b2 = b/BETA;
  if (b2 >= sA) *this = 0;
  else {
    b %= BETA;
    size_t sT = sA-b2;
    Digit* pT = setsize(sT);
    if (b) {
      pT += sT;
      const size_t b2 = BETA-b;
      const Digit* pE = pA+sT;
      Digit e,d = *--pE;
      while (pE != pA) {
        e = *--pE;
        *--pT = (e << b2) | (d >> b);
        if (pE == pA) { d = e; break; }
        d = *--pE;
        *--pT = (d << b2) | (e >> b);
      }
      *--pT = d >>= b;
      if (d == 0 && sT > 1) { p = ++pT; size = --sT; }
    } else {
      const Digit* pE = pA+sT;
      COPY(pT, pA, pA, pE);
    }
  }

  NATURALCONDITION(*this);
}
```

◇

Macro referenced in 409.

Laufzeit. Seien $x, y \in$ Natural und $t \in$ size_t, dann y.rshift(x,t)$\in \mathcal{O}\left(\mathcal{L}(x) - \left\lfloor \frac{t}{\beta} \right\rfloor\right)$.

Eine Rechtsverschiebung um mehrere Digits ist eine Rechtsverschiebung um den Faktor β.

⟨ division of a Natural by a power of 2^β 72b ⟩ ≡

```
void Natural::rmove(size_t a)
// Algorithm:  b.rmove(a)
// Input:      a in size_t, b in Natural.
// Output:     b in Natural such that b := [b/2^(BETA*a)] ||
{
  NATURALCONDITION(*this);

  size_t sT = size;
  if (a >= sT) *this = 0;
  else {
    size_t sA = sT-a;
    Digit* pE = p;
    Digit* pT = pE+sT;
    if (a) {
      Digit* pA = pT-a;
      COPY_BACKWARD(pT, pA, pA, pE);
      p = pT; size = sA;
      FILL_ZERO(pA, pT);
    }
  }

  NATURALCONDITION(*this);
}
◇
```

Macro referenced in 409.

Laufzeit. Seien $x \in$ Natural und $t \in$ `size_t`, dann x`.rmove(`t`)`$\in \mathcal{O}\left(\mathcal{L}(x) - t\right)$.

Verschiebungsoperator

⟨ shift `operator>>` of a Natural 73 ⟩ ≡

```
inline Natural::Natural(const binder_arguments<Natural, size_t,
                                               Natural_rshift_tag>& a)
{
  get_memory(a.x.size+DELTA);
  rshift(a.x, a.y);
}

inline Natural& Natural::operator=(const binder_arguments<Natural, size_t,
                                                          Natural_rshift_tag>& a)
{
  if (this == &a.x) return *this >>= a.y;
  else { rshift(a.x, a.y); return *this; }
}

inline binder_arguments<Natural, size_t, Natural_rshift_tag>
 operator>>(const Natural& a, const size_t& b)
// Algorithm:  c := a >> b
// Input:      a in Natural, b in size_t.
// Output:     c in Natural such that c = [a/2^b] ||
{
  return binder_arguments<Natural, size_t, Natural_rshift_tag>(a, b);
}
◇
```

Macro referenced in 402b.

Laufzeit. Seien $x, y \in$ Natural und $t \in$ size_t, dann x>>$t \sim y$.rshift(x,t).

Zuweisungsoperator

Im Vergleich zur Linksverschiebung bereitet uns der Zuweisungsoperator>>= nicht so große Kopfschmerzen:

⟨ assign operator>>= of a Natural 74 ⟩ ≡

```
Natural& Natural::operator>>=(size_t a)
// Algorithm:  c := c >>= a
// Input:      a in size_t, c in Natural.
// Output:     c in Natural such that c := [c/2^a] ||
{
  NATURALCONDITION(*this);

  const size_t a2 = a/BETA;
  size_t sT = size;
  if (a2 >= sT) *this = 0;
  else {
    a %= BETA;
    size_t sA = sT-a2;
    Digit* pE = p;
    Digit* pT = pE+sT;
    if (a) {
      const Digit* pA = pE+sA;
      const size_t a2 = BETA-a;
      Digit e,d = *--pA;
      while (pA != pE) {
        e = *--pA;
        *--pT = (e << a2) | (d >> a);
        if (pA == pE) { d = e; break; }
        d = *--pA;
        *--pT = (d << a2) | (e >> a);
      }
      *--pT = d >>= a;
      if (pE != pT) FILL_ZERO(pE, pT);
      if (d == 0 && sA > 1) { ++pT; --sA; }
      p = pT; size = sA;
    } else if (a2) {
      Digit* pA = pT-a2;
      COPY_BACKWARD(pT, pA, pA, pE);
      p = pT; size = sA;
      FILL_ZERO(pA, pT);
    }
  }

  NATURALCONDITION(*this);

  return *this;
}
◇
```

Macro referenced in 409.

Laufzeit. Sei $x \in$ Natural und $t \in$ size_t, dann x>>=$t \in \mathcal{O}\left(\mathcal{L}(x) - \left\lfloor \frac{t}{\beta} \right\rfloor\right)$.

2.9.3 Schnelle Verschiebung

Zusätzlich zu den regulären Verschiebeoperationen besteht intern die Möglichkeit, eine schnelle Verschiebung durchzuführen, die die interne Darstellung ausnutzt.

⟨ fast division of a Natural by a power of 2^β 75a ⟩ ≡

```
inline void Natural::fast_rshift(const size_t a)
// Algorithm:  b.fast_rshift(a)
// Input:      b in Natural, a in size_t where L(b) > a.
// Output:     b in Natural such that b := [b/2^(BETA*a)] ||
{
  size -= a;
}
◇
```

Macro referenced in 402b.

Laufzeit. Sei $x \in$ Natural und $t \in$ `size_t`, dann x`.fast_rshift(`t`)`$\in \mathcal{O}(1)$.

⟨ fast multiplication of a Natural by a power of 2^β 75b ⟩ ≡

```
inline void Natural::fast_append(const size_t a)
// Algorithm:  b.fast_append(a)
// Input:      b in Natural, a in size_t.
// Output:     b in Natural such that L(b) := L(b)+a ||
{
  size += a;
}
◇
```

Macro referenced in 402b.

Laufzeit. Sei $x \in$ Natural und $t \in$ `size_t`, dann x`.fast_append(`t`)`$\in \mathcal{O}(1)$.

Diese beiden geschützten Funktionen sind jeweils für sich allein mit äußerster Vorsicht zu genießen und daher ausschließlich in Kombination zu benutzen. Die Funktion `fast_rshift` ist zwar nicht ganz so gefährlich, da sie nur bis zum Aufruf des Destruktors oder bei einem internen Überlaufes den alloziierten Speicher nicht vollständig ausnutzt. Gefährlicher ist aber die Funktion `fast_append`, denn sie setzt voraus, daß nicht genutzter Speicher existiert. Deswegen darf man diese beiden Funktionen auch nur dann auf ein Natural$a$ in Kombination verwenden, wenn man genau weiß, daß in den durchgeführten Operationen zwischen diesen beiden Funktionsaufrufen kein interner Überlauf für das Natural$a$ entstehen kann.

2.10 Bitweise Verknüpfungen

In diesem Abschnitt betrachten wir Operationen, die trivial gelöst sind und bei der Implementierung eher Routinearbeit als Herausforderungen darstellen. Diese Funktionen sind aber der Vollständigkeit wegen notwendig.

2.10.1 Bitweise UND-Verknüpfung

⟨ bitwise and of two Naturals 75c ⟩ ≡

```
void Natural::bitwise_and(const Natural& a, const Natural& b)
// Algorithm:  c.bitwise_and(a, b)
// Input:      a,b in Natural where not a.p = c.p and not b.p = c.p.
```

```
// Output:     c in Natural such that c = a and b ||
{
  NATURALCONDITION(a);
  NATURALCONDITION(b);
  CONDITION(a.p != p && b.p != p);

  const Digit* pA = a.p;
  const size_t sA = a.size;
  const Digit* pB = b.p;
  const size_t sB = b.size;
  size_t sT;
  if (sA >= sB) { pA += sA-sB; sT = sB; }
  else { pB += sB-sA; sT = sA; }
  Digit d;
  do d = *pA++ & *pB++; while (d == 0 && --sT);
  if (sT) {
    Digit* pT = setsize(sT);
    *pT = d;
    const Digit* pE = pT+sT;
    while (++pT != pE) *pT = *pA++ & *pB++;
  } else *this = 0;

  NATURALCONDITION(*this);
}
◇
```

Macro referenced in 409.

Laufzeit. Seien $x, y, z \in$ Natural, dann z`.bitwise_and(`x`,`y`)`$\in \mathcal{O}(\min\{\mathcal{L}(x), \mathcal{L}(y)\})$.

Operatoren

⟨ bitwise `operator&` for Naturals 76 ⟩ ≡

```
inline Natural::Natural(const binder_arguments<Natural, Natural,
                                                Natural_and_tag>& a)
{
  get_memory(min(a.x.size, a.y.size)+DELTA);
  bitwise_and(a.x, a.y);
}

inline Natural& Natural::operator=(const binder_arguments<Natural, Natural,
                                                          Natural_and_tag>& a)
{
  if (this == &a.x) return *this &= a.y;
  else if (this == &a.y) return *this &= a.x;
  else { bitwise_and(a.x, a.y); return *this; }
}

inline binder_arguments<Natural, Natural, Natural_and_tag>
 operator&(const Natural& a, const Natural& b)
// Algorithm:  c := a & b
// Input:      a,b in Natural.
// Output:     c in Natural such that c = a and b ||
{
  return binder_arguments<Natural, Natural, Natural_and_tag>(a, b);
}
◇
```

Macro referenced in 402b.

Laufzeit. Seien $x, y, z \in$ Natural, dann $x\&y \sim z.\texttt{bitwise\_and}(x,y)$.

⟨ bitwise operator& of a Natural with a Digit 77a ⟩ ≡

```
inline Digit operator&(const Natural& a, const Digit b)
// Algorithm:  c := a & b
// Input:      a in Natural, b in Digit.
// Output:     c in Digit such that c = a and b ||
{
  return a.lowest() & b;
}
```

◇

Macro referenced in 402b.

Laufzeit. Sei $x \in$ Natural und $a \in$ Digit, dann $x\&a \in \mathcal{O}(1)$.

Zuweisungsoperatoren

⟨ assign operator&= for Naturals 77b ⟩ ≡

```
Natural& Natural::operator&=(const Natural& a)
// Algorithm:  c := c &= a
// Input:      a,c in Natural.
// Output:     c in Natural such that c := c and a ||
{
  NATURALCONDITION(*this);

  const Digit* pA = a.p;
  const size_t sA = a.size;
  const Digit* pE = pA+sA;
  size_t sT = size;
  Digit* pT = p;
  if (sT > sA) {
    const Digit* pE = pT+sT-sA;
    FILL_ZERO(pT, pE);
    p = pT; size = sA;
  } else pA += sA-sT;
  do *pT++ &= *pA++; while (pA != pE);
  normalize();

  NATURALCONDITION(*this);

  return *this;
}
```

◇

Macro referenced in 409.

Laufzeit. Seien $x, y \in$ Natural, dann $x\&{=}y \in \mathcal{O}(\mathcal{L}(x))$.

⟨ assign operator&= of a Natural with a Digit 78a ⟩ ≡

```
inline Digit Natural::operator&=(const Digit b)
// Algorithm:  c := a &= b
// Input:      a in Natural, b in Digit.
// Output:     a in Natural, c in Digit such that a := a and b, c = a ||
{
  return *this = lowest() & b;
}
```

◇

Macro referenced in 402b.

Laufzeit. Sei $x \in$ Natural und $a \in$ Digit, dann x&=$a \in \mathcal{O}(1)$.

2.10.2 Bitweise inklusive ODER-Verknüpfung

⟨ bitwise inclusive or of two Naturals 78b ⟩ ≡

```
void Natural::bitwise_or(const Natural& a, const Natural& b)
// Algorithm:  c.bitwise_or(a, b)
// Input:      a,b in Natural where not a.p = c.p and not b.p = c.p.
// Output:     c in Natural such that c = a or b ||
{
  NATURALCONDITION(a);
  NATURALCONDITION(b);
  CONDITION(a.p != p && b.p != p);

  const Digit* pA = a.p;
  const size_t sA = a.size;
  const Digit* pB = b.p;
  const size_t sB = b.size;
  if (sA > sB) {
    Digit* pT = setsize(sA);
    const Digit* pE = pA+sA-sB;
    COPY(pT, pA, pA, pE);
    pE += sB;
    do *pT++ = *pA++ | *pB++; while (pA != pE);
  } else if (sA < sB) {
    Digit* pT = setsize(sB);
    const Digit* pE = pB+sB-sA;
    COPY(pT, pB, pB, pE);
    pE += sA;
    do *pT++ = *pA++ | *pB++; while (pB != pE);
  } else {
    Digit* pT = setsize(sA);
    const Digit* pE = pA+sA;
    do *pT++ = *pA++ | *pB++; while (pA != pE);
  }

  NATURALCONDITION(*this);
}
```

◇

Macro referenced in 409.

Laufzeit. Seien $x, y, z \in$ Natural, dann z`.bitwise_or(`x`,`y`)`$\in \mathcal{O}(\max\{\mathcal{L}(x), \mathcal{L}(y)\})$.

Operatoren

⟨ bitwise operator| for Naturals 79a ⟩ ≡

```
inline Natural::Natural(const binder_arguments<Natural, Natural,
                                                Natural_or_tag>& a)
{
  get_memory(max(a.x.size, a.y.size)+DELTA);
  bitwise_or(a.x, a.y);
}

inline Natural& Natural::operator=(const binder_arguments<Natural, Natural,
                                                          Natural_or_tag>& a)
{
  if (this == &a.x) return *this |= a.y;
  else if (this == &a.y) return *this |= a.x;
  else { bitwise_or(a.x, a.y); return *this; }
}

inline binder_arguments<Natural, Natural, Natural_or_tag>
 operator|(const Natural& a, const Natural& b)
// Algorithm:  c := a | b
// Input:      a,b in Natural.
// Output:     c in Natural such that c = a or b ||
{
  return binder_arguments<Natural, Natural, Natural_or_tag>(a, b);
}
◇
```

Macro referenced in 402b.

Laufzeit. Seien $x, y, z \in$ Natural, dann $x|y \sim z$.bitwise_or(x,y).

⟨ bitwise operator| of a Natural with a Digit 79b ⟩ ≡

```
inline Natural operator|(const Natural& a, const Digit b)
// Algorithm:  c := a | b
// Input:      a in Natural, b in Digit.
// Output:     c in Natural such that c = a or b ||
{
  return Natural(a) |= b;
}
◇
```

Macro referenced in 402b.

Laufzeit. Sei $x \in$ Natural und $a \in$ Digit, dann $x|a \sim x|{=}a$.

Zuweisungsoperatoren

⟨ assign operator|= for Naturals 79c ⟩ ≡

```
Natural& Natural::operator|=(const Natural& a)
// Algorithm:  c := c |= a
// Input:      a,c in Natural.
// Output:     c in Natural such that c := c or a ||
{
  NATURALCONDITION(*this);
```

```
  const Digit* pA = a.p;
  size_t sA = a.size;
  Digit* rT = root;
  size_t sT = size;
  Digit* pT = p;
  if (sA > sT)
    if (rT+sA >= pT+sT) {           // a.size >= rootsize()
      Digit* pC = NOTHROW_NEW Digit[sA+DELTA];
      root = pC; size = sA;
      if (!pC) errmsg(2, "(operator|=)");
      FILL_DELTA(pC);
      p = pC += DELTA;
      const Digit* pE = pA+sA-sT;
      COPY(pC, pA, pA, pE);
      pE += sT;
      do *pC++ = *pT++ | *pA++; while (pA != pE);
      delete[] rT;
    } else {
      size = sA; sA -= sT; p = pT -= sA;
      const Digit* pE = pA+sA;
      COPY(pT, pA, pA, pE);
      pE += sT;
      do *pT++ |= *pA++; while (pA != pE);
    }
  else {
    const Digit* pE = pA+sA;
    pT += sT-sA;
    do *pT++ |= *pA++; while (pA != pE);
  }

  NATURALCONDITION(*this);

  return *this;
}
◇
```

Macro referenced in 409.

Laufzeit. Seien $x, y \in$ Natural, dann x |=$y \in \mathcal{O}(\mathcal{L}(y))$.

⟨ assign operator|= of a Natural with a Digit 80a ⟩ ≡

```
inline Natural& Natural::operator|=(const Digit a)
// Algorithm:  c := c |= a
// Input:      c in Natural, a in Digit.
// Output:     c in Natural such that c := c or a ||
{
  p[size-1] |= a;
  return *this;
}
◇
```

Macro referenced in 402b.

Laufzeit. Seien $x, y \in$ Natural, dann x |=$y \in \mathcal{O}(\mathcal{L}(x))$.

2.10.3 Bitwise exklusive ODER-Verknüpfung

⟨ bitwise exclusive or of two Naturals 80b ⟩ ≡

```
void Natural::bitwise_xor(const Natural& a, const Natural& b)
// Algorithm:  c.bitwise_xor(a, b)
// Input:      a,b in Natural where not a.p = c.p and not b.p = c.p.
// Output:     c in Natural such that c = a xor b ||
{
  NATURALCONDITION(a);
  NATURALCONDITION(b);
  CONDITION(a.p != p && b.p != p);

  const Digit* pA = a.p;
  const size_t sA = a.size;
  const Digit* pB = b.p;
  const size_t sB = b.size;
  if (sA > sB) {
    Digit* pT = setsize(sA);
    const Digit* pE = pA+sA-sB;
    COPY(pT, pA, pA, pE);
    pE += sB;
    do *pT++ = *pA++ ^ *pB++; while (pA != pE);
  } else if (sA < sB) {
    Digit* pT = setsize(sB);
    const Digit* pE = pB+sB-sA;
    COPY(pT, pB, pB, pE);
    pE += sA;
    do *pT++ = *pA++ ^ *pB++; while (pB != pE);
  } else {
    Digit* pT = setsize(sA);
    const Digit* pE = pA+sA;
    do *pT++ = *pA++ ^ *pB++; while (pA != pE);
    normalize();
  }

  NATURALCONDITION(*this);
}
◇
```

Macro referenced in 409.

Laufzeit. Seien $x, y, z \in$ Natural, dann z.bitwise_xor(x,y)$\in \mathcal{O}(\max\{\mathcal{L}(x), \mathcal{L}(y)\})$.

Operator

⟨ bitwise operator^ for Naturals 81 ⟩ ≡

```
inline Natural::Natural(const binder_arguments<Natural, Natural,
                                               Natural_xor_tag>& a)
{
  get_memory(max(a.x.size, a.y.size)+DELTA);
  bitwise_xor(a.x, a.y);
}

inline Natural& Natural::operator=(const binder_arguments<Natural, Natural,
                                                          Natural_xor_tag>& a)
{
  if (this == &a.x) return *this ^= a.y;
  else if (this == &a.y) return *this ^= a.x;
  else { bitwise_xor(a.x, a.y); return *this; }
```

```
}

inline binder_arguments<Natural, Natural, Natural_xor_tag>
 operator^(const Natural& a, const Natural& b)
// Algorithm:  c := a ^ b
// Input:      a,b in Natural.
// Output:     c in Natural such that c = a xor b ||
{
  return binder_arguments<Natural, Natural, Natural_xor_tag>(a, b);
}
◇
```

Macro referenced in 402b.

Laufzeit. Seien $x, y, z \in$ Natural, dann $x\hat{}y \sim z$.bitwise_xor(x,y).

Zuweisungsoperator

⟨ assign operator^= for Naturals 82 ⟩ ≡

```
Natural& Natural::operator^=(const Natural& a)
// Algorithm:  c := c ^= a
// Input:      a,c in Natural.
// Output:     c in Natural such that c := c xor a ||
{
  NATURALCONDITION(*this);

  const Digit* pA = a.p;
  size_t sA = a.size;
  Digit* rT = root;
  size_t sT = size;
  Digit* pT = p;
  if (sA > sT)
    if (rT+sA >= pT+sT) {       // a.size >= rootsize()
      Digit* pC = NOTHROW_NEW Digit[sA+DELTA];
      root = pC; size = sA;
      if (!pC) errmsg(2, "(operator^=)");
      FILL_DELTA(pC);
      p = pC += DELTA;
      const Digit* pE = pA+sA-sT;
      COPY(pC, pA, pA, pE);
      pE += sT;
      do *pC++ = *pT++ ^ *pA++; while (pA != pE);
      delete[] rT;
    } else {
      size = sA; sA -= sT; p = pT -= sA;
      const Digit* pE = pA+sA;
      COPY(pT, pA, pA, pE);
      pE += sT;
      do *pT++ ^= *pA++; while (pA != pE);
    }
  else if (sA < sT) {
    const Digit* pE = pA+sA;
    pT += sT-sA;
    do *pT++ ^= *pA++; while (pA != pE);
  } else {
    const Digit* pE = pA+sA;
```

```
    do *pT++ ^= *pA++; while (pA != pE);
    normalize();
  }

  NATURALCONDITION(*this);

  return *this;
}
◇
```

Macro referenced in 409.

Laufzeit. Seien $x, y \in$ Natural, dann x`^=`$y \in \mathcal{O}(\mathcal{L}(y))$.

⟨ assign `operator^=` of a Natural with a Digit 83a ⟩ ≡

```
inline Natural& Natural::operator^=(const Digit a)
// Algorithm:  c := c ^= a
// Input:      c in Natural, a in Digit.
// Output:     c in Natural such that c := c xor a ||
{
  p[size-1] ^= a;
  return *this;
}
◇
```

Macro referenced in 402b.

Laufzeit. Sei $x \in$ Natural und $a \in$ Digit, dann x`^=`$a \in \mathcal{O}(1)$.

2.10.4 Bitweise Negation

⟨ bitwise not of a Natural 83b ⟩ ≡

```
void Natural::bitwise_not(const Natural& a)
// Algorithm:  b.bitwise_not(a)
// Input:      a in Natural.
// Output:     b in Natural such that b = not a ||
{
  NATURALCONDITION(a);

  const size_t sA = a.size;
  const Digit* pA = a.p;
  const Digit* pE = pA+sA;
  Digit* pT = setsize(sA);
  do *pT++ = ~*pA++; while (pA != pE);
  normalize();

  NATURALCONDITION(*this);
}
◇
```

Macro referenced in 409.

Laufzeit. Seien $x, y \in$ Natural, dann y`.bitwise_not(`x`)` $\in \mathcal{O}(\mathcal{L}(x))$.

Bitweiser Negationsoperator

⟨ bitwise operator~ of a Natural 84a ⟩ ≡

```
inline Natural::Natural(const binder_arguments<Natural, Natural,
                                                Natural_not_tag>& a)
{
  get_memory(a.x.size+DELTA);
  bitwise_not(a.x);
}

inline Natural& Natural::operator=(const binder_arguments<Natural, Natural,
                                                          Natural_not_tag>& a)
{
  bitwise_not(a.x);
  return *this;
}

inline binder_arguments<Natural, Natural, Natural_not_tag>
 operator~(const Natural& a)
// Algorithm:  c := ~a
// Input:      a in Natural.
// Output:     c in Natural such that c = not a ||
{
  return binder_arguments<Natural, Natural, Natural_not_tag>(a, a);
}
◇
```

Macro referenced in 402b.

Laufzeit. Seien $x, y \in$ Natural, dann $\sim x \sim y$.not(x).

2.10.5 Bitoperationen

Um ein einzelnes Bit zu setzen, zu löschen oder zu überprüfen, gibt es die folgenden Funktionen:

⟨ sets a bit in a Natural 84b ⟩ ≡

```
void Natural::setbit(const Digit a)
// Algorithm:  c.setbit(a)
// Input:      a in Digit, c in Natural.
// Output:     c in Natural such that c := c or 2^a ||
{
  NATURALCONDITION(*this);

  const size_t  b = size_t(a/BETA) + 1;
  const Digit   c = Digit(1) << (a%BETA);
  const size_t sT = size;
  const Digit* rT = root;
  Digit* pT = p;
  if (b <= sT) pT[sT-b] |= c;
  else if (rT+b < pT+sT) {                  // b < rootsize()
    p = pT -= b-sT; size = b; *pT |= c;
  } else {
    enlarge(DELTA+b-sT);
    pT = p;
    p = pT -= b-sT; size = b; *pT |= c;
```

```
  }

  NATURALCONDITION(*this);
}
```
◇

Macro referenced in 409.

Laufzeit. Sei $x \in$ Natural und $a \in$ Digit, dann x`.setbit(`a`)`$\in \begin{cases} \mathcal{O}(\lfloor \frac{t}{\beta} \rfloor), & \mathcal{R}(x) \leq \lfloor \frac{t}{\beta} \rfloor \\ \mathcal{O}(1), & \mathcal{R}(x) > \lfloor \frac{t}{\beta} \rfloor \end{cases}$.

⟨ clears a bit in a Natural 85a ⟩ ≡

```
void Natural::clearbit(const Digit a)
// Algorithm:  c.clearbit(a)
// Input:      a in Digit, c in Natural.
// Output:     c in Natural such that c := c and not(2^a) ||
{
  NATURALCONDITION(*this);

  const size_t b = size_t(a/BETA) + 1;
  const Digit c = Digit(1) << (a%BETA);
  size_t sT = size;
  Digit* pT = p;
  if (b == sT) {
    Digit d = *pT;
    *pT = d &= ~c;
    if (d == 0) {                       // normalize
      if (sT > 2) {
        do { ++pT; --sT; } while (*pT == 0 && sT > 1);
        p = pT; size = sT;
      } else if (sT == 2) { p = ++pT; size = --sT; }
    }
  } else if (b < sT) pT[sT-b] &= ~c;

  NATURALCONDITION(*this);
}
```
◇

Macro referenced in 409.

Laufzeit. Sei $x \in$ Natural und $a \in$ Digit, dann x`.clearbit(`a`)`$\in \mathcal{O}(1)$.

⟨ tests a bit in a Natural 85b ⟩ ≡

```
bool Natural::testbit(const Digit a) const
// Algorithm:  c := b.testbit(a)
// Input:      a in Digit, b in Natural.
// Output:     c in bool such that if b and 2^a then c = true else c = false ||
{
  NATURALCONDITION(*this);

  const size_t b = size_t(a/BETA) + 1;
  const size_t sT = size;
  if (b > sT) return false;
  return ((p[sT-b] & (Digit(1) << (a%BETA))) != 0);
}
```
◇

Macro referenced in 409.

Laufzeit. Sei $x \in$ Natural und $a \in$ Digit, dann x`.testbit(`a`)`$\in \mathcal{O}(1)$.

⟨ memberfunction Natural.odd 86a ⟩ ≡
```
inline bool Natural::odd() const
// Algorithm:  c := a.odd()
// Input:      a in Natural.
// Output:     c in bool such that if 2|a then c = false else c = true ||
{
  return ((lowest() & 1) != 0);
}
```
◇
Macro referenced in 402b.

Laufzeit. Sei $x \in$ Natural, dann x`.odd()`$\in \mathcal{O}(1)$.

⟨ memberfunction Natural.even 86b ⟩ ≡
```
inline bool Natural::even() const
// Algorithm:  c := a.even()
// Input:      a in Natural.
// Output:     c in bool such that if 2|a then c = true else c = false ||
{
  return ((lowest() & 1) == 0);
}
```
◇
Macro referenced in 402b.

Laufzeit. Sei $x \in$ Natural, dann x`.even()`$\in \mathcal{O}(1)$.

2.11 Multiplikation

2.11.1 Digitoperationen

Wir wollen als erstes zwei Digits miteinander multiplizieren. Dabei erstreckt sich das Produkt zwangsläufig auf zwei Digits. Die einfache klassische Möglichkeit wäre:
Seien $a, b \in$ Digit mit $a = a_1 \cdot 2^{\beta/2} + a_0, b = b_1 \cdot 2^{\beta/2} + b_0$, dann gilt:

$$\begin{array}{rcccc} a \cdot b = & (a_1, a_0) & \cdot & (b_1, b_0) & \\ \hline & a_1 \cdot b_1 & & & \\ & & a_0 \cdot b_1 & & \\ & & a_1 \cdot b_0 & & \\ + & & & a_0 \cdot b_0 & \\ \hline \multicolumn{5}{c}{a_1 b_1 \cdot 2^{\beta} + (a_0 b_1 + a_1 b_0) \cdot 2^{\beta/2} + a_0 b_0} \end{array}$$

Und nun dasselbe als Programm in ANSI-C++:
Hierbei verwenden wir die zwei internen Konstanten $\gamma_{low} := \left\lfloor \frac{\gamma}{2^{\beta/2}} \right\rfloor$ und $\gamma_{high} := \gamma_{low} \cdot 2^{\beta/2}$:

⟨ constant γ_{low} and γ_{high} 86c ⟩ ≡
```
const Digit  GAMMA_LOW   = GAMMA >> (BETA/2);
const Digit  GAMMA_HIGH  = ~GAMMA_LOW;
```
◇
Macro referenced in 396d.

⟨ digitmul in ANSI-C++ 87a ⟩ ≡

```
inline void NumberBase::digitmul(const Digit a, const Digit b,
                                 Digit& x, Digit& y) const
// Algorithm:  n.digitmul(a, b, x, y)
// Input:      n in NumberBase, a,b in Digit.
// Output:     x,y in Digit such that x*2^BETA + y = a*b ||
{
  const Digit c = a >> BETA/2;
  const Digit d = a & GAMMA_LOW;
  const Digit e = b >> BETA/2;
  const Digit f = b & GAMMA_LOW;

  const Digit z1 = c*f;
  Digit z2 = c*e;
  Digit z3 = d*f;
  Digit z4 = d*e + z1;
  if (z4 < z1) z2 += GAMMA_LOW+1;
  z2 += z4 >> BETA/2;
  z3 += z4 <<= BETA/2;
  if (z3 < z4) ++z2;
  y = z3; x = z2;
}
◇
```

Macro referenced in 397d.

Laufzeit. Sei $n \in$ NumberBase und $a, b, d, e \in$ Digit, dann n`.digitmul(`a`,`b`,`d`,`e`)`$\in \mathcal{O}(21)$.

Der 80x86-Prozessor ermittelt intern das Produkt auf zwei Datenregistern. Doch diese Vorgehensweise ist nicht auf allen Rechnern gängig. So erzeugen manche Prozessoren das Ergebnis nur in einem Register und rufen gegebenenfalls einen internen Überlauf auf.
Betrachten wir nun eine äquivalente Multiplikation in Assembler für einen 80386-Prozessor mit dem Visual-C++ Compiler ($\beta = 32$-Bit):

⟨ digitmul in assembler for a i386 processor with the Visual-Compiler 87b ⟩ ≡

```
static void __declspec(naked) __fastcall
            _digitmul(Digit* z, const Digit a, const Digit b)
{
  __asm {
    mov eax,[esp+4]
    mul edx
    mov [ecx],edx
    mov 4[ecx],eax
    ret 4
  }
}

inline void NumberBase::digitmul(const Digit a, const Digit b, Digit& x, Digit& y) const
{
  Digit z[2];
  _digitmul(z, a, b);
  x = z[0]; y = z[1];
}
◇
```

Macro referenced in 398a.

Besonders auffallend ist die prägnante Kürze und vor allem die Möglichkeit, mit nur einer Multiplikation das Produkt zu ermitteln. Bei einem Vergleich mit der ersten ANSI-Lösung sieht es auf den ersten Blick so aus, als ob die Assembler-Variante um mindestens 5 Mal schneller ist (vergleiche die 4 Multiplikationen und den Additions-Overhead). Doch was um so viel schneller erscheint, ist es in Wirklichkeit in dem Maße gar nicht: Eine genaue Zeitanalyse auf einem Pentium-Rechner ergibt nur einen Geschwindigkeitsunterschied um etwa den Faktor 2. Dies liegt daran, daß die Compiler maschinenabhängig optimieren und so zum Beispiel auf einem Pentium zwei unabhängige Befehle gleichzeitig verarbeitet werden, weil er eine Pipeline-Tiefe von zwei hat. Das kann dann bedeuten, daß unsere 4 Multiplikationen aus dem ANSI-Programm bei einer geschickten Ablauffolge die gleiche Zeitspanne beanspruchen wie zwei voneinander abhängige Multiplikationen.
Auf modernen Prozessoren gewinnt die Pipeline-Tiefe immer mehr an Bedeutung; so hat zum Beispiel der PowerPC momentan eine Pipeline-Tiefe von vier, weitere vier sind in späteren Typen bereits geplant. Dies ist auch der Grund, warum die ANSI-Version den höchsten Stellenwert erlangen sollte, auch wenn die Assmblerlösung für die Digit-Multiplikation auf dem ausgewählten Mikroprozessor besser erscheint.

Eine etwas kürzere Formulierung und schnellere Ausführung gelingt dem Watcom-Dialekt gegenüber dem Visual-Compiler für die 32-Bit-Arithmetik auf einem 80386-Prozessor:

⟨ digitmul in assembler for a i386 processor with the Watcom-Compiler 88a ⟩ ≡

```
void _digitmul(const Digit, const Digit, Digit&, Digit&);
#pragma aux _digitmul =                \
   "mul ebx"                           \
   "mov [esi],edx"                     \
   "mov [edi],eax"                     \
   parm [EAX] [EBX] [ESI] [EDI]        \
   modify [ ESI EDI EAX EDX ];

inline void NumberBase::digitmul(const Digit a, const Digit b,
                                 Digit& x, Digit& y) const
{
  _digitmul(a, b, x, y);
}
```

◇

Macro referenced in 398a.

Ein großer Nachteil der Assembler-Realisierung ist, wie man bereits nach zwei Varianten für denselben Prozessor sieht, daß die `asm`-Deklaration keinen Standard anbietet. Der C++-Compiler fordert zum Beispiel von GNU nocheinmal einen völlig anderen Assembler[3], obwohl es sich nach wie vor um den gleichen 80386-Prozessor dreht:

⟨ digitmul in assembler for a i386 processor with the GNU-Compiler 88b ⟩ ≡

```
inline void NumberBase::digitmul(const Digit a, const Digit b,
                                 Digit& x, Digit& y) const
{
  __asm__ ("mull %3"
    : "=a" (y), "=d" (x)
    : "%0" (a), "rm" (b));
}
```

◇

Macro referenced in 398a.

[3] Mit dem GNU-Assembler wird eine optimale Lösung erreicht, weil die Datenübergabe vollständig vom Compiler übernommen wird und dadurch optimiert werden kann.

Selbstverständlich wechselt der Assembler-Code auch bei einem anderen Prozessor, wie zum Beispiel bei der SPARC-v8 Architektur:

⟨ digitmul in assembler for a SPARC v8 processor with the GNU-Compiler 89a ⟩ ≡

```
inline void NumberBase::digitmul(const Digit a, const Digit b,
                                 Digit& x, Digit& y) const
{
  __asm__ ("umul %2,%3,%1; rd %%y,%0"
    : "=r" (x), "=r" (y)
    : "r" (a), "r" (b));
}
◇
```

Macro referenced in 398a.

Anhand dieser vier verschiedenen Assembler-Multiplikationen wird deutlich, daß wir nicht nur prozessorabhängig programmieren, sondern uns auch gleichzeitig auf einen Compiler einschränken. Es ist offensichtlich, daß wir auf heutigen Maschinen höchstens einen Faktor 3 verlieren:

Prozessor/ Compiler	`digitmul` ANSI-C++`inline`	`digitmul` ANSI-C++	`digitmul` Assembler
Pentium:			
Watcom C++ 10.6	100%	262,5%	38,4%
Visual C++ 4.0	100%	474,7%	220,8%
GNU-C 2.7.2:			
Pentium	100%	199,7%	34,1%
SPARCstation10	100%	200,4%	60,1%
UltraSPARC	100%	201,3%	88,1%

Aus diesem Grund und weil die Maschinen und Compiler stets Veränderungen unterworfen sind, ist es unnötig, seine Zeit und Kraft auf Spezialisierungen zu verwenden.
Daher ist es sinnvoll, die ANSI-Version zu bevorzugen und bei ausgewählten Compilern und Maschinen zusätzlich eine Assemblerlösung hinzuzufügen. Der Assembler sollte jedoch keine Überhand nehmen, wie es zum Beispiel bei der GMP-2.0.2 praktiziert wurde (siehe [20]).

Multiplikation zweier Digits

⟨ digitmul of a Digit with a double Digit 89b ⟩ ≡

```
inline void NumberBase::digitmul(const Digit a0, const Digit b0, const Digit b1,
                                 Digit* c) const
// Algorithm:  n.digitmul(a0, b0, b1, d)
// Input:      n in NumberBase, a0,b0,b1 in Digit.
// Output:     d = [d0, d1, d2] in Digit^4
//             such that d0*4^BETA+d1*2^BETA+d2 = a0*(b0*2^BETA+b1) ||
{
  Digit x,y;
  digitmul(a0, b0, c[0], x);
  digitmul(a0, b1, y, c[2]);
  c[1] = y += x;
  if (y < x) ++c[0];
}
◇
```

Macro referenced in 397d.

⟨ digitmul of a double Digit 89c ⟩ ≡

```
inline void NumberBase::digitmul(const Digit a0, const Digit a1, const Digit b0,
                                 const Digit b1, Digit* c) const
// Algorithm:  n.digitmul(a0, a1, b0, b1, d)
// Input:      n in NumberBase, a0,a1,b0,b1 in Digit.
// Output:     d = [d0, d1, d2, d3] in Digit^4 such that
//             d0*8^BETA+d1*4^BETA+d2*2^BETA+d3 = (a0*2^BETA+a1)*(b0*2^BETA+b1) ||
{
  Digit x0,x1,y0,y1,c0,c1,c2;
  digitmul(a0, b0, c0, c1);
  digitmul(a0, b1, x0, x1);
  digitmul(a1, b0, y0, y1);
  digitmul(a1, b1, c2, c[3]);
  x1 += y1;
  if (x1 < y1) {
    x0 += y0+1;
    if (x0 <= y0) ++c0;
  } else {
    x0 += y0;
    if (x0 < y0) ++c0;
  }
  c[2] = c2 += x1;
  if (c2 < x1) {
    c[1] = c1 += x0+1;
    if (c1 <= x0) ++c0;
  } else {
    c[1] = c1 += x0;
    if (c1 < x0) ++c0;
  }
  c[0] = c0;
}
◇
```

Macro referenced in 397d.

Quadratur zweier Digits

⟨ digitsqr of a double Digit 90 ⟩ ≡

```
inline void NumberBase::digitsqr(const Digit a0, const Digit a1, Digit* b) const
// Algorithm:  n.digitsqr(a0, a1, b)
// Input:      n in NumberBase, a0,a1 in Digit.
// Output:     b = [b0, b1, b2, b3] in Digit^4
//             such that b0*8^BETA+b1*4^BETA+b2*2^BETA+b3 = (a0*2^BETA+a1)^2 ||
{
  Digit x0,x1,b0,b1,b2;
  digitmul(a0, a0, b0, b1);
  digitmul(a0, a1, x0, x1);
  digitmul(a1, a1, b2, b[3]);
  b0 += x0 >> (BETA-1); x0 <<= 1; x0 |= x1 >> (BETA-1); x1 <<= 1;
  b[2] = b2 += x1;
  if (b2 < x1) {
    b[1] = b1 += x0+1;
    if (b1 <= x0) ++b0;
  } else {
    b[1] = b1 += x0;
    if (b1 < x0) ++b0;
  }
```

```
  b[0] = b0;
}
◇
```

Macro referenced in 397d.

2.11.2 Konventionelle Multiplikation

Um $a, b \in$ Natural miteinander zu multiplizieren, müssen wir mit der folgenden konventionellen Multiplikation die Funktion `digitmul` $\mathcal{L}(a) \cdot \mathcal{L}(b)$ Mal aufrufen:

⟨ conventional multiplication algorithm 91a ⟩ ≡

```
void Natural::cmul(const Digit* pA, size_t sA, const Digit* pB, const size_t sB,
                   Digit* pC) const
// Algorithm:  x.cmul(r, n, s, m, t)
//             Let a,b,c in Natural where c.p not in {a.p, b.p}.
// Input:      x in Natural, n,m in size_t where n >= m > 0,
//             r,r+n in [a.p, a.p+L(a)],
//             s,s+m in [b.p, b.p+L(b)],
//             t,t+n+m in [c.p, c.p+L(c)].
// Output:     [t, t+n+m[ = [r, r+n[ * [s, s+m[ ||
{
  CONDITION(sA >= sB && sB > 0);

  const Digit* pE = pC+sB;
  FILL_ZERO(pC, pE);
  const Digit* pF = pB;
  pE = pA; pA += sA; pB += sB; pC += sA;
  mul(pE, pA, pC, *--pB);
  while (pB > pF) muladd(pE, pA, --pC, *--pB);
}
◇
```

Macro referenced in 409.

Bemerkung. Dieser Algorithmus durchläuft immer die größte Zahl in der inneren Schleife, um eine bessere Laufzeit zu erreichen.

Die mathematische Deutung hierzu ist:
Seien $a, b \in$ Natural, dann

$$a \cdot b = \left(\sum_{i=0}^{\mathcal{L}(a)-1} a_i 2^{\beta i} \right) \cdot \left(\sum_{i=0}^{\mathcal{L}(b)-1} b_i 2^{\beta i} \right) = \sum_{i=0}^{\mathcal{L}(a)-1} a_i \cdot \left(\sum_{j=0}^{\mathcal{L}(b)-1} b_j 2^{\beta j} \right) 2^{\beta i} = \sum_{i=0}^{\mathcal{L}(a)-1} (a_i \cdot b) 2^{\beta i}.$$

Dafür ist die Multiplikation eines Naturals mit einem Digit erforderlich, die die folgende Implementation hat:

⟨ multiplication of a Natural by a Digit 91b ⟩ ≡

```
void Natural::mul(const Natural& a, const Digit b)
// Algorithm:  c.mul(a, b)
// Input:      a in Natural, b in Digit where not a.p = c.p.
// Output:     c in Natural such that c = a*b ||
{
```

```
  NATURALCONDITION(a);
  CONDITION(a.p != p);

  const Digit* pA = a.p;
  const size_t sA = a.size;
  const size_t sT = sA+1;
  Digit* pT = setsize(sT);
  if (mul(pA, pA+sA, pT+sT, b) == 0) normalize();

  NATURALCONDITION(*this);
}
```
◇

Macro referenced in 409.

Laufzeit. Seien $x, y \in$ Natural und $a \in$ Digit, dann y`.mul(`x`,`a`)`$\in \mathcal{O}(\mathcal{L}(x))$.

Zuweisungsoperator

⟨ assign `operator*=` of a Natural with a Digit 92 ⟩ ≡

```
Natural& Natural::operator*=(const Digit a)
// Algorithm:  c := c *= a
// Input:      a in Digit, c in Natural.
// Output:     c in Natural such that c := c*a ||
{
  NATURALCONDITION(*this);

  Digit* pT = p;
  size_t sT = size;
  if (sT == 1) {
    const Digit x = *pT;
    pT = setsize(2);
    digitmul(x, a, pT[0], pT[1]);
    normalize();
  } else if (sT == 2) {
    const Digit x0 = pT[0];
    const Digit x1 = pT[1];
    pT = setsize(3);
    digitmul(a, x0, x1, pT);
    normalize();
  } else {
    const Digit* rT = root;
    if (pT-rT == 1) {
      enlarge(DELTA);
      pT = p;
    }
    Digit* pE = pT+sT;
    if (mul(pT, pE, pE, a)) {
      p = --pT; size = ++sT;
    } else normalize();
  }

  NATURALCONDITION(*this);

  return *this;
}
```
◇

Macro referenced in 409.

Laufzeit. Sein $x \in$ Natural und $a \in$ Digit, dann x*=$a \in \mathcal{O}(\mathcal{L}(x))$.

Unterprogramm für die Multiplikation

⟨ memberfunction Natural.mul 93a ⟩ ≡

```
Digit Natural::mul(const Digit* pE, const Digit* pA, Digit* pT,
                   const Digit c) const
// Algorithm:  d.mul(r, s, t, c)
//             Let a,b in Natural where R(b) > L(a)+1.
// Input:      d in Natural,
//             r,s in [a.p, a.p+L(a)] where r < s,
//             t in [b.p, b.p+L(b)] where t-s+r-1 also in [b.root+1, b.p+L(b)].
// Output:     [t-(s-r)-1, t[ = [r, s[ * c ||
//
// Note:       t-s+r-1 in [b.root+1, b.p+L(b)].
{
  CONDITION(pE < pA);

  Digit x,y,z = 0;
  do {
    digitmul(*--pA, c, x, y);
    y += z;
    z = x + (y < z);
    *--pT = y;
  } while (pA != pE);
  *--pT = z;
  return z;
}
◇
```

Macro referenced in 409.

Gleichzeitige Multiplikation und Addition

Bei der Langzahlmultiplikation benötigen wir eine Funktion, die ein Natural mit einem Digit nur temporär multipliziert und dann das Ergebins zu einem anderen Natural hinzuaddiert. Eine simple Implementation dazu ist:

```
inline void Natural::muladd(const Natural& a, const Digit b)
{
  Natural tmp = a; tmp *= b;
  *this += tmp;
}
```

Jedoch ist hierfür ein dreimaliges Durchlaufen der Länge $\mathcal{L}(a)$ notwendig. Eine daher verbesserte Implementation ist:

⟨ multiplication and addition of a Natural by a Digit 93b ⟩ ≡

```
void Natural::muladd(const Natural& a, const Digit b)
// Algorithm:  c.muladd(a, b)
// Input:      a,c in Natural, b in Digit.
// Output:     c in Natural such that c := c + a*b ||
```

```
{
  NATURALCONDITION(*this);
  NATURALCONDITION(a);

  const size_t sA = a.size;
  const size_t sT = size;
  const Digit* rT = root;
  Digit* pT = p;
  if (rT+sA+2 >= pT+sT) {       // (a.size+2 >= rootsize())?
    enlarge(sA-sT+2+DELTA);
    pT = p;
  }
  Digit* pA = a.p;
  pT += sT;
  Digit* pC = muladd(pA, pA+sA, pT, b);
  const size_t sz = pT-pC;
  if (sz > sT) { size = sz; p = pC; }
  normalize();

  NATURALCONDITION(*this);
}
◇
```

Macro referenced in 409.

Laufzeit. Seien $x, y \in$ Natural und $a \in$ Digit, dann x.muladd$(y, a) \in \mathcal{O}(\mathcal{L}(y)$.

⟨ protected memberfunction Natural.muladd 94 ⟩ ≡

```
Digit* Natural::muladd(const Digit* pE, const Digit* pA, Digit* pT,
                       const Digit c) const
// Algorithm:  b.muladd(r, s, t, c)
//             Let a in Natural.
// Input:      b in Natural where R(b) > L(a)+2,
//             r,s in [a.p, a.p+L(a)] where r < s,
//             t in [b.p, b.p+L(b)] where t-s+r-1 also in [b.root+2, b.p+L(b)].
// Output:     [t-(s-r)-1, t[ := [t-(s-r)-1, t[ + [r, s[ * c ||
//
// Note:       t-s+r-1 in [b.root+2, b.p+L(b)].
{
  CONDITION(pE < pA);

  Digit x,y,z = 0;
  do {
    digitmul(*--pA, c, x, y);
    y += z;
    z = x + (y < z);
    x = *--pT;
    y += x;
    z += (y < x);
    *pT = y;
  } while (pA != pE);
  x = *--pT;
  *pT = x += z;
  if (x < z)
    while (++(*--pT) == 0);
  return pT;
```

```
}
◇
```

Macro referenced in 409.

Bei dieser Implementation sollten wir uns noch davon überzeugen, daß im Digit `z` kein unerwünschter Überlauf entstehen kann. Dies sieht man aber schnell, da für das Doppel-Digit

$$(\mathtt{x}, \mathtt{y}) \leq \gamma^2 = 2^{2\beta} - 2^{\beta+1} + 1 = (2^\beta - 2, 1)$$

gilt. Und falls nun `z` maximal mit $\mathtt{z} = \gamma$ ist, dann erhalten wir

$$(\mathtt{x}, \mathtt{y}) + \mathtt{z} = (\gamma, 0).$$

Folglich ist $\mathtt{y} = 0$, und somit kann kein Überlauf in `x` mehr entstehen.

2.11.3 Karatsuba-Multiplikation

Für große Zahlen existieren wesentlich bessere Algorithmen, zum Beispiel, indem wir unsere Zahl folgendermaßen aufteilen:

Seien $a, b \in$ Natural mit $\mathcal{L}(a) = \mathcal{L}(b) = n$, so gilt:

$$\begin{aligned} a \cdot b &= a_1 b_1 2^{2m\beta} + (a_1 b_0 + a_0 b_1) 2^{m\beta} + a_0 b_0 \\ &= a_1 b_1 2^{2m\beta} + ((a_1 - a_0)(b_0 - b_1) + a_1 b_1 + a_0 b_0) 2^{m\beta} + a_0 b_0 \end{aligned}$$

mit $m := \lfloor n/2 \rfloor, a := 2^{m\beta} a_1 + a_0$ und $b := 2^{m\beta} b_1 + b_0$.

Wir wollen diese Methode nun verallgemeinern und dabei die Implementation so geschickt umsetzen, daß unser entstandener Mehraufwand an Speicherallokationen und Additionen nicht zu sehr ins Gewicht fallen.
Ein trivialer Ansatz zur Lösung dieses Problems ist die Ermittlung der kleinsten Zweierpotenz, die gerade noch größer oder gleich unserer Längen von a und b ist. Somit hätten wir zwar beim rekursiven Absteigen immer ein gerades n, die unnötig operierenden Nullen und der bei einer Vergößerung wahrscheinlich entstehende Überlauf machen diesen Ansatz aber wieder unbrauchbar.

2.11.4 Anzahl der elementaren Operationen

Der konstante Aufwand für eine elementare Multiplikation sei $m_0 \in \mathbb{N}$ und für eine elementare Addition beziehungsweise Subtraktion $p_0 \in \mathbb{N}$. Der Trade-Off-Point zwischen dem Karatsuba-Algorithmus und der konventionellen Multiplikation sei durch $t \in \mathbb{N}$ bestimmt.

Die Anzahl der Multiplikationen – für zwei Naturals der Länge $2^k \in \mathbb{N}$ – ergibt sich somit aus der folgenden rekursiven Formel:

$$\begin{aligned} M(t) &= m_0 \cdot 4^t \\ M(k) &= 3 \cdot M(k-1) \qquad \text{für } k > t. \end{aligned}$$

Also $M(k) = 4^t \cdot 3^{k-t} \cdot m_0$ für $k > t$.

Die Anzahl der Additionen (und Subtraktionen) wird mit der folgenden rekursiven Formel berechnet:

$$\begin{aligned} P(t) &= p_0(4^t - 2^t) \\ P(k) &= 8 \cdot 2^{k-1} p_0 + 3 \cdot P(k-1) \qquad \text{für } k > t. \end{aligned}$$

Also

$$\begin{aligned} P(k) &= p_0 \left(3^{k-t} \left(4^t - 2^t\right) + \sum_{n=0}^{k-t-1} 2^{k-n+2} 3^n \right) \\ &= p_0 \left(3^{k-t} \left(4^t - 2^t\right) + 2^{k+2} \sum_{n=0}^{k-t-1} \left(\frac{3}{2}\right)^n \right) \\ &= p_0 \left(3^{k-t} \left(4^t - 2^t\right) + 2^{k+2} \frac{\left(\frac{3}{2}\right)^{k-t} - 1}{\frac{3}{2} - 1} \right) \\ &= p_0 \left(3^{k-t} \left(4^t - 2^t\right) + 2^{t+3} \left(3^{k-t} - 2^{k-t}\right)\right) \qquad \text{für } k > t. \end{aligned}$$

Schließlich erhalten wir dann die gesuchte Formel für die Anzahl der benötigten Operationen für die Multiplikation zweier Naturals der Länge 2^k:

$$T(k) = \begin{cases} 3^{k-t} \left(4^t(m_0 + p_0) + 7 \cdot 2^t p_0\right) - 2^{k+3} p_0 \ , & k > t \\ 4^k(m_0 + p_0) - 2^k p_0 \ , & k \leq t \end{cases} .$$

Beispiele.

1. Konventionelle Methode ($t = k$):

$$T(k) = 4^k m_0 + p_0(4^k - 2^k).$$

2. Karatsuba-Algorithmus ($t = 0$):

$$T(k) = 3^k m_0 + 8p_0(3^k - 2^k).$$

Trade-Off-Point

Der Trade-Off-Point kann aus theoretischer Sicht auf $t = 2$ gesetzt werden, weil wir dann für die Multiplikation zweier Naturals der Länge 16 zum Beispiel 268 Additionen/Subtraktionen und 144 Multiplikationen benötigen, wohingegen die konventionelle Methode 240 Additionen und 256 Multiplikationen beansprucht. Damit sparen wir also 84 Operationen ein.
Selbst bei der Betrachtung der Multiplikation zweier Naturals der Länge 8 benötigen wir 68 Additionen/Subtraktionen und 48 Multiplikationen, wohingegen die konventionelle Methode 56 Additionen und 64 Multiplikationen verwendet, und damit mehr zeitaufwendigere Multiplikationen braucht.

2.11.5 Beseitigung der Rekursion

Schematisch läßt sich die rekursive Struktur der Karatsuba-Multiplikation folgendermaßen beschreiben:

```
void mul()
{
  if (trade_off()) conventional();
  else {
L1:  split();
     mul();
L2:  mul();
L3:  abs(); abs();
     mul();
L4:  add(); add(); add();
     unify();
  }
}
```

Seien nun $a, b \in \mathsf{Natural}$ mit $\mathcal{L}(a) = \mathcal{L}(b)$ die zu multiplizierenden Zahlen, so entsteht eine Rekursionstiefe von

$$n = \min\{0, 1 + \log_2 \mathcal{L}(a) - \texttt{trade\_off}\}.$$

Damit haben wir n geschachtelte Schleifen. Die Rekursion kann man nun mit Hilfe eines durch Nullen initialisierten Arrays `st` der Länge n und der `switch`-Anweisung beseitigen:

```
if (trade_off()) conventional();
else {
  i = sizeof(st)-1;
  n = L(a);
  do {
    if (trade_off()) { conventional(); ++i; }
    switch (++st[i]) {
      case 1:
        n /= 2;                 // split
        --i;
        break;
      case 2:
        --i;
        break;
      case 3:
        abs(); abs();
        --i;
        break;
      case 4:
        add(); add(); add();
        n *= 2;                 // unify
        st[i++] = 0;
        break;
    }
  } while (i < sizeof(st));
}
```

2.11.6 Multiplikationsalgorithmus

Nach der ganzen Theorie gelangen wir nun endlich zur Implementierung des Multiplikationsalgorithmus:

⟨ multiplication of two Naturals 97 ⟩ ≡

```
void Natural::mul(const Natural& a, const Natural& b)
// Algorithm:  c.mul(a, b)
// Input:      a,b in Natural.
// Output:     c in Natural such that c = a*b ||
{
  NATURALCONDITION(a);
  NATURALCONDITION(b);
  NATURAL_FOR_CHECK(_a, a);
  NATURAL_FOR_CHECK(_b, b);

  const size_t sA = a.size;
  const size_t sB = b.size;
  const Digit* pA = a.p;
  const Digit* pB = b.p;

  if (sA <= 2 && sB <= 2) {
    if (sA == 1) {
      const Digit x = *pA;
      const Digit y = *pB;
      if (sB == 1) {
        Digit* pT = setsize(sB+1);
        digitmul(x, y, pT[0], pT[1]);
      } else {
        const Digit z = pB[1];
        Digit* pT = setsize(sB+1);
        digitmul(x, y, z, pT);
      }
    } else {
      const Digit x0 = pA[0];
      const Digit x1 = pA[1];
      const Digit y0 = pB[0];
      if (sB == 1) {
        Digit* pT = setsize(sB+2);
        digitmul(y0, x0, x1, pT);
      } else {
        const Digit y1 = pB[1];
        Digit* pT = setsize(sB+2);
        digitmul(x0, x1, y0, y1, pT);
      }
    }
    normalize();
  } else if (sA >= FFT_MUL_MARK && sB >= FFT_MUL_MARK
             && max(sA, sB) <= FFT::max_size()) {
    FFT f(a, false);
    FFT g(b, true, &f);
    g.mul(*this);
  } else {
    Digit* pT = p;
    if (sA < sB)
      if (pA == pT) {
        Natural t(a);
        pT = setsize(sA+sB);
        mul(pB, sB, t.p, sA, pT);
      } else if (pB == pT) {
        Natural t(b);
        pT = setsize(sA+sB);
```

```
        mul(t.p, sB, pA, sA, pT);
      } else {
        pT = setsize(sA+sB);
        mul(b.p, sB, pA, sA, pT);
      }
    else
      if (pA == pT) {
        Natural t(a);
        pT = setsize(sA+sB);
        mul(t.p, sA, pB, sB, pT);
      } else if (pB == pT) {
        Natural t(b);
        pT = setsize(sA+sB);
        mul(pA, sA, t.p, sB, pT);
      } else {
        pT = setsize(sA+sB);
        mul(pA, sA, pB, sB, pT);
      }
    normalize();
  }

  NATURAL_FOR_CHECK(_c, _a+_b);
  CONDITION((*this*2) == (_c*_c-_a*_a-_b*_b));
  NATURALCONDITION(*this);
}
◇
```

Macro referenced in 409.

Laufzeit. Seien $x, y, z \in$ Natural, dann z`.mul(`x`,`y`)`$\in \mathcal{O}(\max\{\mathcal{L}(x), \mathcal{L}(y)\}^{\log_2(3)})$.

Multiplikation zweier Naturals gleicher Länge

Bei der Realisierung der Langzahlmultiplikation gelten die folgenden Beziehungen:

Seien $a, b \in$ Natural mit $\mathcal{L}(a) = \mathcal{L}(b)$, dann

$$\begin{aligned} a \cdot b &= (c2^{2n\beta} + a_1 2^{n\beta} + a_0) \cdot (d2^{2n\beta} + b_1 2^{n\beta} + b_0) \\ &= cd2^{4n\beta} + c(b_1 2^{n\beta} + b_0)2^{2n\beta} + d(a_1 2^{n\beta} + a_0)2^{2n\beta} \\ &\quad +((a_1 - a_0)(b_0 - b_1) + a_1 b_1 + a_0 b_0)2^{n\beta} + a_0 b_0 \\ &= cd2^{4n\beta} + c(b_1 2^{n\beta} + b_0)2^{2n\beta} + d(a_1 2^{n\beta} + a_0)2^{2n\beta} \\ &\quad +(-\text{sign}(a_0 - a_1)\text{sign}(b_0 - b_1)|a_0 - a_1||b_0 - b_1| + a_1 b_1 + a_0 b_0)2^{n\beta} + a_0 b_0 \end{aligned} \tag{2.1}$$

mit $n := \lfloor \frac{\mathcal{L}(a)}{2} \rfloor$, $c, d \in$ Digit.

So haben wir drei Multiplikationen mit Faktoren der Länge n. Wenn wir jetzt weiter rekursiv absteigen, bleibt stets die Forderung erhalten, daß die beiden Faktoren dieselbe Länge besitzen.

⟨ multiplication with equal size 99 ⟩ ≡

```
void Natural::mul(const Digit* pA, const Digit* pB, const size_t sz, Digit* pC) const
// Algorithm:  c.mul(r, s, n, t)
//             Let a,b in Natural.
// Input:      c in Natural where not c.p in {a.p, b.p},
```

```
//             n in size_t where n > KARATSUBA_MUL_MARK,
//             r,r+n in [a.root, a.p+L(a)],
//             s,s+n in [b.root, b.p+L(b)],
//             t,t+2*n in [c.root, c.p+L(c)].
// Output:     [t, t+2*n[ = [r, r+n[ * [s, s+n[ ||
{
  ⟨ initialization of multiplication algorithm 100a ⟩
  do {
    if (n <= KARATSUBA_MUL_MARK) {
      cmul(pA, n, pB, n, pC);
      h -= 4*n;
      pA = st[++i].a; pB = st[i].b; pC = st[i].c;
    }
    ⟨ nonrecursive multiplication algorithm 100b ⟩
  } while (n < sz);
  delete[] h;
  delete[] st;
}
◇
```

Macro referenced in 409.

Zur Steuerung der Iteration benötigen wir die folgende Initialisierung:

⟨ initialization of multiplication algorithm 100a ⟩ ≡

```
struct stack {
  const Digit* a;
  const Digit* b;
  Digit* c;
  char   cr,d;
};

size_t n = sz;
size_t i = 0;
Digit* h = NOTHROW_NEW Digit[8*n];
if (!h) errmsg(2, "(mul)");
while (n >>= 1) ++i;
stack* st = NOTHROW_NEW stack[i];
for (n = 0; n < i; ++n) st[n].d = 0;
n = sz;
--i;
◇
```

Macro referenced in 99.

Durch die vorherige Gleichung (2.1) und die Vorgabe beliebiger Naturals läßt sich der Algorithmus in fünf Berechnungsteile zerlegen:

⟨ nonrecursive multiplication algorithm 100b ⟩ ≡

```
switch (++st[i].d) {
  ⟨ case 1 of multiplication 101a ⟩
  ⟨ case 2 of multiplication 101b ⟩
  ⟨ case 3 of multiplication 101c ⟩
  ⟨ case 4 of multiplication 101d ⟩
  ⟨ case 5 of multiplication 102 ⟩
}◇
```

Macro referenced in 99.

1. Berechnung von $a_0 b_0$:

⟨ case 1 of multiplication 101a ⟩ ≡

```
case 1:
  st[i].cr = 0;
  if (n&1) {
    ++st[i].cr;
    digitmul(*pA, *pB, *pC, pC[1]);
    ++pA; ++pB; pC += 2;
  }
  n >>= 1; st[i].a = pA; st[i].b = pB; st[i].c = pC;
  h += 4*n; --i;
  break;◇
```

Macro referenced in 100b.

2. Berechnung von $a_1 b_1$:

⟨ case 2 of multiplication 101b ⟩ ≡

```
case 2:
  pA += n; pB += n; pC += 2*n;
  h += 4*n; --i;
  break;◇
```

Macro referenced in 100b.

3. Berechnung von $|a_0 - a_1| \cdot |b_0 - b_1|$:

⟨ case 3 of multiplication 101c ⟩ ≡

```
case 3:
  if (abs(h, pA, pA+n, n)*abs(h+n, pB, pB+n, n) == -1) ++st[i].d;
  pA = h; pB = h+n; pC = h+2*n;
  h += 4*n; --i;
  break;◇
```

Macro referenced in 100b.

4. Zusammensetzung für $\mathrm{sign}(a_0 - a_1) \cdot \mathrm{sign}(b_0 - b_1) \geq 0$:

⟨ case 4 of multiplication 101d ⟩ ≡

```
case 4:
  pC += n;
  {
    Digit k = (sub_no_dec(h+2*n, h+4*n, pC+n) == true)
            + (sub_no_dec(h+2*n, h+4*n, pC+3*n) == true);   // k >= 1!
    k -= (sub_no_dec(pC, pC+2*n, h+4*n) == true);
    Digit* pPos = pC;
    *--pPos += k;
    if (*pPos < k)
      while (++(*--pPos) == 0);
  }
  st[i].d = 0;
  if (st[i].cr) {
    muladd(pB, pB+2*n, pC+n, *(pA-1));
    muladd(pA, pA+2*n, pC+n, *(pB-1));
    n <<= 1; ++n;
  } else n <<= 1;
  if (n < sz) { pA = st[++i].a; pB = st[i].b; pC = st[i].c; h -= 4*n; }
  break;◇
```

Macro referenced in 100b.

5. Zusammensetzung für $\text{sign}(a_0 - a_1) \cdot \text{sign}(b_0 - b_1) = -1$:

⟨ case 5 of multiplication 102 ⟩ ≡

```
case 5:
  pC += n;
  {
    Digit k = (add_no_inc(h+2*n, h+4*n, pC+n) == true)
            + (add_no_inc(h+2*n, h+4*n, pC+3*n) == true);
    k += (add_no_inc(pC, pC+2*n, h+4*n) == true);
    Digit* pPos = pC;
    *--pPos += k;
    if (*pPos < k)
      while (++(*--pPos) == 0);
  }
  st[i].d = 0;
  if (st[i].cr) {
    muladd(pB, pB+2*n, pC+n, *(pA-1));
    muladd(pA, pA+2*n, pC+n, *(pB-1));
    n <<= 1; ++n;
  } else n <<= 1;
  if (n < sz) { pA = st[++i].a; pB = st[i].b; pC = st[i].c; h -= 4*n; }
  break;◇
```

Macro referenced in 100b.

Multiplikation zweier Naturals unterschiedlicher Länge

Wir nehmen ohne Einschränkung $\mathcal{L}(a) > \mathcal{L}(b)$ an, weil wir die beiden Naturals a und b vertauschen können. Somit existieren eindeutige $q, r \in \mathbb{N}$ mit $r < \mathcal{L}(b)$ derart, daß $\mathcal{L}(a) = q \cdot \mathcal{L}(b) + r$ gilt. Also ist es möglich, die folgende Aufspaltung durchzuführen:

$$
\begin{aligned}
a \cdot b &= \left(\sum_{i=0}^{\mathcal{L}(a)-1} a_i 2^{\beta i}\right) \cdot \left(\sum_{i=0}^{\mathcal{L}(b)-1} b_i 2^{\beta i}\right) \\
&= \left(\sum_{i=0}^{q-1}\sum_{j=0}^{\mathcal{L}(b)-1} a_{i\cdot\mathcal{L}(b)+j} 2^{\beta(i\cdot\mathcal{L}(b)+j)} + \sum_{i=0}^{r-1} a_{q\cdot\mathcal{L}(b)+i} 2^{\beta(q\cdot\mathcal{L}(b)+i)}\right) \cdot \left(\sum_{i=0}^{\mathcal{L}(b)-1} b_i 2^{\beta i}\right) \\
&= \left(\sum_{i=0}^{q-1} c_{i\cdot\mathcal{L}(b)} 2^{\beta i\cdot\mathcal{L}(b)}\right) + \left(\sum_{i=0}^{r-1} a_{q\cdot\mathcal{L}(b)+i} 2^{\beta(q\cdot\mathcal{L}(b)+i)}\right) \cdot \left(\sum_{i=0}^{\mathcal{L}(b)-1} b_i 2^{\beta i}\right) \\
&\quad \text{mit } c_i = \left(\sum_{j=0}^{\mathcal{L}(b)-1} a_{i+j} 2^{\beta j}\right) \cdot \left(\sum_{j=0}^{\mathcal{L}(b)-1} b_j 2^{\beta j}\right)
\end{aligned}
$$

Falls nun $r_1 := r = 0$ ist, tritt keinerlei zusätzlicher Aufwand ein, da $\mathcal{L}(b)$ ein Teiler von $\mathcal{L}(a)$ ist und somit nur gleichlange Zahlen miteinander multipliziert werden. Andernfalls existieren eindeutige $q_2, r_2 \in \mathbb{N}$ mit $\mathcal{L}(b) = q_2 \cdot r_1 + r_2$ und $r_2 < r_1$, weshalb wir unser obiges Verfahren rekursiv fortsetzen können. Weil unsere Folge $(r_n)_{n\in\mathbb{N}}$ streng monoton fallend ist, existiert ein $n \in \mathbb{N}$ mit $r_n \neq 0$ und $r_{n+1} = 0$, wodurch unser Verfahren ein Ende hat.

Bemerkung. Der ungünstigste Fall, das heißt die längste Rekursionstiefe, tritt zum Beispiel auf, wenn die Länge der beiden zu multiplizierenden Zahlen zwei aufeinanderfolgende Fibonacci-Zahlen sind.

⟨ multiplication with different size 103a ⟩ ≡

```
void Natural::mul(const Digit* pA, size_t sA, const Digit* pB, const size_t sB,
                  Digit* pC) const
// Algorithm:  c.mul(r, n, s, m, t)
//             Let a,b in Natural.
// Input:      c in Natural where not c.p in {a.p, b.p},
//             n,m in size_t where n >= m > 0,
//             r,r+n in [a.root, a.p+L(a)],
//             s,s+m in [b.root, b.p+L(b)],
//             t,t+n+m in [c.root, c.p+L(c)].
// Output:     [t, t+n+m[ = [r, r+n[ * [s, s+m[ ||
{
  CONDITION(sA >= sB && sB > 0);

  if (sB <= KARATSUBA_MUL_MARK) cmul(pA, sA, pB, sB, pC);
  else {
    sA -= sB;
    if (sA) {
      pA += sA;
      const Digit* pE = pC+sA;
      FILL_ZERO(pC, pE);
    }
    mul(pA, pB, sB, pC);
    Digit* pT = NOTHROW_NEW Digit[sA+sB];
    if (!pT) errmsg(2, "(mul)");
    while (sA >= sB) {
      pA -= sB; pC -= sB; sA -= sB;
      mul(pA, pB, sB, pT);
      if (add_no_inc(pC, pC+2*sB, pT+2*sB))
        for (Digit* pPos = pC; ++(*--pPos) == 0;);
    }
    if (sA) {
      pA -= sA; pC -= sA;
      mul(pB, sB, pA, sA, pT);
      sA += sB;
      add_no_inc(pC, pC+sA, pT+sA);
    }
    delete[] pT;
  }
}
```

◇

Macro referenced in 409.

Trade-Off-Point

Um einen effizienten Algorithmus zu erreichen, werden wir auf der unteren Ebene die Multiplikation konventionell durchführen und für größere Zahlen zuerst mit der Karatsuba-Methode beginnen und ab einem vorgegebenen Trade-Off-Point

⟨ trade off points for the Karatsuba algorithm 103b ⟩ ≡

```
const size_t KARATSUBA_MUL_MARK = 8;
const size_t KARATSUBA_SQR_MARK = 2*KARATSUBA_MUL_MARK;
```

◇

Macro referenced in 409.

auf die konventionelle Methode überleiten. Diese Konstanten wurden empirisch ermittelt und sind für die verschiedenen Systeme zwar nicht unbedingt optimal, liefern aber immer gute Resultate.

Multiplikationsoperatoren

Ein gut optimierender Compiler kann zum Zeitpunkt des Compilierens entscheiden, ob es sich um eine Quadratur oder um eine allgemeine Multiplikation handelt.

⟨ multiplicative operator* for Naturals 104a ⟩ ≡

```
inline Natural::Natural(const binder_arguments<Natural, Natural,
                                               Natural_multiplies_tag>& a)
{
  if (&a.x == &a.y) {
    get_memory(2*a.x.size+DELTA);
    sqr(a.x);
  } else {
    get_memory(a.x.size+a.y.size+DELTA);
    mul(a.x, a.y);
  }
}

inline Natural& Natural::operator=(const binder_arguments<Natural, Natural,
                                                          Natural_multiplies_tag>& a)
{
  if (&a.x == &a.y) sqr(a.x);
  else mul(a.x, a.y);
  return *this;
}

inline binder_arguments<Natural, Natural, Natural_multiplies_tag>
 operator*(const Natural& a, const Natural& b)
// Algorithm:  c := a*b
// Input:      a,b in Natural.
// Output:     c in Natural such that c = a*b ||
{
  return binder_arguments<Natural, Natural, Natural_multiplies_tag>(a, b);
}
```

◇

Macro referenced in 402b.

Laufzeit. Seien $x, y, z \in$ Natural, dann $\begin{array}{lcl} x*x & \sim & z.\mathtt{sqr}(x), \\ x*y & \sim & z.\mathtt{mul}(x,y). \end{array}$

⟨ multiplicative operator* of a Natural with a Digit 104b ⟩ ≡

```
inline Natural::Natural(const binder_arguments<Natural, Digit,
                                               Natural_multiplies_tag>& a)
{
  get_memory(a.x.size+DELTA);
  mul(a.x, a.y);
}

inline Natural& Natural::operator=(const binder_arguments<Natural, Digit,
                                                          Natural_multiplies_tag>& a)
{
```

```
  if (this == &a.x) return *this *= a.y;
  else { mul(a.x, a.y); return *this; }
}

inline Natural& Natural::operator+=(const binder_arguments<Natural, Digit,
                                                           Natural_multiplies_tag>& a)
{
  muladd(a.x, a.y);
  return *this;
}

inline Natural& Natural::operator-=(const binder_arguments<Natural, Digit,
                                                           Natural_multiplies_tag>& a)
{
  mulsub(a.x, a.y);
  return *this;
}

inline binder_arguments<Natural, Digit, Natural_multiplies_tag>
 operator*(const Natural& a, const Digit& b)
// Algorithm:  c := a*b
// Input:      a in Natural, b in Digit.
// Output:     c in Natural such that c = a*b ||
{
  return binder_arguments<Natural, Digit, Natural_multiplies_tag>(a, b);
}

inline binder_arguments<Natural, Digit, Natural_multiplies_tag>
 operator*(const Digit& a, const Natural& b)
// Algorithm:  c := a*b
// Input:      a in Digit, b in Natural.
// Output:     c in Natural such that c = a*b ||
{
  return binder_arguments<Natural, Digit, Natural_multiplies_tag>(b, a);
}
◇
```

Macro referenced in 402b.

Laufzeit. Seien $x, y \in$ Natural und $a \in$ Digit, dann

$$\begin{array}{lcl} x\texttt{*}a & \sim & y\texttt{.mul(}x\texttt{,}a\texttt{)}, \\ y\texttt{+=}x\texttt{*}a & \sim & y\texttt{.muladd(}x\texttt{,}a\texttt{)}, \\ y\texttt{-=}x\texttt{*}a & \sim & y\texttt{.mulsub(}x\texttt{,}a\texttt{)}. \end{array}$$

Zuweisungsoperator

⟨ assign `operator*=` for Naturals 105 ⟩ ≡

```
inline Natural& Natural::operator*=(const Natural& a)
// Algorithm:  c := c *= a
// Input:      a,c in Natural.
// Output:     c in Natural such that c := c*a ||
{
  if (this == &a) sqr(*this);
  else mul(*this, a);
  return *this;
}
◇
```

Macro referenced in 402b.

Laufzeit. Seien $x, y \in \mathsf{Natural}$, dann $x\texttt{*=}y \sim x\texttt{.mul(}x\texttt{,}y\texttt{)}$.

2.11.7 Quadratur

Karatsuba-Multiplikation

Beim Quadrieren wird der Karatsuba-Algorithmus nun kürzer und schneller als bei der allgemeinen Multiplikation, da hier die Faktoren gleiche Längen haben.

Sei $a \in \mathsf{Natural}$, dann gilt:

$$a^2 = (a_1 2^{\beta k} + a_0)^2 = a_1^2 \cdot 2^{2\beta k} - (|a_0 - a_1|^2 - a_0^2 - a_1^2) \cdot 2^{\beta k} + a_0^2$$

mit $k := \left\lfloor \frac{\mathcal{L}(a)}{2} \right\rfloor, a \equiv a_0 \pmod{2^{\beta k}}, a_1 := \left\lfloor \frac{a}{2^{\beta k}} \right\rfloor$.

Konventionelle Quadratur

Die Anzahl der Digitmultiplikationen kann beim Quadrieren im Gegensatz zur allgemeinen Multiplikation halbiert werden, weil die Digits kommutativ sind und nach dem **polynomischen Lehrsatz**

$$a^2 = \left(\sum_{i=0}^{\mathcal{L}(a)-1} a_i 2^{\beta i} \right)^2 = \sum_{i=0}^{\mathcal{L}(a)-1} a_i^2 2^{2\beta i} + \sum_{i=0}^{\mathcal{L}(a)-2} a_i \left(\sum_{j=i+1}^{\mathcal{L}(a)-1} a_j 2^{\beta j} \right) 2^{\beta i+1}$$

für $a \in \mathsf{Natural}$ gilt.

⟨ conventional squaring algorithm 106 ⟩ ≡

```
void Natural::sqr(const Digit* pA, Digit* pB, size_t n) const
// Algorithm:  x.sqr(r, s, n)
//             Let a,b in Natural where not a.p = b.p.
// Input:      x in Natural,
//             n in size_t where n > 0,
//             r,r+n in [a.p, a.p+L(a)],
//             s,s+2*n in [b.p, b.p+L(b)].
// Output:     [s, s+2*n[ = ([r, r+n[)^2 ||
{
  CONDITION(n > 0);

  if (n == 1) {
    const Digit x = *pA;
    digitmul(x, x, *pB, pB[1]);
    return;
  } else if (n == 2) {
    digitsqr(pA[0], pA[1], pB);
    return;
  }
  const Digit* pE = pA + (n&(GAMMA-1));
  do {
    digitsqr(*pA, pA[1], pB);
    pA += 2; pB += 4;
  } while (pA != pE);
  pE -= n;
```

```
if (n&1) {
  digitmul(*pA, *pA, *pB, pB[1]);
  ++pE; ++pB; --n;
  const Digit c = *pA;
  Digit x,y;
  do {
    digitmul(*--pA, c, x, y);
    Digit* pC = --pB;
    Digit z = *pB;
    *pC = z += y;
    if (z < y) {
      z = *--pC;
      *pC = z += x+1;
      if (z <= x)
        while (++(*--pC) == 0);
    } else {
      z = *--pC;
      *pC = z += x;
      if (z < x)
        while (++(*--pC) == 0);
    }
    pC = pB; z = *pB;
    *pC = z += y;
    if (z < y) {
      z = *--pC;
      *pC = z += x+1;
      if (z <= x)
        while (++(*--pC) == 0);
    } else {
      z = *--pC;
      *pC = z += x;
      if (z < x)
        while (++(*--pC) == 0);
    }
  } while (pA != pE);
  pB += n-1; pA += n;
}
pA -= 2; --pB; n -= 2;
while (n) {
  const Digit c[2] = { *pA, pA[1] };
  Digit x[4];
  do {
    pA -= 2;
    digitmul(*pA, pA[1], c[0], c[1], x);
    Digit y = x[0] >> (BETA-1);
    pB -= 2;
    x[0] <<= 1; x[0] |= x[1] >> (BETA-1); x[1] <<= 1; x[1] |= x[2] >> (BETA-1);
    x[2] <<= 1; x[2] |= x[3] >> (BETA-1); x[3] <<= 1;
    Digit* pC = pB;
    *pB += x[3];
    if (*pB < x[3]) {
      *--pC += x[2]+1;
      if (*pC <= x[2]) {
        *--pC += x[1]+1;
        if (*pC <= x[1]) {
          *--pC += x[0]+1;
```

```
            if (*pC <= x[0]) ++y;
          } else {
            *--pC += x[0];
            if (*pC < x[0]) ++y;
          }
        } else {
          *--pC += x[1];
          if (*pC < x[1]) {
            *--pC += x[0]+1;
            if (*pC <= x[0]) ++y;
          } else {
            *--pC += x[0];
            if (*pC < x[0]) ++y;
          }
        }
      } else {
        *--pC += x[2];
        if (*pC < x[2]) {
          *--pC += x[1]+1;
          if (*pC <= x[1]) {
            *--pC += x[0]+1;
            if (*pC <= x[0]) ++y;
          } else {
            *--pC += x[0];
            if (*pC < x[0]) ++y;
          }
        } else {
          *--pC += x[1];
          if (*pC < x[1]) {
            *--pC += x[0]+1;
            if (*pC <= x[0]) ++y;
          } else {
            *--pC += x[0];
            if (*pC < x[0]) ++y;
          }
        }
      }
      *--pC += y;
      if (*pC < y)
        while (++(*--pC) == 0);
    } while (pA != pE);
    n -= 2; pB += n-2; pA += n;
  }
}
◇
```

Macro referenced in 409.

2.11.8 Quadrieralgorithmus

Bei der Realisierung der Quadratur gelten die gleichen theoretischen Vorüberlegungen wie beim Multiplizieren.

⟨ squaring of a Natural 108 ⟩ ≡

```
void Natural::sqr(const Natural& a)
// Algorithm:  b.sqr(a)
```

```
// Input:      a,b in Natural.
// Output:     b in Natural such that b = a^2 ||
{
  NATURALCONDITION(a);

  size_t n = a.length();
  ⟨ special case for the calculation of the squaring 109 ⟩
  ⟨ initialization of squaring algorithm 110a ⟩
  const Digit* pA = c.p;
  Digit* pB = p;
  do {
    ⟨ conventional squaring in Karatsuba algorithm 110b ⟩
    ⟨ nonrecursive squaring algorithm 110c ⟩
  } while (n < sC);
  delete[] h;
  delete[] st;
  normalize();

  NATURALCONDITION(*this);
}
◇
```

Macro referenced in 409.

Laufzeit. Seien $x, y \in$ Natural, dann y.sqr$(x) \in \mathcal{O}(\mathcal{L}(x)^{\log_2(3)})$.

Insbesondere berechnen wir den speziellen Fall $\mathcal{L}(a) = 1$ direkt und leiten die Argumentlängen $\mathcal{L}(a) \leq$ KARATSUBA_SQR_MARK an den konventionellen Algorithmus sqr (Seite 106) weiter, wobei $a \in$ Natural zu quadrieren ist:

⟨ special case for the calculation of the squaring 109 ⟩ ≡

```
if (n == 1) {
  Digit x,y,z = *a.p;
  a.digitmul(z, z, x, y);
  setsize(2);
  if (x) { *p = x; p[1] = y; }
  else { *p = 0; *++p = y; --size; }
  return;
}
if (n <= KARATSUBA_SQR_MARK) {
  if (a.p != p) {
    setsize(2*n);
    sqr(a.p, p, n);
  } else {
    const Natural c(a);
    setsize(2*n);
    sqr(c.p, p, n);
  }
  normalize();
  return;
}
if (n >= FFT_SQR_MARK && n <= FFT::max_size()) {
  FFT f(a); f.sqr(*this);
  return;
}
◇
```

Macro referenced in 108.

Zur Steuerung der Iteration benötigen wir die folgende Initialisierung:

⟨ initialization of squaring algorithm 110a ⟩ ≡

```
struct stack {
  const Digit* a;
  Digit* b;
  char   c,d;
};

const Natural c(a);
setsize(2*n);
size_t i = 0;
Digit* h = NOTHROW_NEW Digit[6*n];
if (!h) errmsg(2, "(sqr)");
while (n >>= 1) ++i;
stack* st = NOTHROW_NEW stack[i];
for (n = 0; n < i; ++n) st[n].d = 0;
const size_t sC = c.size;
n = sC;
--i;
◇
```

Macro referenced in 108.

⟨ conventional squaring in Karatsuba algorithm 110b ⟩ ≡

```
if (n <= KARATSUBA_SQR_MARK) {
  sqr(pA, pB, n);
  h -= 3*n;
  pA = st[++i].a; pB = st[i].b;
}◇
```

Macro referenced in 108.

Bei der iterativen Implementierung des Karatsuba-Quadrieralgorithmus gelten folgende Beziehungen:

Sei $a \in \mathsf{Natural}$, dann

$$a^2 = (c2^{2n\beta} + a_1 2^{n\beta} + a_0)^2 = c2^{4n\beta} + 2cx + x^2$$

mit $n := \lfloor \frac{\mathcal{L}(a)}{2} \rfloor$, $c \in \mathsf{Digit}$,

$$\mathsf{Natural} \ni x = (a_1 2^{\beta n} + a_0)^2 = a_1^2 \cdot 2^{2\beta n} - (|a_0 - a_1|^2 - a_0^2 - a_1^2) \cdot 2^{\beta n} + a_0^2.$$

Dadurch läßt sich der Algorithmus in vier Berechnungsteile zerlegen:

⟨ nonrecursive squaring algorithm 110c ⟩ ≡

```
switch (++st[i].d) {
  ⟨ case 1 of squaring 111a ⟩
  ⟨ case 2 of squaring 111b ⟩
  ⟨ case 3 of squaring 111c ⟩
  ⟨ case 4 of squaring 111d ⟩
}◇
```

Macro referenced in 108.

1. Berechnung von a_1^2:

⟨ case 1 of squaring 111a ⟩ ≡

```
case 1:
  st[i].c = 0;
  if (n&1) {
    ++st[i].c;
    digitmul(*pA, *pA, *pB, pB[1]);
    ++pA; pB += 2;
  }
  n >>= 1; st[i].a = pA; st[i].b = pB;
  h += 3*n; --i;
  break;◇
```

Macro referenced in 110c.

2. Berechnung von a_0^2:

⟨ case 2 of squaring 111b ⟩ ≡

```
case 2:
  pA += n; pB += 2*n;
  h += 3*n; --i;
  break;◇
```

Macro referenced in 110c.

3. Berechnung von $|a_0 - a_1|^2$:

⟨ case 3 of squaring 111c ⟩ ≡

```
case 3:
  a.abs(h, pA, pA+n, n);
  pA = h; pB = h+n;
  h += 3*n; --i;
  break;◇
```

Macro referenced in 110c.

4. Zusammensetzung:

⟨ case 4 of squaring 111d ⟩ ≡

```
case 4:
  pB += n;
  {
    int i = sub_no_dec(h+n, h+3*n, pB+n) + sub_no_dec(h+n, h+3*n, pB+3*n);
    i -= sub_no_dec(pB, pB+2*n, h+3*n);
    if (i == 1)
      for (Digit* pPos = pB; ++(*--pPos) == 0;);
    else if (i == 2) {
      Digit* pPos = pB;
      *--pPos += 2;
      if (*pPos < 2)
        while (++(*--pPos) == 0);
    }
  }
  st[i].d = 0;
  if (st[i].c) {
    const Digit d = *(pA-1);
    muladd(pA, pA+2*n, pB+n, d);
```

```
      muladd(pA, pA+2*n, pB+n, d);
      n <<= 1; ++n;
    } else n <<= 1;
    if (n < sC) { pA = st[++i].a; pB = st[i].b; h -= 3*n; }
    break;◇
```

Macro referenced in 110c.

2.11.9 Schnelle Fouriertransformation

Mit der schnellen Fouriertransformation läßt sich die asymptotische Laufzeit des Multiplikationsalgorithmus' noch sehr verbessern.
Bisher haben wir nur die Karatsuba-Multiplikation (siehe Kapitel 2.11.3, Seite 95) kennengelernt und erzielten mit ihr bereits für mittelgroße Zahlen deutliche Laufzeitgewinne.

Der Grundgedanke der Multiplikation durch die Fouriertransformation ist naheliegend:

1. Transformiere die beiden zu multiplizierenden Faktoren in ein System, in dem einfacher multipliziert werden kann.

2. Führe die Multiplikation in dem transformierten System aus.

3. Mache die Transformation für das Produkt rückgängig.

Man sieht an den obigen drei Schritten, daß der Multiplikationsalgorithmus einen Overhead im Vergleich zur konventionellen Multiplikation für die durchzuführenden Transformationen hat. Dieser Overhead ist auch der Grund, warum der in diesem Abschnitt beschriebene Multiplikationsalgorithmus nur für große Zahlen besser als die Karatsuba-Multiplikation ist.

Trade-Off-Point

Eine empirische Analyse weist die folgenden Trade-Off-Points auf:

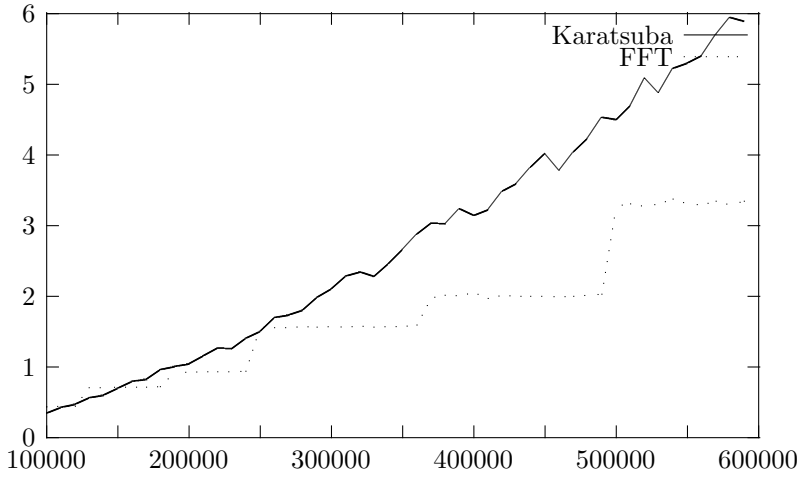
⟨ trade off points for the FFT algorithm 112 ⟩ ≡

```
#ifdef _DigitAsm_
const size_t FFT_SQR_MARK = 8125;
const size_t FFT_MUL_MARK = 5000;
#else
const size_t FFT_SQR_MARK = 11875;
const size_t FFT_MUL_MARK = 16875;
#endif
◇
```

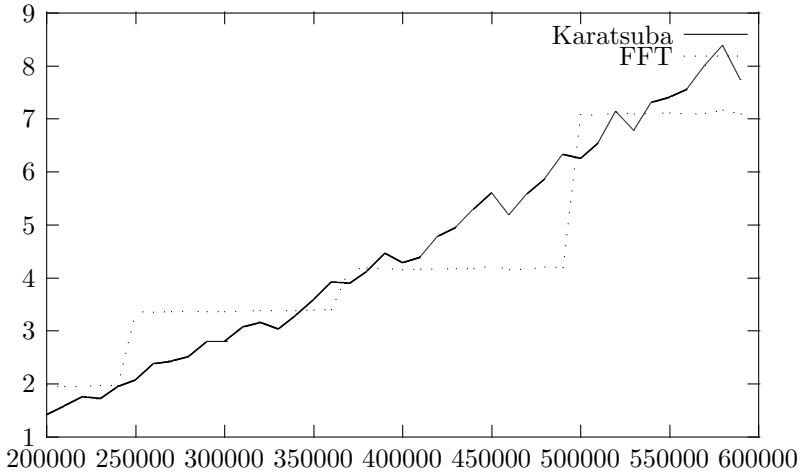
Macro referenced in 409.

Die folgenden vier Laufzeit-Diagramme sind auf einen IBM 6x86 200 MHz mit dem Microsoft Visual C++ 5.0 Compiler erzeugt worden. Dabei gibt die x-Achse die Eingabelänge an und die y-Achse gibt die ermittelte Laufzeit in Sekunden an.

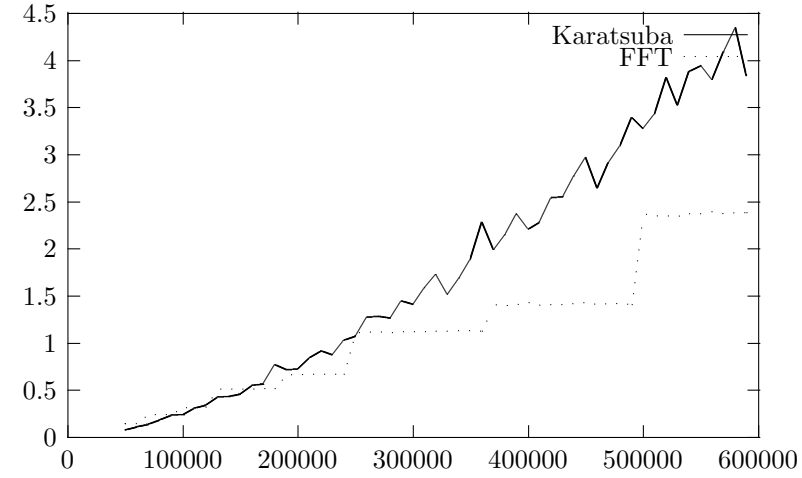
Multiplikation unter Verwendung der Assembler Digit-Multiplikation:



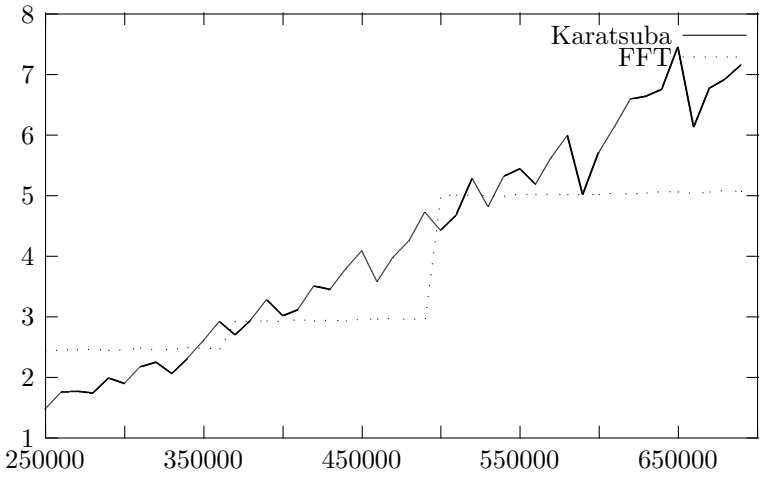
Multiplikation ohne Verwendung der Assembler Digit-Multiplikation:



Quadratur unter Verwendung der Assembler Digit-Multiplikation:



Quadratur ohne Verwendung der Assembler Digit-Multiplikation:



Diskrete Fouriertransformation

Sei $p \in \mathbb{P}$ eine Primzahl mit $p < 2^\beta$, dann übertragen wir als erstes das Zahlensystem eines Naturals eindeutig in einen Vektor über $\mathbb{F}_p := \{0, \ldots, p-1\}$:

$$\alpha : \mathsf{Natural} \hookrightarrow \mathbb{F}_p^M : a = (a_{\mathcal{L}(a)-1}, \ldots, a_0) \mapsto (b_{M-1}, \ldots, b_0, \underbrace{0, \ldots, 0}_{M}) \qquad \text{mit } M \in \mathbb{N}.$$

⟨ FFT base conversion 114 ⟩ ≡

```
void FFT::base_conversion(Natural& t, const Natural& a, const size_t shift)
{
  size_t nbase = size_t(log2(moduli[0]));
  const Digit base = (Digit(1) << nbase) - 1;

  // base conversion:
  const Digit* pA = a.p;
  size_t sA = a.size;
  const Digit* pE = pA+sA;
  size_t sT = (sA*BETA)/nbase;
  sT += shift;
  Digit* pT = t.setsize(sT)+sT;
  if (shift) {
    Digit* pE = pT-shift;
    FILL_ZERO(pE, pT);
    pT -= shift;
  }
  Digit k = *--pE;
  *--pT = k & base;
  k >>= nbase;
  size_t i = BETA-nbase;
  do {
    Digit j = *--pE;
    k |= j << i;
    *--pT = k & base;
    j >>= BETA-i; k >>= nbase;
```

```
    j <<= BETA-nbase; k |= j;
    if (i >= nbase) {
      *--pT = k & base;
      k >>= nbase; i -= nbase;
    }
    i += BETA-nbase;
  } while (pE != pA);
  if (i >= nbase)
    if (k) *--pT = k;
    else { *(pT-1) = 0; --sT; }
  else if (k) { *--pT = k; ++sT; }
  t.size = sT; t.p = pT;
}
◇
```

Macro referenced in 409.

Laufzeit. Sei $f \in$ FFT, $x, y \in$ Natural und $t \in$ `size_t`, dann f`.base_conversion(`x`, `y`, `t`)` $\in \mathcal{O}(2\mathcal{L}(x))$.

⟨ default constructor FFT 115 ⟩ ≡

```
FFT::FFT(const Natural& a, const bool init_static, FFT* b)
// Algorithm:  FFT(a, i, b)
// Input:      a in Natural, i in bool, *b in FFT
//             where 12 < L(a) <= FFT::max_size() and BETA in {32, 64} ||
 : factor(b), arg(a)
{
  NATURALCONDITION(a);
  CONDITION(a.size > 12 && (BETA == 32 || BETA == 64));
  CONDITION(moduli[0] < moduli[1] && moduli[1] < moduli[2]);
  CONDITION(a.size <= FFT::max_size());

  base_conversion(t[0], a, 0);
  size_t sT = t[0].size;
  shift = sT;

  if (init_static) {
    if (b) {
      sT += b->shift;
      const size_t j = size_t(1) << (1+log2(Digit(sT)-1));
      size_t i = (j/4) * 3;
      if (i >= sT) {
        sT = i - b->shift;
        i -= shift;
      } else {
        sT = j - b->shift;
        i = j - shift;
      }
      t[0].lmove(i);
      b->t[0].lmove(sT);
      b->shift = sT;
    } else {
      sT *= 2;
      const size_t j = size_t(1) << (1+log2(Digit(sT)-1));
      const size_t i = (j/4) * 3;
      sT = ((i >= sT)? i : j) - shift;
      t[0].lmove(sT);
    }
```

```
    shift = sT-shift;

    const size_t nn = t[0].size;
    n2 = size_t(log2(Digit((nn%3)? nn : (nn/3))));
    n1 = n2/2;
    n2 -= n1;
    n1 = 1 << n1; n2 = 1 << n2;

    omega2 = NOTHROW_NEW Digit[n2];
    if (!omega2) errmsg(2, "(FFT constructor)");
    order = NOTHROW_NEW size_t[n1];
    if (!order) errmsg(2, "(FFT constructor)");

    init_order(order, n1);

    if (n1 != n2) {
      omega = NOTHROW_NEW Digit[n1];
      if (!omega) errmsg(2, "(FFT constructor)");
    } else omega = omega2;


    CONDITION(n1 >= 4 && n2 >= n1);
  }
}
◇
```

Macro referenced in 409.

Laufzeit. Sei $x \in$ Natural, $b \in$ `bool` und $f \in$ FFT, dann `FFT(`x`,`b`,&`f`)`$\in \mathcal{O}(2\mathcal{L}(x))$.

Bemerkung. Für die Eingabevariable `a` findet ein Basiswechsel von 2^β nach $2^{\log_2(\texttt{moduli[0]})}$ statt. Dabei wird vorausgesetzt, daß `moduli[0]` der kleinste Modulus ist.
Die beiden weiteren Eingabevariablen `init_static` und `b` sind für binäre Operationen notwendig, wie zum Beispiel die Multiplikation. Um die Fouriertransformation besonders schnell verwirklichen zu können, benötigen wir recht viel temporären Speicher. Diesen deklarieren wir als statisch, um bei binären Operationen den Speicher wiederverwenden zu können. Somit erfolgt eine Initialisierung des statischen Speichers erst dann, wenn die Eingabevariable `init_static` den Wert `true` besitzt.
Wieviele statische Variablen für die Fouriertransformation zu initialisieren sind, sieht man an der folgenden FFT Klassen-Deklaration:

⟨ class FFT 116 ⟩ ≡

```
class FFT : private NumberBase {
private:
  Natural             t[3];
  FFT*                factor;
  size_t              shift;
  const Natural&      arg;
  static const Digit* moduli;
  static const Digit* primroots;
  static Digit        m;
  static size_t       n1,n2;
  static Digit*       omega;
  static Digit*       omega2;
  static size_t*      order;
```

```
  static const Digit* init_moduli();
  static const Digit* init_primroots();
  static size_t       max_size();
  static void         setmodulo(const Digit);

  void  digitmulmod(const Digit, const Digit, const Digit, Digit&) const;
  Digit pow(Digit, Digit, const Digit) const;
  Digit digitinv(Digit, const Digit) const;
  void  init_omega(const Digit);
  void  innerfft(const Digit*, Digit*, Digit*, const size_t, const Digit) const;
  void  innerfft(const Digit*, Digit*, Digit*, const size_t, const Digit,
                 const Digit) const;
  void  innerfftinv(const Digit*, Digit*, Digit*, const size_t, const Digit,
                    const Digit) const;
  void  innerfft3(Digit*, const size_t, const Digit) const;
  void  innerfftinv3(Digit*, const size_t, const Digit) const;
  void  fft(Digit*, const Digit*, const size_t) const;
  void  multiply_matrix(Digit*, const Digit, const Digit) const;
  void  fftinv(Digit*, const Digit*, const size_t) const;
  void  five_step(const Digit*, Digit*, const Digit) const;
  void  five_step(const Digit*, Digit*, const Digit, const Digit) const;
  void  chinese_remainder();

  void  square(const size_t);
  void  multiply(const size_t);
  void  result(Natural&) const;

  void  base_conversion(Natural&, const Natural&, const size_t);

  FFT(const Natural&, const bool = true, FFT* = 0);
  ~FFT();

  size_t size() const;
  void sqr(Natural&);
  void mul(Natural&);

  friend class Natural;
};
◇
```

Macro referenced in 409.

Bemerkung. Diese FFT-Klasse hat keine benutzerfreundliche Schnittstelle und wird nur für interne Zwecke eingeführt. Deshalb ist die komplette Klasse privat deklariert und in keiner Headerdatei zu finden.
Bei einer korrekten Anwendung der FFT-Klasse ist die Durchführung der Multiplikation (siehe FFT-Aufruf auf der Seite 97) als auch der Quadratur (siehe FFT-Aufruf auf der Seite 110) einfach.
Im Destruktor wird dann der angeforderte Speicher des Konstruktors wieder freigegeben:

⟨ destructor FFT 117 ⟩ ≡

```
FFT::~FFT()
{
  if (n1 != n2) delete[] omega;
  delete[] order;
  delete[] omega2;
  omega = omega2 = 0;
```

```
  order = 0;
  m = 0; n1 = n2 = 0;
}
```
◇

Macro referenced in 409.

Laufzeit. `~FFT()`$\in \mathcal{O}(12)$.

Für die Durchführung der diskreten Fouriertransformation benötigen wir eine primitive $N = (2M-1)$-te Einheitswurzel w im Körper $\mathbb{F}_p$, daß heißt $w^N = 1$ und $w^k \neq 1$ für $0 < k < N$.
Die diskrete Fouriertransformation ist dann folgendermaßen definiert:

Definition. Die *diskrete Fouriertransformation* $\mathcal{F}_N$ ist definiert durch:

$$\mathcal{F}_N : \mathbb{F}_p^N \to \mathbb{F}_p^N : (x_j)_{j=0}^{N-1} \mapsto \left(\sum_{k=0}^{N-1} x_k w^{kj} \right)_{j=0}^{N-1}$$

für eine Primzahl $p \in \mathbb{P}$, $N \in \mathbb{N}$ und eine primitive $N - te$ Einheitswurzel w im Körper $\mathbb{F}_p$.

Für das Resultat benötigen wir allerdings noch die inverse Fouriertransfomation, die sich aber folgendermaßen elementar herleiten läßt:

Satz. Nach den Voraussetzungen der vorherigen Definition gilt:

$$\mathcal{F}_N^{-1} : \mathbb{F}_p^N \to \mathbb{F}_p^N : (x_j)_{j=0}^{N-1} \mapsto \frac{1}{N} \left(\sum_{k=0}^{N-1} x_k w^{-kj} \right)_{j=0}^{N-1}. \tag{2.2}$$

Beweis.

$$\begin{aligned}
\mathcal{F}_N^{-1}\mathcal{F}_N(x_j)_{j=0}^{N-1} &= \mathcal{F}_N^{-1} \left(\sum_{k=0}^{N-1} x_k w^{kj} \right)_{j=0}^{N-1} \\
&= \left(\frac{1}{N} \sum_{l=0}^{N-1} \sum_{k=0}^{N-1} x_k w^{kl} w^{-lj} \right)_{j=0}^{N-1} \\
&= \left(\frac{1}{N} \sum_{k=0}^{N-1} x_k \sum_{l=0}^{N-1} w^{(k-j)l} \right)_{j=0}^{N-1} \\
&= \left(\frac{1}{N} \left(N x_j + \sum_{\substack{k=0 \\ k\neq j}}^{N-1} x_k \sum_{l=0}^{N-1} w^{(k-j)l} \right) \right)_{j=0}^{N-1} \\
&= \left(x_j + \frac{1}{N} \sum_{\substack{k=0 \\ k\neq j}}^{N-1} x_k \underbrace{\frac{1 - w^{N(k-j)}}{1 - w^{k-j}}}_{=0} \right)_{j=0}^{N-1} \\
&= (x_j)_{j=0}^{N-1}.
\end{aligned}$$

Analog: $\mathcal{F}_N\mathcal{F}_N^{-1}(x_j)_{j=0}^{N-1} = (x_j)_{j=0}^{N-1}$ □

Multiplikationsalgorithmus

Algorithmus.

$$\texttt{mul} : \mathsf{Natural}^2 \to \mathsf{Natural} : (a,b) \mapsto \alpha^{-1}(\mathcal{F}^{-1}(\texttt{mul}(\mathcal{F}(\alpha^{-1}(a)), \mathcal{F}(\alpha^{-1}(b))))),$$

mit

$$\texttt{mul} : \mathbb{F}_p^N \times \mathbb{F}_p^N \to \mathbb{F}_p^N : (x_j, y_j)_{j=0}^{N-1} \mapsto (x_j y_j)_{j=0}^{N-1}.$$

Beweis. Wir können ohne Einschränkung annehmen, daß die Abbildung α die identische Abbildung $(x \mapsto x)$ ist und daß $M := \mathcal{L}(a) = \mathcal{L}(b)$ ist.
Zunächst erweitern wir a und b, indem wir Nullen anhängen durch $a := a \cdot 2^{\beta M}$ und $b := b \cdot 2^{\beta M}$, beachten, daß $\mathcal{R}(a) = \mathcal{R}(b) \geq 4M$ ist und setzen dann $N := 2M (= \mathcal{L}(a) = \mathcal{L}(b))$.
Im folgenden Beweis verstehen wir unter $(a_j)_{j=0}^{N-1} := [a.p, a.p + N[$ für $a \in \mathsf{Natural}$. Demzufolge gilt wegen der Konvention für Naturals (siehe Seite 19) $a_j = 0$ für $\mathcal{L}(a) - \mathcal{R}(a) \leq j < 0$. Es gilt dann die folgende Beziehung:

$$\begin{aligned}
\texttt{mul}((a_j)_{j=0}^{N-1}, (b_j)_{j=0}^{N-1}) &= \mathcal{F}_N^{-1}\left(\texttt{mul}\left(\mathcal{F}_N(a_j)_{j=0}^{N-1}, (b_j)_{j=0}^{N-1}\right)\right) \\
&= \mathcal{F}_N^{-1}\left(\sum_{k=0}^{N-1} a_k w^{kj} \sum_{l=0}^{N-1} b_l w^{lj}\right)_{j=0}^{N-1} \\
&= \mathcal{F}_N^{-1}\left(\sum_{k=0}^{2M-1} \sum_{l=0}^{2M-1} a_k b_l w^{(k+l)j}\right)_{j=0}^{N-1} \\
&= \mathcal{F}_N^{-1}\left(\sum_{k=0}^{2M-1} \sum_{l=-k}^{2M-1} a_k b_l w^{(k+l)j}\right)_{j=0}^{N-1} \\
&= \mathcal{F}_N^{-1}\left(\sum_{k=0}^{2M-1} \sum_{l=-k}^{2M-1-k} a_k b_l w^{(k+l)j}\right)_{j=0}^{N-1} \\
&= \mathcal{F}_N^{-1}\left(\sum_{k=0}^{2M-1} w^{kj} \sum_{l=0}^{2M-1} a_{k-l} b_l\right)_{j=0}^{N-1} \\
&\underset{(2.2)}{=} \left(\sum_{k=0}^{N-1} \sum_{l=0}^{N-1} a_{k-l} b_l\right)_{j=0}^{N-1} \\
&= \left(\sum_{k=0}^{N-1} \sum_{l=0}^{k} a_{k-l} b_l\right)_{j=0}^{N-1} \quad \square
\end{aligned}$$

Die Rücktransformation α^{-1} in ein Natural erfolgt schließlich durch die folgende Funktion:

⟨ memberfunction FFT.result 119 ⟩ ≡

```
void FFT::result(Natural& b) const
// Algorithm:  a.result(b)
// Input:      a in FFT, b in Natural.
// Output:     b in Natural such that b = [a.t_2 / 2^(beta*a.shift)] ||
{
  const size_t nbase = size_t(log2(moduli[0]));
  const size_t sB = (factor)? factor->arg.size+arg.size : 2*arg.size;
  Digit* pF = b.setsize(sB);
```

```
  Digit* pB = pF+sB;
  const Digit* pE = t[2].p;
  const Digit* pT = pE+size()-shift;
  size_t i = 0;
  Digit k = 0;
  do {
    Digit j = *--pT;
    k |= j << i;
    const size_t l = BETA-i;
    if (nbase >= l) {
      *--pB = k;
      k = j >> l;
      i -= BETA;
    }
    i += nbase;
  } while (pT != pE);
  if (k) *--pB = k;
  if (pB != pF) {
    b.size -= pB-pF;
    FILL_ZERO(pF, pB);
    b.p = pB;
  }

  NATURALCONDITION(b);
}
◇
```

Macro referenced in 409.

Laufzeit. Sei $f \in \mathsf{FFT}$ und $x \in \mathsf{Natural}$, dann $f\texttt{.result(}x\texttt{)} \in \mathcal{O}(f\texttt{.size()} - f\texttt{.shift})$.

Transformationslänge

Wir verwenden die folgende Memberfunktion zur Information über die Transformationslänge:

⟨ memberfunction FFT.size 120 ⟩ ≡

```
inline size_t FFT::size() const
// Algorithm:  a := f.size()
// Input:      f in FFT.
// Output:     a in size_t such that a = L(f.t_0) ||
{
  return t[0].size;
}
◇
```

Macro referenced in 409.

Laufzeit. Sei $f \in \mathsf{FFT}$, dann $f\texttt{.size()} \in \mathcal{O}(1)$.

Schnelle Fouriertransformation

Bisher haben wir nur gezeigt, daß wir durch die Fouriertransformation die Multiplikation auf eine andere Weise berechnen können. Bei einer einfachen Durchführung der diskreten Fouriertransformation entsteht eine quadratisch anwachsende Laufzeit bei linear größer werdenden Transformationslänge.
1965 stellten J.W. Cooley und J.W. Tukey (siehe [13]) erstmals eine Reduktionsformel auf, mit der die

Fouriertransformation in $\mathcal{O}(n \log n)$ Operationen durchgeführt werden kann.

Satz. (Reduktionsformel) Sei $p \in \mathbb{P}$ eine Primzahl, $N \in \mathbb{N}$ mit $2 \mid N$, $w \in \mathbb{F}_p$ eine primitive N-te Einheitswurzel, $(x_j)_{j=0}^{\frac{N}{2}-1}, (y_j)_{j=0}^{\frac{N}{2}-1} \in \mathbb{F}_p^{N/2}$ und $(z_j)_{j=0}^{N-1} := (x_0, y_0, x_1, y_1, \ldots, x_{\frac{N}{2}-1}, y_{\frac{N}{2}-1})$. Dann gilt

$$\begin{aligned} \left(\mathcal{F}_N (z_j)_{j=0}^{N-1}\right)_k &= \left(\mathcal{F}_{\frac{N}{2}} (x_j)_{j=0}^{\frac{N}{2}-1}\right)_k + w_N^k \left(\mathcal{F}_{\frac{N}{2}} (y_j)_{j=0}^{\frac{N}{2}-1}\right)_k \qquad \text{für } 0 \le k < \tfrac{N}{2}, \\ \left(\mathcal{F}_N (z_j)_{j=0}^{N-1}\right)_{k+\frac{N}{2}} &= \left(\mathcal{F}_{\frac{N}{2}} (x_j)_{j=0}^{\frac{N}{2}-1}\right)_k - w_N^k \left(\mathcal{F}_{\frac{N}{2}} (y_j)_{j=0}^{\frac{N}{2}-1}\right)_k. \end{aligned}$$

Beweis.

$$\begin{aligned} \left(\mathcal{F}_N (z_j)_{j=0}^{N-1}\right)_k &= \sum_{l=0}^{N-1} z_l w^{kl} \\ &= \sum_{j=0}^{\frac{N}{2}-1} x_j w^{2kj} + \sum_{j=0}^{\frac{N}{2}-1} y_j w^{k(2j+1)} \\ &= \sum_{j=0}^{\frac{N}{2}-1} x_j (w^2)^{kj} + w^k \sum_{j=0}^{\frac{N}{2}-1} y_j (w^2)^{kj} \\ &= \left(\mathcal{F}_{\frac{N}{2}} (x_j)_{j=0}^{\frac{N}{2}-1}\right)_k + w_N^k \left(\mathcal{F}_{\frac{N}{2}} (y_j)_{j=0}^{\frac{N}{2}-1}\right)_k. \end{aligned}$$

Die zweite Gleichung folgt unmittelbar wegen

$$\left(\mathcal{F}_{\frac{N}{2}} (x_j)_{j=0}^{\frac{N}{2}-1}\right)_{k+\frac{N}{2}} = \left(\mathcal{F}_{\frac{N}{2}} (x_j)_{j=0}^{\frac{N}{2}-1}\right)_k$$

und $w^{\frac{N}{2}} = -1$ □

Primitive Einheitswurzeln

Wenn man komplexe Zahlen verwendet, so läßt sich jede beliebige primitive N-te Einheitswurzel durch $e^{\frac{2\pi i}{N}}$ direkt angeben. Die Verwendung dieser komplexen Einheitswurzeln sind besonders in der Numerik beliebt und werden in den meisten Implementationen der Einfachheit wegen verwendet. Bei ihrer Verwendung (repräsentiert durch `double`[2]) entstehen jedoch erhebliche Rundungsprobleme, die die Portabilität erschweren und einer exakten Berechnung nicht würdig sind.
Primitive Einheitswurzeln gibt es auch in der modularen Arithmetik, sie lassen sich allerdings nicht mehr explizit angeben. Desweiteren kann man auch nicht zu jedem Modulus eine beliebige primitive N-te Einheitswurzel finden. Ein hervoragender Modulus sind Primzahlen, denn in diesem Fall gelten für die modulare Arithmetik die Körpereigenschaften, die wir zum Beispiel von den rationalen und reellen Zahlen gewohnt sind. Da die modulare Arithmetik auf die Zahlentheorie basiert[4], wird diese Fouriertransformation auch Zahlentheoretische Transformation (Number-theoretical transforms oder kurz NTT) genannt.
Zwar erreichen wir durch die modulare Arithmetik ein exaktes Rechnen, jedoch muß man auch hier darauf achten, daß kein Überlauf erzeugt wird, denn die Summe

$$c_k := \sum_{l=0}^{N-1} a_{k-l} b_l \qquad \text{(siehe Beweis auf Seite 119)}$$

[4] Die wir später noch im Kapitel 5.4 (Seite 277ff) genauer einführen werden

muß in den ganzen Zahlen berechnet werden. Damit die modulare Berechnung korrekte Ergebnis liefert, muß der Modulus M so groß gewählt werden, daß für die Koeffizienten z_k stets $z_k < M$ gilt. Diese Bedingung ist dann erfüllt, wenn $N \cdot \gamma^2 < M$ ist. Das bedeutet aber wiederum, daß M aus mindestens drei Digits bestehen muß, was erhebliche Laufzeiteinbuße mit sich bringt. Es gibt nun zwei Möglichkeiten, dieses Problem zu lösen:
Die einfachere Lösung besteht darin, die Basis zu verkleinern, daß heißt wir vergrößern das Eingabe-Natural um die vierfache Länge und in jedem Digit nur ein Viertel des ursprünglichen Digits ab. Dadurch wird aber die Transformationslänge vervierfacht, was mindestens eine Verachtfachung der Laufzeit und eine Vervierfachung des Speicherbedarfes bedeutet.
Eine deutlich bessere Lösung bezüglich Laufzeit und Speicherbedarf ist die Verwendung dreier Moduli mit anschließender Anwendung des *Chinesischen Restsatzes* (siehe Kapitel 5.4.3, Seite 278):

⟨ chinese remainder theorem for FFT-class 122 ⟩ ≡

```
void FFT::chinese_remainder()
// Algorithm:  f.chinese_remainder()
// Input:      f in FFT where f.t[0] = f.t[0] mod f.moduli[0],
//                            f.t[1] = f.t[1] mod f.moduli[1],
//                            f.t[2] = f.t[2] mod f.moduli[2].
// Output:     f in FFT such that f.t[2] := (f.moduli[0]*f.moduli[1]*i*f.t[2]
//                                         + f.moduli[0]*f.moduli[2]*j*f.t[1]
//                                         + f.moduli[1]*f.moduli[2]*k*f.t[0])
//                                         mod (f.moduli[0]*f.moduli[1]*f.moduli[2])
//             where i := (f.moduli[0]*f.moduli[1])^(-1) mod f.moduli[2],
//                   j := (f.moduli[0]*f.moduli[2])^(-1) mod f.moduli[1],
//                   k := (f.moduli[1]*f.moduli[2])^(-1) mod f.moduli[0] ||
{
  Digit y;
  Digit m01[2],m02[2],m12[2];
  Digit s[3],x[3],c[3],m[3];
  c[0] = c[1] = c[2] = 0;

  const size_t sT     = t[2].size;
  Digit* pT           = t[2].p+sT;
  const Digit* pA     = t[1].p+sT;
  const Digit* pB     = t[0].p+sT;
  const size_t nbase  = size_t(log2(moduli[0]));
  const Digit base    = (Digit(1) << nbase) - 1;

  digitmul(moduli[0], moduli[1], m01[0], m01[1]);
  digitmul(moduli[0], moduli[2], m02[0], m02[1]);
  digitmul(moduli[1], moduli[2], m12[0], m12[1]);
  digitmod(m01[0], m01[1], moduli[2], y);
  const Digit i = digitinv(y, moduli[2]);
  digitmod(m02[0], m02[1], moduli[1], y);
  const Digit j = digitinv(y, moduli[1]);
  digitmod(m12[0], m12[1], moduli[0], y);
  const Digit k = digitinv(y, moduli[0]);

  digitmul(moduli[2], m01[0], m01[1], m);

  do {
    digitmulmod(i, *--pT, moduli[2], y);

    // s = y * moduli[0] * moduli[1]
    digitmul(y, m01[0], m01[1], s);
```

```
digitmulmod(j, *--pA, moduli[1], y);

// s += y * moduli[0] * moduli[2]
digitmul(y, m02[0], m02[1], x);
bool carry;
s[2] += x[2];
if (s[2] < x[2]) {
  s[1] += x[1]+1;
  if (s[1] <= x[1]) {
    s[0] += x[0]+1;
    carry = (s[0] <= x[0]);
  } else {
    s[0] += x[0];
    carry = (s[0] < x[0]);
  }
} else {
  s[1] += x[1];
  if (s[1] < x[1]) {
    s[0] += x[0]+1;
    carry = (s[0] <= x[0]);
  } else {
    s[0] += x[0];
    carry = (s[0] < x[0]);
  }
}
// if (s >= M) s -= M
if (carry || s[0] > m[0]
    || s[0] == m[0] && (s[1] > m[1] || s[1] == m[1] && s[2] >= m[2])) {
  if (s[2] >= m[2]) {
    s[2] -= m[2];
    if (s[1] >= m[1]) {
      s[1] -= m[1];
      s[0] -= m[0];
    } else {
      s[1] -= m[1];
      s[0] -= m[0]+1;
    }
  } else {
    s[2] -= m[2];
    const Digit c = ~m[1];
    s[1] += c;
    if (s[1] >= c) s[0] -= m[0]+1;
    else s[0] -= m[0];
  }
}

digitmulmod(k, *--pB, moduli[0], y);

// s += y * moduli[1] * moduli[2]
digitmul(y, m12[0], m12[1], x);
s[2] += x[2];
if (s[2] < x[2]) {
  s[1] += x[1]+1;
  if (s[1] <= x[1]) {
    s[0] += x[0]+1;
```

```
    carry = (s[0] <= x[0]);
  } else {
    s[0] += x[0];
    carry = (s[0] < x[0]);
  }
} else {
  s[1] += x[1];
  if (s[1] < x[1]) {
    s[0] += x[0]+1;
    carry = (s[0] <= x[0]);
  } else {
    s[0] += x[0];
    carry = (s[0] < x[0]);
  }
}
// if (s >= M) s -= M
if (carry || s[0] > m[0]
    || s[0] == m[0] && (s[1] > m[1] || s[1] == m[1] && s[2] >= m[2])) {
  if (s[2] >= m[2]) {
    s[2] -= m[2];
    if (s[1] >= m[1]) {
      s[1] -= m[1];
      s[0] -= m[0];
    } else {
      s[1] -= m[1];
      s[0] -= m[0]+1;
    }
  } else {
    s[2] -= m[2];
    const Digit c = ~m[1];
    s[1] += c;
    if (s[1] >= c) s[0] -= m[0]+1;
    else s[0] -= m[0];
  }
}

// c += s
c[2] += s[2];
if (s[2] > c[2]) {
  c[1] += s[1]+1;
  if (s[1] >= c[1]) c[0] += s[0]+1;
  else c[0] += s[0];
} else {
  c[1] += s[1];
  if (s[1] > c[1]) c[0] += s[0]+1;
  else c[0] += s[0];
}

// *pT = c & base
*pT = c[2] & base;

// c >>= nbase
c[2] >>= nbase;
c[2] |= c[1] << (BETA-nbase);
c[1] >>= nbase;
c[1] |= c[0] << (BETA-nbase);
```

```
    c[0] >>= nbase;
  } while (pT != t[2].p);

  const Digit d = c[2];
  if (d) {
    *--t[2].p = d;
    ++(t[2].size);
  } else ++shift;
}
```
◇

Macro referenced in 409.

Laufzeit. Sei $f \in \mathsf{FFT}$, dann f`.chinese_remainder()`$\in \mathcal{O}(9\mathcal{L}(f.t[0]))$.

Die Moduli müssen die Form $kN+1$ besitzen, wobei die Transformationslänge N eine Zweierpotenz sein muß. Dadurch enthält die Einheitengruppe des Modulus eine zyklische Untergruppe der Ordnung N, und es ist eine 2^m-te Einheitswurzel ist für $2^m \leq M$ durch $w^{(M-1)/2^m}$ gegeben, wobei w eine primitive M-te Einheitswurzel ist.

Mit der folgenden Memberfunktion initialisieren wir die Moduli der FFT-Klasse:

⟨ memberfunction FFT.init_moduli 125 ⟩ ≡

```
const Digit* FFT::init_moduli()
// Algorithm:  m := f.init_moduli()
// Input:      f in FFT where BETA in {32, 64}.
// Output:     m in Digit^3 such that m[0],m[1],m[2] prim
//             and m[0] < m[1] < m[2] ||
{
  if (BETA == 32) {
#ifdef _DigitAsm_
    static Digit M[3] = { Digit(2717908993U), Digit(2868903937U), Digit(3221225473U) };
#else
    static Digit M[3] = { (Digit(1) << (BETA-1)) - (Digit(1) << (BETA-4))
                                                 - (Digit(1) << (BETA-6)) + 1,
                          (Digit(1) << (BETA-1)) - (Digit(1) << (BETA-5)) + 1,
                          (Digit(1) << (BETA-1)) - (Digit(1) << (BETA-7)) + 1 };
#endif
    return M;
  } else if (BETA == 64) {
    static Digit M[3] = { Digit(1 - (Digit(1) << (BETA-24))),
                          Digit(1 - (Digit(1) << (BETA-30))),
                          Digit(1 - (Digit(1) << (BETA-32))) };
    return M;
  } else {
    static Digit M[3] = { 0, 0, 0 };
    return M;
  }
}
```
◇

Macro referenced in 409.

Laufzeit. Sei $f \in \mathsf{FFT}$, dann f`.init_moduli()`$\in \mathcal{O}(4)$.

Bemerkung. Wir unterstützen lediglich die beiden Fälle $\beta = 32$ und $\beta = 64$.

Bemerkung. Falls $\beta = 32$ ist, dann existieren nur zwei Moduli ($2^{32} - 2^{30} + 1$ und $2^{32} - 2^{20} + 1$) der Form $kN + 1$, die den Digit-Bereich voll ausnutzen.

Weiterhin müssen wir auch noch mit der folgenden Funktion die primitiven Einheitswurzeln der Moduli initialisieren:

⟨ memberfunction FFT.init_primroots 126a ⟩ ≡

```
const Digit* FFT::init_primroots()
// Algorithm:  p := f.init_primroots()
// Input:      f in FFT where BETA in {32, 64}.
// Output:     p in Digit^3 such that p[j]^(f.n1*f.n2) = 1 (mod f.moduli[j])
//             and not p[j]^k = 1 (mod f.moduli[j]) for 0 < k < f.n1*f.n2
//             and 0 <= j <= 2 ||
{
  if (BETA == 32) {
#ifdef _DigitAsm_
    static Digit p[3] = { 5, 35, 5 };
#else
    static Digit p[3] = { 13, 31, 5 };
#endif
    return p;
  } else if (BETA == 64) {
    static Digit p[3] = { 19, 10, 7 };
    return p;
  } else {
    static Digit p[3] = { 0, 0, 0 };
    return p;
  }
}
```

◇

Macro referenced in 409.

Laufzeit. Sei $f \in \mathsf{FFT}$, dann f`.init_primroots()`$\in \mathcal{O}(4)$.

Bei den vorherigen Überlegungen ist uns aufgefallen, daß die Zahlentheoretische Fouriertransformation nur bis zur Eingabelänge N durchgeführt werden kann. Diese obere Schranke wird durch die folgende Memberfunktion ermittelt:

⟨ memberfunction FFT.max_size 126b ⟩ ≡

```
size_t FFT::max_size()
// Algorithm:  d := f.max_size()
// Input:      f in FFT.
// Output:     d in size_t such that  k*d+1 = f.moduli[0] for k in N, k > 0 ||
{
  CONDITION(moduli[0] < moduli[1] && moduli[1] < moduli[2]);

  static size_t d = 0;
  if (d) return d;

  const Digit a = log2(moduli[0]);
  const Digit b = log2(-(moduli[2] << (BETA-1-a))) + a+2-BETA;
```

```
  const size_t c = 3*(1 << (b-log2(BETA)-1));
  return d = size_t(c*a);
}
◇
```

Macro referenced in 409.

Laufzeit. Sei $f \in \mathsf{FFT}$, dann f`.max_size()`$\sim$ `log2` $: \mathsf{Digit} \to \mathsf{Digit}$.

Den Modulus setzen wir dann mit der folgenden Funktion:

⟨ memberfunction FFT.setmodulo 127a ⟩ ≡

```
inline void FFT::setmodulo(const Digit a)
// Algorithm:  b.setmodulo(a)
// Input:      b in FFT, a in Digit where a in b.moduli.
// Output:     b in FFT such that b.m = a ||
{
  CONDITION(a == moduli[0] || a == moduli[1] || a == moduli[2]);

  m = a;
}
◇
```

Macro referenced in 409.

Laufzeit. Sei $f \in \mathsf{FFT}$ und $a \in \mathsf{Digit}$, dann f`.setmoduluo(`a`)`$\in \mathcal{O}(1)$.

Modulare Multiplikation

Die modulare Multiplikation ist die mit Abstand zeitintensivste Funktion bei unserer Fouriertransformation. Deswegen müssen wir auch besonders an dieser Stelle nach Optimierungsmöglichkeiten suchen.

Die einfachste Lösung ist natürlich eine Doppel-Digit-Multiplikation und -Division durchzuführen:

⟨ memberfunction FFT.digitmulmod 127b ⟩ ≡

```
inline void FFT::digitmulmod(const Digit a, const Digit b, const Digit M, Digit& c) const
// Algorithm:  f.digitmulmod(a, b, m, c)
// Input:      f in FFT, a,b,m in Digit where m > 0, a,b <= m, BETA in {32, 64},
//             m in f.moduli.
// Output:     c in Digit such that c = a*b - m*[(a*b)/m] ||
{
  CONDITION(BETA == 32 || BETA == 64);

  Digit x,y;
  digitmul(a, b, x, y);

#if defined(_DigitAsm_)
  digitdiv(x, y, M, x, c);
#elif defined(_Unknown_Apogee_Bug_)
  digitmod(x, y, M, c);
#else
  if (BETA == 32) {
    const Digit M1 = (Digit(1) << (BETA-1)) - (Digit(1) << (BETA-4))
                                            - (Digit(1) << (BETA-6)) + 1;
    const Digit M2 = (Digit(1) << (BETA-1)) - (Digit(1) << (BETA-5)) + 1;
```

```
    const Digit M3 = (Digit(1) << (BETA-1)) - (Digit(1) << (BETA-7)) + 1;

    CONDITION(M == M1 || M == M2 || M == M3);

    ⟨ modular multiplication with special 32-Bit moduli 130, ... ⟩
  } else {  // BETA == 64
    const Digit M1 = Digit(1 - (Digit(1) << (BETA-24)));
    const Digit M2 = Digit(1 - (Digit(1) << (BETA-30)));
    const Digit M3 = Digit(1 - (Digit(1) << (BETA/2)));

    CONDITION(M == M1 || M == M2 || M == M3);

    ⟨ modular multiplication with special 64-Bit moduli 128, ... ⟩
  }
#endif
}
◇
```

Macro referenced in 409.

Laufzeit. Sei $f \in \mathsf{FFT}$ und $a_0, a_1, b, d \in \mathsf{Digit}$, dann

$$f\texttt{.digitmulmod(}a_0\texttt{,}a_1\texttt{,}b\texttt{,}d\texttt{)} \sim \texttt{ digitmul} : \mathsf{Digit}^2 \to \mathsf{Digit}^2.$$

Wir wissen jedoch speziell für die Fouriertransformation, daß unser Modulus eine Primzahl der Form $kN + 1$ (siehe Seite 125) ist. Ferner treten auch nur höchsten drei verschiedene Primzahlen auf.
Im folgenden werden wir sehen, daß diese drei Primzahlen so geschickt gewählt sind, daß wir die Doppel-Digit-Division durch wenige Verschiebungen, Subtraktionen und Additionen realisieren können, was auf manchen RISC-Architekturen (beispielsweise DEC-Alpha) deutlich schneller als die durch die Hardware unterstützte Division ist.

Nehmen wir nun an, daß wir eine Digit-Multiplikation auf einer 64-Bit Architektur durchgeführt haben, so ist das 128-Bit Ergebnis $a2^{64} + b$ mit $a, b \in \mathsf{Digit}$.

Betrachten wir nun den Modulus $2^{64} - 2^{40} + 1$, dann gilt

$$\begin{aligned} a2^{64} + b &= (2^{64} - 2^{40} + 1)a + (2^{40} - 1)a + b \\ &\equiv (2^{40} - 1)a + b \pmod{2^{64} - 2^{40} + 1}. \end{aligned}$$

Durch die obige Gleichung reduzieren wir das 128-Bit Ergebins $a2^{64}+b$ durch Verschiebung, Subtraktion und Addition auf etwa 104 Bits. Also müssen wir die obige Gleichung höchstens drei Mal ausführen, um schließlich das Ergebnis auf einen Digit zu reduzieren:

⟨ modular multiplication with special 64-Bit moduli 128 ⟩ ≡

```
if (M == M1) {
  Digit s = x << (BETA-24);
  Digit u = y - x;
  x >>= 24; if (y < u) --x;
  y = u+s; if (y < s) ++x;

  s = x << (BETA-24); u = y - x;
  x >>= 24; if (y < u) --x;
  y = u+s; if (y < s) ++x;
```

```
  s = x << (BETA-24); u = y - x;
  x >>= 24; if (y < u) --x;
  y = u+s; if (y < s) ++x;
  if (x || y >= M1) y -= M1;
  c = y;
◇
```

Macro defined by 128, 129ab.
Macro referenced in 127b.

Analog betrachten wir den Modulus $2^{64} - 2^{34} + 1$:

$$\begin{aligned} a2^{64} + b &= (2^{64} - 2^{34} + 1)a + (2^{34} - 1)a + b \\ &\equiv (2^{34} - 1)a + b \pmod{2^{64} - 2^{34} + 1}. \end{aligned}$$

Durch die Gleichung reduzieren wir nun das 128-Bit Ergebins $a2^{64} + b$ auf etwa 98 Bits. Also müssen wir die obige Gleichung dreimal ausführen, um schließlich das Ergebnis auf einen Digit zu reduzieren:

⟨ modular multiplication with special 64-Bit moduli 129a ⟩ ≡

```
} else if (M == M2) {
  Digit s = x << (BETA-30);
  Digit u = y - x;
  x >>= 30; if (y < u) --x;
  y = u+s; if (y < s) ++x;

  s = x << (BETA-30); u = y - x;
  x >>= 30; if (y < u) --x;
  y = u+s; if (y < s) ++x;

  s = x << (BETA-30); u = y - x;
  x >>= 30; if (y < u) --x;
  y = u+s; if (y < s) ++x;
  if (x || y >= M2) y -= M2;
  c = y;
◇
```

Macro defined by 128, 129ab.
Macro referenced in 127b.

Für den letzten Modulus $2^{64} - 2^{32} + 1$ gilt ebenso

$$\begin{aligned} a2^{64} + b &= (2^{64} - 2^{32} + 1)a + (2^{32} - 1)a + b \\ &\equiv (2^{32} - 1)a + b \pmod{2^{64} - 2^{32} + 1}. \end{aligned}$$

Dieses Mal reduzieren wir jedoch das 128-Bit Ergebnis auf etwa 96 Bits, wodurch wir nun die obige Gleichung nur noch zweimal ausführen müssen:

⟨ modular multiplication with special 64-Bit moduli 129b ⟩ ≡

```
} else {  // M == M3
  Digit s = x << BETA/2;
  Digit u = y - x;
  x >>= BETA/2; if (y < u) --x;
  y = u+s; if (y < s) ++x;

  s = x << BETA/2; u = y - x;
  x >>= BETA/2; if (y < u) --x;
  y = u+s; if (y < s) ++x;
  if (x || y >= M3) y -= M3;
  c = y;
}
◇
```

Macro defined by 128, 129ab.
Macro referenced in 127b.

Bei einer 32-Bit Arithmetik können die Moduli nicht mehr ganz so effizient die Division durchführen.

Betrachten wir den ersten und ineffizientesten Modulus $2^{31} - 2^{28} - 2^{26} + 1$, dann gilt

$$\begin{aligned} a2^{32} + b &= 2(2^{31} - 2^{28} - 2^{26} + 1)a + (2^{29} + 2^{27} - 2)a + b \\ &\equiv (2^{29} + 2^{27} - 2)a + b \pmod{2^{31} - 2^{28} - 2^{26} + 1}. \end{aligned}$$

Durch die obige Gleichung reduzieren wir das 62-Bit Ergebins $a2^{32} + b$ um etwa 3 Bits.

⟨ modular multiplication with special 32-Bit moduli 130 ⟩ ≡

```
if (M == M1) {
  do {
    const Digit r = x << (BETA-5);
    const Digit s = x << (BETA-3);
    const Digit u = y - (x << 1);
    x >>= BETA-29; x += x >> 2;
    if (y < u) --x;
    y = u+s; if (y < s) ++x;
    y += r; if (y < r) ++x;
  } while (x);
  if (y >= M1) {
    y -= M1;
    if (y >= M1) {
      y -= M1;
      if (y >= M1) y -= M1;
    }
  }
  c = y;
◇
```

Macro defined by 130, 131ab.
Macro referenced in 127b.

Beim zweiten Modulus $2^{31} - 2^{27} + 1$ reduzieren wir das Ergebnis um etwa 4 Bits:

$$\begin{aligned} a2^{32} + b &= 2(2^{31} - 2^{27} + 1)a + (2^{28} - 2)a + b \\ &\equiv (2^{28} - 2)a + b \pmod{2^{31} - 2^{27} + 1}. \end{aligned}$$

⟨ modular multiplication with special 32-Bit moduli 131a ⟩ ≡

```
} else if (M == M2) {
  do {
    const Digit s = x << (BETA-4);
    const Digit u = y - (x << 1);
    x >>= BETA-28; if (y < u) --x;
    y = u+s; if (y < s) ++x;
  } while (x);
  if (y >= M2) {
    y -= M2;
    if (y >= M2) {
      y -= M2;
      if (y >= M2) y -= M2;
    }
  }
  c = y;
◇
```

Macro defined by 130, 131ab.
Macro referenced in 127b.

Und schließlich beim dritten Modulus $2^{31} - 2^{25} + 1$ reduzieren wir das Ergebnis um etwa 6 Bits:

$$\begin{aligned} a2^{32} + b &= 2(2^{31} - 2^{25} + 1)a + (2^{26} - 2)a + b \\ &\equiv (2^{26} - 2)a + b \pmod{2^{31} - 2^{25} + 1}. \end{aligned}$$

⟨ modular multiplication with special 32-Bit moduli 131b ⟩ ≡

```
} else { // M = M3
  do {
    const Digit s = x << (BETA-6);
    const Digit u = y - (x << 1);
    x >>= BETA-26; if (y < u) --x;
    y = u+s; if (y < s) ++x;
  } while (x);
  if (y >= M3) {
    y -= M3;
    if (y >= M3) {
      y -= M3;
      if (y >= M3) y -= M3;
    }
  }
  c = y;
}
◇
```

Macro defined by 130, 131ab.
Macro referenced in 127b.

Modulares Potenzieren

Das modulare Potenzieren wird bis auf die modulare Multiplikation analog zur Potenzierung eines Naturals (siehe Kapitel 2.13, Seite 148) durchgeführt.

⟨ memberfunction FFT.pow 131c ⟩ ≡

```
Digit FFT::pow(Digit a, Digit b, const Digit m) const
// Algorithm:  c := f.pow(a, b, m)
// Input:      f in FFT, a,b,m in Digit.
// Output:     c in Digit such that c = a^b (mod m) ||
{
  if (b == 1) return a;
  else if (b > 1) {
    Digit c = 1;
    do {
      if (b&1) digitmulmod(a, c, m, c);
      b >>= 1;
      digitmulmod(a, a, m, a);
    } while (b > 1);
    digitmulmod(a, c, m, c);
    return c;
  }
  return 1;
}
```
◇

Macro referenced in 409.

Laufzeit. Sei $f \in$ FFT und $a, b, c \in$ Digit, dann f.pow(a,b,c)$\in \log_2(b)\mathcal{O}(\texttt{digitmulmod}\colon \text{Digit}^3 \to \text{Digit})$.

Aus Effizienzgründen werden wir das modulare Potenzieren nur bei der Initialisierung verwenden, indem wir alle notwendigen Potenzen in den beiden Feldern `omega` und `omega2` der FFT-Klasse durch die folgende Funktion ablegen:

⟨ memberfunction FFT.init_omega 132 ⟩ ≡

```
void FFT::init_omega(const Digit c)
// Algorithm:  b.init_omega(a)
// Input:      b in FFT, a in Digit ||
//
// Note:       initialize all necessary powers of c in b.omega and b.omega2.
{
  size_t sT = t[0].size;
  if (sT%3 == 0) sT /= 3;
  const size_t n = n2;
  const Digit  M = m;
  const Digit  s = pow(c, sT/n, M);
  if (n == n1) {
    Digit* a = omega;
    const Digit* e = a+n-1;
    Digit x = s;
    *a = 1; *++a = s;
    do {
      digitmulmod(x, s, M, x);
      *++a = x;
    } while (a != e);
  } else {
    Digit* a = omega;
    Digit* b = omega2;
    const Digit* e = b+n-1;
    Digit x = s;
    *a = *b = 1; *++b = s;
    do {
```

```
        digitmulmod(x, s, M, x);
        ***a = ***b = x;
        digitmulmod(x, s, M, x);
        ***b = x;
      } while (b != e);
    }
  }
  ◇
```

Macro referenced in 409.

Laufzeit. Sei $f \in \mathsf{FFT}$ und $a \in \mathsf{Digit}$, dann f`.init_omega(`a`)`$\in f$`.n2`$\mathcal{O}$(`digitmulmod`: $\mathsf{Digit}^3 \to \mathsf{Digit}$).

Modulare Inversion

Für eine deteillierte Beschreibung der Berechnung des modularen Inversen verweise ich auf das Kapitel 5.4.2 auf der Seite 278.

⟨ memberfunction FFT.digitinv 133 ⟩ ≡

```
Digit FFT::digitinv(Digit a, const Digit m) const
// Algorithm:  c := f.digitinv(a, m)
// Input:      f in FFT, a,m in Digit.
// Output:     c in Digit such that c = a^{-1} (mod m) ||
{
  Digit b = m;
  Digit q = 1;
  Digit p = 0;

  while (true) {
    Digit s = a/b;
    a -= s*b;
    if (a == 0) return m - p;
    q += s*p;
    s = b/a;
    b -= s*a;
    if (b == 0) return q;
    p += s*q;
  }
}
◇
```

Macro referenced in 409.

Laufzeit. Sei $f \in \mathsf{FFT}$ und $a, b \in \mathsf{Digit}$, dann f`.digitinv(`a`,`b`)`$\in \log_2(a)\mathcal{O}(14)$.

Fünf-Schritte-Algorithmus

1990 führte David H. Baily in [4] den folgenden Sechs-Schritte-Algorithmus ein, der sich besonders für ein hierarchisches Speichermodell eignet:

Sei $N, n_1, n_2 \in \mathbb{N}$ mit $n_1 \cdot n_2 = N$, $0 \leq j < N, p \in \mathbb{P}$ eine Primzahl und $w \in \mathbb{F}_p$ eine primitive N-te Einheitswurzel. Zusätzlich betrachten wir den zu transformierenden Vektor $(x_j)_{j=0}^{N-1} \in \mathbb{F}_p^N$ als eine zeilenbasierte $n_1 \times n_2$-Matrix, das heißt daß die (k_1, k_2)-Komponente der $n_1 \times n_2$-Matrix die $k_1 n_2 + k_2$-Komponente des Vektor $(x_j)_{j=0}^{N-1}$ ist.
Dann lauten die sechs Schritte:

1. Transponiere die Eingabedaten, als eine $n_1 \times n_2$-Matrix betrachtet, in eine $n_2 \times n_1$-Matrix:
$$x^{(1)}_{j_2 n_1 + j_1} := x_{j_1 n_2 + j_2}.$$
2. Führe zeilenweise n_1 voneinander unabhängige Fouriertransformationen der Länge n_2 an der resultierenden $n_2 \times n_1$-Matrix durch:
$$x^{(2)}_{j_2 n_1 + j_1} := \sum_{k_1=0}^{n_1-1} x^{(1)}_{j_2 n_1 + k_1} w^{k_1 j_1 n_2}.$$
3. Multipliziere komponentenweise die Einträge (j_1, j_2) der resultierenden $n_2 \times n_1$-Matrix mit $w^{j_1 j_2}$:
$$x^{(3)}_{j_2 n_1 + j_1} := w^{j_1 j_2} x^{(2)}_{j_2 n_1 + j_1}.$$
4. Transponiere die resultierende $n_2 \times n_1$-Matrix in eine $n_1 \times n_2$ Matrix:
$$x^{(4)}_{j_1 n_2 + j_2} := x^{(3)}_{j_2 n_1 + j_1}.$$
5. Führe zeilenweise n_2 voneinander unabhängige Fouriertransformationen der Länge n_1 an der resultierenden $n_1 \times n_2$-Matrix durch:
$$x^{(5)}_{j_1 n_2 + j_2} := \sum_{k_2=0}^{n_2-1} x^{(4)}_{j_1 n_2 + k_2} w^{k_2 j_2 n_1}.$$
6. Transponiere die resultierende $n_1 \times n_2$-Matrix in eine $n_2 \times n_1$ Matrix:
$$x^{(6)}_{j_2 n_1 + j_1} := x^{(5)}_{j_1 n_2 + j_2}.$$

Behauptung.
$$(x^{(6)}_j)_{j=0}^{N-1} = \mathcal{F}_N (x_j)_{j=0}^{N-1}.$$

Beweis. Sei $0 \leq j < N$ und $j_1, j_2 \in \mathbb{N}$ mit $j_2 n_1 + j_1 = j$, dann gilt

$$\begin{aligned}
x^{(6)}_{j_2 n_1 + j_1} &= x^{(5)}_{j_1 n_2 + j_2} \\
&= \sum_{k_2=0}^{n_2-1} x^{(4)}_{j_1 n_2 + k_2} w^{k_2 j_2 n_1} \\
&= \sum_{k_2=0}^{n_2-1} x^{(3)}_{k_2 n_1 + j_1} w^{k_2 j_2 n_1} \\
&= \sum_{k_2=0}^{n_2-1} x^{(2)}_{k_2 n_1 + j_1} w^{k_2 j_1} w^{k_2 j_2 n_1} \\
&= \sum_{k_2=0}^{n_2-1} \sum_{k_1=0}^{n_1-1} x^{(1)}_{k_2 n_1 + k_1} w^{k_1 j_1 n_2} w^{k_2 j_1 + k_2 j_2 n_1} \\
&\underset{(w^{n_1 n_2}=1)}{=} \sum_{k_1=0}^{n_1-1} \sum_{k_2=0}^{n_2-1} x_{k_1 n_2 + k_2} w^{(k_1 n_2 + k_2)(j_2 n_1 + j_1)} \\
&= \sum_{k=0}^{N-1} x_k w^{kj} \qquad \square
\end{aligned}$$

Wir werden im folgenden den sechsten Schritt des obigen Algorithmus wegfallen lassen, weil wir nach der Fouriertransformation nur eine komponentenweise Quadratur oder Multiplikation durchführen und wieder die inverse Fouriertransformation anschließen, bei der wir dann dafür den ersten Schritt auslassen.

Bemerkung. Bei der Implementation des Fünf-Schritte-Algorithmus habe ich das "High Performance Arithmetic Package" `apfloat` von Mikko Tommila ([51]) als Grundlage genommen und gemäß meinen Richtlinien überarbeitet.

⟨ five step FFT algorithm 135a ⟩ ≡

```
void FFT::five_step(const Digit* pE, Digit* pA, const Digit w) const
// Algorithm:  c.five_step(e, a, w)
// Input:      c in FFT, w in Digit, a in Digit^(c.n1*c.n2),
//             e in [a_0, a_(c.n1*c.n2)] where e = a_(c.n1*c.n2).
// Output:     a in Digit^(c.n1*c.n2) such that a := F_(c.n1*c.n2)(a) ||
{
  const size_t nn = n1*n2;
  do {
    transpose(pA, n1, n2);          // first step

    const Digit* pF = pA+nn;        // second step
    do {
      fft(pA, omega, n1);
      brevorder(n1, pA, order);
      pA += n1;
    } while (pA != pF);
    pA -= nn;

    transpose(pA, n2, n1);          // third step

    multiply_matrix(pA, w, 1);      // forth step

    pF = pA+nn;                     // fifth step
    do {
      fft(pA, omega2, n2);
      pA += n2;
    } while (pA != pF);
  } while (pA != pE);
}
```

◇

Macro referenced in 409.

⟨ five step FFT algorithm for inversion 135b ⟩ ≡

```
void FFT::five_step(const Digit* pE, Digit* pA, const Digit w, const Digit iw) const
// Algorithm:  c.five_step(e, a, w, v)
// Input:      c in FFT, v,w in Digit, a in Digit^(c.n1*c.n2),
//             e in [a_0, a_(c.n1*c.n2)] where e = a_(c.n1*c.n2).
// Output:     a in Digit^(c.n1*c.n2) such that a := F_(c.n1*c.n2)^(-1)(a) ||
{
  const size_t nn = n1*n2;
  do {
    const Digit* pF = pA+nn;        // second step
    do {
      fftinv(pA, omega2, n2);
      pA += n2;
```

```
    } while (pA != pF);
    pA -= nn;

    multiply_matrix(pA, w, iw);     // third step

    transpose(pA, n1, n2);          // forth step

    pF = pA+nn;                     // fifth step
    do {
      brevorder(n1, pA, order);
      fftinv(pA, omega, n1);
      pA += n1;
    } while (pA != pF);
    pA -= nn;

    transpose(pA, n2, n1);          // sixth step
    pA += nn;
  } while (pA != pE);
}
```

◇

Macro referenced in 409.

Das Transponieren der Matrix im zweiten Schritt wird aus Speichergründen in der Eingabematrix durchgeführt:

⟨ transpose and copy a matrix 136 ⟩ ≡

```
static void transpose_sqr(Digit* a, size_t n, size_t m)
{
  CONDITION(n <= m);

  size_t nn = m-n;
  const Digit* c = a+n;
  const Digit* d = a+n*m-nn;
  do {
    Digit* b = a+m;
    while (++a != c) { swap(*a, *b); b += m; }
    a += ++nn; c += m;
  } while (c != d);
}

static void transpose(Digit* a, size_t n, size_t m)
{
  if (n == m) transpose_sqr(a, n, n);
  else if (m == 2*n) {
    transpose_sqr(a, n, m);
    transpose_sqr(a+n, n, m);
    CONDITION(n >= 2);

    Digit* s = new Digit[n];
    bool*  r = new bool[m];
    bool*  t = r+m;
    FILL_ZERO(r, t);
    r -= m;

    --m;
```

```
    size_t j = 1;
    do {
      size_t i = j;
      Digit* p = a+n*i;
      Digit* t = p+n;
      COPY(s, p, p, t);
      while (true) {
        i = (i < n) ? 2*i : 2*(i-n)+1;
        if (i == j) break;
        r[i] = true;
        t = p-n;
        Digit* q = a+n*i;
        COPY(t, q, t, p);
        i = (i < n) ? 2*i : 2*(i-n)+1;
        if (i == j) { p = q; break; }
        r[i] = true;
        t = q-n;
        p = a+n*i;
        COPY(t, p, t, q);
      }
      t = p-n; s -= n;
      COPY(t, s, t, p);
      s -= n;
      while (r[++j]);
    } while (j < m);

    delete[] r;
    delete[] s;
  } else {
    CONDITION(n == 2*m && m >= 2);

    Digit* s = new Digit[m];
    bool*  r = new bool[n];
    bool*  t = r+n;
    FILL_ZERO(r, t);
    r -= n;

    --n;
    size_t j = 1;
    do {
      size_t i = j;
      Digit* p = a+m*i;
      Digit* t = p+m;
      COPY(s, p, p, t);
      while (true) {
        i = (i&1)? i/2+m : i/2;
        if (i == j) break;
        r[i] = true;
        t = p-m;
        Digit* q = a+m*i;
        COPY(t, q, t, p);
        i = (i&1)? i/2+m : i/2;
        if (i == j) { p = q; break; }
        r[i] = true;
        t = q-m;
        p = a+m*i;
```

```
        COPY(t, p, t, q);
      }
      t = p-m; s -= m;
      COPY(t, s, t, p);
      s -= m;
      while (r[++j]);
    } while (j < n);

    delete[] r;
    delete[] s;

    ++n;
    transpose_sqr(a, m, n);
    transpose_sqr(a+m, m, n);
  }
}
```
◇

Macro referenced in 409.

Im dritten Schritt werden die Matrixeinträge (j_1, j_2) komponentenweise mit $w^{j_1 j_2}$ multipliziert:

⟨ multiply matrix components by a power of a primitive root of unity 138 ⟩ ≡

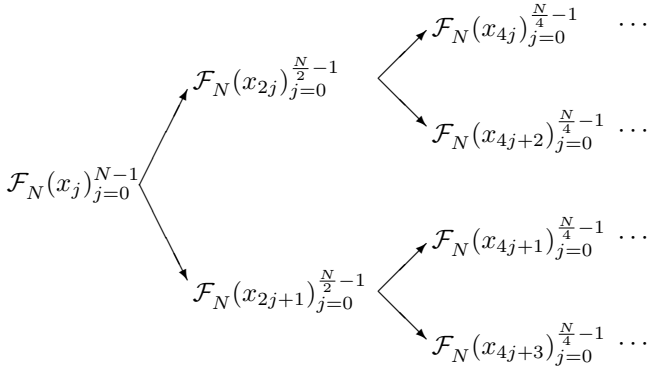
```
void FFT::multiply_matrix(Digit* pA, const Digit w, const Digit iw) const
// Algorithm:  c.multiply_matrix(a, w, v)
// Input:      c in FFT, v,w in Digit,
//             a = ([a_0, a_c.n2[,...,[a_((c.n1-1)*c.n2), a_(c.n1*c.n2)[)
//             in Digit^(c.n1 x c.n2).
// Output:     a in Digit^(c.n1*c.n2)
//             such that a := (a_ij * v * w^(i*j))_(0 <= i,j < c.n1*c.n2) ||
{
  const Digit M = m;
  Digit u = 1;
  const size_t k = n2;
  const Digit* pE = pA+n1*k;
  do {
    const Digit* pF = pA+k;
    Digit v = iw;
    do {
      digitmulmod(*pA, v, M, *pA);
      digitmulmod(v, u, M, v);
    } while (++pA != pF);
    digitmulmod(u, w, M, u);
  } while (pA != pE);
}
```
◇

Macro referenced in 409.

Nichtrekursive schnelle Fouriertransformation

Die Reduktionsformel im Kapitel 2.11.9 auf der Seite 121 ist der Durchbruch bei der Suche nach effizienten Algorithmen. Allerdings verlangt eine Implementierung der rekursiven Reduktionsformel einen hohen Organisationsaufwand zur Bewältigung der fortgesetzten Unterteilung des zu transformierenden Vektors und der Speicherung der diversen Teilvektoren.
Beo der Betrachtung der Indizes des zu transformierenden Vektors in der Binärdarstellung läßt sich

zeigen, daß das der schnellen Fouriertransformation zugrundeliegende Unterteilungsschema ohne die Rekursion direkt erzeugt werden kann. Dabei setzen wir voraus, daß die Transformationslänge $N = 2^L$ eine Zweierpotenz ist und berechnen $\mathcal{F}_N(x_j)_{j=0}^{N-1}$ für $(x_j)_{j=0}^{N-1} \in \mathbb{F}_p^N$ durch rekursive Anwendung der Rekursionsformeln:



Beispiel. Betrachten wir die Binärdarstellung der Indizes des obigen Schaubildes für den Fall $N = 8$, so stellen wir folgendes fest:

	Bitumkehrung →		
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Den Beweis, daß die Bitumkehrung in der Tat der Schlüssel zur Implementierung der nichtrekursiven schnellen Fouriertransformation ist, kann man zum Beispiel im Buch [17] von Otto Forster auf den Seiten 177-183 nachlesen.

Für die Bitumkehrung verwenden wir die folgende Funktion:

⟨ brevorder initialization 139 ⟩ ≡

```
static void init_order(size_t* a, size_t b)
// Algorithm:  init_order(a, b)
// Input:      a in size_t^b,
//             b in size_t where b > 2 and b = 2^x for x in N.
// Output:     a = [a_0, a_b[ in size_t^b such that
//             a_i = bitreverse(i) for 0 <= i < b ||
{
  CONDITION(b > 2 && (b & (~b+1)) == b);

  const size_t* c = a+b-1;
  b >>= 1;
  size_t j,i = b;
  *a = 0; *++a = i;
  do {
    for (j = b; j <= i; j >>= 1) i -= j;
```

```
    ***a = i += j;
  } while (a != c);
}
◇
```

Macro referenced in 409.

Die Bitumkehrung benötigen wir für die Umordnung in die "brevorder"-Darstellung:

⟨ brevorder an array 140a ⟩ ≡

```
static void brevorder(const size_t a, Digit* b, const size_t* c)
// Algorithm:  brevorder(a, b, c)
// Input:      a in size_t, b,c in Digit^a where a > 2.
// Output:     b = [b_0, b_a[ in Digit^a such that
//             t := b_1, b_1 := b_(a/2), b_(a/2) := t,
//             t := b_i, b_i := b_j, b_j := t  if i < j
//             with j = c_i and 2 <= i < a ||
{
  CONDITION(a > 2);

  Digit i = 2;
  swap(b[1], b[a/2]);
  do {
    const Digit j = c[i];
    if (i < j) swap(b[i], b[j]);
  } while (++i < a);
}
◇
```

Macro referenced in 409.

Die inneren Schleifen der Fouriertransfomation führen wir aus Effizienzgründen mit den folgenden drei Funktionen durch:

⟨ inner loop of fft without multiplication 140b ⟩ ≡

```
void FFT::innerfft(const Digit* pE, Digit* pA, Digit* pB,
                   const size_t i, const Digit M) const
{
  do {
    Digit a = *pA;
    Digit b = *pB;
    Digit c = a+b;
    if (M-a <= b) c -= M;
    if (a < b) a += M;
    *pA = c; *pB = a-b;
    pA += i; pB += i;
  } while (pA < pE);
}
◇
```

Macro referenced in 409.

⟨ inner loop of fft 140c ⟩ ≡

```
void FFT::innerfft(const Digit* pE, Digit* pA, Digit* pB,
                   const size_t i, const Digit M, const Digit o) const
{
  do {
```

```
    Digit a = *pA;
    Digit b = *pB;
    Digit c = a+b;
    if (M-a <= b) c -= M;
    if (a < b) a += M;
    a -= b;
    digitmulmod(o, a, M, a);
    *pA = c; *pB = a;
    pA += i; pB += i;
  } while (pA < pE);
}
```

◇

Macro referenced in 409.

⟨ inner loop of inverse fft 141a ⟩ ≡

```
void FFT::innerfftinv(const Digit* pE, Digit* pA, Digit* pB,
                      const size_t i, const Digit M, const Digit o) const
{
  do {
    Digit a = *pA;
    Digit b = *pB;
    digitmulmod(o, b, M, b);
    Digit c = a+b;
    if (M-a <= b) c -= M;
    if (a < b) a += M;
    *pA = c; *pB = a-b;
    pA += i; pB += i;
  } while (pA < pE);
}
```

◇

Macro referenced in 409.

Die eigentliche Fouriertransfomation führen wir dann mit den folgenden beiden Funktionen durch:

⟨ memberfunction FFT.fft 141b ⟩ ≡

```
void FFT::fft(Digit* a, const Digit* b, const size_t c) const
{
  CONDITION((c & (~c+1)) == c);

  for (size_t i = 1, j = c/2; i != c; i *= 2, j /= 2) {
    innerfft(a+c, a, a+j, 2*j, m);

    const Digit* s = b+i;
    for (size_t k = 1; k < j; ++k) {
      innerfft(a+c, a+k, a+k+j, 2*j, m, *s);
      s += i;
    }
  }
}
```

◇

Macro referenced in 409.

⟨ memberfunction FFT.fftinv 141c ⟩ ≡

```
void FFT::fftinv(Digit* a, const Digit* b, const size_t c) const
{
  CONDITION((c & (~c+1)) == c);

  for (size_t i = 1, j = c/2; i != c; i *= 2, j /= 2) {
    innerfft(a+c, a, a+i, 2*i, m);

    const Digit* s = b+j;
    for (size_t k = 1; k < i; ++k) {
      innerfftinv(a+c, a+k, a+k+i, 2*i, m, *s);
      s += j;
    }
  }
}
◇
```

Macro referenced in 409.

Transformationslänge teilbar durch drei

Die Transformationslänge kann zusätzlich zu der Zweierpotenz noch einen Faktor 3 enthalten. Falls wir nun einen Faktor 3 in der Transformationslänge haben, dann werden die folgenden beiden Funktionen eingesetzt:

⟨ memberfunction FFT.innerfft3 142 ⟩ ≡

```
void FFT::innerfft3(Digit* pT, const size_t sT, const Digit w) const
{
  const Digit M = m;
  Digit ww;
  digitmulmod(w, w, M, ww);
  Digit w1 = digitinv(2, M);
  Digit w2 = pow(w, sT, M);
  if (M-w2 <= w1) w2 -= M;
  w2 += w1;
  digitmulmod(3, w1, M, w1);
  w1 = M - w1;            // w1 != 0

  Digit o  = 1;
  Digit oo = 1;
  const Digit* pE = pT+sT;
  do {
    Digit x = pT[0];
    Digit y = pT[sT];
    Digit z = pT[2*sT];

    Digit s = y+z;
    if (M-y <= z) s -= M;
    if (y < z) z -= M;
    z = y - z;
    if (M-x <= s) x -= M;
    x += s;
    digitmulmod(s, w1, M, s);
    digitmulmod(z, w2, M, z);
    if (M-s <= x) s -= M;
    s += x;
    y = s + z;
```

```
    if (M-s <= z) y -= M;
    if (s < z) z -= M;
    z = s - z;
    digitmulmod(y, o, M, y);
    digitmulmod(z, oo, M, z);

    pT[0] = x;
    pT[sT] = y;
    pT[2*sT] = z;

    digitmulmod(o, w, M, o);
    digitmulmod(oo, ww, M, oo);
  } while (++pT != pE);
  static Digit* x = pT-sT;       // to avoid a Microsoft Visual C++ 6.0 "/O2" bug
}
◇
```

Macro referenced in 409.

⟨ memberfunction FFT.innerfftinv3 143 ⟩ ≡

```
void FFT::innerfftinv3(Digit* pT, const size_t sT, const Digit w) const
{
  const Digit M = m;
  Digit ww;
  digitmulmod(w, w, M, ww);
  Digit w1 = digitinv(2, M);
  Digit w2 = pow(w, sT, M);
  if (M-w2 <= w1) w2 -= M;
  w2 += w1;
  digitmulmod(3, w1, M, w1);
  w1 = M - w1;            // w1 != 0

  Digit o  = 1;
  Digit oo = 1;
  const Digit* pE = pT+sT;
  do {
    Digit x = pT[0];
    Digit y = pT[sT];
    Digit z = pT[2*sT];

    digitmulmod(y, o, M, y);
    digitmulmod(z, oo, M, z);
    Digit s = y+z;
    if (M-y <= z) s -= M;
    if (y < z) z -= M;
    z = y - z;
    if (M-x <= s) x -= M;
    x += s;
    digitmulmod(s, w1, M, s);
    digitmulmod(z, w2, M, z);
    if (M-s <= x) s -= M;
    s += x;
    y = s + z;
    if (M-s <= z) y -= M;
    if (s < z) z -= M;
    z = s - z;
```

```
    pT[0] = x;
    pT[sT] = y;
    pT[2*sT] = z;

    digitmulmod(o, w, M, o);
    digitmulmod(oo, ww, M, oo);
  } while (++pT != pE);
  static Digit* x = pT-sT;       // to avoid a Microsoft Visual C++ 6.0 "/O2" bug
}
```

◇

Macro referenced in 409.

Bemerkung. Nach [51] erhalten wir einen Laufzeitgewinn von 25%, wenn wir den Faktor 3 in der Transformationslänge berücksichtigen.

Quadratur

Zur Durchführung der Quadratur eines Naturals mit Hilfe der schnellen Fouriertransformation benötigen wir eine N-te Einheitswurzel, die wir aus der Primitivwurzel und der Transformationslänge berechnen. Das (Fourier-)transformierte Natural wird komponentenweise quadriert und dann wieder zurück transformiert.

⟨ squaring of a Natural with FFT 144 ⟩ ≡

```
void FFT::sqr(Natural& b)
// Algorithm:  a.sqr(b)
// Input:      a in FFT, b in Natural.
// Output:     b in Natural such that b = (a.t_0)^2 ||
{
  t[2] = t[1] = t[0];
  const size_t sT = size();
  if (sT%3 == 0) {
    for (size_t i = 0; i < 3; ++i) {
      setmodulo(moduli[i]);
      Digit w  = pow(primroots[i], (m-1)/(sT/3), m);
      Digit w2 = pow(primroots[i], (m-1)/sT, m);
      init_omega(w);

      // fft:
      Digit* pT = t[i].p;
      innerfft3(pT, sT/3, w2);
      five_step(pT+sT, pT, w);

      square(i);

      // fftinv:
      w = digitinv(w, m);
      w2 = digitinv(w2, m);
      init_omega(w);
      five_step(pT+sT, pT, w, digitinv(sT, m));
      innerfftinv3(pT, sT/3, w2);
    }
  } else {
    for (size_t i = 0; i < 3; ++i) {
      setmodulo(moduli[i]);
      Digit w  = pow(primroots[i], (m-1)/sT, m);
```

```
      init_omega(w);

      // fft:
      Digit* pT = t[i].p;
      five_step(pT+sT, pT, w);

      square(i);

      // fftinv:
      w = digitinv(w, m);
      init_omega(w);
      five_step(pT+sT, pT, w, digitinv(sT, m));
    }
  }
  chinese_remainder();
  result(b);
}
◇
```

Macro referenced in 409.

Die komponentenweise Quadratur führen wir dann mit der folgenden Funktion aus:

⟨ componentwise squaring 145a ⟩ ≡

```
void FFT::square(const size_t a)
// Algorithm:  b.square(a)
// Input:      b in FFT, a in size_t.
// Output:     b in FFT such that (b.t_a)_i := ((b.t_a)_i)^2 (mod b.m)
//             for 0 <= i < b.t_a.size ||
{
  Digit* pT = t[a].p;
  const Digit* pE = pT+t[a].size;
  const Digit M = m;
  do {
    const Digit x = *pT;
    digitmulmod(x, x, M, *pT);
  } while (++pT != pE);
}
◇
```

Macro referenced in 409.

Multiplikation

Bei der Duchführung der Multiplikation benötigen wir insgesamt $3 \cdot 3$ Fouriertransformationen statt der $3 \cdot 2$ Transformationen bei der Quadratur:

⟨ multiplication of two Naturals with FFT 145b ⟩ ≡

```
void FFT::mul(Natural& b)
// Algorithm:  a.mul(b)
// Input:      a in FFT, b in Natural where not a.factor = 0
//             and a.factor->size = a.size.
// Output:     b in Natural such that b = (a.t_0)*(a.factor->t_0) ||
{
  CONDITION(factor && factor->size() == size());
```

```
    t[2] = t[1] = t[0];
    const size_t sT = size();
    if (sT%3 == 0) {
      for (size_t i = 0; i < 3; ++i) {
        setmodulo(moduli[i]);
        Digit w  = pow(primroots[i], (m-1)/(sT/3), m);
        Digit w2 = pow(primroots[i], (m-1)/sT, m);
        init_omega(w);

        // factor->fft:
        if (i >= 1) base_conversion(factor->t[0], factor->arg, factor->shift);
        Digit* pT = factor->t[0].p;
        innerfft3(pT, sT/3, w2);
        five_step(pT+sT, pT, w);

        // fft:
        pT = t[i].p;
        innerfft3(pT, sT/3, w2);
        five_step(pT+sT, pT, w);

        multiply(i);

        // fftinv:
        w = digitinv(w, m);
        w2 = digitinv(w2, m);
        init_omega(w);
        five_step(pT+sT, pT, w, digitinv(sT, m));
        innerfftinv3(pT, sT/3, w2);
      }
    } else {
      for (size_t i = 0; i < 3; ++i) {
        setmodulo(moduli[i]);
        Digit w  = pow(primroots[i], (m-1)/sT, m);
        init_omega(w);

        // factor->fft:
        if (i >= 1) base_conversion(factor->t[0], factor->arg, factor->shift);
        Digit* pT = factor->t[0].p;
        five_step(pT+sT, pT, w);

        // fft:
        pT = t[i].p;
        five_step(pT+sT, pT, w);

        multiply(i);

        // fftinv:
        w = digitinv(w, m);
        init_omega(w);
        five_step(pT+sT, pT, w, digitinv(sT, m));
      }
    }
    chinese_remainder();
    result(b);
  }
  ◇
```

Macro referenced in 409.

Die komponentenweise Multiplikation führen wir dann mit der folgenden Funktion aus:

⟨ componentwise multiplication 147a ⟩ ≡

```
void FFT::multiply(const size_t a)
// Algorithm:  b.multiply(a)
// Input:      b in FFT, a in size_t.
// Output:     b in FFT such that (b.t_a)_i := ((b.t_a)_i)*((v.factor->t_0)_i) (mod b.m)
//             for 0 <= i < b.t_a.size ||
{
  CONDITION(factor && factor->t[0].size == t[a].size);

  Digit* pT = t[a].p;
  Digit* pA = factor->t[0].p;
  const Digit* pE = pT+t[a].size;
  const Digit M = m;
  do {
    const Digit x = *pT;
    const Digit y = *pA++;
    digitmulmod(x, y, M, *pT);
  } while (++pT != pE);
}
◇
```

Macro referenced in 409.

2.12 Binärer Logarithmus

Binärer Logarithmus für Digits

Zuerst betrachten wir den binären Logarithmus eines Digits:

$$\mathsf{log2} : \mathsf{Digit} \to \mathsf{Digit} : a \mapsto \begin{cases} 0\,, & a = 0 \\ \lfloor \log_2(a) \rfloor\,, & a > 0 \end{cases}.$$

Eine Lösungsmöglichkeit dafür sind Schiebeoperationen:

⟨ binary logarithm for Digits 147b ⟩ ≡

```
STATIC_VS_INLINE Digit log2(Digit a)
// Algorithm:  b := log2(a)
// Input:      a in Digit.
// Output:     b in Digit
//             such that if a > 0 then b = [log2(a)] else b = 0 ||
{
  Digit b = 0;

#if CHAR_BIT == 8
  static const Digit c[16] = {0, 0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3 };

  if (a > GAMMA_LOW) { b += BETA/2; a >>= BETA/2; }
  if (a >= Digit(1) << BETA/4) { b += BETA/4; a >>= BETA/4; }
  if (a >= Digit(1) << BETA/8) { b += BETA/8; a >>= BETA/8; }
  if (BETA > 32) {
    if (a >= Digit(1) << BETA/16) { b += BETA/16; a >>= BETA/16; }
    if (BETA > 64) {
      while (a >= 16) { b += 4; a >>= 4; }
```

```
    }
  }
  return b+c[a];
#else
  while (a >>= 1) ++b;
  return b;
#endif
}
```
◇

Macro referenced in 397d.

Laufzeit. Sei $a \in$ Digit, dann `log2(`a`)`$\in \mathcal{O}(\frac{\beta}{2})$.

Binärer Logarithmus für Naturals

Mit dem gerade vorgestellten Programm und der Längeninformation des Naturals kann nun der binäre Logarithmus direkt angegeben werden:

⟨ binary logarithm for Naturals 148a ⟩ ≡
```
inline Digit log2(const Natural& a)
// Algorithm:  b := log2(a)
// Input:      a in Natural.
// Output:     b in Digit
//             such that if a > 0 then b = [log2(a)] else b = 0 ||
{
  return log2(a.highest()) + (a.length()-1)*BETA;
}
```
◇

Macro referenced in 402b.

Laufzeit. Sei $x \in$ Natural, dann `log2(`x`)` $\sim$ `log2(`x`.highest())`.

2.13 Potenzieren

Seien $a, b \in$ Natural und $b > 0$, so betrachten wir, um die Potenz a^b berechnen zu können, die Binärdarstellung von b:

$$b = \sum_{i=0}^{n} b_i \cdot 2^i \qquad \text{mit } b_i \in \{0,1\}, n = \lfloor \log_2 b \rfloor.$$

Daher gilt

$$a^b = \prod_{i=0}^{n} a^{b_i \cdot 2^i}.$$

Eine rekursive Darstellung ist demzufolge

$$a_n := a^{b_n}, a_i := a^{b_i} \cdot a_{i+1}^2 \qquad \text{für } 0 \le i \le n.$$

Also können wir das Potenzieren folgendermaßen implementieren:

⟨ calculates the power of a Natural by a Digit 148b ⟩ ≡

```
Natural pow(const Natural& a, Digit b)
// Algorithm:  c := pow(a, b)
// Input:      a in Natural, b in Digit.
// Output:     c in Natural such that c = a^b ||
{
  if (b == 0) return 1;
  Digit c = Digit(1) << log2(b);

  Natural d(a);
  while (b != c) {
    b &= ~c; c >>= 1;
    d *= d;
    if (b & c) d *= a;
  }
  while (b >>= 1) d *= d;
  return d;
}
◇
```

Macro referenced in 409.

Laufzeit. Sei $x \in$ Natural und $a \in$ Digit, dann pow$(x,a) \in \log_2(a)\mathcal{O}($mul : Natural$^2 \to$ Natural$)$.

Wir können die Potenz a^b auch aus der anderen Richtung ermitteln, nämlich mit der Rekursion

$$a^b = \begin{cases} (a^{\frac{b}{2}})^2\,, & 2 \mid b \\ a \cdot (a^{\frac{b-1}{2}})^2\,, & 2 \nmid b \end{cases}\,.$$

Dies ergibt den folgenden Potenzieralgorithmus:

⟨ calculates the power of a Natural by a Natural 149 ⟩ ≡

```
Natural pow(const Natural& a, Natural b)
// Algorithm:  c := pow(a, b)
// Input:      a,b in Natural.
// Output:     c in Natural such that c = a^b ||
{
  if (b == 1) return a;
  else if (b > 1) {
    Natural c = 1;
    Natural d = a;
    do {
      if (b.odd()) c *= d;
      b >>= 1;
      d *= d;
    } while (b > 1);
    return c *= d;
  } else return 1;
}
◇
```

Macro referenced in 409.

Laufzeit. Seien $x, y \in$ Natural, dann pow$(x,y) \in \log_2(y)\mathcal{O}($mul : Natural$^2 \to$ Natural$)$.

2.14 Division

2.14.1 Digitoperationen

Bevor wir im nächsten Abschnitt mit Langzahlen operieren, betrachten wir hier zuerst die Division eines Doppel-Digits durch ein Digit, das heißt wir berechnen die Division $(a2^\beta + b)/c$ für $a, b, c \in$ Digit.

Eingeschränkte Version

Wir nehmen $a < c$ und $c \geq 2^{\beta-1}$ an. Dadurch sieht die Implementation folgnedermaßen aus:

⟨ digitdiv in ANSI-C++ 150 ⟩ ≡

```
void NumberBase::digitdiv(const Digit a, const Digit b, const Digit c,
                          Digit& q, Digit& r) const
// Algorithm:  n.digitdiv(a, b, c, q, r)
// Input:      n in NumberBase, a,b,c in Digit where a < c, c >= 2^(BETA-1).
// Output:     q,r in Digit such that q*c + r = a*2^BETA + b ||
{
  CONDITION(a < c && c >= (Digit(1) << (BETA-1)));

  Digit d = c >> BETA/2;
  Digit e = c & GAMMA_LOW;
  Digit x = a/d;
  Digit z = a-x*d;
  Digit y = x*e;
  z = (z << BETA/2) | (b >> BETA/2);
  if (z < y) {
    --x; z += c;
    if (z >= c && z < y) { --x; z += c; }
  }
  q = x << BETA/2;
  z -= y;
  x = z/d;
  z -= x*d;
  y = x*e;
  z = (z << BETA/2) | (b & GAMMA_LOW);
  if (z < y) {
    --x; z += c;
    if (z >= c && z < y) { --x; z += c; }
  }
  q |= x; r = z-y;
}
```

◇

Macro referenced in 398b.

Laufzeit. Sei $n \in$ NumberBase und $a_0, a_1, b, d, e \in$ Digit, dann n.digitdiv(a_0,a_1,b,d,e)$\in \mathcal{O}(1)$.

Dieses Resultat ist im Gegensatz zur Assembler-Lösung auf einem 80386-Prozessor mit dem Visual-C++ Compiler eher bescheiden. Dort sieht sie nämlich folgendermaßen aus:

⟨ digitdiv in assembler for a i386 processor with the Visual-Compiler 151a ⟩ ≡

```
static void __declspec(naked) __fastcall _digitdiv(Digit* q, Digit* r, const Digit a,
                                                   const Digit b, const Digit c)
{
  __asm {
    push edi
    mov edi,edx
    mov edx,[esp+8]
    mov eax,[esp+12]
    div dword ptr [esp+16]
    mov [ecx],eax
    mov [edi],edx
    pop edi
    ret 12
  }
}

inline void NumberBase::digitdiv(const Digit a, const Digit b, const Digit c,
                                 Digit& q, Digit& r) const
{
  _digitdiv(&q, &r, a, b, c);
}
```

◇

Macro referenced in 398a.

Für $c \leq \gamma_{low}$ verlieren wir durch unsere ANSI-Forderung einen Faktor von 2,2 bis 2,7 je nach Compiler, für den Fall, daß $c > \gamma_{low}$ ist, sogar einen Faktor 4,5 bis 7,1. Bei der 32-Bit Version werden die Ergebnisse nochmals deutlich schlechter, da die Assembler-Version nur von 31 auf 47 Taktzyklen steigt, also um etwa 51%, während die ANSI-Version ihre Laufzeit genau verdoppelt.

Für den Watcom-Compiler ist der Assembler-Code als `inline` realisierbar. Dies ist schneller und kompakter als die vorige Version für den Visual-Compiler, da die Eingabewerte vom Compiler übernommen werden:

⟨ digitdiv in assembler for a i386 processor with the Watcom-Compiler 151b ⟩ ≡

```
void _digitdiv(const Digit, const Digit, const Digit, Digit&, Digit&);
#pragma aux _digitdiv =               \
   "div ebx"                          \
   "mov [esi],eax"                    \
   "mov [edi],edx"                    \
   parm [EDX] [EAX] [EBX] [ESI] [EDI] \
   modify [ ESI EDI EAX EDX ];

inline void NumberBase::digitdiv(const Digit a, const Digit b, const Digit c,
                                 Digit& q, Digit& r) const
{
  _digitdiv(a, b, c, q, r);
}
```

◇

Macro referenced in 398a.

Noch kompakter und schneller arbeitet der erzeugte Assembler-Code des GNU-Compilers, weil er alle Argumente übernimmt:

⟨ digitdiv in assembler for a i386 processor with the GNU-Compiler 152a ⟩ ≡

```
inline void NumberBase::digitdiv(const Digit a, const Digit b, const Digit c,
                                 Digit& q, Digit& r) const
{
  __asm__ ("divl %4"
    : "=a" (q), "=d" (r)
    : "0"  (b), "1"  (a), "rm" (c));
}
```

◇

Macro referenced in 398a.

Im Gegensatz zum 80x86-Prozessor muß der Divisionsrest auf einem SPARC-v8-Prozessor zusätzlich berechnet werden:

⟨ digitdiv in assembler for a SPARC v8 processor with the GNU-Compiler 152b ⟩ ≡

```
inline void NumberBase::digitdiv(const Digit a, const Digit b, const Digit c,
                                 Digit& q, Digit& r) const
{
  __asm__ ("mov %2,%%y;nop;nop;nop;udiv %3,%4,%0;umul %0,%4,%1;sub %3,%1,%1"
    : "=&r" (q), "=&r" (r)
    : "r" (a), "r" (b), "r" (c));
}
```

◇

Macro referenced in 398a.

Moduloberechnung

⟨ digitmod of a double Digit 152c ⟩ ≡

```
inline void NumberBase::digitmod(Digit a, Digit b, const Digit c, Digit& r) const
// Algorithm:  n.digitmod(a, b, c, r)
// Input:      n in NumberBase, a,b,c in Digit where not c = 0.
// Output:     r in Digit such that r = a*2^BETA+b - c*[(a*2^BETA+b)/c] ||
{
#ifdef _DigitAsm_
  if (a < c) {
    Digit d;
    digitdiv(a, b, c, d, r);
  } else {
    Digit d,e = 0;
    digitdiv(e, a, c, d, e);
    digitdiv(e, b, c, d, r);
  }
#else
  if (c > GAMMA/2) {
    Digit d,e = 0;
    digitdiv(e, a, c, d, e);
    digitdiv(e, b, c, d, r);
  } else {
    const Digit n = log2(c)+1;
    const Digit n2 = BETA-n;
    const Digit c2 = c << n2;
    Digit d,z = a >> n;
    digitdiv(z, (a << n2) | (b >> n), c2, d, z);
    digitdiv(z, b << n2, c2, d, z);
    r = z >> n2;
  }
#endif
```

```
}
```
◇

Macro referenced in 397d.

Laufzeit. Sei $n \in \mathsf{NumberBase}$ und $a_0, a_1, b, d, e \in \mathsf{Digit}$, dann

$$n.\texttt{digitmod}(a_0,a_1,b,e) \sim 2 \cdot n.\texttt{digitdiv}(a_0,a_1,b,d,e).$$

2.14.2 Divisions-Algorithmus

Beim Divisions-Algorithmus müssen wir nun wie beim handschriftlichen Dividieren die Leitziffer betrachten und eine möglichst gute Abschätzung ermitteln, um die anschließende Korrektur gering zu halten. Dies erzielen wir zum Beispiel durch eine Erweiterung von Dividend und Divisor, bis die Leitziffer des Dividenden nahe an γ liegt, so daß wir am Ende nur noch den entstanden Rest korrigieren müssen.
Dieser Divisions-Algorithmus läßt sich nun folgendermaßen verwirklichen:

⟨ division of two Naturals with remainder 153 ⟩ ≡

```
void div(const Natural& a, const Natural& b, Natural& q, Natural& r)
// Algorithm:  div(a, b, q, r)
// Input:      a,b,q,r in Natural where not b = 0, not q = r.
// Output:     q,r in Natural such that q = [a/b], r = a - q*b ||
{
  NATURALCONDITION(a);
  NATURALCONDITION(b);
  NATURAL_FOR_CHECK(_a, a);
  NATURAL_FOR_CHECK(_b, b);

  if (b == 0) b.errmsg(4, "(div)");
  if (q.p == r.p) r.errmsg(5, "(div)");
  switch (b.compare(a)) {
    case 0: r = 0; q = 1; return;
    case 1: r = a; q = 0; return;
  }
  const size_t sA = a.size;
  const size_t sB = b.size;
  if (sB == 1) {
    const Digit c = *b.p;     // important! (q = b)
    q = a; r = q.mod_div(c);
    return;
  } else if (sB >= NEWTON_DIV_MARK2 && sA >= sB+NEWTON_DIV_MARK1) {
    Natural* a2 = 0;
    Natural* b2 = 0;
    if (&q == &a) {
      a2 = new Natural(a);
      if (!a2) a.errmsg(2, "(div)");
    }
    if (&q == &b) {
      b2 = new Natural(b);
      if (!b2) b.errmsg(2, "(div)");
    }
    if (&r == &a) {
      a2 = new Natural(a);
      if (!a2) a.errmsg(2, "(div)");
    }
```

```
  if (a2)
    if (b2) q.div(*a2, *b2, r);
    else q.div(*a2, b, r);
  else if (b2) q.div(a, *b2, r);
  else q.div(a, b, r);
  delete b2;
  delete a2;
  return;
}

Digit m = 1;
int m2 = 0;
Natural c(3, b);    // because subpos
r = a;
Digit d = *c.p;
if (d != GAMMA) {
  m = GAMMA/(d+1);
  c *= m; r *= m;
  d = *c.p;
}
if (d == GAMMA) { m2 = 1; c *= GAMMA; r *= GAMMA; d = *c.p; }
if (d <= GAMMA/2) {
  m2 += 2; c <<= 1; r <<= 1; d = *c.p;
  if (d == GAMMA) { m2 += 4; c *= GAMMA; r *= GAMMA; d = *c.p; }
}
const Digit d2 = d+1;
const size_t sQ = r.size-c.size+(*r.p >= d);
Digit* pQ = q.setsize(sQ);
const Digit* pE = pQ+sQ;
FILL_ZERO(pQ, pE);
pQ -= sQ;
const size_t sC = c.size;
const Digit* pC = c.p+sC;
do {
  Digit* pR = r.p;
  const Digit k = *pR;
  if (k == d) {
    size_t sR = r.size;
    r.size = sC;
    if (r < c) {
      ++pR; ++pQ; r.size = sC+1;
      Digit s,t;
      q.digitdiv(k, *pR, d2, s, t);
      r.mulsub(c, s);
      while (r >= c) {
        subpos(r.p, r.p+r.size, pC);
        r.normalize();
        ++s;
      }
      *pQ++ = s;
      const Digit* pF = r.p;
      const size_t l = pE-pQ;
      const size_t m = pF-pR;
      r.size = sR-m-1;
      pQ += (l <= m)? l : m - (*pF >= d);
    } else {
```

```
        ++(*pQ);
        subpos(pR, pR+sC, pC);
        if (++pQ != pE) {
          Digit k = *++pR;
          if (k == 0)
            do { ++pR; --sR; k = *pR; } while (++pQ != pE && k == 0);
          pQ += (pQ != pE && k < d);
          r.p = pR; r.size = sR-1;
        } else r.size = sR;
      }
    } else if (k > d) {
      subpos(pR, pR+sC, pC);
      ++(*pQ); ++pQ;
      r.normalize();
      if (r < c) break;
    } else {
      Digit s,t;
      const size_t sR = r.size;
      r.size = sC+1;
      q.digitdiv(*pR, pR[1], d2, s, t);
      if (s) r.mulsub(c, s);
      else r.normalize();
      while (r >= c) {
        subpos(r.p, r.p+r.size, pC);
        r.normalize();
        ++s;
      }
      *pQ++ = s;
      const Digit* pF = r.p;
      const size_t l = pE-pQ;
      const size_t m = pF-pR-1;
      r.size = sR-m-1;
      pQ += (l <= m)? l : m - (*pF >= d);
    }
  } while (pQ != pE);
  q.normalize(); r.normalize();

  if (r == c) {
    r = 0; ++q;
  } else {
    r /= m;
    if (m2&1) r /= GAMMA;
    if (m2&2) r >>= 1;
    if (m2&4) r /= GAMMA;
  }

  CONDITION(r < _b);
  CONDITION(r+q*_b == _a);
  NATURALCONDITION(q);
  NATURALCONDITION(r);
}
◇
```

Macro referenced in 409.

Laufzeit. Seien $x, y, z_0, z_1 \in$ Natural, dann `div(`x`,`y`,`z_0`,`z_1`)`$\in \mathcal{O}(\mathcal{L}(x)\mathcal{L}(y) - \mathcal{L}(y)^2)$.

Gleichzeitige Multiplikation und Subtraktion

Die hierfür notwendige Funktion `mulsub` hat eine äquivalente Struktur zur Funktion `muladd` (siehe Kapitel 2.11.2, Seite 93) und besitzt die folgende Gestalt:

⟨ multiplication and subtraction of a Natural by a Digit 156 ⟩ ≡

```
void Natural::mulsub(const Natural& a, const Digit b)
// Algorithm:  c.mulsub(a, b)
// Input:      a,c in Natural, b in Digit.
// Output:     c in Natural such that c := c - a*b ||
{
  NATURALCONDITION(*this);
  NATURALCONDITION(a);

  const size_t sT = size;
  const size_t sA = a.size;
  if (sA > sT) errmsg(3, "(mulsub)");
  const Digit* pA = a.p;
  const Digit* pE = pA+sA;
  Digit* pT = p+sT;
  Digit x,y,z = 0;

  do {
    digitmul(*--pE, b, x, y);
    y += z;
    z = (y < z) + x;
    x = *--pT;
    z += (x < y);
    *pT = x-y;
  } while (pA != pE);
  if (z) {
    const Digit* pE = p;
    if (pT == pE) errmsg(3, "(mulsub)");
    Digit x = *--pT;
    if (x < z) dec(pT);
    *pT = x -= z;
    if (x == 0 && pT == pE) {       // normalize()
      size_t sT = size;
      if (sT > 2) {
        do { ++pT; --sT; } while (*pT == 0 && sT > 1);
        p = pT; size = sT;
      } else if (sT == 2) { p = ++pT; size = --sT; }
    }
  }

  NATURALCONDITION(*this);
}
```

◇

Macro referenced in 409.

Laufzeit. Seien $x, y \in$ Natural und $a \in$ Digit, dann x`.mulsub(`y`,`a`)` $\in \mathcal{O}(\mathcal{L}(y))$.

Division eines Naturals durch ein Digit

Eine häufig benötigte Funktion ist das Dividieren eines Naturals durch ein Digit. Bei der gerade vorgestellten allgemeinen Divisionsfunktion wird bei jedem Aufruf das Digit in ein Natural umgewandelt

und der verbleibende Divisionsrest durch ein Natural repräsentiert. Dies ist eine stabile Variante, weil bei fortlaufenden Berechnungen kein unerwünschter Überlauf entstehen kann. Für den Spezialfall der Division eines Naturals durch ein Digit lohnt sich die folgende Implementation:

⟨ division of a Natural by a Digit 157 ⟩ ≡

```
#if defined(_DigitAsm_) && _M_IX86 >= 300 && defined(_MSC_VER)
static void x_div(const Digit* pE, const Digit* pA, Digit* pC, const Digit b, Digit& d)
{
  __asm {
    xor edx,edx
    mov edi,pA
    mov esi,pC
    mov ebx,b
    mov ecx,pE
L1: mov eax,[edi]
    div ebx
    mov [esi],eax
    add edi,4
    add esi,4
    cmp edi,ecx
    jne L1
    mov eax,d
    mov [eax],edx
  }
}
#endif

void div(const Natural& a, const Digit b, Natural& c, Digit& d)
// Algorithm:  div(a, b, q, r)
// Input:      a in Natural, b in Digit where not b = 0.
// Output:     q in Natural, r in Digit such that q = [a/b], r = a - q*b ||
{
  NATURALCONDITION(a);

  if (b == 0) a.errmsg(4, "(div)");

  const size_t sA = a.size;
  const Digit* pA = a.p;
  Digit* pC = c.p;
  if (pA != pC) pC = c.setsize(sA);
  if (sA == 1) {
    Digit x = *pA;
    Digit y = x/b;
    *pC = y; d = x - y*b;
  } else {
    const Digit* pE = pA+sA;
#if defined(_DigitAsm_) && _M_IX86 >= 300 && defined(_MSC_VER)
    x_div(pE, pA, pC, b, d);
#elif defined(_DigitAsm_)
    Digit e = 0;
    do {
      a.digitdiv(e, *pA, b, *pC, e);
      ++pC;
    } while (++pA != pE);
    d = e;
#else
```

```
    if (b > GAMMA/2) {
      Digit e = 0;
      do {
        a.digitdiv(e, *pA, b, *pC, e);
        ++pC;
      } while (++pA != pE);
      d = e;
    } else {
      const Digit n = log2(b)+1;
      const Digit n2 = BETA-n;
      const Digit b2 = b << n2;
      Digit x,y = *pA++;
      Digit z = y >> n;
      do {
        x = *pA;
        a.digitdiv(z, (y << n2) | (x >> n), b2, *pC, z);
        ++pC;
        if (++pA == pE) { y = x; break; }
        y = *pA;
        a.digitdiv(z, (x << n2) | (y >> n), b2, *pC, z);
        ++pC;
      } while (++pA != pE);
      a.digitdiv(z, y << n2, b2, *pC, z);
      d = z >> n2;
    }
#endif
    c.normalize();
  }

  NATURALCONDITION(c);
}
◇
```

Macro referenced in 409.

Laufzeit. Seien $x, y \in$ Natural und $a, b \in$ Digit, dann `div(`x`,`a`,`y`,`b`)` $\in \mathcal{O}(\mathcal{L}(x))$.

Dieselbe Funktionalität erfüllt auch die geschützte Funktion `mod_div`, die sogar mit weniger Argumenten auskommt:

⟨ memberfunction Natural.mod_div 158 ⟩ ≡

```
inline Digit Natural::mod_div(const Digit b)
// Algorithm:  c := a.mod_div(b)
// Input:      a in Natural, b in Digit where not b = 0.
// Output:     a in Natural, c in Digit such that c = a - [a/b]*b, a := [a/b] ||
{
  Digit c;
  ::div(*this, b, *this, c);
  return c;
}
◇
```

Macro referenced in 402b.

Laufzeit. Sei $x \in$ Natural und $a \in$ Digit, dann x`.mod_div(`a`)` $\in \mathcal{O}(\mathcal{L}(x))$.

Divisionsoperatoren

Die Operatoren dürften nun keine Probleme mehr bereiten:

⟨ multiplicative operator/ for Naturals 159a ⟩ ≡

```
inline Natural::Natural(const binder_arguments<Natural, Natural,
                                               Natural_divides_tag>& a)
{
  get_memory(a.x.size+DELTA);
  Natural t(a.y.size+DELTA, ' ');
  div(a.x, a.y, t);
}

inline Natural& Natural::operator=(const binder_arguments<Natural, Natural,
                                                          Natural_divides_tag>& a)
{
  Natural t(a.y.size+DELTA, ' ');
  div(a.x, a.y, t);
  return *this;
}

inline binder_arguments<Natural, Natural, Natural_divides_tag>
 operator/(const Natural& a, const Natural& b)
// Algorithm:  c := a/b
// Input:      a,b in Natural where not b = 0.
// Output:     c in Natural such that c = [a/b] ||
{
  return binder_arguments<Natural, Natural, Natural_divides_tag>(a, b);
}
◇
```

Macro referenced in 402b.

Laufzeit. Seien $x, y, z_0, z_1 \in$ Natural, dann $x/y \sim$ `div(`x`,`y`,`z_0`,`z_1`)`.

⟨ multiplicative operator% for Naturals 159b ⟩ ≡

```
inline Natural::Natural(const binder_arguments<Natural, Natural,
                                               Natural_modulus_tag>& a)
{
  Natural t(a.x.size+DELTA, ' ');
  get_memory(a.y.size+DELTA);
  ::div(a.x, a.y, t, *this);
}

inline Natural& Natural::operator=(const binder_arguments<Natural, Natural,
                                                          Natural_modulus_tag>& a)
{
  Natural t(a.x.size+DELTA, ' ');
  ::div(a.x, a.y, t, *this);
  return *this;
}

inline binder_arguments<Natural, Natural, Natural_modulus_tag>
 operator%(const Natural& a, const Natural& b)
// Algorithm:  c := a%b
// Input:      a,b in Natural where not b = 0.
```

```
// Output:     c in Natural such that c = a - [a/b]*b ||
{
  return binder_arguments<Natural, Natural, Natural_modulus_tag>(a, b);
}
◇
```

Macro referenced in 402b.

Laufzeit. Seien $x, y, z_0, z_1 \in$ Natural, dann $x\%y \sim$ `div(`x`,`y`,`z_0`,`z_1`)`.

⟨ multiplicative `operator/` of a Natural with a Digit 160a ⟩ ≡

```
inline Natural operator/(const Natural& a, const Digit b)
// Algorithm:  c := a/b
// Input:      a in Natural, b in Digit where not b = 0.
// Output:     c in Natural such that c = [a/b] ||
{
  const size_t sA = a.length();
  if (sA == 1) return Natural(a.highest() / b);
  else return Natural(a) /= b;
}
◇
```

Macro referenced in 402b.

Laufzeit. Sei $x \in$ Natural und $a \in$ Digit, dann $x/a \sim x/{=}a$.

⟨ multiplicative `operator%` of a Natural with a Digit 160b ⟩ ≡

```
#if defined(_DigitAsm_) && _M_IX86 >= 300 && defined(_MSC_VER)
static Digit x_mod(const Digit* pE, const Digit* pA, const Digit b)
{
  Digit r;
  __asm {
    xor edx,edx
    mov edi,pA
    mov ebx,b
    mov ecx,pE
L1: mov eax,[edi]
    div ebx
    add edi,4
    cmp edi,ecx
    jne L1
    mov r,edx
  }
  return r;
}
#endif

Digit operator%(const Natural& a, const Digit b)
// Algorithm:  c := a%b
// Input:      a in Natural, b in Digit where not b = 0.
// Output:     c in Digit such that c = a - [a/b]*b ||
{
  NATURALCONDITION(a);

  if (b == 0) a.errmsg(4, "(operator%)");

  const size_t sA = a.size;
```

```
  const Digit* pA = a.p;
  if (sA == 1) return *pA % b;
  else {
    const Digit* pE = pA+sA;
#if defined(_DigitAsm_) && _M_IX86 >= 300 && defined(_MSC_VER)
    return x_mod(pE, pA, b);
#elif defined(_DigitAsm_)
    Digit d,c = 0;
    do a.digitdiv(c, *pA, b, d, c); while (++pA != pE);
    return c;
#else
    if (b > GAMMA/2) {
      Digit d,c = 0;
      do a.digitdiv(c, *pA, b, d, c); while (++pA != pE);
      return c;
    } else {
      const Digit n = log2(b)+1;
      const Digit n2 = BETA-n;
      const Digit b2 = b << n2;
      Digit x,y = *pA++;
      Digit d,z = y >> n;
      do {
        x = *pA;
        a.digitdiv(z, (y << n2) | (x >> n), b2, d, z);
        if (++pA == pE) { y = x; break; }
        y = *pA;
        a.digitdiv(z, (x << n2) | (y >> n), b2, d, z);
      } while (++pA != pE);
      a.digitdiv(z, y << n2, b2, d, z);
      return z >> n2;
    }
#endif
  }
}
◇
```

Macro referenced in 409.

Laufzeit. Sei $x \in$ Natural und $a \in$ Digit, dann $x\%a \in \mathcal{O}(\mathcal{L}(x))$.

Zuweisungsoperatoren

⟨ assign operator/= for Naturals 161 ⟩ ≡

```
inline Natural& Natural::operator/=(const Natural& a)
// Algorithm:  c := c /= a
// Input:      a,c in Natural where not a = 0.
// Output:     c in Natural such that c := [c/a] ||
{
  Natural t(size+DELTA, ' ');
  div(*this, a, t);
  return *this;
}
◇
```

Macro referenced in 402b.

Laufzeit. Seien $x, y, z_0, z_1 \in$ Natural, dann x`/=`$y \sim$ `div(`x`,`y`,`z_0`,`z_1`)`.

⟨ assign operator%= for Naturals 162a ⟩ ≡
```
inline Natural& Natural::operator%=(const Natural& a)
// Algorithm:  c := c %= a
// Input:      a,c in Natural where not a = 0.
// Output:     c in Natural such that c := c - [c/a]*a ||
{
  Natural t(size+DELTA, ' ');
  ::div(*this, a, t, *this);
  return *this;
}
```
◇
Macro referenced in 402b.

Laufzeit. Seien $x, y, z_0, z_1 \in$ Natural, dann x%=$y \sim$ div(x,y,z_0,z_1).

⟨ assign operator/= of a Natural with a Digit 162b ⟩ ≡
```
inline Natural& Natural::operator/=(const Digit a)
// Algorithm:  c := c /= a
// Input:      c in Natural, a in Digit where not a = 0.
// Output:     c in Natural such that c := [c/a] ||
{
  mod_div(a);
  return *this;
}
```
◇
Macro referenced in 402b.

Laufzeit. Sei $x \in$ Natural und $a \in$ Digit, dann x/=$a \sim x$.mod_div(a).

⟨ assign operator%= of a Natural with a Digit 162c ⟩ ≡
```
inline Digit Natural::operator%=(const Digit b)
// Algorithm:  c := a %= b
// Input:      a in Natural, b in Digit where not b = 0.
// Output:     c in Digit, a in Natural such that c = a - [a/b]*b, a = c ||
{
  Digit c = (*this)%b;
  *this = c;
  return c;
}
```
◇
Macro referenced in 402b.

Laufzeit. Sei $x \in$ Natural und $a \in$ Digit, dann x%=$a \sim x$%a.

2.14.3 Newton-Iteration

Das Ziel dieses Abschnittes ist die Implementierung der Newton-Iteration. Für eine mathematisch korrekte Einführung in die Newton-Iteration verweise ich bespielsweise auf das Buch [49] von Josef Stoer.

Die Newton-Iteration ist eine Methode zur näherungsweisen Nullstellensuche einer gegebenen differenzierbaren Funktion $f(x)$. Die Idee dabei ist, einen jeweils neuen Näherungswert durch die Nullstelle der Tangente an der Kurve von $f(x)$ des vorherigen Näherungswertes anzugeben.

Iterationsvorschrift:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Eine entscheidende Bedingung ist, daß die erste Ableitung der gegebenen Funktion $f(x)$ bekannt sein muß, um die Newton-Iteration überhaupt durchführen zu können. Zusätzlich ist zu beachten, daß das Verfahren in manchen Fällen nicht oder nur sehr langsam konvergiert, nämlich dann, wenn der Startwert x_0 zu weit von der gesuchten Nullstelle entfernt ist. Genaugenommen wird vom Startwert x_0 die folgende Konvergenzbedingung verlangt:

$$\left| \frac{f(x_0) \cdot f''(x_0)}{f'(x_0)^2} \right| < 1.$$

Der Vorteil der Newton-Iteration ist ihre quadratische Konvergenz, was nichts anderes bedeutet, als daß sich die Anzahl der korrekten Stellen nach jeder Iteration verdoppelt. Daher können wir durch die folgende Funktion sowohl die notwendigen Längen der Zahlen als auch die Anzahl der erforderlichen Iterationen ermitteln:

⟨ getting sizes for quadratic convergence algorithm 163a ⟩ ≡

```
size_t* NumberBase::quad_convergence_sizes(const size_t a, size_t& b) const
// Algorithm:  s := c.quad_convergence_sizes(a, b)
// Input:      c in NumberBase, a,b in size_t where a >= 2.
// Output:     b in size_t, s in size_t^b such that b = log2(a)+2,
//             s[i] = [a/2^i]+epsilon, 0 <= i < b ||
{
  CONDITION(a >= 2);

  size_t* s = NOTHROW_NEW size_t[log2(a)+2];
  if (!s) errmsg(2, "(quad_convergence_sizes)");
  size_t i = 1;
  size_t j = a+2;
  s[0] = j; s[1] = j /= 2;
  do { ++j; s[++i] = j /= 2; } while (j > 1);
  b = i+1;

  CONDITION(b >= 2);

  return s;
}
```

◇

Macro referenced in 398b.

Laufzeit. Sei $n \in$ NumberBase und $t_0, t_1 \in$ `size_t`, dann $n.(t_0,t_1) \in \mathcal{O}(\log_2(t_0))$.

Wir suchen die Nullstelle der Funktion $f : x \mapsto x^{-1} - b$, um die Division a/b zweier Naturals a und b durchführen zu können. Dabei ist die Nullstelle der Funktion $f(x)$ ist nichts anderes als der Kehrwert von b. Diesen Kehrwert $1/b$ multiplizieren wir schließlich mit a, um den Quotienten von a/b zu erhalten. Die Newton-Iteration lautet somit:

$$x_{n+1} = 2x_n - bx_n^2, \quad n \in \mathbb{N}.$$

Bei der Implementierung der Division verwenden wir eine interne Festkommadarstellung, deren Länge je nach Iteration durch die vorherige Funktion `quad_convergence_sizes` gesteuert wird.

⟨ division of two Naturals 163b ⟩ ≡

```
void Natural::div(const Natural& a, Natural b, Natural& r)
// Algorithm:  c.div(a, b, r)
// Input:      a,b,r in Natural where not b = 0.
// Output:     c,r in Natural such that c = [a/b], r = a - c*b ||
{
  NATURALCONDITION(a);
  NATURALCONDITION(b);
  NATURAL_FOR_CHECK(_b, b);

  const size_t sA = a.size;
  const size_t sB = b.size;
  if (sB == 1) {
    const Digit x = *b.p;
    if (x == 0) b.errmsg(4, "(div)");
    *this = a / x;
    return;
  } else if (sB < NEWTON_DIV_MARK2 || sA < sB+NEWTON_DIV_MARK1) {
    Natural t(sA+DELTA, ' ');
    ::div(a, b, *this, t);
    return;
  }

  CONDITION(b.length() >= 2 && a.length() > b.length());

  Natural* a2 = 0;
  if (this == &a) {
    a2 = new Natural(a); // if this == &a
    if (!a2) a.errmsg(2, "(div)");
  }
  const size_t l = BETA-1-size_t(log2(b.highest()));
  b <<= l;
  Digit q,q2 = b.highest();
  if (q2 == ~(GAMMA/2)) q = GAMMA;
  else digitdiv(~(GAMMA/2), 0, q2, q, q2);
  Natural y = q;
  size_t m;
  size_t* s = quad_convergence_sizes(a.size-sB+3, m);
  size_t sY = s[--m];
  do {
    const size_t k = s[--m];
    *this = y*y;
    if (size > k) this->rmove(size-k);
    if (sY < sB) { b.size = sY+1; *this *= b; b.size = sB; }
    else *this *= b;
    *this >>= (size-k)*BETA - 1;
    y <<= (k-sY)*BETA + 1; y -= *this;
    sY = k;
  } while (m);
  sY += 2;
  *this = y*y; this->rmove(size-sY-2);
  *this *= b; *this >>= (size-sY)*BETA - 1;
  y <<= 1+2*BETA; y -= *this;
  if (a2) *this = y * (*a2);
  else *this = y * a;
  *this >>= (sB+sY)*BETA-l-1;
  delete[] s;
```

```
  b >>= l;
  y = (*this) * b;
  if (a2) r = (*a2) - y;
  else r = a - y;
  if (r == b) {
    r = 0; ++(*this);
  }
  delete a2;

  CONDITION(r < _b);
  NATURALCONDITION(*this);
}
◇
```

Macro referenced in 409.

Laufzeit. Seien $x, y, z_0, z_1 \in \mathsf{Natural}$, dann

$$z_0\texttt{.div}(x\texttt{,}y\texttt{,}z_1) \begin{cases} \sim \texttt{div} : \mathsf{Natural}^2 \to \mathsf{Natural}^2 , & \mathcal{L}(x) < \mathcal{L}(y) + \texttt{NEWTON\_DIV\_MARK1} \\ & \vee\ \mathcal{L}(y) < \texttt{NEWTON\_DIV\_MARK2} \\ \in \log_2(\mathcal{L}(x) - \mathcal{L}(y)) \cdot \mathcal{O}(\texttt{mul} : \mathsf{Natural}^2 \to \mathsf{Natural}) , & \mathcal{L}(x) \geq \mathcal{L}(y) + \texttt{NEWTON\_DIV\_MARK1} \\ & \wedge\ \mathcal{L}(y) \geq \texttt{NEWTON\_DIV\_MARK2} \end{cases}$$

Den Trade-Off-Point für den Divisionsalgorithmus anhand der Newton-Iteration wurde empirisch ermittelt:

⟨ trade off points for the newton-iteration of the division algorithm 165a ⟩ ≡

```
const size_t NEWTON_DIV_MARK1 = 500;
const size_t NEWTON_DIV_MARK2 = 2500;       // >= 2
◇
```

Macro referenced in 409.

2.14.4 Splitting

Eine Zahl $x \in \mathsf{Natural}$ wird durch die Abbildung

$$x\texttt{.split} : \texttt{size\_t} \times \mathsf{Natural}^2 \to \mathsf{Natural}^2 : (n, a, b) \mapsto \left(\left\lfloor \frac{x}{2^{\beta n}} \right\rfloor , -2^{\beta n} \cdot \left\lfloor \frac{x}{2^{\beta n}} \right\rfloor \right)$$

in einen oberen und einen unteren Teil zerlegt. Durch das interne 2^β-System unserer zu zerlegenden Naturals bereitet das Splitting keine Schwierigkeiten:

⟨ splits a Natural 165b ⟩ ≡

```
void Natural::split(const size_t n, Natural& a, Natural& b) const
// Algorithm:  c.split(n, a, b)
// Input:      a,b,c in Natural, n in size_t where not a = b.
// Output:     a,b in Natural such that a = [c/2^(BETA*n)], b = c - a*2^(BETA*n) ||
{
  NATURALCONDITION(*this);

  Digit* pA = a.p;
  Digit* pB = b.p;
  size_t sT = size;
  if (pA == pB) a.errmsg(5, "(split)");
  if (sT <= n) { b = *this; a = 0; return; }
```

```
  if (n == 0) { a = *this; b = 0; return; }

  const Digit* pT = p;
  if (pT == pB) {
    sT -= n;
    pA = a.setsize(sT);
    const Digit* pE = pB+sT;
    MOVE(pA, pB, pB, pE);
    b.p = pB; b.size = n;
  } else if (pT == pA) {
    pB = b.setsize(n);
    pA += sT;
    Digit* pE = pA-n;
    COPY(pB, pE, pE, pA);
    pE -= n; sT -= n;
    MOVE_BACKWARD(pA, pE, pE, pT);
    a.p = pA; a.size = sT;
    FILL_ZERO(pE, pA);
  } else {
    sT -= n;
    pA = a.setsize(sT);
    const Digit* pE = pT+sT;
    COPY(pA, pT, pT, pE);
    pB = b.setsize(n);
    pE += n;
    COPY(pB, pT, pT, pE);
  }
  b.normalize();

  NATURALCONDITION(a);
  NATURALCONDITION(b);
}
◇
```

Macro referenced in 409.

Laufzeit. Seien $x, y, z \in$ Natural und $t \in$ size_t, dann x.split(t,y,z)$\in \mathcal{O}(\mathcal{L}(x))$.

2.15 Wurzelberechnung

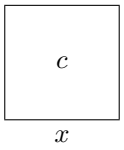
2.15.1 Einführung zur Quadratwurzel

Wurzeliteration von Heron

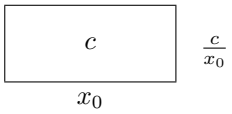
Ein bekanntes Verfahren zur Berechnung der Quadratwurzel ist die Wurzeliteration von Heron (um 60 n.Chr.): Wir schreiben $c \in \mathbb{R}_{>0}$ als Produkt $c = a_0 b_0$ mit $a_0, b_0 > 0$ und $a_0 < \sqrt{c} < b_0$ und iterieren $b_{n+1} = \frac{a_n + b_n}{2}$ und $a_{n+1} = \frac{c}{b_{n+1}}$ für $n \in \mathbb{N}$, solange $a_n \neq b_n$. Schließlich gilt dann $\sqrt{c} = a_n$.

Geometrische Deutung

Die geometrische Deutung dieses Verfahrens ist:



Gesucht ist die Seitenlänge $x \in \mathbb{R}_{>0}$ eines Quadrates mit dem Flächeninhalt $c \in \mathbb{R}_{>0}$.



Wenn wir mit einem Näherungswert $x_0 \in \mathbb{R}_{>0}$ als Länge eines Rechtecks beginnen, das denselben Flächeninhalt $c \in \mathbb{R}_{>0}$ wie das vorgegebene Quadrat hat, so muß die Breite gleich $\frac{c}{x_0}$ sein.

Für den Umfang des Rechtecks gilt $U = 2x_0 + 2\frac{c}{x_0}$, womit wir nach Umformung der Gleichung den besseren Näherungswert

$$x_1 := \frac{U}{4} = \frac{1}{2}\left(x_0 + \frac{c}{x_0}\right)$$

erhalten. Es ergibt sich daraus die allgemeine Formel

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{c}{x_n}\right), n \in \mathbb{N}.$$

Newton-Iteration

Die Newton-Iteration beruht auf anderen mathematischen Theorien, weil hier die Nullstelle der Funktion $f : x \mapsto x^2 - c$ gesucht wird. Aus algorithmischer Sicht aber ist die Rekursionsformel

$$x_{n+1} = x_n - \frac{x_n^2 - c}{2x_n}, n \in \mathbb{N}$$

identisch mit dem Heronschen Verfahren.
Zu Anfang dieser Darstellung beschränken wir uns auf das Wurzelziehen eines Digits, das heißt wir betrachten $\lfloor\sqrt{c}\rfloor$ mit $c \in$ Digit.

```
Digit heron(const Digit c)
{
  Digit a = c;
  for (Digit b = 1; a > b; b = c/a) a = (a+b)/2;
  return a;
}
```

Dieser Algorithmus läßt sich noch soweit optimieren, daß durch eine gute Vorabschätzung weniger Iterationsschritte nötig sind und dadurch auch weniger Divisionen durchgeführt werden müssen. Die Konvergenz dieses Verfahrens ist quadratisch und somit besonders für große Zahlen wichtig. Dennoch ist es die Division, die einen Rechner viel Rechenzeit kostet, und damit in diesem Algorithmus sehr hinderlich ist.

Handschriftliches Wurzelziehen

Eine andere Variante bietet das handschriftliche Wurzelziehen. Wir werden diese Methode für das Dezimalsystem anhand eines kurzen Beispieles darstellen.

Beispiel. Wollen wir eine Quadratwurzel berechnen, so unterteilen wir unsere Zahl vor dem Komma von rechts nach links in Gruppen zu je zwei Ziffern und führen dasselbe nach dem Komma von links nach rechts durch, was wir je nach gewünschter Genauigkeit mit beliebig vielen Nullen so fortsetzen können. Die Berechnung selbst erfolgt jeweils nach der Anzahl der Zweiergruppen mittels der Formel:
$(a+b+c+\ldots)^2 = a^2 + (2a+b)b + (2a+2b+c)c + \ldots$
Nun betrachten wir die erste Gruppe (8) und ziehen aus der nächstliegenden ganzen Quadratzahl (4) die Wurzel (2). Hiermit ist die erste Ziffer des Endergebnisses (28,284) gefunden. Um nun die zweite Ziffer zu finden, subtrahieren wir unsere Quadratzahl (4) von der ersten Gruppe, holen zur Differenz (4) die zweite Gruppe (00) herunter und dividieren die so erweiterte Differenz (400) nach Abstrich der letzten Ziffer durch das Doppelte von 2 (4). Versuchsweise ermitteln wir dann das ganzzahlige Divisionsergebnis ($\lfloor 400/4 \rfloor = 10 \rightarrow 9$). Wenn das Produkt aus dem Divisor und dem Quotienten ($49 \cdot 9 = 441$) größer ist als die erweiterte Differenz (400), dann ist der Divisor durch die passende kleinere Ziffer (8) zu ersetzen. Um die dritte Ziffer des Ergebnisses zu finden, subtrahieren wir das Produkt (384) von der erweiterten Differenz (400), holen zur neuen Differenz (16) die dritte Gruppe (00) herunter und verfahren ebenso weiter...

$$
\begin{array}{rl}
\sqrt{\;8|00{,}00|00|00\;} & = 28{,}284\ldots \\
-4 & \rightarrow \lfloor \frac{40}{2\cdot 2} \rfloor = 10 \\
\hline
\;4\;00 & \rightarrow 49 \cdot 9 = 441 \\
-3\;84 & \rightarrow 48 \cdot 8 = 384 \\
\hline
\;\;16\;00 & \rightarrow \lfloor \frac{160}{2\cdot 28} \rfloor = 2 \\
-\;11\;24 & \rightarrow 562 \cdot 2 = 1124 \\
\hline
\;\;\;4\;76\;00 & \rightarrow \lfloor \frac{4760}{2\cdot 282} \rfloor = 8 \\
-4\;51\;84 & \rightarrow 5648 \cdot 8 = 45184 \\
\hline
\;\;\;\;24\;16\;00 & \rightarrow \lfloor \frac{24160}{2\cdot 2824} \rfloor = 4 \\
-\;22\;59\;36 & \\
\hline
\;\;\;\;\;1\;56\;64 & \ldots
\end{array}
$$

2.15.2 Binärer Algorithmus zur Berechnung der Quadratwurzel

Quadratwurzel eines Digits

Dieses Rechenverfahren ist in seiner dezimalen Form wesentlich schlechter als das Heronsche Verfahren, da wir für jede Ziffer eine Division und mindestens eine Multiplikation benötigen. Wenn wir es aber auf das binäre System übertragen, dann können wir wesentlich effizienter vorgehen und kommen dabei nur mit elementaren Operationen aus. Damit erhalten wir folgende Gestalt für das binäre Wurzelziehen[5]:

⟨ square root of a Digit 168 ⟩ ≡

```
Digit sqrt(Digit a)
// Algorithm:  c := sqrt(a)
// Input:      a in Digit.
// Output:     c in Digit such that c = [sqrt(a)] ||
{
  Digit b = Digit(1) << (BETA-2);
  Digit c = Digit(1) << (BETA-1) | Digit(1) << (BETA-2);

  do {
    if (a >= b) { a -= b; b |= c; }
    c >>= 2;
    b >>= 1;
    b ^= c;
  } while (c != 3);
```

[5] Sebastian Wedeniwski, "Wurzel ziehen", Amiga Magazin Faszination Programmieren, Sonderheft 2/93, 71-72.

```
  if (a >= b) b |= 2;
  return b >>= 1;
}
◇
```

Macro referenced in 398b.

Laufzeit. Sei $a \in$ Digit, dann `sqrt(`a`)`$\in \mathcal{O}(4\beta - 3)$.

Hierbei ist die Konvergenz zwar nur linear; da aber nur elementare Grundoperationen benötigt werden, hat der Rechner lediglich eine Laufzeit von $4\beta - 3$ Taktzyklen (ohne Pipelining). Der im Pentium-Coprozessor festverdrahtete Quadratwurzelalgorithmus benötigt dagegen konstante 70 Taktzyklen, womit sich also auch hier der optionale Aufruf `Digit(sqrt(double(c)))` anbietet. Wir wollen diesen internen `double` Typ jedoch nicht benutzen, sondern gemäß unserer Forderung nach einem Grunddatentyp bei den Digits bleiben.
Dieser Algorithmus `sqrt` kann dann auch ohne Mehraufwand den verbleibenden **Rest**[6] exakt zurückgeben:

⟨ square root and remainder of a Digit 169a ⟩ ≡

```
void sqrt(Digit a, Digit& x, Digit& y)
// Algorithm:  sqrt(a, x, y)
// Input:      a in Digit.
// Output:     x,y in Digit such that x = [sqrt(a)], y = a - x^2 ||
{
  Digit b = Digit(1) << (BETA-2);
  Digit c = Digit(1) << (BETA-1) | Digit(1) << (BETA-2);

  do {
    if (a >= b) { a -= b; b |= c; }
    c >>= 2;
    b >>= 1;
    b ^= c;
  } while (c != 3);

  if (a >= b) { a -= b; b |= 2; }
  y = a; x = b >>= 1;
}
◇
```

Macro referenced in 398b.

Laufzeit. Seien $a, d_0, d_1 \in$ Digit, dann `sqrt(`a`,`d_0`,`d_1`)`$\in \mathcal{O}(4\beta)$.

Quadratwurzel zweier Digits

Das Resultat einer Quadratwurzel paßt auch noch in ein Digit, wenn das Argument aus zwei Digits besteht. Dabei müssen nicht alle Variablen des Algorithmus verdoppelt werden.
Wir teilen den Algorithmus in zwei Bereiche auf und verdoppeln nur das Argument mit dem verbleibenden Rest:

⟨ square root and remainder of a double Digit 169b ⟩ ≡

[6] Unter einem Rest r bei der Quadratwurzelberechnung von a verstehen wir $r := a - \lfloor\sqrt{a}\rfloor^2$.

```
Digit sqrt(Digit a, Digit b, Digit& x, Digit& y)
// Algorithm:  c := sqrt(a, b, x, y)
// Input:      a,b in Digit.
// Output:     c,x,y in Digit such that c = [sqrt(a*2^BETA+b)],
//             x*2^BETA+y = a*2^BETA+b - c^2 ||
{
  const Digit HIBIT = Digit(1) << (BETA-1);
  Digit c = Digit(1) << (BETA-2);
  Digit d = Digit(1) << (BETA-1) | Digit(1) << (BETA-2);

  do {
    if (a >= c) { a -= c; c |= d; }
    d >>= 2;
    c >>= 1;
    c ^= d;
  } while (d != 3);
  if (a >= c) { a -= c; c |= 2; }
  c >>= 1;

  d = Digit(1) << (BETA-1) | Digit(1) << (BETA-2);
  Digit e = Digit(1) << (BETA-2);
  do {
    if (a > c) {
      a -= c;
      if (b < e) --a;
      b -= e;
      e |= d;
    } else if (a == c && b >= e) { a = 0; b -= e; e |= d; }
    d >>= 2;
    e >>= 1;
    if (c&1) e |= HIBIT;
    c >>= 1;
    e ^= d;
  } while (d);
  x = a; y = b;
  return e;
}
◇
```

Macro referenced in 398b.

Laufzeit. Seien $a_0, a_1, d_0, d_1 \in$ Digit, dann `sqrt(`a_0`,`a_1`,`d_0`,`d_1`)`$\in \mathcal{O}(11\beta + 2)$.

Vereinfachter Funktionsaufruf

Der einfache Quadratwurzelaufruf ist nur noch ein Spezialfall der gerade vorgestellten Quadratwurzelberechnung zweier Digits:

⟨ square root of a double Digit 171 ⟩ ≡
```
inline Digit sqrt(Digit a, Digit b)
// Algorithm:  c := sqrt(a, b)
// Input:      a,b in Digit.
// Output:     c in Digit such that c = [sqrt(a*2^BETA+b)] ||
{
  Digit x,y;
  return sqrt(a, b, x, y);
}
```
◇
Macro referenced in 397d.

Laufzeit. Seien $a_0, a_1, d_0, d_1 \in$ Digit, dann `sqrt(`a_0`,`a_1`)` $\sim$ `sqrt(`a_0`,`a_1`,`d_0`,`d_1`)`.

Quadratwurzel eines Naturals

Um dieses Verfahren auf die Natural-Arithmetik anwenden zu können, betrachten wir zuersteinmal die einfache Variante, nämlich die direkte Übertragung:

```
Natural binsqrt1(Natural a)
{
  Natural b(4, a.size);
  Natural c(3, a.size);

  do {
    b >>= 1;
    b ^= c;
    if (a >= b) {
      a -= b;
      b |= c;
    }
    c >>= 2;
  } while (c != 0);
  return b >>= 1;
}
```

Laufzeitverbesserung:

`b >>= 1;`	$\longrightarrow$	$\mathcal{O}(\mathcal{L}(b))$	$\overset{(2.15.3)}{\longrightarrow}$	$\mathcal{O}(\mathcal{L}(b))$	$\overset{(2.15.4)}{\longrightarrow}$	$\mathcal{O}(\beta)$
`b ^= c;`	$\longrightarrow$	$\mathcal{O}(\mathcal{L}(b))$	$\longrightarrow$	$\mathcal{O}(1)$		
`if (a >= b) {`	$\longrightarrow$	$\mathcal{O}(1)$	$\longrightarrow$	$\mathcal{O}(1)$		
`a -= b;`	$\longrightarrow$	$\mathcal{O}(\mathcal{L}(b))$	$\longrightarrow$	$\mathcal{O}(\mathcal{L}(b))$		
`b |= c;`	$\longrightarrow$	$\mathcal{O}(\mathcal{L}(b))$	$\longrightarrow$	$\mathcal{O}(1)$		
`c >>= 2;`	$\longrightarrow$	$\mathcal{O}(\mathcal{L}(c))$	$\longrightarrow$	$\mathcal{O}(1)$		
`} while (c != 0);`	$\longrightarrow$	$\mathcal{O}(1)$	$\longrightarrow$	$\mathcal{O}(1)$		
`return b >>= 1;`	$\longrightarrow$	$\mathcal{O}(\mathcal{L}(b))$	$\longrightarrow$	$\mathcal{O}(\mathcal{L}(b))$		

Ein Manko hierbei ist, daß viele unnötige Nullen mitoperieren. Für einen geringeren Speicherverbrauch und eine höhere Übersichtlichkeit bietet sich folgende schnellere Alternative an:

```
Natural binsqrt2(Natural a)
{
  Natural b(1, a.size);
  size_t  c = a.size*BETA + 1;

  do {
    c -= 2;
    b.setbit(c); b >>= 1; b.clearbit(c);
    b.normalize();
    if (a >= b) { a -= b; b.setbit(c); }
  } while (c > 1);
  return b >>= 1;
}
```

2.15.3 Implementation des binären Algorithmus

Um aber einen wirklich leistungsfähigen Algorithmus zu erhalten, sollten wir eine Gliederung der Quadratwurzelberechnung vornehmen. Dabei ist es wichtig, daß unser Algorithmus (wegen seiner linearen Konvergenz) eine sehr hohe Anzahl an Schleifendurchläufen hat. Deshalb erzielen wir eine umso höhere Effizienz, je elementarer wir den Schleifeninhalt lösen.
Die Aufteilung wird in vier Abschnitte gegliedert:

1. Initialisierung
2. Subtraktion
3. Vergleichsoperation
4. Schiebeoperation

1. Initialisierung

In ANSI wird ein einzelner Bitzugriff – wie in `binsqrt2` (siehe Kapitel 2.15.2, Seite 171) – nicht von Haus aus unterstützt und erfordert damit für Digits mehr Rechenaufwand als die bitweisen logischen Operationen aus `binsqrt1` (siehe Kapitel 2.15.2, Seite 171). Also versuchen wir `binsqrt1` zu optimieren. Bei einer genaueren Betrachtung des Naturals `c` aus der Funktion `binsqrt1` fällt auf, daß `c` nur aus zwei Bits besteht, die in einem Digit nach rechts rotieren[7], und zwar um zwei Stellen je Durchlauf. Somit können wir nun `c` durch einen Positionszeiger `pC` und ein mit den zwei Bits belegtes statisches Feld `ax` ersetzen.

⟨ initialize the static variables for the calculation of the positive square root 172a ⟩ ≡
```
static Digit ax[BETA] = { Digit(1) << (BETA-2),
                          Digit(1) << (BETA-3) | Digit(1) << (BETA-4),
                          0 };
if (ax[2] == 0) {
  size_t i = 2;
  do ax[i] = ax[i-1] >> 2; while (++i < BETA/2);
  ax[i] = Digit(1) << (BETA-1);
  while (++i < BETA) ax[i] = ax[i-1] >> 2;
}
```
◇
Macro referenced in 174c.

Also wird die Anweisung `b ^= c` durch

⟨ set the bits in the result of the positive square root 172b ⟩ ≡
```
*pC ^= ax[s];
```
◇
Macro referenced in 174c.

ersetzt und `c >>= 2` durch die Konstruktion

⟨ rotates the bits for the calculation of the positive square root 172c ⟩ ≡
```
if (++s == BETA/2) { s = 0; ++pC; ++pD; }
```
◇
Macro referenced in 174c.

[7] Bei einer Rotation nach rechts werden alle Bits nach rechts verschoben, das Bit an der Position 0 wird an die höchstwertige Position gesetzt.

2. Subtraktion

Die Subtraktion ist äquivalent zur Funktion `subpos` (siehe Kapitel 2.8.3, Seite 64). Dabei werden die beiden Zeiger `pDif` und `pSub` aber um eine Position weiter nach links übergeben.

⟨ internal function `sqrtsub` 173a ⟩ ≡

```
inline void sqrtsub(const Digit* pT, Digit* pDif, const Digit* pSub)
// Algorithm:  sqrtsub(r, s, t)
//             Let a,b in Natural.
// Input:      r,s in [a.root, a.p+L(a)] where r < s,
//             t in [b.root, b.p+L(b)] where t-(s-r) also in [b.root, b.p+L(b)],
//             where R(a) > L(b), [r, s] >= [t-(s-r), t].
// Output:     [r, s] := [r, s] - [t-(s-r), t] ||
{
  do {
    Digit c = *pDif;
    Digit d = *pSub;
    *pDif -= d;
    if (c < d)
      do {
        c = *--pDif;
        d = *--pSub;
        *pDif -= d+1;
      } while (c <= d);
    --pDif;
  } while (pT <= --pSub);
}
```

◇

Macro referenced in 409.

3. Vergleichsoperation

Wir zerlegen den `operator>=` in zwei Fälle, indem wir seine Länge überprüfen. Dadurch wir nur bei gleicher Länge ein tatsächlicher Vergleich durchgeführt.

⟨ compares two Naturals in the binary square root algorithm 173b ⟩ ≡

```
if (sT == sY) {    // (a >= y)?
  Digit* pA = pT;
  Digit* pB = pY;
  Digit x,y;
  do { x = *pA++; y = *pB++; } while (x == y);
  if (x > y) {
    sqrtsub(pY, pD, pC);
    while (*pT == 0) { ++pT; --sT; }
    *pC |= ax[s+BETA/2];
  }
} else if (sT > sY) {
  sqrtsub(pY, pD, pC);
  while (*pT == 0) { ++pT; --sT; }
  *pC |= ax[s+BETA/2];
}
```

◇

Macro referenced in 174c.

Diese Vergleichsoperation würde ohne eine zusätzliche Konvention nicht terminieren. Deshalb hängen wir an unsere beiden Naturals a und y rechts jeweils ein zusätzliches Digit an und setzen dieses bei a auf 1 und bei y auf 0.

⟨ convention for Naturals in the binary square root algorithm 174a ⟩ ≡

```
pT[sT] = 1;    // because compare
```

◇

Macro referenced in 174c.

Dadurch terminiert dieser Algortihmus auch für die Gleichheit von a und y.

4. Schiebeoperation

Bei der Realisierung der Schiebeoperation müssen wir leider manuell vorgehen:

⟨ divides a Natural by 2 in the binary square root algorithm 174b ⟩ ≡

```
Digit* pA = pC;
Digit x = *pA;
x >>= 1;
while (pA != pY) {        // y >>= 1
  Digit y = *--pA;
  pA[1] = x | ((y&1) << (BETA-1));
  x = y >> 1;
}
*pA = x;
if (x == 0) { ++pY; --sY; }
```

◇

Macro referenced in 174c.

Wir durchlaufen unser Natural von hinten nach vorne und schieben es dabei Digit für Digit nach rechts, wobei wir das untere Bit des darüberliegenden Digits mitübertragen. Diese Methode ist jedoch aufgrund unserer Einhaltung des ANSI-C++ Standards ineffizient, weil der ganze Vorgang zum Beispiel auch durch Flagzugriffe mit einer Schiebeoperation durchgeführt werden könnte.

Laufzeitanalyse

Die Verschiebe- und Vergleichsoperation wird $\mathcal{L}(a) \cdot \beta/2$ mal durchlaufen, die Subtraktionsoperation hingegen nur etwa halb so oft. Bei $\sqrt{8 \cdot 10^{96}}$ auf einer 32-Bit Maschine würden wir beispielweise 160mal verschieben und vergleichen und 80mal subtrahieren.

Implementation

Eine komplette Implementation sieht demnach folgendermaßen aus:

⟨ square root and remainder of a Natural 174c ⟩ ≡

```
void sqrt(const Natural& b, Natural& c, Natural& d)
// Algorithm:  sqrt(b, c, d)
// Input:      b in Natural.
// Output:     c,d in Natural such that c = [sqrt(b)], d = b - c^2 ||
{
  NATURALCONDITION(b);
```

```
  NATURAL_FOR_CHECK(_b, b);

  ⟨ special cases for the calculation of the positive square root 175 ⟩
  ⟨ initialize the static variables for the calculation of the positive square root 172a ⟩
  Natural a(b, 1);
  size_t sT = a.size;
  Natural y((Digit)0, sT);      // a.size >= 2!
  size_t sY       = --sT;
  Digit* pY       = y.p;
  Digit* pT       = a.p;
  Digit* pC       = pY;
  Digit* pD       = pT;
  const Digit* pE = pY+sY;
  size_t s        = 0;
  ⟨ convention for Naturals in the binary square root algorithm 174a ⟩
  do {
    ⟨ set the bits in the result of the positive square root 172b ⟩
    ⟨ compares two Naturals in the binary square root algorithm 173b ⟩
    ⟨ divides a Natural by 2 in the binary square root algorithm 174b ⟩
    ⟨ rotates the bits for the calculation of the positive square root 172c ⟩
  } while (pC < pE);
  y.p = pY; y.size = sY;
  if (sT) { a.size = sT; a.p = pT; }
  else { a.size = 1; a.p = --pT; }
  c = y; d = a;

  CONDITION(d+c*c == _b);
  NATURALCONDITION(c);
  NATURALCONDITION(d);
}
◇
```

Macro referenced in 409.

Laufzeit. Seien $x, y, z \in$ Natural, dann `sqrt(`x`,`y`,`z`)`$\in \mathcal{O}(\frac{3}{8}\beta\mathcal{L}(x)^2)$.

Sonderfälle

Die Variable `sY` enthält für den Fall `sqrt(0)` den Wert 0. Diese Längenangabe ist nicht zulässig und verlangt eine zusätzliche Überprüfung. Wir können dies umgehen, indem wir zu Anfang eine Wurzelberechnung nur für Digits durchführen ($0 \in$ Digit), die dann auch deutlich schneller ist als die entsprechende für Naturals.

⟨ special cases for the calculation of the positive square root 175 ⟩ ≡

```
const size_t sB = b.size;
if (b == 0) { c = d = 0; return; }
else if (sB == 1) {
  Digit c2,d2;
  sqrt(*b.p, c2, d2);
  c = c2; d = d2;
  return;
} else if (sB == 2) {
  Digit d1,d2;
  c = sqrt(*b.p, b.p[1], d1, d2);
  if (d1) {
```

```
    Digit* pD = d.setsize(2);
    *pD = d1; pD[1] = d2;
  } else d = d2;
  return;
}
◇
```

Macro referenced in 174c.

Durch diesen Algorithmus erhalten wir den Rest der Quadratwurzelberechnung ohne Mehraufwand und erfüllen gleichzeitig die Gleichung $b = c^2 + d$. Dies bietet zum Beispiel einen Vorteil bei der Primzahlzerlegung von Lehman (siehe Kapitel 5.7.6, Seite 295).

2.15.4 Binäre Quadratwurzelberechnung ohne Verschiebung

Betrachten wir die Langzahlverschiebung des Resultates b in der Grundform des binären Quadratwurzel-Algorithmus (`binsqrt1`, Kapitel 2.15.2, Seite 171), so fällt auf, daß nur am Ende während der Iteration die Bits modifiziert werden. Das bedeutet aber, daß sich der linear wachsende vordere Teil des Naturals b spätestens nach β Rechtsverschiebungen wiederholt.
Eine einfache Möglichkeit, sich diesen Verschiebungsvorgang vorzustellen, ist diese:

- Wir haben einen Vektor $(y_i)_{i=0}^{\beta-1} \in \mathsf{Natural}^\beta$, wobei $2^i y_i = b$ für $0 \le i < \beta$ ist.
- Eine Rechtsverschiebung von b um $n \in$ `size_t` Bits bedeutet einen Zugriff auf die $y_{n \bmod \beta}$ Komponente mit der Länge $\mathcal{L}(y_{n \bmod \beta}) - \lfloor n/\beta \rfloor$.

Beispiel. Falls wir $\lfloor b/2^{\beta+3} \rfloor$ benötigen, so greifen wir auf die Komponente y_3 mit der Länge $\mathcal{L}(y_3)-1$ zu.

Dafür müssen wir allerdings stattdessen in unserem Algorithmus β Mal ein Bit setzen. Dies ist jedoch ein konstanter Aufwand im Gegensatz zum linearen Aufwand der jetzt eingesparten Rechtsverschiebung.

⟨ square root of a Natural 176 ⟩ ≡

```
void Natural::sqrt(const Natural& a)
// Algorithm:  b.sqrt(a)
// Input:      a in Natural.
// Output:     b in Natural such that b = [sqrt(a)] ||
{
  NATURALCONDITION(a);

  const size_t sA = a.size;
  if (sA == 1) *this = ::sqrt(*a.p);
  else if (sA == 2) *this = ::sqrt(*a.p, a.p[1]);

#ifdef FASTER_BINSQRT
  else if (sA <= 2*BETA) {
    Natural c;
    ::sqrt(a, b, c);
  } else if (sA <= NEWTON_SQRT_MARK) {
    Natural x(sA+2+DELTA, ' ');
    Digit* y[BETA];
    Digit  i;
    size_t j = x.size/2 + 2;

    for (i = 0; i < BETA; ++i) {
```

```
  Digit* pY = y[i] = NOTHROW_NEW Digit[j];
  const Digit* pE = pY+j;
  FILL_ZERO(pY, pE);
  ++y[i];                       // because subpos!
}
--x.size;
const size_t SHIFT_H = BETA - size_t(log2(*a.p))-1;
const size_t SHIFT   = (SHIFT_H&1)? SHIFT_H-1 : SHIFT_H;
x = a << SHIFT;

size_t sX = x.size;
size_t sY = sX;
Digit* pX = x.p;
pX[sX] = 1;

i = Digit(1) << (BETA-2); j = 0;
size_t k = 0;
do {
  y[j][k] |= i;
  if (sX == sY) {
    Digit* pA = pX;
    Digit* pB = y[j];
    Digit x2,y2;
    do { x2 = *pA++; y2 = *pB++; } while (x2 == y2);
    if (x2 > y2) {
      sqrtsub(y[j], pX+k, y[j]+k);
      while (*pX == 0) { ++pX; --sX; }
      size_t j2 = j;
      size_t k2 = k;
      do {
        if (++j2 == BETA) { j2 = 0; --k2; }
        y[j2][k2] |= i; i >>= 1;
        if (i == 0) { i = Digit(1) << (BETA-1); ++k2; }
      } while (j2 != j);
    }
  } else if (sX > sY) {
    if (j) sqrtsub(y[j], pX+k, y[j]+k);
    else sqrtsub(y[j], pX+k+1, y[j]+k);
    while (*pX == 0) { ++pX; --sX; }
    size_t j2 = j;
    size_t k2 = k;
    do {
      if (++j2 == BETA) { j2 = 0; --k2; }
      y[j2][k2] |= i; i >>= 1;
      if (i == 0) { i = Digit(1) << (BETA-1); ++k2; }
    } while (j2 != j);
  }
  y[j][k] &= ~i;

  if (++j == BETA) {
    i = Digit(1) << (BETA-2);
    j = 0; --sY;
  } else if (i < 4) { i = Digit(1) << (BETA-2); ++k; }
  else i >>= 2;
} while ((x.size&1) && (j == BETA/2-1 || k <= x.size/2)
  || (x.size&1) == 0 && (j || k < x.size/2));
```

```
    sX = x.size;
    if ((sX&1) == 0) ++sY;
    pX = setsize(sY);
    Digit* pA = (sX&1)? y[BETA/2-1] : y[BETA-1];
    const Digit* pE = pA+sY;
    COPY(pX, pA, pA, pE);
    *this >>= SHIFT/2 + 1;
    // b = *this, a*2^SHIFT = (b/2)^2 + x,
    // but a != (b/2^(SHIFT/2+1))^2 + x/2^SHIFT
    // because b is not minimal!
    for (i = 0; i < BETA; ++i) delete[] --y[i];
    x.normalize();  // just to be on the safe side
  }
#else
  else if (sA <= NEWTON_SQRT_MARK) {
    Natural c;
    ::sqrt(a, *this, c);
#endif

  } else newton_sqrt(a);

  NATURALCONDITION(*this);
}
◇
```

Macro referenced in 409.

Laufzeit. Seien $x, y \in \mathsf{Natural}$, dann

$$y.\texttt{sqrt}(x)\begin{cases} \sim \texttt{newton\_sqrt}(a)\ , & x > \texttt{NEWTON\_SQRT\_MARK} \\ \in \mathcal{O}(\frac{1}{8}\beta\mathcal{L}(x)^2)\ , & x \leq \texttt{NEWTON\_SQRT\_MARK} \end{cases}.$$

Bemerkung. Dieser Algorithmus läßt sich duch die elementare Zerlegung des Digits in `char` weiter optimieren, wofür die folgende Struktur ein Beispiel sein soll:

```
union SemiDigit {
  unsigned char c[sizeof(Digit)];
  Digit         d;
};
```

Sei $x \in$ `SemiDigit`, so muß man zusätzlich auf die Reihenfolge der Bytes in einem Digit achtgeben, die je nach Architektur verschieden ist. Zur Zeit existieren die folgenden beiden Modelle:

Little Endian. `x.c[0] = 1` ist gleichbedeutend mit `x.d = 1`.

Big Endian. `x.c[0] = 1` ist gleichbedeutend mit `x.d =` $2^{\beta-\beta/\texttt{sizeof}(\mathsf{Digit})}$.

Diese Verbesserungsmöglichkeit werden wir in einer späteren Arbeit noch genauer analysieren.

2.15.5 Divisionsfreie Newton-Iteration

Wenn wir uns zurückerinnern, so war bei der Newton-Iteration aus dem Kapitel 2.15.1 auf der Seite 167 die Langzahldivision der entscheidende Nachteil für die Realisierung des Algorithmus. Dies läßt sich jedoch beheben, wenn wir die Nullstelle der Funktion $f : x \mapsto x^{-2} - c$ mit $c \in \mathsf{Natural}$ bestimmen, denn dann lautet die Newton-Iteration:

$$x_{n+1} = \frac{x_n(3 - cx_n^2)}{2}, n \in \mathbb{N}.$$

Die gesuchte Lösung $\lfloor\sqrt{c}\rfloor$ erhalten wir jetzt durch das Produkt $c \cdot x_k$, wobei für alle $n > k : x_n = x_k$ gilt.

Wir werden eine interne Festkommadarstellung für die Iteration verwenden, weil unsere Zahlen $x_n \leq 1$ für alle $n \in \mathbb{N}$ sind.

Eine einfache Realisierung ist die folgende Umsetzung:

```
Natural sqrt(const Natural& a)
// Algorithm:  b := sqrt(a)
// Input:      a in Natural.
// Output:     b in Natural such that b = [sqrt(a)] ||
{
  Digit q,q2,r,d = sqrt(*a.p, a.p[1]);            // Initialvalue
  a.digitdiv(GAMMA, GAMMA, d, q, q2, r);
  const size_t point = a.size*BETA;
  Natural b = q; b <<= BETA; b |= q2; b <<= point/2-BETA;
  Natural x = 3; x <<= point+2*BETA;
  Natural c,y,z;

  for (size_t n = 2; n < a.size; n *= 2) {        // Newton-Iteration
    sqr(b, c); c >>= point/2-2*BETA;
    z = a*c; z >>= point/2;
    z = x-z;
    c = b*z;
    b = c >> (point+1+2*BETA);
  }
  sqr(b, c); c >>= point/2-2*BETA;
  z = a*c; z >>= point/2;
  z = x-z;
  c = b*z;
  c >>= point;
  b = c*a;
  b >>= point+1+2*BETA;
  return b;
}
```

⟨ trade off points for the newton-iteration of the square root algorithm 179a ⟩ ≡

```
const size_t NEWTON_SQRT_MARK = 500;       // >= 2
```

◇

Macro referenced in 409.

Es müssen jedoch bei der Newton-Iteration nur so viele Nachkommastellen mitoperieren, wie es die Konvergenz erlaubt. Dadurch wird vor allem die Längensteuerung aufwendiger:

⟨ square root of a Natural with newton iteration 179b ⟩ ≡

```
void Natural::newton_sqrt(Natural a)
// Algorithm:  b.newton_sqrt(a)
// Input:      a in Natural where L(a) >= 2.
// Output:     b in Natural such that b = [sqrt(a)] ||
{
  NATURALCONDITION(a);
  CONDITION(a.size >= 2);

  const size_t sA = a.size;
  const size_t l = BETA-1-size_t(log2(a.highest()));
```

```
  a <<= l & (~size_t(0)-1);
  const Digit d = ::sqrt(a.p[0], a.p[1]);
  Digit q,r;
  if (d == Digit(~(GAMMA/2))) q = GAMMA;
  else a.digitdiv(~(GAMMA/2), 0, d, q, r);
  Natural t,b = q;
  size_t m;
  size_t* s = quad_convergence_sizes(sA/2+1, m);
  size_t sB = s[--m];
  do {
    *this = b*b;
    if (sB < sA) a.size = sB+1;
    t = *this * a; a.size = sA;
    const size_t k = s[--m];
    t >>= (t.size-k)*BETA - 1;
    *this = t * b;
    *this >>= (size-k)*BETA - 1;
    b.lmove(k-sB); t = b;
    b <<= 1; b += t; b -= *this; b >>= 1;
    sB = k;
  } while (m);
  *this = b*b; t = *this * a; t >>= (t.size-sB)*BETA - 1;
  *this = t * b; *this >>= (size-sB)*BETA - 1;
  t = b; b <<= 1; b += t; b -= *this;
  *this = b * a;
  *this >>= sA*(BETA/2) + sB*BETA + l/2;
  delete[] s;

  NATURALCONDITION(*this);
}
◇
```

Macro referenced in 409.

Laufzeit. Seien $x, y \in \mathsf{Natural}$, dann $y.\texttt{newton\_sqrt}(x) \in \log_2(\mathcal{L}(x)) \cdot \mathcal{O}(\texttt{mul} : \mathsf{Natural}^2 \to \mathsf{Natural})$.

Vereinfachter Funktionsaufruf

Um die `sqrt`-Funktion in gewohnter Weise aufrufen zu können, gibt es noch die folgende Abbildungsmöglichkeit:

⟨ simple call for square root of a Natural 180 ⟩ ≡

```
inline Natural::Natural(const binder_arguments<Natural, Natural,
                                               Natural_square_root_tag>& a)
{
  get_memory(a.x.size/2+DELTA);
  sqrt(a.x);
}

inline Natural& Natural::operator=(const binder_arguments<Natural, Natural,
                                                          Natural_square_root_tag>& a)
{
  sqrt(a.x);
  return *this;
}
```

```
inline binder_arguments<Natural, Natural, Natural_square_root_tag>
 sqrt(const Natural& a)
// Algorithm:  b := sqrt(a)
// Input:      a in Natural.
// Output:     b in Natural such that b = [sqrt(a)] ||
{
  return binder_arguments<Natural, Natural, Natural_square_root_tag>(a, a);
}
◇
```

Macro referenced in 402b.

Laufzeit. Seien $x, y \in$ Natural, dann `sqrt(`x`)` $\sim$ y`.sqrt(`x`)`.

2.15.6 Beliebige Wurzelberechnung

Hierbei verwenden wir lediglich eine Verallgemeinerung der Funktion $x \mapsto x^2 - a$, indem wir zur Berechnung von $\sqrt[n]{a}$ die Nullstelle der Funktion $f : x \mapsto x^n - a$ für ein gegebenes $a \in$ Natural suchen. Somit erhalten wir die folgende Rekursionsformel als Newton-Iteration:

$$x_{i+1} = x_i - \frac{x_i^n - a}{n \cdot x_i^{n-1}} \qquad \text{für } i \in \mathbb{N}.$$

Nun benötigen wir nur noch eine gute Abschätzung, um die Anzahl der Iterationsschritte gering halten zu können:

⟨ calculates the root of a Natural 181 ⟩ ≡

```
Natural root(const Natural& a, const Digit n)
// Algorithm:  b := root(a, n)
// Input:      a in Natural, n in Digit.
// Output:     b in Natural such that b = [a^(1/n)] ||
{
  if (n == 0) return 1;
  else if (n == 1) return a;
  else if (n == 2) return sqrt(a);
  const Digit k = log2(a)+1;
  const Digit m = n-1;
  const Digit l = k/n;
  Natural b(4);
  if (l >= 2) { b = (4*(k - l*n))/n + 5; b <<= size_t(l)-2; }
  Natural c,q,r;
  while (true) {
    c = pow(b, m);
    div(a, c, q, r);
    if (b == q) return b;
    else if (b < q) {
      c = pow(++b, n);
      if (a < c) --b;
      return b;
    }
    b *= m;
    b += q;
    b /= n;
  }
}
◇
```

Macro referenced in 409.

Laufzeit. Seien $x, y \in \mathsf{Natural}$ und $a \in \mathsf{Digit}$, dann

$$\texttt{root}(x,a) \in \begin{cases} \mathcal{O}(1)\,, & a = 0 \\ \mathcal{O}(\mathcal{L}(x))\,, & a = 1 \\ \mathcal{O}(\texttt{sqrt} : \mathsf{Natural} \to \mathsf{Natural})\,, & a = 2 \\ \log_2(\mathcal{L}(x)) \cdot (\mathcal{O}(\texttt{pow} : \mathsf{Natural} \times \mathsf{Digit} \to \mathsf{Natural}) + \mathcal{O}(\texttt{div} : \mathsf{Natural}^2 \to \mathsf{Natural}^2))\,, & a > 2 \end{cases}.$$

2.16 Zufallszahl

Wir werden uns nicht mit der Generierung von Zufallszahlen beschäftigen, weil wir sie im Digitbereich mit der Standardfunktion `rand()` aus `<stdlib.h>` erzeugen können. Um auch eine zufällige Langzahl zu erhalten, brauchen wir nur noch unsere Digit-Zufallszahlen aneinanderzuhängen, was durch Verschiebungen sehr einfach möglich ist:

⟨ calculates a Natural random number 182 ⟩ ≡

```
// Correct ?!
#ifndef RAND_MAX
# define RAND_MAX ((unsigned(1) << (CHAR_BIT*sizeof(int)-1)) - 1)
#endif

void Natural::rand(size_t n)
// Algorithm:  a.rand(n)
// Input:      n in size_t.
// Output:     a in Natural such that a < 2^n (random number) ||
{
  NATURALCONDITION(*this);

  const size_t sT = max(n/BETA + (n%BETA != 0), size_t(1));
  const size_t s  = min(1+size_t(log2(Digit(RAND_MAX))), BETA);
  Digit* pT = setsize(sT);
  pT += sT;
  size_t i = 0;
  Digit k = 0;
  while (n >= s) {
    const Digit j = (Digit)::rand();
    k |= j << i;
    const size_t l = BETA-i;
    if (s >= l) {
      *--pT = k;
      k = j >> l;
      i -= BETA;
    }
    i += s; n -= s;
  }
  if (n) {
    const Digit k2 = Digit(1) << n;
    const Digit j = Digit(::rand()) & (k2-1);
    *--pT = k | (j << i);
    const size_t l = BETA-i;
    if (s > l) *--pT = j >> l;
  } else *--pT = k;
  normalize();

  NATURALCONDITION(*this);
```

```
}
```
◇

Macro referenced in 409.

Laufzeit. Sei $x \in$ Natural und $a \in$ Digit, dann x`.rand(`a`)`$\in \mathcal{O}(a/\log_2(\texttt{RAND\_MAX}))$.

Somit erzeugen wir eine Zufallszahl, die kleiner als 2^n für einen gegebenen Wert $n \in$ Digit ist.

Bemerkung. Nicht jeder Compiler unterstützt die ANSI-Konstante `RAND_MAX`.

Kapitel 3

Die ganzen Zahlen

3.1 Zahlendarstellung

3.1.1 Darstellung negativer Zahlen durch das 2^n-Komplement

Arithmetik-Systeme wie das INTARI ([42]) verwenden zur Darstellung negativer Zahlen das 2^n-Komplement. Dabei erhalten sie die negative Zahl dadurch, daß die Binärziffern der entsprechenden positiven Zahl komplementiert werden und an der letzten Stelle eine Eins aufaddiert wird. Hierbei muß während der Operationen darauf geachtet werden, daß eine führende Eins eine negative Zahl repräsentiert.

Beispiel. $(\gamma) = -1, (0, \gamma) = 2^\beta - 1$

Um dies zu gewähren, müssen wir zum Beispiel die Funktion `normalize` (siehe Kapitel 2.1.3, Seite 19) aus unserer Natural-Arithmetik folgendermaßen überladen:

```
void normalize()
{
  const Digit HIBIT = 1 << (BETA-1);
  while (*p == 0 && size > 1 && (p[1] & HIBIT) == 0) { ++p; --size; }
  while (*p == GAMMA && size > 1 && (p[1] & HIBIT)) { ++p; --size; }
}
```

Ein Vorteil dieses Ansatzes ist, daß unsere bisherigen Funktionen weitgehend unverändert und ohne Fallunterscheidungen implementiert werden können.

Beispiel. $(\gamma) + (0, \gamma) \equiv (0, \gamma - 1) \pmod{2^\beta}$,
$(-1) \cdot \gamma = (\gamma) \cdot (0, \gamma) = (\gamma - 1, 1) - (\gamma) \cdot 2^\beta \equiv (\gamma, 1) = -\gamma \pmod{2^{2\beta}}$.

Auch wenn dies die Art ist, wie der Computer intern Zahlen darstellt und festverdrahtet miteinander operieren läßt, hat sie für uns den Nachteil, daß wir uns wie bei der Natural-Arithmetik überwiegend auf der elementaren Ebene befinden und nur Bruchteile aus der geerbten Klasse Natural übernehmen, womit wir auch beim Austauschen von Algorithmen unflexibler sind.

Der Basisdatentyp SignDigit ist ein elementarer vorzeichenbehafteter Typ, der die gleiche Größe wie der Datentyp Digit hat.

⟨ type declaration of SignDigit 185 ⟩ ≡
```
typedef long SignDigit;
```
◇
Macro referenced in 2.

⟨ testing size of Digit and SignDigit 186a ⟩ ≡
```
if (sizeof(Digit) != sizeof(SignDigit)) errmsg(0, "Wrong size of SignDigit!");
```
◇
Macro referenced in 398b.

3.1.2 Darstellung durch Betrag und Vorzeichen

Unsere Integer-Klasse gestaltet sich in dieser Art:

⟨ class Integer 186b ⟩ ≡
```
class Integer : public Natural {
public:
  ⟨ output variables for representation of an Integer 192a ⟩
private:
  int sgn;

  Integer(const size_t, char);

  ⟨ private memberfunctions of Integer 423 ⟩

public:
  Integer(const SignDigit = 0);
  Integer(const Natural&);
  Integer(const Integer&);
#ifndef _Old_STD_
explicit
#endif
  Integer(const char*, const Digit = 10);
  ~Integer();

  ⟨ public memberfunctions and friends of Integer 424 ⟩
};
```
◇
Macro referenced in 419.

Hierbei gilt für die hinzugekommene Variable:

$$a.\mathtt{sgn} = \operatorname{sign}(a) = \begin{cases} 1\,, & a > 0 \\ 0\,, & a = 0 \\ -1\,, & a < 0 \end{cases} \qquad \text{für } a \in \mathsf{Integer}.$$

Dadurch erfüllen wir nun die Eigenschaft:

$$\mathsf{Integer} := \{x \mid x \in \mathsf{Natural}\} \cup \{-x \mid x \in \mathsf{Natural}\} = \mathsf{Natural} \cup -\mathsf{Natural} \subset \mathbb{Z}.$$

Diagnosemakro

Wir benötigen das folgende Diagnosemakro, um die Bedingung an die interne Darstellung eines Integers zu überprüfen.

⟨ condition of an Integer 187a ⟩ ≡
```
# define INTEGERCONDITION(a)                              \
  CONDITION((a).highest() == 0 && sign(a) == 0 ||         \
            (a).highest() != 0 &&                         \
            (sign(a) == 1 || sign(a) == -1));

# define INTEGER_FOR_CHECK(a, b)                          \
  Integer (a) = (b);
```
◇
Macro referenced in 6b.

3.2 Elementare Funktionen und Operationen

Alle Operationen auf Integers werden wir wegen der internen Betragsdarstellung grundsätzlich nur in elementarer Weise auf die abgeleiteten Natural-Operationen abbilden und somit jede Funktion in verschiedene Fälle unterteilen. Zusätzlich muß noch jedes Mal die hinzugekommene Variable `sgn` aktualisiert werden.

3.2.1 Konstruktoren und Destruktor

Geschützt

⟨ protected constructor Integer without the initialization of the elements 187b ⟩ ≡
```
inline Integer::Integer(const size_t a, char b)
 : Natural(a, b)
// Algorithm:  c := Integer(a, b)
// Input:      b in char, a in size_t where a >= 1.
// Output:     c in Integer such that L(c) = R(c) = a;
//             map Integer(a, b) to Natural(a, b) ||
//
// Note:       This constructor don't fulfill the conditions for Integers.
//
// internal constructor without the initialization of the elements.
{
}
```
◇
Macro referenced in 419.

Laufzeit. Sei $t \in$ `size_t` und $c \in$ `char`, dann `Integer(`t`,`c`)` $\sim$ `Natural(`t`,`c`)`.

Öffentlich

⟨ default constructor Integer 187c ⟩ ≡
```
inline Integer::Integer(const SignDigit a)
 : Natural((a < 0)? -a : a)
// Algorithm:  c := Integer(a)
// Input:      a in SignDigit.
// Output:     c in Integer such that c = a ||
{
  if (a < 0) sgn = -1;
  else if (a > 0) sgn = 1;
  else sgn = 0;
}
```
◇
Macro referenced in 419.

Laufzeit. Sei $a \in$ SignDigit, dann `Integer(`a`)` $\sim$ `Natural(`a`)`.

⟨ overloaded constructor Integer for Natural 188a ⟩ ≡
```
inline Integer::Integer(const Natural& a)
 : Natural(a)
// Algorithm:  c := Integer(a)
// Input:      a in Natural.
// Output:     c in Integer such that c = a ||
{
  sgn = (a != 0);
}
```
◇
Macro referenced in 419.

Laufzeit. Sei $x \in$ Natural, dann `Integer(`x`)` $\sim$ `Natural(`x`)`.

⟨ copy constructor Integer 188b ⟩ ≡
```
inline Integer::Integer(const Integer& a)
: Natural(::abs(a))
// Algorithm:  c := Integer(a)
// Input:      a in Integer.
// Output:     c in Integer such that c = a ||
{
  sgn = a.sgn;
}
```
◇
Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer, dann `Integer(`u`)` $\sim$ `Natural(abs(`u`))`.

⟨ overloaded constructor Integer for ASCII-string conversion 188c ⟩ ≡
```
inline Integer::Integer(const char* a, const Digit b)
 : Natural(a + (*a == '-'), b)
// Algorithm:  c := Integer(a, b)
// Input:      a in String, b in Digit.
// Output:     c in Integer such that c = a ||
{
  if (*a == '-') sgn = -1;
  else sgn = (::abs(*this) != 0);
}
```
◇
Macro referenced in 419.

Laufzeit. Sei $s \in$ String und $a \in$ Digit, dann `Integer(`s`,`a`)` $\sim$ `Natural(`s`,`a`)`.

⟨ destructor Integer 188d ⟩ ≡
```
inline Integer::~Integer()
{
}
```
◇
Macro referenced in 419.

Laufzeit. `~Integer()`$\sim$`~Natural()`.

3.2.2 Absolutbetrag

⟨ absolute value of an Integer 189a ⟩ ≡

```
inline const Natural& abs(const Integer& a)
// Algorithm:  c := abs(a)
// Input:      a in Integer.
// Output:     c in Natural such that c = |a| ||
{
  return (const Natural&)a;
}

inline Natural& Integer::abs()
// Algorithm:  c := a.abs()
// Input:      a in Integer.
// Output:     c in Natural such that c = |a| ||
{
  return (Natural&)*this;
}
```

◇

Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer, dann `abs(`u`)` $\in \mathcal{O}(1)$.
Laufzeit. Sei $u \in$ Integer, dann u`.abs()` $\in \mathcal{O}(1)$.

3.2.3 Vorzeichen/Einheiten

⟨ sign of an Integer 189b ⟩ ≡

```
inline int sign(const Integer& a)
// Algorithm:  c := sign(a)
// Input:      a in Integer.
// Output:     c in int such that if a = 0 then c = 0
//             else if a > 0 then c = 1 else c = -1 ||
{
  return a.sgn;
}
```

◇

Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer, dann `sign(`u`)`$\in \mathcal{O}(1)$.

⟨ units of an Integer 189c ⟩ ≡

```
inline int units(Integer& a)
// Algorithm:  c := units(a)
// Input:      a in Integer.
// Output:     a in Integer, c in int such that a := |a|,
//             if a >= 0 then c = 1 else c = -1 ||
{
  if (sign(a) >= 0) return 1;
  else { a = -a; return -1; }
}
```

◇

Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer, dann `units(`u`)`$\in \mathcal{O}(1)$.

3.2.4 Vertauschen zweier Integers

⟨ function swap for Integers 190a ⟩ ≡

```
void swap(Integer& a, Integer& b)
// Algorithm:  swap(a, b)
// Input:      a,b in Integer.
// Output:     a,b in Integer such that t := a, a := b, b := t
//             where t in Integer ||
{
  INTEGERCONDITION(a);
  INTEGERCONDITION(b);

  swap(a.abs(), b.abs());
  int t = a.sgn; a.sgn = b.sgn; b.sgn = t;

  INTEGERCONDITION(a);
  INTEGERCONDITION(b);
}
```

◇

Macro referenced in 426.

Laufzeit. Seien $u, v \in$ Integer, dann `swap(`u`,`v`)` $\sim$ `swap(abs(`u`),abs(`v`))`.

3.2.5 Kopierfunktionen

⟨ assign operator= for Integers 190b ⟩ ≡

```
inline Integer& Integer::operator=(const Integer& a)
// Algorithm:  c := c = a
// Input:      a,c in Integer.
// Output:     c in Integer such that c = a ||
{
  sgn = a.sgn;
  abs() = ::abs(a);
  return *this;
}
```

◇

Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer, dann u`=`v $\sim$ `abs(`u`)=abs(`v`)`.

⟨ assign operator= for an Integer with a Natural 190c ⟩ ≡

```
inline Integer& Integer::operator=(const Natural& a)
// Algorithm:  c := c = a
// Input:      a in Natural,c in Integer.
// Output:     c in Integer such that c = a ||
{
  sgn = (a > 0);
  abs() = a;
  return *this;
}
```

◇

Macro referenced in 419.

Laufzeit. Sei $x \in$ Natural und $u \in$ Integer, dann u`=`x $\sim$ `abs(`u`)=`x.

⟨ assign operator= of an Integer with a SignDigit 191a ⟩ ≡
```
inline SignDigit Integer::operator=(const SignDigit a)
// Algorithm:  c := b = a
// Input:      a in SignDigit, b in Integer.
// Output:     b in Integer, c in SignDigit such that b = a, c = a ||
{
  if (a >= 0) { sgn = (a > 0); abs() = a; }
  else { sgn = -1; abs() = -a; }
  return a;
}
```
◇
Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer und $a \in$ SignDigit, dann u=a $\sim$ abs(u)=a.

3.2.6 Streamausgabe

⟨ puts an Integer on output stream 191b ⟩ ≡
```
inline OSTREAM& operator<<(OSTREAM& out, const Integer& a)
// Algorithm:  o := o << a
// Input:      o in ostream, a in Integer.
// Output:     o in ostream ||
//
// Note:       puts Integer a on output stream.
{
  if (sign(a) < 0) {
    const int b = out.width();
    if (b > 0) { out.width(0); out << '-'; out.width(b-1); }
    else out << '-';
  }
  return out << abs(a);
}
```
◇
Macro referenced in 419.

Laufzeit. Sei $o \in$ostream und $u \in$ Integer, dann o<<$u \sim$ o<< abs(u).

Interne Darstellung

⟨ puts internal representation of an Integer on output stream 191c ⟩ ≡
```
inline OSTREAM& operator<<(OSTREAM& out, const Integer::rep& a)
// puts internal representation of Integer a on output stream.
{
  if (a.sgn == -1) out << '-';
  return out << print(a.nat, a.bin);
}
```
◇
Macro referenced in 419.

Laufzeit. Sei $o \in$ostream und $r \in$ Integer::rep, dann o<<$r \in \mathcal{O}(\mathcal{L}(r))$.

⟨ output variables for representation of an Integer 192a ⟩ ≡

```
struct rep {
  const int   sgn;
  const       Natural& nat;
  const bool  bin;
  rep(const int a, const Natural& b, const bool c = false)
    : sgn(a), nat(b), bin(c) {}
};
```

◇

Macro referenced in 186b.

⟨ function print of an Integer 192b ⟩ ≡

```
inline Integer::rep print(const Integer& a, bool b)
// Algorithm:  o := o << print(a, b)
// Input:      o in ostream, a in Integer, b in bool.
// Output:     o in ostream ||
//
// Note:       puts internal representation of Integer a on an output stream.
{
  return Integer::rep(sign(a), abs(a), b);
}
```

◇

Macro referenced in 419.

3.2.7 Streameingabe

⟨ gets an Integer from input stream 192c ⟩ ≡

```
ISTREAM& operator>>(ISTREAM& in, Integer& a)
// Algorithm:  i := i >> a
// Input:      i in istream.
// Output:     i in istream, a in Integer ||
//
// Note:       gets Integer a from input stream.
{
  INTEGERCONDITION(a);

  if (!in.good()) return in;
  char ch = 0;
  if (in.get(ch) && ch != '-') in.putback(ch);
  in >> a.abs();
  a.sgn = (abs(a) != 0);
  if (ch == '-') a.neg();

  INTEGERCONDITION(a);

  return in;
}
```

◇

Macro referenced in 426.

Laufzeit. Sei $i \in$ `istream` und $u \in$ Integer, dann i`>>`$u \sim$ i`>> abs(`u`)`.

Interne Darstellung

⟨ gets internal representation of an Integer from input stream 192d ⟩ ≡

```
bool Integer::scan(ISTREAM& in)
```

```
// Algorithm:  b := a.scan(i)
// Input:      a in Integer, i in istream.
// Output:     a in Integer, i in istream, b in bool ||
//
// Note:       gets Integer a as an internal representation from input stream
//             if b is true.
{
  if (!in.good()) return false;
  char c = 0;
  if (in.get(c) && c != '-') in.putback(c);
  const bool b = Natural::scan(in);
  sgn = (::abs(*this) != 0);
  if (c == '-') neg();
  return b;
}
◇
```

Macro referenced in 426.

Laufzeit. Sei $i \in$`ostream`, $u \in$ Integer, dann u`.scan(`i`)`$\sim$ `abs(`u`).scan(`i`)`.

Konvertierung in ein beliebiges Stellenwertsystem

⟨ converts an Integer to a string 193a ⟩ ≡

```
char* Itoa(const Integer& a, char* b, const Digit c)
// Algorithm:  c := Itoa(a, c, b)
// Input:      a in Integer, b in Digit, c in String
//             where 2 <= b <= 36, sizeof(c) > BETA*L(a)/log2(b).
// Output:     c in String such that c = a ||
//
// Note:       conversion Integer to string.
{
  INTEGERCONDITION(a);

  if (sign(a) >= 0) Ntoa(abs(a), b, c);
  else { *b = '-'; Ntoa(abs(a), b+1, c); }
  return b;
}
◇
```

Macro referenced in 426.

Laufzeit. Sei $u \in$ Integer, $s \in$ String und $a \in$ Digit, dann

$$\texttt{Itoa(}u\texttt{,}s\texttt{,}a\texttt{)} \sim \texttt{Ntoa(abs(}u\texttt{),}s\texttt{,}a\texttt{)}.$$

Konvertierung aus einem beliebigen Stellenwertsystem

⟨ converts a string to an Integer 193b ⟩ ≡

```
const char* Integer::atoI(const char* a, const Digit b)
// Algorithm:  c := d.atoI(a, b)
// Input:      d in Integer, a in String, b in Digit where 2 <= b <= 36.
// Output:     d in Integer, c in String such that d = a ||
//
// Note: Returns a pointer to the first occurrence of a non-digit character
```

```
//       in a.
{
  const bool d = (*a == '-');
  a = atoN(a + d, b);
  sgn = (::abs(*this) != 0);
  if (d) neg();

  INTEGERCONDITION(*this);

  return a;
}
◇
```

Macro referenced in 426.

Laufzeit. Sei $u \in$ Integer, $s \in$ String und $a \in$ Digit, dann u.atoI(s,a) $\sim$ u.atoN(s,a).

⟨ converts a string to an Integer by function call 194a ⟩ ≡

```
inline Integer atoI(const char* a, const Digit b)
// Algorithm:  c := atoI(a, b)
// Input:      a in String, b in Digit where 2 <= b <= 36.
// Output:     c in Integer such that c = a ||
//
// Note:       conversion string to Integer; return 0 by conversion error.
{
  Integer result;
  result.atoI(a, b);
  return result;
}
◇
```

Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer, $s \in$ String und $a \in$ Digit, dann atoI(s,a) $\sim$ u.atoI(s,a).

3.2.8 Vergleichsoperatoren

⟨ comparison operator== for Integers 194b ⟩ ≡

```
inline bool operator==(const Integer& a, const Integer& b)
// Algorithm:  c := a == b
// Input:      a,b in Integer.
// Output:     c in bool such that if a = b then c = true else c = false ||
{
  return (sign(a) == sign(b) && abs(a) == abs(b));
}
◇
```

Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer, dann u==v $\sim$ abs(u) == abs(v).

⟨ comparison operator!= for Integers 195a ⟩ ≡

```
inline bool operator!=(const Integer& a, const Integer& b)
// Algorithm:  c := a != b
// Input:      a,b in Integer.
// Output:     c in bool such that if a = b then c = false else c = true ||
{
  return (sign(a) != sign(b) || abs(a) != abs(b));
}
```

◇

Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer, dann u`!=`$v \sim$ `abs(`u`) != abs(`v`)`.

⟨ comparison operator< for Integers 195b ⟩ ≡

```
inline bool operator<(const Integer& a, const Integer& b)
// Algorithm:  c := a < b
// Input:      a,b in Integer.
// Output:     c in bool such that if a < b then c = true else c = false ||
{
  return (sign(a) < sign(b) || sign(a) == sign(b)
   && ((sign(a) > 0)? abs(a) < abs(b) : abs(b) < abs(a)));
}
```

◇

Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer, dann u`<`$v \sim$ `abs(`u`) < abs(`v`)`.

⟨ comparison operator<= for Integers 195c ⟩ ≡

```
inline bool operator<=(const Integer& a, const Integer& b)
// Algorithm:  c := a <= b
// Input:      a,b in Integer.
// Output:     c in bool such that if a <= b then c = true else c = false ||
{
  return !(b < a);
}
```

◇

Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer, dann u`<=`$v \sim v$`<`u.

⟨ comparison operator> for Integers 195d ⟩ ≡

```
inline bool operator>(const Integer& a, const Integer& b)
// Algorithm:  c := a > b
// Input:      a,b in Integer.
// Output:     c in bool such that if a > b then c = true else c = false ||
{
  return (b < a);
}
```

◇

Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer, dann u`>`$v \sim v$`<`u.

⟨ comparison operator>= for Integers 196a ⟩ ≡

```
inline bool operator>=(const Integer& a, const Integer& b)
// Algorithm:  c := a >= b
// Input:      a,b in Integer.
// Output:     c in bool such that if a >= b then c = true else c = false ||
{
  return !(a < b);
}
◇
```

Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer, dann u>=$v \sim u$<v.

SignDigit-Vergleiche

⟨ comparison operator== of an Integer with a SignDigit 196b ⟩ ≡

```
inline bool operator==(const Integer& a, const SignDigit b)
// Algorithm:  c := a == b
// Input:      a in Integer, b in SignDigit.
// Output:     c in bool such that if a = b then c = true else c = false ||
{
  return (b < 0)? (sign(a) < 0 && abs(a) == Digit(-b))
                : (sign(a) >= 0 && abs(a) == Digit(b));
}
◇
```

Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer und $a \in$ SignDigit, dann u==$a \sim$ abs(u) ==a.

⟨ comparison operator!= of an Integer with a SignDigit 196c ⟩ ≡

```
inline bool operator!=(const Integer& a, const SignDigit b)
// Algorithm:  c := a != b
// Input:      a in Integer, b in SignDigit.
// Output:     c in bool such that if not a = b then c = true else c = false ||
{
  return (b < 0)? (sign(a) >= 0 || abs(a) != Digit(-b))
                : (sign(a) < 0 || abs(a) != Digit(b));
}
◇
```

Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer und $a \in$ SignDigit, dann u!=$a \sim$ abs(u) !=a.

⟨ comparison operator< of an Integer with a SignDigit 196d ⟩ ≡

```
inline bool operator<(const Integer& a, const SignDigit b)
// Algorithm:  c := a < b
// Input:      a in Integer, b in SignDigit.
// Output:     c in bool such that if a < b then c = true else c = false ||
{
  return (b < 0)? (sign(a) < 0 && abs(a) > Digit(-b))
                : (sign(a) < 0 || abs(a) < Digit(b));
}
◇
```

Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer und $a \in$ SignDigit, dann u<$a \sim$ abs(u) <a.

⟨ comparison operator<= of an Integer with a SignDigit 197a ⟩ ≡
```
inline bool operator<=(const Integer& a, const SignDigit b)
// Algorithm:  c := a <= b
// Input:      a in Integer, b in SignDigit.
// Output:     c in bool such that if a <= b then c = true else c = false ||
{
  return (b < 0)? (sign(a) < 0 && abs(a) >= Digit(-b))
                : (sign(a) <= 0 || abs(a) <= Digit(b));
}
```
◇
Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer und $a \in$ SignDigit, dann u<=$a \sim$ abs(u) <=a.

⟨ comparison operator> of an Integer with a SignDigit 197b ⟩ ≡
```
inline bool operator>(const Integer& a, const SignDigit b)
// Algorithm:  c := a > b
// Input:      a in Integer, b in SignDigit.
// Output:     c in bool such that if a > b then c = true else c = false ||
{
  return (b <= 0)? (sign(a) > 0 || abs(a) < Digit(-b))
                 : (sign(a) > 0 && abs(a) > Digit(b));
}
```
◇
Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer und $a \in$ SignDigit, dann u>$a \sim$ abs(u) >a.

⟨ comparison operator>= of an Integer with a SignDigit 197c ⟩ ≡
```
inline bool operator>=(const Integer& a, const SignDigit b)
// Algorithm:  c := a >= b
// Input:      a in Integer, b in SignDigit.
// Output:     c in bool such that if a >= b then c = true else c = false ||
{
  return (b <= 0)? (sign(a) >= 0 || abs(a) <= Digit(-b))
                 : (sign(a) > 0 && abs(a) >= Digit(b));
}
```
◇
Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer und $a \in$ SignDigit, dann u>=$a \sim$ abs(u) >=a.

3.2.9 Additive Operationen

Inkrementierung

⟨ prefix incrementation of an Integer 197d ⟩ ≡
```
const Integer& Integer::operator++()
// Algorithm:  c := ++a
// Input:      a in Integer.
// Output:     a,c in Integer such that a := a+1, c := a ||
{
  INTEGERCONDITION(*this);
```

```
  const int sT = sgn;
  if (sT == 0) *this = 1;
  else if (sT > 0) ++abs();
  else if (::abs(*this) == 1) *this = 0;
  else --abs();

  INTEGERCONDITION(*this);

  return *this;
}
◇
```

Macro referenced in 426.

Laufzeit. Sei $u \in$ Integer, dann ++u $\sim$ ++abs(u).

⟨ postfix incrementation of an Integer 198a ⟩ ≡

```
inline Integer Integer::operator++(int)
// Algorithm:  c := a++
// Input:      a in Integer.
// Output:     a,c in Integer such that c := a, a := a+1 ||
{
  Integer a(*this);
  ++(*this);
  return a;
}
◇
```

Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer, dann u++ $\sim$ abs(u)++.

Decrementierung

⟨ prefix decrementation of an Integer 198b ⟩ ≡

```
const Integer& Integer::operator--()
// Algorithm:  c := --a
// Input:      a in Integer.
// Output:     a,c in Integer such that a := a-1, c := a ||
{
  INTEGERCONDITION(*this);

  const int sT = sgn;
  if (sT == 0) { abs() = 1; sgn = -1; }
  else if (sT < 0) ++abs();
  else if (::abs(*this) == 1) *this = 0;
  else --abs();

  INTEGERCONDITION(*this);

  return *this;
}
◇
```

Macro referenced in 426.

Laufzeit. Sei $u \in$ Integer, dann --u $\sim$ --abs(u).

⟨ postfix decrementation of an Integer 199a ⟩ ≡
```
inline Integer Integer::operator--(int)
// Algorithm:  c := a--
// Input:      a in Integer.
// Output:     a,c in Integer such that c := a, a := a-1 ||
{
  Integer a(*this);
  --(*this);
  return a;
}
```
◇
Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer, dann u`--` $\sim$ `abs(`u`)--`.

Negation

⟨ negation of an Integer 199b ⟩ ≡
```
inline void Integer::neg()
// Algorithm:  a.neg()
// Input:      a in Integer.
// Output:     a in Integer such that a := -a ||
{
  sgn = -sgn;
}
```
◇
Macro defined by 199bc.
Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer, dann u`.neg()`$\in \mathcal{O}(1)$.

⟨ negation of an Integer 199c ⟩ ≡
```
inline void Integer::neg(const Integer& a)
// Algorithm:  b.neg(a)
// Input:      a in Integer.
// Output:     b in Integer such that b = -a ||
{
  *this = a; neg();
}
```
◇
Macro defined by 199bc.
Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer, dann v`.neg(`u`)`$\in \mathcal{O}(\mathcal{L}(|u|))$.

⟨ negation operator- of an Integer 199d ⟩ ≡
```
inline Integer::Integer(const binder_arguments<Integer, Integer,
                                               Integer_negate_tag>& a)
 : Natural(' ', ' ')
{
  get_memory(a.x.length()+DELTA);
  neg(a.x);
}

inline Integer& Integer::operator=(const binder_arguments<Integer, Integer,
                                                          Integer_negate_tag>& a)
```

```
{
  if (this == &a.x) neg();
  else neg(a.x);
  return *this;
}

inline binder_arguments<Integer, Integer, Integer_negate_tag>
 operator-(const Integer& a)
// Algorithm:  c := -a
// Input:      a in Integer.
// Output:     c in Integer such that c = -a ||
{
  return binder_arguments<Integer, Integer, Integer_negate_tag>(a, a);
}
◇
```

Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer, dann $\begin{array}{lcl} u = \text{-}u & \sim & u.\mathrm{neg}(), \\ \text{-}u & \sim & v.\mathrm{neg}(u). \end{array}$

Addition

⟨ addition of two Integers 200a ⟩ ≡

```
void Integer::add(const Integer& a, const Integer& b)
// Algorithm:  c.add(a, b)
// Input:      a,b in Integer.
// Output:     c in Integer such that c = a+b ||
{
  INTEGERCONDITION(a);
  INTEGERCONDITION(b);

  const int sA = a.sgn;
  const int sB = b.sgn;
  if (sA == 0) *this = b;
  else if (sB == 0) *this = a;
  else if (sA == sB) {
    abs() = ::abs(a) + ::abs(b);
    sgn = sA;
  } else sgn = sA*::abs(::abs(a), ::abs(b), abs());

  INTEGERCONDITION(*this);
}
◇
```

Macro referenced in 426.

Laufzeit. Seien $u, v, w \in$ Integer, dann

$$w.\mathtt{add}(u,v) \sim \begin{cases} w.\mathtt{add}(\mathtt{abs}(u),\mathtt{abs}(v)) , & \mathtt{sign}(u) = \mathtt{sign}(v) \\ \mathtt{abs}(\mathtt{abs}(u),\mathtt{abs}(v),\mathtt{abs}(w)) , & \mathtt{sign}(u) \neq \mathtt{sign}(v) \end{cases} .$$

⟨ addition of an Integer with a Natural 200b ⟩ ≡

```
void Integer::add(const Integer& a, const Natural& b)
// Algorithm:  c.add(a, b)
// Input:      a in Integer, b in Natural.
```

```
// Output:     c in Integer such that c = a+b ||
{
  INTEGERCONDITION(a);

  const int sA = a.sgn;
  if (sA == 0) *this = b;
  else if (sA == 1) {
    abs() = ::abs(a) + b;
    sgn = 1;
  } else sgn = -::abs(::abs(a), b, abs());

  INTEGERCONDITION(*this);
}
◇
```

Macro referenced in 426.

Laufzeit. Seien $u, v \in$ Integer und $x \in$ Natural, dann

$$v.\mathtt{add}(u,x) \sim \begin{cases} v.\mathtt{add}(\mathtt{abs}(u),x)\ , & \mathtt{sign}(u) \geq 0 \\ \mathtt{abs}(\mathtt{abs}(u),x,\mathtt{abs}(v))\ , & \mathtt{sign}(u) < 0 \end{cases}.$$

⟨ addition of an Integer with a SignDigit 201a ⟩ ≡

```
void Integer::add(const Integer& a, const SignDigit b)
// Algorithm:  c.add(a, b)
// Input:      a in Integer, b in SignDigit where not &a = &c.
// Output:     c in Integer such that c = a+b ||
{
  INTEGERCONDITION(a);
  CONDITION(&a != this);

  if (b < 0) sub(a, -b);
  else {
    const int sA = a.sgn;
    sgn = sA;
    if (sA > 0) Natural::add(::abs(a), b);
    else if (sA == 0) *this = b;
    else if (::abs(a) > Digit(b)) Natural::sub(::abs(a), b);
    else *this = b - a.highest();
  }

  INTEGERCONDITION(*this);
}
◇
```

Macro referenced in 426.

Laufzeit. Sei $u, v \in$ Integer und $a \in$ SignDigit, dann

$$v.\mathtt{add}(u,a) \sim \begin{cases} \mathtt{abs}(v).\mathtt{add}(u,a)\ , & \mathtt{sign}(u) \geq 0 \\ \mathtt{abs}(v).\mathtt{sub}(u,a)\ , & \mathtt{sign}(u) < 0 \end{cases}.$$

⟨ additive operator+ for Integers 201b ⟩ ≡

```
inline Integer::Integer(const binder_arguments<Integer, Integer,
                                                Integer_plus_tag>& a)
 : Natural(' ', ' ')
{
```

```
  get_memory(max(a.x.length(), a.y.length())+DELTA);
  add(a.x, a.y);
}

inline Integer& Integer::operator=(const binder_arguments<Integer, Integer,
                                                          Integer_plus_tag>& a)
{
  add(a.x, a.y);
  return *this;
}

inline binder_arguments<Integer, Integer, Integer_plus_tag>
 operator+(const Integer& a, const Integer& b)
// Algorithm:  c := a+b
// Input:      a,b in Integer.
// Output:     c in Integer such that c = a+b ||
{
  return binder_arguments<Integer, Integer, Integer_plus_tag>(a, b);
}
◇
```

Macro referenced in 419.

Laufzeit. Seien $u, v, w \in$ Integer, dann $u{+}v \sim w.\mathtt{add}(u, v)$.

⟨ additive operator+ of an Integer with a Natural 202 ⟩ ≡

```
inline Integer::Integer(const binder_arguments<Integer, Natural,
                                               Integer_plus_tag>& a)
 : Natural(' ', ' ')
{
  get_memory(max(a.x.length(), a.y.length())+DELTA);
  add(a.x, a.y);
}

inline Integer& Integer::operator=(const binder_arguments<Integer, Natural,
                                                          Integer_plus_tag>& a)
{
  add(a.x, a.y);
  return *this;
}

inline binder_arguments<Integer, Natural, Integer_plus_tag>
 operator+(const Integer& a, const Natural& b)
// Algorithm:  c := a+b
// Input:      a in Integer, b in Natural.
// Output:     c in Integer such that c = a+b ||
{
  return binder_arguments<Integer, Natural, Integer_plus_tag>(a, b);
}

inline binder_arguments<Integer, Natural, Integer_plus_tag>
 operator+(const Natural& a, const Integer& b)
// Algorithm:  c := a+b
// Input:      a in Natural, b in Integer.
// Output:     c in Integer such that c = a+b ||
{
  return binder_arguments<Integer, Natural, Integer_plus_tag>(b, a);
```

```
}
◇
```

Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer und $x \in$ Natural, dann u+$x \sim v$.add(u,x).
Laufzeit. Seien $u, v \in$ Integer und $x \in$ Natural, dann x+$u \sim v$.add(u,x).

⟨ additive operator+ of an Integer with a SignDigit 203 ⟩ ≡

```
inline Integer::Integer(const binder_arguments<Integer, SignDigit,
                                                Integer_plus_tag>& a)
 : Natural(' ', ' ')
{
  get_memory(a.x.length()+DELTA);
  add(a.x, a.y);
}

inline Integer& Integer::operator=(const binder_arguments<Integer, SignDigit,
                                                          Integer_plus_tag>& a)
{
  if (this == &a.x) return *this += a.y;
  else { add(a.x, a.y); return *this; }
}

inline binder_arguments<Integer, SignDigit, Integer_plus_tag>
 operator+(const Integer& a, const SignDigit& b)
// Algorithm:  c := a+b
// Input:      a in Integer, SignDigit b.
// Output:     c in Integer such that c = a+b ||
{
  return binder_arguments<Integer, SignDigit, Integer_plus_tag>(a, b);
}

inline binder_arguments<Integer, SignDigit, Integer_plus_tag>
 operator+(const SignDigit& a, const Integer& b)
// Algorithm:  c := a+b
// Input:      a in SignDigit, Integer b.
// Output:     c in Integer such that c = a+b ||
{
  return binder_arguments<Integer, SignDigit, Integer_plus_tag>(b, a);
}
◇
```

Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer und $a \in$ SignDigit, dann u+$a \sim v$.add(u,a).

Zuweisungsoperatoren

⟨ assign operator+= for Integers 204a ⟩ ≡

```
inline Integer& Integer::operator+=(const Integer& a)
// Algorithm:  c := c += a
// Input:      a,c in Integer.
// Output:     c in Integer such that c := c+a ||
{
  add(a, *this);
  return *this;
}
```

◇

Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer, dann u+=$v \sim u$.add(u,v).

⟨ assign operator+= of an Integer with a Natural 204b ⟩ ≡

```
inline Integer& Integer::operator+=(const Natural& a)
// Algorithm:  c := c += a
// Input:      a in Natural, c in Integer.
// Output:     c in Integer such that c := c+a ||
{
  add(*this, a);
  return *this;
}
```

◇

Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer und $x \in$ Natural, dann u+=$x \sim u$.add(u,x).

⟨ assign operator+= of an Integer with a SignDigit 204c ⟩ ≡

```
Integer& Integer::operator+=(const SignDigit a)
// Algorithm:  c := c += a
// Input:      a SignDigit, c in Integer.
// Output:     c in Integer such that c := c+a ||
{
  INTEGERCONDITION(*this);

  if (a < 0) return *this -= -a;
  const int sT = sgn;
  if (sT > 0) abs() += a;
  else if (sT == 0) *this = a;
  else if (::abs(*this) > Digit(a)) abs() -= a;
  else *this = a - highest();

  INTEGERCONDITION(*this);

  return *this;
}
```

◇

Macro referenced in 426.

Laufzeit. Sei $u \in$ Integer und $a \in$ SignDigit, dann u+=$a \sim \begin{cases} \texttt{abs}(u)\texttt{+=a}\,, & \texttt{sign}(u) \geq 0 \\ \texttt{abs}(u)\texttt{-=a}\,, & \texttt{sign}(u) < 0 \end{cases}$.

Subtraktion

⟨ subtraction of two Integers 205a ⟩ ≡

```
void Integer::sub(const Integer& a, const Integer& b)
// Algorithm:  c.sub(a, b)
// Input:      a,b in Integer.
// Output:     c in Integer such that c = a-b ||
{
  INTEGERCONDITION(a);
  INTEGERCONDITION(b);

  const int sA = a.sgn;
  const int sB = b.sgn;
  if (sA == 0) neg(b);
  else if (sB == 0) *this = a;
  else if (sA != sB) {
    abs() = ::abs(a) + ::abs(b);
    sgn = sA;
  } else sgn = sA*::abs(::abs(a), ::abs(b), abs());

  INTEGERCONDITION(*this);
}
```

◇

Macro referenced in 426.

Laufzeit. Seien $u, v, w \in$ Integer, dann

$$w.\mathtt{sub}(u,v) \sim \begin{cases} \mathtt{abs}(w).\mathtt{add}(\mathtt{abs}(u),\mathtt{abs}(v)) , & \mathtt{sign}(u) \neq \mathtt{sign}(v) \\ \mathtt{abs}(\mathtt{abs}(u),\mathtt{abs}(v),\mathtt{abs}(w)) , & \mathtt{sign}(u) = \mathtt{sign}(v) \end{cases} .$$

⟨ subtraction of an Integer with a Natural 205b ⟩ ≡

```
void Integer::sub(const Integer& a, const Natural& b)
// Algorithm:  c.sub(a, b)
// Input:      a in Integer, b in Natural.
// Output:     c in Integer such that c = a-b ||
{
  INTEGERCONDITION(a);

  const int sA = a.sgn;
  if (sA == 0) { *this = b; neg(); }
  else if (sA == -1) {
    abs() = ::abs(a) + b;
    sgn = -1;
  } else sgn = ::abs(::abs(a), b, abs());

  INTEGERCONDITION(*this);
}
```

◇

Macro referenced in 426.

Laufzeit. Seien $u, v \in$ Integer und $x \in$ Natural, dann

$$v.\mathtt{sub}(u,x) \sim \begin{cases} \mathtt{abs}(v).\mathtt{add}(\mathtt{abs}(u),x) , & \mathtt{sign}(u) < 0 \\ \mathtt{abs}(\mathtt{abs}(u),x,\mathtt{abs}(v)) , & \mathtt{sign}(u) \geq 0 \end{cases} .$$

⟨ subtraction of a Natural with an Integer 206a ⟩ ≡

```
void Integer::sub(const Natural& a, const Integer& b)
// Algorithm:  c.sub(a, b)
// Input:      a in Natural, b in Integer.
// Output:     c in Integer such that c = a-b ||
{
  INTEGERCONDITION(b);

  const int sB = b.sgn;
  if (sB == 0) *this = a;
  else if (sB == -1) {
    abs() = a + ::abs(b);
    sgn = 1;
  } else sgn = ::abs(a, ::abs(b), abs());

  INTEGERCONDITION(*this);
}
```

◇

Macro referenced in 426.

Laufzeit. Seien $u, v \in$ Integer und $x \in$ Natural, dann

$$v\texttt{.sub}(x,u) \sim \begin{cases} \texttt{abs}(v)\texttt{.add}(x\texttt{,abs}(u))\ , & \texttt{sign}(u) < 0 \\ \texttt{abs}(x\texttt{,abs}(u)\texttt{,abs}(v))\ , & \texttt{sign}(u) \geq 0 \end{cases} .$$

⟨ subtraction of an Integer with a SignDigit 206b ⟩ ≡

```
void Integer::sub(const Integer& a, const SignDigit b)
// Algorithm:  c.sub(a, b)
// Input:      a,c in Integer, b in SignDigit where not &a = &c.
// Output:     c in Integer such that c = a-b ||
{
  INTEGERCONDITION(a);
  CONDITION(&a != this);

  if (b < 0) add(a, -b);
  else {
    const int sA = a.sgn;
    sgn = sA;
    if (sA < 0) Natural::add(::abs(a), b);
    else if (sA == 0) neg(b);
    else if (::abs(a) > Digit(b)) Natural::sub(::abs(a), b);
    else {
      *this = b - a.highest();
      neg();
    }
  }

  INTEGERCONDITION(*this);
}
```

◇

Macro referenced in 426.

Laufzeit. Sei $u, v \in$ Integer und $a \in$ SignDigit, dann $v\texttt{.sub}(u\texttt{,}a) \sim \begin{cases} \texttt{abs}(v)\texttt{.sub}(u\texttt{,}a)\ , & \texttt{sign}(u) \geq 0 \\ \texttt{abs}(v)\texttt{.add}(u\texttt{,}a)\ , & \texttt{sign}(u) < 0 \end{cases}$.

⟨ additive operator- for Integers 207a ⟩ ≡

```
inline Integer::Integer(const binder_arguments<Integer, Integer,
                                                Integer_minus_tag>& a)
 : Natural(' ', ' ')
{
  get_memory(max(a.x.length(), a.y.length())+DELTA);
  sub(a.x, a.y);
}

inline Integer& Integer::operator=(const binder_arguments<Integer, Integer,
                                                          Integer_minus_tag>& a)
{
  sub(a.x, a.y);
  return *this;
}

inline binder_arguments<Integer, Integer, Integer_minus_tag>
 operator-(const Integer& a, const Integer& b)
// Algorithm:  c := a-b
// Input:      a,b in Integer.
// Output:     c in Integer such that c = a-b ||
{
  return binder_arguments<Integer, Integer, Integer_minus_tag>(a, b);
}
```

◇

Macro referenced in 419.

Laufzeit. Seien $u, v, w \in$ Integer, dann $u\text{-}v \sim w.\mathtt{sub}(u, v)$.

⟨ additive operator- of an Integer with a Natural 207b ⟩ ≡

```
inline Integer::Integer(const binder_arguments<Integer, Natural,
                                                Integer_minus_tag>& a)
 : Natural(' ', ' ')
{
  get_memory(max(a.x.length(), a.y.length())+DELTA);
  sub(a.x, a.y);
}

inline Integer& Integer::operator=(const binder_arguments<Integer, Natural,
                                                          Integer_minus_tag>& a)
{
  sub(a.x, a.y);
  return *this;
}

inline Integer::Integer(const binder_arguments<Natural, Integer,
                                                Integer_minus_tag>& a)
 : Natural(' ', ' ')
{
  get_memory(max(a.x.length(), a.y.length())+DELTA);
  sub(a.x, a.y);
}

inline Integer& Integer::operator=(const binder_arguments<Natural, Integer,
                                                          Integer_minus_tag>& a)
```

```
{
  sub(a.x, a.y);
  return *this;
}

inline binder_arguments<Integer, Natural, Integer_minus_tag>
 operator-(const Integer& a, const Natural& b)
// Algorithm:  c := a-b
// Input:      a in Integer, b in Natural.
// Output:     c in Integer such that c = a-b ||
{
  return binder_arguments<Integer, Natural, Integer_minus_tag>(a, b);
}

inline binder_arguments<Natural, Integer, Integer_minus_tag>
 operator-(const Natural& a, const Integer& b)
// Algorithm:  c := a-b
// Input:      a in Natural, b in Integer.
// Output:     c in Integer such that c = a-b ||
{
  return binder_arguments<Natural, Integer, Integer_minus_tag>(a, b);
}
◇
```

Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer und $x \in$ Natural, dann $u\texttt{-}x \sim v.\texttt{sub}(u,x)$.
Laufzeit. Seien $u, v \in$ Integer und $x \in$ Natural, dann $x\texttt{-}u \sim v.\texttt{sub}(x,u)$.

⟨ additive operator- of an Integer with a SignDigit 208 ⟩ ≡

```
inline Integer::Integer(const binder_arguments<Integer, SignDigit,
                                               Integer_minus_tag>& a)
 : Natural(' ', ' ')
{
  get_memory(a.x.length()+DELTA);
  sub(a.x, a.y);
}

inline Integer& Integer::operator=(const binder_arguments<Integer, SignDigit,
                                                          Integer_minus_tag>& a)
{
  if (this == &a.x) return *this -= a.y;
  else { sub(a.x, a.y); return *this; }
}

inline Integer::Integer(const binder_arguments<SignDigit, Integer,
                                               Integer_minus_tag>& a)
 : Natural(' ', ' ')
{
  get_memory(a.y.length()+DELTA);
  sub(a.y, a.x);
  neg();
}

inline Integer& Integer::operator=(const binder_arguments<SignDigit, Integer,
                                                          Integer_minus_tag>& a)
```

```
{
  if (this == &a.y) *this -= a.x;
  else sub(a.y, a.x);
  neg();
  return *this;
}

inline binder_arguments<Integer, SignDigit, Integer_minus_tag>
 operator-(const Integer& a, const SignDigit& b)
// Algorithm:  c := a-b
// Input:      a in Integer, SignDigit b.
// Output:     c in Integer such that c = a-b ||
{
  return binder_arguments<Integer, SignDigit, Integer_minus_tag>(a, b);
}

inline binder_arguments<SignDigit, Integer, Integer_minus_tag>
 operator-(const SignDigit& a, const Integer& b)
// Algorithm:  c := a-b
// Input:      a in SignDigit, Integer b.
// Output:     c in Integer such that c = a+b ||
{
  return binder_arguments<SignDigit, Integer, Integer_minus_tag>(a, b);
}
◇
```

Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer und $a \in$ SignDigit, dann u+$a \sim v$.sub(u,a).

Zuweisungsoperatoren

⟨ assign operator-= for Integers 209a ⟩ ≡

```
inline Integer& Integer::operator-=(const Integer& a)
// Algorithm:  c := c -= a
// Input:      a,c in Integer.
// Output:     c in Integer such that c := c-a ||
{
  sub(*this, a);
  return *this;
}
◇
```

Macro referenced in 419.

Laufzeit. Seien $u, v, w \in$ Integer, dann u-=$v \sim w$.sub(u,v).

⟨ assign operator-= of an Integer with a Natural 209b ⟩ ≡

```
inline Integer& Integer::operator-=(const Natural& a)
// Algorithm:  c := c -= a
// Input:      a in Natural, c in Integer.
// Output:     c in Integer such that c := c-a ||
{
  sub(*this, a);
  return *this;
}
◇
```

Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer und $x \in$ Natural, dann u`-=`$x \sim v$`.sub(`u`,`x`)`.

⟨ assign `operator-=` of an Integer with a SignDigit 210a ⟩ ≡

```
Integer& Integer::operator-=(const SignDigit a)
// Algorithm:  c := c -= a
// Input:      a SignDigit, c in Integer.
// Output:     c in Integer such that c := c-a ||
{
  INTEGERCONDITION(*this);

  if (a < 0) return *this += -a;
  const int sT = sgn;
  if (sT < 0) abs() += a;
  else if (sT == 0) neg(a);
  else if (::abs(*this) > Digit(a)) abs() -= a;
  else {
    *this = a - highest();
    neg();
  }

  INTEGERCONDITION(*this);

  return *this;
}
```

◇

Macro referenced in 426.

Laufzeit. Sei $u \in$ Integer und $a \in$ SignDigit, dann u`-=`$a \sim \begin{cases} \texttt{abs}(u)\texttt{-=a}\,, & \texttt{sign}(u) \geq 0 \\ \texttt{abs}(u)\texttt{+=a}\,, & \texttt{sign}(u) < 0 \end{cases}$.

3.2.10 Schiebeoperationen

Linksverschiebung

⟨ multiplication of an Integer by a power of 2 210b ⟩ ≡

```
inline void Integer::lshift(const Integer& a, const size_t b)
// Algorithm:  c.lshift(a, b)
// Input:      a in Integer, b in size_t.
// Output:     c in Integer such that c = a*2^b ||
{
  abs() = ::abs(a) << b;
  sgn = a.sgn;
}
```

◇

Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer und $t \in$ `size_t`, dann

$$v\texttt{.lshift(}u\texttt{,}t\texttt{)} \sim \texttt{abs(}v\texttt{)=abs(}u\texttt{)<<}t.$$

⟨ shift `operator<<` of an Integer 210c ⟩ ≡

```
inline Integer::Integer(const binder_arguments<Integer, size_t,
                                               Integer_lshift_tag>& a)
 : Natural(' ', ' ')
{
  get_memory(a.x.length()+a.y%BETA+DELTA);
  lshift(a.x, a.y);
}

inline Integer& Integer::operator=(const binder_arguments<Integer, size_t,
                                                          Integer_lshift_tag>& a)
{
  lshift(a.x, a.y);
  return *this;
}

inline binder_arguments<Integer, size_t, Integer_lshift_tag>
 operator<<(const Integer& a, const size_t& b)
// Algorithm:  c := a << b
// Input:      a in Integer, b in size_t.
// Output:     c in Integer such that c = a*2^b ||
{
  return binder_arguments<Integer, size_t, Integer_lshift_tag>(a, b);
}
◇
```

Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer und $t \in$ size_t, dann u<<$t \sim v$.lshift(u,t).

Zuweisungsoperator

⟨ assign operator<<= of an Integer 211 ⟩ ≡

```
inline Integer& Integer::operator<<=(const size_t a)
// Algorithm:  c := c <<= a
// Input:      a in size_t, c in Integer.
// Output:     c in Integer such that c := c*2^a ||
{
  abs() <<= a;
  return *this;
}
◇
```

Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer und $t \in$ size_t, dann u<<=$t \sim$ abs(u)<<=t.

Rechtsverschiebung

⟨ division of an Integer by a power of 2 212a ⟩ ≡

```
inline void Integer::rshift(const Integer& a, const size_t b)
// Algorithm:  c.rshift(a, b)
// Input:      a in Integer, b in size_t.
// Output:     c in Integer such that c = sign(a)*[|a|/2^b] ||
{
  abs() = ::abs(a) >> b;
  if (::abs(*this) == 0) sgn = 0;
  else sgn = a.sgn;
}
◇
```

Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer und $t \in$ size_t, dann

$$v\texttt{.rshift(}u\texttt{,}t\texttt{)} \sim \texttt{abs(}v\texttt{)=abs(}u\texttt{)>>}t.$$

⟨ shift operator>> of an Integer 212b ⟩ ≡

```
inline Integer::Integer(const binder_arguments<Integer, size_t,
                                               Integer_rshift_tag>& a)
 : Natural(' ', ' ')
{
  get_memory(a.x.length()+DELTA);
  rshift(a.x, a.y);
}

inline Integer& Integer::operator=(const binder_arguments<Integer, size_t,
                                                          Integer_rshift_tag>& a)
{
  rshift(a.x, a.y);
  return *this;
}

inline binder_arguments<Integer, size_t, Integer_rshift_tag>
 operator>>(const Integer& a, const size_t& b)
// Algorithm:  c := a >> b
// Input:      a in Integer, b in size_t.
// Output:     c in Integer such that c = sign(a)*[|a|/2^b] ||
{
  return binder_arguments<Integer, size_t, Integer_rshift_tag>(a, b);
}
◇
```

Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer und $t \in$ size_t, dann $u\texttt{>>}t \sim v\texttt{.rshift(}u\texttt{,}t\texttt{)}$.

Zuweisungsoperator

⟨ assign operator>>= of an Integer 212c ⟩ ≡

```
inline Integer& Integer::operator>>=(const size_t a)
// Algorithm:  c := c >>= a
// Input:      a in size_t, c in Integer.
// Output:     c in Integer such that c := sign(c)*[|c|/2^a] ||
```

```
{
  abs() >>= a;
  if (::abs(*this) == 0) sgn = 0;
  return *this;
}
◇
```

Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer und $t \in$ size_t, dann u>>=$t \sim$ abs(u)>>=t.

3.2.11 Bitweise Verknüpfungen

Wir verwenden bei den bitweisen Verknüpfungen die Darstellung des 2^n-Komplements (siehe Abschnitt 3.1.1, Seite 185) für negative Zahlen, um die gleichen Eigenschaften wie die eingebauten Typen (zum Beispiel int) zu besitzen. Dadurch unterstützen wir die folgenden drei Rechenregeln:

$$\begin{aligned} -a &= \tilde{}(a-1) && (3.1)\\ &= \tilde{}a+1 && (3.2)\\ \tilde{}(a \;\&\; b) &= \tilde{}a \mid \tilde{}b && (3.3)\\ \tilde{}(a \mid b) &= \tilde{}a \;\&\; \tilde{}b && (3.4) \end{aligned}$$

Bitweise UND-Verknpüpfung

⟨ bitwise and of two Integers 213 ⟩ ≡

```
void Integer::bitwise_and(const Integer& a, const Integer& b)
// Algorithm:  c.bitwise_and(a, b)
// Input:      a,b in Integer.
// Output:     c in Integer such that c = a and b ||
{
  INTEGERCONDITION(a);
  INTEGERCONDITION(b);

  const int sA = a.sgn;
  const int sB = b.sgn;
  if (sA >= 0) {
    if (sB >= 0) {
      abs() = ::abs(a) & ::abs(b);
      sgn = (::abs(*this) != 0);
    } else {
      ⟨ case 1 of bitwise_and 214a ⟩
    }
  } else if (sB < 0) {
    ⟨ case 2 of bitwise_and 214c ⟩
  } else {
    ⟨ case 3 of bitwise_and 214b ⟩
  }

  INTEGERCONDITION(*this);
}
◇
```

Macro referenced in 426.

Laufzeit. Seien $u, v, w \in$ Integer, dann

$$w\texttt{.bitwise\_and(}u\texttt{,}v\texttt{)} \sim \texttt{abs(}w\texttt{)=abs(}u\texttt{)\&abs(}v\texttt{)}.$$

Falls die beiden Eingabeargumente der Funtion `bitwise_and` positiv sind, so werden die Argumente einfach an die Funktion `bitwise_and` für Naturals weitergeleitet. Andernfalls müssen wir die folgenden beiden Fälle betrachten:

1. Es gilt die folgende Rechenregel, falls genau ein Argument negativ ist:

$$\texttt{a \& -b} \underset{(3.2)}{=} \texttt{a \& (\~{}b+1)}$$

   Und wir erhalten das Programm:

   ⟨ case 1 of bitwise_and 214a ⟩ ≡

```
const Natural c = ::abs(a);
Natural::bitwise_not(::abs(b));
++abs();
const size_t sC = c.length();
const size_t sT = length();
abs() &= c;
if (sT < sC) { copy(c, sC-sT); sgn = 1; }
else sgn = (::abs(*this) != 0);◇
```

   Macro referenced in 213.

   Wir vertauschen die Aufgaben von `a` und `b`, falls nur das Argument `a` negativ ist.

   ⟨ case 3 of bitwise_and 214b ⟩ ≡

```
const Natural c = ::abs(b);
Natural::bitwise_not(::abs(a));
++abs();
const size_t sC = c.length();
const size_t sT = length();
abs() &= c;
if (sT < sC) { copy(c, sC-sT); sgn = 1; }
else sgn = (::abs(*this) != 0);◇
```

   Macro referenced in 213.

2. Falls beide Argumente negativ sind, wird der Rechenaufwand ein wenig größer:

$$\begin{aligned}
\texttt{-a \& -b} &\underset{(3.1)}{=} \texttt{\~{}(a-1) \& \~{}(b-1)} \\
&\underset{(3.4)}{=} \texttt{\~{}((a-1) | (b-1))} \\
&\underset{(3.2)}{=} \texttt{-(((a-1) | (b-1))+1)} \\
&= \texttt{-(((a-1) | (b-1))+1)}
\end{aligned}$$

   Der Programm-Code ist aber sehr elementar:

   ⟨ case 2 of bitwise_and 214c ⟩ ≡

```
Natural c = b;
--c;
*this = a;
--abs(); abs() |= c; ++abs();◇
```

Macro referenced in 213.

⟨ bitwise operator& for Integers 215a ⟩ ≡

```
inline Integer::Integer(const binder_arguments<Integer, Integer,
                                                Integer_and_tag>& a)
 : Natural(' ', ' ')
{
  get_memory(min(a.x.length(), a.y.length())+DELTA);
  bitwise_and(a.x, a.y);
}

inline Integer& Integer::operator=(const binder_arguments<Integer, Integer,
                                                          Integer_and_tag>& a)
{
  bitwise_and(a.x, a.y);
  return *this;
}

inline binder_arguments<Integer, Integer, Integer_and_tag>
 operator&(const Integer& a, const Integer& b)
// Algorithm:  c := a & b
// Input:      a,b in Integer.
// Output:     c in Integer such that c = a and b ||
{
  return binder_arguments<Integer, Integer, Integer_and_tag>(a, b);
}
```

◇

Macro referenced in 419.

Laufzeit. Seien $u, v, w \in$ Integer, dann $u\&v \sim w$.bitwise_and(u,v).

⟨ bitwise operator& of an Integer with a Digit 215b ⟩ ≡

```
inline Digit operator&(const Integer& a, const Digit b)
// Algorithm:  c := a & b
// Input:      a in Integer, b in Digit.
// Output:     c in Integer such that c = a and b ||
{
  return (sign(a) >= 0)? abs(a)&b : ((~a.lowest())+1)&b;
}
```

◇

Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer und $a \in$ Digit, dann $u\&a \sim$ abs(u)&a.

Zuweisungsoperator

⟨ assign operator&= for Integers 215c ⟩ ≡

```
inline Integer& Integer::operator&=(const Integer& a)
// Algorithm:  c := c &= a
// Input:      a,c in Integer.
// Output:     c in Integer such that c := c and a ||
{
  bitwise_and(*this, a);
  return *this;
}
```

◇

Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer, dann u`&=`$v \sim u$`.bitwise_and(`u`,`v`)`.

⟨ assign `operator&=` of an Integer with a Digit 216a ⟩ ≡

```
Digit Integer::operator&=(const Digit b)
// Algorithm:  c := a &= b
// Input:      a in Integer, b in Digit.
// Output:     a in Integer, c in Digit such that a := a and b, c = a ||
{
  INTEGERCONDITION(*this);

  Digit c;
  if (sgn >= 0) {
    c = abs() &= b;
    sgn = (c != 0);
  } else *this = c = ((~lowest())+1)&b;

  INTEGERCONDITION(*this);

  return c;
}
```

◇

Macro referenced in 426.

Laufzeit. Sei $u \in$ Integer und $a \in$ Digit, dann u`&=`$a \sim$ `abs(`u`)&=`a.

Bitwise inklusive ODER-Verknpüpfung

⟨ bitwise inclusive or of two Integers 216b ⟩ ≡

```
void Integer::bitwise_or(const Integer& a, const Integer& b)
// Algorithm:  c.bitwise_or(a, b)
// Input:      a,b in Integer.
// Output:     c in Integer such that c = a or b ||
{
  INTEGERCONDITION(a);
  INTEGERCONDITION(b);

  const int sA = a.sgn;
  const int sB = b.sgn;
  if (sA >= 0) {
    if (sB >= 0) {
      abs() = ::abs(a) | ::abs(b);
      sgn = (::abs(*this) != 0);
    } else {
      ⟨ case 1 of bitwise_or 217a ⟩
    }
  } else if (sB < 0) {
    ⟨ case 2 of bitwise_or 218a ⟩
  } else {
    ⟨ case 3 of bitwise_or 217b ⟩
  }

  INTEGERCONDITION(*this);
}
```

◇

Macro referenced in 426.

Laufzeit. Seien $u, v, w \in$ Integer, dann

$$w\texttt{.bitwise\_or(}u\texttt{,}v\texttt{)} \sim \texttt{abs(}w\texttt{)=abs(}u\texttt{)|abs(}v\texttt{)}.$$

Falls die beiden Eingabeargumente der Funtion `bitwise_or` positiv sind, so werden die Argumente einfach an die Funktion `bitwise_or` für Naturals weitergeleitet. Ansonsten müssen wir die folgenden beiden Fälle betrachten:

1. Es gilt die folgende Rechnung, falls genau ein Argument negativ ist:

$$\begin{aligned} \texttt{a | -b} &\underset{(3.1)}{=} \texttt{a | ~(b-1)} \\ &\underset{(3.3)}{=} \texttt{~(~a \& (b-1))} \\ &\underset{(3.2)}{=} \texttt{-(~a \& (b-1))-1} \end{aligned}$$

   Und wir erhalten das Programm:

   ⟨ case 1 of bitwise_or 217a ⟩ ≡

```
Natural c = b;
--c;
Natural::bitwise_not(::abs(a));
const size_t sC = c.length();
const size_t sT = length();
abs() &= c;
if (sT < sC) { copy(c, sC-sT); sgn = -1; }
else { const int b = (::abs(*this) != 0); sgn = -b; }
--(*this);◇
```

   Macro referenced in 216b.

   Wir vertauschen die Aufgaben von `a` und `b`, falls nur das Argument `a` negativ ist.

   ⟨ case 3 of bitwise_or 217b ⟩ ≡

```
Natural c = a;
--c;
Natural::bitwise_not(::abs(b));
const size_t sC = c.length();
const size_t sT = length();
abs() &= c;
if (sT < sC) { copy(c, sC-sT); sgn = -1; }
else { const int b = (::abs(*this) != 0); sgn = -b; }
--(*this);◇
```

   Macro referenced in 216b.

2. Falls beide Argumente negativ sind, so führen wir die folgende Rechnung durch:

$$\begin{aligned} \texttt{-a | -b} &\underset{(3.1)}{=} \texttt{~(a-1) | ~(b-1)} \\ &\underset{(3.3)}{=} \texttt{~((a-1) \& (b-1))} \\ &\underset{(3.2)}{=} \texttt{-(((a-1) \& (b-1))+1)} \end{aligned}$$

   und erhalten das Programm:

⟨ case 2 of bitwise_or 218a ⟩ ≡

```
Natural c = b;
--c;
*this = a;
--abs(); abs() &= c; ++abs();◇
```

Macro referenced in 216b.

⟨ bitwise operator| for Integers 218b ⟩ ≡

```
inline Integer::Integer(const binder_arguments<Integer, Integer,
                                                Integer_or_tag>& a)
 : Natural(' ', ' ')
{
  get_memory(max(a.x.length(), a.y.length())+DELTA);
  bitwise_or(a.x, a.y);
}

inline Integer& Integer::operator=(const binder_arguments<Integer, Integer,
                                                          Integer_or_tag>& a)
{
  bitwise_or(a.x, a.y);
  return *this;
}

inline binder_arguments<Integer, Integer, Integer_or_tag>
 operator|(const Integer& a, const Integer& b)
// Algorithm:  c := a | b
// Input:      a,b in Integer.
// Output:     c in Integer such that c = a or b ||
{
  return binder_arguments<Integer, Integer, Integer_or_tag>(a, b);
}
◇
```

Macro referenced in 419.

Laufzeit. Seien $u, v, w \in$ Integer, dann $u|v \sim w$.bitwise_or(u,v).

⟨ bitwise operator| of an Integer with a Digit 218c ⟩ ≡

```
inline Integer operator|(const Integer& a, const Digit b)
// Algorithm:  c := a | b
// Input:      a in Integer, b in Digit.
// Output:     c in Integer such that c = a or b ||
{
  return Integer(a) |= b;
}
◇
```

Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer und $a \in$ Digit, dann $u|a \sim u|=a$.

Zuweisungsoperator

⟨ assign operator|= for Integers 219a ⟩ ≡

```
inline Integer& Integer::operator|=(const Integer& a)
// Algorithm:  c := c |= a
// Input:      a,c in Integer.
// Output:     c in Integer such that c := c or a ||
{
  bitwise_or(*this, a);
  return *this;
}
```

◇

Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer, dann u|=$v \sim u$.bitwise_or(u,v).

⟨ assign operator|= of an Integer with a Digit 219b ⟩ ≡

```
inline Integer& Integer::operator|=(const Digit a)
// Algorithm:  c := c |= a
// Input:      c in Integer, a in Digit.
// Output:     c in Integer such that c := c or a ||
{
  if (sgn >= 0) abs() |= a;
  else *this |= Integer(a);
  return *this;
}
```

◇

Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer und $a \in$ Digit, dann u|=$a \sim$ abs(u)|=a.

Bitwise exklusive ODER-Verknpüpfung

⟨ bitwise exclusive or of two Integers 219c ⟩ ≡

```
void Integer::bitwise_xor(const Integer& a, const Integer& b)
// Algorithm:  c.bitwise_xor(a, b)
// Input:      a,b in Integer.
// Output:     c in Integer such that c = a xor b ||
{
  INTEGERCONDITION(a);
  INTEGERCONDITION(b);

  if (a.sgn >= 0 && b.sgn >= 0) {
    abs() = ::abs(a) ^ ::abs(b);
    sgn = (::abs(*this) != 0);
  } else {
    Integer s,t;
    s.bitwise_not(a); t.bitwise_not(b);
    s &= b; t &= a;
    bitwise_or(s, t);
  }

  INTEGERCONDITION(*this);
}
```

◇

Macro referenced in 426.

Laufzeit. Seien $u, v, w \in$ Integer, dann

$$w.\texttt{bitwise\_xor}(u,v) \sim \texttt{abs}(w)=\texttt{abs}(u)\hat{}\texttt{abs}(v).$$

⟨ bitwise operator^ for Integers 220a ⟩ ≡

```
inline Integer::Integer(const binder_arguments<Integer, Integer,
                                                Integer_xor_tag>& a)
 : Natural(' ', ' ')
{
  get_memory(max(a.x.length(), a.y.length())+DELTA);
  bitwise_xor(a.x, a.y);
}

inline Integer& Integer::operator=(const binder_arguments<Integer, Integer,
                                                          Integer_xor_tag>& a)
{
  bitwise_xor(a.x, a.y);
  return *this;
}

inline binder_arguments<Integer, Integer, Integer_xor_tag>
 operator^(const Integer& a, const Integer& b)
// Algorithm:  c := a ^ b
// Input:      a,b in Integer.
// Output:     c in Integer such that c = a xor b ||
{
  return binder_arguments<Integer, Integer, Integer_xor_tag>(a, b);
}
◇
```

Macro referenced in 419.

Laufzeit. Seien $u, v, w \in$ Integer, dann $u\hat{}v \sim w.\texttt{bitwise\_xor}(u,v)$.

Zuweisungsoperator

⟨ assign operator^= for Integers 220b ⟩ ≡

```
inline Integer& Integer::operator^=(const Integer& a)
// Algorithm:  c := c ^= a
// Input:      a,c in Integer.
// Output:     c in Integer such that c := c xor a ||
{
  bitwise_xor(*this, a);
  return *this;
}
◇
```

Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer, dann $u\hat{}=v \sim u.\texttt{bitwise\_xor}(u,v)$.

Bitwise Negation

⟨ bitwise not of an Integer 221a ⟩ ≡

```
inline void Integer::bitwise_not(const Integer& a)
// Algorithm:  b.not(a)
// Input:      a in Integer.
// Output:     b in Integer such that b = not a = -a-1 ||
{
  *this = a; neg(); --*this;
}
◇
```

Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer, dann

$$v\texttt{.bitwise\_not}(u) \sim u\texttt{=}v.$$

⟨ bitwise operator~ of an Integer 221b ⟩ ≡

```
inline Integer::Integer(const binder_arguments<Integer, Integer,
                                               Integer_not_tag>& a)
 : Natural(' ', ' ')
{
  get_memory(a.x.length()+DELTA);
  bitwise_not(a.x);
}

inline Integer& Integer::operator=(const binder_arguments<Integer, Integer,
                                                          Integer_not_tag>& a)
{
  bitwise_not(a.x);
  return *this;
}

inline binder_arguments<Integer, Integer, Integer_not_tag>
 operator~(const Integer& a)
// Algorithm:  c := ~a
// Input:      a in Integer.
// Output:     c in Integer such that c = not a ||
{
  return binder_arguments<Integer, Integer, Integer_not_tag>(a, a);
}
◇
```

Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer, dann $\texttt{~}u \sim v\texttt{.bitwise\_not}(u)$.

3.2.12 Bitoperationen

Bei den Bitoperationen gilt genauso wie bei den bitweisen Verknüpfungen (siehe Abschnitt 3.2.11, Seite 213) die Darstellung des 2^n-Komplements für negative Zahlen.

⟨ sets a bit in an Integer 221c ⟩ ≡

```
void Integer::setbit(const size_t a)
// Algorithm:  c.setbit(a)
```

```
// Input:      a in size_t, c in Integer.
// Output:     c in Integer such that c := c or 2^a ||
{
  INTEGERCONDITION(*this);
  INTEGER_FOR_CHECK(_t, *this);

  if (sgn >= 0) Natural::setbit(a);
  else {
    const size_t b = a/BETA;
    if (b <= length()) {
      Digit* pT;
      const size_t c = trailing_zeros(pT);
      if (c < b) Natural::clearbit(a);
      else if (c == b) *pT = ((*pT-1) & ~(Digit(1)<<(a%BETA)))+1;
      else {
        pT += c-b;                    // not very good!
        *pT = ~(Digit(1) << (a%BETA))+1;
        dec(pT);
      }
    }
  }

  CONDITION(*this == (_t | Integer(1) << a));
  INTEGERCONDITION(*this);
}
◇
```

Macro referenced in 426.

Laufzeit. Sei $u \in$ Integer und $t \in$ size_t, dann u.setbit(t)$\sim abs$(u).$setbit$(t).

⟨ clears a bit in an Integer 222 ⟩ ≡

```
void Integer::clearbit(const size_t a)
// Algorithm:  c.clearbit(a)
// Input:      a in size_t, c in Integer.
// Output:     c in Integer such that c := c and not(2^a) ||
{
  INTEGERCONDITION(*this);
  INTEGER_FOR_CHECK(_t, *this);

  if (sgn >= 0) Natural::clearbit(a);
  else {
    const size_t b = a/BETA;
    Digit* pT;
    const size_t c = trailing_zeros(pT);
    if (c < b) Natural::setbit(a);
    else if (c == b) *pT = ((*pT-1) | (Digit(1)<<(a%BETA)))+1;
  }

  CONDITION(*this == (_t & ~(Integer(1) << a)));
  INTEGERCONDITION(*this);
}
◇
```

Macro referenced in 426.

Laufzeit. Sei $u \in$ Integer und $t \in$ size_t, dann u.clearbit(t)$\sim abs$(u).$clearbit$(t).

⟨ tests a bit in an Integer 223a ⟩ ≡

```
bool Integer::testbit(const size_t a) const
// Algorithm:  c := b.testbit(a)
// Input:      a in size_t, b in Integer.
// Output:     c in bool such that if b and 2^a then c = true else c = false ||
{
  INTEGERCONDITION(*this);

  if (sgn >= 0) return Natural::testbit(a);
  const size_t b = a/BETA;
  Digit* pT;
  const size_t c = trailing_zeros(pT);
  if (c < b) return !Natural::testbit(a);
  else if (c > b) return false;
  else return (((*pT-1) & (Digit(1)<<(a%BETA))) == 0);
}
```

◇

Macro referenced in 426.

Laufzeit. Sei $u \in$ Integer und $t \in$ `size_t`, dann u`.testbit(`t`)`$\sim abs(\mathtt{u}).testbit(\mathtt{t})$.

3.2.13 Multiplikation

⟨ multiplication of an Integer by a SignDigit 223b ⟩ ≡

```
inline void Integer::mul(const Integer& a, const SignDigit b)
// Algorithm:  c.mul(a, b)
// Input:      a in Integer, b in SignDigit.
// Output:     c in Integer such that c = a*b ||
{
  if (b > 0) {
    abs() = ::abs(a) * Digit(b);
    sgn = a.sgn;
  } else if (b < 0) {
    abs() = ::abs(a) * Digit(-b);
    sgn = -a.sgn;
  } else *this = 0;
}
```

◇

Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer und $a \in$ SignDigit, dann

$$v\texttt{.mul(}u\texttt{,}a\texttt{)} \sim \texttt{abs(}v\texttt{)=abs(}u\texttt{)*}a.$$

⟨ multiplication of an Integer by a Natural 223c ⟩ ≡

```
inline void Integer::mul(const Integer& a, const Natural& b)
// Algorithm:  c.mul(a, b)
// Input:      a in Integer, b in Natural.
// Output:     c in Integer such that c = a*b ||
{
  abs() = ::abs(a) * b;
  sgn = a.sgn;
}
```

◇

Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer und $x \in$ Natural, dann

$$v.\texttt{mul}(u,x) \sim \texttt{abs}(v)=\texttt{abs}(u)*x.$$

⟨ multiplication of two Integers 224a ⟩ ≡

```
inline void Integer::mul(const Integer& a, const Integer& b)
// Algorithm:  c.mul(a, b)
// Input:      a,b in Integer.
// Output:     c in Integer such that c = a*b ||
{
  abs() = ::abs(a) * ::abs(b);
  sgn = a.sgn*b.sgn;
}
◇
```

Macro referenced in 419.

Laufzeit. Seien $u, v, w \in$ Integer, dann

$$w.\texttt{mul}(u,v) \sim \texttt{abs}(w)=\texttt{abs}(u)*\texttt{abs}(v).$$

⟨ multiplicative operator* for Integers 224b ⟩ ≡

```
inline Integer::Integer(const binder_arguments<Integer, Integer,
                                               Integer_multiplies_tag>& a)
 : Natural(' ', ' ')
{
  get_memory(a.x.length()+a.y.length()+DELTA);
  mul(a.x, a.y);
}

inline Integer& Integer::operator=(const binder_arguments<Integer, Integer,
                                                          Integer_multiplies_tag>& a)
{
  mul(a.x, a.y);
  return *this;
}

inline binder_arguments<Integer, Integer, Integer_multiplies_tag>
 operator*(const Integer& a, const Integer& b)
// Algorithm:  c := a*b
// Input:      a,b in Integer.
// Output:     c in Integer such that c = a*b ||
{
  return binder_arguments<Integer, Integer, Integer_multiplies_tag>(a, b);
}
◇
```

Macro referenced in 419.

Laufzeit. Seien $u, v, w \in$ Integer, dann $u*v \sim w.\texttt{mul}(u,v)$.

⟨ multiplicative operator* of an Integer with a Natural 224c ⟩ ≡

```
inline Integer::Integer(const binder_arguments<Integer, Natural,
                                               Integer_multiplies_tag>& a)
 : Natural(' ', ' ')
{
```

```
  get_memory(a.x.length()+a.y.length()+DELTA);
  mul(a.x, a.y);
}

inline Integer& Integer::operator=(const binder_arguments<Integer, Natural,
                                                          Integer_multiplies_tag>& a)
{
  mul(a.x, a.y);
  return *this;
}

inline binder_arguments<Integer, Natural, Integer_multiplies_tag>
 operator*(const Integer& a, const Natural& b)
// Algorithm:  c := a*b
// Input:      a in Integer, b in Natural.
// Output:     c in Integer such that c = a*b ||
{
  return binder_arguments<Integer, Natural, Integer_multiplies_tag>(a, b);
}

inline binder_arguments<Integer, Natural, Integer_multiplies_tag>
 operator*(const Natural& a, const Integer& b)
// Algorithm:  c := a*b
// Input:      a in Natural, b in Integer.
// Output:     c in Integer such that c = a*b ||
{
  return binder_arguments<Integer, Natural, Integer_multiplies_tag>(b, a);
}
◇
```

Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer und $x \in$ Natural, dann $u*x \sim v.\texttt{mul}(u,x)$.

⟨ multiplicative operator* of an Integer with a SignDigit 225 ⟩ ≡

```
inline Integer::Integer(const binder_arguments<Integer, SignDigit,
                                               Integer_multiplies_tag>& a)
 : Natural(' ', ' ')
{
  get_memory(a.x.length()+DELTA);
  mul(a.x, a.y);
}

inline Integer& Integer::operator=(const binder_arguments<Integer, SignDigit,
                                                          Integer_multiplies_tag>& a)
{
  mul(a.x, a.y);
  return *this;
}

inline Integer& Integer::operator+=(const binder_arguments<Integer, SignDigit,
                                                           Integer_multiplies_tag>& a)
{
  muladd(a.x, a.y);
  return *this;
}
```

```
inline Integer& Integer::operator-=(const binder_arguments<Integer, SignDigit,
                                                           Integer_multiplies_tag>& a)
{
  mulsub(a.x, a.y);
  return *this;
}

inline binder_arguments<Integer, SignDigit, Integer_multiplies_tag>
 operator*(const Integer& a, const SignDigit& b)
// Algorithm:  c := a*b
// Input:      a in Integer, b in SignDigit.
// Output:     c in Integer such that c = a*b ||
{
  return binder_arguments<Integer, SignDigit, Integer_multiplies_tag>(a, b);
}

inline binder_arguments<Integer, SignDigit, Integer_multiplies_tag>
 operator*(const SignDigit& a, const Integer& b)
// Algorithm:  c := a*b
// Input:      a in SignDigit, b in Integer.
// Output:     c in Integer such that c = a*b ||
{
  return binder_arguments<Integer, SignDigit, Integer_multiplies_tag>(b, a);
}
◇
```

Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer und $a \in$ SignDigit, dann u*$a \sim u$*=a.

Zuweisungsoperatoren

⟨ assign operator*= for Integers 226 ⟩ ≡

```
inline Integer& Integer::operator*=(const Integer& a)
// Algorithm:  c := c *= a
// Input:      a,c in Integer.
// Output:     c in Integer such that c := c*a ||
{
  sgn *= a.sgn;
  abs() *= ::abs(a);
  return *this;
}
◇
```

Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer, dann u*=$v \sim$ abs(u)*=abs(v).

⟨ assign operator*= of an Integer with a Natural 227a ⟩ ≡
```
inline Integer& Integer::operator*=(const Natural& a)
// Algorithm:  c := c *= a
// Input:      a in Natural, c in Integer.
// Output:     c in Integer such that c := c*a ||
{
  if (a != 0) abs() *= a;
  else *this = 0;
  return *this;
}
```
◇
Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer und $x \in$ Natural, dann u*=$x \sim$ abs(u)*=x.

⟨ assign operator*= of an Integer with a SignDigit 227b ⟩ ≡
```
inline Integer& Integer::operator*=(const SignDigit a)
// Algorithm:  c := c *= a
// Input:      a in SignDigit, c in Integer.
// Output:     c in Integer such that c := c*a ||
{
  if (a > 0) abs() *= a;
  else if (a < 0) { abs() *= -a; sgn = -sgn; }
  else *this = 0;
  return *this;
}
```
◇
Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer und $a \in$ SignDigit, dann u*=$a \sim$ abs(u)*=a.

Gleichzeitige Multiplikation und Addition

⟨ multiplication and addition of an Integer by a SignDigit 227c ⟩ ≡
```
void Integer::muladd(const Integer& a, const SignDigit b)
// Algorithm:  c.muladd(a, b)
// Input:      a,c in Integer, b in SignDigit.
// Output:     c in Integer such that c := c + a*b ||
{
  INTEGERCONDITION(*this);
  INTEGERCONDITION(a);

  if (b < 0) mulsub(a, -b);
  else {
    const int sT = sgn;
    if (sT == 0) mul(a, b);
    else if (sT == a.sgn) Natural::muladd(::abs(a), b);
    else if (a.length()+1 < length()) Natural::mulsub(::abs(a), b);
    else {
      Integer c = a*b;
      *this += c;
    }
  }

  INTEGERCONDITION(*this);
}
```

◇

Macro referenced in 426.

Laufzeit. Seien $u, v \in$ Integer und $a \in$ SignDigit, dann u.muladd(v,a) $\sim$ u.muladd(abs(v),a).

Gleichzeitige Multiplikation und Subtraktion

⟨ multiplication and subtraction of an Integer by a SignDigit 228a ⟩ ≡

```
void Integer::mulsub(const Integer& a, const SignDigit b)
// Algorithm:  c.mulsub(a, b)
// Input:      a,c in Integer, b in SignDigit.
// Output:     c in Integer such that c := c - a*b ||
{
  INTEGERCONDITION(*this);
  INTEGERCONDITION(a);

  if (b < 0) muladd(a, -b);
  else {
    const int sT = sgn;
    if (sT == 0) { mul(a, b); neg(); }
    else if (sT != a.sgn) Natural::muladd(::abs(a), b);
    else if (a.length()+1 < length()) Natural::mulsub(::abs(a), b);
    else {
      Integer c = a*b;
      *this -= c;
    }
  }

  INTEGERCONDITION(*this);
}
```

◇

Macro referenced in 426.

Laufzeit. Seien $u, v \in$ Integer und $a \in$ SignDigit, dann u.mulsub(v,a) $\sim$ u.mulsub(abs(v),a).

3.2.14 Binärer Logarithmus

⟨ binary logarithm for Integers 228b ⟩ ≡

```
inline Digit log2(const Integer& a)
// Algorithm:  b := log2(a)
// Input:      a in Integer.
// Output:     b in Digit
//             such that if |a| > 0 then b = [log2(|a|)] else b = 0 ||
{
  return log2(abs(a));
}
```

◇

Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer, dann log2(u) $\sim$ log2(abs(u)).

3.2.15 Potenzieren

⟨ power of an Integer by a SignDigit 229a ⟩ ≡

```
Integer pow(const Integer& a, const SignDigit b)
// Algorithm:  c := pow(a, b)
// Input:      a in Integer, b in SignDigit.
// Output:     c in Integer such that c = a^b ||
{
  INTEGERCONDITION(a);

  if (b < 0 && abs(a) != 1) return SignDigit(0);
  Integer c(pow(abs(a), Digit(b)));
  c.sgn = (abs(c) != 0);
  if (a.sgn == -1 && (b&1)) c.neg();

  INTEGERCONDITION(c);

  return c;
}
◇
```

Macro referenced in 426.

Laufzeit. Sei $u \in$ Integer und $a \in$ SignDigit, dann `pow(`u`,`a`)` $\sim$ `pow(abs(`u`),`a`)`.

⟨ power of an Integer by an Integer 229b ⟩ ≡

```
Integer pow(const Integer& a, const Integer& b)
// Algorithm:  c := pow(a, b)
// Input:      a,b in Integer.
// Output:     c in Integer such that c = sign(a)^b*[|a|^b] ||
{
  INTEGERCONDITION(a);
  INTEGERCONDITION(b);

  if (b < 0 && abs(a) != 1) return SignDigit(0);
  Integer c;
  c.abs() = pow(abs(a), abs(b));
  c.sgn = (abs(c) != 0);
  if (a.sgn == -1 && (b&1)) c.neg();

  INTEGERCONDITION(c);

  return c;
}
◇
```

Macro referenced in 426.

Laufzeit. Seien $u, v \in$ Integer, dann `pow(`u`,`v`)` $\sim$ `pow(abs(`u`),abs(`v`))`.

3.2.16 Division

⟨ division of two Integers with remainder 229c ⟩ ≡

```
void div(const Integer& a, const Integer& b, Integer& c, Integer& d)
// Algorithm:  div(a, b, c, d)
// Input:      a,b,c,d in Integer where not b = 0, not c = d;
// Output:     c,d in Integer such that c = sign(a*b)*[|a/b|], d = a-c*b ||
{
  INTEGERCONDITION(a);
  INTEGERCONDITION(b);
  INTEGER_FOR_CHECK(_a, a);
  INTEGER_FOR_CHECK(_b, b);

  div(abs(a), abs(b), c.abs(), d.abs());
  const int sA = a.sgn;
  const int sB = b.sgn;
  c.sgn = (abs(c) != 0)? sA*sB : 0;
  d.sgn = (abs(d) != 0)? sA : 0;

  CONDITION(d+c*_b == _a);
  INTEGERCONDITION(c);
  INTEGERCONDITION(d);
}
◇
```

Macro referenced in 426.

Laufzeit. Seien $u, v, w_0, w_1 \in$ Integer, dann

$$\texttt{div(}u\texttt{,}v\texttt{,}w_0\texttt{,}w_1\texttt{)} \sim \texttt{div(abs(}u\texttt{),abs(}v\texttt{),abs(}w_0\texttt{),abs(}w_1\texttt{))}.$$

⟨ division of two Integers 230a ⟩ ≡

```
void Integer::div(const Integer& a, const Integer& b)
// Algorithm:  c.div(a, b)
// Input:      a,b,c in Integer where not b = 0;
// Output:     c in Integer such that c = sign(a*b)*[|a/b|] ||
{
  INTEGERCONDITION(a);
  INTEGERCONDITION(b);

  Natural t;
  Natural::div(::abs(a), ::abs(b), t);
  sgn = (::abs(*this) != 0)? a.sgn*b.sgn : 0;

  INTEGERCONDITION(*this);
}
◇
```

Macro referenced in 426.

Laufzeit. Seien $u, v, w \in$ Integer, dann

$$w\texttt{.div(}u\texttt{,}v\texttt{)} \sim \texttt{abs(}w\texttt{).div(abs(}u\texttt{),abs(}v\texttt{))}.$$

⟨ multiplicative operator/ for Integers 230b ⟩ ≡

```
inline Integer::Integer(const binder_arguments<Integer, Integer,
                                               Integer_divides_tag>& a)
 : Natural(' ', ' ')
{
  get_memory(a.x.length()+DELTA);
```

```
  div(a.x, a.y);
}

inline Integer& Integer::operator=(const binder_arguments<Integer, Integer,
                                                          Integer_divides_tag>& a)
{
  div(a.x, a.y);
  return *this;
}

inline binder_arguments<Integer, Integer, Integer_divides_tag>
 operator/(const Integer& a, const Integer& b)
// Algorithm:  c := a/b
// Input:      a,b in Integer where not b = 0.
// Output:     c in Integer such that c = sign(a*b)*[|a/b|] ||
{
  return binder_arguments<Integer, Integer, Integer_divides_tag>(a, b);
}
◇
```

Macro referenced in 419.

Laufzeit. Seien $u, v, w \in$ Integer, dann $u/v \sim w$.div(u,v).

⟨ multiplicative operator% for Integers 231a ⟩ ≡

```
inline Integer::Integer(const binder_arguments<Integer, Integer,
                                               Integer_modulus_tag>& a)
 : Natural(' ', ' ')
{
  Integer t(a.x.length()+DELTA, ' ');
  get_memory(a.y.length()+DELTA);
  ::div(a.x, a.y, t, *this);
}

inline Integer& Integer::operator=(const binder_arguments<Integer, Integer,
                                                          Integer_modulus_tag>& a)
{
  Integer t(a.x.length()+DELTA, ' ');
  ::div(a.x, a.y, t, *this);
  return *this;
}

inline binder_arguments<Integer, Integer, Integer_modulus_tag>
 operator%(const Integer& a, const Integer& b)
// Algorithm:  c := a%b
// Input:      a,b in Integer where not b = 0.
// Output:     c in Integer such that c = a - sign(a*b)*[|a/b|]*b ||
{
  return binder_arguments<Integer, Integer, Integer_modulus_tag>(a, b);
}
◇
```

Macro referenced in 419.

Laufzeit. Seien $u, v, w_0, w_1 \in$ Integer, dann $u\%v \sim$ div(u,v,w_0,w_1).

⟨ division of an Integer by a Natural 231b ⟩ ≡

```
void Integer::div(const Integer& a, const Natural& b)
// Algorithm:  c.div(a, b)
// Input:      a,c in Integer, b in Natural where not b = 0;
// Output:     c in Integer such that c = sign(a)*[|a|/b] ||
{
  INTEGERCONDITION(a);

  Natural t;
  Natural::div(::abs(a), b, t);
  sgn = (::abs(*this) != 0)? a.sgn : 0;

  INTEGERCONDITION(*this);
}
◇
```

Macro referenced in 426.

Laufzeit. Seien $u, v \in$ Integer und $x \in$ Natural, dann

$$v.\texttt{div}(u,x) \sim \texttt{abs}(v).\texttt{div}(\texttt{abs}(u),x).$$

⟨ division of a Natural by an Integer 232a ⟩ ≡

```
void Integer::div(const Natural& a, const Integer& b)
// Algorithm:  c.div(a, b)
// Input:      a in Natural, b,c in Integer where not b = 0;
// Output:     c in Integer such that c = sign(b)*[a/|b|] ||
{
  INTEGERCONDITION(b);

  Natural t;
  Natural::div(a, ::abs(b), t);
  sgn = (::abs(*this) != 0)? b.sgn : 0;

  INTEGERCONDITION(*this);
}
◇
```

Macro referenced in 426.

Laufzeit. Seien $u, v \in$ Integer und $x \in$ Natural, dann

$$v.\texttt{div}(x,u) \sim \texttt{abs}(v).\texttt{div}(x,\texttt{abs}(u)).$$

⟨ multiplicative operator/ of an Integer with a Natural 232b ⟩ ≡

```
inline Integer::Integer(const binder_arguments<Integer, Natural,
                                               Integer_divides_tag>& a)
 : Natural(' ', ' ')
{
  get_memory(a.x.length()+DELTA);
  div(a.x, a.y);
}

inline Integer& Integer::operator=(const binder_arguments<Integer, Natural,
                                                          Integer_divides_tag>& a)
{
  div(a.x, a.y);
  return *this;
```

```
}

inline Integer::Integer(const binder_arguments<Natural, Integer,
                                               Integer_divides_tag>& a)
 : Natural(' ', ' ')
{
  get_memory(a.x.length()+DELTA);
  div(a.x, a.y);
}

inline Integer& Integer::operator=(const binder_arguments<Natural, Integer,
                                                          Integer_divides_tag>& a)
{
  div(a.x, a.y);
  return *this;
}

inline binder_arguments<Integer, Natural, Integer_divides_tag>
 operator/(const Integer& a, const Natural& b)
// Algorithm:  c := a/b
// Input:      a in Integer, b in Natural where not b = 0.
// Output:     c in Integer such that c = sign(a)*[|a|/b] ||
{
  return binder_arguments<Integer, Natural, Integer_divides_tag>(a, b);
}

inline binder_arguments<Natural, Integer, Integer_divides_tag>
 operator/(const Natural& a, const Integer& b)
// Algorithm:  c := a/b
// Input:      a in Natural, b in Integer where not b = 0.
// Output:     c in Integer such that c = sign(b)*[a/|b|] ||
{
  return binder_arguments<Natural, Integer, Integer_divides_tag>(a, b);
}
◇
```

Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer und $x \in$ Natural, dann $u/x \sim v$.div(u,x).
Laufzeit. Seien $u, v \in$ Integer und $x \in$ Natural, dann $x/u \sim v$.div(x,u).

⟨ division of an Integer by a SignDigit 233 ⟩ ≡

```
void div(const Integer& a, const SignDigit b, Integer& c, SignDigit& d)
// Algorithm:  div(a, b, c, d)
// Input:      a in Integer, b in SignDigit where not b = 0.
// Output:     c in Integer, d in SignDigit
//             such that c = sign(a*b)*[|a/b|], d = a - c*b ||
{
  INTEGERCONDITION(a);

  Digit r;
  if (b >= 0) {
    div(abs(a), Digit(b), c.abs(), r);
    c.sgn = (abs(c) != 0)? a.sgn : 0;
  } else {
    div(abs(a), Digit(-b), c.abs(), r);
```

```
    c.sgn = (abs(c) != 0)? -a.sgn : 0;
  }
  d = (sign(a) == -1)? -r : r;

  INTEGERCONDITION(c);
}
◇
```

Macro referenced in 426.

Laufzeit. Seien $u, v, w \in$ Integer und $a, b \in$ SignDigit, dann

$$\texttt{div(}u\texttt{,}a\texttt{,}v\texttt{,}w\texttt{)} \sim \texttt{div(abs(}u\texttt{),}a\texttt{,abs(}v\texttt{),}\ b\texttt{)}.$$

⟨ multiplicative operator/ of an Integer with a SignDigit 234a ⟩ ≡

```
inline Integer operator/(const Integer& a, const SignDigit b)
// Algorithm:  c := a/b
// Input:      a in Integer, b in SignDigit where not b = 0.
// Output:     c in Integer such that c = sign(a*b)*[|a/b|] ||
{
  return Integer(a) /= b;
}
◇
```

Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer und $a \in$ SignDigit, dann $u\texttt{/}a \sim u\texttt{/=}a$.

⟨ multiplicative operator% of an Integer with a SignDigit 234b ⟩ ≡

```
inline SignDigit operator%(const Integer& a, const SignDigit b)
// Algorithm:  c := a%b
// Input:      a in Integer, b in SignDigit where not b = 0.
// Output:     c in SignDigit such that c = a - sign(a*b)*[|a/b|]*b ||
{
  SignDigit c = abs(a) % Digit((b >= 0)? b : -b);
  if (sign(a) == -1) c = -c;
  return c;
}
◇
```

Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer und $a \in$ SignDigit, dann $u\texttt{\%}a \sim \texttt{abs(}u\texttt{)\%}a$.

Zuweisungsoperatoren

⟨ assign operator/= for Integers 234c ⟩ ≡

```
inline Integer& Integer::operator/=(const Integer& a)
// Algorithm:  c := c /= a
// Input:      a,c in Integer where not a = 0.
// Output:     c in Integer such that c := sign(a*c)*[|c/a|] ||
{
  abs() /= ::abs(a);
  sgn = (::abs(*this) != 0)? sgn*a.sgn : 0;
  return *this;
}
◇
```

Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer, dann u/=$v \sim$ abs(u)/=abs(v).

⟨ assign operator%= for Integers 235a ⟩ ≡
```
inline Integer& Integer::operator%=(const Integer& a)
// Algorithm:  c := c %= a
// Input:      a,c in Integer where not a = 0.
// Output:     c in Integer such that c := c - sign(a*c)*[|c/a|]*a ||
{
  abs() %= ::abs(a);
  if (::abs(*this) == 0) sgn = 0;
  return *this;
}
```
◇
Macro referenced in 419.

Laufzeit. Seien $u, v \in$ Integer, dann u%=$v \sim$ abs(u)%=abs(v).

⟨ assign operator/= of an Integer with a Natural 235b ⟩ ≡
```
inline Integer& Integer::operator/=(const Natural& a)
// Algorithm:  c := c /= a
// Input:      c in Integer, a in Natural where not a = 0.
// Output:     c in Integer such that c := sign(c)*[|c|/a] ||
{
  abs() /= a;
  if (::abs(*this) == 0) sgn = 0;
  return *this;
}
```
◇
Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer und $x \in$ Natural, dann u/=$x \sim$ abs(u)/=x.

⟨ assign operator%= of an Integer with a Natural 235c ⟩ ≡
```
inline Integer& Integer::operator%=(const Natural& a)
// Algorithm:  c := c %= a
// Input:      c in Integer, a in Natural where not a = 0.
// Output:     c in Integer such that c := c - sign(c)*[|c|/a]*a ||
{
  abs() %= a;
  if (::abs(*this) == 0) sgn = 0;
  return *this;
}
```
◇
Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer und $x \in$ Natural, dann u%=$x \sim$ abs(u)%=x.

⟨ assign operator/= of an Integer with a SignDigit 236a ⟩ ≡
```
inline Integer& Integer::operator/=(const SignDigit a)
// Algorithm:  c := c /= a
// Input:      c in Integer, a in SignDigit where not a = 0.
// Output:     c in Integer such that c := sign(a*c)*[|c/a|] ||
{
  if (a >= 0)
    if (::abs(*this) < Digit(a)) *this = 0;
    else abs() /= a;
  else
    if (::abs(*this) < Digit(-a)) *this = 0;
    else { abs() /= -a; sgn = -sgn; }
  return *this;
}
◇
```
Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer und $a \in$ SignDigit, dann u/=$a \sim$ abs(u)/=a.

⟨ assign operator%= of an Integer with a SignDigit 236b ⟩ ≡
```
inline Integer& Integer::operator%=(const SignDigit a)
// Algorithm:  c := c %= a
// Input:      c in Integer, a in SignDigit where not a = 0.
// Output:     c in Integer such that c := c - sign(a*c)*[|c/a|]*a ||
{
  abs() %= (a >= 0)? a : -a;
  if (::abs(*this) == 0) sgn = 0;
  return *this;
}
◇
```
Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer und $a \in$ SignDigit, dann u%=$a \sim$ abs(u)%=a.

3.2.17 Splitting

⟨ splits an Integer 236c ⟩ ≡
```
inline void Integer::split(const size_t b, Integer& c, Integer& d) const
// Algorithm:  a.split(b, c, d)
// Input:      a,c,d in Integer, b in size_t where not c = d.
// Output:     c,d in Integer such that c = sign(a)*[|a|/2^(BETA*b)],
//             d = a - c*2^(BETA b) ||
{
  Natural::split(b, c.abs(), d.abs());
  c.sgn = (::abs(c) != 0)? sgn : 0;
  d.sgn = (::abs(d) != 0)? sgn : 0;
}
◇
```
Macro referenced in 419.

Laufzeit. Seien $u, v, w \in$ Natural und $t \in$ size_t, dann

$$u\texttt{.split(}t\texttt{,}v\texttt{,}w\texttt{)} \sim \texttt{abs(}u\texttt{).split(}t\texttt{),abs(}v\texttt{),abs(}w\texttt{).}$$

3.2.18 Wurzelberechnung

Quadratwurzel

⟨ square root of an Integer 237a ⟩ ≡

```
void Integer::sqrt(const Integer& a)
// Algorithm:  b.sqrt(a)
// Input:      a,b in Integer where a >= 0.
// Output:     b in Integer such that b = [sqrt(a)] ||
{
  INTEGERCONDITION(a);

  const int sA = a.sgn;
  if (sA == -1) errmsg(6, "(sqrt)");
  Natural::sqrt(::abs(a));
  sgn = sA;

  INTEGERCONDITION(*this);
}
```
◇

Macro referenced in 426.

Laufzeit. Sei $u \in$ Integer, dann `sqrt(`u`)` $\sim$ `sqrt(abs(`u`))`.

⟨ positive square root and remainder of an Integer 237b ⟩ ≡

```
void sqrt(const Integer& a, Integer& b, Integer& c)
// Algorithm:  sqrt(a, b, c)
// Input:      a in Integer where a >= 0.
// Output:     b,c in Integer such that b = [sqrt(a)], c = a - b^2 ||
{
  INTEGERCONDITION(a);
  INTEGER_FOR_CHECK(_a, a);

  const int sA = a.sgn;
  if (sA == -1) a.errmsg(6, "(sqrt)");
  else if (sA == 0) b = c = 0;
  else {
    sqrt(abs(a), b.abs(), c.abs());
    b.sgn = 1;
    c.sgn = (abs(c) != 0);
  }

  CONDITION(c+b*b == _a);
  INTEGERCONDITION(b);
  INTEGERCONDITION(c);
}
```
◇

Macro referenced in 426.

Laufzeit. Seien $u, v, w \in$ Integer, dann

`sqrt(`u`,`v`,`w`)` $\sim$ `sqrt(abs(`u`),abs(`v`),abs(`w`))`.

⟨ simple call for square root of an Integer 237c ⟩ ≡

```
inline Integer::Integer(const binder_arguments<Integer, Integer,
                                                Integer_square_root_tag>& a)
 : Natural(' ', ' ')
{
  get_memory(a.x.length()/2+DELTA);
  sqrt(a.x);
}

inline Integer& Integer::operator=(const binder_arguments<Integer, Integer,
                                                          Integer_square_root_tag>& a)
{
  sqrt(a.x);
  return *this;
}

inline binder_arguments<Integer, Integer, Integer_square_root_tag>
 sqrt(const Integer& a)
// Algorithm:  b := sqrt(a)
// Input:      a in Integer where a >= 0.
// Output:     b in Integer such that b = [sqrt(a)] ||
{
  return binder_arguments<Integer, Integer, Integer_square_root_tag>(a, a);
}
◇
```

Macro referenced in 419.

Laufzeit. Sei $u \in$ Integer und $x \in$ Natural, dann `sqrt(`u`)` $\sim$ `sqrt(abs(`u`),`x`)`.

Beliebige Wurzel

⟨ calculates the root of an Integer 238a ⟩ ≡

```
Integer root(const Integer& a, const SignDigit b)
// Algorithm:  c := root(a, b)
// Input:      a in Integer, b in SignDigit where 2|b or a >= 0.
// Output:     c in Integer such that c = sign(a)^b*[|a|^(1/b)] ||
{
  INTEGERCONDITION(a);

  if (b < 0 && abs(a) != 1) return SignDigit(0);
  const int sA = sign(a);
  if ((b&1) && sA < 0) return -Integer(root(abs(a), Digit(b)));
  if (sA < 0) a.errmsg(6, "(root)");
  return root(abs(a), Digit(b));
}
◇
```

Macro referenced in 426.

Laufzeit. Sei $u \in$ Integer und $a \in$ SignDigit, dann `root(`u`,`a`)` $\sim$ `root(abs(`u`),`a`)`.

3.2.19 Zufallszahl

⟨ calculates an Integer random number 238b ⟩ ≡

```
void Integer::rand(const size_t n)
// Algorithm:  a.rand(n)
// Input:      n in size_t.
// Output:     a in Integer such that |a| < 2^n (random number) ||
{
  INTEGERCONDITION(*this);

  Natural::rand(n);
  if (::abs(*this) == 0) sgn = 0;
  else sgn = (::rand()&1)? 1 : -1;

  INTEGERCONDITION(*this);
}
◇
```

Macro referenced in 426.

Laufzeit. Sei $u \in$ Integer und $a \in$ Digit, dann u`.rand(`a`)` $\sim$ `abs(`u`).rand(`a`)`.

Kapitel 4

Die rationale Arithmetik

⟨ class Rational 241a ⟩ ≡
```
// Unfortunatelly not:
//typedef Rational<Integer, Natural> Rational;
```
◇
Macro referenced in 427.

4.1 Konstruktoren und Destruktor

⟨ default constructor Rational 241b ⟩ ≡
```
inline Rational::Rational(const SignDigit a)
 : num(a), den(1)
// Algorithm:  c := Rational(a)
// Input:      a in SignDigit.
// Output:     c in Rational such that c = a/1 ||
{
}
```
◇
Macro referenced in 427.

⟨ overloaded constructor Rational for Integer 241c ⟩ ≡
```
inline Rational::Rational(const Integer& a)
 : num(a), den(1)
// Algorithm:  c := Rational(a)
// Input:      a in Integer.
// Output:     c in Rational such that c = a/1 ||
{
}
```
◇
Macro referenced in 427.

⟨ overloaded constructor Rational for Integer and Natural 241d ⟩ ≡
```
Rational::Rational(const Integer& a, const Natural& b)
 : num(a), den(b)
// Algorithm:  c := Rational(a, b)
// Input:      a in Integer, b in Natural where not b = 0.
// Output:     c in Rational such that c = a/b ||
{
```

```
  if (b == 0) den.errmsg(4, "(constructor)");
  const Natural t = gcd(abs(a), b);
  num /= t; den /= t;
}
◇
```

Macro referenced in 431.

⟨ copy constructor Rational 242a ⟩ ≡

```
inline Rational::Rational(const Rational& a)
 : num(a.num), den(a.den)
// Algorithm:  c := Rational(a)
// Input:      a in Rational.
// Output:     c in Rational such that c = a ||
{
}
◇
```

Macro referenced in 427.

⟨ destructor Rational 242b ⟩ ≡

```
inline Rational::~Rational()
{
}
◇
```

Macro referenced in 427.

4.2 Selektoren

⟨ quotient field numerator (memberfunction) 242c ⟩ ≡

```
inline const Integer& Rational::numerator() const
// Algorithm:  c := a.numerator()
// Input:      a in Rational.
// Output:     c in Integer such that c = a_1 where a_1/a_2 = a ||
{
  return num;
}
◇
```

Macro referenced in 427.

⟨ quotient field numerator 242d ⟩ ≡

```
inline const Integer& numerator(const Rational& a)
// Algorithm:  c := numerator(a)
// Input:      a in Rational.
// Output:     c in Integer such that c = a_1 where a_1/a_2 = a ||
{
  return a.numerator();
}
◇
```

Macro referenced in 427.

⟨ quotient field denominator (memberfunction) 243a ⟩ ≡

```
inline const Natural& Rational::denominator() const
// Algorithm:  c := a.denominator()
// Input:      a in Rational.
// Output:     c in Natural such that c = a_2 where a_1/a_2 = a ||
{
  return den;
}
```

◇

Macro referenced in 427.

⟨ quotient field denominator 243b ⟩ ≡

```
inline const Natural& denominator(const Rational& a)
// Algorithm:  c := denominator(a)
// Input:      a in Rational.
// Output:     c in Natural such that c = a_2 where a_1/a_2 = a ||
{
  return a.denominator();
}
```

◇

Macro referenced in 427.

4.3 Absolutbetrag

⟨ absolute value of a Rational 243c ⟩ ≡

```
inline Rational abs(const Rational& a)
// Algorithm:  c := abs(a)
// Input:      a in Rational.
// Output:     c in Rational such that c = |a| ||
{
  return Rational(abs(a.numerator()), a.denominator());
}
```

◇

Macro referenced in 427.

4.4 Vorzeichen

⟨ sign of a Rational 243d ⟩ ≡

```
inline int sign(const Rational& a)
// Algorithm:  c := sign(a)
// Input:      a in Rational.
// Output:     c in int such that if a = 0 then c = 0
//             else if a > 0 then c = 1 else c = -1 ||
{
  return sign(a.numerator());
}
```

◇

Macro referenced in 427.

4.5 Vertauschen zweier Rationals

⟨ function swap for Rationals 243e ⟩ ≡

```
inline void Rational::swap(Rational& a)
// Algorithm:  a.swap(b)
// Input:      a,b in Rational.
// Output:     a,b in Rational such that t := a, a := b, b := t
//             where t in Rational ||
{
  ::swap(num, a.num);
  ::swap(den, a.den);
}

inline void swap(Rational& a, Rational& b)
// Algorithm:  swap(a, b)
// Input:      a,b in Rational.
// Output:     a,b in Rational such that t := a, a := b, b := t
//             where t in Rational ||
{
  a.swap(b);
}
◇
```

Macro referenced in 427.

4.6 Kopierfunktionen

⟨ assign operator= for Rationals 244a ⟩ ≡

```
inline Rational& Rational::operator=(const Rational& a)
// Algorithm:  c := c = a
// Input:      a,c in Rational.
// Output:     c in Rational such that c = a ||
{
  num = a.num; den = a.den;
  return *this;
}
◇
```

Macro referenced in 427.

⟨ assign operator= for a Rational with a SignDigit 244b ⟩ ≡

```
inline SignDigit Rational::operator=(const SignDigit a)
// Algorithm:  c := b = a
// Input:      a in SignDigit, b in Rational.
// Output:     b in Rational, c in SignDigit such that b = a/1, c = a ||
{
  num = a; den = 1;
  return a;
}
◇
```

Macro referenced in 427.

⟨ assign operator= for a Rational with a Integer 245a ⟩ ≡
```
inline Rational& Rational::operator=(const Integer& a)
// Algorithm:  c := c = a
// Input:      a in Integer, c in Rational.
// Output:     c in Rational such that c = a/1 ||
{
  num = a; den = 1;
  return *this;
}
```
◇
Macro referenced in 427.

4.7 Streamausgabe

⟨ puts a Rational on output stream 245b ⟩ ≡

```
inline OSTREAM& operator<<(OSTREAM& out, const Rational& a)
// Algorithm:  o := o << a
// Input:      o in ostream, a in Rational.
// Output:     o in ostream ||
//
// Note:       puts Rational a on output stream.
{
  return out << a.numerator() << '/' << a.denominator();
}
```
◇

Macro referenced in 427.

4.7.1 Interne Darstellung

⟨ puts internal representation of a Rational on output stream 245c ⟩ ≡
```
inline OSTREAM& operator<<(OSTREAM& out, const Rational::rep& a)
// puts internal representation of Rational a on output stream.
{
  return out << print(a.num, a.bin) << '/' << print(a.den, a.bin);
}
```
◇
Macro referenced in 427.
⟨ output variables for representation of a Rational 245d ⟩ ≡
```
struct rep {
  const Integer& num;
  const Natural& den;
  const bool     bin;
  rep(const Integer& a, const Natural& b, const bool c = false)
    : num(a), den(b), bin(c) {}
};
```
◇
Macro referenced in 427.

⟨ function print of a Rational 246a ⟩ ≡

```
inline Rational::rep print(const Rational& a, bool b)
// Algorithm:  o := o << print(a, b)
// Input:      o in ostream, a in Rational, b in bool.
// Output:     o in ostream ||
//
// Note:       puts internal representation of Rational a on an output stream.
{
  return Rational::rep(a.numerator(), a.denominator(), b);
}
◇
```

Macro referenced in 427.

4.8 Streameingabe

⟨ gets a Rational from input stream 246b ⟩ ≡

```
ISTREAM& operator>>(ISTREAM& in, Rational& a)
// Algorithm:  i := i >> a
// Input:      i in istream.
// Output:     i in istream, a in Rational ||
//
// Note:       gets Rational a from input stream.
{
  in >> a.num;
  char ch = 0;
  if (in.get(ch) && ch != '/') { in.putback(ch); return in; }
  in >> a.den;
  if (a.den == 0) a.den = 1;
  Natural b = gcd(abs(a.num), a.den);
  a.num /= b; a.den /= b;
  return in;
}
◇
```

Macro referenced in 431.

4.8.1 Interne Darstellung

⟨ gets internal representation of a Rational from input stream 246c ⟩ ≡

```
bool Rational::scan(ISTREAM& in)
// Algorithm:  b := a.scan(i)
// Input:      a in Rational, i in istream.
// Output:     a in Rational, i in istream, b in bool ||
//
// Note:       gets Rational a as an internal representation from input stream
//             if b is true.
{
  if (!num.scan(in)) return false;
  char c = 0;
  if (in.get(c) && c != '/') { in.putback(c); return false; }
  return den.scan(in);
}
◇
```

Macro referenced in 431.

4.8.2 Konvertierung in ein beliebiges Stellenwertsystem

⟨ converts a Rational to a string 247a ⟩ ≡

```
char* Rtoa(const Rational& a, char* b, const Digit c)
// Algorithm:  c := Rtoa(a, c, b)
// Input:      a in Rational, b in Digit, c in String
//             where 2 <= b <= 36, sizeof(c) > BETA*L(a)/log2(b).
// Output:     c in String such that c = a ||
//
// Note:       conversion Rational to string.
{
  Itoa(a.numerator(), b, c);
  char* d = b+strlen(b);
  *d = '/';
  Ntoa(a.denominator(), ++d, c);
  return b;
}
```

◇

Macro referenced in 431.

4.8.3 Konvertierung aus einem beliebigen Stellenwertsystem

⟨ converts a string to a Rational 247b ⟩ ≡

```
const char* Rational::atoR(const char* a, const Digit b)
// Algorithm:  c := d.atoR(a, b)
// Input:      d in Rational, a in String, b in Digit where 2 <= b <= 36.
// Output:     d in Rational, c in String such that d = a ||
//
// Note: Returns a pointer to the first occurrence of a non-digit character
//       in a after the character '/'.
{
  a = num.atoI(a, b);
  if (*a == '/') a = den.atoN(++a, b);
  if (den == 0) den = 1;
  const Natural c = gcd(abs(num), den);
  num /= c; den /= c;
  return a;
}
```

◇

Macro referenced in 431.

⟨ converts a string to an Rational by function call 247c ⟩ ≡

```
inline Rational atoR(const char* a, const Digit b)
// Algorithm:  c := atoR(a, b)
// Input:      a in String, b in Digit where 2 <= b <= 36.
// Output:     c in Rational such that c = a ||
//
// Note:       conversion string to Rational; return 0 by conversion error.
{
  Rational result;
  result.atoR(a, b);
  return result;
}
```

◇

Macro referenced in 427.

4.9 Vergleichsoperatoren

⟨ comparison operator== for Rationals 248a ⟩ ≡

```
inline bool operator==(const Rational& a, const Rational& b)
// Algorithm:  c := a == b
// Input:      a,b in Rational.
// Output:     c in bool such that if a = b then c = true else c = false ||
{
  return (a.numerator() == b.numerator() && a.denominator() == b.denominator());
}
```

◇

Macro referenced in 427.

⟨ comparison operator!= for Rationals 248b ⟩ ≡

```
inline bool operator!=(const Rational& a, const Rational& b)
// Algorithm:  c := a != b
// Input:      a,b in Rational.
// Output:     c in bool such that if a = b then c = false else c = true ||
{
  return (a.numerator() != b.numerator() || a.denominator() != b.denominator());
}
```

◇

Macro referenced in 427.

⟨ comparison operator< for Rationals 248c ⟩ ≡

```
inline bool operator<(const Rational& a, const Rational& b)
// Algorithm:  c := a < b
// Input:      a,b in Rational.
// Output:     c in bool such that if a < b then c = true else c = false ||
{
  return (a.numerator()*b.denominator() < a.denominator()*b.numerator());
}
```

◇

Macro referenced in 427.

⟨ comparison operator<= for Rationals 248d ⟩ ≡

```
inline bool operator<=(const Rational& a, const Rational& b)
// Algorithm:  c := a <= b
// Input:      a,b in Rational.
// Output:     c in bool such that if a <= b then c = true else c = false ||
{
  return (a.numerator()*b.denominator() <= a.denominator()*b.numerator());
}
```

◇

Macro referenced in 427.

⟨ comparison operator> for Rationals 248e ⟩ ≡

```
inline bool operator>(const Rational& a, const Rational& b)
// Algorithm:  c := a > b
// Input:      a,b in Rational.
// Output:     c in bool such that if a > b then c = true else c = false ||
{
  return (a.numerator()*b.denominator() > a.denominator()*b.numerator());
}
```

◇

Macro referenced in 427.

⟨ comparison operator>= for Rationals 249a ⟩ ≡
```
inline bool operator>=(const Rational& a, const Rational& b)
// Algorithm:  c := a >= b
// Input:      a,b in Rational.
// Output:     c in bool such that if a >= b then c = true else c = false ||
{
  return (a.numerator()*b.denominator() >= a.denominator()*b.numerator());
}
```
◇
Macro referenced in 427.

4.9.1 SignDigit-Vergleiche

⟨ comparison operator== of a Rational with a SignDigit 249b ⟩ ≡
```
inline bool operator==(const Rational& a, const SignDigit b)
// Algorithm:  c := a == b
// Input:      a in Rational, b in SignDigit.
// Output:     c in bool such that if a = b then c = true else c = false ||
{
  return (a.numerator() == b && a.denominator() == 1);
}
```
◇
Macro referenced in 427.

⟨ comparison operator!= of a Rational with a SignDigit 249c ⟩ ≡
```
inline bool operator!=(const Rational& a, const SignDigit b)
// Algorithm:  c := a != b
// Input:      a in Rational, b in SignDigit.
// Output:     c in bool such that if a = b then c = false else c = true ||
{
  return (a.numerator() != b || a.denominator() != 1);
}
```
◇
Macro referenced in 427.

⟨ comparison operator< of a Rational with a SignDigit 249d ⟩ ≡
```
inline bool operator<(const Rational& a, const SignDigit b)
// Algorithm:  c := a < b
// Input:      a in Rational, b in SignDigit.
// Output:     c in bool such that if a < b then c = true else c = false ||
{
  return (b < 0)? (a.numerator() > a.denominator()*Digit(-b))
                : (a.numerator() < a.denominator()*Digit(b));
}
```
◇
Macro referenced in 427.

⟨ comparison operator<= of a Rational with a SignDigit 249e ⟩ ≡
```
inline bool operator<=(const Rational& a, const SignDigit b)
// Algorithm:  c := a <= b
// Input:      a in Rational, b in SignDigit.
// Output:     c in bool such that if a <= b then c = true else c = false ||
{
  return (b < 0)? (a.numerator() >= a.denominator()*Digit(-b))
                : (a.numerator() <= a.denominator()*Digit(b));
}
```
◇
Macro referenced in 427.

⟨ comparison operator> of a Rational with a SignDigit 250a ⟩ ≡

```
inline bool operator>(const Rational& a, const SignDigit b)
// Algorithm:  c := a > b
// Input:      a in Rational, b in SignDigit.
// Output:     c in bool such that if a > b then c = true else c = false ||
{
  return (b < 0)? (a.numerator() < a.denominator()*Digit(-b))
                : (a.numerator() > a.denominator()*Digit(b));
}
◇
```

Macro referenced in 427.

⟨ comparison operator>= of a Rational with a SignDigit 250b ⟩ ≡

```
inline bool operator>=(const Rational& a, const SignDigit b)
// Algorithm:  c := a >= b
// Input:      a in Rational, b in SignDigit.
// Output:     c in bool such that if a >= b then c = true else c = false ||
{
  return (b < 0)? (a.numerator() <= a.denominator()*Digit(-b))
                : (a.numerator() >= a.denominator()*Digit(b));
}
◇
```

Macro referenced in 427.

4.10 Additive Operationen

4.10.1 Inkrementierung

⟨ prefix incrementation of a Rational 250c ⟩ ≡

```
inline const Rational& Rational::operator++()
// Algorithm:  c := ++a
// Input:      a in Rational.
// Output:     a,c in Rational such that a := a+1, c := a ||
{
  num += den;
  return *this;
}
◇
```

Macro referenced in 427.

⟨ postfix incrementation of a Rational 250d ⟩ ≡

```
inline Rational Rational::operator++(int)
// Algorithm:  c := a++
// Input:      a in Rational.
// Output:     a,c in Rational such that c := a, a := a+1 ||
{
  Rational a(*this);
  ++(*this);
  return a;
}
◇
```

Macro referenced in 427.

4.10.2 Decrementierung

⟨ prefix decrementation of a Rational 250e ⟩ ≡

```
inline const Rational& Rational::operator--()
// Algorithm:  c := --a
// Input:      a in Rational.
// Output:     a,c in Rational such that a := a-1, c := a ||
{
  num -= den;
  return *this;
}
◇
```

Macro referenced in 427.

⟨ postfix decrementation of a Rational 251a ⟩ ≡

```
inline Rational Rational::operator--(int)
// Algorithm:  c := a--
// Input:      a in Rational.
// Output:     a,c in Rational such that c := a, a := a-1 ||
{
  Rational a(*this);
  --(*this);
  return a;
}
◇
```

Macro referenced in 427.

4.10.3 Negation

⟨ negation operator- of a Rational 251b ⟩ ≡

```
inline Rational::Rational(const binder_arguments<Rational, Rational,
                                                 Rational_negate_tag>& a)
 : num(-a.x.num), den(a.x.den)
{
}

inline Rational& Rational::operator=(const binder_arguments<Rational, Rational,
                                                            Rational_negate_tag>& a)
{
  num = -a.x.num;
  if (this != &a.x) den = a.x.den;
  return *this;
}

inline binder_arguments<Rational, Rational, Rational_negate_tag>
 operator-(const Rational& a)
// Algorithm:  c := -a
// Input:      a in Rational.
// Output:     c in Rational such that c = -a ||
{
  return binder_arguments<Rational, Rational, Rational_negate_tag>(a, a);
}
◇
```

Macro referenced in 427.

4.10.4 Addition

⟨ addition of two Rationals 251c ⟩ ≡

```
void Rational::add(const Rational& a, const Rational& b)
// Algorithm:  c.add(a, b)
// Input:      a,b in Rational.
// Output:     c in Rational such that c = a + b ||
{
  if (sign(a.num) == 0) { num = b.num; den = b.den; }
  else if (sign(b.num) == 0) { num = a.num; den = a.den; }
  else {
    Natural g = gcd(a.den, b.den);
    if (g == 1) {
      ⟨ special case of Rational addition 252a ⟩
    } else {
      ⟨ general case of Rational addition 252b ⟩
    }
  }
}
◇
```

Macro referenced in 431.

$$\frac{\texttt{num}}{\texttt{den}} = \frac{\texttt{a.num} \cdot \texttt{b.den} + \texttt{a.den} \cdot \texttt{b.num}}{\texttt{a.den} \cdot \texttt{b.den}} :$$

⟨ special case of Rational addition 252a ⟩ ≡

```
const Integer h = a.den * b.num;
num = a.num * b.den;
num += h;
den = a.den * b.den;◇
```

Macro referenced in 251c.

$$\begin{aligned} g &= \gcd(\texttt{a.den}, \texttt{b.den}), \\ t &= \texttt{a.num} \cdot \frac{\texttt{b.den}}{g} + \texttt{b.num} \cdot \frac{\texttt{a.den}}{g}, \\ h &= \gcd(t, g), \\ \texttt{num} &= \frac{t}{h}, \\ \texttt{den} &= \frac{\texttt{a.den}}{g} \cdot \frac{\texttt{b.den}}{h} : \end{aligned}$$

⟨ general case of Rational addition 252b ⟩ ≡

```
Natural s = b.den / g;
Integer t = a.num * s;
s = a.den / g;
t += b.num * s;
g = gcd(abs(t), g);
num = t / g;
g = b.den / g;
den = s * g;◇
```

Macro referenced in 251c.

⟨ additive operator+ for Rationals 252c ⟩ ≡

```
inline Rational::Rational(const binder_arguments<Rational, Rational,
                                                 Rational_plus_tag>& a)
{
```

```
  add(a.x, a.y);
}

inline Rational& Rational::operator=(const binder_arguments<Rational, Rational,
                                                            Rational_plus_tag>& a)
{
  add(a.x, a.y);
  return *this;
}

inline binder_arguments<Rational, Rational, Rational_plus_tag>
 operator+(const Rational& a, const Rational& b)
// Algorithm:  c := a+b
// Input:      a,b in Rational.
// Output:     c in Rational such that c = a+b ||
{
  return binder_arguments<Rational, Rational, Rational_plus_tag>(a, b);
}
◇
```

Macro referenced in 427.

Zuweisungsoperator

⟨ assign `operator+=` for Rationals 253a ⟩ ≡

```
inline Rational& Rational::operator+=(const Rational& a)
// Algorithm:  c := c += a
// Input:      a,c in Rational.
// Output:     c in Rational such that c := c+a ||
{
  add(*this, a);
  return *this;
}
◇
```

Macro referenced in 427.

4.10.5 Subtraktion

⟨ subtraction of two Rationals 253b ⟩ ≡

```
void Rational::sub(const Rational& a, const Rational& b)
// Algorithm:  c.sub(a, b)
// Input:      a,b in Rational.
// Output:     c in Rational such that c = a - b ||
{
  if (sign(a.num) == 0) { num = -b.num; den = b.den; }
  else if (sign(b.num) == 0) { num = a.num; den = a.den; }
  else {
    Natural g = gcd(a.den, b.den);
    if (g == 1) {
      ⟨ special case of Rational subtraction 254a ⟩
    } else {
      ⟨ general case of Rational subtraction 254b ⟩
    }
  }
}
◇
```

Macro referenced in 431.

$$\frac{\texttt{num}}{\texttt{den}} = \frac{\texttt{a.num} \cdot \texttt{b.den} - \texttt{a.den} \cdot \texttt{b.num}}{\texttt{a.den} \cdot \texttt{b.den}} :$$

⟨ special case of Rational subtraction 254a ⟩ ≡

```
const Integer h = a.den * b.num;
num = a.num * b.den;
num -= h;
den = a.den * b.den;◇
```

Macro referenced in 253b.

$$\begin{aligned} g &= \gcd(\texttt{a.den}, \texttt{b.den}), \\ t &= \texttt{a.num} \cdot \frac{\texttt{b.den}}{g} - \texttt{b.num} \cdot \frac{\texttt{a.den}}{g}, \\ h &= \gcd(t, g), \\ \texttt{num} &= \frac{t}{h}, \\ \texttt{den} &= \frac{\texttt{a.den}}{g} \cdot \frac{\texttt{b.den}}{h} : \end{aligned}$$

⟨ general case of Rational subtraction 254b ⟩ ≡

```
Natural s = b.den / g;
Integer t = a.num * s;
s = a.den / g;
t -= b.num * s;
g = gcd(abs(t), g);
num = t / g;
g = b.den / g;
den = s * g;◇
```

Macro referenced in 253b.

⟨ additive operator- for Rationals 254c ⟩ ≡

```
inline Rational::Rational(const binder_arguments<Rational, Rational,
                                                 Rational_minus_tag>& a)
{
  sub(a.x, a.y);
}

inline Rational& Rational::operator=(const binder_arguments<Rational, Rational,
                                                            Rational_minus_tag>& a)
{
  sub(a.x, a.y);
  return *this;
}

inline binder_arguments<Rational, Rational, Rational_minus_tag>
 operator-(const Rational& a, const Rational& b)
// Algorithm:  c := a-b
// Input:      a,b in Rational.
// Output:     c in Rational such that c = a-b ||
{
  return binder_arguments<Rational, Rational, Rational_minus_tag>(a, b);
}
◇
```

Macro referenced in 427.

Zuweisungsoperator

⟨ assign `operator-=` for Rationals 255a ⟩ ≡

```
inline Rational& Rational::operator-=(const Rational& a)
// Algorithm:  c := c -= a
// Input:      a,c in Rational.
// Output:     c in Rational such that c := c-a ||
{
  sub(*this, a);
  return *this;
}
```

◇

Macro referenced in 427.

4.11 Schiebeoperationen

4.11.1 Linksverschiebung

⟨ multiplication of a Rational by a power of 2 255b ⟩ ≡

```
void Rational::lshift(const Rational& a, size_t b)
// Algorithm:  c.lshift(a, b)
// Input:      a in Rational, b in size_t.
// Output:     c in Rational such that c = a * 2^b ||
{
  size_t i = 0;
  const Natural& c = a.den;
  while (i < b && !c.testbit(i)) ++i;
  den = a.den >> i;
  num = a.num << (b-i);
}
```

◇

Macro referenced in 431.

⟨ shift `operator<<` of a Rational 255c ⟩ ≡

```
inline Rational::Rational(const binder_arguments<Rational, size_t,
                                                 Rational_lshift_tag>& a)
{
  lshift(a.x, a.y);
}

inline Rational& Rational::operator=(const binder_arguments<Rational, size_t,
                                                            Rational_lshift_tag>& a)
{
  lshift(a.x, a.y);
  return *this;
}

inline binder_arguments<Rational, size_t, Rational_lshift_tag>
 operator<<(const Rational& a, const size_t& b)
// Algorithm:  c := a << b
```

```
// Input:      a in Rational, b in size_t.
// Output:     c in Rational such that c = a * 2^b ||
{
  return binder_arguments<Rational, size_t, Rational_lshift_tag>(a, b);
}
◇
```

Macro referenced in 427.

Zuweisungsoperator

⟨ assign operator<<= of a Rational 256a ⟩ ≡

```
inline Rational& Rational::operator<<=(const size_t a)
// Algorithm:  c := c <<= a
// Input:      a in size_t, c in Rational.
// Output:     c in Rational such that c := c * 2^a ||
{
  lshift(*this, a);
  return *this;
}
◇
```

Macro referenced in 427.

4.11.2 Rechtsverschiebung

⟨ division of a Rational by a power of 2 256b ⟩ ≡

```
void Rational::rshift(const Rational& a, size_t b)
// Algorithm:  c.rshift(a, b)
// Input:      a in Rational, b in size_t.
// Output:     c in Rational such that c = a / 2^b ||
{
  size_t i = 0;
  const Natural& c = abs(a.num);
  while (i < b && !c.testbit(i)) ++i;
  num = a.num >> i;
  den = a.den << (b-i);
}
◇
```

Macro referenced in 431.

⟨ shift operator>> of a Rational 256c ⟩ ≡

```
inline Rational::Rational(const binder_arguments<Rational, size_t,
                                                 Rational_rshift_tag>& a)
{
  rshift(a.x, a.y);
}

inline Rational& Rational::operator=(const binder_arguments<Rational, size_t,
                                                            Rational_rshift_tag>& a)
{
  rshift(a.x, a.y);
  return *this;
}
```

```
inline binder_arguments<Rational, size_t, Rational_rshift_tag>
 operator>>(const Rational& a, const size_t& b)
// Algorithm:  c := a >> b
// Input:      a in Rational, b in size_t.
// Output:     c in Rational such that c = a / 2^b ||
{
  return binder_arguments<Rational, size_t, Rational_rshift_tag>(a, b);
}
◇
```

Macro referenced in 427.

Zuweisungsoperator

⟨ assign operator>>= of a Rational 257a ⟩ ≡

```
inline Rational& Rational::operator>>=(const size_t a)
// Algorithm:  c := c >>= a
// Input:      a in size_t, c in Rational.
// Output:     c in Rational such that c := c / 2^a ||
{
  rshift(*this, a);
  return *this;
}
◇
```

Macro referenced in 427.

4.12 Multiplikative Operationen

4.12.1 Quadratur

⟨ squaring of a Rational 257b ⟩ ≡

```
inline void Rational::sqr(const Rational& a)
// Algorithm:  b.sqr(a)
// Input:      a in Rational.
// Output:     b in Rational such that b = a^2 ||
{
  num = a.num*a.num;
  den = a.den*a.den;
}
◇
```

Macro referenced in 427.

4.12.2 Multiplikation

⟨ multiplication of two Rationals 257c ⟩ ≡

```
void Rational::mul(const Rational& a, const Rational& b)
// Algorithm:  c.mul(a, b)
// Input:      a,b in Rational.
// Output:     c in Rational such that c = a * b ||
{
  Natural g = gcd(abs(a.num), b.den);
  Natural h = gcd(a.den, abs(b.num));
  if (g == 1) {
    if (h == 1) {
```

```
    ⟨ special case 1 of Rational multiplication 258a ⟩
  } else {
    ⟨ special case 2 of Rational multiplication 258b ⟩
  }
} else if (h == 1) {
  ⟨ special case 3 of Rational multiplication 258c ⟩
} else {
  ⟨ general case of Rational multiplication 259a ⟩
}
}
◇
```

Macro referenced in 431.

$$\frac{\texttt{num}}{\texttt{den}} = \frac{\texttt{a.num} \cdot \texttt{b.num}}{\texttt{a.den} \cdot \texttt{b.den}} :$$

⟨ special case 1 of Rational multiplication 258a ⟩ ≡

```
num = a.num * b.num;
den = a.den * b.den;◇
```

Macro referenced in 257c.

$$\begin{aligned} h &= \gcd(\texttt{a.den}, \mathrm{abs}(\texttt{b.num})), \\ \frac{\texttt{num}}{\texttt{den}} &= \frac{\texttt{a.num} \cdot \frac{\texttt{b.num}}{h}}{\frac{\texttt{a.den}}{h} \cdot \texttt{b.den}} : \end{aligned}$$

⟨ special case 2 of Rational multiplication 258b ⟩ ≡

```
Integer t = b.num / h;
g = a.den / h;
num = a.num * t;
den = g * b.den;◇
```

Macro referenced in 257c.

$$\begin{aligned} g &= \gcd(\mathrm{abs}(\texttt{a.num}), \texttt{b.den}), \\ \frac{\texttt{num}}{\texttt{den}} &= \frac{\frac{\texttt{a.num}}{g} \cdot \texttt{b.num}}{\texttt{a.den} \cdot \frac{\texttt{b.den}}{g}} : \end{aligned}$$

⟨ special case 3 of Rational multiplication 258c ⟩ ≡

```
Integer t = a.num / g;
h = b.den / g;
num = t * b.num;
den = a.den * h;◇
```

Macro referenced in 257c.

$$\begin{aligned} g &= \gcd(\mathrm{abs}(\texttt{a.num}), \texttt{b.den}), \\ h &= \gcd(\texttt{a.den}, \mathrm{abs}(\texttt{b.num})), \\ \frac{\texttt{num}}{\texttt{den}} &= \frac{\frac{\texttt{a.num}}{g} \cdot \frac{\texttt{b.num}}{h}}{\frac{\texttt{a.den}}{h} \cdot \frac{\texttt{b.den}}{g}} : \end{aligned}$$

⟨ general case of Rational multiplication 259a ⟩ ≡

```
Integer s = a.num / g;
Integer t = b.num / h;
h = a.den / h;
g = b.den / g;
num = s * t;
den = h * g;◇
```

Macro referenced in 257c.

⟨ multiplicative operator* for Rationals 259b ⟩ ≡

```
inline Rational::Rational(const binder_arguments<Rational, Rational,
                                                 Rational_multiplies_tag>& a)
{
  if (&a.x == &a.y) sqr(a.x);
  else mul(a.x, a.y);
}

inline Rational& Rational::operator=(const binder_arguments<Rational, Rational,
                                                            Rational_multiplies_tag>& a)
{
  if (&a.x == &a.y) sqr(a.x);
  else mul(a.x, a.y);
  return *this;
}

inline binder_arguments<Rational, Rational, Rational_multiplies_tag>
 operator*(const Rational& a, const Rational& b)
// Algorithm:  c := a*b
// Input:      a,b in Rational.
// Output:     c in Rational such that c = a*b ||
{
  return binder_arguments<Rational, Rational, Rational_multiplies_tag>(a, b);
}
◇
```

Macro referenced in 427.

Zuweisungsoperator

⟨ assign operator*= for Rationals 259c ⟩ ≡

```
inline Rational& Rational::operator*=(const Rational& a)
// Algorithm:  c := c *= a
// Input:      a,c in Rational.
// Output:     c in Rational such that c := c*a ||
{
  if (this == &a) sqr(*this);
  else mul(*this, a);
  return *this;
}
◇
```

Macro referenced in 427.

4.12.3 Division

⟨ division of two Rationals 259d ⟩ ≡

```
void Rational::div(const Rational& a, const Rational& b)
```

```
// Algorithm:  c.div(a, b)
// Input:      a,b in Rational where not b = 0.
// Output:     c in Rational such that c = a / b ||
{
  if (b.num == 0) b.num.errmsg(4, "(div)");
  Natural g = gcd(abs(a.num), abs(b.num));
  Natural h = gcd(a.den, b.den);
  if (g == 1) {
    if (h == 1) {
      ⟨ special case 1 of Rational division 260a ⟩
    } else {
      ⟨ special case 2 of Rational division 260b ⟩
    }
  } else if (h == 1) {
    ⟨ special case 3 of Rational division 261a ⟩
  } else {
    ⟨ general case of Rational division 261b ⟩
  }
}
◇
```

Macro referenced in 431.

$$\frac{\texttt{num}}{\texttt{den}} = \frac{\texttt{a.num} \cdot \texttt{b.den} \cdot \operatorname{sign}(\texttt{b.num})}{\texttt{a.den} \cdot \texttt{b.num} \cdot \operatorname{sign}(\texttt{b.num})} :$$

⟨ special case 1 of Rational division 260a ⟩ ≡

```
if (this == &b) {
  const bool i = (sign(num) == -1);
  g = a.den * abs(num);
  num = a.num * den;
  den = g;
  if (i) num = -num;
} else {
  num = a.num * b.den;
  den = a.den * abs(b.num);
  if (sign(b.num) == -1) num = -num;
}◇
```

Macro referenced in 259d.

$$\begin{aligned} h &= \gcd(\texttt{a.den}, \texttt{b.den}), \\ \frac{\texttt{num}}{\texttt{den}} &= \frac{\texttt{a.num} \cdot \frac{\texttt{b.den}}{h} \cdot \operatorname{sign}(\texttt{b.num})}{\frac{\texttt{a.den}}{h} \cdot \texttt{b.num} \cdot \operatorname{sign}(\texttt{b.num})} : \end{aligned}$$

⟨ special case 2 of Rational division 260b ⟩ ≡

```
Natural t = b.den / h;
g = a.den / h;
den = g * abs(b.num);
if (sign(b.num) == -1) {
  num = a.num * t;
  num = -num;
} else num = a.num * t;◇
```

Macro referenced in 259d.

$$\begin{aligned} g &= \gcd(\mathrm{abs}(\texttt{a.num}), \mathrm{abs}(\texttt{b.num})), \\ \frac{\texttt{num}}{\texttt{den}} &= \frac{\frac{\texttt{a.num}}{g} \cdot \texttt{b.den} \cdot \mathrm{sign}(\texttt{b.num})}{\texttt{a.den} \cdot \frac{\texttt{b.num}}{g} \cdot \mathrm{sign}(\texttt{b.num})} : \end{aligned}$$

⟨ special case 3 of Rational division 261a ⟩ ≡

```
Integer t = a.num / g;
Integer s = b.num / g;
num = t * b.den;
den = a.den * abs(s);
if (sign(s) == -1) num = -num;◇
```

Macro referenced in 259d.

$$\begin{aligned} g &= \gcd(\mathrm{abs}(\texttt{a.num}), \mathrm{abs}(\texttt{b.num})), \\ h &= \gcd(\texttt{a.den}, \texttt{b.den}), \\ \frac{\texttt{num}}{\texttt{den}} &= \frac{\frac{\texttt{a.num}}{g} \cdot \frac{\texttt{b.den}}{h} \cdot \mathrm{sign}(\texttt{b.num})}{\frac{\texttt{a.den}}{h} \cdot \frac{\texttt{b.num}}{g} \cdot \mathrm{sign}(\texttt{b.num})} : \end{aligned}$$

⟨ general case of Rational division 261b ⟩ ≡

```
Integer s = a.num / g;
Natural t = b.den / h;
Integer r = b.num / g;
num = s * t;
t = a.den / h;
den = t * abs(r);
if (sign(r) == -1) num = -num;◇
```

Macro referenced in 259d.

⟨ multiplicative operator/ for Rationals 261c ⟩ ≡

```
inline Rational::Rational(const binder_arguments<Rational, Rational,
                                                 Rational_divides_tag>& a)
{
  div(a.x, a.y);
}

inline Rational& Rational::operator=(const binder_arguments<Rational, Rational,
                                                            Rational_divides_tag>& a)
{
  div(a.x, a.y);
  return *this;
}

inline binder_arguments<Rational, Rational, Rational_divides_tag>
 operator/(const Rational& a, const Rational& b)
// Algorithm:  c := a/b
// Input:      a,b in Rational.
// Output:     c in Rational such that c = a/b ||
{
  return binder_arguments<Rational, Rational, Rational_divides_tag>(a, b);
}
◇
```

Macro referenced in 427.

Zuweisungsoperator

⟨ assign operator/= for Rationals 262a ⟩ ≡

```
inline Rational& Rational::operator/=(const Rational& a)
// Algorithm:  c := c /= a
// Input:      a,c in Rational where not a = 0.
// Output:     c in Rational such that c := c/a ||
{
  div(*this, a);
  return *this;
}
◇
```

Macro referenced in 427.

4.12.4 Inversion

⟨ inversion of a Rational 262b ⟩ ≡

```
Rational inv(const Rational& a)
// Algorithm:  b := inv(a)
// Input:      a in Rational where not a = 0.
// Output:     b in Rational such that b = a^(-1) ||
{
  if (a.num == 0) a.num.errmsg(4, "(inv)");
  Rational b(a.den, abs(a.num));
  if (sign(a.num) == -1) b.num = -b.num;
  return b;
}
◇
```

Macro referenced in 431.

4.13 Potenzieren

⟨ power of a Rational by a SignDigit 262c ⟩ ≡

```
Rational pow(const Rational& a, const SignDigit b)
// Algorithm:  c := pow(a, b)
// Input:      a in Rational, b in SignDigit.
// Output:     c in Rational such that c = a^b ||
{
  if (b >= 0) return Rational(pow(a.numerator(), b), pow(a.denominator(), Digit(b)));
  else {
    Rational c(pow(a.denominator(), Digit(-b)), pow(abs(a.numerator()), Digit(-b)));
    if ((b&1) && sign(a) == -1) c = -c;
    return c;
  }
}
◇
```

Macro referenced in 431.

⟨ power of a Rational by an Integer 262d ⟩ ≡

```
Rational pow(const Rational& a, const Integer& b)
// Algorithm:  c := pow(a, b)
// Input:      a in Rational, b in Integer.
// Output:     c in Rational such that c = a^b ||
```

```
{
  if (sign(b) >= 0) return Rational(pow(a.numerator(), b), pow(a.denominator(), abs(b)));
  else {
    const Natural& d = abs(b);
    Rational c(pow(a.denominator(), d), pow(abs(a.numerator()), d));
    if ((b&1) && sign(a) == -1) c = -c;
    return c;
  }
}
◇
```

Macro referenced in 431.

4.14 Rundungsfunktionen

⟨ function `ceil` of a Rational 263a ⟩ ≡

```
Integer ceil(const Rational& a)
// Algorithm:  c := ceil(a)
// Input:      a in Rational.
// Output:     c in Integer such that c = min{x in Integer | x >= a} ||
{
  if (a == 0) return SignDigit(0);
  Natural q,r;
  div(abs(a.numerator()), a.denominator(), q, r);
  if (sign(a) == -1) return -Integer(q);
  r <<= 1;
  if (r >= q) ++q;
  return q;
}
◇
```

Macro referenced in 431.

⟨ function `floor` of a Rational 263b ⟩ ≡

```
Integer floor(const Rational& a)
// Algorithm:  c := floor(a)
// Input:      a in Rational.
// Output:     c in Integer such that c = max{x in Integer | x <= a} ||
{
  Natural q,r;
  div(abs(a.numerator()), a.denominator(), q, r);
  Integer t = q;
  if (sign(a) == -1) {
    t = -t;
    if (r != 0) --t;
  }
  return t;
}
◇
```

Macro referenced in 431.

⟨ function round of a Rational 264a ⟩ ≡

```
inline Integer round(const Rational& a)
// Algorithm:  c := round(a)
// Input:      a in Rational.
// Output:     c in Integer such that c = floor(a+1/2) ||
{
  return floor(a + Rational(1, 2));
}
```

◇

Macro referenced in 427.

⟨ function trunc of a Rational 264b ⟩ ≡

```
Integer trunc(const Rational& a)
// Algorithm:  c := trunc(a)
// Input:      a in Rational.
// Output:     c in Integer such that c = sign(a)*[|a|] ||
{
  Natural q,r;
  div(abs(a.numerator()), a.denominator(), q, r);
  if (sign(a) == -1) return -Integer(q);
  return q;
}
```

◇

Macro referenced in 431.

4.15 Zufallszahl

⟨ calculates a Rational random number 264c ⟩ ≡

```
inline void Rational::rand(const size_t n)
// Algorithm:  a.rand(n)
// Input:      n in size_t.
// Output:     a in Rational
//             such that |a_1| < 2^n, a_2 <= 2^n where a_1/a_2 = a (random number) ||
{
  num.rand(n);
  den.rand(n); ++den;
  const Natural t = gcd(abs(num), den);
  num /= t; den /= t;
}
```

◇

Macro referenced in 427.

Kapitel 5

Kleine Zahlentheorie

5.1 Fibonacci-Zahlen

Die Fibonacci-Zahlen sind durch die folgende Rekursion definiert:

$$F_0 = 0, F_1 = 1 \text{ und } F_n = F_{n-1} + F_{n-2} \quad \text{für } n \geq 2.$$

Eine direkte Umsetzung dieser Definition haben wir bereits im Kapitel 1.6.3 auf der Seite 13 kennengelernt.
In den folgenden vier Schritten werden wir uns dagegen einen wesentlich schnelleren Algorithmus herleiten:

1. **Behauptung.**

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} \text{ für alle } n \geq 1. \tag{5.1}$$

Beweis. Durch vollständige Induktion über n.
Induktionsanfang für $n = 1$: $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix}$ ✓.
Induktionsschritt: Sei (5.1) unsere Induktionsvoraussetzung für n, dann
$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n+1} = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} F_{n+1} + F_n & F_{n+1} \\ F_n + F_{n-1} & F_n \end{pmatrix} = \begin{pmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{pmatrix}$
□

2. **Behauptung.**

$$F_{2n-1} = F_n^2 + F_{n-1}^2, \tag{5.2}$$
$$F_{2n} = F_n^2 + 2F_nF_{n-1} = (F_n + F_{n-1})^2 - F_{n-1}^2. \tag{5.3}$$

Beweis. Nach (5.1) gilt:

$$\begin{pmatrix} F_{2n+1} & F_{2n} \\ F_{2n} & F_{2n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{2n} = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}^2$$
$$= \begin{pmatrix} F_{n+1}^2 + F_n^2 & F_n^2 + F_{n-1}F_n \\ F_n^2 + 2F_nF_{n-1} & F_n^2 + F_{n-1}^2 \end{pmatrix}$$

□

3. **Folgerung.** $F_n = \begin{cases} F_{\frac{n-1}{2}}^2 + F_{\frac{n+1}{2}}^2 \,, & 2 \nmid n \\ F_{\frac{n}{2}}^2 + 2F_{\frac{n}{2}}F_{\frac{n}{2}-1} \,, & 2 \mid n \end{cases}$.

4. Die Implementation der Folgerung würde nun drei Quadraturen benötigen. Die Anzahl der Quadraturen läßt sich jedoch durch die folgende Beziehung auf zwei reduzieren:
Behauptung.

$$F_n \cdot F_{n-1} = F_n^2 - F_{n-1}^2 + (-1)^n \qquad \text{für alle } n \geq 1. \tag{5.4}$$

Beweis. Durch vollständige Induktion über n.
Induktionsanfang für $n \in \{1, 2\}$:

$$\begin{aligned} F_1 \cdot F_0 &= 0 = 1^2 - 0^2 - 1 = F_1^2 - F_0^2 - 1 \quad \checkmark \\ F_2 \cdot F_1 &= 1 = 1^2 - 1^2 + 1 = F_2^2 - F_1^2 + 1 \quad \checkmark. \end{aligned}$$

Induktionsschritt: Sei (5.4) unsere Induktionsvoraussetzung für n, dann

$$\begin{aligned} F_{n+2} \cdot F_{n+1} &= (F_{n+1} + F_n) \cdot (F_n + F_{n-1}) \\ &= (2F_n + F_{n-1}) \cdot (F_n + F_{n-1}) \\ &= 2F_n^2 + 3F_nF_{n-1} + F_{n-1}^2 \\ &\underset{(IS)}{=} 2F_n^2 + 2F_nF_{n-1} + F_{n-1}^2 + (F_n^2 - F_{n-1}^2 + (-1)^n) \\ &= 3F_n^2 + 2F_nF_{n-1} + (-1)^n \\ &= 4F_n^2 + 4F_nF_{n-1} + F_{n-1}^2 - F_n^2 - 2F_nF_{n-1} - F_{n-1}^2 + (-1)^n \\ &= (2F_n + F_{n-1})^2 - (F_n + F_{n-1})^2 + (-1)^n \\ &= F_{n+2}^2 - F_{n+1}^2 + (-1)^n \end{aligned}$$

□

5. Die Implementation erhalten wir dann schließlich durch die Binärdarstellung von n:

⟨ fast Fibonacci algorithm 266 ⟩ ≡

```
Digit a = Digit(1) << log2(n);
do {
  k = j*j; t = i*i;
  j = k+t;                    // (4.2)
  --k; i = t-k;
  i <<= 1; i += t;            // (4.4)
  while (n&a) {
    j += i; swap(i, j);
    a >>= 1;
    if (a == 0) return i;
    k = j*j; t = i*i;
    j = k+t;                  // (4.2)
    i = t-k;
    --i; i <<= 1; i += t;     // (4.4)
  }
} while (a >>= 1);
return i;◇
```

Macro referenced in 22a.

5.2 Der größte gemeinsame Teiler

Definition. Sei $n \in \mathbb{N}$, so setze für alle $a_k \in \mathbb{Z}, 0 \leq k \leq n$:

$$\begin{aligned} \operatorname{lcm}(a_k)_{k=0}^{n} &:= \min\{x \in \mathbb{N} \mid a_k | x, 0 \leq k \leq n\} \\ \gcd(a_k)_{k=0}^{n} &:= \max\{x \in \mathbb{N} \mid x | a_k, 0 \leq k \leq n\}, \end{aligned}$$

wobei die Konvention $\gcd(0^n) := 0$ gilt.

5.2.1 Einschränkung auf zwei Argumente

Es gilt die Eigenschaft:

$$\gcd(a, b, c) = \gcd(a, \gcd(b, c)) = \gcd(\gcd(a, b), c).$$

Dies ist auch für eine beliebige Anzahl an Argumenten möglich, was programmtechnisch mit den Makros aus der Standarddatei `<stdarg.h>` realisiert wird. Der Vorteil dieser Programmierweise ist, daß sie nicht nur komfortabel, sondern auch durch interne Sortiermethoden schneller ist.

Beispiel. Der Aufruf

$$\gcd(\gcd(F_{21}, F_{17}), F_{14})$$

mit den Fermat-Zahlen $F_n := 2^{2^n} + 1$ benötigt auf dem GNU-Compiler 2.7.2 72,6 Sekunden und ist im Vergleich zur äquivalenten Lösung

$$\gcd(F_{21}, \gcd(F_{17}, F_{14}))$$

mit 0,6 Sekunden deutlich langsamer.

Beliebig viele Argumente können folgendermaßen implementiert werden:

```
Natural gcd(const Natural& a, const Natural& b, ...)
{
  va_list pData;
  va_start(pData, b);
  /*
      gcd Algorithm
  */
  va_end(pData);
}
```

Die Hürde ist dabei, die unbekannten Argumente zu übernehmen, weil der Zugriff ja über Makros erfolgt. So muß bei jedem Aufruf des Makros `va_arg(pData, Natural)` der Typ des erwarteten Arguments angegeben werden, da man normalerweise nicht feststellen kann, ob wirklich ein Argument diesen Typs übergeben wird.
Damit würde zum Beispiel die folgende Konstruktion fehlerhafte Ergebnisse liefern:

```
Digit a = 641;
Natural b = Natural(1) << 32;
cout << "gcd=" << gcd(a, ++b) << endl;
```

Hier wäre es notwendig, stattdessen `gcd(Natural(a), Natural(b))` aufzurufen, was dem Anwender ein zusätzliches Maß an Aufmerksamkeit abverlangt und gegen unsere Konvention der bedingungslosen

Argumentübergabe aus Kapitel 1.1 verstößt.

Aus diesem Grund werden wir uns in dieser Arbeit lediglich mit der Berechnung des größten gemeinsamen Teilers von zwei Argumenten befassen.

5.2.2 Euklidischer Algorithmus für Digits

Der bekannteste Algorithmus zur Ermittlung des größten gemeinsamen Teilers stammt von Euklid aus der Zeit um 300 vor Christus:

```
Digit gcd(Digit a, Digit b)
// Algorithm:  c := gcd(a, b)
// Input:      a,b in Digit.
// Output:     c in Digit such that c = gcd(a, b) ||
{
  while (b) {
    a %= b;
    if (a == 0) return b;
    b %= a;
  }
  return a;
}
```

Hierbei nutzen wir die folgende rekursive Eigenschaft des größten gemeinsamen Teilers aus:

$$\gcd : \mathbb{N}^2 \to \mathbb{N} : (a,b) \mapsto \begin{cases} \gcd(b, a \bmod b)\,, & a \geq b \\ \gcd(a, b \bmod a)\,, & a < b \\ a\,, & b = 0 \\ b\,, & a = 0 \end{cases}.$$

5.2.3 Binärer Algorithmus von Stein

Josef Stein entdeckte 1961 ([47]) einen binären Algorithmus, der auf den folgenden rekursiven Eigenschaften beruht:

$$\gcd : \mathbb{N}^2 \to \mathbb{N} : (a,b) \mapsto \begin{cases} 2\gcd(\frac{a}{2}, \frac{b}{2})\,, & 2 \mid a, 2 \mid b \\ \gcd(\frac{a}{2}, b)\,, & 2 \mid a, 2 \nmid b \\ \gcd(a, \frac{b}{2})\,, & 2 \nmid a, 2 \mid b \\ \gcd(|a-b|, b)\,, & 2 \nmid a, 2 \nmid b \end{cases}.$$

Hierfür werden nur Subtraktionen und Divisionen durch 2 benötigt, die schneller als eine beliebige Division von einem Rechner verarbeitet werden. Durch diese Eigenschaft ist das folgende Programm im Digit-Bereich deutlich schneller als der herkömmliche Euklidische Algorithmus aus dem vorherigen Abschnitt, obwohl mehr Iterationsschritte durchgeführt werden:

⟨ greatest common divisor of two Digits 268 ⟩ ≡

```
Digit gcd(Digit a, Digit b)
// Algorithm:  c := gcd(a, b)
// Input:      a,b in Digit.
// Output:     c in Digit such that c = gcd(a, b) ||
{
  if (a == 0) return b;
  if (b == 0) return a;
```

```
  Digit i,j;
  for (i = 0; (a&1) == 0; ++i) a >>= 1;
  for (j = 0; (b&1) == 0; ++j) b >>= 1;
  while (a != b)
    if (a > b) {
      a -= b;
      do a >>= 1; while ((a&1) == 0);
    } else {
      b -= a;
      do b >>= 1; while ((b&1) == 0);
    }
  return a << ((i > j)? j : i);
}
◇
```

Macro referenced in 398b.

Laufzeit. Seien $a, b \in$ Digit, dann gcd$(a,b) \in \mathcal{O}(1)$.

Selbst bei zwei Digits weist der binäre Algorithmus ein besseres Verhalten als der Euklidische auf, obwohl der Code nun erheblich anwächst:

⟨ greatest common divisor of two double Digits 269 ⟩ ≡

```
void NumberBase::gcd(Digit a, Digit b, Digit c, Digit d, Digit& x, Digit& y) const
// Algorithm:  n.gcd(a, b, c, d; x, y)
// Input:      n in NumberBase, a,b,c,d in Digit.
// Output:     x,y in Digit such that x*2^BETA+y = gcd(a*2^BETA+b, c*2^BETA+d) ||
{
  if (a == 0 && b == 0) { x = c; y = d; }
  else if (c == 0 && d == 0) { x = a; y = b; }
  else {
    Digit i = 0;
    Digit j = 0;
    ⟨ find power of 2 in the binary gcd algorithm for double Digits 270a ⟩
    while (b != d && a != c)
      if (a > c) {
        a -= c + (b < d); b -= d;
        j = 0;
        do { ++j; b >>= 1; } while ((b&1) == 0);
        b |= a << (BETA-j); a >>= j;
      } else {
        c -= a + (d < b); d -= b;
        j = 0;
        do { ++j; d >>= 1; } while ((d&1) == 0);
        d |= c << (BETA-j); c >>= j;
      }
    if (b != d || a != c) {
      if (a == c) {
        if (b > d) {
          b -= d;
          digitmod(c, d, b, a);
          a = ::gcd(a, b);
        } else {
          d -= b;
          digitmod(a, b, d, c);
          a = ::gcd(c, d);
```

```
        }
      } else {
        if (a > c) {
          a -= c;
          digitmod(c, d, a, b);
          a = ::gcd(a, b);
        } else {
          c -= a;
          digitmod(a, b, c, d);
          a = ::gcd(c, d);
        }
      }
      if (i >= BETA) { x = a << (i-BETA); y = 0; }
      else if (i) { x = a >> (BETA-i); y = a << i; }
      else { x = 0; y = a; }
    } else if (i >= BETA) { x = b << (i-BETA); y = 0; }
    else if (i) { x = a << i | b >> (BETA-i); y = b << i; }
    else { x = a; y = b; }
  }
}
◇
```

Macro referenced in 398b.

Laufzeit. Sei $n \in \mathsf{NumberBase}$ und $a_0, a_1, b_0, b_1, d_0, d_1 \in \mathsf{Digit}$, dann $n\,.\,\mathtt{gcd}(a_0\,,a_1\,,b_0\,,b_1\,,d_0\,,d_1) \in \mathcal{O}(1)$.

Hierbei wird zunächst die größte Zweierpotenz aus den Eingabe-Digits herausgezogen, um dann später wieder zum Resultat hinzumultipliziert zu werden:

⟨ find power of 2 in the binary gcd algorithm for double Digits 270a ⟩ ≡

```
if (b == 0) {
  b = a; a = 0;
  for (i = BETA; (b&1) == 0; ++i) b >>= 1;
} else if ((b&1) == 0) {
  do { ++i; b >>= 1; } while ((b&1) == 0);
  b |= a << (BETA-i); a >>= i;
}
if (d == 0) {
  d = c; c = 0;
  for (j = BETA; (d&1) == 0; ++j) d >>= 1;
} else if ((d&1) == 0) {
  do { ++j; d >>= 1; } while ((d&1) == 0);
  d |= c << (BETA-j); c >>= j;
}
if (j < i) i = j;◇
```

Macro referenced in 269.

5.2.4 Euklidischer Algorithmus für Naturals

Bei der Berechnung des größten gemeinsamen Teilers für Naturals verwenden wir zuerst den Euklidischen Algorithmus für Langzahlen und wechseln dann bei einer Länge kleiner oder gleich zwei über zum binären Algorithmus für Digits.

⟨ greatest common divisor of two Naturals 270b ⟩ ≡

```
Natural gcd(Natural a, Natural b)
// Algorithm:  c := gcd(a, b)
// Input:      a,b in Natural.
// Output:     c in Natural such that c = max{x in Natural : x|a, x|b} ||
{
  size_t sA = a.size;
  size_t sB = b.size;
  // small sizes
  if (sA == 1) {
    Digit x = *a.p;
    if (sB == 2) {
      Digit y;
      a.gcd(b.p[0], b.p[1], 0, x, x, y);
      if (x == 0) return y;
      Natural t(x, size_t(2));
      t.p[1] = y;
      return t;
    }
    if (x == 0) return b;
    if (sB > 2) b %= x;
    return gcd(x, *b.p);
  } else if (sB == 1) {
    Digit x = *b.p;
    if (sA == 2) {
      Digit y;
      a.gcd(a.p[0], a.p[1], 0, x, x, y);
      if (x == 0) return y;
      Natural t(x, size_t(2));
      t.p[1] = y;
      return t;
    }
    if (x == 0) return a;
    a %= x;
    return gcd(x, *a.p);
  } else if (sA == 2 && sB == 2) {
    Digit x,y;
    a.gcd(a.p[0], a.p[1], b.p[0], b.p[1], x, y);
    if (x == 0) return y;
    Natural t(x, size_t(2));
    t.p[1] = y;
    return t;
  }
  if (a == b) return a;
  // extract factor 2
  size_t i = 0;
  Digit* pA = a.p+sA;
  if (*--pA == 0) {
    if (sA == 1) return b;
    do i += BETA; while (*--pA == 0);
  }
  size_t j = 0;
  Digit* pB = b.p+sB;
  if (*--pB == 0) {
    if (sB == 1) return a;
    do j += BETA; while (*--pB == 0);
  }
```

```
// count trailing zeros
Digit x = *pA;
i += log2(x&-x);
if (i) a >>= i;
x = *pB; j += log2(x&-x);
if (j) b >>= j;
if (i > j) i = j;
sA = a.size; sB = b.size;
// small sizes
if (sA == 1) {
  x = *a.p;
  if (sB == 2) {
    Digit y;
    a.gcd(b.p[0], b.p[1], 0, x, x, y);
    if (x == 0) {
      if (log2(y)+i < BETA) return y << i;
      Natural t = y;
      return t <<= i;
    }
    Natural t(x, size_t(2));
    t.p[1] = y;
    return t <<= i;
  } else if (sB > 2) b %= x;
  x = gcd(x, *b.p);
  if (log2(x)+i < BETA) return x << i;
  Natural t = x;
  return t <<= i;
} else if (sB == 1) {
  x = *b.p;
  if (sA == 2) {
    Digit y;
    a.gcd(a.p[0], a.p[1], 0, x, x, y);
    if (x == 0) {
      if (log2(y)+i < BETA) return y << i;
      Natural t = y;
      return t <<= i;
    }
    Natural t(x, size_t(2));
    t.p[1] = y;
    return t;
  }
  a %= x;
  x = gcd(x, *a.p);
  if (log2(x)+i < BETA) return x << i;
  Natural t = x;
  return t <<= i;
} else if (sA == 2 && sB == 2) {
  Digit y;
  a.gcd(a.p[0], a.p[1], b.p[0], b.p[1], x, y);
  if (x == 0) {
    if (log2(y)+i < BETA) return y << i;
    Natural t = y;
    return t <<= i;
  }
  Natural t(x, size_t(2));
  t.p[1] = y;
```

```
  return t <<= i;
}

// k-ary gcd algorithm
Natural t1,t2;
const size_t K_ARY_BETA = (BETA > 32)? 16 : BETA/2;
const Digit K_ARY = Digit(1) << K_ARY_BETA;
const Digit K_BASE = K_ARY-1;
// first trial division phase
Natural n = 1;
pA = NumberBase::k_ary_gcd_trial_division;
sA = NumberBase::k_ary_gcd_trial_division_size;
for (j = 0; j < sA; ++j) {
  x = pA[j];
  while (true) {
    Digit y1,y2;
    div(a, x, t1, y1);
    if (y1 > 0) y1 = gcd(y1, x);
    if (y1 == 1) break;
    div(b, x, t2, y2);
    if (y2 > 0) y2 = gcd(y2, x);
    if (y2 == 1) break;
    if (y1 == y2) {
      if (y1 == 0) { n *= x; swap(a, t1); swap(b, t2); }
      else { x = y1; n *= x; a /= x; b /= x; }
    } else {
      x = gcd(y1, y2);
      if (x == 1) break;
      n *= x; a /= x; b /= x;
    }
  }
}

// Euclidean algorithm for numbers with a large difference
if (a < b) swap(a, b);
while (b != 0) {
  sA = a.size;
  sB = b.size;
  if (sA <= sB+1) break;
  if (sA == 2) {
    Digit x,y;
    if (sB == 2) a.gcd(a.p[0], a.p[1], b.p[0], b.p[1], x, y);
    else a.gcd(a.p[0], a.p[1], 0, b.p[0], x, y);
    b = 0;
    if (x == 0) a = y;
    else { a = x; a.lmove(1); a |= y; }
    break;
  } else if (sA == 1) { a = gcd(a.p[0], b.p[0]); b = 0; break; }
  div(a, b, t1, a);
  swap(a, b);
}
// main loop
while (a != 0 && b != 0) {
  sA = a.size;
  sB = b.size;
  x = a&K_BASE;
```

```
    if (x == 0) a >>= K_ARY_BETA;
    else if ((x&1) == 0) a >>= log2(x&-x);
    else {
      Digit y = b&K_BASE;
      if (y == 0) b >>= K_ARY_BETA;
      else if ((y&1) == 0) b >>= log2(y&-y);
      else {
        y = (x*NumberBase::k_ary_gcd_inverse[y >> 1])&K_BASE;
        x = NumberBase::k_ary_gcd_linear_id[y];
        if (NumberBase::k_ary_gcd_linear_id_neg[y]) {
          y = (x*y)&K_BASE;
          if (a >= b) {
            a *= x; t1 = b*y; a = abs(a, t1) >> K_ARY_BETA;
          } else {
            b *= y; t1 = a*x; b = abs(b, t1) >> K_ARY_BETA;
          }
        } else {
          y = K_ARY - ((x*y)&K_BASE);
          if (a >= b) { a *= x; a += y*b; a >>= K_ARY_BETA; }
          else { b *= y; b += x*a; b >>= K_ARY_BETA; }
        }
      }
    }
  }
  // trial division phase
  if (a == 0) swap(a, b);
  sA = NumberBase::k_ary_gcd_trial_division_size;
  for (j = 0; j < sA; ++j) {
    x = pA[j];
    while (true) {
      const Digit y = gcd(a%x, x);
      if (y == 1) break;
      a /= y;
    }
  }
  if (n > 1) a *= n;
  return a << i;
}
◇
```

Macro referenced in 409.

Laufzeit. Seien $x, y \in$ Natural, dann gcd$(x,y) \in \mathcal{O}(n^2)$.

Bemerkung. Es existieren heute schon weitaus schnellere Algorithmen wie zum Beispiel der von K. Weber ([52]).
Diese werden wir jedoch erst in einer späteren Arbeit behandeln und implementieren.

5.2.5 Der erweiterte Euklidische Algorithmus

In diesem Abschnitt nutzen wir die folgende Eigenschaft der ganzen Zahlen aus:
Es sei $n \in \mathbb{N}$ und $a_i \in \mathbb{N}_+ (0 \leq i \leq n)$, so existieren $b_i \in \mathbb{Z}$ mit $\gcd(a_i)_{i=0}^n = \sum_{j=0}^n a_j b_j$ für $0 \leq i \leq n$.

Da uns aber nur der größte gemeinsame Teiler von zwei Argumenten interessiert (siehe Kapitel 5.2.1), implementieren wir hier nur den Fall $n = 2$:

Für alle $a, b \in \mathbb{Z}$, existieren $x, y, z \in \mathbb{Z}$ mit $\gcd(a, b) = z = ax + by$.

⟨ extented greatest common divisor of two Integers 275a ⟩ ≡

```
void gcd(Integer a, Integer b, Integer& x, Integer& y, Integer& z)
// Algorithm:  gcd(a, b, x, y, z)
// Input:      a,b in Integer.
// Output:     x,y,z in Integer such that z = a*x + b*y = gcd(a, b) ||
{
  if (&x == &y || &x == &z || &y == &z) z.errmsg(5, "(gcd)");
  x = 0; y = 1;
  if (b == 0) z = a;
  else {
    const int as = sign(a);
    const int bs = sign(b);
    if (as == -1) a = -a;
    if (bs == -1) b = -b;
    Integer u = 1;
    Integer v = SignDigit(0);
    Integer t;

    while (true) {
      div(a, b, z, a);
      if (a == 0) { z = b; break; }
      t = x*z; u -= t;
      t = y*z; v -= t;

      div(b, a, z, b);
      if (b == 0) { z = a; x = u; y = v; break; }
      t = u*z; x -= t;
      t = v*z; y -= t;
    }
    if (as == -1) x = -x;
    if (bs == -1) y = -y;
  }
}
```

◇

Macro referenced in 426.

Laufzeit. Seien $u_0, u_1, v, w_0, w_1 \in$ Integer, dann gcd$(u_0, u_1, v, w_0, w_1) \in \mathcal{O}(n^2)$.

5.2.6 Das kleinste gemeinsame Vielfache

Um das kleinste gemeinsame Vielfache zu ermitteln, müssen wir nur die folgende Beziehung zwischen dem größten gemeinsamen Teiler und dem kleinsten gemeinsamen Vielfachen verwenden:

$$ab = \gcd(a, b) \cdot \mathrm{lcm}(a, b) \qquad \text{für } a, b \in \mathbb{N}.$$

⟨ least common multiple of two Naturals 275b ⟩ ≡

```
Natural lcm(const Natural& a, const Natural& b)
// Algorithm:  c := lcm(a, b)
// Input:      a,b in Natural.
// Output:     c in Natural such that c = min{x in Natural : a|x, b|x} ||
{
  if (a == 0) return b;
```

```
  if (b == 0) return a;

  Natural q,r;
  div(a, gcd(a, b), q, r);
  return q*b;
}
```
◇

Macro referenced in 409.

Laufzeit. Seien $x, y \in$ Natural, dann `lcm(`x`,`y`)` $\sim$ `gcd(`x`,`y`)`.

5.3 Pell'sche Gleichung

Sei $x \in \mathbb{N}$ und kein Quadrat, dann ist die *Pell'sche Gleichung*, die ganzzahlige Lösung von

$$u^2 - xv^2 = 1.$$

Eine schnelle Lösung dieser Gleichung erfolgt über die Kettenbruchentwicklung von $\sqrt{x}$ und kann z. B. in [17] nachgelesen werden.

⟨ solving Pell's equation 276 ⟩ ≡

```
void pell(const Digit x, Natural& u, Natural& v)
// Algorithm:  pell(a, b, c)
// Input:      a in Digit where not sqrt(a) = [sqrt(a)].
// Output:     b,c in Natural such that b^2 + a * c^2 = 1 ||
{
  short sign = -1;
  Digit q;
  if (x == 5) {
    u = 1; v = 1; q = 4;
  } else {
    Digit w;
    sqrt(x, w, q);
    if (q == 0) { u = v = 0; return; }
    u = w; v = 1;
    Digit q0 = 1;
    Digit m = w;
    Natural u0 = 1;
    Natural v0 = Digit(0);
    while (q != 4 && q != 1) {
      const Digit a = (m+w)/q;
      const Digit m1 = a*q - m;
      const Digit q1 = (m-m1)*a + q0;
      Natural u1 = a*u;
      u1 += u0;
      Natural v1 = a*v;
      v1 += v0;
      m = m1; q0 = q; q = q1;
      u0 = u; u = u1; v0 = v; v = v1;
      sign = -sign;
    }
  }
  if (q == 4) {
    Natural u2 = u*u;
```

```
    if (sign < 0) {
      ++u2;
      Natural t = u2 >> 1;
      v *= t;
      u2 += 2;
    } else {
      --u2;
      Natural t = u2 >> 1;
      v *= t;
      u2 -= 2;
    }
    u2 >>= 1;
    u *= u2;
  }
  if (sign < 0) {
    v *= u; v <<= 1;
    u *= u; u <<= 1; ++u;
  }
}
◇
```

Macro referenced in 433.

5.4 Kongruenzen

Definition. Für $a, b, m \in \mathbb{Z} : a \equiv b \pmod{m} \Leftrightarrow a - b \in m\mathbb{Z} \Leftrightarrow$ es existiert ein $k \in \mathbb{Z} a = b + k \cdot m$.

Es werden vorerst nur einige Eigenschaften der modularen Arithmetik zum besseren Verständnis der Primfaktorisierung eingeführt. In einer späteren Arbeit werden wir diese Arithmetik noch ausführlicher betrachten.

5.4.1 Modulares Potenzieren

Das modulare Potenzieren läßt sich analog zum Potenzieralgorithmus aus Kapitel 2.13 implementieren.

⟨ modular power algorithm 277 ⟩ ≡

```
template<class T>
T pow(T a, T b, const T& c)
// Algorithm:  d := pow(a, b, c)
// Input:      a,b,c in T where c > 0.
// Output:     d in T such that d == a^b (mod c) ||
{
  a %= c;
  if (b == 1) return a%c;
  else if (b == 0) return 1;
  else if (a == 0) return Digit(0);

  while ((b&1) == 0) { a *= a; a %= c; b >>= 1; }

  T z = a;
  while (--b != 0) {
    while ((b&1) == 0) { a *= a; a %= c; b >>= 1; }
    z *= a; z %= c;
  }
```

```
  return z;
}
◇
```

Macro referenced in 432a.

5.4.2 Berechnung des modularen Inversen

Um zu gegebenem $a \in \mathbb{Z}$ sein Inverses x mit $ax \equiv 1 \pmod{m}$ zu finden, wird der Euklidische Algorithmus durchgeführt. Dieser liefert dann

$$ax + my = \gcd(a, m) \qquad \Rightarrow \qquad ax \equiv \gcd(a, m) \pmod{m}.$$

x ist das Inverse von a in der Restklasse mod m, wenn noch zusätzlich $\gcd(a, m) = 1$ gilt.

⟨ modular inverse 278 ⟩ ≡

```
template<class T>
T inverse(const T& x, const T& m)
// Algorithm:  c := inverse(a, b)
// Input:      a,b in T where b > 0.
// Output:     c in T such that c == a^(-1) (mod b) ||
{
  T a = x;
  T b = m;
  T q = 1;
  T p = 0;
  T s,t;
  do {
    t = a/b;
    a -= s = t*b;
    q += s = t*p;
    if (a == 0) return m - p;
    t = s = b/a;
    b -= s = t*a;
    p += s = t*q;
  } while (b >= 1);
  return q;
}
◇
```

Macro referenced in 432a.

Laufzeit. Seien $x, y \in T$, dann `inverse(x,y)`$\in \mathcal{O}(n^2)$.

5.4.3 Simultane Kongruenzen

Chinesischer Restsatz. Sind $\{m_i\}_{i=1}^n$ paarweise teilerfremde ganze Zahlen und $m = \prod_{i=1}^n m_i$ und $b, \{a_i\}_{i=1}^n$ beliebige ganze Zahlen, dann existiert genau ein x mit

$$b \le x < b + m \text{ und } x \equiv a_i \pmod{m_i} \qquad \text{für alle } 1 \le i \le n.$$

Wenn wir den Chinesischen Restsatz nur für den speziellen Fall mit zwei Kongruenzen lösen wollen, so kann man die Lösung einfach explizit angeben:
Gegeben. $x \equiv a_1 \pmod{m_1}$ und $x \equiv a_2 \pmod{m_2}, \gcd(m_1, m_2) = 1$.

Gesucht. x.
Sei $c \cdot m_1 \equiv 1 \pmod{m_2}$, so $x \equiv (a_2 - a_1) \cdot c \cdot m_1 + a_1 \pmod{m_1 m_2}$.

Bei der Umsetzung verallgemeinern wir den Chinesischen Restsatz, indem die Moduli nicht paarweise teilerfremd sein müssen (siehe zum Beispiel [3] auf Seite 106).

⟨ chinese remainder theorem 279 ⟩ ≡

```
template<class T>
T chinese(const T& m1, const T& m2, const T& a1, const T& a2)
// Algorithm:  x := chinese(m1, m2, a1, a2)
// Input:      m1,m2,a1,a2 in T where 0 <= a1 < m1, 0 <= a2 < m2.
// Output:     x in T such that x == a1 (mod m1) and x == a2 (mod m2)
//             where 0 <= x < m1*m2 ||
{
  const T d = gcd(m1, m2);
  if (d == 1) {
    const T c = inverse(m1, m2);
    const T m = m1*m2;
    T t;
    if (a2 >= a1) t = a2-a1;
    else { t = m-a1; t += a2; }
    t *= c; t *= m1; t += a1;
    return t %= m;
  }
  T m = m1/d;
  const T b = m2/d;
  const T c = inverse(m, b);
  m *= m2;
  T t = a2%d;
  if (t != a1%d) return T();
  if (a2 >= a1) { t = a2-a1; t /= d; }
  else { t = a1-a2; t /= d; t %= b; t = b-t; }
  t *= c; t *= m1; t += a1;
  return t %= m;
}

template<class Container>
void chinese(const typename Container::value_type& m1,
             const typename Container::value_type& m2,
             const Container& l1, const Container& l2,
             Container& r, typename Container::value_type& m)
// Algorithm:  chinese(m1, m2, l1, l2, r, m)
// Input:      m1,m2 in T, l1,l2 are containers over T
//             where 0 <= a1 < m1, 0 <= a2 < m2.
{
  const typename Container::value_type d = gcd(m1, m2);
  const typename Container::value_type b = m2/d;
  const typename Container::value_type b2 = m1/d;
  const typename Container::value_type c = inverse(b2, b);
  m = b2*m2;
  typename Container::value_type s,t;

  Container* r2 = &r;
  if (r2 == &l1 || r2 == &l2) r2 = new Container();
  r2->clear();
  const typename Container::const_iterator j1 = l1.end();
```

```
      for (typename Container::const_iterator i1 = l1.begin(); i1 != j1; ++i1) {
        s = *i1 % d;
        const typename Container::const_iterator j2 = l2.end();
        for (typename Container::const_iterator i2 = l2.begin(); i2 != j2; ++i2)
          if (*i2 % d == s) {       // condition: x1 == x2 (mod gcd(m1, m2))
            if (*i2 >= *i1) { t = *i2 - *i1; t /= d; }
            else { t = *i1 - *i2; t /= d; t %= b; t = b - t; }
            t *= c; t *= m1; t += *i1; t %= m;
            r2->push_back(t);
          }
        }
        if (&r != r2) { r = *r2; delete r2; }
      }
      ◇
```

Macro referenced in 432a.

5.5 Quadratische Reste / Jacobi-Symbol

Definition. Seien $x, m \in \mathbb{Z}$. Wir nennen x einen quadratischen Rest mod m, falls ein $a \in \mathbb{Z}$ existiert mit $a^2 \equiv x \pmod{m}$, andernfalls einen quadratischen Nichtrest.
Sei $p \geq 3$ Primzahl, $x \in \mathbb{Z}$, so ist das Legendre-Symbol folgendermaßen definiert:

$$\left(\frac{x}{p}\right) := \begin{cases} 1\,, & \text{ex. } a \in \mathbb{Z} \text{ mit } a^2 \equiv x \pmod{p}. \\ 0\,, & p \mid x. \\ -1\,, & \text{für alle } a \in \mathbb{Z}\ a^2 \not\equiv x \pmod{p}. \end{cases}.$$

Einfaches Durchtesten

Ein erster Ansatz, der nur die Tatsache ausnutzt, daß in der Einheitengruppe $(\mathbb{Z}/p\mathbb{Z})^*$ genausoviele quadratische Reste wie Nichtreste existieren, und die Symmetrie

$$(p-x)^2 = p^2 - 2px + x^2 \equiv x^2 \pmod{p}$$

ausnutzt, sieht wie folgt aus:

```
void root(const Digit p)
// Algorithm:  root(p)
// Input:      p prim ||
{
  char* r = new char[p];
  if (!r) { cerr << "Out of Memory!\n"; return; }

  Digit i;
  for (i = 0; i < p; ++i) r[i] = 0;
  i = 1;                          // Null auslassen
  for (Digit j = 3; j <= p; j += 2) {
    r[i] = 1; i += j;
    if (i >= p) i -= p;
  }

  // Falls r[i] = 1, so ist i Quadratischer Rest mod p
  delete[] r;
}
```

Hierbei wird für große Zahlen p viel Speicher (genau p Bytes) alloziiert, und es wächst die Laufzeit linear zu p.

Das Legendre-Symbol

Quadratisches Reziprozitätsgesetz. Seien $p, q \geq 3$ Primzahlen, so gilt:

$$\left(\frac{p}{q}\right) = (-1)^{\frac{p-1}{2}\cdot\frac{q-1}{2}}\left(\frac{q}{p}\right) = \begin{cases} \left(\frac{q}{p}\right), & p \equiv 1 \pmod 4 \text{ oder } q \equiv 1 \pmod 4 \\ -\left(\frac{q}{p}\right), & p \equiv 3 \pmod 4 \text{ und } q \equiv 3 \pmod 4 \end{cases}.$$

Desweiteren gelten die **Ergänzungssätze**:

$$\left(\frac{2}{p}\right) = (-1)^{\frac{p^2-1}{8}} = \begin{cases} 1, & p \equiv \pm 1 \pmod 8 \\ -1, & p \equiv \pm 3 \pmod 8 \end{cases},$$

die bis ± 101 in [41] aufgeführt werden.

Das Jacobi-Symbol

Eine Implementation des Legendre-Symbols ist aber nicht sehr schön, weil hierbei Bedingungen an den Benutzer gestellt werden, was gegen unsere Anfangsforderung (Kapitel 1.1) verstößt. Das Programm wird schneller und allgemeingültig, wenn wir stattdessen das Jacobi-Symbol verwenden:
Definition. Seien $n, m \in \mathbb{N}$ mit $\gcd(n, m) = 1$, m ungerade und $m = \prod_{i=0}^{k} p_i^{a_i}$ für $k \in \mathbb{N}, a_i \in \mathbb{N}_+$ die Primfaktorzerlegung, so setze:

$$\underbrace{\left(\frac{n}{m}\right)}_{\text{Jacobi-Symbol}} := \prod_{i=0}^{k} \underbrace{\left(\frac{n}{p_i}\right)}_{\text{Legendre-Symbol}}$$

Nach J. O. Shallit und J. Sorenson [45] kann das Jacobi-Symbol analog zum Euklidischen Algorithmus (siehe Abschnitt 5.2.3 auf Seite 268) binär implementiert werden.

⟨ jacobi symbol 281 ⟩ ≡

```
template<class T>
int jacobi(T a, T b)
// Algorithm:  c := jacobi(a, b)
// Input:      a,b in T where a,b > 0 and not 2 | b.
// Output:     c in {-1,0,1} such that c = (a|b) ||
{
  if (a == 1) return 1;
  if (a == 2) {
    const int c = b&7;
    return (c == 1 || c == 7)? 1 : -1;
  }
  int h,c = 1;
  while (a != 0) {
    for (h = 0; (a&1) == 0; ++h) a >>= 1;
    int d = b&7;
    if (h&1 && d != 1 && d != 7) c = -c;
    if (a < b) {
      if ((d&3) == 3 && (a&3) == 3) c = -c;
      d = a&7;
      swap(a, b);
```

```
    }
    a -= b; a >>= 1;
    if (d == 3 || d == 5) c = -c;
  }
  return (b == 1)? c : 0;
}
```
◇

Macro referenced in 432b.

Laufzeit. Seien $x, y \in T$, dann `jacobi(`x`,`y`)`$\in \mathcal{O}(n^2)$.

Hierbei setzen wir noch $\left(\frac{n}{m}\right) := 0$ für den Fall $\gcd(n, m) > 1$.

Bemerkung. Auch das Jacobi-Symbol kann wie das Legendre-Symbol bei Primzahlen eine Aussage über den quadratischen Rest machen. Bei einer Nicht-Primzahl ist es lediglich eine Rechenerweiterung des Legendre-Symbols.

Beispiel. $\left(\frac{5}{9}\right) = \left(\frac{2}{3}\right)^2 = 1$, aber 5 ist kein Quadrat mod 9.
Wenn jedoch $\left(\frac{n}{m}\right) = -1$ gilt, so ist n quadratischer Nichtrest mod m.

5.6 Quadratwurzel modulo einer Primzahl

Für den Fall $m \equiv 3 \pmod 4$ läßt sich die Quadratwurzel durch das einfache Potenzieren lösen und folgt aus dem Euler-Kriterium. Der Fall $m \equiv 1 \pmod 4$ ist etwas aufwändiger und kann nur in einer quadratischen Körpererweiterung berechnet werden. Genaueres hierzu findet man zum Beispiel in [17] auf der Seite 135. Es existiert jedoch für den Fall $m \equiv 5 \pmod 8$ eine schnelle und direkte Lösung (siehe [27] auf Seite 681).

⟨ modular square root 282 ⟩ ≡

```
template<class T>
T sqrt(const T& a, const T& b)
// Algorithm:  c := sqrt(a, b)
// Input:      a,b in T where b is prime and (a|b) = 1.
// Output:     c in T such that c^2 == a (mod b) ||
{
  if ((b&3) == 3) {
    T c = b+1;
    c >>= 2;
    return pow(a, c, b);
  } else if ((b&7) == 5) {
    const T c = a << 1;
    const T v = pow<T>(c, (b - 5) >> 3, b);
    T i = c*(v*v);
    --i; i *= a*v;
    return i % b;
  }

  Digit k = 1;
  T d = b-a;
  ++d;
```

```
  while (true) {
    const int x = jacobi(d, b);
    if (x == 0) return 1;
    else if (x < 0) break;
    d += 2*k+1; ++k;
  }

  T b3 = k;
  d %= b;
  // (k + sqrt(d))^([b/2]+1):
  T c = b >> 1; ++c;
  T t,t2,t3,b2 = 1;
  T x = 1;
  T x2 = 0;

  while (c != 0) {
    while ((c&1) == 0) {
      t = b3*b3; t2 = b2*b2; t += d*t2;
      t2 = b3*b2; t2 <<= 1;
      b3 = t % b; b2 = t2 % b;
      // (b3, b2) := (b3*b3 + d*b2*b2, 2*b2*b3) = (b3, b2)^2
      c >>= 1;
    }
    t = b3+b2; t2 = x+x2; t3 = b2*x2;
    x2 = t*t2; x2 -= t = b3*x; x2 -= t3;
    x2 %= b;
    t += d*t3;
    x = t % b;
    // (x, x2) := (b3*x+d*b2*x2, (b3+b2)*(x+x2)-b3*x-b2*x2) = (b3, b2)*(x, x2)
    --c;
  }
  return x;
}
◇
```

Macro referenced in 432a.

5.7 Primzahlen und die Primfaktorzerlegung

Definition. $\mathbb{P}$ sei die Menge aller Primzahlen.

Definition. Eine Zahl $n \in \mathbb{Z}$ heißt **unzerlegbar**, falls für alle $x \in \mathbb{Z} - \mathbb{Z}^*$ aus $x \mid n$ stets $x \in \{\pm 1\}$ oder $x \in \{\pm n\}$ folgt.

Definition. Eine Zahl $n \in \mathbb{Z}$ heißt **Primzahl**, falls für alle $x, y \in \mathbb{Z}$ aus $n \mid x \cdot y$ stets $n \mid x$ oder $n \mid y$ folgt.

Bemerkung. Für den Ring $\mathbb{Z}$ ist die Eigenschaft "unzerlegbar" und die Bezeichnung "Primzahl" gleichbedeutend.

Alle Methoden, die wir im folgenden betrachten werden, beruhen auf dem

Fundamentalsatz der Zahlentheorie. Jede natürliche Zahl $n \geq 2$ läßt sich eindeutig als das Produkt von Primzahlen darstellen:

$$n = \prod_{i=0}^{t} p_i^{a_i} \qquad \text{für } t \in \mathbb{N}, a_i \in \mathbb{N}_+, 0 \leq i \leq t.$$

5.7.1 Siebmethode von Eratosthenes

Primzahlenermittlung

Doch bevor wir genauer auf die Primfaktorzerlegung einer natürlichen Zahl eingehen, benötigen wir zunächst die Ermittlung einer Menge endlich vieler Primzahlen. Hierzu beschrieb bereits um 230 vor Christus der griechische Gelehrte Eratosthenes von Kyrene einen Algorithmus zur Auffindung aller Primzahlen zwischen 2 und $n \in \mathbb{N}$, mit welchem alle zusammengesetzten Zahlen aus der Menge $\{2, \ldots, n\}$ entfernt werden:

1. Schreibe alle Zahlen von 2 bis n auf.
2. Beginne mit der Zahl 2 und streiche alle Vielfachen heraus, die größer oder gleich 4 sind.
3. Fahre mit der nächstgrößeren nicht durchgestrichenen Zahl p fort und streiche wieder alle Vielfachen, die größer oder gleich p^2 sind, heraus.
4. Führe Schritt 3 solange durch, bis $p^2 \leq n$ ist.

Eine effiziente Implementation des Siebverfahrens kann demnach so aussehen:

```
void Sieb1(const Digit N)
// Algorithm:  Sieb1(N)
// Input:      N in Digit where N >= 2 ||
{
  const Digit n = N/2;
  char* primes = new char[n];
  if (!primes) { cerr << "Out of Memory!\n"; return; }

  Digit i;
  for (i = 0; i < n; ++i) primes[i] = 1;
  for (i = 0; i < n; ++i)
    if (primes[i]) {
      const Digit p = 2*i+3;
      for (Digit k = p+i; k < n; k += p) primes[k] = 0;
    }

  // Primzahlausgabe:
  cout << 2;
  for (i = 0; i < n; ++i)
    if (primes[i]) cout << ',' << 2*i+3;
  cout << endl;

  delete[] primes;
}
```

Hierbei sind nur die ungeraden Zahlen ab drei im Speicher abgelegt, weil die Eigenschaft ausgenutzt wird, daß alle geraden Zahlen bis auf die Primzahl zwei zerlegbar sind.
Noch effizienter arbeiten wir, wenn wir keine Elemente wiederholt aussieben, wie es aber im vorigen Algorithmus `Sieb1` geschieht. Dort werden nämlich alle Zahlen, die aus zwei verschiedenen Primzahlen zusammengesetzt sind, doppelt ausgesiebt.
Eine Implementation, die dies vermeidet und die auszusiebende Menge noch weiter verkleinert, stellte Xuedong Luo in [34] vor. Bei dieser sind alle ungeraden und nicht durch drei teilbaren Zahlen ab der Primzahl fünf im Speicher abgelegt:

```
void Sieb2(const Digit N)
// Algorithm:  Sieb2(N)
// Input:      N in Digit where N >= 3 ||
{
  const Digit n = N/3;
  char* primes = new char[n];
  if (!primes) { cerr << "Out of Memory!\n"; return; }

  Digit i;
  for (i = 0; i < n; ++i) primes[i] = 1;
  Digit c = GAMMA;
  Digit k = 1;
  Digit q = 2*(sqrt(N)/3);

  for (i = 3; i <= q; i += 2) {
    k += 3;
    Digit j = c += 2*k;
    k += 3;
    while (j < n) {
      primes[j] = 0; j += i;
      if (j >= n) break;
      primes[j] = 0; j += k;
    }

    i += 2;
    if (i > q) break;
    j = c += ++k; ++k;

    while (j < n) {
      primes[j] = 0; j += k;
      if (j >= n) break;
      primes[j] = 0; j += i;
    }
  }

  // Primzahlausgabe:
  cout << 2 << ',' << 3;
  k = 5;
  for (i = 0; i < n; ++i) {
    if (primes[i]) cout << ',' << k;
    k += 2;
    if (++i > n) break;
    if (primes[i]) cout << ',' << k;
    k += 4;
  }
  cout << endl;

  delete[] primes;
}
```

Um den Speicher nun optimal ausnutzen zu können, werden wir jede Zahl nicht mehr durch ein Byte, sondern nur noch durch ein Bit repräsentieren. Hierfür müssen wir kaum eine Veränderung durchführen und reduzieren unseren Speicherverbrauch auf $\frac{1}{\texttt{CHAR\_BIT}}$. Allerdings ist diese Änderung inkonsequent gegenüger unserer Forderung nach schnellstmöglicher Laufzeit, da dieser bitweise Zugriff mehr Zeit benötigt als der byteweise. Doch ist es in diesem Fall sinnvoll, den Speicherplatz auf Kosten der Geschwindigkeit zu minimieren, weil es sich bei der Primzahlzerlegung um eine endliche Aufgabe (`SieveSize`) handelt, die

aufgrund unserer anderen optimalen Vorgaben in genügend schneller Zeit bearbeitet werden kann.

⟨ generating prime numbers 286a ⟩ ≡

```
const Digit n = 8*SieveSize;
unsigned char* p = primes = NOTHROW_NEW unsigned char[SieveSize];
if (!primes) errmsg(2, "(constructor)");
firstPrime();
const unsigned char* pE = p+SieveSize;
do *p++ = (unsigned char)~0; while (p != pE);
p -= SieveSize;

Digit c = GAMMA;
Digit k = 1;
Digit q = 2*(sqrt(Digit(24*SieveSize))/3);

for (Digit i = 3; i <= q; i += 2) {
  k += 3;
  Digit j = c += 2*k;
  k += 3;

  while (j < n) {
    p[j/8] &= (unsigned char)~(1 << (j%8)); j += i;
    if (j >= n) break;
    p[j/8] &= (unsigned char)~(1 << (j%8)); j += k;
  }

  i += 2;
  if (i > q) break;
  j = c += ++k; ++k;

  while (j < n) {
    p[j/8] &= (unsigned char)~(1 << (j%8)); j += k;
    if (j >= n) break;
    p[j/8] &= (unsigned char)~(1 << (j%8)); j += i;
  }
}
```

◇

Macro referenced in 433.

Die Ausgabe aller ermittelten Primzahlen erfolgt dann über die beiden Funktionen

1. ⟨ getting first prime 286b ⟩ ≡

```
inline Digit Primes::firstPrime()
// Algorithm:  c := n.firstPrime()
// Input:      n in Primes.
// Output:     c in Digit such that c = 2 ||
{
  primPos = primes;
  idx = 1;
  return primNumber = 2;
}
```

◇

Macro referenced in 432b.

Laufzeit. Sei $n \in$ Primes, dann n`.firstPrime()`$\in \mathcal{O}(1)$.

2. ⟨ getting next prime 287a ⟩ ≡

```
bool Primes::nextPrime(Digit& a)
// Algorithm:  c := n.nextPrime(a)
// Input:      n in Primes.
// Output:     a in Digit, c in bool
//             such that if n.primNumber <= a.lastPrime()
//             then c = true else c = false ||
{
  if (primNumber == 2) { a = 3; primNumber = 5; }
  else {
    Digit p;
    do {
      if (primPos - primes == SieveSize) return false;
      p = *primPos & idx;
      if (p) a = primNumber;
      if (idx == 1 << (CHAR_BIT-1)) { idx = 1; ++primPos; primNumber += 4; }
      else {
        if (idx == 2 || idx == 8 || idx == 32) primNumber += 2;
        primNumber += 2;
        idx *= 2;
      }
    } while (p == 0);
  }
  return true;
}
◇
```

Macro referenced in 433.

Laufzeit. Sei $n \in$ Primes und $a \in$ Digit, dann n`.nextPrime(`a`)`$\in \mathcal{O}(1)$.

Die letzte Primzahl in unserem Speicher (beziehungsweise unserer endlichen Menge) läßt sich auch direkt berechnen:

⟨ getting last prime 287b ⟩ ≡

```
Digit Primes::lastPrime() const
// Algorithm:  c := n.lastPrime()
// Input:      n in Primes.
// Output:     c in Digit such that c is a prime ||
{
  static Digit lprim = 0;
  if (lprim) return lprim;
  Digit i = SieveSize;
  while (primes[--i] == 0);
  const Digit p = log2(Digit(primes[i]));
  i *= 24; i += 3*p + 5;
  return lprim = i - (p&1);
}
◇
```

Macro referenced in 433.

Laufzeit. Sei $n \in$ Primes, dann n`.lastPrime()` $\sim$ `log2(`γ`)`.

Anzahl der Primzahlen

Es müssen alle Primzahlen im Speicher gezählt werden, um eine genaue Aussage über die Anzahl der Primzahlen aus unserer endlichen Menge zu machen. Hierbei treffen wir die folgende Konvention:
Sei $n \in \mathsf{NumberBase}$, dann

$$n.\texttt{numberOfPrimes} : \mathsf{Digit} \to \mathsf{Digit} : a \mapsto \begin{cases} \pi(a)\,, & a \leq n.\texttt{lastPrime()} \\ 0\,, & a > n.\texttt{lastPrime()} \end{cases}$$

mit der **Primzahlfunktion**

$$\pi : \mathbb{N} \to \mathbb{N} : n \mapsto |\{p \in \mathbb{P} \mid p \leq n\}| = \sum_{\mathbb{P} \ni p \leq n} 1.$$

⟨ getting the number of primes 288 ⟩ ≡

```
size_t Primes::numberOfPrimes(Digit n) const
// Algorithm:  c := n.numberOfPrimes(a)
// Input:      n in Primes, a in Digit.
// Output:     c in Digit
//             such that if a <= n.lastPrime() then c = pi(a) else c = 0 ||
{
  if (n < 5) return size_t(n/2 + (n == 3));
  n -= 5;
  if (n >= 24*SieveSize) return 0;
  const unsigned char* q = primes + n/24;
  n %= 24;
  size_t i = 2;
  for (unsigned char j = char(1 << (n/3 + ((n%6) == 2))); j >= 1; j /= 2)
    if (*q & j) ++i;

  for (unsigned char* p = primes; p < q; ++p) {
    for (unsigned char j = 1; j; j *= 2)
      if (*p & j) ++i;
  }
  return i;
}
◇
```

Macro referenced in 433.

Laufzeit. Sei $n \in \mathsf{Primes}$ und $a \in \mathsf{Digit}$, dann

$$n.\texttt{NumberOfPrimes}(a) \in \begin{cases} \mathcal{O}(1)\,, & a < 5 \text{ oder } a \geq 24 \cdot \texttt{SieveSize} \\ \mathcal{O}(a)\,, & \text{sonst} \end{cases}.$$

Wenn wir nun $N = 10^7$ betrachten, dann erhalten wir die folgenden Laufzeiten auf einen Pentium 100 MHz mit dem Watcom Compiler 10.6:

Algorithmus	Laufzeit	Speicherverbrauch
Sieb1	2.47 s	4.77 MB
Sieb2	0.95 s	3.18 MB
Sieb3	2.63 s	407 kB

5.7.2 Primzahltest von Brillhart und Selfridge

Wir werden in diesem Abschnitt Methoden vorstellen, die entscheiden können, ob eine gegebene natürliche Zahl eine Primzahl ist.

Kleiner Satz von Fermat

Zur Überprüfung der Primzahleigenschaft spielt die folgende Beziehung eine wichtige Rolle:

$$\text{Ist } p \in \mathbb{P}, a \in \mathbb{N} \text{ und } \gcd(a,p) = 1, \text{so gilt } a^{p-1} \equiv 1 \pmod{p}.$$

Dadurch haben wir einen einfachen Primzahltest. Ist nämlich zum Beispiel $2^{n-1} \not\equiv 1 \pmod{n}$, so wissen wir wegen der Kontraposition des kleinen Satzes von Fermat, ohne die Zerlegung von n zu kennen, daß n zusammengesetzt ist.
Die Implementation dieses Satzes beruht auf dem Potenzieralgorithmus aus Abschnitt 5.4.1 und läßt sich durch eine Erweiterung auf sogenannte Starke Pseudoprimzahlen von J. L. Selfridge erstmals eingeführt

$$\{a \in (\mathbb{Z}/n\mathbb{Z})^* \mid b = \nu_2(n-1), a^{(n-1)\cdot 2^{-b}} = 1 \text{ or } \exists_{0 \le c < b} : a^{(n-1)\cdot 2^{c-b}} = -1\}$$

abbilden.

⟨ strong pseudoprime test 289a ⟩ ≡

```
template<class T>
bool spsp(const T& base, const T& n)
// Algorithm:  c := spsp(b, n)
// Input:      b,n in T where n == 1 (mod 2).
// Output:     c in bool
//             such that if b^((n-1)/2^k) == 1 (mod n) or b^((n-1)/2^l) == -1 (mod n)
//             for 0 <= l < k, where 2^k|n-1 and not 2^(k+1)|n-1,
//             then c = true else c = false ||
{
  const T m = n-1;
  T l = m;
  int k = 0;
  do { ++k; l >>= 1; } while ((l&1) == 0);
  l = pow(base, l, n);
  if (l != 1 && l != m) {
    int j = k-1;
    if (j == 0) return false;
    do {
      l *= l; l %= n;
      if (l == 1) return false;
    } while (l != m && --j);
    if (j == 0) return false;
  }
  return true;
}
```

◇

Macro referenced in 432b.

Laufzeit. Seien $x, y \in T$, dann `spsp(`x`,`y`)` $\sim$ `pow(`x`,`y`,`y`)`.

Miller-Rabin Test

Der Miller-Rabin Test überprüft, ob die gegebene natürliche Zahl n für mehrere Primzahlbasen eine starke Pseudo-Primzahl ist [37].

⟨ Miller-Rabin test 289b ⟩ ≡

```
bool MillerRabin(unsigned int i, const Natural& n)
// Algorithm:  c := MillerRabin(i, n)
// Input:      i in unsigned int, n in Natural where n == 1 (mod 2).
// Output:     c in bool
//             such that if for all primes b less than or equal the ith prime
//             b^((n-1)/2^k) == 1 (mod n) or b^((n-1)/2^l) == -1 (mod n)
//             for 0 <= l < k, where 2^k|n-1 and not 2^(k+1)|n-1,
//             then c = true else c = false ||
{
  Primes p;
  Digit a = p.firstPrime();
  Natural b;
  do {
    b = a;
    if (!spsp(b, n)) return false;
  } while (p.nextPrime(a) && --i);
  return true;
}
◇
```

Macro referenced in 433.

Primzahltest

Anhand des kleinen Satzes von Fermat kann nur überprüft werden, ob eine gegebene natürliche Zahl zusammengesetzt ist, nicht jedoch, ob es sich dabei um eine Primzahl handelt. Denn aus $a^{p-1} \equiv 1 \pmod{p}$ folgt nicht zwingend $p \in \mathbb{P}$, wie dieses Beispiel zeigt:
Es gilt zwar $4^{15-1} \equiv 1 \pmod{15}$, aber $15 = 3 \cdot 5$ ist zerlegbar und damit keine Primzahl.

Ein allgemeingültiger Primzahltest für eine gegebene Zahl $n \in \mathbb{N}$ stammt von J. Brillhart und J. L. Selfridge ([9]), bei dem die Faktorisierung von $n-1$ vorausgesetzt wird.
Sei $n \geq 3$ ungerade und

$$n - 1 = \prod_{i=0}^{t} p_i^{a_i} \qquad \text{für } t \in \mathbb{N}, a_i \in \mathbb{N}_+, 0 \leq i \leq t$$

die Primfaktorzerlegung der Zahl $n-1$, so gilt die folgende Aussage:
Falls für jeden Primteiler p_i von $n-1$ ein $x \in \mathbb{N}$ existiert mit den Eigenschaften

$$\gcd(x, n) = 1, \quad x^{n-1} \equiv 1 \pmod{n}, \quad x^{(n-1)/p_i} \not\equiv 1 \pmod{n} \qquad \text{für } 1 \leq i \leq t,$$

dann ist n eine Primzahl.

Für kleinere Zahlen n hat G. Jaeschke in [25] explizite Basen ermittelt, um einen optimalen Primzahltest zu erhalten.

Es gibt auch eine deterministische Variante von M. R. Fellows und N. Koblitz [16], als auch eine deterministische Variante, die nur eine partielle Faktorisierung von $n-1$ benötigt (siehe [28]). Diese Variante werden wir in einer späteren Arbeit implementieren.

⟨ primality test 290 ⟩ ≡

```
bool isprime(const Natural& n)
// Algorithm:  c := isprime(a)
// Input:      a in Natural.
// Output:     c in bool such that
//             if a is a prime element then c = true else c = false ||
{
  static const bool prim[32] = { 0, 0, 1, 1, 0, 1, 0, 1, 0, 0,
                                 0, 1, 0, 1, 0, 0, 0, 1, 0, 1,
                                 0, 0, 0, 1, 0, 0, 0, 0, 0, 1,
                                 0, 1 };
  static const bool mod[30] = { 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1,
                                1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0 };
  const size_t sT = n.length();
  const Digit z = n.highest();
  if (sT == 1 && z < 32) return prim[z];
  if (mod[n%30]) return false;
  Primes primes;
  if (sT == 1) {
    const Digit y = primes.lastPrime();
    if (z == y) return true;
    else if (z < y) {
      Digit q = primes.firstPrime();
      while (primes.nextPrime(q) && q < z);
      if (q == z) return true;
      else if (q > z) return false;
    }
  }

  static const Natural y0("4DQKT65", 32);
  static const Natural y1("T3AAA4OO", 32);
  static const Natural y2("1UKFJVRHR", 32);
  static const Natural y3("3543JG76V", 32);
  static const Natural y4("9MKD9B5I61", 32);

  if (n < Digit(9080191)) return (spsp(Natural(31), n) && spsp(Natural(73), n));
  else if (n < y0) return (spsp(Natural(2), n) && spsp(Natural(7), n)
                           && spsp(Natural(61), n));
  else if (n < y1) return (spsp(Natural(2), n) && spsp(Natural(13), n)
                           && spsp(Natural(23), n) && spsp(Natural(1662803), n));
  else if (n < y2) return MillerRabin(5, n);
  else if (n < y3) return MillerRabin(6, n);
  else if (n < y4) return MillerRabin(7, n);

  list<Natural> p;
  Natural b = n-1;
  factoring(b, p);
  Digit k = primes.firstPrime();
  while (true) {
    if (!spsp(Natural(k), n)) return false;
    list<Natural>::iterator i = p.begin();
    while (true) {
      const Natural c = b / *i;
      const Digit k2 = k;
      while (pow(Natural(k), c, n) == 1) {
        if (!primes.nextPrime(k)) k += 2;
      }
```

```
          if (k != k2) break;

          list<Natural>::const_iterator j = i;
          do
            if (++i == p.end()) return true;
          while (*j == *i);
        }
      }
    }
    ◇
```

Macro referenced in 433.

Laufzeit. Sei $x \in$ Natural und C ein Container über Natural, dann

$$x\texttt{.isprime()} \sim \texttt{factoring(}x-1\texttt{,}C\texttt{)}.$$

5.7.3 Primfaktorzerlegung

Der einfachste Algorithmus zur Primfaktorzerlegung von $n \in \mathbb{N}$ ist das sukzessive Dividieren durch alle Primzahlen. Hierbei werden alle möglichen Faktoren von zwei bis $\sqrt{n}$ durchprobiert, wozu eine Laufzeit von $\mathcal{O}(\sqrt{n})$ benötigt wird.

Verbesserungsvorschlag

Sei $\mathbb{N} \ni p \leq \sqrt{n}$, so betrachten wir beim sukzessiven Dividieren den Divisionsrest $r := n \bmod p$. Dadurch ist n genau dann durch p teilbar, wenn $r = 0$ ist. Bevor wir aber zur nächsten größeren Zahl $\mathbb{N} \ni q > p$ wechseln, um mit der Iteration des sukzessiven Dividierens fortzufahren, wollen wir uns mit der folgenden Frage beschäftigen:

$$\text{Sei } n \in \mathbb{N} \text{ und } \mathbb{N} \ni p < \sqrt{n}. \text{ Für welche } x \in \mathbb{N}_+ \text{ gilt } n \bmod (p) \leq n \bmod (p+x)?$$

Damit wollen wir ein Intervall $I := [p, p+x]$ bestimmen, in dem der Divisionsrest auf jeden Fall ungleich Null ist und folglich n nicht durch p teilbar sein kann.

Es gilt $0 \leq a - b\lfloor \frac{a}{b} \rfloor \leq a - b\lfloor \frac{a}{b+x} \rfloor < b + x$.
Gesucht. $a - b\lfloor \frac{a}{b} \rfloor \leq a - (b+x)\lfloor \frac{a}{b+x} \rfloor$
Eine mögliche Bedingung: $\lfloor \frac{a}{b+x} \rfloor = \lfloor \frac{a-b}{b+x} \rfloor$.

Beispiel. Sei $n = 9887$ und $p = 2$, so erhalten wir $1 \leq x \leq 4941$ und wir müssen die nächste Division nicht mit $q = 3$ durchführen, sondern erst mit der Primzahl $q = p + 4941 = 4943$ fortfahren. Wir sind mit dieser einen Division jedoch bereits fertig, weil wir nur bis $\lfloor \sqrt{n} \rfloor = 99$ nach Teilern suchen müssen und mit 4943 diese Grenze schon weit überschritten haben. Daher können wir folgern, daß n eine Primzahl ist.

Beispiel. Angenommen, wir haben alle Primzahlen bis 23 gegeben und wir wollen $n = 998$ faktorisieren, dann benötigen wir mit dem sukzessiven Dividieren 2 Quadratwurzel- und 9 Divisionsoperationen. Dagegen benötigen wir nur 2 Quadratwurzel- und 2 Divisionsoperationen, wenn wir zusätzlich den Divisionsrest betrachten, weil bereits aus $499 \bmod (2) \leq 499 \bmod (2+x)$ für $1 \leq x \leq 20$ die Primzahleigenschaft von 499 folgt.

Offensichtlich gilt die folgende Gleichung:
Sei $x, y, z \in \mathbb{N}, y > x, \gcd(z, x) = 1$,
so erfüllt $y \in x + \{1, \ldots, z - l - 1\}$ mit $x \equiv l \pmod{z}$ die Gleichung $\lfloor \frac{x}{z} \rfloor = \lfloor \frac{y}{z} \rfloor$.
Das ist der einfache Fall, uns interessiert aber eine Auflösung nach z.

5.7.4 Faktorisierung durch sukzessives Dividieren.

Bereits beim Siebalgorithmus (siehe Kapitel 5.7.1) ist aufgefallen, daß es keine handliche Formel zur Erzeugung beliebig vieler Primzahlen gibt. Deswegen müssen wir eine Teilmenge der natürlichen Zahlen konstruieren, die alle Primzahlen enthält, aber möglichst klein ist, um die Anzahl der durchzuführenden Divisionen für die Faktorisierung gering zu halten.
Die triviale Menge hierzu wäre natürlich $\mathbb{N}$. Eine weitaus bessere Menge aber – mit der Eigenschaft, daß alle Primzahlen außer 2 ungerade sind – ist

$$\{p \in \mathbb{N} \mid \ p \equiv 1 \pmod 2\} \dot\cup \{2\},$$

denn hier sind alle Vielfachen von zwei ausgesiebt, womit die Anzahl der durchzuführenden Divisionen um 50% reduziert ist. Werden nun zusätzlich auch alle Vielfachen von 3 aussortiert, so erhalten wir die Menge

$$\{p \in \mathbb{N} \mid \ p \equiv \pm 1 \pmod 6\} \dot\cup \{2, 3\},$$

und es bleiben $(1 - \frac{1}{2})(1 - \frac{1}{3}) \cdot 100 = 33\%$ der ursprüglichen Divisionen übrig.
Wenn wir zusätzlich noch die Vielfachen von 5 herausnehmen, dann erhalten wir die Menge

$$\mathbb{P}' := \{p \in \mathbb{N} \mid \ p \equiv q \pmod{30}, q \in \{1, 7, 11, 13, 17, 19, 23, 29\}\} \dot\cup \{2, 3, 5\}$$

und wir gelangen auf 26,7% der ursprünglichen Divisionsanzahl. Diese Vorgehensweise könnten wir noch weiter fortsetzen, doch wird der Prozentsatz der insgesamt eingesparten Divisionen immer geringer, und es muß auch bedacht werden, daß unsere konstruierte Menge umso unhandlicher und komplexer wird, je mehr Elemente wir aussieben. Schließlich versagt dieses Verfahren für sehr große Zahlen, weil trotz der Einsparungen alle Primzahlvorgänger bis $\lfloor\sqrt{n}\rfloor$ durchprobiert werden müssen.

Vorgehensweise

Sei $a \in$ Natural die zu faktorisierende Zahl, so testen wir zuerst alle Primzahlen unserer endlichen Menge `primes`, die kleiner oder gleich $\lfloor\sqrt{a}\rfloor$ sind:

⟨ successive division for less factors 293 ⟩ ≡

```
if (a == 0) return;
while (a.even()) {                        // less factors
  p.push_back(2);
  a >>= 1;
}
if (a == 1) return;
Natural t;
Digit i;
Primes primes;
Digit prim = primes.firstPrime();            // didn't need 2
while (primes.nextPrime(prim)) {
  while (true) {
    div(a, prim, t, i);
    if (i) break;
    p.push_back(prim);
    if (t == 1) return;
    swap(a, t);
  }
  if (t < prim) { p.push_back(a); return; }
}
```

◇

Macro referenced in 296.

Falls alle Primzahlen durchlaufen sind und die Faktorisierung noch nicht abgeschlossen ist, setzen wir mit der größeren Menge $\mathbb{P}'$ das sukzessive Dividieren bis $\lfloor\sqrt{a}\rfloor$ fort:

⟨ successive division for greater factors 294 ⟩ ≡

```
Natural s = prim;
const Digit n[8] = { 4, 2, 4, 2, 4, 6, 2, 6 };
i = 0;
switch (s % 30) {
  case 1:  ++i;
  case 29: ++i;
  case 23: ++i;
  case 19: ++i;
  case 17: ++i;
  case 13: ++i;
  case 11: ++i;
}
Natural q,r;
t = root(a, 3);
for (s += n[i]; s.length() == 1 && s <= t; s += n[i&7]) {
  if (a%s.highest() == 0) {
    p.push_back(s);
    a /= s.highest();
    if (isprime(a)) { p.push_back(a); return; }
    t = root(a, 3);
  }
  ++i;
}
while (s <= t) {
  div(a, s, q, r);
  if (r == 0) {
    p.push_back(s);
    if (isprime(q)) { p.push_back(q); return; }
    swap(a, q);
    t = root(a, 3);
  }
  ++i;
  s += n[i&7];
}
```

◇

Macro referenced in 296.

Danach wissen wir, daß a ein Produkt höchstens zweier Primzahlen ist. Deshalb wird ab jetzt unser Interesse darin bestehen, eine gegebene natürliche Zahl in zwei Faktoren (falls überhaupt möglich) zu zerlegen.

5.7.5 Faktorisierungsalgorithmus von Fermat

Ist n eine ungerade natürliche Zahl und Differenz zweier Quadratzahlen, so läßt sie sich zerlegen in

$$n = pq = \left(\frac{p+q}{2}\right)^2 - \left(\frac{p-q}{2}\right)^2 = x^2 - y^2 = (x+y)(x-y).$$

Ausgehend von $\lfloor\sqrt{n}\rfloor$, suchen wir eine iterative Lösung für $r = x^2 - y^2 - n$. Falls $r = 0$ ist, haben wir unsere Zerlegung gefunden, ansonsten wird x, solange $r > 0$ ist, schrittweise verkleinert oder y für $r < 0$ dementsprechend vergrößert.

```
void fermat(const Natural& n, Natural& a, Natural& b)
// Algorithm:  fermat(n; a, b)
// Input:      n in Natural where n is odd.
// Output:     a,b in Natural such that ab = n ||
{
  Natural r = sqrt(n);
  Natural x = 2*r+1;
  Natural y = 1;
  r *= r;
  while (r < n) { r += x; r -= y; x += 2; y += 2; }
  r -= n;
  while (r != 0) {
    while (r >= y) { r -= y; y += 2; }
    if (r != 0)
      do { r += x; r -= y; x += 2; y += 2; } while (r < y);
  }
  a = (x-y)/2;
  b = (x+y-2)/2;
}
```

Dieser Algorithmus zeichnet sich besonders dadurch aus, daß er lediglich mit Additionen und Subtraktionen auskommt. Noch geschickter ist es aber, nur alle Quadratzahlen von x zu durchlaufen und jedesmal zu überprüfen, ob $x - n$ eine Quadratzahl ist. Die nächste Quadratzahl läßt sich dann einfach durch Addieren ermitteln: $(x+1)^2 = x^2 + 2x + 1$. Zusätzlich betrachten wir $y \equiv z \pmod{16}$ und wissen, daß y keine Quadratzahl ist, falls $z \notin \{0, 1, 4, 9\}$ ist.
Dennoch kann dieser Ansatz in der Praxis schlechter als das sukzessive Dividieren sein, das die Ordnung $\mathcal{O}(n^{\frac{1}{2}})$ hat, nämlich wenn zum Beispiel $p \approx n^{\frac{1}{3}}$ und $q \approx n^{\frac{2}{3}}$ ist. Dann gilt nämlich

$$\frac{(\sqrt{n} - \sqrt[3]{n})^2}{2\sqrt[3]{n}} = \frac{n^{\frac{2}{3}}(\sqrt[6]{n} - 1)^2}{2\sqrt[3]{n}} \in \mathcal{O}(n^{\frac{2}{3}}).$$

5.7.6 Faktorisierungsalgorithmus von Lehman

Der vorige Ansatz von Fermat kann verwendet werden, um eine Laufzeit von $\mathcal{O}(n^{\frac{1}{3}})$ zu erzielen. Dies demonstrierte R. S. Lehman 1974 in [29]:
Zunächst zerlegt man die zu faktorisierende Zahl $n \in \mathbb{N}$ durch sukzessives Dividieren in

$$m = \frac{n}{\prod_{i=0}^{t} p_i} \qquad \text{mit allen Primteilern } p_i \leq \sqrt[3]{n} \text{ von } n \text{ für } 0 \leq i \leq t.$$

Desweiteren prüft man, ob sich $m = pq$ zerlegen läßt mit

$$m^{\frac{1}{3}} < p \leq q < m^{\frac{2}{3}},$$

indem geprüft wird, ob $m = p^2 = q^2$ ist oder ob $(\lfloor\sqrt{4km}\rfloor + d)^2 - 4km$ eine Quadratzahl für $1 \leq k \leq \lfloor\sqrt[3]{m}\rfloor, 1 \leq d \leq \left\lfloor \frac{\sqrt[6]{m}}{4\sqrt{k}} \right\rfloor + 1$ ist. Falls ein solches Paar (k, d) nicht existiert, so ist m eine Primzahl. Andernfalls setzt man

$$a := \lfloor\sqrt{4km}\rfloor + d, b := \sqrt{a^2 - 4km},$$

und somit gilt $m = \gcd(a+b, m) \cdot \frac{m}{\gcd(a+b,m)} = \gcd(a+b, m) \cdot \gcd(a-b, m)$.

Laufzeitbeispiel. Wenn wir die Mersennezahl $2^{67} - 1 = 193707721 \cdot 761838257287$ faktorisieren wollen, dann benötigt das sukzessive Divisionsverfahren 126 Sekunden mit dem GNU-Compiler auf einem

Pentium 100 MHz (639 Sekunden für die volle ANSI-Version), während unter denselben Bedingungen der Algorithmus von R. S. Lehman mit lediglich 40 Sekunden (52 Sekunden für die volle ANSI-Version) auskommt. Dies liegt aber hauptsächlich daran, weil wir für ihn mit unserem Wurzelzieh-Algorithmus aus Kapitel 2.15 eine gute Grundlage gesetzt haben und schon vorher berechnete Ergebnisse wiederverwenden, wenn es lediglich um die Berechnung des Nachfolgers geht.
So gilt zum Beispiel für ein $a \in \mathbb{N}$ die Gleichung $a = \lfloor\sqrt{a^2+k}\rfloor$ für $0 \leq k \leq 2a$, da erst mit $\lfloor\sqrt{a^2+2a+1}\rfloor = \lfloor\sqrt{(a+1)^2}\rfloor = a+1$ der Nachfolger von a erreicht wird.
Das Ganze führt nun zur folgenden Implementation:

⟨ prime factorization of a Natural 296 ⟩ ≡

```
template <class Container>
void factoring(Natural a, Container& p)
// Algorithm:  factoring(a, c)
// Input:      a in Natural.
// Output:     c is a container over Natural with output-iterator
//             such that if a >= 1 then a = prod_{c.begin() <= i < c.end()} *i
//             else c.begin() = c.end(),
//             for all i in [c.begin(), c.end()[, j in [i, c.end()[ : *i <= *j ||
{
  p.erase(p.begin(), p.end());
  ⟨ successive division for less factors 293 ⟩
  if (isprime(a)) { p.push_back(a); return; }   // greater
  ⟨ successive division for greater factors 294 ⟩
  Natural w;                                          // large factors
  sqrt(a, s, w);
  if (w == 0) { p.push_back(s); p.push_back(s); return; }
  s = root(a, 6);
  q = a << 2;
  t *= a; t <<= 2;

  Natural x,y,z,d = 4;
  Natural e = 2;
  r = s >> 2;
  const char c[32] = { 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
                       1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0 };
  Natural k = q;
  for (; k <= t; k += q) {
    if (e == 0) {
      d += 4;
      if (d > s) break;
      r = d >> 1;
      div(s, d, r, x);
    } else --e;
    sqrt(k, x, y);
    z = x << 1; ++x; ++z; z -= y;
    while (true) {
      if (c[z.lowest() & 31]) {
        sqrt(z, y, w);
        if (w == 0) {
          y = gcd(x+y, a);
          a /= y;
          if (y < a) { p.push_back(y); p.push_back(a); }
          else { p.push_back(a); p.push_back(y); }
          return;
        }
      }
```

```
      if (r == 0) break;
      --r; z = x << 1; ++x; ++z;
    }
  }
  while (k <= t) {
    sqrt(k, x, z);
    y = x << 1; ++x; ++y; y -= z;
    if (c[y.lowest() & 31]) {
      sqrt(y, y, w);
      if (w == 0) {
        y = gcd(x+y, a);
        a /= y;
        if (y < a) { p.push_back(y); p.push_back(a); }
        else { p.push_back(a); p.push_back(y); }
        return;
      }
    }
    k += q;
  }
  p.push_back(a);
}
```
◇

Macro referenced in 432b.

Laufzeit. Sei $x \in$ Natural und C ein Container über Natural, dann `factoring(`x`, `C`)`$\in \mathcal{O}(x^{\frac{1}{3}})$.

5.7.7 Faktorisierung nach Gauß/Legendre

Trotz dieses guten Programms können dennoch bereits dreißigstellige Zahlen (wie zum Beispiel $2^{101}-1 = 7432339208719 \cdot 341117531003194129$) nur mit viel Rechenaufwand zerlegt werden.
Sehr gute und ausführliche Erklärungen zu den folgenden drei Methoden (Gauß/Legendre, Dixon, Morrision und Brillhart) zur Lösung dieses Faktorisierungsproblems können genauer im Buch von H. Riesel [41] nachgelesen werden. Wir werden hier hauptsächlich die Implementation und den praktischen Nutzen betrachten.

Die eigentliche Idee von Gauß besteht darin, quadratische Reste Modulo der zu faktorisierenden Zahl $a \in \mathbb{N}$ zu finden. Dabei gilt folgendes:

$$y \equiv x^2 \pmod{a} \quad \Longrightarrow \quad y \equiv x^2 \pmod{p} \qquad \text{für alle Teiler } p \text{ von } a.$$

Wenn wir nun einen quadratischen Rest mod a haben, so können wir mit Hilfe der Ergänzungssätze (siehe Kapitel 5.5) die Anzahl der für das sukzessive Dividieren notwendigen Divisoren halbieren.

Beispiel. Sei $a \in \mathbb{N}$ die zu faktorisierende Zahl, $D := \{p \in \mathbb{N} \mid \ p \equiv \pm 1 \pmod{6}\} \dot\cup \{2,3\}$ unsere Menge aller Divisoren und 5 ein quadratischer Rest mod a. So ist $\mathbb{Z}/5\mathbb{Z} \supseteq Q = \{1^2, 2^2\} = \{1,4\}$ die Menge aller quadratischen Reste mod 5 außer Null. Die Menge Q kann dabei mit dem Programm `root` (siehe Seite 280) ermittelt werden. Wenden wir dann noch den Chinesischen Restsatz `chinese` (siehe Seite 278) an, so erhalten wir die Menge

$$D' = \{p \in \mathbb{N} \mid \ p \equiv q \pmod{30},\ q \in \{1, 11, 19, 29\}\} \dot\cup \{2,3\}.$$

Bis heute ist es noch nicht gelungen, kleine quadratische Reste mod a explizit angeben zu können. C. F. Gauß wendete deswegen den folgenden Trick an:

$$y \equiv x^2 \pmod{a} \quad \Longleftrightarrow \quad x^2 - y = ka \qquad \text{für ein } k \in \mathbb{N}.$$

Wählen wir nun x nahe bei $\lfloor\sqrt{ka}\rfloor$, so läßt sich y leichter faktorisieren, denn wir brauchen nur alle Exponenten unserer Primteiler von y Modulo 2 zu betrachten, weil das Produkt zweier quadratischer Reste oder zweier Nichtreste mod a ein quadratischer Rest mod a ist.

Beispiel. Sei $a = 11111$, so ist 5 ein quadratischer Rest mod a, weil

$$\lfloor\sqrt{11111} + 1\rfloor - 5^3 = 106^2 - 5^3 = 11111.$$

Bemerkung. A. M. Legendre hat zur Auffindung der quadratischen Reste mod a die Kettenbruchentwicklung von $\sqrt{a}$ vorgeschlagen (siehe H. Riesel [41]).

5.7.8 Faktorisierungalgorithmus von Dixon

Es gilt die folgende Beziehung: Sei $a \in \mathbb{N}$ eine ungerade Zahl, die wir faktorisieren wollen, so ist der Hauptgedanke in J. D. Dixons Faktorisierungalgorithmus ([14]), zwei ganze Zahlen x und y mit $\gcd(x, a) = \gcd(y, a) = 1$ zu finden, wobei $x^2 \equiv y^2 \pmod{a}$ und $x \not\equiv y \pmod{a}$ ist. Daraus folgt, daß $x^2 - y^2 = (x - y)(x + y) \equiv 0 \pmod{a}$ und $x + y \not\equiv 0 \pmod{a}$ gilt. Somit ist $\gcd(x + y, a)$ ein nichttrivialer Teiler von a.

5.7.9 Faktorisierungalgorithmus von Morrision und Brillhart

Die Grundgedanken von C. F. Gauß, A. M. Legendre und J. D. Dixon wurden 1970 mit großem Erfolg zur Faktorisierung der siebten Fermatzahl

$$F_7 = 2^{128} + 1 = 59649589127497217 \cdot 5704689200685129054721$$

eingesetzt. Detaillierte Informationen und genauere Arbeiten findet man in der Originalarbeit von M. A. Morrison und J. Brillhart in [38]. Wir werden hier ein paar Punkte herausnehmen und realisieren. Sei $a \in \mathbb{N}$ eine ungerade Zahl, die wir faktorisieren wollen und $k \in \mathbb{N}_+$. Bevor wir nun die Kettenbruchentwicklung von $\sqrt{ka}$ durchführen, betrachten wir eine endliche Menge von Primzahlen $P(k, a, m) = \{p_i \in \mathbb{P} \mid 1 \le i \le m, (ka)^{\frac{p_i-1}{2}} \equiv x \pmod{p_i}, x \in \{0, 1\}\} \cup \{2\}$ mit $m \in \mathbb{N}$, die das Euler-Kriterium erfüllen:

"cfrac.cpp" 298 ≡

```
#include <string.h>
#include <time.h>
#include "natural.h"

⟨ new operator 396e ⟩

Natural cfrac(const Natural& a, const Digit m, const Digit c)
// Algorithm:  q := cfrac(a, b, c)
// Input:      a in Natural, b,c in Digit where b >= 50, c >= 1.
// Output:     b in Natural such that q | a ||
{
  struct prim_t {
    Natural* x;
    char*    c;
  };
  Natural s,s2,s3,r,v,y;
```

```
  Natural t = 1;
  const Digit pm = 3;
  Digit* p = NOTHROW_NEW Digit[m];
  if (!p) a.errmsg(2, "(cfrac)");
  p[0] = p[1] = 0;

  prim_t* d = NOTHROW_NEW prim_t[m+1];
  if (!d) a.errmsg(2, "(cfrac)");
  Digit i;
  for (i = 0; i <= m; ++i) {
    d[i].x = NOTHROW_NEW Natural[3];
    if (!d[i].x) a.errmsg(2, "(cfrac)");
    d[i].c = NOTHROW_NEW char[m];
    if (!d[i].c) a.errmsg(2, "(cfrac)");
    memset(d[i].c, 0, m);
  }
  for (Natural b = c*a; t == 1; b += a) {
    sqrt(b, r, v);
    Natural w = r;
    Natural z = 1;
    Natural u = 1;
    r <<= 1;
    Natural x = r;
    Digit prim = a.firstPrime();
    Digit n = 2;
    while (a.nextPrime(prim) && n < m) {
      Digit q = b % prim;
      Digit q2 = 1;
      for (i = prim/2; i > 1; i /= 2) {
        if (i&1) { q2 *= q; q2 %= prim; }
        q *= q; q %= prim;
      }
      q2 *= q; q2 %= prim;
      if (q2 <= 1) p[n++] = prim;
    }
```

◇

File defined by 298, 299, 300, 301, 302.

Nun suchen wir diese Zahlen v in unserer Kettenbruchentwicklung, deren Primteiler alle aus P sind. Um nicht m Divisionen durchführen zu müssen, setzen wir

$$T := \prod_{p \in P} p \text{ und betrachten } T \equiv s \pmod{v}.$$

Ist nun $s = 0$, so haben wir ein gesuchtes v gefunden, ansonsten quadrieren wir s und betrachten $s^2 \equiv s_2 \pmod{v}$. Ist nun $s_2 = 0$, so läßt sich v in Primfaktoren aus P mit maximal quadratischem Exponenten zerlegen. Je nach Größenordnung unserer gegebenen Zahl a können wir auf diese Weise fortfahren. Es ist jedoch sinnvoller, die Zerlegung in zwei Bereiche aufzuteilen:

1. Kleine Primzahlen können einen beliebig hohen Exponenten haben.

2. Größere Primzahlen dürfen einen Exponeten haben, der höchstens gleich vier ist.

"cfrac.cpp" 299 ≡

```
        Natural trial = 1;
        for (i = 2; i < pm; ++i);
        while (i < n) trial *= p[i++];

        Digit nd = 0;
        Digit iteration = 0;
        while (true) {
          ++iteration;
          t = v;
          if (t == 1) break;
          while ((t&3) == 0) t >>= 2;
          d[nd].c[1] = char(t.even());
          if (d[nd].c[1]) t >>= 1;
          for (i = 2; i < pm && t > 1; ++i) {
            const Digit k = p[i]*p[i];
            const Digit l = k*p[i];
            Digit m;
            while (true) {
              div(t, k*k, s, m);
              if (m) break;
              swap(t, s);
            }
            if (m%l == 0) {
              d[nd].c[i] = 1;
              t /= l;
            } else if (m%k == 0) {
              d[nd].c[i] = 0;
              t /= k;
            } else if (m%p[i] == 0) {
              d[nd].c[i] = 1;
              t /= p[i];
            } else d[nd].c[i] = 0;
          }
          if (t > 1) {
            div(trial, t, s, s2);
            if (s2 != 0) {
              s2 *= s2; s2 %= t;
              if (s2 != 0) { s2 *= s2; s2 %= t; }
            }
          }
```

◇

File defined by 298, 299, 300, 301, 302.

Falls wir nun v komplett durch Primzahlen aus P faktorisieren können, halten wir die vorkommenden Exponenten Modulo 2 im Vektor c der Struktur `prim_t` fest, wobei hier c_0 für -1, c_1 für 2 und so weiter stehen. Unser Vektor d enthält dadurch die einzelnen Vektoren c, mit denen er eine Matrix über $(\mathbb{Z}/2\mathbb{Z})^m \times (\mathbb{Z}/2\mathbb{Z})^m$ bildet, sowie die faktorisierten Zahlen v und die Quadratwurzel w von v mod a:

"cfrac.cpp" 300 ≡

```
          if (t == 1 || s2 == 0) {
            if (t > 1) {
              s = t;
              Digit j,k;
              do {
                k = p[i]*p[i];
```

```
        div(s, k, t, j);
        if (j == 0) {
          swap(s, t);
          --i;
        } else if (j%p[i] == 0) {
          d[nd].c[i] = 1;
          s /= p[i];
        }
        ++i;
      } while (s > k);
      if (s != 1) {
        while (s > p[i]) ++i;
        d[nd].c[i] = 1;
      }
    }
    d[nd].c[0] = char(iteration&1);
    d[nd].x[0] = w; d[nd].x[1] = v; d[nd].x[2] = 1;
```

◇

File defined by 298, 299, 300, 301, 302.

Nun fügen wir unseren Vektor c in unsere Matrix ein. Dabei beachten wir, daß die erste 1 in unserem Vektor c nur einmal in der Matrix vorkommen darf. Dadurch erzeugen wir durch Linearkombinationen sukzessive eine Diagonalform. Bei den Linearkombinationen können wir unsere v-Einträge (quadratische Reste mod a) einfach modular miteinander multiplizieren, während unsere w-Einträge (quadratische Wurzeln mod a) weiterhin regulär miteinander multipliziert werden müssen. Diese Zahlen würde rasch große Dimensionen annehmen und somit die Laufzeit stark hemmen, wenn man nicht zuvor den größten gemeinsamen Teiler herausnähme, der auf jeden Fall quadratisch vorkommt:

"cfrac.cpp" 301 ≡

```
    Digit j;
    for (j = 0; j < n && d[nd].c[j] == 0; ++j);
    Digit k = 0;
    for (i = 0; i < nd && j < n; ++i) {
      while (d[i].c[k] == 0) ++k;
      if (k == j) {
      d[nd].c[k] = 0;
      for (Digit l = ++k; l < n; ++l) d[nd].c[l] ^= d[i].c[l];
        d[nd].x[0] *= d[i].x[0];
        d[nd].x[0] %= a;
        t = gcd(d[nd].x[1], d[i].x[1]);
        d[nd].x[1] *= d[i].x[1];
        d[nd].x[1] /= t*t;
        d[nd].x[2] *= t;
        d[nd].x[2] *= d[i].x[2];
        d[nd].x[2] %= a;
        while (j < n && d[nd].c[j] == 0) ++j;
      } else if (k > j) {
        prim_t tP = d[nd];
        Digit l;
        for (l = nd; l > i; --l) d[l] = d[l-1];
        d[l] = tP;
        ++i;
        while (true) {
          if (d[l].c[k]) {
            d[l].c[k] = 0;
```

```
          for (Digit l2 = k+1; l2 < n; ++l2) d[l].c[l2] ^= d[i].c[l2];
          d[l].x[0] *= d[i].x[0];
          d[l].x[0] %= a;
          t = gcd(d[l].x[1], d[i].x[1]);
          d[l].x[1] *= d[i].x[1];
          d[l].x[1] /= t*t;
          d[l].x[2] *= t;
          d[l].x[2] *= d[i].x[2];
          d[l].x[2] %= a;
        }
        if (++i > nd) break;
        while (d[i].c[++k] == 0);
      }
      for (i = 0; i < l; ++i)
        if (d[i].c[j]) {
        d[i].c[j] = 0;
        for (Digit l2 = j+1; l2 < n; ++l2) d[i].c[l2] ^= d[l].c[l2];
        d[i].x[0] *= d[l].x[0];
        d[i].x[0] %= a;
        t = gcd(d[l].x[1], d[i].x[1]);
        d[i].x[1] *= d[l].x[1];
        d[i].x[1] /= t*t;
        d[i].x[2] *= t;
        d[i].x[2] *= d[l].x[2];
        d[i].x[2] %= a;
      }
      i = ++nd;
    } else ++k;
  }
```

◇

File defined by 298, 299, 300, 301, 302.

Wenn wir nun einen linear abhängigen Vektor ermittelt haben, dann brauchen wir nur noch zu überprüfen, ob die Relation von J. D. Dixon zutrifft:

"cfrac.cpp" 302 ≡

```
  if (j == n) {   // Solution?
    Natural y = sqrt(d[nd].x[1]) * d[nd].x[2];
    y %= a;
    t = a-y;
    if (d[nd].x[0] != y && d[nd].x[0] != t) {
      y += d[nd].x[0];
      t = gcd(y, a);
      v = 2;
      break;
    } else memset(d[nd].c, 0, n);
  } else if (i == nd) {
    for (i = 0; i < nd; ++i)
      if (d[i].c[j]) {
        d[i].c[j] = 0;
        for (Digit l2 = j+1; l2 < n; ++l2) d[i].c[l2] ^= d[nd].c[l2];
        d[i].x[0] *= d[nd].x[0];
        d[i].x[0] %= a;
        t = gcd(d[nd].x[1], d[i].x[1]);
        d[i].x[1] *= d[nd].x[1];
```

```
            d[i].x[1] /= t*t;
            d[i].x[2] *= t;
            d[i].x[2] *= d[nd].x[2];
            d[i].x[2] %= a;
          }
          ++nd;
        }
      }
      div(x, v, s, t);        // Kettenbruchentwicklung
      y = r-t; z += w*s; z %= a;
      if (x >= y) {
        t = x-y; t *= s; u += t;
      } else {
        t = y-x; t *= s; u -= t;
      }
      swap(v, u); swap(x, y); swap(w, z);
    }
    for (i = 0; i < nd; ++i) memset(d[i].c, 0, n);

    cout << "Iterationen=" << iteration << endl;
    cout << "Matrix size=" << nd << endl;
  }
  for (i = 0; i <= m; ++i) {
    delete[] d[i].x;
    delete[] d[i].c;
  }
  delete[] d;
  delete[] p;
  return t;
}

int main()
{
  Natural a = 1;
  a <<= 128; ++a;

  clock_t start = clock();
  Natural c = cfrac(a, 400, 257);
  clock_t stop = clock();
  cout << "time [s] = " << double(stop-start)/CLOCKS_PER_SEC << endl;

  cout << c << '*' << a/c << endl;

  return 1;
}
◇
```

File defined by 298, 299, 300, 301, 302.

Somit haben wir jetzt eine Funktion, die große zusammengesetzte Zahlen in zwei Faktoren aufteilt. Diese können wir so lange rekursiv anwenden, bis wir die gesuchte Primfaktorisierung gefunden haben oder für kleinere Zahlen zu den herkömmlichen Faktorisierungsmethoden überwechseln.

Auf einen Pentium 100 MHz erreichen wir mit dem Watcom-Compiler 10.6 die folgenden Zeiten:

a	b	m	Laufzeit
$2^{59}-1$	a	50	0.88 s
$2^{67}-1$	a	50	2.86 s
$2^{101}-1$	a	100	150.9 s
$2^{128}+1$	$257a$	400	2546.2 s

5.7.10 Euler-Funktion

Sei $n \in \mathbb{N}$, so bezeichnet man mit der **Euler-Funktion** $\varphi(n)$ die Anzahl aller natürlichen Zahlen, die kleiner n und teilerfremd zu n sind, das heißt

$$\varphi : \mathbb{N} \to \mathbb{N} : n \mapsto |\{\mathbb{N} \in x \leq n \mid \gcd(x,n) = 1\}|.$$

Für die Primfaktorzerlegung

$$n = \prod_{i=0}^{t} p_i^{a_i} \qquad \text{für } t \in \mathbb{N}, a_i \in \mathbb{N}_+, 0 \leq i \leq t$$

gilt

$$\varphi(n) = n \prod_{i=0}^{t} \left(1 - \frac{1}{p_i}\right).$$

Durch diese Eigenschaft läßt sich die Funktionswert $\varphi(n)$ direkt berechnen:

⟨ euler function 304 ⟩ ≡

```
Natural euler(Natural a)
// Algorithm:  c := euler(a)
// Input:      a in Natural.
// Output:     c in Natural such that c = phi(a) ||
{
  list<Natural> p;
  factoring(a, p);
  Natural c,d;

  list<Natural>::iterator i = p.begin();
  while (i != p.end()) {
    div(a, *i, c, d); a -= c;
    list<Natural>::const_iterator j = i;
    while (++i != p.end() && *j == *i);
  }
  return a;
}
```

◇

Macro referenced in 433.

Laufzeit. Sei $x \in$ Natural und C ein Container über Natural, dann

$$\texttt{euler}(x) \sim \texttt{factoring}(x,C).$$

5.8 Kombinatorik

5.8.1 Fakultät

Definition. $a! := \prod_{i=1}^{a} i$ für $a \in \mathbb{N}$.

Eine einfache Realisierung dieser Definition erhalten wir, indem wir das Produkt sukzessiv durch die einzelnen Faktoren erzeugen:

```
Natural factorial_conventional(const Digit a)
// Algorithm:  b := factorial_conventional(a)
// Input:      a in Digit.
// Output:     b in Natural such that b = a! ||
{
   Digit d = a;
   Digit i = 0;
   while (d > 1) { d >>= 1; i += a; }

   Natural::NumberOfDigits(i/BETA+1);
   Natural c = 1;
   Natural::RestoreSize();
   d = 1;
   for (i = 2; i <= a; i++) {
      Digit d1,d2;
      c.digitmul(i, d, d1, d2);
      if (d1) { c *= d; d = i; }
      else d = d2;
   }
   return c *= d;
}
```

Um keine internen Überläufe hervorzurufen, schätzen wir mit einem Trick von Gauß unsere Fakultätsfunktion nach oben ab:

$$(a!)^2 = \prod_{i=1}^{a} i(a+1-i) \leq \prod_{i=1}^{a} \frac{(a+1)^2}{4}$$

und somit

$$a! \leq \left(\frac{a+1}{2}\right)^a.$$

5.8.2 Fakultätsberechnung durch die Primfaktorzerlegung

Satz. Seien $a \in \mathbb{N}_+$, p Primzahl, $e_p(a) = \max\{\, n \in \mathbb{N} \mid p^n | a! \,\}$, so gilt

$$e_p(a) = \sum_{i \geq 1} \left\lfloor \frac{a}{p^i} \right\rfloor.$$

Beweis (Vollständige Induktion über a). Es gilt $e_p(1) = 0$ für alle Primzahlen p.
Induktionsannahme: Behauptung sei für $(a-1)!$ korrekt. Setze $f_p(a) := \sum_{i \geq 1} \left(\left\lfloor \frac{a}{p^i} \right\rfloor - \left\lfloor \frac{a-1}{p^i} \right\rfloor \right)$. Also ist $f_p(a)$ maximal mit $p^{f_p(a)} | a$, weil $\left\lfloor \frac{a}{p^i} \right\rfloor - \left\lfloor \frac{a-1}{p^i} \right\rfloor = \begin{cases} 1\,, & p^i \mid a \\ 0\,, & p^i \nmid a \end{cases}$ gilt.
Wegen $a \cdot (a-1)! = a!$ folgt somit $f_p(a) \cdot e_p(a-1) = e_p(a)$. □

Korollar. Es gilt

$$a! = \prod_{p \in P} p^{e_p(a)} \qquad \text{mit } P = \{p \in \mathbb{N} \mid p \text{ prim}\},\ e_p(a) = \sum_{i=1}^{\lceil \log_p a \rceil} \left\lfloor \frac{a}{p^i} \right\rfloor.$$

Korollar. $e_2(a) = a - \sum_{i=0}^{n} a_i$, wobei $a = \sum_{i=0}^{n} a_i \cdot 2^i$ mit $n = \lfloor \log_2 a \rfloor, a_i \in \{0,1\}$ für $0 \leq i \leq n$ die Binärdarstellung von a ist.

Beispiel. $10! = 2^8 \cdot 3^4 \cdot 5^2 \cdot 7 = 2^8 \cdot (3^2 \cdot 5)^2 \cdot 3 \cdot 7$, wegen
$e_2(10) = \left\lfloor \frac{10}{2} \right\rfloor + \left\lfloor \frac{10}{4} \right\rfloor + \left\lfloor \frac{10}{8} \right\rfloor = 8, e_3(10) = \left\lfloor \frac{10}{3} \right\rfloor + \left\lfloor \frac{10}{9} \right\rfloor = 4, e_5(10) = \left\lfloor \frac{10}{5} \right\rfloor = 2, e_7(10) = \left\lfloor \frac{10}{7} \right\rfloor = 1.$
Hier benötigen wir anstatt der neun Multiplikationen nur noch drei Multiplikationen, zwei Quadraturen und eine bitweise Verschiebung.

Mit dem Potenzieralgorithmus ist das Produkt schnell zu erzeugen:

⟨ calculates the function factorial 306 ⟩ ≡

```
Natural factorial(const Digit a)
// Algorithm:  b := factorial(a)
// Input:      a in Digit.
// Output:     b in Natural such that b = a! ||
{
  if (a <= 1) return 1;
  Digit d = a >> 1;
  Digit i = a;
  while (d > 1) { d >>= 1; i += a; }

  Natural::NumberOfDigits(size_t(i)/BETA+1);
  Natural c(1);
  Natural::RestoreSize();
  Primes primes;
  if (a > primes.lastPrime()) {
    d = 1;
    for (i = 2; i <= a; ++i) {
      Digit d1,d2;
      c.digitmul(i, d, d1, d2);
      if (d1) { c *= d; d = i; }
      else d = d2;
    }
    return c *= d;
  }
  const size_t t = primes.numberOfPrimes(a) - 1;
  Digit* e = NOTHROW_NEW Digit[t];
  if (!e) c.errmsg(2, "(factorial)");
  Digit* p = NOTHROW_NEW Digit[t];
  if (!p) c.errmsg(2, "(factorial)");
  primes.firstPrime();
  for (i = 0; i < t; ++i) {
    primes.nextPrime(p[i]);
    e[i] = d = a/p[i];
    while (d >= p[i]) e[i] += d /= p[i];
  }
```

```
  for (Digit j = Digit(1) << log2(e[0]);; j >>= 1) {
    d = 1;
    for (i = 0; i < t; ++i)
      if (e[i]&j) {
        Digit d1,d2;
        c.digitmul(d, p[i], d1, d2);
        if (d1) { c *= d; d = p[i]; }
        else d = d2;
      }
    c *= d;
    if (j == 1) break;
    c = c*c;
  }
  d = a;
  for (i = a; i; i >>= 1) d -= i&1;

  delete[] e;
  delete[] p;

  return c <<= size_t(d);
}
◇
```

Macro referenced in 433.

Laufzeit. Sei $a \in \mathsf{Digit}$, dann `factorial(`a`)`$\in \log(a) \cdot \mathcal{O}(\mathtt{sqr} : \mathsf{Natural} \to \mathsf{Natural})$.

Auf einen Pentium mit 100 MHz erreichen wir mit dem GNU-Compiler 2.7.2 die folgenden Zeiten:

	konventioneller Algorithmus	durch Primfaktorzerlegung
2000!	0.04 s	0.02 s
10000!	1.72 s	0.53 s
50000!	56.2 s	12.8 s

5.8.3 Binomialkoeffizient

Definition. Der Binomialkoeffizient wird folgendermaßen berechnet:

$$\binom{a}{b} := \frac{a!}{(a-b)! \cdot b!} = \prod_{i=1}^{b} \frac{a+1-i}{i}.$$

Zusätzlich besitzt er die Eigenschaft

$$\binom{a}{b} = 0 \qquad \text{, für } b > a.$$

Satz. Es gilt die Rekursionsformel $\binom{a}{b} = \binom{a}{b-1} \cdot \frac{a-b+1}{b}$.

Beweis. Es ist $\binom{a}{b}\binom{b}{c} = \frac{\prod_{i=1}^{b}(a+1-i)}{b!} \cdot \frac{b!}{(b-c)!\cdot c!} = \frac{\prod_{i=1}^{c}(a+1-i)}{c!} \cdot \frac{\prod_{i=1}^{b-c}(a+1-(i+c))}{(b-c)!} = \binom{a}{c}\binom{a-c}{b-c}$ für $a, b, c \in \mathbb{N}_+, c \le b \le a$. Und mit $c = b-1$ ist die Behauptung des Satzes bewiesen. □

Hieraus ergibt sich unter Beachtung der Teilbarkeit unmittelbar ein Algorithmus:

```
Natural binomial(const Digit a, const Digit b)
// Algorithm:  c := binomial(a, b)
// Input:      a,b in Digit.
// Output:     c in Natural such that c = a!/((a-b)!b!) ||
{
  if (b > a) return Digit(0);
  Natural c = 1;
  for (Digit i = 1; i <= b; ++i) { c *= a-i+1; c /= i; }
  return c;
}
```

5.8.4 Binomialkoeffizient durch die Primfaktorzerlegung

Schneller sind wir jedoch, wenn wir wie bei der Fakultätsberechnung die Primfaktorzerlegung verwenden.

Korollar. Es gilt

$$\binom{a}{b} = \prod_{p \in P} p^{e_p(a,b)} \qquad \text{mit } P = \{p \in \mathbb{N} \mid p \text{ prim}\},\ e_p(a,b) = \sum_{i=1}^{\lceil \log_p a \rceil} \left(\left\lfloor \frac{a}{p^i} \right\rfloor - \left\lfloor \frac{a-b}{p^i} \right\rfloor - \left\lfloor \frac{b}{p^i} \right\rfloor \right).$$

⟨ calculates the function binomial coefficient 308 ⟩ ≡

```
Natural binomial(const Digit a, const Digit b)
// Algorithm:  c := binomial(a, b)
// Input:      a,b in Digit.
// Output:     c in Natural such that c = a!/((a-b)!b!) ||
{
  if (b > a) return Digit(0);
  if (a <= 1) return 1;
  Natural c(1);
  Primes primes;
  if (a > primes.lastPrime()) {
    for (Digit i = 1; i <= b; ++i) {
      c *= a-i+1;
      c /= i;
    }
    return c;
  }
  const size_t t = primes.numberOfPrimes(a) - 1;
  Digit* e = NOTHROW_NEW Digit[t];
  if (!e) c.errmsg(2, "(binomial)");
  Digit* p = NOTHROW_NEW Digit[t];
  if (!p) c.errmsg(2, "(binomial)");
  primes.firstPrime();
  Digit d,i,j = 0;
  for (i = 0; i < t; ++i) {
    primes.nextPrime(p[i]);
    e[i] = d = a/p[i];
    while (d >= p[i]) e[i] += d /= p[i];
    e[i] -= d = b/p[i];
```

```
    while (d >= p[i]) e[i] -= d /= p[i];
    e[i] -= d = (a-b)/p[i];
    while (d >= p[i]) e[i] -= d /= p[i];
    if (e[i] > j) j = e[i];
  }
  for (j = Digit(1) << log2(j);; j >>= 1) {
    d = 1;
    for (i = 0; i < t; ++i)
      if (e[i]&j) {
        Digit d1,d2;
        c.digitmul(d, p[i], d1, d2);
        if (d1) { c *= d; d = p[i]; }
        else d = d2;
      }
    c *= d;
    if (j == 1) break;
    c = c*c;
  }
  d = 0;
  for (i = b; i; i >>= 1) d += i&1;
  for (i = a-b; i; i >>= 1) d += i&1;
  for (i = a; i; i >>= 1) d -= i&1;

  delete[] e;
  delete[] p;

  return c <<= size_t(d);
}
◇
```

Macro referenced in 433.

Laufzeit. Seien $a, b \in$ Digit, dann `binomial(`a`,`b`)`$\in \log(a) \cdot \mathcal{O}($`sqr` : Natural $\to$ Natural$)$.

Auf einem Pentium mit 100 MHz erreichen wir mit dem GNU-Compiler 2.7.2 die folgenden Zeiten:

	konventioneller Algorithmus	durch Primfaktorzerlegung
$\binom{20000}{10000}$	2.99 s	0.07 s
$\binom{20000}{5000}$	1.06 s	0.04 s
$\binom{20000}{2000}$	0.23 s	0.02 s

Kapitel 6

Konstanten-Berechnung

6.1 Eine einfache Festkommaklasse

Für die Berechnung von Konstanten genügt es, eine Festkomma-Arithemtik einzuführen, weil wir keinen direkten Gebrauch eines flexiblen Exponenten brauchen, wie er zum Beispiel bei der Gleitkomma-Arithmetik nötig ist. Stattdessen verwenden wir einen impliziten Exponenten, der ja nach Objekt an einer anderen Stelle liegt. Ein Beispiel: Das Objekt `Fixed(n)` mit $n \in$ `size_t` reserviert n Dezimalen des abgeleiteten Naturals für die Nachkommastellen.
Die Klasse Fixed ist jedoch im Gegensatz zu den Klassen Natural oder Integer momentan noch nicht zur direkten Nutzung durch den Benutzer vorgesehen. Sie ist von der Klasse Natural abgeleitet und besteht hauptsächlich aus dem Konstruktor und der Streamausgabe:

⟨ class Fixed 311a ⟩ ≡

```
class Fixed : private Natural {
private:
  size_t prec,dec;

public:
  Fixed(const size_t);
  Fixed(const Fixed&);

  void      output(OSTREAM&);
  void      set_decimals(const size_t);
  size_t    precision() const;
  size_t    decimals() const;
  Natural&  value() const;
};
```

◇

Macro referenced in 434.

⟨ inline implementation of class Fixed 311b ⟩ ≡

```
inline Natural& Fixed::value() const
{
  return (Natural&)*this;
}

inline Fixed::Fixed(const size_t n)
 : prec(NumberOfDecimals(n+ALPHA_WIDTH)), dec(n)
```

```
{
  RestoreSize();
}

inline Fixed::Fixed(const Fixed& a)
 : Natural(a.value()), prec(a.prec), dec(a.dec)
{
}

inline void Fixed::set_decimals(const size_t n)
{
  prec = NumberOfDecimals(n+ALPHA_WIDTH);
  dec = n;
  RestoreSize();
}

inline size_t Fixed::precision() const
{
  return prec;
}

inline size_t Fixed::decimals() const
{
  return dec;
}

inline OSTREAM& operator<<(OSTREAM& out, Fixed a)
// Algorithm:  o := operator<<(o, a)
// Input:      o in ostream, a in Fixed.
// Output:     o in ostream ||
//
// Note:       puts Fixed a on output stream.
{
  a.output(out);
  return out;
}
◇
```

Macro referenced in 434.

6.1.1 Streamausgabe

Die Streamausgabe entspricht bis auf die interne Festkommadarstellung der Ausgabe eines Naturals aus dem Kapitel 2.6.1:

⟨ puts Fixed on output stream 312 ⟩ ≡

```
static void output(OSTREAM& out, const Natural& a, const Natural* c,
                   const size_t m, size_t n)
{
  Natural q,r;
  div(a, *c, q, r);
  if (--n == 0) {
    out.width(m);
    out.fill('0');
    out << q;
    out.width(m);
```

```
    out.fill('0');
    out << r;
  } else {
    output(out, q, c+1, m, n);
    output(out, r, c+1, m, n);
  }
}

void Fixed::output(OSTREAM& out)
// Algorithm:  a.output(o)
// Input:      o in ostream, a in Fixed.
// Output:     o in ostream ||
//
// Note:       puts Fixed a on output stream.
{
  size_t n = decimals();
  size_t steps = 1;

  if (n >= 80000000) steps = 4096;
  else if (n >= 40000000) steps = 2048;
  else if (n >= 20000000) steps = 1024;
  else if (n >= 8000000) steps = 512;
  else if (n >= 4000000) steps = 256;
  else if (n >= 2000000) steps = 128;
  else if (n >= 800000) steps = 64;
  else if (n >= 400000) steps = 32;
  else if (n >= 200000) steps = 16;
  else if (n >= 100000) steps = 8;
  else if (n >= 50000) steps = 4;
  else if (n >= 10000) steps = 2;

  const size_t block_size = 100*steps;
  const size_t d          = log2(steps)+1;
  size_t m = precision();

  Natural* c = new Natural[d];
  if (c == 0) errmsg(2, "(output)");
  c[d-1] = pow(Natural(ALPHA), Digit(block_size/steps));
  for (int i = d-2; i >= 0; --i) c[i] = c[i+1]*c[i+1];

  const size_t sB = length();
  enlarge((sB > m)? 3 : 3+m-sB);
  if (sB > m) {
    fast_rshift(m);
    out << value();
    if (n) out << '.';
    value() = 0; fast_append(m); normalize();
  } else out << "0.";
  while (n >= ALPHA_WIDTH*(block_size/steps)) {
    int j = 0;
    do {
      if (n >= ((ALPHA_WIDTH*block_size) >> j)) {
        value() *= c[j];
        out.width((ALPHA_WIDTH*block_size) >> j);
        break;
      }
```

```
      ++j;
    } while (j < d);
    out.fill('0');
    if (length() > m) {
      fast_rshift(m);
      if (j == d-1) out << value();
      else ::output(out, value(), c+j+1, ALPHA_WIDTH*(block_size/steps), d-j-1);
      value() = 0; fast_append(m); normalize();
      if (value() != 0) {
        Digit* pT;
        const size_t sz = trailing_zeros(pT);
        fast_rshift(sz); m -= sz;
      }
    } else out << '0';
    n -= (block_size*ALPHA_WIDTH) >> j;
  }
  Digit* pE = last();
  Digit* pT = pE-m;
  while (n) {
    value() *= ALPHA;
    if (n <= ALPHA_WIDTH) {
       out.width(n);
       out.fill('0');
       Digit d = highest();
       for (n = ALPHA_WIDTH-n; n; --n) d /= 10;
       out << d;
       break;
    }
    out.width(ALPHA_WIDTH);
    out.fill('0');
    if (length() > m) {
      out << *pT;
      *pT = 0;
      if (*pE == 0) { --pE; fast_rshift(1); --m; }
      normalize();
    } else out << '0';
    n -= ALPHA_WIDTH;
  }
  delete[] c;
}
◇
```

Macro referenced in 437.

Laufzeit. Sei $o \in$`ostream` und $f \in$ Fixed, dann f`.output(`o`)` $\in \mathcal{O}\left(\frac{f\text{.precision()}}{2} \cdot \left\lfloor \frac{\beta f\text{.precision()}}{\log_2 \alpha} \right\rfloor\right)$.

Bemerkung. Die Umwandlung ins Dezimalsystem wird blockweise durchgeführt, um die Ausgabe zu beschleunigen.

6.2 π-Berechnung

Zu den Klassikern der Mathematikgeschichte gehört das Problem der Quadratur des Kreises mit Zirkel und Lineal. Die antiken Mathematiker haben sich erfolglos darum bemüht; ungelöst schleppte sich die Aufgabe durch die Jahrhunderte fort bis weit in die Neuzeit hinein, genaugenommen bis ins Jahr 1882, als F. Lindemann zeigen konnte, daß π transzendent über den rationalen Zahlen ist. Da jegliche

Konstruktionen mit Zirkel und Lineal in der Algebra aber quadratischen und linearen Gleichungen entsprechen, ist die Konstruktion also unmöglich.
Weil man parallel zu den geometrischen Lösungen auch nach numerischen gesucht hatte, machte schon 200 Jahre vor der endgültigen Absage an die Konstruktion G. W. Leibniz einen entscheidenden Fortschritt in der π-Berechnung mit der Entdeckung der Reihe $\frac{\pi}{4} = \arctan(1) = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$. Daraufhin haben viele Mathematiker durch trickreiches Anwenden der Additionstheoreme der Tangensfunktion eine immense Anzahl von neuen Arkustangens-Reihen gebastelt (siehe Anhang E). Genaueres zu diesen Reihen findet man in dem Artikel [30] von D. H. Lehmer.
Im Laufe der Zeit sind noch andere merk- und denkwürdige Beziehungen zur Zahl π hinzugekommen, wie zum Beispiel die, daß die Wahrscheinlichkeit, daß zwei natürliche Zahlen teilerfremd sind, gerade $\frac{6}{\pi^2}$ beträgt, oder die Entstehung von neuen Multiplikationsalgorithmen (A. Karatsuba, siehe Kapitel 2.11.3) oder auch die Formel $e^{\pi i} = -1$ mit der Eulerschen Zahl $e = \lim_{n\to\infty}(1+\frac{1}{n})^n$ und der imaginären Einheit $i^2 = -1$.
Heute wird die Kreiszahlermittlung gerne für Benchmark beziehungsweise Prozessor- oder Arithmetik-Testsysteme verwendet, da sich schon kleine Fehler in der Berechnung sofort bemerkbar machen.

6.2.1 Historische Zusammenfassung über die π-Berechnung

1. Durch handschriftliche Berechnung:
   Die ersten ernsthaften Versuche, die Zahl π genauer zu bestimmen, gehen auf **Archimedes** (287-212 v. Chr.) zurück. Er führte Untersuchungen an regelmäßigen Vielecken durch und konnte π auf 2 Dezimalen genau eingrenzen ($3\frac{10}{71} < \pi < 3\frac{1}{7}$).
   Ptolemäus (150) ..... auf 4 Dezimalen genau mit dem Bruch $\frac{377}{120}$.
   Tsu Ch'ung (480) ..... auf 6 Dezimalen genau mit dem Bruch $\frac{355}{113}$.
   Ludolph von Coelen (1539-1610) ..... auf 32 Dezimalen genau
   Er machte es zu seiner Lebensaufgabe, die Umfänge von ein- und umbeschriebenen Vielecken mit 2^{62} Seiten zu berechnen.
   Abraham Sharp (1653-1742) ..... auf 72 Dezimalen genau mit der Formel $\frac{\pi}{2} = \sqrt{3}\sum_{n=0}^{\infty} \frac{(-1)^n}{3^n(2n+1)}$.
   John Machin (1706) ..... auf 100 Dezimalen mit der Formel $\frac{\pi}{4} = 4\arctan\left(\frac{1}{5}\right) - \arctan\left(\frac{1}{239}\right)$.
   De Lagny (1719) ..... auf 127 Dezimalen (davon 112 korrekt)
   Vega (1754-1802) ..... auf 140 Dezimalen
   Zacharias Dase (1844) ..... auf 200 Dezimalen
   Deutscher Rechenkünstler, konnte hundertstellige Zahlen in etwa acht Stunden im Kopf multiplizieren. Er benutzte die Formel $\frac{\pi}{4} = \arctan\left(\frac{1}{2}\right) + \arctan\left(\frac{1}{5}\right) + \arctan\left(\frac{1}{8}\right)$ und benötigte knapp zwei Monate zur Berechnung der 200 Stellen von π.
   William Rutherford (1853) ..... auf 400 Dezimalen
   William Shanks (1853) ..... auf 707 Dezimalen (davon 527 korrekt)
   Es vergingen 92 Jahre, bis der Rechenfehler entdeckt wurde.

2. Mittels elektronischer Rechenanlagen:
   D. F. Ferguson (1947) ..... auf 808 Dezimalen mit der Formel $\frac{\pi}{4} = 3\arctan\left(\frac{1}{4}\right) + \arctan\left(\frac{1}{20}\right) + \arctan\left(\frac{1}{1985}\right)$.
   John von Neumann et al. (1949) ..... auf 2037 Dezimalen
   berechnet in 70 Stunden mit Machins Formel auf dem Röhrenmonstrum ENIAC (18 000 Vakuumröhren).
   S. C. Nicholson, J. Jeenel (1954) ..... auf 3092 Dezimalen
   G. E. Felton (1957) ..... 10 000 Dezimalen (davon 7480 korrekt)

berechnet auf einem Ferranti PEGASUS in 33 Stunden.
François Genuys (1958) ..... 10 000 Dezimalen
berechnet auf einem IBM 704 in 100 Minuten.
Jean Guilloud (1959) ..... 16 167 Dezimalen
John W. Wrench jr., Daniel Shanks (1961) ..... 100 265 Dezimalen
berechnet auf einem IBM 7090 in weniger als 9 Stunden mit der Formel von Störmer
$\frac{\pi}{4} = 6 \arctan\left(\frac{1}{8}\right) + 2 \arctan\left(\frac{1}{57}\right) + \arctan\left(\frac{1}{239}\right)$.
J. Guilloud, J. Filliatre (1966) ..... 250 000 Dezimalen
J. Guilloud, M. Dichampt (1967) ..... 500 000 Dezimalen
berechnet auf einer CDC 6600.
Jean Guilloud, Martine Bouyer (1973) ..... 1 001 250 Dezimalen
berechnet auf einer CDC 7600 in knapp einem Tag Rechenzeit mit der Formel von C. F. Gauß
$\frac{\pi}{4} = 12 \arctan\left(\frac{1}{18}\right) + 8 \arctan\left(\frac{1}{57}\right) - 5 \arctan\left(\frac{1}{239}\right)$.
Y. Tamura (1982) ..... 2 097 144 Dezimalen
Y. Tamura, Y. Kanada (1982) ..... 4 194 288 Dezimalen
Y. Tamura, Y. Kanada (1982) ..... 8 388 576 Dezimalen
Kannada, Tamura, Yoshino, Ushiro (1983) ..... 16 777 216 Dezimalen
berechnet auf einer HITAC M-280H in 30 Stunden mit der quadratisch konvergenten Iteration von P. Brent und E. Salamin $a_0 = 1, b_0 = \frac{\sqrt{2}}{2}, c_0 = \frac{1}{4}$;
$a_{n+1} = \frac{a_n + b_n}{2}, b_{n+1} = \sqrt{b_n a_n}, c_{n+1} = c_n - 2^n (a_{n+1} - a_n)^2$ für $n \in \mathbb{N}$.
Es gilt $\lim_{n\to\infty} \frac{(a_n + b_n)^2}{4 c_n} = \pi$.
R. William Gosper (1985) ..... 17 526 200 Dezimalen
berechnet auf einer Symbolics 3670 mit der Reihe von Srinivasa Ramanujan
$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{n=0}^{\infty} \frac{(4n)!}{(n!)^4} \frac{1103 + 26390n}{396^{4n}}$, wobei noch heute kein Beweis existiert, ob diese Summe tatsächlich gegen $\frac{1}{\pi}$ konvergiert.
David H. Bailey (1986) ..... 29 360 111 Dezimalen
berechnet auf einem CRAY-2-Supercomputer der amerikanischen Luft- und Raumfahrtbehörde NASA in weniger als 28 Stunden.
Y. Kanada, Y. Tamura (1986) ..... 33 554 414 Dezimalen
Y. Kanada, Y. Tamura (1986) ..... 67 108 839 Dezimalen
Y. Kanada, Y. Tamura, Y. Kubo, et al. (1987) ..... 134 214 700 Dezimalen
berechnet auf einem SX-2-Supercomputer der Firma NEC.
Y. Kanada, Y. Tamura (1988) ..... 201 326 551 Dezimalen
G. V. Chudnovsky und D. V. Chudnovsky (1989) ..... 480 000 000 Dezimalen
G. V. Chudnovsky und D. V. Chudnovsky (1989) ..... 525 229 270 Dezimalen
Y. Kanada, Y. Tamura (1989) ..... 536 870 898 Dezimalen
G. V. Chudnovsky und D. V. Chudnovsky (1989) ..... 1 011 196 691 Dezimalen
berechnet auf einer IBM 3090 mit ihrer eigenen Formel
$\frac{1}{\pi} = \frac{1}{53360\sqrt{640320}} \sum_{n=0}^{\infty} (-1)^n \frac{(6n)!}{(3n)!(n!)^3} \frac{13591409 + 545140134n}{640320^{3n}}$
Y. Kanada, Y. Tamura (1989) ..... 1 073 741 799 Dezimalen
G. V. Chudnovsky und D. V. Chudnovsky (1991) ..... 2 260 000 000 Dezimalen
D. Takahashi, Y. Kanada (1995) ..... 3 221 225 466 Dezimalen
berechnet auf einer HITAC S-3800/480 in knapp 37 Stunden mit dem quartisch konvergenten Algorithmus der Gebrüder Borwein $a_0 = 6 - 4\sqrt{2}, b_0 = \sqrt{2} - 1$,
$b_{n+1} = \frac{1 - \sqrt[4]{1 - b_n^4}}{1 + \sqrt[4]{1 - b_n^4}}, a_{n+1} = a_n (1 + b_{n+1})^4 - 2^{2n+3} b_{n+1} (1 + b_{n+1} + b_{n+1}^2)$ für $n \in \mathbb{N}$.
G. V. Chudnovsky und D. V. Chudnovsky (1994) ..... 4 044 000 000 Dezimalen
D. Takahashi, Y. Kanada (1995) ..... 4 294 967 286 Dezimalen
D. Takahashi, Y. Kanada (1995) ..... 6 442 450 938 Dezimalen
berechnet auf einer HITAC S-3800/480 in 112 Stunden mit dem quartisch konvergenten Algorithmus der Gebrüder Borwein.

Y. Kanada, D. Takahashi (1997) ..... 51 539 600 000 Dezimalen berechnet auf einer HITACHI SR2201 mit 1024 Prozessoren in 29 Stunden mit dem quartisch konvergenten Algorithmus der Gebrüder Borwein.

6.2.2 π beschrieben durch den Arkustangens

Wir greifen nun eine ganz bestimmte Formel zur π-Berechnung bis etwa 10 000 Dezimalen heraus:

Arkustangens-Reihe von Störmer

$$\pi = 24\arctan\left(\frac{1}{8}\right) + 8\arctan\left(\frac{1}{57}\right) + 4\arctan\left(\frac{1}{239}\right).$$

Um die Arkustangens-Formel von Störmer einsetzen zu können, benötigen wir noch die Arkustangens-Reihe, die im Jahre 1671 von Gregory entdeckt wurde:

$$\begin{aligned}\arctan\left(\frac{1}{x}\right) &= \sum_{n\geq 0}\frac{(-1)^n}{(2n+1)x^{2n+1}} \\ &= \sum_{n\geq 0}\frac{1}{(4n+1)x^{4n+1}} - \sum_{n\geq 0}\frac{1}{(4n+3)x^{4n+3}} \\ &= \sum_{n\geq 0}\frac{(4n+3)-(4n+1)x^2}{(4n+1)(4n+3)x^{4n+3}}.\end{aligned}$$

Um die Konvergenz etwas zu beschleunigen, fassen wir immer zwei aufeinanderfolgende Summanden aus der alternierenden Reihe zusammen.
Die Implementierung sieht dann folgendermaßen aus:

⟨ Pi-Calculation with Stoermer-Algorithms 317 ⟩ ≡

```
void Pi::stoermer(const size_t n, Natural& pi)
// Algorithm:  p.stoermer(n, a)
// Input:      p in Pi, a in Natural, n in size_t
//             where n < 3(GAMMA_LOW+1)/BETA, BETA >= 32.
// Output:     a in Natural such that |a-pi*2^(BETA*n)| < 1 ||
{
  const size_t sz = BETA*n;
  Natural t;
  Natural a = Digit(8*24);
  Natural b = Digit(57*8);
  Natural c = Digit(239*4);
  pi        = 0;
  Digit   k = 0;

  a <<= sz; b <<= sz; c <<= sz;
  do {
    a >>= 12;
    t = a * (63*k+191);
    if (b != 0) {
      b /= Digit(57*57*57*57);
      t += b * Digit(3248*k+9746);
      if (c != 0) {
```

```
        c /= Digit(239UL*239*239*239);
        t += c * Digit(57120*k+171362);
      }
    }
    pi += t /= (k+1)*(k+3);
    k += 4;
  } while (t != 0);
}
```

◇

Macro referenced in 437.

Laufzeit. Sei $p \in \mathsf{Pi}$, $x \in \mathsf{Natural}$ und $t \in$ `size_t`, dann

$$p.\texttt{stoermer}(t,x) \in \frac{t\beta}{3\log_{10}\alpha} \cdot \mathcal{O}(\texttt{operator/} : \mathsf{Natural} \times \mathsf{Digit} \to \mathsf{Natural}).$$

Dieser Algorithmus ist für einige tausend Dezimalstellen kompakt und sehr schnell, weil der am langsamsten konvergierende Summand eine Zweierpotenz ist und dadurch unserer internen Darstellung angepaßt ist. Genauer gesagt liegt unsere Genauigkeitsschranke bei $2^{\frac{\beta}{2}} \cdot \log_{10}(8)$ Stellen, da danach ein Überlauf im Digitbereich bei `(k+1)*(k+3)` auftritt. In Zahlen ausgedrückt heißt das, daß wir bei einem 32-Bit-System 59184 dezimale Nachkommastellen ermitteln können. Doch für diese Größen existieren weitaus schneller konvergierende Algorithmen.

6.2.3 Ramanujan-Reihen

Srinivasa Ramanujan hat sehr viele Reihen zur Berechnung von π angegeben (siehe Anhang E.3, Seite 452f). Die Chudnovsky Brüder formulierten 1987 aus der Verallgemeinerung dieser Reihen die am schnellsten konvergierende Reihe, die mit 32-Bit Digits auskommt:

$$\begin{aligned}
\frac{1}{\pi} &= \frac{12}{C^{\frac{3}{2}}} \sum_{n\geq 0} (-1)^n \frac{(6n)!}{(3n)!(n!)^3} \frac{A+Bn}{C^{3n}} \\
&= \frac{12}{C^{\frac{3}{2}}} \sum_{n\geq 0} (-1)^n \left(\prod_{k=1}^{n} \frac{6k(6k-1)(6k-2)(6k-3)(6k-4)(6k-5)}{3k(3k-1)(3k-2)} \right) \frac{A+Bn}{(n!)^3 \cdot C^{3n}} \\
&= \frac{12}{C^{\frac{3}{2}}} \sum_{n\geq 0} (-1)^n \left(\prod_{k=1}^{n} 8(6k-1)(6k-3)(6k-5) \right) \frac{A+Bn}{(n!)^3 \cdot C^{3n}} \\
&= \frac{12}{C^{\frac{3}{2}}} \sum_{n\geq 0} (-1)^n \frac{(\prod_{k=1}^{n}(6k-1)(2k-1)(6k-5))}{(n!)^3} \frac{A+Bn}{26680 \cdot C^{2n}} \\
&\quad \text{mit } A = 13591409, B = 545140134, C = 640320.
\end{aligned}$$

Diese Formel liefert etwa 14,1643 dezimale Stellen pro Term.

Bei der Berechnung dieser linear konvergierenden Reihe verwenden wir einen rekursiven Algorithmus, der die schnelle Multiplikation ausnutzt. Kurz gesagt werden alle rationalen Zahlen (die Terme) miteinander addiert ohne ihren gemeinsamen Teiler zu entfernen. Eine genaue Beschreibung dieses Algorithmus einschließlich einiger Beispiele führen Bruno Haible und Thomas Papanikolaou in [21] durch.

⟨ Pi-Calculation with Chudnovsky-Algorithms 318 ⟩ ≡

```
void Pi::chudnovsky(Digit n, const Digit m, Integer& p, Integer& q, Integer& t) const
// Algorithm:  p.chudnovsky(n, m, p, q, t)
// Input:      p in Pi, p,q,t in Integer, n,m in Digit where n < 2^BETA/6, n < m.
// Output:     p,q,t in Integer ||
{
  CONDITION(n < m && n < ((size_t(1) << (BETA-1))/3));

  const SignDigit A = SignDigit(13591409);
  const SignDigit B = SignDigit(545140134);
  const SignDigit C = SignDigit(640320);
  switch (m-n) {
    case 1: {
      ⟨ special case to evaluate one term 320a ⟩
      break;
    }
    case 2: {
      ⟨ special case to evaluate two terms 320b ⟩
      break;
    }
    case 3: {
      ⟨ special case to evaluate three terms 321a ⟩
      break;
    }
    default: {
      Integer p1,q1,t1;
      Integer p2,q2,t2;

      const size_t l = (n+m)/2;
      chudnovsky(n, l, p1, q1, t1);
      chudnovsky(l, m, p2, q2, t2);
      t = t1*q2; t += t1 = t2*p1;
      p = p1*p2; q = q1*q2;
    }
  }
}
```

◇

Macro defined by 318, 319.
Macro referenced in 437.

Laufzeit. Sei $p \in \mathsf{Pi}$, $u, v, w \in \mathsf{Integer}$ und $t_0, t_1 \in \mathsf{Digit}$, dann

$$p.\mathtt{chudnovsky}(t_0, t_1, u, v, w) \in \mathcal{O}(\log_2(t_1 - t_0)^2 \cdot \mathcal{O}(\mathtt{mul} : \mathsf{Integer}^2 \to \mathsf{Integer}).$$

⟨ Pi-Calculation with Chudnovsky-Algorithms 319 ⟩ ≡

```
void Pi::chudnovsky2(const Digit n, const Digit m, Integer& q, Integer& t) const
// Algorithm:  p.chudnovsky(n, m, q, t)
// Input:      p in Pi, q,t in Integer, n,m in Digit where n < m.
// Output:     q,t in Integer ||
{
  CONDITION(n < m);

  Integer p1,q1,t1;
  Integer q2,t2;

  const Digit l = (n+m)/2;
  chudnovsky(n, l, p1, q1, t1);
```

```
  if (m-l <= 3) {
    Integer p2;
    chudnovsky(l, m, p2, q2, t2);
  } else chudnovsky2(l, m, q2, t2);
  t = t1*q2; t += t1 = t2*p1;
  q = q1*q2;
}
```
◇

Macro defined by 318, 319.
Macro referenced in 437.

Laufzeit. Sei $p \in$ Pi, $u, v, w \in$ Integer und $t_0, t_1 \in$ Digit, dann

$$p.\texttt{chudnovsky2}(t_0,t_1,v,w) \sim p.\texttt{chudnovsky}(t_0,t_1,u,v,w).$$

Zur schnellen Berechnung der Terme und Verringerung der Rekursionstiefe berechnen wir bis zu drei Terme explizit durch die folgenden drei Programmteile:

⟨ special case to evaluate one term 320a ⟩ ≡

```
if (n <= GAMMA_LOW/8) p = (6*n-5)*(2*n-1);
else { p = 6*n-5; p *= 2*n-1; }
p *= 6*n-1; p = -p;
if (n <= GAMMA_LOW/2) q = n*n;
else { q = n; q *= n; }
q *= n; q *= 230*C; q *= 116*C;
t = B; t *= n; t += A; t *= p;
```
◇

Macro referenced in 318.

⟨ special case to evaluate two terms 320b ⟩ ≡

```
Integer s1,s2;
if (n <= GAMMA_LOW/8) {
  p = (6*n-5)*(2*n-1);
  s1 = p *= 6*n-1; s1 = -s1;
  p *= 6*n+5; p *= (2*n+1)*(6*n+1);
} else {
  p = 6*n-5; p *= 2*n-1;
  s1 = p *= 6*n-1; s1 = -s1;
  p *= 6*n+5; p *= 2*n+1; p *= 6*n+1;
}
if (n < GAMMA_LOW/2) {
  const size_t k = n+1;
  q = k*k; q *= k; q *= 230*C;
  s2 = q *= 116*C; q *= n*n;
} else {
  const size_t k = n+1;
  q = k; q *= k; q *= k; q *= 230*C;
  s2 = q *= 116*C; q *= n; q *= n;
}
q *= n; q *= 230*C; q *= 116*C;
t = B; t *= n; t += A;
const Integer s3 = t;
t += B;
const Integer s4 = t*p;
t = s1*s2*s3;
t += s4;
```
◇

Macro referenced in 318.

⟨ special case to evaluate three terms 321a ⟩ ≡

```
Integer s1,s2;
if (n <= GAMMA_LOW/12) {
  p = (6*n+1)*(2*n-1); p *= (6*n-1)*(6*n-5);
  s1 = p *= (2*n+1)*(6*n+5);
  p *= (6*n+7)*(2*n+3);
} else {
  p = 6*n-5; p *= 2*n-1; p *= 6*n-1;
  p *= 6*n+5; p *= 2*n+1;
  s1 = p *= 6*n+1;
  p *= 6*n+7; p *= 2*n+3;
}
p *= 6*n+11; p = -p;
if (n < GAMMA_LOW/2-1) {
  const size_t k = n+1;
  q = k*k; q *= k; q *= 230*C; q *= 116*C;
  const size_t k2 = n+2;
  q *= k2*k2; q *= k2; q *= 230*C;
  s2 = q *= 116*C; q *= n*n;
} else {
  const size_t k = n+1;
  q = k; q *= k; q *= k; q *= 230*C; q *= 116*C;
  const size_t k2 = n+2;
  q *= k2; q *= k2; q *= k2; q *= 230*C;
  s2 = q *= 116*C; q *= n; q *= n;
}
q *= n; q *= 230*C; q *= 116*C;
t = B; t *= n; t += A;
const Integer s3 = t;
t += B;
const Integer s4 = t;
t += B;
const Integer s5 = t*p;
t = s2*s3;
if (n <= GAMMA_LOW/8) t *= (6*n-5)*(2*n-1);
else { t *= 6*n-5; t *= 2*n-1; }
t *= 6*n-1; t = -t;
s2 = s1*s4;
n += 2;
if (n <= GAMMA_LOW/2) s2 *= n*n;
else { s2 *= n; s2 *= n; }
s2 *= n; s2 *= 230*C; s2 *= 116*C;
t += s2; t += s5;
◇
```

Macro referenced in 318.

6.2.4 Quadratwurzel

Eine genauere Beschreibung der Quadratwurzelberechnung findet man im Abschnitt 6.3.

⟨ Pi-Sqrt-Series-Calculation 321b ⟩ ≡

```
void Pi::sqrt_series(Digit n, const Digit m, Integer& p, Integer& q, Integer& t) const
// Algorithm:  z.series(n, m, p, q, t)
// Input:      z in Zeta3, p,q,t in Integer, n,m in Digit where n < 2^BETA/64-32, n < m.
// Output:     p,q,t in Integer ||
{
  CONDITION(n < m && n < GAMMA/2);

  const static Natural u2("2JSQ4J31843NN94KLN88HF3G1EA6DH2NQ24PCG707KEGHQ1", 32);
  switch (m-n) {
    case 1: {
      p = 2*n-3;
      q = u2; q *= 2*n;
      t = p; t = -t;
      break;
    }
    case 2: {
      t = q = u2;
      q *= 2*n; t *= 2*n+2; q *= t;
      t += 2*n-1; t *= 2*n-3; t = -t;
      if (n < GAMMA/8) p = (2*n-3)*(2*n-1);
      else { p = 2*n-3; p *= 2*n-1; }
      break;
    }
    case 3: {
      if (n < GAMMA/8) p = (2*n-1)*(2*n+1);
      else { p = 2*n-1; p *= 2*n+1; }
      p *= 2*n-3;
      t = q = u2;
      q *= 2*n+2; t *= 2*n+4; q *= t;
      t += 2*n+1; t *= 2*n-1; t = -t;
      t -= q; t *= 2*n-3;
      q *= 2*n; q *= u2;
      break;
    }
    default: {
      Integer p1,q1,p2,q2;
      Integer t1,t2;

      const Digit l = (n+m)/2;
      sqrt_series(n, l, p1, q1, t1);
      sqrt_series(l, m, p2, q2, t2);
      t = t1*q2; t += t1 = t2*p1;
      p = p1*p2; q = q1*q2;
    }
  }
}
◇
```

Macro defined by 321b, 322.
Macro referenced in 437.

Laufzeit. Sei $p \in$ Pi, $u, v, w \in$ Integer und $t_0, t_1 \in$ Digit, dann

$$p.\texttt{sqrt\_series}(t_0, t_1, u, v, w) \in \mathcal{O}(\log_2(t_1 - t_0)^2 \cdot \mathcal{O}(\texttt{mul} : \mathsf{Integer}^2 \to \mathsf{Integer}).$$

⟨ Pi-Sqrt-Series-Calculation 322 ⟩ ≡

```
void Pi::sqrt_series2(const Digit n, const Digit m, Integer& q, Integer& t) const
```

```
// Algorithm:  z.series2(n, m, q, t)
// Input:      z in Zeta3, q,t in Integer, n,m in Digit where n < m.
// Output:     q,t in Integer ||
{
  CONDITION(n < m);

  Integer p1,q1,q2;
  Integer t1,t2;

  const Digit l = (n+m)/2;
  sqrt_series(n, l, p1, q1, t1);
  if (m-l <= 3) {
    Integer p2;
    sqrt_series(l, m, p2, q2, t2);
  } else sqrt_series2(l, m, q2, t2);
  t = t1*q2; t += t1 = t2*p1;
  q = q1*q2;
}
◇
```

Macro defined by 321b, 322.
Macro referenced in 437.

Laufzeit. Sei $p \in \mathsf{Pi}$, $u, v, w \in \mathsf{Integer}$ und $t_0, t_1 \in \mathsf{Digit}$, dann

$$p.\texttt{sqrt\_series2}(t_0,t_1,v,w) \sim p.\texttt{sqrt\_series}(t_0,t_1,u,v,w).$$

6.2.5 Das arithmetisch-geometrische Mittel

Die bisherigen Formeln haben den Vorteil, daß sie ausschließlich elementare Grundoperationen benutzen, aber den Nachteil, daß sie nur linear konvergieren. Das bedeutet also zum Beispiel, daß sich die Laufzeit vervierfacht, wenn wir die gewünschte Stellenanzahl verdoppeln.
Die neuesten Methoden zur Berechnung von π beruhen auf den Untersuchungen, die C. F. Gauß zum arithmetischen und geometrischen Mittel zweier Zahlen machte:

Seien $a_0 = 1, b_0 = \frac{1}{2}\sqrt{2}, c_0 = \frac{1}{4}$ und

$$\begin{aligned} a_{n+1} &= \frac{a_n + b_n}{2}, \\ b_{n+1} &= \sqrt{b_n a_n}, \\ c_{n+1} &= c_n - 2^n (a_{n+1} - a_n)^2 \quad \text{für } n \in \mathbb{N}, \end{aligned}$$

dann gilt

$$\lim_{n\to\infty} \frac{(a_n + b_n)^2}{4c_n} = \pi.$$

Eine ausführliche Analyse und Betrachtung dieser Gleichung sowie theoretische Hintergründe findet man im Buch [6] von J. M. Borwein und P. B. Borwein.
Zusammenfassend gesagt beruht dieser Algorithmus auf einer Schleife, die bei jedem Durchlauf die Anzahl der korrekten Stellen annähernd verdoppelt. Dies ergibt zum Beispiel bereits bei der neunten Iteration ungefähr eine Millionen Stellen.

Bei der Umsetzung dieser Formel verwenden wir einige Optimierungsvorschläge von A. Schönhage aus [42] (Seite 266):

⟨ Pi-Calculation with Schönhage-Algorithms 324a ⟩ ≡

```
void Pi::schoenhage(const size_t n, Natural& pi)
// Algorithm:  p.schoenhage(n, a)
// Input:      p in Pi, a in Natural, n in size_t.
// Output:     a in Natural such that |a-pi*2^(BETA*n)| < 1 ||
{
  const size_t m = size_t(log2(BETA*n+BETA)-1.68025927);
  const size_t sz = BETA*n;

  Integer s,t,a(1),A(1),b(1),B(1),c(1);
  a <<= sz; A <<= sz; B <<= sz-1; c <<= sz-2;
  for (size_t k = 0; k <= m; ++k) {
    t = A+B; t >>= 2;
    s = B << sz; b = sqrt(s);
    a += b; a >>= 1;
    A = a*a; A >>= sz;
    B = A-t; B <<= 1;
    t = A-B; t <<= k; c -= t;
  }
  A <<= sz;
  div(A, c, t, a);
  pi = abs(t);
}
```

◇

Macro referenced in 437.

Laufzeit. Sei $p \in$ Pi, $x \in$ Natural und $t \in$ size_t, dann

$$p.\texttt{schoenhage}(t,x) \in \mathcal{O}(\log_2(t) \cdot \mathcal{O}(\texttt{sqrt} : \text{Natural} \to \text{Natural}).$$

6.2.6 Auswahl der Methoden

Im Konstruktor der Klasse Pi wählen wir nun zwischen den vorgestellten Algorithmen denjenigen aus, der die gewünschte Anzahl an dezimalen Nachkommastellen von π am schnellsten berechnet:

⟨ select π-Algorithm 324b ⟩ ≡

```
Pi::Pi(const size_t n)
// Algorithm:  a := Pi(n)
// Input:      n in size_t.
// Output:     a in Pi such that |a-pi*10^n| < 1 ||
 : pi(n)
{
  const size_t sz = pi.precision();

  if (BETA < 32) schoenhage(sz, pi.value());
  else if (sz < (3*GAMMA_LOW+3)/BETA && n < 10000)
    stoermer(sz, pi.value());
  else if (n >= Digit(100000000) || n >= ((size_t(1) << (BETA-1))/3))
    schoenhage(sz, pi.value());
  else {
    Integer p1,q1,t1;
    Integer q2,t2;

    // sqrt(C):
```

```
    const static Natural u("1JQ2P3OQIFH25EO9DDDA1KD1", 32);
    const static Natural v("229Q8MPR3DNV36SDE9P7HD", 32);
    Digit k = 2 + Digit(n/69.65551373);   // 2*log10(u), k >= 5

    Digit l = (k+1)/2;
    sqrt_series(2, l, p1, q1, t1);
    q2 = u; q2 *= u; q2 <<= 1; t1 -= q1; q1 *= q2; t1 += q1;
    sqrt_series2(l, k, q2, t2);
    Integer t = t1*q2;
    t += t1 = t2*p1; t *= u;
    q1 *= v;
    Integer t3 = q1*q2;


    // Pi:
    const SignDigit A = SignDigit(13591409);
    const SignDigit C = SignDigit(640320);
    k = size_t(n/14.1643);
    l = (k+1)/2;
    chudnovsky(1, l, p1, q1, t1);
    chudnovsky2(l, k, q2, t2);
    Integer t4 = t1*q2;
    t4 += t1 = t2*p1;
    p1 = q1*q2;
    t4 += p1*A;
    p1 *= SignDigit(C/12);


    t4 *= t3;
    t *= p1;
    t.lmove(sz);
    pi.value() = abs(t) / abs(t4);

    /*
    t2 = C; t2.lmove(sz*2);
    t1 = sqrt(t2);
    t2 = p1*t1;
    pi.value() = abs(t2) / abs(t4);
    */
  }
}
◇
```

Macro referenced in 437.

Laufzeit. Sei $p \in \mathsf{Pi}$, $x \in \mathsf{Natural}$, $u, v, w \in \mathsf{Integer}$ und $t \in \texttt{size\_t}$, dann

$$\texttt{Pi}(t) \sim \begin{cases} p.\texttt{schoenhage}(t,x)\ , & \beta < 32 \\ p.\texttt{stoermer}(t,x)\ , & \frac{t\beta\log_{10}(2)}{\log_{10}(\alpha)} < \frac{3}{\beta}(\gamma_{low}+3) \wedge t < 10000 \\ p.\texttt{schoenhage}(t,x)\ , & t \geq 10^8 \vee t \geq \frac{2^\beta}{6} \\ p.\texttt{chudnovsky}(0,t,u,v,w)\ , & \text{sonst} \end{cases}.$$

Die Ausgabe der Zahl π erfolgt durch die Weiterleitung an die Festkommaklasse Fixed:

⟨ puts Pi on output stream 325 ⟩ ≡

```
inline OSTREAM& operator<<(OSTREAM& out, const Pi& a)
// Algorithm:  o := o << a
// Input:      o in ostream, a in Pi.
// Output:     o in ostream ||
//
// Note:       puts Pi a on output stream.
{
  return out << a.pi;
}
◇
```

Macro referenced in 434.

Laufzeit. Sei $o \in$`ostream`, $p \in$ Pi und $f \in$ Fixed, dann o<<$p \sim$ o<<f.

6.3 Quadratwurzel

Um die Quadratwurzel von $n \in \mathbb{N}$ zu berechnen, lösen wir die folgende Pell'sche Gleichung (siehe Abschnitt 5.3 auf der Seite 276) für $p, q \in \mathbb{N}$:

$$p^2 - n \cdot p^2 = 1.$$

Hieraus erhalten wir dann durch die folgende Umformung, die Summenreihe zur Quadratwurzelberechnung:

$$\begin{aligned} \sqrt{d} &= \frac{p}{q}\sqrt{\left(1 - \frac{1}{p^2}\right)} \\ &= 1 - \sum_{n \geq 1} \left(\prod_{k=1}^{n} \frac{2k-1}{2k} \right) \frac{1}{(2n-1)p^2}. \end{aligned}$$

Sobald wir die kleinste Lösung für diese Gleichung gefunden haben, folgen alle weiteren Lösungen durch die Rekursion:

$$\begin{aligned} u_1 &= 2 \cdot u_0^2 - 1 \\ v_1 &= 2 \cdot u_0 v_0 \\ u_{n+2} &= 2 \cdot u_0 u_{n+1} - u_n \\ v_{n+2} &= 2 \cdot u_0 v_{n+1} - v_n \qquad \text{für } n \in \mathbb{N}, \end{aligned}$$

wobei $u_0, v_0 \in \mathbb{N}$ die kleinste Lösung der Pell'schen Gleichung ist.

Genauso wie bei der π-Berechnung verwenden wir nun auch hier den rekursiven Algorithmus, der von Bruno Haible und Thomas Papanikolaou in [21] beschrieben wird:

⟨ Sqrt-Series-Calculation 326 ⟩ ≡

```
void Sqrt::series(Digit n, const Digit m, Natural& p, Natural& q, Integer& t) const
// Algorithm:  z.series(n, m, p, q, t)
// Input:      z in Sqrt, p,q,t in Natural, n,m in Digit where n < GAMMA/2, n < m.
// Output:     p,q,t in Natural ||
{
  CONDITION(n < m && n < GAMMA/2);

  switch (m-n) {
```

```
    case 1: {
      p = 2*n-3;
      if (d == 0) q = u*u;
      else { q = d; q *= d; }
      q *= 2*n;
      t = p; t = -t;
      break;
    }
    case 2: {
      if (d == 0) t = q = u*u;
      else { q = d; t = q *= d; }
      q *= 2*n; t *= 2*n+2; q *= t;
      t += 2*n-1; t *= 2*n-3; t = -t;
      if (n < GAMMA/8) p = (2*n-3)*(2*n-1);
      else { p = 2*n-3; p *= 2*n-1; }
      break;
    }
    case 3: {
      if (n < GAMMA/8) p = (2*n-1)*(2*n+1);
      else { p = 2*n-1; p *= 2*n+1; }
      p *= 2*n-3;
      if (d == 0) t = q = u*u;
      else { q = d; t = q *= d; }
      q *= 2*n+2; t *= 2*n+4; q *= t;
      t += 2*n+1; t *= 2*n-1; t = -t;
      t -= q; t *= 2*n-3;
      if (d == 0) q *= u*u;
      else { q *= d; q *= d; }
      q *= 2*n;
      break;
    }
    default: {
      Natural p1,q1,p2,q2;
      Integer t1,t2;

      const Digit l = (n+m)/2;
      series(n, l, p1, q1, t1);
      series(l, m, p2, q2, t2);
      t = t1*q2; t += t1 = t2*p1;
      p = p1*p2; q = q1*q2;
    }
  }
}
```

◇

Macro defined by 326, 327.
Macro referenced in 437.

Laufzeit. Sei $z \in$ `Sqrt`, $x, y, z \in$ Natural und $t_0, t_1 \in$ Digit, dann

$$z\texttt{.series}(t_0,t_1,x,y,z) \in \mathcal{O}(\log_2(t_1 - t_0)^2 \cdot \mathcal{O}(\texttt{mul} : \mathsf{Natural}^2 \to \mathsf{Natural}).$$

⟨ Sqrt-Series-Calculation 327 ⟩ ≡

```
void Sqrt::series2(const Digit n, const Digit m, Natural& q, Integer& t) const
// Algorithm:  z.series2(n, m, q, t)
// Input:      z in Sqrt, q,t in Natural, n,m in Digit where n < m.
// Output:     q,t in Natural ||
```

```
{
  CONDITION(n < m);

  Natural p1,q1,q2;
  Integer t1,t2;

  const Digit l = (n+m)/2;
  series(n, l, p1, q1, t1);
  if (m-l <= 3) {
    Natural p2;
    series(l, m, p2, q2, t2);
  } else series2(l, m, q2, t2);
  t = t1*q2; t += t1 = t2*p1;
  q = q1*q2;
}
◇
```

Macro defined by 326, 327.
Macro referenced in 437.

Laufzeit. Sei $z \in$ Sqrt, $x, y, z \in$ Integer und $t_0, t_1 \in$ Digit, dann

$$z.\texttt{series2}(t_0,t_1,y,z) \sim p.\texttt{series}(t_0,t_1,x,y,z).$$

⟨ select Sqrt-Algorithmus 328 ⟩ ≡

```
Sqrt::Sqrt(const Digit a, const size_t n)
// Algorithm:  b := Sqrt(a, n)
// Input:      a in Digit, n in size_t.
// Output:     b in Sqrt such that |b-sqrt(a)*10^n| < 1 ||
 : sqrt(n), d(0)
{
  const size_t sz = sqrt.precision();
  Digit b,c;
  ::sqrt(a, b, c);
  if (c == 0) {
    sqrt.value() = b;
    sqrt.value().lmove(sz);
  } else {
    Natural p1,q1,q2;
    Integer t1,t2;

    pell(a, u, v);
    if (u.length() == 1) {
      Natural c = u << 1;
      Natural u0 = c*u;
      Natural v0 = c*v;
      --u0;
      if (u0.length() == 1) {
        Natural u1,v1;
        while (true) {
          u1 = c*u0; u1 -= u;
          v1 = c*v0; v1 -= v;
          u = u0; v = v0;
          if (u1.length() > 1) break;
          u0 = u1; v0 = v1;
        }
```

```
      }
    }
    if (u.length() == 1) d = u&GAMMA;

    Digit k = 2 + n/Digit((d == 0)? (2*log10(double(GAMMA))) : (2*log10(double(d))));
    if (k < 5) k = 5;
    const Digit l = (k+1)/2;

    series(2, l, p1, q1, t1);
    if (d == 0) q2 = u*u;
    else { q2 = d; q2 *= d; }
    q2 <<= 1; t1 -= q1; q1 *= q2; t1 += q1;
    series2(l, k, q2, t2);

    Integer t = t1*q2;
    t += t1 = t2*p1; t *= u;
    t2 = q1*q2; t2 *= v;
    t.lmove(sz);
    sqrt.value() = abs(t) / abs(t2);
  }
}
◇
```

Macro referenced in 437.

Die Ausgabe der Quadratwurzel erfolgt durch die Weiterleitung an die Festkommaklasse Fixed:

⟨ puts Sqrt-class on output stream 329 ⟩ ≡

```
inline OSTREAM& operator<<(OSTREAM& out, const Sqrt& a)
// Algorithm:  o := o << a
// Input:      o in ostream, a in Sqrt.
// Output:     o in ostream ||
//
// Note:       puts Sqrt a on output stream.
{
  return out << a.sqrt;
}
◇
```

Macro referenced in 434.

Laufzeit. Sei $o \in$`ostream`, $z \in$ `Sqrt` und $f \in$ Fixed, dann o<<$z \sim$ o<<f.

6.4 Apéry's Konstante

Die Apéry's Konstante ist die Riemannsche Zetafunktion an der Stelle 3, $\zeta(3)$, wobei die Riemannsche Zetafunktion folgendermaßen definiert ist:

$$\zeta(x) := \sum_{n \geq 1} \frac{1}{n^x} \text{für } x \in \mathbb{N},\ x \geq 2.$$

$\zeta(3) \approx 1.202056903\ldots$ wird erst seit 1979, aber wohl verdient als Apéry's Konstante bezeichnet, weil es ihm in diesem Jahr gelungen ist, die Irrationalität dieser Konstanten zu beweisen.
Theodor Amdeberhan und Doron Zeilberger stellten in [1] die folgende schnell konvergente Reihe vor:

$$
\begin{aligned}
\zeta(3) &= \frac{1}{2}\sum_{n\geq 1}(-1)^{n-1}\frac{205n^2-160n+32}{n^5\binom{2n}{n}^5} \\
&= \frac{1}{2}\sum_{n\geq 1}(-1)^{n-1}\frac{(205n^2-160n+32)(n!)^{10}}{n^5((2n)!)^5} \\
&= \frac{1}{2}\sum_{n\geq 1}(-1)^{n-1}\frac{205n^2-160n+32}{n^5}\left(\prod_{k=1}^{n}\frac{k\cdot k}{k(n+k)}\right)^5 \\
&= \frac{1}{2}\sum_{n\geq 1}(-1)^{n-1}\frac{205n^2-160n+32}{n^5}\left(\prod_{k=1}^{n}\frac{k\cdot k}{2k(2k-1)}\right)^5 \\
&= \frac{1}{2}\sum_{n\geq 1}(-1)^{n-1}\frac{(205n^2-160n+32)(n!)^5}{n^5\cdot 32^n\cdot\prod_{k=1}^{n}(2k-1)^5} \\
&= \frac{1}{2}\sum_{n\geq 0}(-1)^{n}\frac{(205(n+1)^2-160(n+1)+32)((n+1)!)^5}{(n+1)^5\cdot 32^{n+1}\cdot\prod_{k=1}^{n+1}(2k-1)^5} \\
&= \frac{1}{64}\sum_{n\geq 0}(-1)^{n}\frac{(205n^2+250n+77)(n!)^5}{32^n\cdot\prod_{k=0}^{n}(2k+1)^5}
\end{aligned}
$$

Eine direkte Implementierung der linear konvergienden Reihe ist bis zu 4000 dezimalen Stellen effizient und schnell realisiert:

⟨ Zeta(3)-Calculation 330 ⟩ ≡

```
void Zeta3::linear(const size_t n, Natural& z) const
// Algorithm:  b.linear(n, a)
// Input:      b in Zeta3, a in Natural, n in size_t where b.decimals()/574 <= GAMMA.
// Output:     a in Natural such that |a-Zeta(3)*2^(BETA*n)| < 1 ||
{
  CONDITION((zeta.decimals()/1.4)/410 <= GAMMA);

  Natural a = 1;
  Natural c = 77;
  Natural t;

  a <<= n*BETA+2;
  z = 0;
  Digit k = 1;
  do {
    t = a * c;
    if (k&1) z += t;
    else z -= t;

    c += 410*k+45;
    if (k <= GAMMA_LOW) { a *= k*k; a *= k*k; }
    else { a *= k; a *= k; a *= k; a *= k; }
    a *= k;
    const Digit i = 2*k+1;
    if (i <= GAMMA_LOW) { a /= i*i; a /= i*i; }
    else { a /= i; a /= i; a /= i; a /= i; }
    a /= 32*i;
    ++k;
```

```
  } while (a != 0);
  z >>= 8;
}
◇
```

Macro referenced in 437.

Laufzeit. Sei $z \in$ `Zeta3`, $x \in$ Natural und $t \in$ `size_t`, dann

$$z\texttt{.linear(}t\texttt{,}x\texttt{)} \in \mathcal{O}(\texttt{operator/=} : \mathsf{Natural} \times \mathsf{Digit} \to \mathsf{Natural}).$$

Für den rekursiven Algorithmus verwenden wir eine etwas schneller konvergierende Formel [1]

$$\zeta(3) = \frac{1}{24} \sum_{n \geq 0} (-1)^n \frac{A(n) \cdot ((2n+1)!(2n)!n!)^3}{(3n+2)!((4n+3)!)^3}$$

mit $A(n) := 126392n^5 + 412708n^4 + 531578n^3 + 336367n^2 + 104000n + 12463$.

⟨ Zeta(3)-Series-Calculation 331 ⟩ ≡

```
void Zeta3::series(Digit n, const Digit m, Integer& p, Integer& q, Integer& t) const
// Algorithm:  z.series(n, m, p, q, t)
// Input:      z in Zeta3, p,q,t in Integer, n,m in Digit where n < 2^BETA/64-32, n < m.
// Output:     p,q,t in Integer ||
{
  CONDITION(n < m && n < ((size_t(1) << (BETA-1))/64)-32);

  if (n+1 == m) {
    if (n <= GAMMA_LOW/2) { p = n*n; p *= n*n; }
    else { p = n; p = p*p; p = p*p; }
    const Digit k = 2*n-1;
    if (n <= GAMMA_LOW/8) { p *= k*n; p *= k*k; }
    else { p *= n; p *= k; p *= k; p *= k; }
    p = -p;
    t = 126392; t *= n;
    t += 412708; t *= n;
    t += 531578; t *= n;
    t += 336367; t *= n;
    t += 104000; t *= n;
    t += 12463; t *= p;

    const Digit k2 = 4*n+1;
    const Digit k3 = k2+2;
    if (k3 <= GAMMA_LOW/2) { q = k2*k2; q *= k2*k3; q *= k3*k3; }
    else {
      q = k2; q *= k2; q *= k2;
      q *= k3; q *= k3; q *= k3;
    }
    q *= 72*n+24; q *= 3*n+2;
  } else {
    Integer p1,q1,t1;
    Integer p2,q2,t2;

    const Digit l = (n+m)/2;
    series(n, l, p1, q1, t1);
    series(l, m, p2, q2, t2);
    t = t1*q2; t += t1 = t2*p1;
```

```
    p = p1*p2; q = q1*q2;
  }
}
◇
```

Macro defined by 331, 332a.
Macro referenced in 437.

Laufzeit. Sei $z \in$ `Zeta3`, $u, v, w \in$ Integer und $t_0, t_1 \in$ Digit, dann

$$z.\texttt{series}(t_0,t_1,u,v,w) \in \mathcal{O}(\log_2(t_1 - t_0)^2 \cdot \mathcal{O}(\texttt{mul} : \text{Integer}^2 \to \text{Integer}).$$

⟨ Zeta(3)-Series-Calculation 332a ⟩ ≡

```
void Zeta3::series2(const Digit n, const Digit m, Integer& q, Integer& t) const
// Algorithm:  z.series2(n, m, q, t)
// Input:      z in Zeta3, q,t in Integer, n,m in Digit where n < m.
// Output:     q,t in Integer ||
{
  CONDITION(n < m);

  Integer p1,q1,t1;
  Integer q2,t2;

  const Digit l = (n+m)/2;
  series(n, l, p1, q1, t1);
  if (m-l <= 1) {
    Integer p2;
    series(l, m, p2, q2, t2);
  } else series2(l, m, q2, t2);
  t = t1*q2; t += t1 = t2*p1;
  q = q1*q2;
}
◇
```

Macro defined by 331, 332a.
Macro referenced in 437.

Laufzeit. Sei $z \in$ `Zeta3`, $u, v, w \in$ Integer und $t_0, t_1 \in$ Digit, dann

$$z.\texttt{series2}(t_0,t_1,v,w) \sim p.\texttt{series}(t_0,t_1,u,v,w).$$

Im Konstruktor der Klasse `Zeta3` wählen wir bei einer kleineren Anzahl an dezimalen Nachkommastellen von $\zeta(3)$ die lineare Implementation `linear` aus, anderenfalls die rekursive Implementation `series`:

⟨ select $\zeta(3)$-Algorithm 332b ⟩ ≡

```
Zeta3::Zeta3(const size_t n)
// Algorithm:  a := Zeta3(n)
// Input:      n in size_t.
// Output:     a in Zeta3 such that |a-zeta(3)*10^n| < 1 ||
 : zeta(n)
{
  const size_t sz = zeta.precision();

  if (n <= 4000 && n/574 <= GAMMA) linear(sz, zeta.value());
  else {
    Integer p1,q1,t1;
    Integer q2,t2;
```

```
    const size_t k = (n >= 100000)? size_t(n/5.03) : n/5;
    const size_t l = (k+1)/2;
    series(1, l, p1, q1, t1);
    series2(l, k, q2, t2);
    Integer t = t1*q2;
    t += t1 = t2*p1;
    t2 = q1*q2;
    t += t2*SignDigit(12463); t2 *= SignDigit(10368);
    size_t i = 0;
    if (t2 != 0)
      while (!abs(t2).testbit(i)) ++i;
    t2 >>= i; t.lmove(sz);
    zeta.value() = abs(t) / abs(t2);
    zeta.value() >>= i;
  }
}
◇
```

Macro referenced in 437.

Die Ausgabe der Zahl $\zeta(3)$ erfolgt durch die Weiterleitung an die Festkommaklasse Fixed:

⟨ puts Zeta3-class on output stream 333a ⟩ ≡

```
inline OSTREAM& operator<<(OSTREAM& out, const Zeta3& a)
// Algorithm:  o := o << a
// Input:      o in ostream, a in Zeta3.
// Output:     o in ostream ||
//
// Note:       puts Zeta3 a on output stream.
{
  return out << a.zeta;
}
◇
```

Macro referenced in 434.

Laufzeit. Sei $o \in$ostream, $z \in$ Zeta3 und $f \in$ Fixed, dann o<<$z \sim$ o<<f.

6.5 Eulersche Zahl

Eine direkte Implementierung der linear konvergienden Reihe ist bis zu 4000 dezimalen Stellen effizient und schnell realisiert:

⟨ Exp(1)-Calculation 333b ⟩ ≡

```
void Exp1::linear(const size_t n, Natural& z) const
// Algorithm:  b.linear(n, a)
// Input:      b in Exp1, a in Natural, n in size_t.
// Output:     a in Natural such that |a-Exp(1)*2^(BETA*n)| < 1 ||
{
  Natural a = 1;
  a <<= n*BETA;
  z = a << 1;
  Digit k = 2;
  do {
```

```
    a /= k;
    z += a;
    ++k;
  } while (a != 0);
}
```
◇

Macro referenced in 437.

Laufzeit. Sei $z \in$ `Exp1`, $x \in$ Natural und $t \in$ `size_t`, dann

$$z.\texttt{linear}(t,x) \in \mathcal{O}(\texttt{operator/=} : \text{Natural} \times \text{Digit} \to \text{Natural}).$$

Genauso wie bei der π-Berechnung verwenden wir nun auch hier den rekursiven Algorithmus, der von Bruno Haible und Thomas Papanikolaou in [21] beschrieben wird:

⟨ Exp(1)-Series-Calculation 334 ⟩ ≡

```
void Exp1::series(Digit n, const Digit m, Natural& q, Natural& t) const
// Algorithm:  e.series(n, m, q, t)
// Input:      e in Exp1, q,t in Natural, n,m in Digit where n < m.
// Output:     q,t in Natural ||
{
  CONDITION(n < m);

  switch (m-n) {
    case 1: {
      q = (n == 0)? 1 : n;
      t = 1;
      break;
    }
    case 2: {
      const Digit l = (n+m)/2;
      if (n == 0) q = l;
      else if (l < GAMMA_LOW) q = n*l;
      else { q = n; q *= l; }
      t = l+1;
      break;
    }
    case 3: {
      const Digit l = (n+m)/2;
      const Digit l2 = (l+m)/2;
      if (l2 < GAMMA_LOW) {
        const Digit x = l*l2;
        t = x; q = x;
      } else {
        t = l; t *= l2; q = t;
      }
      if (n != 0) q *= n;
      t += l2+1;
      break;
    }
    default: {
      Natural q1,t1;
      Natural q2,t2;

      const Digit l = (n+m)/2;
      series(n, l, q1, t1);
```

```
      series(l, m, q2, t2);
      t = t1*q2; t += t2;
      q = q1*q2;
      break;
    }
  }
}
◇
```

Macro referenced in 437.

Laufzeit. Sei $z \in$ `Exp1`, $u, v, w \in$ Integer und $t_0, t_1 \in$ `size_t`, dann

$$z.\texttt{series}(t_0, t_1, u, v, w) \in \mathcal{O}(\log_2(t_1 - t_0)^2 \cdot \mathcal{O}(\texttt{mul} : \text{Integer}^2 \to \text{Integer}).$$

Im Konstruktor der Klasse `Exp1` wählen wir bei einer kleineren Anzahl an dezimalen Nachkommastellen von Exp(1) die lineare Implementation `linear` aus, anderenfalls die rekursive Implementation `series`:

⟨ select Exp(1)-Algorithm 335 ⟩ ≡

```
Exp1::Exp1(const size_t n)
// Algorithm:  a := Exp1(n)
// Input:      n in size_t.
// Output:     a in Exp1 such that |a-exp(1)*10^n| < 1 ||
 : exp(n)
{
  const size_t sz = exp.precision();

  if (n <= 4000) linear(sz, exp.value());
  else {
    Natural t;
    // n < (m + 1/2)*log10(double(m)) - 11/25*m
    Digit m = Digit(n/4.239174208864);
    Digit m2 = m/2;
    Digit d = Digit(log10(double(m)) * (m+0.5) - (m/25.0)*11);
    while (d <= n) {
      m *= 2;
      d = Digit(log10(double(m)) * (m+0.5) - (m/25.0)*11);
    }
    Digit d2 = Digit(log10(double(m2)) * (m2+0.5) - (m2/25.0)*11);
    while (d2 > n) {
      m2 /= 2;
      d2 = Digit(log10(double(m2)) * (m2+0.5) - (m2/25.0)*11);
    }
    while (m-m2 > 1) {
      Digit m3 = (m+m2)/2;
      Digit d3 = Digit(log10(double(m3)) * (m3+0.5) - (m3/25.0)*11);
      if (d3 > n) m = m3;
      else m2 = m3;
    }
    series(0, m, ((Natural&)*this), t);
    size_t i = 0;
    if (((Natural&)*this) != 0) {
      for (const Digit* p = last(); *p == 0 && i < BETA*sz; --p) i += BETA;
      while (!testbit(i) && i < BETA*sz) ++i;
    }
    ((Natural&)*this) >>= i; t <<= BETA*sz-i;
    exp.value() = t / ((Natural&)*this);
  }
```

```
}
◇
```

Macro referenced in 437.

Die Ausgabe der Zahl e erfolgt durch die Weiterleitung an die Festkommaklasse Fixed:

⟨ puts Exp1-class on output stream 336a ⟩ ≡

```
inline OSTREAM& operator<<(OSTREAM& out, const Exp1& a)
// Algorithm:  o := o << a
// Input:      o in ostream, a in Exp1.
// Output:     o in ostream ||
//
// Note:       puts Exp1 a on output stream.
{
  return out << a.exp;
}
◇
```

Macro referenced in 434.

Laufzeit. Sei $o \in$`ostream`, $z \in$ `Exp1` und $f \in$ Fixed, dann o<<$z \sim$ o<<f.

6.6 Natürlicher Logarithmus

Für den natürlichen Logarithmus von 2 gilt die folgende Formel

$$\ln(2) = 144\text{atanh}(\tfrac{1}{255}) + 54\text{atanh}(\tfrac{1}{449}) + 24\text{atanh}(\tfrac{1}{4801}) + 20\text{atanh}(\tfrac{1}{31751}) + 82\text{atanh}(\tfrac{1}{32257}).$$

⟨ Arctanh-Calculation 336b ⟩ ≡

```
Natural Ln::atanh_inv_linear(const Digit a, const size_t n) const
// Algorithm:  c := b.atanh_inv_linear(a, n)
// Input:      a in Digit, b in Ln, n in size_t.
// Output:     c in Natural such that |c-atanh(1/a)*2^(BETA*n)| < 1 ||
{
  Natural c = Digit(0);
  Natural s,t = a;
  t <<= BETA*n;
  Digit k = 1;
  if (a < GAMMA_LOW) {
    const Digit b = a*a;
    do {
      t /= b;
      c += s = t / k;
      k += 2;
    } while (s != 0);
  } else {
    do {
      t /= a; t /= a;
      c += s = t / k;
      k += 2;
    } while (s != 0);
  }
  return c;
}
```

```
void Ln::atanh_inv_series(Digit n, const Digit m,
                          Natural& b, Natural& q, Natural& t) const
// Algorithm:  l.atanh_inv_series(n, m, b, q, t)
// Input:      l in Ln, b,q,t in Natural, n,m in Digit where n < m.
// Output:     b,q,t in Natural ||
{
  CONDITION(n < m);

  if (n+1 == m) {
    b = 2*n+1;
    t = 1;
    if (n == 0) q = x;
    else if (x < GAMMA_LOW) q = x*x;
    else { q = x; q *= q; }
  } else {
    Natural b1,q1,t1;
    Natural b2,q2,t2;
    const Digit l = (n+m)/2;
    atanh_inv_series(n, l, b1, q1, t1);
    atanh_inv_series(l, m, b2, q2, t2);
    q = q1*q2; b = b1*b2;
    t = t2*b1; b1 = b2*t1; t += b2 = b1*q2;
  }
}

Natural Ln::atanh_inv_series(const Digit a)
// Algorithm:  c := b.atanh_inv_linear(a)
// Input:      a in Digit, b in Ln.
// Output:     c in Natural such that |c-atanh(1/a)*2^(BETA*b.precision())| < 1 ||
{
  const size_t n = ln.precision();
  Digit m = Digit(ln.decimals()/log10(double(a)))+1;
  x = a;
  Natural b,q,t;
  atanh_inv_series(0, m, b, q, t);
  size_t i = 0;
  if (q != 0)
    while (!q.testbit(i)) ++i;
  size_t j = 0;
  if (b != 0)
    while (!b.testbit(j)) ++j;
  q >>= i; b >>= j; t <<= BETA*n-i-j;
  q *= b;
  b = t / q;
  return b;
}
◇
```

Macro referenced in 437.

⟨ Ln(2)-Calculation 337 ⟩ ≡

```
Natural Ln::ln2(const size_t n)
// Algorithm:  c := b.ln2(n)
// Input:      b in Ln, n in size_t.
// Output:     c in Natural such that |c-ln(2)*2^(BETA*n)| < 1 ||
{
```

```
  if (n <= 400) {
    Natural ln = 144*atanh_inv_linear(255, n);
    ln += 54*atanh_inv_linear(449, n);
    ln += 24*atanh_inv_linear(4801, n);
    ln += 20*atanh_inv_linear(31751, n);
    return ln += 82*atanh_inv_linear(32257, n);
  }
  Natural ln = 144*atanh_inv_series(255);
  ln += 54*atanh_inv_series(449);
  ln += 24*atanh_inv_series(4801);
  ln += 20*atanh_inv_series(31751);
  return ln += 82*atanh_inv_series(32257);
}
◇
```

Macro referenced in 437.

⟨ Ln-Calculation 338a ⟩ ≡

```
void Ln::linear(const size_t n, Natural& ln) const
// Algorithm:  a.linear(n, b)
// Input:      a in Ln, b in Natural, n in size_t.
// Output:     b in Natural such that |b-ln(a.u/2^a.v)*2^(BETA*n)| < 1 ||
{
  Natural s,t = u;
  t <<= BETA*n-v;
  Digit k = 2;
  ln = t;
  do {
    t *= u; t >>= v;
    s = t / k;
    if (k&1) ln += s;
    else ln -= s;
    ++k;
  } while (s != 0);
}
◇
```

Macro referenced in 437.

⟨ Ln-Series-Calculation 338b ⟩ ≡

```
void Ln::series(Digit n, const Digit m,
                  Natural& b, Integer& p, Digit& q, Integer& t) const
// Algorithm:  l.series(n, m, b, p, q, t)
// Input:      l in Ln, b,q in Natural, p,t in Integer, n,m in Digit where n < m.
// Output:     b,q in Natural, p,t in Integer ||
{
  CONDITION(n < m);

  if (n+1 == m) {
    b = n+1;
    p = u; q = v;
    if (n != 0) p = -p;
    t = p;
  } else {
    Natural b1,b2;
    Digit   q1,q2;
    Integer p1,t1,p2,t2;
```

```
    const Digit l = (n+m)/2;
    series(n, l, b1, p1, q1, t1);
    series(l, m, b2, p2, q2, t2);
    p = p1*p2; q = q1+q2; b = b1*b2;
    p2 = t2*b1; t = p2*p1;
    b1 = b2 << q2; t += p2 = b1*t1;
  }
}
◇
```

Macro referenced in 437.

Seien $a, u, v, s \in \mathbb{N}$, so daß gilt

$$a = \frac{u}{v} \cdot 2^s \text{ und } \frac{1}{2} \leq \frac{u}{v} < 1.$$

Dann können wir den natürlichen Logarithmus anhand der folgenden Gleichung lösen:

$$\begin{aligned} \ln(a) &= \ln(\tfrac{u}{v}) + s \cdot \ln(2) \\ &= \ln(x-1) + s \cdot \ln(2) \end{aligned}$$

mit $x = 1 + \frac{u}{v}$ und $|x-1| < 1$.

⟨ select Ln-Algorithm 339 ⟩ ≡

```
Ln::Ln(const Digit a, const size_t n)
// Algorithm:  b := Ln(a, n)
// Input:      a in Digit, n in size_t.
// Output:     b in Ln such that |b-ln(a)*10^n| < 1 ||
 : ln(n)
{
  const size_t sz = ln.precision();
  switch (a) {
    case 0: ln.value().errmsg(0, "value undefined!"); break;
    case 1: ln.value() = 0; break;
    case 2: ln.value() = ln2(sz); break;
    default: {
      v = log2(a);
      u = a - (Digit(1) << v);
      if (sz < 4000) linear(sz, ln.value());
      else {
        const Digit m = Digit(n/log10((Digit(1) << v)/double(u)))+1;
        Integer p,t;
        Digit q;
        series(0, m, (Natural&)*this, p, q, t);
        size_t i = 0;
        if (((Natural&)*this) != 0) {
          for (const Digit* p = last(); *p == 0; --p) i += BETA;
          while (!testbit(i)) ++i;
        }
        ((Natural&)*this) >>= i;
        const size_t j = BETA*sz;
        if (i >= j) t >>= i-j+q;
        else {
          i = j-i;
          if (i >= q) t <<= i-q;
          else t >>= q-i;
```

```
        }
        ln.value() = ::abs(t) / ((Natural&)*this);
      }
      ln.value() += v*ln2(sz);
      break;
    }
  }
}
◇
```

Macro referenced in 437.

⟨ puts Ln-class on output stream 340a ⟩ ≡

```
inline OSTREAM& operator<<(OSTREAM& out, const Ln& a)
// Algorithm:  o := o << a
// Input:      o in ostream, a in Ln.
// Output:     o in ostream ||
//
// Note:       puts Ln a on output stream.
{
  return out << a.ln;
}
◇
```

Macro referenced in 434.

Laufzeit. Sei $o \in$ostream, $z \in$ Ln und $f \in$ Fixed, dann o<<$z \sim$ o<<f.

6.7 Eulersche Konstante

⟨ EulerGamma-Calculation 340b ⟩ ≡

```
void EulerGamma::linear(const size_t n, Natural& euler)
// Algorithm:  a.linear(n, b)
// Input:      a in EulerGamma, b in Natural, n in size_t.
// Output:     b in Natural such that |b-gamma*2^(BETA*n)| < 1 ||
{
  const Digit a = Digit(0.17328679514*n*BETA)+1;  // ln(2)/4
  const Digit m = Digit(3.591121477*a);

  Natural f = 1;
  Natural g = Digit(0);
  f.lmove(n);
  Natural s = f;
  Natural s2 = Digit(0);
  if (a < GAMMA_LOW) {
    const Digit b = a*a;
    if (m < GAMMA_LOW) {
      for (Digit k = 1; k < m; ++k) {
        f *= b; g *= b;
        f /= k*k; g /= k; g += f; g /= k;
        s += f; s2 += g;
      }
    } else {
      Digit k;
      for (k = 1; k < GAMMA_LOW; ++k) {
```

```
        f *= b; g *= b;
        f /= k*k; g /= k; g += f; g /= k;
        s += f; s2 += g;
      }
      while (k < m) {
        f *= b; g *= b;
        f /= k; f /= k;
        g /= k; g += f; g /= k;
        s += f; s2 += g;
        ++k;
      }
    }
  } else {
    if (m < GAMMA_LOW) {
      for (Digit k = 1; k < m; ++k) {
        f *= a; f *= a; g *= a; g *= a;
        f /= k*k; g /= k; g += f; g /= k;
        s += f; s2 += g;
      }
    } else {
      Digit k;
      for (k = 1; k < GAMMA_LOW; ++k) {
        f *= a; f *= a; g *= a; g *= a;
        f /= k*k; g /= k; g += f; g /= k;
        s += f; s2 += g;
      }
      while (k < m) {
        f *= a; f *= a; g *= a; g *= a;
        f /= k; f /= k;
        g /= k; g += f; g /= k;
        s += f; s2 += g;
        ++k;
      }
    }
  }
  s2.lmove(n);
  euler = s2/s;
  Ln l(a, this->euler.decimals());
  euler -= l.value();
}
◇
```

Macro referenced in 437.

⟨ EulerGamma-Series-Calculation 341 ⟩ ≡

```
void EulerGamma::series(Digit n, const Digit m,
                        Natural& p, Natural& q, Natural& t,
                        Natural& c, Natural& d, Natural& v) const
// Algorithm:  e.series(n, m, p, q, t, c, d, v)
// Input:      e in EulerGamma, p,q,t,c,d,v in Natural, n,m in Digit where n < m.
// Output:     p,q,t,c,d,v in Natural ||
{
  CONDITION(n < m);

  switch (m-n) {
    case 1: {
      const Digit l = n+1;
```

```
  if (l < GAMMA_LOW) q = l*l;
  else { q = l; q *= l; }
  v = t = p = ((Natural&)*this); c = 1; d = l;
  break;
}
case 2: {
  const Digit l1 = n+1;
  const Digit l2 = n+2;
  const Digit l3 = l1+l2;
  if (l2 < GAMMA_LOW) { d = q = l1*l2; t = ((Natural&)*this)*(l2*l2); }
  else { q = l1; d = q *= l2; t = ((Natural&)*this)*l2; t *= l2; }
  q *= q; t += p = ((Natural&)*this)*((Natural&)*this);
  if (l3 >= l1) c = l3;
  else { c = l1; c += l2; }
  v = l2*t; v += l1*p;
  break;
}
case 3: {
  const Digit l1 = n+1;
  const Digit l2 = n+2;
  const Digit l3 = n+3;
  const Digit l4 = l1+l2;
  Natural s = ((Natural&)*this)*((Natural&)*this);
  v = l1*s; p = s*((Natural&)*this);
  if (l3 < GAMMA_LOW) {
    d = q = l1*l2; q *= l3;
    s += ((Natural&)*this)*(l2*l2); c = l4*l3;
    v += s*l2;
    const Digit l5 = l3*l3;
    t = l5*s; v *= l5; v += l4*p;
  } else {
    q = l1; d = q *= l2; q *= l3;
    s += (((Natural&)*this)*l2)*l2;
    v += s*l2;
    const Digit l5 = l3*l3;
    t = l5*s; v *= l5;
    if (l4 >= l1) { c = l4; v += l4*p; }
    else { c = l1; c += l2; v += c*p; }
    c *= l3;
  }
  q *= q; c += d; t += p;
  v *= l3; v += d*p; d *= l3;
  break;
}
default: {
  Natural p1,q1,t1,c1,d1,v1;
  Natural p2,q2,t2,c2,d2,v2;

  const Digit l = (n+m)/2;
  series(n, l, p1, q1, t1, c1, d1, v1);
  series(l, m, p2, q2, t2, c2, d2, v2);
  p = p1*p2; q = q1*q2; p2 = p1*t2;
  t = t1*q2; t += p2;
  c = c1*d2; d = d1*d2;
  p2 *= c1; q1 = c2*d1;
  c += q1;
```

```
      q1 = q2*v1; q1 += p2;
      v = d2*q1; q1 = d1*p1; p2 = q1*v2; v += p2;
      break;
    }
  }
}

void EulerGamma::series2(Digit n, const Digit m,
                         Natural& q, Natural& t, Natural& d, Natural& v) const
// Algorithm:  e.series2(n, m, q, t, d, v)
// Input:      e in EulerGamma, q,t,d,v in Natural, n,m in Digit where n < m.
// Output:     q,t,d,v in Natural ||
{
  CONDITION(n < m);

  switch (m-n) {
    case 1: {
      const Digit l = n+1;
      if (l < GAMMA_LOW) q = l*l;
      else { q = l; q *= l; }
      v = t = ((Natural&)*this); d = l;
      break;
    }
    case 2: {
      const Natural p = ((Natural&)*this)*((Natural&)*this);
      const Digit l1 = n+1;
      const Digit l2 = n+2;
      if (l2 < GAMMA_LOW) { d = q = l1*l2; t = ((Natural&)*this)*(l2*l2); }
      else { q = l1; d = q *= l2; t = ((Natural&)*this)*l2; t *= l2; }
      q *= q; t += p;
      v = l2*t; v += l1*p;
      break;
    }
    case 3: {
      const Digit l1 = n+1;
      const Digit l2 = n+2;
      const Digit l3 = n+3;
      const Digit l4 = l1+l2;
      Natural s = ((Natural&)*this)*((Natural&)*this);
      Natural p = s*((Natural&)*this);
      v = l1*s;
      if (l3 < GAMMA_LOW) {
        d = q = l1*l2; q *= l3;
        s += ((Natural&)*this)*(l2*l2);
        v += s*l2;
        const Digit l5 = l3*l3;
        t = l5*s; v *= l5; v += l4*p;
      } else {
        q = l1; d = q *= l2; q *= l3;
        s += (((Natural&)*this)*l2)*l2;
        v += s*l2;
        const Digit l5 = l3*l3;
        t = l5*s; v *= l5;
        if (l4 >= l1) v += l4*p;
        else { s = l1; s += l2; v += s*p; }
      }
```

```
      q *= q; t += p;
      v *= l3; v += d*p; d *= l3;
      break;
    }
    default: {
      Natural p1,q1,t1,c1,d1,v1;
      Natural q2,t2,d2,v2;

      const Digit l = (n+m)/2;
      series(n, l, p1, q1, t1, c1, d1, v1);
      series2(l, m, q2, t2, d2, v2);
      q = q1*q2; d = d1*d2;
      t = t1*q2; v = p1*t2; t += v; c1 *= v;
      q1 = q2*v1; q1 += c1;
      v = d2*q1; d2 = d1*p1; q1 = d2*v2; v += q1;
      break;
    }
  }
}
◇
```

Macro referenced in 437.

⟨ select EulerGamma-Algorithm 344 ⟩ ≡

```
EulerGamma::EulerGamma(const size_t n)
// Algorithm:  a := EulerGamma(n)
// Input:      n in size_t.
// Output:     a in EulerGamma such that |a-gamma*10^n| < 1 ||
 : euler(n)
{
  const size_t sz = euler.precision();

  if (n <= 10000) linear(sz, euler.value());
  else {
    const Digit a = Digit(0.17328679514*sz*BETA)+1;  // ln(2)/4
    Digit m = Digit(3.591121477*a);
    ((Natural&)*this) = a; ((Natural&)*this) *= a;

    Natural* q = new Natural(); // To reduce memory consumption
    Natural* t = new Natural();
    Natural* d = new Natural();
    Natural* v = new Natural();
    series2(0, m, *q, *t, *d, *v);
    *q += *t;
    delete t;
    ((Natural&)*this) = (*q)*(*d);
    delete d;
    delete q;
    size_t i = 0;
    if (((Natural&)*this) != 0) {
      for (const Digit* p = last(); *p == 0 && i < BETA*sz; --p) i += BETA;
      while (!testbit(i) && i < BETA*sz) ++i;
    }
    ((Natural&)*this) >>= i; *v <<= BETA*sz-i;

    euler.value() = *v / ((Natural&)*this);
    delete v;
```

```
    Ln ln(a, euler.decimals());
    euler.value() -= ln.value();
  }
}
◇
```

Macro referenced in 437.

⟨ puts EulerGamma-class on output stream 345a ⟩ ≡

```
inline OSTREAM& operator<<(OSTREAM& out, const EulerGamma& a)
// Algorithm:  o := o << a
// Input:      o in ostream, a in EulerGamma.
// Output:     o in ostream ||
//
// Note:       puts EulerGamma a on output stream.
{
  return out << a.euler;
}
◇
```

Macro referenced in 434.

Laufzeit. Sei $o \in$`ostream`, $z \in$ `EulerGamma` und $f \in$ Fixed, dann o<<$z \sim$ o<<f.

6.8 Einfaches Programm zur Berechnung der Konstanten

"constant.cpp" 345b ≡

```
////////////////////////////
//
// Piologie V1.3
//
// (c) Sebastian Wedeniwski
// 08/13/1999
//
// Note: Timings on Pentium II 266 MHz, Windows NT 4.0, MS Visual C++ 6.0
// constant 1 131072
// Computing time [s] = 17.955
//
// constant 1 1048576
// official: 270 sec
// Computing time [s] = 239.264


#include <time.h>
#include "pi.h"

#ifndef _Old_STD_
# include <fstream>
using namespace std;
#else
# include <fstream.h>
#endif

void calcPi(Digit n)
{
```

```
  clock_t start,stop;
  double  duration;

  start = clock();
  Pi pi(n);
  stop = clock();
  duration = double(stop-start)/CLOCKS_PER_SEC;
  cout << "Computing time [s] = " << duration << endl;

  cout << "Writing result to file pi.txt ..." << endl;
  ofstream fout("pi.txt");
  fout << pi;
  stop = clock();
}

void calcExp1(Digit n)
{
  clock_t start,stop;
  double  duration;

  start = clock();
  Exp1 exp(n);
  stop = clock();
  duration = double(stop-start)/CLOCKS_PER_SEC;
  cout << "Computing time [s] = " << duration << endl;

  cout << "Writing result to file exp1.txt ..." << endl;
  ofstream fout("exp1.txt");
  fout << exp;
  stop = clock();
}

void calcZeta3(Digit n)
{
  clock_t start,stop;
  double  duration;

  start = clock();
  Zeta3 zeta(n);
  stop = clock();
  duration = double(stop-start)/CLOCKS_PER_SEC;
  cout << "Computing time [s] = " << duration << endl;

  cout << "Writing result to file zeta3.txt ..." << endl;
  ofstream fout("zeta3.txt");
  fout << zeta;
  stop = clock();
}

void calcGamma(Digit n)
{
  clock_t start,stop;
  double  duration;

  start = clock();
  EulerGamma gamma(n);
```

```
  stop = clock();
  duration = double(stop-start)/CLOCKS_PER_SEC;
  cout << "Computing time [s] = " << duration << endl;

  cout << "Writing result to file gamma.txt ..." << endl;
  ofstream fout("gamma.txt");
  fout << gamma;
  stop = clock();
}

void calcLn2(Digit n)
{
  clock_t start,stop;
  double  duration;

  start = clock();
  Ln ln(2, n);
  stop = clock();
  duration = double(stop-start)/CLOCKS_PER_SEC;
  cout << "Computing time [s] = " << duration << endl;

  cout << "Writing result to file ln2.txt ..." << endl;
  ofstream fout("ln2.txt");
  fout << ln;
  stop = clock();
}


int main(int argc, char** argv)
{
  if (argc != 3) {
    cerr << "USAGE: " << argv[0]
         << " <constant> <digits>\n1: pi\n2: exp(1)\n3: zeta(3)\n4: gamma\n5: ln(2)"
         << endl;
    return 1;
  }
  switch (atoi(argv[1])) {
    case 2:   calcExp1(atoi(argv[2])); break;
    case 3:   calcZeta3(atoi(argv[2])); break;
    case 4:   calcGamma(atoi(argv[2])); break;
    case 5:   calcLn2(atoi(argv[2])); break;
    default:  calcPi(atoi(argv[2]));
  }

  return 0;
}
```

◇

Anhang A

Notation und Symbolik

Datentypen

	Datentypen	C++-Notation	auf Seite
(1)	Digit	`typedef unsigned long Digit;`	3
(2)	`size_t`	`#include <stdlib.h>`	4
(3)	String	synonym für `char*` (Null-terminierender String)	41

Konstanten

	Konstanten	C++-Notation	auf Seite
(4)	β	`const size_t BETA = sizeof(Digit)*CHAR_BIT;`	3
(5)	$\gamma := 2^{\beta} - 1$	`const Digit GAMMA = ~Digit(0);`	3
(6)	`ALPHA_WIDTH`	`const size_t ALPHA_WIDTH = size_t(log10(GAMMA+1.0));`	4
(7)	$\alpha := 10^{\lfloor \log_{10}(2^{\beta}) \rfloor}$	`const Digit ALPHA = pow10(ALPHA_WIDTH);`	4

Grundmengen

auf Seite

(8) Die Menge aller natürlichen Zahlen:
$\mathbb{N} := \{0,1,2,3,4,...\}$ 15

(9) Die Menge aller positiven natürlichen Zahlen:
$\mathbb{N}_+ := \mathbb{N} - \{0\}$

(10) Die Menge aller ganzen Zahlen:
$\mathbb{Z} := \{x \mid \ x \in \mathbb{N}\} \cup \{-x \mid \ x \in \mathbb{N}\} = \mathbb{N} \cup -\mathbb{N}$ 186

(11) Die multiplikative Gruppe (Einheitengruppe) der ganzen Zahlen:
$\mathbb{Z}^* := \{-1, 1\}$

(12) Die Menge aller reellen Zahlen:
$\mathbb{R}$

(13) Die Menge aller reellen positiven Zahlen:
$\mathbb{R}_{>0} := \{x \in \mathbb{R} \mid \ x > 0\}$

(14) Die Menge aller Primzahlen:
$\mathbb{P} := \{p \in \mathbb{N} \mid \ \forall_{x,y \in \mathbb{N}_+} : (p|xy \Rightarrow \ p|x \vee p|y)\} - \{1\}$ 283

(15) Die Menge aller Pseudo-Primzahlen:
$\mathbb{P}' := \{p \in \mathbb{N} \mid \ p \equiv q \pmod{30}, q \in \{1, 7, 11, 13, 17, 19, 23, 29\}\}\dot\cup\{2, 3, 5\}$ 293

(16) Die Klasse Natural:
$\mathsf{Natural} \subset \mathbb{N}$ 15

(17) Die Klasse Integer:
$\mathsf{Integer} := \mathsf{Natural} \cup -\mathsf{Natural} \subset \mathbb{Z}$ 186

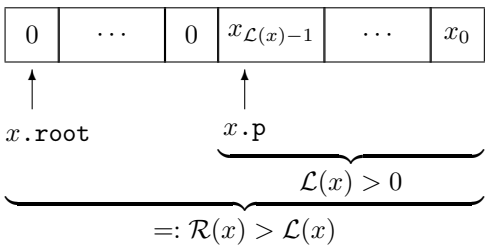
Längeninformation

(18) $\mathcal{L} : \mathsf{Natural} \to \texttt{size\_t} : x \mapsto \begin{cases} \lfloor \log_{2^\beta} x \rfloor + 1\,, & x > 0 \\ 1\,, & x = 0 \end{cases}$, auf Seite 4

Zahlendarstellung

(19) Sei $x \in \mathsf{Natural}$, dann gilt

$$x = \sum_{k=0}^{\mathcal{L}(x)-1} x_k 2^{\beta k} \quad \text{mit } 0 \le x_k \le \gamma \text{ für alle } 0 \le k < \mathcal{L}(x),$$



auf Seite 17

Intervallschreibweise

		auf Seite
(20)	$\mathsf{Natural} \ni x = [x.\texttt{root}, x.\texttt{p} + x.\texttt{size}[= (0)_{k=0}^{\mathcal{R}(x)-\mathcal{L}(x)} \times (x_k)_0^{k=\mathcal{L}(x)-1} \in \mathsf{Digit}^{\mathcal{R}(x)}$	17
(21)	$(x_0, x_1) \times (x_2, x_3) := (x_0, x_1, x_2, x_3)$	
	$[x_0, x_4[:= (x_0, x_1, x_2, x_3)$	
	$(x_k)_0^{k=n} := (x_n, \ldots, x_0)$	
	$(x_k)_{k=0}^n := (x_0, \ldots, x_n)$	
	$0^n := \underbrace{(0, \ldots, 0)}_{n}$ für ein $n \in \mathbb{N}$	17

Ordnung einer Funktion

		auf Seite
(22)	Seien $f, g : \mathbb{N}^k \to \mathbb{R}$ Funktionen für $k \in \mathbb{N}$, dann	
	$\mathcal{O}(f) := \{g : \mathrm{Im}(f) \to \mathbb{R} \mid \exists_{a,b \in \mathbb{R}_{>0}} \forall_{x \in \mathrm{Im}(f)} : a\lvert g(x)\rvert \leq \lvert f(x)\rvert \leq b\lvert g(x)\rvert\}$	7
(23)	$f \sim g :\Leftrightarrow \ f \in \mathcal{O}(g)$	7

Rest bei der Quadratwurzelberechnung

(24) $r := a - \lfloor\sqrt{a}\rfloor^2$ mit $a \in \mathbb{N}$ auf Seite 169

Eigenschaften von Zahlen

(25) Eine Zahl $n \in \mathbb{Z}$ heißt **unzerlegbar**, falls für alle $x \in \mathbb{Z} - \mathbb{Z}^*$ aus $x \mid n$ stets $x \in \{\pm 1\}$ oder $x \in \{\pm n\}$ folgt auf Seite 283.

(26) Eine Zahl $n \in \mathbb{Z}$ heißt **Primzahl**, falls für alle $x, y \in \mathbb{Z}$ aus $n \mid x \cdot y$ stets $n \mid x$ oder $n \mid y$ folgt auf Seite 283.

Zahlen

			auf Seite
(27)	$F_0 = 0, F_1 = 1$ und $F_n = F_{n-1} + F_{n-2}$ für $n \geq 2$.	Fibonacci-Zahlen	265
(28)	$F_n := 2^{2^n} + 1$	Fermat-Zahlen	267

Funktionen

			auf Seite
(29)	$a! := \prod_{i=1}^{a} i$ für $a \in \mathbb{N}$.	Fakultät	305
(30)	Sei $n \in \mathbb{N}$, so setze für alle $a_k \in \mathbb{Z}, 0 \leq k \leq n$:		
	$\mathrm{lcm}(a_k)_{k=0}^n := \min\{x \in \mathbb{N} \mid a_k \mid x, 0 \leq k \leq n\}$	kleinstes gemeinsames Vielfaches	267
	$\gcd(a_k)_{k=0}^n := \max\{x \in \mathbb{N} \mid x \mid a_k, 0 \leq k \leq n\}$.	größter gemeinsamer Teiler	267
(31)	Mit der Konvention $\gcd(0^n) := 0$.	größter gemeinsamer Teiler	267
(32)	$\pi : \mathbb{N} \to \mathbb{N} : n \mapsto \lvert\{p \in \mathbb{P} \mid p \leq n\}\rvert = \sum_{\mathbb{P} \ni p \leq n} 1$.	Primzahlfunktion	288
(33)	$\varphi : \mathbb{N} \to \mathbb{N} : n \mapsto \lvert\{\mathbb{N} \in x \leq n \mid \gcd(x, n) = 1\}\rvert$.	Euler-Funktion	304

Anhang B

Testprogramm und Laufzeiten

CPU	Compiler	OS	fib1	fib2	sqrt	mul	sqr	div	pi
Pentium II 400 MHz	GNU C++ 2.7.2 (Asm)	Win NT 4.0	1.07 s	1.16 s	3.73 s	0.98 s	1.46 s	1.48 s	1.36 s
Pentium II 400 MHz	MS Visual 5.0 (Asm)	Win NT 4.0	1.20 s	1.26 s	4.11 s	1.11 s	1.67 s	1.53 s	1.56 s
(4x)MIPS R10000 180 MHz	SGI MIPSpro C++ 7.1	IRIX 6.4	1.27 s	1.33 s	5.12 s	2.17 s	1.71 s	1.36 s	1.53 s
Pentium II 400 MHz	MS Visual 5.0	Win NT 4.0	1.31 s	1.52 s	6.19 s	2.55 s	3.70 s	1.55 s	1.64 s
Pentium II 300 MHz	GNU C++ 2.7.2 (Asm)	Win NT 4.0	1.42 s	1.53 s	4.91 s	1.3 s	1.94 s	2.0 s	1.8 s
Pentium II 300 MHz	Watcom C++ 11.0 (Asm)	Win NT 4.0	1.46 s	1.56 s	5.14 s	1.36 s	2.05 s	2.09 s	1.89 s
Pentium II 400 MHz	IBM C++ 3.6	Win NT 4.0	1.48 s	1.74 s	19.7 s	7.56 s	3.60 s	12.8 s	5.35 s
Pentium II 300 MHz	MS Visual 5.0 (Asm)	Win NT 4.0	1.59 s	1.67 s	5.43 s	1.46 s	2.18 s	2.1 s	2.06 s
(2x)Pentium II 233 MHz	GNU C++ 2.7.2 (Asm)	Win NT 4.0	1.61 s	1.73 s	5.59 s	1.49 s	2.22 s	2.27 s	2.03 s
(2x)Pentium II 233 MHz	Watcom C++ 11.0 (Asm)	Win NT 4.0	1.67 s	1.77 s	5.84 s	1.56 s	2.36 s	2.36 s	2.14 s
Pentium II 300 MHz	MS Visual 5.0	Win NT 4.0	1.71 s	1.98 s	8.01 s	3.42 s	5 s	1.98 s	1.9 s
(2x)Pentium II 233 MHz	MS Visual 5.0 (Asm)	Win NT 4.0	1.86 s	1.95 s	6.28 s	1.72 s	2.55 s	2.42 s	2.45 s
PowerPC 604e 233 MHz	GNU C++ 2.8.0	AIX 4.1.5	1.98 s	2.16 s	7.92 s	3.07 s	4.65 s	2.00 s	2.06 s
(2x)Pentium II 233 MHz	MS Visual 5.0	Win NT 4.0	1.98 s	2.28 s	9.2 s	3.91 s	5.72 s	2.31 s	2.27 s
Pentium II 300 MHz	IBM C++ 3.6	Win NT 4.0	1.99 s	2.32 s	26.5 s	10.1 s	4.86 s	17.3 s	7.2 s
IBM S/390 G5 CMOS 420 MHz	IBM C/C++ 2.6	OS/390 2.6	2.23 s	2.41 s	9.61 s	3.02 s	4.47 s	2.77 s	2.78 s
Pentium II 300 MHz	Watcom C++ 11.0	Win NT 4.0	2.23 s	2.38 s	8.99 s	3.33 s	4.89 s	2.45 s	2.22 s
MIPS R8000 75 MHz	SGI MIPSpro C++ 7.1	IRIX 6.3	2.24 s	2.6 s	9.32 s	3.95 s	3.28 s	2.5 s	2.93 s
Pentium II 300 MHz	GNU C++ 2.7.2	Win NT 4.0	2.45 s	2.74 s	11 s	4.13 s	6.06 s	3.01 s	2.67 s
IBM J50 PowerPC 604e 200MHz	IBM CSet++ 3.1.4	AIX 4.2.1	2.45 s	2.83 s	34.5 s	14.2 s	7.03 s	25.0 s	9.08
(2x)Pentium II 233 MHz	Watcom C++ 11.0	Win NT 4.0	2.53 s	2.70 s	10.2 s	3.8 s	5.58 s	2.78 s	2.5 s
(2x)Pentium II 233 MHz	GNU C++ 2.7.2	Win NT 4.0	2.77 s	3.09 s	12.4 s	4.69 s	6.88 s	3.41 s	3.02 s
IBM 397 Power2 SC 160MHz	IBM CSet++ 3.1.4	AIX 4.1.5	2.78 s	3.18 s	37.6 s	14.2 s	6.73 s	24.6 s	10.4 s
(2x)UltraSPARC 200 MHz	GNU C++ 2.8.0	SunOS 5.6	2.8 s	2.9 s	9.2 s	2.67 s	3.88 s	3.11 s	3.01 s
Pentium II 300 MHz	Borland C++ 5.02	Win NT 4.0	2.81 s	3.2 s	11.9 s	4.15 s	6.13 s	3.31 s	3.18 s
(4x)MIPS R10000 180 MHz	SGI MIPSpro C++ 7.1 (32 Bit)	IRIX 6.4	2.83 s	3.06 s	11.2 s	3.64 s	5.34 s	3.13 s	3.24 s
MIPS R10000 174 MHz	SGI MIPSpro C++ 7.1 (32 Bit)	IRIX 6.3	2.89 s	3.18 s	11.6 s	3.77 s	5.59 s	3.32 s	3.34 s
DEC Alpha 150 MHz	DEC C++ 5.0	Digital Unix	3.03 s	3.28 s	11.6 s	4.87 s	5.13 s	3.07 s	3.78 s
IBM 6x86 200 MHz	GNU C++ 2.7.2 (Asm)	Win NT 4.0	3.05 s	3.33 s	10.6 s	2.82 s	4.37 s	4.14 s	4.07 s
(2x)Pentium II 233 MHz	Borland C++ 5.02	Win NT 4.0	3.19 s	3.64 s	13.4 s	4.72 s	6.97 s	3.73 s	3.59 s
IBM 6x86 200 MHz	Watcom C++ 11.0 (Asm)	Win NT 4.0	3.21 s	3.4 s	11.1 s	3.04 s	4.57 s	4.34 s	4.1 s
MIPS R10000 174 MHz	GNU C++ 2.7.2	IRIX 6.3	3.24 s	3.45 s	14 s	5.95 s	8.74 s	3.43 s	3.33 s
UltraSPARC 167 MHz	GNU C++ 2.8.0 (Asm)	SunOS 5.6	3.36 s	3.48 s	11.1 s	3.19 s	4.69 s	3.78 s	3.61 s
Power2 SC 135 MHz	GNU C++ 2.8.0	AIX 4.1.5	3.41 s	3.7 s	13.5 s	4.91 s	7.35 s	3.57 s	3.75 s
Pentium 133 MHz	Watcom C++ 11.0 (Asm)	Win NT 4.0	3.58 s	3.76 s	12.5 s	3.79 s	5.56 s	4.55 s	4.48 s
IBM 6x86 200 MHz	MS Visual 5.0 (Asm)	Win NT 4.0	3.66 s	3.79 s	12.4 s	3.29 s	4.95 s	4.91 s	4.97 s
Pentium 133 MHz	GNU C++ 2.7.2 (Asm)	Win NT 4.0	3.69 s	3.96 s	12.9 s	3.74 s	5.55 s	4.92 s	4.88 s
Pentium 133 MHz	MS Visual 5.0 (Asm)	Win NT 4.0	4.08 s	4.24 s	13.7 s	3.88 s	5.83 s	5.15 s	5.41 s
MIPS R5000 180 MHz	SGI MIPSpro C++ 7.1	IRIX 6.3	4.14 s	4.57 s	16.8 s	5.61 s	8.29 s	4.74 s	4.74 s
MIPS R5000 180 MHz	GNU C++ 2.7.2	IRIX 6.3	4.26 s	4.5 s	18.3 s	6.91 s	10.2 s	4.85 s	4.8 s
UltraSPARC 134 MHz	GNU C++ 2.7.2.1 (Asm)	Solaris 2.5.1	4.38 s	4.55 s	14.9 s	4.05 s	6.14 s	5.67 s	5.41 s
(2x)UltraSPARC 200 MHz	GNU C++ 2.8.0	SunOS 5.6	4.49 s	4.67 s	15.9 s	4.61 s	6.67 s	5.52 s	4.55 s
(2x)UltraSPARC 200 MHz	SUN WorkShop C++ 4.2	SunOS 5.6	4.53 s	4.82 s	16.3 s	4.64 s	6.94 s	5.04 s	4.67 s
IBM 9021-982	IBM C/C++	OS/390	4.57 s	4.97 s	20.1 s	6.57 s	9.66 s	5.73 s	5.65 s
Pentium 100 MHz	GNU C++ 2.7.2.2 (Asm)	Linux 2.0	4.76 s	5.11 s	16.5 s	4.49 s	6.61 s	6.45 s	6.37 s
IBM S/390 G3 CMOS	IBM C/C++ 2.4	OS/390 2.4	5.14 s	5.47 s	21.3 s	6.43 s	9.49 s	6.2 s	6.31 s
UltraSPARC 167 MHz	GNU C++ 2.8.0	SunOS 5.6	5.39 s	5.58 s	19.2 s	5.47 s	8.08 s	5.88 s	5.45 s
UltraSPARC 167 MHz	SUN WorkShop C++ 4.2	SunOS 5.6	5.44 s	5.76 s	19.6 s	5.61 s	8.34 s	6.04 s	5.6 s
(2x)UltraSPARC 200 MHz	Apogee C++ 3.0	SunOS 5.6	5.82 s	5.91 s	24.3 s	9.26 s	13.5 s	6.53 s	5.96 s
IBM 6x86 200 MHz	MS Visual 5.0	Win NT 4.0	5.85 s	6.22 s	23.1 s	7.1 s	10.5 s	7.04 s	6.85 s
Pentium 133 MHz	MS Visual 5.0	Win NT 4.0	6.48 s	6.89 s	25.4 s	7.75 s	11.5 s	7.66 s	7.73 s
IBM 6x86 200 MHz	Watcom C++ 11.0	Win NT 4.0	6.6 s	6.76 s	23.8 s	6.93 s	10.2 s	7.1 s	6.9 s
IBM 6x86 200 MHz	GNU C++ 2.7.2	Win NT 4.0	6.92 s	7.11 s	27 s	8.54 s	12.9 s	7.9 s	7.54 s
UltraSPARC 167 MHz	Apogee C++ 3.0	SunOS 5.6	7 s	7.08 s	29.2 s	11.1 s	16.2 s	7.96 s	7.16 s
MIPS R4600 134 MHz	GNU C++ 2.7.2	IRIX 6.3	7.09 s	7.33 s	29.2 s	10 s	14.7 s	8.07 s	8 s
IBM 6x86 200 MHz	Borland C++ 5.02	Win NT 4.0	7.2 s	7.85 s	26.1 s	7.93 s	11.8 s	7.54 s	7.34 s
Pentium 133 MHz	Watcom C++ 11.0	Win NT 4.0	7.33 s	7.47 s	25.9 s	7.81 s	11.5 s	7.68 s	7.45 s
Pentium 133 MHz	GNU C++ 2.7.2	Win NT 4.0	8.3 s	8.88 s	33.7 s	11.1 s	16.3 s	9.69 s	9.32 s
(2x)UltraSPARC 200 MHz	EDG C++ front end 2.33	SunOS 5.6	8.7 s	8.58 s	28.9 s	6.51 s	9.65 s	9.6 s	8.98 s
Pentium 133 MHz	Borland C++ 5.02	Win NT 4.0	8.73 s	9.45 s	31.6 s	9.68 s	14.3 s	9.21 s	9.08 s
PowerPC 80 MHz	GNU C++ 2.7.2	AIX 3.2.5	10.2 s	10.5 s	41.8 s	14.7 s	21.5 s	11 s	11.1 s
UltraSPARC 167 MHz	EDG C++ front end 2.33	SunOS 5.6	10.5 s	10.3 s	35.3 s	7.87 s	11.6 s	11.5 s	10.8 s
(2x)SuperSPARC 50 MHz	SUN WorkShop C++ 4.2	SunOS 5.6	10.5 s	11.1 s	41.2 s	12.6 s	20.8 s	13.8 s	11.9 s
PowerPC 80 MHz	IBM C Set++	AIX 3.2.5	10.5 s	11.1 s	41.3 s	12.6 s	18.4 s	12 s	11.9 s
Pentium 100 MHz	KAI C++ 3.2	Linux 2.0	11.1 s	11.8 s	41.7 s	12.1 s	17.7 s	12.6 s	12.1 s
Pentium 100 MHz	GNU C++ 2.7.2.2	Linux 2.0	11.1 s	12 s	44.3 s	14.4 s	21.1 s	12.8 s	12.3 s
HP PA 7100	GNU C++ 2.8.0	HP-UX	12.3 s	13.6 s	53 s	19.8 s	29 s	14.1 s	14.2 s
HP PA 7100	HP C++ A.10.22	HP-UX	12.3 s	17.4 s	82.7 s	15.3 s	24.4 s	31.1 s	30.3 s
(2x)SuperSPARC 50 MHz	GNU C++ 2.8.0	SunOS 5.6	12.9 s	13.8 s	48 s	15.7 s	22 s	11.3 s	12.3 s
MIPS R4000 100 MHz	GNU C++ 2.6.2	IRIX 5.2	13.3 s	14.7 s	53.3 s	18.6 s	27.2 s	14.7 s	14.6 s
(2x)SuperSPARC 50 MHz	Apogee C++ 3.0	SunOS 5.6	16.7 s	16.8 s	62.3 s	28.6 s	39.1 s	20.8 s	17.8 s
(2x)SuperSPARC 50 MHz	EDG C++ front end 2.33	SunOS 5.6	27.3 s	27.1 s	96.4 s	21.8 s	35.8 s	31.1 s	28.9 s

B.1 Testprogramm

"test.cpp" 355 ≡

```
////////////////////////////////
//
// Piologie V 1.3
// multi-precision arithmetic
// simple test
//
// (c) 1996-1999 HiPiLib
//               www.hipilib.de
//
// Sebastian Wedeniwski
// 12/13/1999
//

#include <time.h>
#include "pi.h"
#include "natural.h"

#ifdef _NEEDS_PIOLOGIE_KEY_
# include "pkey.h"
#endif

#ifndef _Old_STD_
using namespace std;
#endif


int main()
{
  clock_t start,stop;
  double  duration;
  Natural a,b,c,d,e;

  // time(0) is better than clock() on some systems, e.g. PowerPC!

  start = clock();
  a = fibonacci(800000);
  stop = clock();
  duration = double(stop-start)/CLOCKS_PER_SEC;
  cout << "fib1 time [s] = " << duration << ", (" << a%ALPHA << ')' << endl;

  start = clock();
  b = fibonacci(900000);
  stop = clock();
  duration = double(stop-start)/CLOCKS_PER_SEC;
  cout << "fib2 time [s] = " << duration << ", (" << b%ALPHA << ')' << endl;

  start = clock();
  c = sqrt(a);
  stop = clock();
  duration = double(stop-start)/CLOCKS_PER_SEC;
  cout << "sqrt time [s] = " << duration << ", (" << c%ALPHA << ')' << endl;

  start = clock();
```

```
  d = a*b;
  stop = clock();
  duration = double(stop-start)/CLOCKS_PER_SEC;
  cout << "mul time [s] = " << duration << ", (" << d%ALPHA << ')' << endl;

  start = clock();
  e = d*d;
  stop = clock();
  duration = double(stop-start)/CLOCKS_PER_SEC;
  cout << "sqr time [s] = " << duration << ", (" << e%ALPHA << ')' << endl;

  start = clock();
  b /= a;
  stop = clock();
  duration = double(stop-start)/CLOCKS_PER_SEC;
  cout << "div time [s] = " << duration << ", (" << b%ALPHA << ')' << endl;

  start = clock();
  Pi pi(20000);
  stop = clock();
  duration = double(stop-start)/CLOCKS_PER_SEC;
  cout << "pi time [s] = " << duration << endl;

  return 0;
}
◇
```

B.1.1 Schnittstelle zu GMP

"gmp_natural.h" 356 ≡

```
#include "gmp.h"
#include <iostream.h>
#include <limits.h>

typedef unsigned long Digit;

const Digit ALPHA      = 1000000000;
const size_t BETA      = sizeof(Digit)*CHAR_BIT;
const Digit  GAMMA     = ~Digit(0);                  // 2^BETA - 1
const Digit  GAMMA_LOW = GAMMA >> (BETA/2);


struct Natural {
  mpz_t value;

  Natural(Digit a = 0) { mpz_init_set_ui(value, a); }
  Natural(const Natural& a) { mpz_init_set(value, a.value); }
  ~Natural() { mpz_clear(value); }

  Natural& operator=(const Natural& a)
   { mpz_set(value, a.value); return *this; }
  Natural& operator/=(const Natural& a)
   { mpz_tdiv_q(value, value, a.value); return *this; }
  Natural& operator+=(const Natural& a)
   { mpz_add(value, value, a.value); return *this; }
  Natural& operator--()
```

```
   { mpz_sub_ui(value, value, 1); return *this; }
  Natural& operator<<=(const Digit a)
   { mpz_mul_2exp(value, value, a); return *this; }
};

inline Digit operator%(const Natural& a, const Digit b)
{
  Natural c;
  mpz_tdiv_r_ui(c.value, a.value, b);
  if (c.value->_mp_size == 0) return 0;
  return c.value->_mp_d[0];
}

inline Natural sqrt(const Natural& a)
{
  Natural c;
  mpz_sqrt(c.value, a.value);
  return c;
}

inline void add(const Natural& a, const Natural& b, Natural& c)
{
  mpz_add(c.value, a.value, b.value);
}

inline void sub(const Natural& a, const Natural& b, Natural& c)
{
  mpz_sub(c.value, a.value, b.value);
}

inline void mul(const Natural& a, const Natural& b, Natural& c)
{
  mpz_mul(c.value, a.value, b.value);
}

inline void sqr(const Natural& a, Natural& c)
{
  mpz_mul(c.value, a.value, a.value);
}

inline void swap(Natural& a, Natural& b)
{
  mpz_t t = a.value; a.value = b.value; b.value = t;
}


inline Digit log2(Digit a)
// Algorithm:  b := log2(a)
// Input:      a in Digit.
// Output:     b in Digit
//             such that if a > 0 then b = floor(log2(a)))+1 else b = 0 ||
{
  Digit b = 0;

#if CHAR_BIT == 8
  static const Digit c[16] = {0, 0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3 };
```

```
  if (a > GAMMA_LOW) { b += BETA/2; a >>= BETA/2; }
  if (a >= Digit(1) << BETA/4) { b += BETA/4; a >>= BETA/4; }
  if (a >= Digit(1) << BETA/8) { b += BETA/8; a >>= BETA/8; }
  if (BETA > 32) {
    if (a >= Digit(1) << BETA/16) { b += BETA/16; a >>= BETA/16; }
    if (BETA > 64) {
      while (a >= 16) { b += 4; a >>= 4; }
    }
  }
  return b+c[a];
#else
  while (a >>= 1) ++b;
  return b;
#endif
}


Natural fibonacci(Digit n)
// Algorithm:  c := fibonacci(n)
// Input:      n in Digit.
// Output:     c in Natural such that c is the n-th Fibonacci number ||
{
  if (n <= 1) return n;
  Natural k,i,j(1);
  Natural t;
  Digit a = Digit(1) << log2(n);
  do {
    sqr(j, k); sqr(i, t);
    add(k, t, j);               // (4.2)
    --k; sub(t, k, i);
    i <<= 1; i += t;            // (4.4)
    while (n&a) {
      j += i; swap(i, j);
      a >>= 1;
      if (a == 0) return i;
      sqr(j, k); sqr(i, t);
      add(k, t, j);             // (4.2)
      sub(t, k, i);
      --i; i <<= 1; i += t;     // (4.4)
    }
  } while (a >>= 1);
  return i;
}


struct Pi {
  Pi(Digit n) {}
};
◇
```

B.1.2 Schnittstelle zu freelip

"freelip_natural.h" 358 ≡

```
extern "C" {
#include "lip.h"
```

```
}
#include <iostream.h>
#include <limits.h>

typedef unsigned long Digit;

const Digit ALPHA      = 1000000000;
const size_t BETA      = sizeof(Digit)*CHAR_BIT;
const Digit  GAMMA     = ~Digit(0);                 // 2^BETA - 1
const Digit  GAMMA_LOW = GAMMA >> (BETA/2);


struct Natural {
  verylong value;

  Natural(Digit a = 0) { value = 0; zintoz(a, &value); }
  Natural(const Natural& a) { value = 0; zcopy(a.value, &value); }
  ~Natural() { zfree(&value); }

  Natural& operator=(const Natural& a)
   { zcopy(a.value, &value); return *this; }
  Natural& operator/=(const Natural& a)
   { Natural x; zdiv(value, a.value, &value, &x.value); return *this; }
  Natural& operator+=(const Natural& a)
   { zadd(value, a.value, &value); return *this; }
  Natural& operator--()
   { zsadd(value, -1, &value); return *this; }
  Natural& operator<<=(const Digit a)
   { zlshift(value, a, &value); return *this; }
};

inline Digit operator%(const Natural& a, const Digit b)
{
  Natural c;
  return zsdiv(a.value, b, &c.value);
}

inline Natural sqrt(const Natural& a)
{
  Natural c,d;
  zsqrt(a.value, &c.value, &d.value);
  return c;
}

inline void add(const Natural& a, const Natural& b, Natural& c)
{
  zadd(a.value, b.value, &c.value);
}

inline void sub(const Natural& a, const Natural& b, Natural& c)
{
  zsub(a.value, b.value, &c.value);
}

inline void mul(const Natural& a, const Natural& b, Natural& c)
{
```

```
  zmul(a.value, b.value, &c.value);
}

inline void sqr(const Natural& a, Natural& c)
{
  zsq(a.value, &c.value);
}

inline void swap(Natural& a, Natural& b)
{
  verylong t = a.value; a.value = b.value; b.value = t;
}


inline Digit log2(Digit a)
// Algorithm:  b := log2(a)
// Input:      a in Digit.
// Output:     b in Digit
//             such that if a > 0 then b = floor(log2(a)))+1 else b = 0 ||
{
  Digit b = 0;

#if CHAR_BIT == 8
  static const Digit c[16] = {0, 0, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3 };

  if (a > GAMMA_LOW) { b += BETA/2; a >>= BETA/2; }
  if (a >= Digit(1) << BETA/4) { b += BETA/4; a >>= BETA/4; }
  if (a >= Digit(1) << BETA/8) { b += BETA/8; a >>= BETA/8; }
  if (BETA > 32) {
    if (a >= Digit(1) << BETA/16) { b += BETA/16; a >>= BETA/16; }
    if (BETA > 64) {
      while (a >= 16) { b += 4; a >>= 4; }
    }
  }
  return b+c[a];
#else
  while (a >>= 1) ++b;
  return b;
#endif
}


Natural fibonacci(Digit n)
// Algorithm:  c := fibonacci(n)
// Input:      n in Digit.
// Output:     c in Natural such that c is the n-th Fibonacci number ||
{
  if (n <= 1) return n;
  Natural k,i,j(1);
  Natural t;
  Digit a = Digit(1) << log2(n);
  do {
    sqr(j, k); sqr(i, t);
    add(k, t, j);               // (4.2)
    --k; sub(t, k, i);
    i <<= 1; i += t;            // (4.4)
```

```
    while (n&a) {
      j += i; swap(i, j);
      a >>= 1;
      if (a == 0) return i;
      sqr(j, k); sqr(i, t);
      add(k, t, j);             // (4.2)
      sub(t, k, i);
      --i; i <<= 1; i += t;     // (4.4)
    }
  } while (a >>= 1);
  return i;
}


struct Pi {
  Pi(Digit n) {}
};
◇
```

B.2 Check-Programm

"check.cpp" 361 ≡

```
////////////////////////////////
//
// Piologie V 1.3
// multi-precision arithmetic
// intensive test
//
// (c) 1996-1999 HiPiLib
//               www.hipilib.de
//
// Sebastian Wedeniwski
// 12/13/1999
//

#include "rational.h"
#include "nmbrthry.h"
#include <time.h>
#ifndef _Old_STD_
# include <fstream>
using namespace std;
#else
# include <fstream.h>
#endif

// BETA >= 16!

size_t ITERATION;
Digit MAX_FACTORIAL;
Digit MAX_FIBONACCI;
Digit MAX_RANDOM;
Digit INNER_ITERATION;
```

```
template <class T>
T gcd2(T a, T b)
{
  T c;
  while (b != 0) { c = a%b; a = b; b = c; }
  return abs(a);
}

Natural factorial2(Natural a)
{
  Natural b = 1;
  while (a > 0) { b *= a; --a; }
  return b;
}

Natural fibonacci2(Natural a)
{
  if (a <= 1) return a;
  Natural b = 1;
  Natural t,c = Digit(0);
  do { t = b; b += c; c = t; } while (--a > 1);
  return b;
}

Natural fibonacci3(Natural a)
{
  Natural b = 1;
  Natural c = Digit(0);
  while (a != 0) { c += b; b = c-b; --a; }
  return c;
}

Natural fibonacci4(Natural a)
{
  if (a == 0) return Digit(0);
  Natural b = 1;
  Natural t,c = Digit(0);
  while (--a != 0) {
    t = c+b;
    if (--a == 0) return t;
    c = b+t;
    if (--a == 0) return c;
    b = t+c;
  }
  return b;
}


template <class T, class D>
void identity_generic(T a, D b)
{
  T r,s,t;

  assert((a + b) == (b + a));
  s = a + b; t = b + a;
  assert(s == t);
```

```
  r = s;
  s = a; s += b;
  t = b; t += a;
  assert(r == s && s == t);

  assert((a * b) == (b * a));
  s = a * b; t = b * a;
  assert(s == t);
  r = s;
  s = a; s *= b;
  t = b; t *= a;
  assert(r == s && s == t);

  assert(((a + b) - b) == a);
  t = a + b; t = t - b;
  assert(t == a);
  t = a; t += b; t -= b;
  assert(t == a);

  assert(((a + b) - a) == b);
  t = a + b; t = t - a;
  assert(t == b);
  t = a; t += b; t -= a;
  assert(t == b);

  t = b;
  assert((T(b) - t) == 0);    // if T == Natural
  s = T(b) - t;
  assert(s == 0);
  s = b; s -= t;
  assert(s == 0);
  assert((t - b) == 0);
  s = t - b;
  assert(s == 0);
  s = t; s -= b;
  assert(s == 0);

  assert(((a * b) / b) == a);
  assert(((a * b) / a) == b);
  assert(((a * b) / (b * a)) == 1);
}

template <class T, class D>
void identity_Natural_Digit(T a, D b)
{
  T r,s,t;

  identity_generic(a, b);

  assert(((a * b) / b) == a);
  assert(((a * b) % b) == 0);
  t = a * b; div(t, T(b), t, s);
  assert(t == a && s == 0);
  t = a; t *= b; t /= b;
  s = a; s *= b; s %= b;
  assert(t == a && s == 0);
```

```
  assert(((a * b) / a) == b);
  assert(((a * b) % a) == 0);
  t = a * b; div(t, a, t, s);
  assert(t == b && s == 0);
  t = a; t *= b; t /= a;
  s = a; s *= b; s %= a;
  assert(t == b && s == 0);

  assert(((a * b) / (b * a)) == 1);
  t = a * b; s = b * a; div(t, s, t, s);
  assert(t == 1 && s == 0);
  t = a; t *= b; s = b; s *= a; r = t; t /= s; s %= r;
  assert(t == 1 && s == 0);

  assert((b * (a / b) + (a % b)) == a);
  div(a, T(b), s, t); s = b * s; s = t + s;
  assert(s == a);
  s = a; s /= b; s *= b; t = a; t %= b; s += t;
  assert(s == a);

  // Logic:
  assert((a | b) == (b | a));
  s = a | b; t = b | a;
  assert(s == t);
  s = a; s |= b; t = b; t |= a;
  assert(s == t);

  assert((a & b) == (b & a));
  s = a & b; t = b & a;
  assert(s == t);
  s = a; s &= b; t = b; t &= a;
  assert(s == t);

  assert((a ^ b) == (b ^ a));
  s = a ^ b; t = b ^ a;
  assert(s == t);
  s = a; s ^= b; t = b; t ^= a;
  assert(s == t);

  assert((a & (a | b)) == a);
  s = a | b; s = a & s;
  assert(s == a);
  s = a; s |= b; s &= a;
  assert(s == a);

  assert((a | (a & b)) == a);
  s = a & b; s = a | s;
  assert(s == a);
  s = a; s &= b; s |= a;
  assert(s == a);
}

template <class T, class D>
void identity_Natural_Digit2(const T a, const D b)
// a >= b if a in Natural.
```

```
{
  T s,t;

  s = a-b;
  assert(s == a-b);
  t = a; t -= b;
  assert(s == t);

  assert(((a - b) + b) == a);
  s = a-b; s = s+b;
  assert(s == a);
  t = a; t -= b; t += b;
  assert(s == t);
}

template <class T>
void identity_generic2(const T a, const T b, const T c)
{
  T i,j,r,s,t;

  assert((a + (b + c)) == ((a + b) + c));
  s = b+c; s = a+s; t = a+b; t = t+c;
  assert(s == t);
  r = s;
  s = b; s += c; s += a;
  t = a; t += b; t += c;
  assert(r == s && s == t);

  assert((a * (b * c)) == ((a * b) * c));
  s = b*c; s = s*a; t = a*b; t = c*t;
  assert(s == t);
  r = s;
  s = b; s *= c; s *= a;
  t = a; t *= b; t *= c;
  assert(r == s && s == t);

  assert(((a + b) * (b + c)) == (a*b + b*b + a*c + b*c));
  s = a+b; r = b+c; s = s*r;
  t = a*b; r = b*b; t = t+r;
  r = a*c; t = t+r; r = b*c; t = r+t;
  assert(s == t);
  s = a; s += b; r = b; r += c; s *= r;
  t = a; t *= b; r = b; r *= b; t += r;
  r = a; r *= c; t += r; r = b; r *= c; t += r;
  assert(s == t);

  assert(((a + c) * (c + a)) == (a*a + 2*a*c + c*c));
  s = a+c; r = c+a; s = s*r;
  assert(((a + c) * (c + a)) == s);
  t = a*a; r = a*c; r = r*2; t = r+t;
//  assert(r.even());
  assert(t == (a*a + 2*a*c));
  r = c*c; t = t+r;
  assert(s == t);
  s = a; s += c; r = c; r += a; s *= r;
  t = a; t *= a; r = 2; r *= a; r *= c; t += r;
```

```
r = c; r *= c; t += r;
assert(s == t);

assert((a * (b + c)) == ((b * a) + (c * a)));
s = b+c; s = a*s;
t = b*a; r = c*a; t = t+r;
assert(s == t);
r = b; r += c; s = a; s *= r;
t = b; t *= a; r = c; r *= a; t += r;
assert(s == t);

assert(((a + b) * c) == ((c * a) + (c * b)));
s = a+b; s = s*c;
t = c*a; r = c*b; t = t+r;
assert(s == t);
s = a; s += b; s *= c;
t = c; t *= a; r = c; r *= b; t += r;
assert(s == t);

assert((a - a) == 0);
t = a-a;
assert(t == 0);
t = a; t -= a;
assert(t == 0);

assert((a << 0) == a);
t = a << 0;
assert(t == a);
t = a; t <<= 0;
assert(t == a);

assert((a >> 0) == a);
t = a >> 0;
assert(t == a);
t = a; t >>= 0;
assert(t == a);

// constructor:
T t2  = a + b;
T t3  = a * b;
T t4  = a * a;
T t5  = a / b;
T t7  = a * 103;
T t8  = a << 103;
T t9  = a >> 103;
t = b + a;
assert(t2 == t);
t = b * a;
assert(t3 == t);
t = a * a;
assert(t4 == t);
t = a / b;
assert(t5 == t);
t = a * 103;
assert(t7 == t);
t = a << 103;
```

```
  assert(t8 == t);
  t = a >> 103;
  assert(t9 == t);
}

template <class T>
void identity_Natural(const T& a, const T& b, const T& c)
{
  T i,j,r,s,t;

  identity_generic2(a, b, c);

  s = (a+b) % c;
  t = ((a % c) + (b % c)) % c;
  if (s < 0) s = s+abs(c);
  if (t < 0) t = t+abs(c);
  assert(s == t);
  div(a, c, s, r); div(b, c, s, t); t = r+t;
  div(t, c, t, s); t = a+b; div(t, c, r, t);
  if (s < 0) s = s + abs(c);
  if (t < 0) t = t + abs(c);
  assert(s == t);
  s = a; s += b; s %= c;
  t = a; t %= c; r = b; r %= c; t += r; t %= c;
  if (s < 0) s += abs(c);
  if (t < 0) t += abs(c);
  assert(s == t);

  i = j = 101;
  assert(i == j && i == 101 && j.odd() && (i&15) == 5);
  i = j = i%13;
  assert(i == j && i == 10 && j.even() && (i&3) == 2);
  i = j = 0;
  assert(i == j && i == 0 && j.even() && (i&3) == 0);
  i = j = 1;
  assert(i == j && i == 1 && j.odd() && (i&121) == 1);
  i = j = 111;
  assert(i == j && i == 111 && j.odd() && (i&3) == 3);
  i = j = j/13;
  assert(i == j && i == 8 && j.even() && (i&7) == 0);

  size_t l = size_t(log2(a));
  t = a; s = 1; s <<= l;
  assert((T(1) << l) == s && s <= T(abs(t)));
  for (i = 1, j = 100; i < 100; ++i, j--) {
    r = t; s = t << 1; t <<= 1;
    assert((a << size_t(i.highest())) == s && s == t);
    s = r << 1; r = r << 1;
    assert(r == s && r == t);
    r = a << size_t(i.lowest());
    assert(r == s && r == t);
    r = T(1) << size_t(i.highest()); r *= a;
    assert(r == t);
    swap(r, t);
    ++l; s = 1; s <<= l;
    assert((T(1) << l) == s && s <= T(abs(t)));
```

```
  assert(size_t(log2(t)) == l);
  assert(i+j == 101);
}
assert(j == 1);

l = size_t(log2(a));
t = a; s = 1; s <<= l;
assert((T(1) << l) == s && s <= T(abs(t)));
for (i = 1, j = 100; i <= 100; i++, --j) {
  r = t; s = t >> 1; t >>= 1;
  assert((a >> size_t(i.lowest())) == s && s == t);
  s = r >> 1; r = r >> 1;
  assert(r == s);
  r = a >> size_t(i.highest());
  assert(r == s);
  r = T(1) << size_t(i.highest()); s = a/r;
  assert(s == t);
  swap(s, t);
  if (l) {
    --l; s = 1; s <<= l;
    assert((T(1) << l) == s && s <= T(abs(t)));
    assert(size_t(log2(t)) == l);
  }
  assert(i+j == 101);
}
assert(j == 0);

t = s = a;
j = INNER_ITERATION+1;
for (i = 1; i < j; ++i) {
  assert(++t == ((s++)+1) && t == s);
  assert(s == (a+i) && s == (a+i%j));
  r = a; r += i.highest();
  assert(s == r);
}

for (i = 0; i < 100; i += 3) {
  t = a+(b*(i%101));
  r = a; r += b * (i%101);
  assert(r == t);
  r = b * (i%101); r += a;
  assert(r == t);
  r = b; r *= (i%101); r += a;
  assert(r == t);
}

for (i = 0; i < 100; ++i) {
  t = a; t.split(size_t((i%101)), s, t);
  r = a; r >>= BETA*size_t((i%101));
  assert(r == s && r == (a >> (BETA*size_t(i%101))));
  s = 1; s <<= BETA*size_t(i%101); --s;
  r = abs(a); r &= s;
  assert(abs(r) == abs(t) && abs(r) == (abs(a)&abs(s)));
  r = T(abs(a)) & s;
  assert(abs(r) == abs(t));
}
```

```
t = pow(a, 5);
s = root(t, 5);
assert(s == a);
t /= a; s = sqrt(t); s = sqrt(s);
assert(abs(s) == abs(a));
t = a*a + 101;
sqrt(t, r, s);
assert(r*r+s == t);

t = gcd(a, b);
s = gcd(a, c);
assert(t == gcd2(a, b));
assert(s == gcd2(a, c));
assert(T(gcd(b, c)) == gcd2(b, c));
assert(t*lcm(a, b) == T(abs(T(a*b))));
assert(lcm(c, a)*s == T(abs(T(c*a))));

// Logic:
assert((a & a) == a);
s = a & a;
assert(s == a);
s = a; s &= a;
assert(s == a);

assert((a | a) == a);
s = a | a;
assert(s == a);
s = a; s |= a;
assert(s == a);

assert((a ^ a) == 0);
s = a ^ a;
assert(s == 0);
s = a; s ^= a;
assert(s == 0);

assert((a | (b | c)) == ((a | b) | c));
s = b | c; s = a | s; t = a | b; t = t | c;
assert(s == t);
s = b; s |= c; s |= a;
t = a; t |= b; t |= c;
assert(s == t);

assert((a & (b & c)) == ((a & b) & c));
s = b & c; s = a & s; t = a & b; t = t & c;
assert(s == t);
s = b; s &= c; s &= a;
t = a; t &= b; t &= c;
assert(s == t);

assert((a & (b | c)) == ((a & b) | (a & c)));
s = b | c; s = a & s;
t = a & b; r = a & c; t = t | r;
assert(s == t);
s = b; s |= c; s &= a;
```

```
r = t = a; t &= b; r &= c; t |= r;
assert(s == t);

assert((a | (b & c)) == ((a | b) & (a | c)));
s = b & c; s = a | s;
t = a | b; r = a | c; t = t & r;
assert(s == t);
s = b; s &= c; s |= a;
t = a; t |= b; r = a; r |= c; t &= r;
assert(s == t);

assert((~(~a) & ~a) == 0);
s = ~a; s = ~s; t = ~a; s = s & t;
assert(s == 0);
s = ~a; s = ~s; s &= ~a;
assert(s == 0);

// b can't be a Digit!
assert((~(a | b) & ~(~a & ~b)) == 0);
s = a | b; s = ~s;
r = ~a; t = ~b; r = r & t; r = ~r; s = s & r;
assert(s == 0);
s = a; s |= b; s = ~s;
r = ~a; t = ~b; r &= t; r = ~r; s &= r;
assert(s == 0);

// b can't be a Digit!
assert((~(a & b) & ~(~a | ~b)) == 0);
s = a & b; s = ~s;
r = ~a; t = ~b; r = r | t; r = ~r; s = s & r;
assert(s == 0);
s = a; s &= b; s = ~s;
r = ~a; t = ~b; r |= t; r = ~r; s &= r;
assert(s == 0);

// b can't be a Digit!
assert(((a ^ b) & ~((a & ~b) | (~a & b))) == 0);
s = a ^ b; r = ~b; r = a & r;
t = ~a; t = t & b; r = r | t; r = ~r; s = s & r;
assert(s == 0);
s = a; s ^= b; r = ~b; r &= a;
t = ~a; t &= b; r |= t; r = ~r; s &= r;
assert(s == 0);

s = a;
r = INNER_ITERATION;
for (i = 0, j = 3; i < r; i += 3, j += 5) {
  t = s; t.setbit(size_t(j.highest()));
  s |= (T(1) << size_t(j.highest()));
  assert(s == t && s.testbit(size_t(j.highest())) == true);
  s.clearbit(size_t(i.highest()));
  assert(s.testbit(size_t(i.highest())) == false);
}

// constructor:
T t6  = a % b;
```

```
  T t10 = a & b;
  T t11 = a | b;
  T t12 = a ^ b;
  T t13 = ~a;
  T t14 = sqrt(T(abs(a)));
  t = a % b;
  assert(t6 == t);
  t = b & a;
  assert(t10 == t);
  t = b | a;
  assert(t11 == t);
  t = b ^ a;
  assert(t12 == t);
  t = ~a;
  assert(t13 == t);
  t = sqrt(T(abs(a)));
  assert(t14 == t);
}

template <class T>
void identity_Natural2(const T& a, const T& b)
// a > b if a,b in Natural.
{
  T r,s,t;

  for (T i = 100; i >= 0; i -= 5) {
    t = a-(b*(i%101));
    r = a; r -= b * (i%101);
    assert(r == t);
    r = b * (i%101); r = a-r;
    assert(r == t);
    r = b; r *= (i%101); s = a; s -= r;
    assert(s == t);
    if (i == 0) break;
  }

  // constructor:
  T t2 = a - b;
  t = a - b;
  assert(t2 == t);
}

template <class T>
void identity_generic3(const T a, const T b, const T c)
// T at least Integer
{
  T r,s,t;

  assert(-(-a) == a);
  s = -a; s = -s;
  assert(s == a);
  s = a; s = -s; s = -s;
  assert(s == a);

  assert((a + (-b)) == (a - b));
  s = -b; s = a+s; t = a-b;
```

```
  assert(s == t);
  s = b; s = -s; s += a; t = a; t -= b;
  assert(s == t);

  assert((a * (-b)) == -(a * b));
  s = -b; s = a*s; t = a*b; t = -t;
  assert(s == t);
  s = b; s = -s; s *= a;
  t = a; t *= b; t = -t;
  assert(s == t);

  assert((a / (-b)) == -(a / b));

  assert(((a - b) + b) == a);
  s = a-b; s = s+b;
  assert(s == a);
  s = a; s -= b; s += b;
  assert(s == a);

  assert((a / (-b)) == -(a / b));
  r = b; r = -r; s = a; s /= r;
  t = a; t /= b; t = -t;
  assert(s == t);

  // constructor:
  T t2 = a - b;
  T t3 = -a;
  t = a - b;
  assert(t2 == t);
  t = -a;
  assert(t3 == t);
  t = a; t = -t;
  assert(t3 == t);

  assert((a - b) == -(b - a));
  s = a-b; t = b-a; t = -t;
  assert(s == t);
  s = a; s -= b; t = b; t -= a; t = -t;
  assert(s == t);
}

template <class T>
void io_check(const T& a)
{
  ofstream fout("t.tmp");
  fout << print(a);
  fout.close();
  ifstream fin("t.tmp");
  T b;
  assert(b.scan(fin) == true);
  assert(a == b);
  fin.close();

  ofstream fout2("t.tmp");
  fout2 << a;
  fout2.close();
```

```
  ifstream fin2("t.tmp");
  ++b;                        // b != a
  fin2 >> b;
  assert(a == b);
}

template <class T, class D>
void check_block_generic(T a, T b, T c, const D d)
// T at least Integer
{
  identity_generic(b, a);
  identity_generic(c, b);
  identity_generic(a, c);
  identity_Natural_Digit2(a, max(a, b));
  identity_Natural_Digit2(b, max(a, b));
  identity_Natural_Digit2(a, max(a, c));
  identity_Natural_Digit2(c, max(a, c));
  identity_Natural_Digit2(b, max(b, c));
  identity_Natural_Digit2(c, max(b, c));
  identity_generic2(b, c, a);
  identity_generic2(c, a, b);
  identity_generic3(a, b, c);
  identity_generic3(c, a, b);
  io_check(a);
  io_check(b);
  io_check(c);
}

template <class T, class D>
void check_generic(T a, T b, T c, const D d)
{
  cout << ';' << flush;
  check_block_generic(T(-a), b, c, d);
  cout << ';' << flush;
  check_block_generic(a, T(-b), c, d);
  cout << ';' << flush;
  check_block_generic(T(-a), T(-b), c, d);
  cout << ';' << flush;
  check_block_generic(a, b, T(-c), d);
  cout << ';' << flush;
  check_block_generic(T(-a), b, T(-c), d);
  cout << ';' << flush;
  check_block_generic(a, T(-b), T(-c), d);
  cout << ';' << flush;
  check_block_generic(T(-a), T(-b), T(-c), d);
}

template <class T, class D>
void check_block(T a, T b, T c, const D d)
{
  identity_Natural_Digit(a, b);
  identity_Natural_Digit(b, c);
  identity_Natural_Digit(c, a);
  identity_Natural_Digit2(max(a, b), a);
  identity_Natural_Digit2(max(a, b), b);
  identity_Natural_Digit2(max(a, c), a);
```

```
  identity_Natural_Digit2(max(a, c), c);
  identity_Natural_Digit2(max(b, c), b);
  identity_Natural_Digit2(max(b, c), c);
  identity_Natural2(T((a+b+c)*113), a);
  identity_Natural(a, b, c);
  identity_Natural(c, a, b);
  identity_Natural_Digit(a, d);
  identity_Natural_Digit(b, d);
  identity_Natural_Digit(c, d);
  if (a <= d) { a *= d; a += 241; }
  identity_Natural_Digit2(a, d);
  io_check(a);
  io_check(b);
  io_check(c);
}

template <class T, class D>
void check(T a, T b, T c, const D d)
{
  cout << ':' << flush;
  check_block(T(-a), b, c, d);
  cout << ':' << flush;
  check_block(a, T(-b), c, d);
  cout << ':' << flush;
  check_block(T(-a), T(-b), c, d);
  cout << ':' << flush;
  check_block(a, b, T(-c), d);
  cout << ':' << flush;
  check_block(T(-a), b, T(-c), d);
  cout << ':' << flush;
  check_block(a, T(-b), T(-c), d);
  cout << ':' << flush;
  check_block(T(-a), T(-b), T(-c), d);
}

void checkConversion(const Natural& t)
{
  char* x = new char[t.length()*BETA+1];
  for (Digit j = 2; j <= 36; ++j) {
    Ntoa(t, x, j);
    assert(atoN(x, j) == t);
  }
  delete[] x;
}

int main(int argc, char** argv)
{
  if (argc == 1) {
    ITERATION       = 10;
    MAX_FACTORIAL   = 255;
    MAX_FIBONACCI   = 1000;
    MAX_RANDOM      = 1000;
    INNER_ITERATION = 1000;
  } else if (argc != 6) {
    cout << "arg: ITERATION       (= 10)\n";
    cout << "     MAX_FACTORIAL   (= 255)\n";
```

```
  cout << "     MAX_FIBONACCI   (= 1000)\n";
  cout << "     MAX_RANDOM      (= 1000)\n";
  cout << "     INNER_ITERATION (= 1000)\n";
  return 1;
} else {
  ITERATION       = atoi(argv[1]);
  MAX_FACTORIAL   = atoi(argv[2]);
  MAX_FIBONACCI   = atoi(argv[3]);
  MAX_RANDOM      = atoi(argv[4]);
  INNER_ITERATION = atoi(argv[5]);
  srand((unsigned)time(0));
}

Natural a1,a2,a3,a4;
Natural b1,b2,b3,b4;
Natural c1,c2,c3,c4;
size_t i;

cout << "Pass 1";
for (i = 0; i < ITERATION; ++i) {
  cout << '.' << flush;
  Digit k = rand() % MAX_FIBONACCI + 1;
  a1 = fibonacci(k);
  a2 = fibonacci2(k);
  a3 = fibonacci3(k);
  a4 = fibonacci4(k);
  assert(a1 == a2 && a2 == a3 && a3 == a4);
  checkConversion(a1);
  k = rand() % MAX_FIBONACCI + 1;
  b1 = fibonacci(k);
  b2 = fibonacci2(k);
  b3 = fibonacci3(k);
  b4 = fibonacci4(k);
  assert(b1 == b2 && b2 == b3 && b3 == b4);
  checkConversion(b1);
  k = rand() % MAX_FIBONACCI + 1;
  c1 = fibonacci(k);
  c2 = fibonacci2(k);
  c3 = fibonacci3(k);
  c4 = fibonacci4(k);
  assert(c1 == c2 && c2 == c3 && c3 == c4);
  checkConversion(c1);
  cout << ':' << flush;
  check_block(a1, b1, c1, k);
  check(Integer(a1), Integer(b1), Integer(c1), SignDigit(k));
  a2 = fibonacci(rand() % MAX_FIBONACCI + 1);
  b2 = fibonacci(rand() % MAX_FIBONACCI + 1);
  c2 = fibonacci(rand() % MAX_FIBONACCI + 1);
  check_generic(Integer(a2), Integer(b2), Integer(c2), SignDigit(k));
  check_generic(Rational(a1, a2), Rational(b1, b2), Rational(c1, c2), SignDigit(k));
}
cout << "\nPass 2";
for (i = 0; i < ITERATION; ++i) {
  cout << '.' << flush;
  Digit k = rand() % MAX_FACTORIAL + 1;
  a1 = factorial(k);
```

```
    a2 = factorial2(k);
    assert(a1 == a2);
    checkConversion(a1);
    Digit k2 = rand() % MAX_FACTORIAL + 1;
    b1 = factorial(k2);
    b2 = factorial2(k2);
    assert(b1 == b2);
    checkConversion(b1);
    k = rand() % MAX_FACTORIAL + 1;
    c1 = factorial(k);
    c2 = factorial2(k);
    assert(c1 == c2);
    checkConversion(c1);
    c4 = binomial(k, k2);
    Digit k3 = (k >= k2)? k-k2 : k2-k;
    assert(c4 == (c1/(b1*factorial(k3))));
    cout << ':' << flush;
    check_block(a1, b1, c1, k);
    check(Integer(a1), Integer(b1), Integer(c1), SignDigit(k));
    a2 = factorial(rand() % MAX_FACTORIAL + 1);
    b2 = factorial(rand() % MAX_FACTORIAL + 1);
    c2 = factorial(rand() % MAX_FACTORIAL + 1);
    check_generic(Integer(a2), Integer(b2), Integer(c2), SignDigit(k));
    check_generic(Rational(a1, a2), Rational(b1, b2), Rational(c1, c2), SignDigit(k));
  }
  cout << "\nPass 3";
  for (i = 0; i < ITERATION; ++i) {
    cout << '.' << flush;
    Digit k = rand() % MAX_RANDOM + 1;
    a1.rand(size_t(k));
    assert(log2(a1) < k);
    checkConversion(a1);
    k = rand() % MAX_RANDOM + 1;
    b1.rand(size_t(k));
    assert(log2(b1) < k);
    checkConversion(b1);
    k = rand() % MAX_RANDOM + 1;
    c1.rand(size_t(k));
    assert(log2(c1) < k);
    checkConversion(c1);
    cout << ':' << flush;
    check_block(a1, b1, c1, k);
    check(Integer(a1), Integer(b1), Integer(c1), SignDigit(k));
    a2.rand(rand() % MAX_RANDOM + 1);
    b2.rand(rand() % MAX_RANDOM + 1);
    c2.rand(rand() % MAX_RANDOM + 1);
    check_generic(Integer(a2), Integer(b2), Integer(c2), SignDigit(k));
    check_generic(Rational(a1, a2), Rational(b1, b2), Rational(c1, c2), SignDigit(k));
  }
  cout << endl;
  return 0;
}
◇
```

Anhang C

$\zeta(3)$-Rekord

http://www.cecm.sfu.ca/projects/ISC/records.html
http://www.lacim.uqam.ca/pi/records.html

Dear Simon Plouffe,

I send you Apery's constant to 128,000,026 decimal digits using the new formula

$$\text{Zeta}(3) = \frac{1}{24} \sum_{n \geq 0}^{\infty} (-1)^n \frac{A(n)\ ((2n+1)!\ (2n)!\ n!)^3}{(3n+2)!\ ((4n+3)!)^3}$$

$$\text{with}\quad A(n) := 126392\,n^5 + 412708\,n^4 + 531578\,n^3 + 336367\,n^2 + 104000\,n + 12463$$

given by Theodor Amdeberhan and Doron Zeilberger (see [1]).

Best Regards,

Sebastian Wedeniwski (wedeniws@de.ibm.com)

Timings:
========

The computation (of the not optimized code) that was done on December 13, 1998

```
took 39 hours and 22 minutes on the 32-bit machine

   IBM S/390 G5 CMOS (9672-RX6), ca 420 Mhz, 2 GB central storage,
                                 14 GB expanded storage,
                                 C/C++ Compiler for OS/390 Version 2.6,
                                 OS/390 Version 2.6.

This S/390 system has 10 processors.


A great machine!

It was a fine experience to work at this computer, especially to see, how fast
and parallel it handles with the giant mass of data (1120 GB)!



Thanks:
=======

This computational record was only possible with the machine support of
IBM Deutschland Entwicklung GmbH, Boeblingen.

Many thanks are going to my managers
Dr. Oskar von Dungern, Reinhold Krause and Joerg Thielges
for their immediate reaction and helpful coordination and to
Henry Koplien (IBM PowerPC) and Hans Dieter Mertiens (IBM S/390)
for their very cooperative and administrative aid. Besides it had not
been successful without the great help of Hans Dieter Mertiens on the
IBM S/390.



Verification:
=============

I used two different binary splitting and FFT algorithm.
The second computation was done on October 10, 1998, and it took about 2 weeks
on the following 64-bit emulating workstations:

1. IBM Power2 SC 135 MHz, 2 GB RAM, GNU C++ 2.8.0, AIX 4.1.5.
2. IBM PowerPC 604e 233 MHz, 1 GB RAM, GNU C++ 2.8.0, AIX 4.1.5.

and they agree with all 128,000,026 digits of the new record.



Technical Details:
==================

The computation of Zeta(3) was splitted up in 560 processes on 10 processors
and takes 20 GB main memory. During the computation it takes 16 GB hard disc
```

splitted up on 112 folders.

- partial computations:

```
 (0)   0*10^6  -   2*10^6  in 12295 sec
 (1)   2*10^6  -   4*10^6  in 13762 sec
 (2)   4*10^6  -   6*10^6  in 14413 sec
 (3)   6*10^6  -   8*10^6  in 14722 sec
 (4)   8*10^6  -  10*10^6  in 12491 sec
 (5)  10*10^6  -  12*10^6  in 16260 sec
 (6)  12*10^6  -  14*10^6  in 17873 sec
 (7)  14*10^6  -  16*10^6  in 16942 sec
 (8)  16*10^6  -  18*10^6  in 17927 sec
 (9)  18*10^6  -  20*10^6  in 17626 sec
(10)  20*10^6  -  22*10^6  in 18950 sec
(11)  22*10^6  -  24*10^6  in 19190 sec
(12)  24*10^6  -  26*10^6  in 19333 sec
(13)  26*10^6  -  28*10^6  in 18066 sec
(14)  28*10^6  -  30*10^6  in 18210 sec
(15)  30*10^6  -  32*10^6  in 18341 sec
(16)  32*10^6  -  34*10^6  in 19640 sec
(17)  34*10^6  -  36*10^6  in 19356 sec
(18)  36*10^6  -  38*10^6  in 19596 sec
(19)  38*10^6  -  40*10^6  in 19530 sec
(20)  40*10^6  -  42*10^6  in 19877 sec
(21)  42*10^6  -  44*10^6  in 19292 sec
(22)  44*10^6  -  46*10^6  in 16242 sec
(23)  46*10^6  -  48*10^6  in 19179 sec
(24)  48*10^6  -  50*10^6  in 21083 sec
(25)  50*10^6  -  52*10^6  in 20713 sec
(26)  52*10^6  -  54*10^6  in 20937 sec
(27)  54*10^6  -  56*10^6  in 20601 sec
(28)  56*10^6  -  58*10^6  in 21002 sec
(29)  58*10^6  -  60*10^6  in 18716 sec
(30)  60*10^6  -  62*10^6  in 14604 sec
(31)  62*10^6  -  64*10^6  in 13087 sec
(32)  64*10^6  -  66*10^6  in 21363 sec
(33)  66*10^6  -  68*10^6  in 20994 sec
(34)  68*10^6  -  70*10^6  in 21112 sec
(35)  70*10^6  -  72*10^6  in 20915 sec
(36)  72*10^6  -  74*10^6  in 21237 sec
(37)  74*10^6  -  76*10^6  in 18972 sec
(38)  76*10^6  -  78*10^6  in 14801 sec
(39)  78*10^6  -  80*10^6  in 13284 sec
(40)  80*10^6  -  82*10^6  in 21532 sec
(41)  82*10^6  -  84*10^6  in 21340 sec
(42)  84*10^6  -  86*10^6  in 21445 sec
(43)  86*10^6  -  88*10^6  in 21170 sec
(44)  88*10^6  -  90*10^6  in 20874 sec
(45)  90*10^6  -  92*10^6  in 18602 sec
(46)  92*10^6  -  94*10^6  in ????? sec
```

```
(47)  94*10^6  -  96*10^6  in ????? sec
(48)  96*10^6  -  98*10^6  in 21859 sec
(49)  98*10^6  - 100*10^6  in 21560 sec
(50)  100*10^6 - 102*10^6  in 21755 sec
(51)  102*10^6 - 104*10^6  in 21457 sec
(52)  104*10^6 - 106*10^6  in 21113 sec
(53)  106*10^6 - 108*10^6  in 18467 sec
(54)  108*10^6 - 110*10^6  in ????? sec
(55)  110*10^6 - 112*10^6  in ????? sec
(56)  112*10^6 - 114*10^6  in 21857 sec
(57)  114*10^6 - 116*10^6  in 21545 sec
(58)  116*10^6 - 118*10^6  in 21680 sec
(59)  118*10^6 - 120*10^6  in 21405 sec
(60)  120*10^6 - 122*10^6  in 21385 sec
(61)  122*10^6 - 124*10^6  in 19063 sec
(62)  124*10^6 - 126*10^6  in 14030 sec
(63)  126*10^6 - 128*10^6  in 13562 sec


- combinations (each of them parallelized in three processes):

 (0)- (1) in 1069 sec /  715 sec /  345 sec
 (2)- (3) in 1445 sec /  723 sec /  736 sec
 (4)- (5) in 1344 sec /  654 sec /  723 sec
 (6)- (7) in 1455 sec /  723 sec /  727 sec
 (8)- (9) in 1888 sec /  721 sec / 1168 sec
(10)-(11) in 2336 sec /  733 sec / 1161 sec
(12)-(13) in 1821 sec /  685 sec / 1145 sec
(14)-(15) in 1821 sec /  685 sec / 1145 sec
(16)-(17) in 2310 sec /  637 sec / 1056 sec
(18)-(19) in 2350 sec /  728 sec / 1161 sec
(20)-(21) in 1707 sec /  511 sec /  900 sec
(22)-(23) in 1697 sec /  518 sec /  869 sec
(24)-(25) in 2284 sec / 1112 sec / 1077 sec
(26)-(27) in 2057 sec /  958 sec /  892 sec
(28)-(29) in 1429 sec /  690 sec / 1163 sec
(30)-(31) in 1693 sec /  511 sec /  865 sec
(32)-(33) in 2256 sec / 1096 sec / 1079 sec
(34)-(35) in 1990 sec /  844 sec /  964 sec
(36)-(37) in ???? sec /  ??? sec /  ??? sec
(38)-(39) in ???? sec /  ??? sec /  ??? sec
(40)-(41) in 1649 sec /  534 sec / 1003 sec
(42)-(43) in 1935 sec /  843 sec /  982 sec
(44)-(45) in ???? sec /  ??? sec /  ??? sec
(46)-(47) in ???? sec /  ??? sec /  ??? sec
(48)-(49) in 2205 sec / 1002 sec / 1006 sec
(50)-(51) in 1734 sec /  976 sec /  980 sec
(52)-(53) in ???? sec /  ??? sec /  ??? sec
(54)-(55) in ???? sec /  ??? sec /  ??? sec
(56)-(57) in 1779 sec / 1049 sec / 1005 sec
(58)-(59) in 1818 sec /  921 sec /  978 sec
(60)-(61) in 1427 sec /  712 sec /  712 sec
```

```
(62)-(63) in ???? sec /  ??? sec /  ??? sec

 (0)- (3) in 3015 sec / 1516 sec / 1154 sec
 (4)- (7) in 3008 sec / 1462 sec / 1453 sec
 (8)-(11) in 3800 sec / 1286 sec / 2017 sec
(16)-(19) in 4273 sec / 1328 sec / 1884 sec
(24)-(27) in 4137 sec / 1825 sec / 2105 sec
(32)-(35) in 3996 sec / 2041 sec / 1926 sec
(40)-(43) in 3273 sec / 1314 sec / 1778 sec
(44)-(47) in ???? sec / ???? sec / ???? sec
(48)-(51) in 4041 sec / 2445 sec / 2464 sec
(52)-(55) in ???? sec / ???? sec / ???? sec
(56)-(59) in 4045 sec / 2111 sec / 2452 sec
(60)-(63) in ???? sec / ???? sec / ???? sec

 (0)- (7) in 4764 sec / 2415 sec / 2415 sec
 (8)-(15) in 6729 sec / 2425 sec / 4191 sec
(16)-(23) in 6764 sec / 2418 sec / 4167 sec
(24)-(31) in 9423 sec / 2428 sec / 4167 sec
(32)-(39) in ???? sec / ???? sec / ???? sec
(40)-(47) in ???? sec / ???? sec / ???? sec
(48)-(55) in ???? sec / ???? sec / ???? sec
(56)-(63) in ???? sec / ???? sec / ???? sec

 (0)-(15) in 14194 sec / 6193 sec /  6348 sec
(16)-(31) in 29289 sec / 6376 sec / 13479 sec
(32)-(47) in ????? sec / ???? sec / ????? sec
(48)-(63) in ????? sec / ???? sec / ????? sec
```

I have taken advantage of the binary splitting method (see [2]) with some extentions and my arithmetic Piologie Version 1.2.1 (see [3]), and have also simplified the Zeta(3) formula to

$$\text{Zeta}(3) = \frac{1}{192} \sum_{n \geq 0} \frac{(-1)^{n+1}\, A(n)\, (n!)^5}{B(n)}$$

$$\text{with } A(n) := 126392\, n^5 + 412708\, n^4 + 531578\, n^3 + 336367\, n^2 + 104000\, n + 12463,$$

$$B(n) := \prod_{k=1}^{n} (2k-1)^3 \quad .$$

```
             |     |           3         3
             |     |  (4 k + 1) (4 k + 3) (3 k + 1) (3 k + 2)
              k = 0
```

References:
==========

[1] T. Amdeberhan und D. Zeilberger: Hypergeometric Series Acceleration via the WZ Method, Electronic Journal of Combinatorics (Wilf Festschrift Volume) 4 (1997).

[2] B. Haible, T. Papanikolaou: Fast multiprecision evaluation of series of rational numbers, Technical Report TI-97-7, Darmstadt University of Technology, April 1997.

[3] S. Wedeniwski: Piologie - Eine exakte arithmetische Bibliothek in C++, Technical Report WSI 96-35, Tuebingen University, available by anonymous ftp from "ftp://ftp.informatik.uni-tuebingen.de/pub/CA/software/Piologie/" or "ftp://ruediger.informatik.uni-tuebingen.de/Piologie/".

"makesh.cpp" 382 ≡

```
////////////////////////////////
//
// Piologie V 1.3
// multi-precision arithmetic
// Generate shell files for
// Zeta(3) record
//
// (c) 1996-1999 HiPiLib
//               www.hipilib.de
//
// Sebastian Wedeniwski
// 12/13/1999
//

#include <string.h>
#include <fstream.h>
#include <stdio.h>
#include <stdlib.h>


int main(int argc, char** argv)
{
  if (argc != 3) {
    cerr << "useage:  makesh <processes> <steps>\nexample:  makesh 4 8\n";
    return 1;
  }

  const int n = atoi(argv[1]);
  const int steps = atoi(argv[2]);
```

```
if (n < 4 || n > steps || steps < 4 || steps%n != 0) {
  cerr << "error!\n";
  return 1;
}
if ((steps & (~steps+1)) != steps || (n & (~n+1)) != n) {
  cerr << "error2!\n";
  return 1;
}

char filename[100];
int start = 0;
int stop = steps/n;
int i;
ofstream fout("x.sh");
fout << "#!/bin/sh\n";
for (i = 0; i < n; ++i) {
  fout << "sh x" << i << ".sh $1 | tee -a x" << i << ".log ";
  if (i < n-1) fout << '&';
  fout << '\n';
}
fout.close();
for (i = 0; i < n; ++i) {
  sprintf(filename, "x%d.sh", i);
  ofstream fout(filename);

  fout << "#!/bin/sh\ndate\n";
  int j,step;
  for (j = start; j < stop; ++j) {
    fout << "mkdir " << j << "\ncd " << j
         << "; ../zeta $1 " << j << ' ' << steps << "; cd ..\ndate\n";
  }
  for (step = 2; step < stop-start; step <<= 1) {
    for (j = start; j < stop; j += step) {
      fout << "mkdir " << j  << '-' << (j+step-1)
           << "\ncd " << j << '-' << (j+step-1) << '\n';
      if (step == 2) {
        fout << "../zeta 1 ../" << j << "/ ../" << (j+step-1) << "/\ndate\n";
        fout << "../zeta 2 ../" << j << "/ ../" << (j+step-1) << "/\ndate\n";
        fout << "../zeta 3 ../" << j << "/ ../" << (j+step-1) << "/\ndate\n";
      } else {
        fout << "../zeta 1 ../" << j << '-' << (j+step/2-1)
             << "/ ../" << (j+step/2) << '-' << (j+step-1) << "/\ndate\n";
        fout << "../zeta 2 ../" << j << '-' << (j+step/2-1)
             << "/ ../" << (j+step/2) << '-' << (j+step-1) << "/\ndate\n";
        fout << "../zeta 3 ../" << j << '-' << (j+step/2-1)
             << "/ ../" << (j+step/2) << '-' << (j+step-1) << "/\ndate\n";
      }
      fout << "cd ..\n";
    }
  }
  if (j >= step) {
    j -= step;
    fout << "mkdir " << j  << '-' << (j+step-1)
         << "\ncd " << j << '-' << (j+step-1) << '\n';
    if (step == 2) {
      fout << "../zeta 1 ../" << j << "/ ../" << (j+step-1) << "/\ndate\n";
```

```
            if (j != stop-1)
              fout << "../zeta 2 ../" << j << "/ ../" << (j+step-1) << "/\ndate\n";
            fout << "../zeta 3 ../" << j << "/ ../" << (j+step-1) << "/\ndate\n";
          } else {
            fout << "../zeta 1 ../" << j << '-' << (j+step/2-1)
                 << "/ ../" << (j+step/2) << '-' << (j+step-1) << "/\ndate\n";
            if (j != stop-1)
              fout << "../zeta 2 ../" << j << '-' << (j+step/2-1)
                   << "/ ../" << (j+step/2) << '-' << (j+step-1) << "/\ndate\n";
            fout << "../zeta 3 ../" << j << '-' << (j+step/2-1)
                 << "/ ../" << (j+step/2) << '-' << (j+step-1) << "/\ndate\n";
          }
          fout << "../zeta $1 " << steps << ' ' << steps << " 0\n";
          if (j != stop-1) fout << "../zeta $1 " << steps << ' ' << steps << " 1\n";
          fout << "cd ..\n";
        }

        if (i == n-1) {
          for (int j = steps-step; j > 0; j -= step) {
            fout << "cd " << (j-step) << '-' << (j-1);
            fout << "\n../zeta 4 ./ ../" << j << '-' << (j+step-1) << "/zeta.tmp";
            if (j == steps-step) fout << '0';
            fout << "\n../zeta 5 ./ ./\n";
            if (j == step) fout << "../zeta $1\n";
            fout << "cd ..\n";
          }
        }
        start = stop; stop += steps/n;
      }

      return 0;
    }
```

◇

"zeta.cpp" 384 ≡

```
////////////////////////////////
//
// Piologie V 1.3
// multi-precision arithmetic
// Program for Zeta(3) record
//
// (c) 1996-1999 HiPiLib
//               www.hipilib.de
//
// Sebastian Wedeniwski
// 12/13/1999
//

#include "pi.h"
#include "integer.h"

#ifdef _Old_STD_
# include <fstream.h>
#else
# include <fstream>
using namespace std;
#endif
```

```
#include <string.h>
#include <time.h>



//#define _Old_Formula_         // Use old formula
//#define _FAST_OUTPUT_         // hex-output



#define LOG(a)  { cout << a << flush; \
                { ofstream lout("../zeta.log", ios::app); \
                  lout << a; } }
#define LOGLN(a)  { cout << a << endl; \
                  { ofstream lout("../zeta.log", ios::app); \
                    lout << a << endl; } }


#ifdef _Old_Formula_
const double ITERATIONS = 3.01;
#else
const double ITERATIONS = 5.03;
#endif


struct Zeta {
  Fixed zeta;

  void series(Digit, const Digit, Integer&, Integer&, Integer&) const;

  Zeta(const size_t, int);
  Zeta(const size_t, const size_t, const size_t, const Digit);

  void fast_output(OSTREAM&);
  void fast_input(ISTREAM&);

  const Fixed& value() const;
};

inline const Fixed& Zeta::value() const
{
  return zeta;
}

inline OSTREAM& operator<<(OSTREAM& out, const Zeta& a)
{
  return out << a.zeta;
}


void Zeta::series(Digit n, const Digit m, Integer& p, Integer& q, Integer& t) const
// n < 2^BETA/72-24
{
  // CONDITION(n < m);
```

```
  if (n+1 == m) {

#ifdef _Old_Formula_

    if (n == 0) { p = 1; q = 32; t = 77; }
    else {
      if (n <= GAMMA_LOW/2) { p = n*n; p *= n*n; }
      else { p = n; p = p*p; p = p*p; }
      p *= n; p = -p;
      if (n < GAMMA/414) t = 205*n+250;
      else { t = n; t *= 205; t += 250; }
      t *= n; t += 77; t *= p;
      n = 2*n+1;
      if (n <= GAMMA_LOW/2) { q = n*n; q *= n*n; }
      else { q = n; q = q*q; q = q*q; }
      q *= 32*n;
    }

#else

    if (n == 0) { p = 1; q = 10368; t = 12463; }
    else {
      if (n <= GAMMA_LOW/2) { p = n*n; p *= n*n; }
      else { p = n; p = p*p; p = p*p; }
      const Digit k = 2*n-1;
      if (n <= GAMMA_LOW/8) { p *= k*n; p *= k*k; }
      else { p *= n; p *= k; p *= k; p *= k; }
      p = -p;
      t = 126392; t *= n;
      t += 412708; t *= n;
      t += 531578; t *= n;
      t += 336367; t *= n;
      t += 104000; t *= n;
      t += 12463; t *= p;

      const Digit k2 = 4*n+1;
      const Digit k3 = k2+2;
      if (k3 <= GAMMA_LOW/2) { q = k2*k2; q *= k2*k3; q *= k3*k3; }
      else {
        q = k2; q *= k2; q *= k2;
        q *= k3; q *= k3; q *= k3;
      }
      q *= 72*n+24; q *= 3*n+2;
    }

#endif

  } else {
    Integer p1,q1,t1;
    Integer p2,q2,t2;

    const Digit l = (n+m)/2;
    series(n, l, p1, q1, t1);
    series(l, m, p2, q2, t2);
    t = t1*q2; t += t2*p1;
    p = p1*p2; q = q1*q2;
```

```
    const Digit k = m-n;
    if (k > 8 && k <= 16) {
      p1 = gcd(p, gcd(t, q));
      p /= p1; q /= p1; t /= p1;
    }
  }
}

static void format(bool point)
{
  ifstream fin("zeta3.tmp");
  ofstream fout("zeta3-f.tmp");
  int i = 0;
  char c;
  if (point)
    while (!fin.eof()) {
      if (++i == 2) { fout << '.'; continue; }
      if (!fin.get(c)) break;
      fout.put(c);
      if (i == 78) { fout << "\\\n"; i = 0; break; }
    }
  while (!fin.eof()) {
    if (!fin.get(c)) break;
    fout.put(c);
    if (++i == 78) { fout << "\\\n"; i = 0; }
  }
}

static void reduction(Integer& p, Integer& q, Integer& t)
{
  size_t i = 0;
  if (p != 0)
    while (!abs(p).testbit(i)) ++i;
  size_t j = 0;
  if (q != 0)
    while (!abs(q).testbit(j)) ++j;
  size_t k = 0;
  if (t != 0)
    while (!abs(t).testbit(k)) ++k;
  if (i > j) i = j;
  if (i > k) i = k;
  p >>= i; q >>= i; t >>= i;
}

Zeta::Zeta(const size_t n, int idx)
 : zeta(n)
{
  if (idx == 0) {
    ifstream fin("zeta.tmp");
    size_t x;
    fin >> x;
    if (x != zeta.precision()) { LOG("Note (different BETA)!\n"); }
    char c;
    fin.get(c);
    fin >> x;
```

```
    if (x != zeta.decimals()) { LOG("Error!\n"); return; }
    fin.get(c);
    if (!zeta.value().scan(fin)) { LOG("Error!\n"); return; }

//    zeta.value() >>= 64;        // BETA: 64 -> 32

#ifdef _Old_Formula_
    zeta.value() >>= 1;
#endif

  } else if (idx == 1) {
    ifstream fin("zeta3.tmp2");
    fast_input(fin);
  } else LOGLN("ERROR!");
}

Zeta::Zeta(const size_t n, const size_t idx, const size_t max, const Digit st)
 : zeta(n)
{
  time_t start,stop;
  const size_t sz = zeta.precision();

  Integer p,q,t;
  if (idx < max) {
    const Digit m = Digit(n/ITERATIONS);
    const Digit k = m/max;
    const Digit i = k*idx;

    start = time(0);
    if (i+2*k > m) {
      LOGLN("process:" << idx << ", from " << ((st)? st : i) << " to " << m);
      series((st)? st : i, m, p, q, t);
    } else {
      LOGLN("process:" << idx << ", from " << ((st)? st : i) << " to " << i+k);
      series((st)? st : i, i+k, p, q, t);
    }
    stop = time(0);
    LOGLN("zeta time [s] = " << difftime(stop, start));

    reduction(p, q, t);

    ofstream foutp("zeta-p.tmp");
    foutp << print(p);
    foutp.close();

    ofstream foutq("zeta-q.tmp");
    foutq << print(q);
    foutq.close();

    ofstream foutt("zeta-t.tmp");
    foutt << print(t);
    foutt.close();
  } else {
    switch (st) {
    case 0: {
      ifstream finq("zeta-q.tmp");
```

```
      ifstream fint("zeta-t.tmp");
      q.scan(finq);
      t.scan(fint);

      const bool sign = (q <= 0 && t <= 0 || q >= 0 && t >= 0);

      LOGLN("process division");
      start = time(0);
      size_t i = 0;
      while (!abs(q).testbit(i)) ++i;
      q >>= i; t <<= BETA*sz-i;
      zeta.value() = abs(t) / abs(q);
      stop = time(0);
      LOGLN("zeta time [s] = " << difftime(stop, start));
      ofstream fout("zeta.tmp0");
      fout << zeta.precision() << ',' << zeta.decimals() << ',';
      if (!sign) fout << '-';
      fout << print(zeta.value());
      break;
    }
    case 1: {
      ifstream finq("zeta-q.tmp");
      ifstream finp("zeta-p.tmp");
      q.scan(finq);
      p.scan(finp);

      const bool sign = (q <= 0 && p <= 0 || q >= 0 && p >= 0);

      LOGLN("process division");
      start = time(0);
      size_t i = 0;
      while (!abs(q).testbit(i)) ++i;
      q >>= i; p <<= BETA*sz-i;
      zeta.value() = abs(p) / abs(q);
      stop = time(0);
      LOGLN("zeta time [s] = " << difftime(stop, start));
      ofstream fout("zeta.tmp1");
      fout << zeta.precision() << ',' << zeta.decimals() << ',';
      if (!sign) fout << '-';
      fout << print(zeta.value());
      break;
    }
    default: LOG("ERROR!\n");
    }
  }
}

void Zeta::fast_output(OSTREAM& fout)
{
  const size_t SIZE = zeta.decimals();
  size_t m = zeta.precision();
  Natural c = pow(Natural(10), zeta.decimals());
  zeta.value() *= c;
  zeta.value().rmove(m);
  char* result = new char[SIZE+1];
  Ntoa(zeta.value(), result, 16);
```

```
  fout << result;
  delete[] result;
}

void Zeta::fast_input(ISTREAM& fin)
{
  const size_t SIZE = zeta.decimals();
  size_t m = zeta.precision();
  char* a = new char[SIZE+1];
  char* b = a;

  while (!fin.eof()) {
    char c;
    if (!fin.get(c) || c == '\n') break;
    *b++ = c;
  }
  *b = 0;
  zeta.value() = atoN(a, 16);
  delete[] a;
}

static size_t read(const char* path, const char* name,
                   Integer& q, const bool prec, size_t& decimals)
{
  char str[100];
  ifstream fin(strcat(strcpy(str, path), name));
  LOG(name << " read...");

  size_t precision = 0;
  if (prec) {
    char c;
    fin >> precision;
    fin.get(c);
    fin >> decimals;
    fin.get(c);
  }
  if (q.scan(fin)) { LOGLN("...ok!"); }
  else { LOG("...ERROR!\n"); }
  return precision;
}

static void combine(const char* path1, const char* path2, int idx)
{
  time_t start,stop;

  switch (idx) {
  case 1: {
    Integer q,t,p1,t1,z;
    size_t x;
    read(path1, "zeta-p.tmp", p1, false, x);
    read(path1, "zeta-t.tmp", t1, false, x);
    read(path2, "zeta-q.tmp", q, false, x);
    read(path2, "zeta-t.tmp", z, false, x);

    LOGLN("combine1");
    start = time(0);
```

```
  t = z*p1;
  LOGLN("combine2");
  t += z = t1*q;
  stop = time(0);
  LOGLN("zeta time [s] = " << difftime(stop, start));

  ofstream fout("zeta-t.tmp");
  fout << print(t);
  break;
}
case 2: {
  Integer p,p1;
  size_t x;
  read(path1, "zeta-p.tmp", p1, false, x);
  read(path2, "zeta-p.tmp", p, false, x);

  LOGLN("combine" << idx);
  start = time(0);
  p *= p1;
  stop = time(0);
  LOGLN("zeta time [s] = " << difftime(stop, start));

  ofstream fout("zeta-p.tmp");
  fout << print(p);
  break;
}
case 3: {
  Integer q,q1;
  size_t x;
  read(path1, "zeta-q.tmp", q1, false, x);
  read(path2, "zeta-q.tmp", q, false, x);

  start = time(0);
  LOGLN("combine" << idx);
  q *= q1;
  stop = time(0);
  LOGLN("zeta time [s] = " << difftime(stop, start));

  ofstream foutq("zeta-q.tmp");
  foutq << print(q);
  break;
}
case 4: {
  Integer pq,tq;
  size_t decimals;
  size_t precision = read(path1, "zeta.tmp1", pq, true, decimals);
  if (precision != read(path2, "", tq, true, decimals)) {
    LOGLN("precision error!");
    return;
  }

  LOGLN("combine" << idx);
  start = time(0);
  pq *= tq;
  pq.rmove(precision);
  stop = time(0);
```

```
    LOGLN("zeta time [s] = " << difftime(stop, start));

    ofstream foutq("zeta.tmp2");
    foutq << precision << ',' << decimals << ',' << print(pq);
    break;
  }
  case 5: {
    Integer s1,s2;
    size_t decimals;
    size_t precision = read(path1, "zeta.tmp0", s1, true, decimals);
    if (precision != read(path2, "zeta.tmp2", s2, true, decimals)) {
      LOGLN("precision error!");
      return;
    }

    LOGLN("combine" << idx);
    start = time(0);
    s1 += s2;
    stop = time(0);
    LOGLN("zeta time [s] = " << difftime(stop, start));

    ofstream foutq("zeta.tmp");
    foutq << precision << ',' << decimals << ',' << print(s1);
    break;
  }
  default: LOG("ERROR!\n");
  }
}

static void pre_output(ostream& out, const Natural& a, size_t n)
{
  cout << "level " << n << endl;
  if (n <= 1000) {
    out.width(n);
    out.fill('0');
    out << a;
  } else {
    const size_t m = n/2;
    Natural q;
    cout << "pow" << endl;
    Natural r = pow(Natural(10), m);
    cout << "div" << endl;
    div(a, r, q, r);
    pre_output(out, q, m + (n&1));
    pre_output(out, r, m);
  }
}


int main(int argc, char** argv)
{
  size_t i,j,n;
  Digit  m = 0;

  for (i = 0; i < argc; ++i) LOG(argv[i] << ' ');
```

```
  if (argc == 4) {
    n = atoi(argv[1]);
    if (n >= 1 && n <= 5) {
      combine(argv[2], argv[3], n);
      return 0;
    }
    i = atoi(argv[2]);
    j = atoi(argv[3]);
  } else if (argc == 5) {
    n = atoi(argv[1]);
    i = atoi(argv[2]);
    j = atoi(argv[3]);
    m = atoi(argv[4]);
  } else if (argc == 3) {
    i = atoi(argv[2]);
    Zeta z(atoi(argv[1]), i);
    time_t start,stop;
    if (i == 0) {
      ofstream fout("zeta3.tmp2");
      start = time(0);
      z.fast_output(fout);
      stop = time(0);
    } else if (i == 1) {
      ofstream fout("zeta3.tmp");
      start = time(0);
      pre_output(fout, z.value().value(), z.value().decimals());
      stop = time(0);
      fout.close();
      format(true);
    } else LOGLN("ERROR!\n");
    LOGLN("zeta time [s] = " << difftime(stop, start));
    return 0;
  } else if (argc == 2) {
    Zeta z(atoi(argv[1]), 0);
    time_t start,stop;
    ofstream fout("zeta3.tmp");
    start = time(0);
    fout << z << endl;
    stop = time(0);
    LOGLN("zeta time [s] = " << difftime(stop, start));
    fout.close();
    format(false);
    return 0;
  } else {
    cout << "useage:  zeta <decimals> <step> <parts> [start]\n";
    cout << "         zeta <decimals> <step> <parts> <idx>\n";
    cout << "         zeta <decimals> <idx>\n";
    cout << "         zeta <decimals>\n";
    cout << "      or zeta <idx> <path1> <path2>\n";
    return 1;
  }

  Zeta z(n, i, j, m);

  return 0;
}
```

$\diamond$

Anhang D

Implementation

D.1 Digit

⟨ included files of the file "digit.h" 395 ⟩ ≡

```
#ifdef _Old_STD_
```

⟨ standard system include file <stdlib.h> 4c ⟩

```
#include <iostream.h>
#include <ctype.h>
#include <assert.h>
```

⟨ standard system include file <limits.h> 3c ⟩

```
# define bool int
# define true 1
# define false 0

# define OSTREAM ostream
# define ISTREAM istream

#else
// New standard

# include <iostream>
# include <cstdlib>
# include <cctype>
# include <cassert>
# include <climits>

# include <algorithm>
# include <utility>

# define OSTREAM std::ostream
# define ISTREAM std::istream

#endif
```

◇

Macro referenced in 2.

⟨ macro definitions of the file "digit.h" 396a ⟩ ≡
```
#define _DigitAsm_          // use assembler if ever it is possible
//#define _Piologie_Debug_    // Debug modus for assert
```
◇
Macro referenced in 2.

⟨ declarations of error-handling 396b ⟩ ≡
```
// error-handling:
typedef void (*piologie_error_handler_t)(const int, const char*);
piologie_error_handler_t set_piologie_error_handler(piologie_error_handler_t);
```
◇
Macro referenced in 2.

⟨ function prototypes of the file "digit.h" 396c ⟩ ≡
```
#ifndef STATIC_VS_INLINE        // some compiler have problems with inline
#define STATIC_VS_INLINE inline
#endif

STATIC_VS_INLINE Digit log2(Digit);
Digit pow10(size_t);
inline void  swap(Digit&, Digit&);
inline Digit sqrt(Digit, Digit);
Digit sqrt(Digit);
void  sqrt(Digit, Digit&, Digit&);
Digit sqrt(Digit, Digit, Digit&, Digit&);
Digit gcd(Digit, Digit);
```
◇

Macro referenced in 2.

⟨ constants of PIOLOGIE 396d ⟩ ≡
```
#ifdef DEFINE_VS_CONST        // some compiler have problems with const
# define BETA       (sizeof(Digit)*CHAR_BIT)
# define GAMMA      (~Digit(0))
# define GAMMA_LOW  (GAMMA >> (BETA/2))
# define GAMMA_HIGH (~GAMMA_LOW)
#else
```
⟨ constant β 3b ⟩
⟨ constant γ 3d ⟩
⟨ constant γ_{low} and γ_{high} 86c ⟩
```
#endif
```
⟨ constant δ 4a ⟩
⟨ constant α 4b ⟩
◇
Macro referenced in 2.

⟨ new operator 396e ⟩ ≡
```
#ifndef _Old_STD_
using namespace std;
# define NOTHROW_NEW  new(nothrow)
#else
# define NOTHROW_NEW  new
#endif
```
◇
Macro referenced in 298, 398b, 409, 433.

D.2 NumberBase

⟨ protected attributes 397a ⟩ ≡
```
static Digit* k_ary_gcd_inverse;
static Digit* k_ary_gcd_linear_id;
static bool* k_ary_gcd_linear_id_neg;
static Digit* k_ary_gcd_trial_division;
static size_t k_ary_gcd_trial_division_size;
```
◇
Macro referenced in 6a.

⟨ protected function prototypes 397b ⟩ ≡
```
size_t* quad_convergence_sizes(const size_t, size_t&) const;

void digitmul(const Digit, const Digit, const Digit, Digit*) const;
void digitmul(const Digit, const Digit, const Digit, const Digit, Digit*) const;
void digitsqr(const Digit, const Digit, Digit*) const;
```
◇
Macro referenced in 6a.

⟨ function prototypes for Digitoperations in NumberBase 397c ⟩ ≡
```
void digitmul(const Digit, const Digit, Digit&, Digit&) const;
void digitdiv(const Digit, const Digit, const Digit, Digit&, Digit&) const;
void digitmod(Digit a, Digit b, const Digit c, Digit& r) const;

void gcd(Digit, Digit, Digit, Digit, Digit&, Digit&) const;
```
◇
Macro referenced in 6a.

⟨ inline implementation of the file "digit.h" 397d ⟩ ≡
```
////////////////// Inline-Implementation ////////////////////////
```
⟨ default constructor of NumberBase 400a ⟩
⟨ destructor of NumberBase 400b ⟩
```
#ifdef _DigitAsm_
```
⟨ digitmul and digitdiv in assembler 398a ⟩
```
#endif // _DigitAsm_

#ifndef _DigitAsm_
```
⟨ digitmul in ANSI-C++ 87a ⟩
```
#endif
```
⟨ digitmod of a double Digit 152c ⟩
⟨ digitmul of a Digit with a double Digit 89b ⟩
⟨ digitmul of a double Digit 89c ⟩
⟨ digitsqr of a double Digit 90 ⟩
⟨ binary logarithm for Digits 147b ⟩
⟨ the function swap for Digits 5 ⟩
⟨ square root of a double Digit 171 ⟩
```
#if defined(_Old_STD_) && !defined(__MINMAX_DEFINED) || defined(_MSC_VER)

#ifdef min
# undef min
#endif
#ifdef max
# undef max
#endif
```

```
template <class T>
inline const T& min(const T& a, const T& b)
{
  return (a < b)? a : b;
}

template <class T>
inline const T& max(const T& a, const T& b)
{
  return (a < b)? b : a;
}
#else
# define min std::min
# define max std::max
#endif
```

⟨ binder for the arguments of an operator 34a ⟩
◇

Macro referenced in 2.

⟨ digitmul and digitdiv in assembler 398a ⟩ ≡

```
#if defined(__386__) && defined(__WATCOMC__)
```
⟨ digitmul in assembler for a i386 processor with the Watcom-Compiler 88a ⟩
⟨ digitdiv in assembler for a i386 processor with the Watcom-Compiler 151b ⟩
```
#elif _M_IX86 >= 300 && _MSC_VER
```
⟨ digitmul in assembler for a i386 processor with the Visual-Compiler 87b ⟩
⟨ digitdiv in assembler for a i386 processor with the Visual-Compiler 151a ⟩
```
#elif (defined (__i386__) || defined (__i486__)) && defined (__GNUC__)
```
⟨ digitmul in assembler for a i386 processor with the GNU-Compiler 88b ⟩
⟨ digitdiv in assembler for a i386 processor with the GNU-Compiler 152a ⟩
```
#elif defined (__sparc_v8__) && defined (__GNUC__)
```
⟨ digitmul in assembler for a SPARC v8 processor with the GNU-Compiler 89a ⟩
⟨ digitdiv in assembler for a SPARC v8 processor with the GNU-Compiler 152b ⟩
```
#else
# undef _DigitAsm_
#endif
```
◇

Macro referenced in 397d.

"digit.cpp" 398b ≡

```
////////////////////////////////
//
// Piologie V 1.3.2
// multi-precision arithmetic
// Digit / NumberBase
//
// (c) 1996-2001 HiPiLib
//               www.hipilib.de
//
// Sebastian Wedeniwski
// 10/21/2001
//

#include "digit.h"
```

```
#include "nmbrthry.h"
#ifdef _NEEDS_PIOLOGIE_KEY_
# include "key.h"
#endif
```

⟨ macro definitions for internal conditions 6b ⟩
⟨ macro definitions for internal memory manipulations 9 ⟩
⟨ new operator 396e ⟩

```
//////////////////// Class NumberBase ////////////////////////////

Digit* NumberBase::k_ary_gcd_inverse = 0;
Digit* NumberBase::k_ary_gcd_linear_id = 0;
bool* NumberBase::k_ary_gcd_linear_id_neg = 0;
Digit* NumberBase::k_ary_gcd_trial_division = 0;
size_t NumberBase::k_ary_gcd_trial_division_size = 0;


class NumberBase_init : private NumberBase {
private:
  static NumberBase_init CheckNumberBase;

public:
  NumberBase_init(char);
  ~NumberBase_init();
};

NumberBase_init NumberBase_init::CheckNumberBase(' ');

NumberBase_init::NumberBase_init(char)
{
  // Testing the bits for low ending
  {
    ⟨ testing the bits for low ending 68a ⟩
  }
  // Testing size of Digit and SignDigit
  ⟨ testing size of Digit and SignDigit 186a ⟩
  // precomputation phase for k-ary gcd
  ⟨ initialize gcd 401 ⟩
#ifdef _NEEDS_PIOLOGIE_KEY_
  checkPiologieKey();
#endif
}

NumberBase_init::~NumberBase_init()
{
  delete[] k_ary_gcd_inverse;
  delete[] k_ary_gcd_linear_id;
  delete[] k_ary_gcd_linear_id_neg;
  delete[] k_ary_gcd_trial_division;
  k_ary_gcd_inverse = 0;
  k_ary_gcd_linear_id = 0;
  k_ary_gcd_linear_id_neg = 0;
  k_ary_gcd_trial_division = 0;
}
```

⟨ getting sizes for quadratic convergence algorithm 163a ⟩
⟨ definitions of error-handling 35 ⟩

```
#ifndef _DigitAsm_
```
⟨ digitdiv in ANSI-C++ 150 ⟩
```
#endif
```

⟨ calculates 10 to the power of a Digit 402a ⟩
⟨ square root of a Digit 168 ⟩
⟨ square root and remainder of a Digit 169a ⟩
⟨ square root and remainder of a double Digit 169b ⟩
⟨ greatest common divisor of two Digits 268 ⟩
⟨ greatest common divisor of two double Digits 269 ⟩
◇

⟨ default constructor of NumberBase 400a ⟩ ≡
```
inline NumberBase::NumberBase()
{
}
```
⟨ destructor of NumberBase 400b ⟩ ≡
◇

Macro referenced in 397d.

```
inline NumberBase::~NumberBase()
{
}
```
◇

Macro referenced in 397d.

⟨ initialize gcd 401 ⟩ ≡

```
const Digit K_ARY = Digit(1) << ((BETA > 32)? 16 : BETA/2);
const Digit K_BASE = K_ARY-1;

size_t k = K_ARY >> 1;
k_ary_gcd_inverse = NOTHROW_NEW Digit[k];
if (!k_ary_gcd_inverse) errmsg(2, "(precomputation phase for k-ary gcd)");
size_t i;
for (i = 0; i < k; ++i) k_ary_gcd_inverse[i] = inverse(Digit(i+i+1), K_ARY);
k_ary_gcd_linear_id = NOTHROW_NEW Digit[K_ARY];
k_ary_gcd_linear_id_neg = NOTHROW_NEW bool[K_ARY];
if (!k_ary_gcd_linear_id || !k_ary_gcd_linear_id_neg) errmsg(2, "(precomputation phase for k-ary gcd)");
k = sqrt(K_ARY);
for (i = 1; i < K_ARY; ++i) {
  for (Digit a = 1; a <= k; ++a) {
    const Digit b = (a*i)&K_BASE;
    const Digit c = K_ARY - b;
    if (b <= k || c <= k) { k_ary_gcd_linear_id[i] = a; k_ary_gcd_linear_id_neg[i] = (b <= k); break; }
  }
}

k_ary_gcd_trial_division = NOTHROW_NEW Digit[++k];
if (!k_ary_gcd_trial_division) errmsg(2, "(precomputation phase for k-ary gcd)");
k_ary_gcd_trial_division_size = 0;
k_ary_gcd_trial_division[k_ary_gcd_trial_division_size] = 1;
Digit a;
Primes p;
p.firstPrime();
while (p.nextPrime(a) && a <= k) {
  Digit d1,d2;
  digitmul(a, k_ary_gcd_trial_division[k_ary_gcd_trial_division_size], d1, d2);
  if (d1) k_ary_gcd_trial_division[++k_ary_gcd_trial_division_size] = a;
  else k_ary_gcd_trial_division[k_ary_gcd_trial_division_size] = d2;
}
Digit* d = NOTHROW_NEW Digit[++k_ary_gcd_trial_division_size];
if (!d) errmsg(2, "(precomputation phase for k-ary gcd)");
const Digit* dE = d+k_ary_gcd_trial_division_size;
COPY(d, k_ary_gcd_trial_division, d, dE);
k_ary_gcd_trial_division -= k_ary_gcd_trial_division_size;
delete[] k_ary_gcd_trial_division;
k_ary_gcd_trial_division = d-k_ary_gcd_trial_division_size;
◇
```

Macro referenced in 398b.

⟨ calculates 10 to the power of a Digit 402a ⟩ ≡

```
Digit pow10(size_t a)
// Algorithm:  b := pow10(a)
// Input:      a in size_t where a <= ALPHA_WIDTH.
// Output:     b in Digit such that b = 10^a ||
{
  CONDITION(a <= ALPHA_WIDTH);

  static const Digit c[10] = {1, 10, 100, Digit(1000), Digit(10000),
                              Digit(100000), Digit(1000000),
                              Digit(10000000), Digit(100000000),
                              Digit(1000000000) };

  if (ALPHA_WIDTH < 10 || a < 10) return c[a];
  Digit b = c[9];
  for (a -= 9; a >= 9; a -= 9) b *= c[9];
  return b*c[a];
}
◇
```

Macro referenced in 398b.

D.3 Natural

"natural.h" 402b ≡

```
////////////////////////////////
//
// Piologie V 1.3.2
// multi-precision arithmetic
// Natural
//
// (c) 1996-2001 HiPiLib
//               www.hipilib.de
//
// Sebastian Wedeniwski
// 10/21/2001
//

#ifndef _Include_Natural_H_
#define _Include_Natural_H_

#include "digit.h"

#ifdef setbit       // some "math.h" libraries have a defined setbit
# undef setbit
#endif

struct Natural_plus_tag {};
struct Natural_minus_tag {};
struct Natural_multiplies_tag {};
struct Natural_divides_tag {};
struct Natural_modulus_tag {};
struct Natural_square_root_tag {};
struct Natural_lshift_tag {};
struct Natural_rshift_tag {};
struct Natural_and_tag {};
struct Natural_or_tag {};
```

```
struct Natural_xor_tag {};
struct Natural_not_tag {};


class Natural : public NumberBase {
public:
  ⟨ output variables for representation of a Natural 38a ⟩
private:
  ⟨ variables for representation of a Natural 16 ⟩
  ⟨ static variables for memory management of Naturals 20b ⟩

  void          inc(Digit*);
  void          add_with_inc(const Digit*, Digit*, const Digit*);
  void          add_with_inc(const Digit*, Digit*, const Digit*, const Digit*);
  bool          add_no_inc(const Digit*, Digit*, const Digit*) const;
  bool          add_no_inc(const Digit*, Digit*, const Digit*, const Digit*) const;
  void          sub(Digit*, Digit*, const Digit*);
  void          sub(const Digit*, Digit*, const Digit*, const Digit*);
  bool          sub_no_dec(const Digit*, Digit*, const Digit*) const;
  int           abs(Digit*, const Digit*, const Digit*, size_t) const;

  void          sqr(const Digit*, Digit*, size_t) const;
  Digit         mul(const Digit*, const Digit*, Digit*, const Digit) const;
  Digit*        muladd(const Digit*, const Digit*, Digit*, const Digit) const;
  void          mul(const Digit*, const Digit*, const size_t, Digit*) const;
  void          cmul(const Digit*, size_t, const Digit*, const size_t, Digit*) const;
  void          mul(const Digit*, size_t, const Digit*, const size_t, Digit*) const;
  Digit         mod_div(const Digit);

protected:
  void          dec(Digit*);      // bad! because Integer::setbit
  void          get_memory(const size_t);

  Natural(char, char);
  Natural(const size_t, char);
  Natural(const Digit, size_t);
  Natural(size_t, const Natural&);
  Natural(const Natural&, size_t);

  const Digit*  first() const;
  Digit*        last() const;
  size_t        rootsize() const;
  void          normalize();
  size_t        trailing_zeros(Digit*&) const;
  Digit*        setsize(const size_t);
  Natural&      copy(const Natural&, const size_t);
  void          enlarge(const size_t);
  void          fast_rshift(const size_t);
  void          fast_append(const size_t);

  int           compare(const Natural&) const;
  void          add(const Natural&, const Natural&);
  void          sub(const Natural&, const Natural&);
  void          mul(const Natural&, const Natural&);
  void          sqr(const Natural&);
  void          div(const Natural&, Natural, Natural&);
```

```
  void          sqrt(const Natural&);
  void          newton_sqrt(Natural);

  void          lshift(const Natural&, size_t);
  void          rshift(const Natural&, size_t);

  void          bitwise_and(const Natural&, const Natural&);
  void          bitwise_or(const Natural&, const Natural&);
  void          bitwise_xor(const Natural&, const Natural&);
  void          bitwise_not(const Natural&);

  void          add(const Natural&, const Digit);
  void          sub(const Natural&, const Digit);
  void          mul(const Natural&, const Digit);
  void          muladd(const Natural&, const Digit);
  void          mulsub(const Natural&, const Digit);


public:
  static size_t NumberOfDecimals(const size_t);
  static size_t NumberOfDigits(const size_t);
  static void   RestoreSize();
```

⟨ public constructors and destructor of Natural 25a ⟩

```
  inline Natural& operator=(const Natural&);
  Natural&        operator+=(const Natural&);
  Natural&        operator-=(const Natural&);
  Natural&        operator*=(const Natural&);
  Natural&        operator/=(const Natural&);
  Natural&        operator%=(const Natural&);
  Natural&        operator&=(const Natural&);
  Natural&        operator|=(const Natural&);
  Natural&        operator^=(const Natural&);

  Digit           operator=(const Digit);
  Natural&        operator+=(const Digit);
  Natural&        operator-=(const Digit);
  Natural&        operator*=(const Digit);
  Natural&        operator/=(const Digit);
  Digit           operator%=(const Digit);
  Digit           operator&=(const Digit);
  Natural&        operator|=(const Digit);
  Natural&        operator^=(const Digit);
  Natural&        operator>>=(size_t);
  Natural&        operator<<=(size_t);
  void            lmove(size_t);
  void            rmove(size_t);

  Natural&        operator++();
  Natural&        operator--();
  const Natural   operator++(int);
  const Natural   operator--(int);


  Natural(const binder_arguments<Natural, Natural, Natural_plus_tag>&);
  Natural& operator=(const binder_arguments<Natural, Natural, Natural_plus_tag>&);
```

```
Natural(const binder_arguments<Natural, Natural, Natural_minus_tag>&);
Natural& operator=(const binder_arguments<Natural, Natural, Natural_minus_tag>&);

Natural(const binder_arguments<Natural, Natural, Natural_multiplies_tag>&);
Natural& operator=(const binder_arguments<Natural, Natural, Natural_multiplies_tag>&);

Natural(const binder_arguments<Natural, Natural, Natural_divides_tag>&);
Natural& operator=(const binder_arguments<Natural, Natural, Natural_divides_tag>&);

Natural(const binder_arguments<Natural, Natural, Natural_modulus_tag>&);
Natural& operator=(const binder_arguments<Natural, Natural, Natural_modulus_tag>&);


Natural(const binder_arguments<Natural, size_t, Natural_lshift_tag>&);
Natural& operator=(const binder_arguments<Natural, size_t, Natural_lshift_tag>&);

Natural(const binder_arguments<Natural, size_t, Natural_rshift_tag>&);
Natural& operator=(const binder_arguments<Natural, size_t, Natural_rshift_tag>&);

Natural(const binder_arguments<Natural, Natural, Natural_and_tag>&);
Natural& operator=(const binder_arguments<Natural, Natural, Natural_and_tag>&);

Natural(const binder_arguments<Natural, Natural, Natural_or_tag>&);
Natural& operator=(const binder_arguments<Natural, Natural, Natural_or_tag>&);

Natural(const binder_arguments<Natural, Natural, Natural_xor_tag>&);
Natural& operator=(const binder_arguments<Natural, Natural, Natural_xor_tag>&);

Natural(const binder_arguments<Natural, Natural, Natural_not_tag>&);
Natural& operator=(const binder_arguments<Natural, Natural, Natural_not_tag>&);


Natural(const binder_arguments<Natural, Digit, Natural_minus_tag>&);
Natural& operator=(const binder_arguments<Natural, Digit, Natural_minus_tag>&);

Natural(const binder_arguments<Natural, Digit, Natural_plus_tag>&);
Natural& operator=(const binder_arguments<Natural, Digit, Natural_plus_tag>&);

Natural(const binder_arguments<Natural, Digit, Natural_multiplies_tag>&);
Natural& operator=(const binder_arguments<Natural, Digit, Natural_multiplies_tag>&);
Natural& operator+=(const binder_arguments<Natural, Digit, Natural_multiplies_tag>&);
Natural& operator-=(const binder_arguments<Natural, Digit, Natural_multiplies_tag>&);


friend Digit        operator%(const Natural&, const Digit);

inline friend bool  operator==(const Natural&, const Natural&);
inline friend bool  operator!=(const Natural&, const Natural&);
inline friend bool  operator<(const Natural&, const Natural&);
inline friend bool  operator<=(const Natural&, const Natural&);
inline friend bool  operator>(const Natural&, const Natural&);
inline friend bool  operator>=(const Natural&, const Natural&);

inline friend bool  operator==(const Natural&, const Digit);
inline friend bool  operator!=(const Natural&, const Digit);
```

```
  inline friend bool  operator<(const Natural&, const Digit);
  inline friend bool  operator<=(const Natural&, const Digit);
  inline friend bool  operator>(const Natural&, const Digit);
  inline friend bool  operator>=(const Natural&, const Digit);


  Natural(const binder_arguments<Natural, Natural, Natural_square_root_tag>&);
  Natural& operator=(const binder_arguments<Natural, Natural, Natural_square_root_tag>&);


  friend void         div(const Natural&, const Natural&, Natural&, Natural&);
  friend void         div(const Natural&, const Digit, Natural&, Digit&);
  friend void         sqrt(const Natural&, Natural&, Natural&);
  friend void         swap(Natural&, Natural&);
  inline friend bool  sign(const Natural&);


  void    split(const size_t, Natural&, Natural&) const;
  Digit   highest() const;
  Digit   lowest() const;
  size_t  length() const;
  void    setbit(const Digit);
  void    clearbit(const Digit);
  bool    testbit(const Digit) const;
  bool    odd() const;
  bool    even() const;
  void    rand(size_t);
  bool    scan(ISTREAM&);

  const char* atoN(const char*, const Digit = 10);

  inline friend rep print(const Natural&, bool = false);

  friend Natural    abs(const Natural&, const Natural&);
  friend int        abs(const Natural&, const Natural&, Natural&);
  friend Natural    gcd(Natural, Natural);

  friend class   FFT;
};

inline binder_arguments<Natural, Natural, Natural_plus_tag>
 operator+(const Natural&, const Natural&);
inline binder_arguments<Natural, Natural, Natural_minus_tag>
 operator-(const Natural&, const Natural&);
inline binder_arguments<Natural, Natural, Natural_multiplies_tag>
 operator*(const Natural&, const Natural&);
inline binder_arguments<Natural, Natural, Natural_divides_tag>
 operator/(const Natural&, const Natural&);
inline binder_arguments<Natural, Natural, Natural_modulus_tag>
 operator%(const Natural&, const Natural&);
inline binder_arguments<Natural, size_t, Natural_lshift_tag>
 operator<<(const Natural&, const size_t&);
inline binder_arguments<Natural, size_t, Natural_rshift_tag>
 operator>>(const Natural&, const size_t&);
inline binder_arguments<Natural, Natural, Natural_and_tag>
 operator&(const Natural&, const Natural&);
```

```
inline binder_arguments<Natural, Natural, Natural_or_tag>
 operator|(const Natural&, const Natural&);
inline binder_arguments<Natural, Natural, Natural_xor_tag>
 operator^(const Natural&, const Natural&);
inline binder_arguments<Natural, Natural, Natural_not_tag>
 operator~(const Natural&);
inline binder_arguments<Natural, Digit, Natural_minus_tag>
 operator-(const Natural&, const Digit&);
inline binder_arguments<Natural, Digit, Natural_plus_tag>
 operator+(const Natural&, const Digit&);
inline binder_arguments<Natural, Digit, Natural_plus_tag>
 operator+(const Digit&, const Natural&);
inline binder_arguments<Natural, Digit, Natural_multiplies_tag>
 operator*(const Natural&, const Digit&);
inline binder_arguments<Natural, Digit, Natural_multiplies_tag>
 operator*(const Digit&, const Natural&);
inline Natural  operator/(const Natural&, const Digit);
inline Digit    operator&(const Natural&, const Digit);
inline Natural  operator|(const Natural&, const Digit);
inline binder_arguments<Natural, Natural, Natural_square_root_tag>
 sqrt(const Natural&);

char*           Ntoa(const Natural&, char*, const Digit = 10);
inline Natural  atoN(const char*, const Digit = 10);
OSTREAM&        operator<<(OSTREAM&, const Natural&);
OSTREAM&        operator<<(OSTREAM&, const Natural::rep&);
ISTREAM&        operator>>(ISTREAM&, Natural&);

inline Digit  log2(const Natural&);
Natural       lcm(const Natural&, const Natural&);
Natural       pow(const Natural&, Natural);
Natural       pow(const Natural&, Digit);
Natural       root(const Natural&, const Digit);

void          div(const Natural&, const Natural&, Natural&, Natural&);
void          div(const Natural&, const Digit, Natural&, Digit&);
void          sqrt(const Natural&, Natural&, Natural&);
void          swap(Natural&, Natural&);
Natural       abs(const Natural&, const Natural&);
int           abs(const Natural&, const Natural&, Natural&);
Natural       gcd(Natural, Natural);


////////////////////////// Inline-Implementation ////////////////////

inline Natural::Natural(char, char)
{
}
```

⟨ comparison `operator==` for Naturals 47b ⟩
⟨ comparison `operator!=` for Naturals 48a ⟩
⟨ comparison `operator<` for Naturals 48b ⟩
⟨ comparison `operator<=` for Naturals 48c ⟩
⟨ comparison `operator>` for Naturals 48d ⟩
⟨ comparison `operator>=` for Naturals 49a ⟩
⟨ comparison `operator==` of a Natural with a Digit 49b ⟩

⟨ comparison operator!= of a Natural with a Digit 50a ⟩
⟨ comparison operator< of a Natural with a Digit 50b ⟩
⟨ comparison operator<= of a Natural with a Digit 50c ⟩
⟨ comparison operator> of a Natural with a Digit 50d ⟩
⟨ comparison operator>= of a Natural with a Digit 51a ⟩
⟨ protected constructor Natural without the initialization of the elements 27b ⟩
```
inline bool sign(const Natural& a)
// Algorithm:  c := sign(a)
// Input:      a in Natural.
// Output:     c in bool such that if a = 0 then c = false else c = true ||
{
  return (*a.p != 0);
}
```
⟨ memberfunction Natural.highest 18b ⟩
⟨ memberfunction Natural.lowest 18c ⟩
⟨ memberfunction Natural.length 18a ⟩
⟨ memberfunction Natural.odd 86a ⟩
⟨ memberfunction Natural.even 86b ⟩
```
inline const Digit* Natural::first() const
// Algorithm:  c := a.first()
// Input:      a in Natural.
// Output:     c in [a.p, a.p+L(a)[ such that c = a.p ||
{
  return p;
}

inline Digit* Natural::last() const
// Algorithm:  c := a.last()
// Input:      a in Natural.
// Output:     c in [a.p, a.p+L(a)[ such that c = a.p+L(a)-1 ||
{
  return p+size-1;
}
```
⟨ memberfunction Natural.rootsize 17 ⟩
⟨ memberfunction Natural.normalize 19a ⟩
⟨ fast division of a Natural by a power of 2^β 75a ⟩
⟨ fast multiplication of a Natural by a power of 2^β 75b ⟩
⟨ memberfunction Natural.sub with 3 arguments 63a ⟩
⟨ memberfunction Natural.mod_div 158 ⟩
⟨ memberfunction Natural.NumberOfDecimals 21b ⟩
⟨ memberfunction Natural.NumberOfDigits 21c ⟩
⟨ memberfunction Natural.RestoreSize 21d ⟩
⟨ assign operator*= for Naturals 105 ⟩
⟨ assign operator/= for Naturals 161 ⟩
⟨ assign operator%= for Naturals 162a ⟩
⟨ assign operator/= of a Natural with a Digit 162b ⟩
⟨ assign operator%= of a Natural with a Digit 162c ⟩
⟨ shift operator>> of a Natural 73 ⟩
⟨ shift operator<< of a Natural 70a ⟩
⟨ assign operator= for Naturals 32c ⟩
⟨ assign operator|= of a Natural with a Digit 80a ⟩
⟨ assign operator&= of a Natural with a Digit 78a ⟩
⟨ assign operator^= of a Natural with a Digit 83a ⟩
⟨ prefix incrementation of a Natural 52b ⟩
⟨ prefix decrementation of a Natural 53a ⟩
⟨ postfix incrementation of a Natural 53b ⟩

⟨ postfix decrementation of a Natural 53c ⟩
⟨ bitwise operator~ of a Natural 84a ⟩
⟨ additive operator+ for Naturals 34b ⟩
⟨ additive operator- for Naturals 60b ⟩
⟨ multiplicative operator* for Naturals 104a ⟩
⟨ multiplicative operator/ for Naturals 159a ⟩
⟨ multiplicative operator% for Naturals 159b ⟩
⟨ bitwise operator& for Naturals 76 ⟩
⟨ bitwise operator| for Naturals 79a ⟩
⟨ bitwise operator^ for Naturals 81 ⟩
⟨ additive operator- of a Natural with a Digit 62b ⟩
⟨ additive operator+ of a Natural with a Digit 57 ⟩
⟨ multiplicative operator* of a Natural with a Digit 104b ⟩
⟨ multiplicative operator/ of a Natural with a Digit 160a ⟩
⟨ bitwise operator& of a Natural with a Digit 77a ⟩
⟨ bitwise operator| of a Natural with a Digit 79b ⟩
⟨ binary logarithm for Naturals 148a ⟩
⟨ converts a string to a Natural by function call 46 ⟩
⟨ function print of a Natural 38b ⟩
⟨ simple call for square root of a Natural 180 ⟩
#endif
◇

"natural.cpp" 409 ≡

```
////////////////////////////////
//
// Piologie V 1.3.2
// multi-precision arithmetic
// Natural
//
// (c) 1996-2001 HiPiLib
//               www.hipilib.de
//
// Sebastian Wedeniwski
// 10/21/2001
//

#include "natural.h"
#include <string.h>


⟨ trade off points for the Karatsuba algorithm 103b ⟩
⟨ trade off points for the FFT algorithm 112 ⟩
⟨ trade off points for the newton-iteration of the division algorithm 165a ⟩
⟨ trade off points for the newton-iteration of the square root algorithm 179a ⟩
⟨ trade off point for the Natural to string conversion 42a ⟩
⟨ trade off point for the string to Natural conversion 45a ⟩

//#define FASTER_BINSQRT      // binary square root without shifting
                              // need BETA/2 times more memory! (Bug!)


⟨ macro definitions for internal conditions 6b ⟩
⟨ macro definitions for internal memory manipulations 9 ⟩
⟨ new operator 396e ⟩
```

⟨ class FFT 116 ⟩

⟨ memberfunction FFT.size 120 ⟩
⟨ memberfunction FFT.setmodulo 127a ⟩
```
#if defined(_DigitAsm_) && _M_IX86 >= 300 && defined(_MSC_VER)
static Digit __declspec(naked) __fastcall _digitmulmod(const Digit a, const Digit b,
                                                       const Digit m)
{
  __asm {
    mov eax,ecx
    mul edx
    div dword ptr [esp+4]
    mov eax,edx
    ret 4
  }
}

inline void FFT::digitmulmod(const Digit a, const Digit b, const Digit m, Digit& c) const
// Algorithm:  f.digitmulmod(a, b, m, c)
// Input:      f in FFT, a,b,m in Digit where m > 0, a,b <= m.
// Output:     c in Digit such that c = a*b - f.m*[(a*b)/f.m] ||
{
  c = _digitmulmod(a, b, m);
}
#elif defined(_DigitAsm_) && (defined (__i386__) || defined (__i486__)) && defined (__GNUC__)
inline void FFT::digitmulmod(const Digit a, const Digit b, const Digit m, Digit& c) const
{
  int dummy;
  __asm__ ("mull %3; divl %4"
    : "=&d" (c), "=a" (dummy)
    : "%a" (a), "rm" (b), "rm" (m));
}
#else
```
⟨ memberfunction FFT.digitmulmod 127b ⟩
```
#endif


////////////////// Natural arithmetic ////////////////////////
```
⟨ definition of static variables for memory management of Naturals 21a ⟩
⟨ memberfunction Natural.get_memory 27c ⟩
⟨ protected constructors Natural for the optimal allocation 28a, ... ⟩
⟨ memberfunction Natural.trailing_zeros 19b ⟩
⟨ memberfunction Natural.setsize 23 ⟩
⟨ memberfunction Natural.copy 30b ⟩
⟨ memberfunction Natural.enlarge 22b ⟩
⟨ memberfunction Natural.inc 51b ⟩
⟨ memberfunction Natural.dec 52a ⟩
⟨ memberfunction Natural.add_with_inc with 3 arguments 58a ⟩
⟨ memberfunction Natural.add_with_inc with 4 arguments 58b ⟩
⟨ memberfunction Natural.add_no_inc with 3 arguments 11 ⟩
⟨ memberfunction Natural.add_no_inc with 4 arguments 12 ⟩
⟨ memberfunction Natural.sub with 4 arguments 64a ⟩
⟨ memberfunction Natural.sub_no_dec with 3 arguments 63b ⟩
⟨ internal function subpos with 3 arguments 64b ⟩
⟨ internal function subpos with 4 arguments 65 ⟩

⟨ memberfunction Natural.abs 67b ⟩

```
#if defined(_DigitAsm_) && _M_IX86 >= 300 && defined(_MSC_VER)
Digit Natural::mul(const Digit* pE, const Digit* pA, Digit* pT,
                   const Digit c) const
{
  Digit r;
  __asm {
    xor ebx,ebx
    mov edi,pA
    mov esi,pT
    mov ecx,pE
L1: sub edi,4
    sub esi,4
    mov eax,[edi]
    mul c
    add eax,ebx
    adc edx,0
    mov [esi],eax
    mov ebx,edx
    cmp edi,ecx
    jne L1
    mov [esi-4],ebx
    mov r,ebx
  }
  return r;
}

Digit* Natural::muladd(const Digit* pE, const Digit* pA, Digit* pT,
                       const Digit c) const
{
  Digit* r;
  __asm {
    xor ebx,ebx
    mov edi,pA
    mov esi,pT
    mov ecx,pE
L1: sub edi,4
    sub esi,4
    mov eax,[edi]
    mul c
    add eax,ebx
    adc edx,0
    add eax,[esi]
    adc edx,0
    mov [esi],eax
    mov ebx,edx
    cmp edi,ecx
    jne L1
    sub esi,4
    mov eax,[esi]
    add eax,ebx
    mov [esi],eax
    jnc L3
L2: sub esi,4
    mov eax,[esi]
    inc eax
```

```
    mov [esi],eax
    jz  L2
L3: mov r,esi
  }
  return r;
}
#else
```
⟨ memberfunction Natural.mul 93a ⟩
⟨ protected memberfunction Natural.muladd 94 ⟩
```
#endif
```
⟨ conventional squaring algorithm 106 ⟩
⟨ conventional multiplication algorithm 91a ⟩
⟨ multiplication with equal size 99 ⟩
⟨ multiplication with different size 103a ⟩
⟨ default constructor Natural 25b ⟩
⟨ copy constructor Natural 26a ⟩
⟨ overloaded constructor Natural for ASCII-string conversion 26b ⟩
⟨ destructor Natural 27a ⟩
⟨ assign operator+= for Naturals 54b ⟩
⟨ assign operator-= for Naturals 61a ⟩
⟨ assign operator&= for Naturals 77b ⟩
⟨ assign operator|= for Naturals 79c ⟩
⟨ assign operator^= for Naturals 82 ⟩
⟨ compares two Naturals 47a ⟩
⟨ function swap for Naturals 30a ⟩
⟨ splits a Natural 165b ⟩
⟨ addition of two Naturals 54a ⟩
⟨ subtraction of two Naturals 59 ⟩
⟨ multiplication of two Naturals 97 ⟩
⟨ squaring of a Natural 108 ⟩
⟨ division of two Naturals with remainder 153 ⟩
⟨ division of two Naturals 163b ⟩
⟨ bitwise and of two Naturals 75c ⟩
⟨ bitwise inclusive or of two Naturals 78b ⟩
⟨ bitwise exclusive or of two Naturals 80b ⟩
⟨ bitwise not of a Natural 83b ⟩
⟨ multiplication of a Natural by a power of 2 68b ⟩
⟨ multiplication of a Natural by a power of 2^β 69 ⟩
⟨ division of a Natural by a power of 2 72a ⟩
⟨ division of a Natural by a power of 2^β 72b ⟩
⟨ addition of a Natural with a Digit 56a ⟩
⟨ subtraction of a Natural with a Digit 61b ⟩
⟨ multiplication of a Natural by a Digit 91b ⟩
⟨ division of a Natural by a Digit 157 ⟩
⟨ assign operator+= of a Natural with a Digit 56b ⟩
⟨ assign operator-= of a Natural with a Digit 62a ⟩
⟨ assign operator*= of a Natural with a Digit 92 ⟩
⟨ assign operator>>= of a Natural 74 ⟩
⟨ assign operator<<= of a Natural 70b ⟩
⟨ assign operator= of a Natural with a Digit 33 ⟩
⟨ multiplicative operator% of a Natural with a Digit 160b ⟩
⟨ multiplication and addition of a Natural by a Digit 93b ⟩
⟨ multiplication and subtraction of a Natural by a Digit 156 ⟩
⟨ internal function sqrtsub 173a ⟩
⟨ square root of a Natural 176 ⟩
⟨ square root of a Natural with newton iteration 179b ⟩

⟨ square root and remainder of a Natural 174c ⟩
⟨ calculates the root of a Natural 181 ⟩
⟨ sets a bit in a Natural 84b ⟩
⟨ clears a bit in a Natural 85a ⟩
⟨ tests a bit in a Natural 85b ⟩
⟨ calculates a Natural random number 182 ⟩
⟨ converts a string to a Natural 44 ⟩
⟨ converts a Natural to a string 41, ... ⟩
⟨ puts a Natural on output stream 36 ⟩
⟨ gets a Natural from input stream 40 ⟩
⟨ puts internal representation of a Natural on output stream 37 ⟩
⟨ gets internal representation of a Natural from input stream 38c ⟩
⟨ distance of two Naturals 66, ... ⟩
⟨ greatest common divisor of two Naturals 270b ⟩
⟨ least common multiple of two Naturals 275b ⟩
⟨ calculates the power of a Natural by a Digit 148b ⟩
⟨ calculates the power of a Natural by a Natural 149 ⟩

```
////////////////////////// FFT class ////////////////////////////

const Digit*  FFT::moduli     = FFT::init_moduli();
const Digit*  FFT::primroots  = FFT::init_primroots();
Digit         FFT::m          = 0;
size_t        FFT::n1         = 0;
size_t        FFT::n2         = 0;
Digit*        FFT::omega      = 0;
Digit*        FFT::omega2     = 0;
size_t*       FFT::order      = 0;
```

⟨ memberfunction FFT.init_moduli 125 ⟩
⟨ memberfunction FFT.init_primroots 126a ⟩
⟨ memberfunction FFT.max_size 126b ⟩
⟨ memberfunction FFT.pow 131c ⟩
⟨ memberfunction FFT.digitinv 133 ⟩
⟨ memberfunction FFT.init_omega 132 ⟩
⟨ transpose and copy a matrix 136 ⟩
⟨ brevorder an array 140a ⟩
⟨ inner loop of fft without multiplication 140b ⟩
⟨ inner loop of fft 140c ⟩
⟨ inner loop of inverse fft 141a ⟩
⟨ memberfunction FFT.innerfft3 142 ⟩
⟨ memberfunction FFT.innerfftinv3 143 ⟩

```
#if defined(_DigitAsm_) && _M_IX86 >= 300 && defined(_MSC_VER)
void FFT::fft(Digit* a, const Digit* b, const size_t c) const
{
  Digit o;
  Digit* s;
  Digit* pE;
  Digit M = m;
  __asm {
    mov ebx,c   // j
    mov eax,1   // i
    mov ecx,ebx
    shr ebx,1
    cmp ecx,1
```

```
        je  L10
    L1: mov edi,a
        mov ecx,c
        mov edx,M
        lea ecx,[edi+ecx*4]
        push eax
        push ebp
        shl ebx,1
        mov ebp,ecx

        // innerfft(a+c, a, a+j, 2*j, m);
    L2: mov eax,[edi]         // a
        mov esi,[edi+ebx*2]   // b
        mov ecx,edx
        sub ecx,eax
        cmp ecx,esi
        lea ecx,[eax+esi]
        ja  L3
        sub ecx,edx
    L3: sub eax,esi
        jae L4
        add eax,edx
    L4: mov [edi],ecx
        mov [edi+ebx*2],eax
        lea edi,[edi+ebx*4]
        cmp edi,ebp
        jb  L2

        shr ebx,1
        pop ebp
        pop eax
        mov edx,b
        lea edx,[edx+eax*4]
        mov s,edx
        mov ecx,1   // k
        cmp ebx,1
        jbe L9
    L5: mov esi,a
        mov edx,c
        lea edi,[esi+ecx*4]
        lea edx,[esi+edx*4]
        mov pE,edx
        push eax
        push ecx
        mov edx,s
        mov ecx,[edx]
        mov o,ecx
        shl ebx,1

        // innerfft(a+c, a+k, a+k+j, 2*j, m, *s);
    L6: mov eax,[edi]
        mov esi,[edi+ebx*2]
        mov edx,M
        sub edx,eax
        cmp edx,esi
        mov edx,M
```

```
        lea ecx,[eax+esi]
        ja  L7
        sub ecx,edx
    L7: sub eax,esi
        jae L8
        add eax,edx
    L8: mul o
        div M
        mov [edi],ecx
        mov [edi+ebx*2],edx
        mov ecx,pE
        lea edi,[edi+ebx*4]
        cmp edi,ecx
        jb  L6

        shr ebx,1
        pop ecx
        pop eax
        mov edx,s
        lea edx,[edx+eax*4]
        mov s,edx
        inc ecx
        cmp ecx,ebx
        jne L5
    L9: shr ebx,1
        shl eax,1
        mov edx,c
        cmp eax,edx
        jne L1
    L10:
      }
    }

    void FFT::fftinv(Digit* a, const Digit* b, const size_t c) const
    {
      Digit o;
      Digit* s;
      Digit* pE;
      Digit M = m;
      __asm {
        mov ebx,c   // j
        mov eax,1   // i
        mov ecx,ebx
        shr ebx,1
        cmp ecx,1
        je  L10
    L1: mov edi,a
        mov ecx,c
        mov edx,M
        lea ecx,[edi+ecx*4]
        push ebx
        push ebp
        mov ebp,ecx
        shl eax,1

        // innerfft(a+c, a, a+i, 2*i, m);
```

```
L2: mov esi,[edi]
    mov ebx,[edi+2*eax]
    mov ecx,edx
    sub ecx,ebx
    cmp ecx,esi
    lea ecx,[ebx+esi]
    ja  L3
    sub ecx,edx
L3: sub esi,ebx
    jae L4
    add esi,edx
L4: mov [edi],ecx
    mov [edi+2*eax],esi
    lea edi,[edi+eax*4]
    cmp edi,ebp
    jb  L2

    shr eax,1
    pop ebp
    pop ebx
    mov edx,b
    lea edx,[edx+ebx*4]
    mov s,edx
    mov ecx,1   // k
    cmp eax,1
    jbe L9
L5: mov esi,a
    mov edx,c
    lea edi,[esi+ecx*4]
    lea edx,[esi+edx*4]
    mov pE,edx
    push ebx
    push ecx
    mov edx,s
    mov ecx,[edx]
    mov o,ecx
    lea esi,[eax*2]

    // innerfftinv(a+c, a+k, a+k+i, 2*i, m, *s);
    mov ecx,M
L6: mov eax,[edi+esi*2]
    mov ebx,[edi]
    mul o
    div ecx
    mov eax,ecx
    sub eax,ebx
    cmp eax,edx
    lea eax,[ebx+edx]
    ja  L7
    sub eax,ecx
L7: sub ebx,edx
    jae L8
    add ebx,ecx
L8: mov [edi],eax
    mov [edi+esi*2],ebx
    mov eax,pE
```

```
    lea edi,[edi+esi*4]
    cmp edi,eax
    jb  L6

    pop ecx
    pop ebx
    mov eax,esi
    shr eax,1
    mov edx,s
    lea edx,[edx+ebx*4]
    mov s,edx
    inc ecx
    cmp ecx,eax
    jne L5
L9: shr ebx,1
    shl eax,1
    mov edx,c
    cmp eax,edx
    jne L1
L10:
  }
}

void FFT::multiply_matrix(Digit* pA, const Digit w, const Digit iw) const
{
  size_t n1_ = n1;
  size_t n2_ = n2;
  Digit M = m;
  Digit u;
  __asm {
    mov u,1
    mov esi,pA
    mov eax,n1_
    mov ecx,n2_
    imul eax,ecx
    lea edi,[esi+eax*4]
L1: mov edx,n2_
    lea ecx,[esi+edx*4]
    mov ebx,iw
L2: mov eax,[esi]
    mul ebx
    div M
    mov [esi],edx
    mov eax,ebx
    mul u
    div M
    mov ebx,edx
    add esi,4
    cmp esi,ecx
    jne L2
    mov eax,u
    mul w
    div M
    mov u,edx
    cmp esi,edi
    jne L1
```

```
  }
}
#else
⟨ memberfunction FFT.fft 141b ⟩
⟨ memberfunction FFT.fftinv 141c ⟩
⟨ multiply matrix components by a power of a primitive root of unity 138 ⟩
#endif
⟨ five step FFT algorithm 135a ⟩
⟨ five step FFT algorithm for inversion 135b ⟩
⟨ chinese remainder theorem for FFT-class 122 ⟩
#if defined(_DigitAsm_) && _M_IX86 >= 300 && defined(_MSC_VER)
void FFT::square(const size_t a)
{
  Digit* pT = t[a].p;
  Digit* pE = pT+t[a].size;
  Digit M = m;
  __asm {
    mov edi,pT
    mov ecx,pE
    mov ebx,M
    sub ecx,edi
    mov eax,ecx
    shr ecx,3
    and eax,4
    je  L1
    mov eax,[edi]
    mul eax
    div ebx
    mov [edi],edx
    add edi,4
L1: mov eax,[edi]
    mul eax
    div ebx
    mov [edi],edx
    mov eax,4[edi]
    mul eax
    div ebx
    mov 4[edi],edx
    add edi,8
    dec ecx
    jne L1
  }
}

void FFT::multiply(const size_t a)
{
  Digit* pT = t[a].p;
  Digit* pE = pT+t[a].size;
  Digit* pA = factor->t[0].p;
  Digit M = m;
  __asm {
    mov edi,pT
    mov esi,pA
    mov ecx,pE
    mov ebx,M
    sub ecx,edi
```

```
        mov eax,ecx
        shr ecx,3
        and eax,4
        je  L1
        mov edx,[esi]
        mov eax,[edi]
        mul edx
        div ebx
        mov [edi],edx
        add esi,4
        add edi,4
    L1: mov edx,[esi]
        mov eax,[edi]
        mul edx
        div ebx
        mov [edi],edx
        mov eax,4[esi]
        mov edx,4[edi]
        mul edx
        div ebx
        mov 4[edi],edx
        add esi,8
        add edi,8
        dec ecx
        jne L1
      }
    }
    #else
    ⟨ componentwise squaring 145a ⟩
    ⟨ componentwise multiplication 147a ⟩
    #endif
    ⟨ memberfunction FFT.result 119 ⟩
    ⟨ squaring of a Natural with FFT 144 ⟩
    ⟨ multiplication of two Naturals with FFT 145b ⟩
    ⟨ brevorder initialization 139 ⟩
    ⟨ FFT base conversion 114 ⟩
    ⟨ default constructor FFT 115 ⟩
    ⟨ destructor FFT 117 ⟩
    ◇
```

D.4 Integer

"integer.h" 419 ≡

```
    ////////////////////////////////
    //
    // Piologie V 1.3.2
    // multi-precision arithmetic
    // Integer
    //
    // (c) 1996-2001 HiPiLib
    //               www.hipilib.de
    //
    // Sebastian Wedeniwski
    // 10/21/2001
    //
```

```
#ifndef _Include_Integer_H_
#define _Include_Integer_H_

#include "natural.h"

struct Integer_plus_tag {};
struct Integer_negate_tag {};
struct Integer_minus_tag {};
struct Integer_multiplies_tag {};
struct Integer_divides_tag {};
struct Integer_modulus_tag {};
struct Integer_square_root_tag {};
struct Integer_lshift_tag {};
struct Integer_rshift_tag {};
struct Integer_and_tag {};
struct Integer_or_tag {};
struct Integer_xor_tag {};
struct Integer_not_tag {};
```

⟨ class Integer 186b ⟩

```
inline binder_arguments<Integer, Integer, Integer_negate_tag>
 operator-(const Integer&);
inline binder_arguments<Integer, Integer, Integer_plus_tag>
 operator+(const Integer&, const Integer&);
inline binder_arguments<Integer, Integer, Integer_minus_tag>
 operator-(const Integer&, const Integer&);
inline binder_arguments<Integer, Integer, Integer_multiplies_tag>
 operator*(const Integer&, const Integer&);
inline binder_arguments<Integer, Integer, Integer_divides_tag>
 operator/(const Integer&, const Integer&);
inline binder_arguments<Integer, Integer, Integer_modulus_tag>
 operator%(const Integer&, const Integer&);
inline binder_arguments<Integer, size_t, Integer_lshift_tag>
 operator<<(const Integer&, const size_t&);
inline binder_arguments<Integer, size_t, Integer_rshift_tag>
 operator>>(const Integer&, const size_t&);
inline binder_arguments<Integer, Integer, Integer_and_tag>
 operator&(const Integer&, const Integer&);
inline binder_arguments<Integer, Integer, Integer_or_tag>
 operator|(const Integer&, const Integer&);
inline binder_arguments<Integer, Integer, Integer_xor_tag>
 operator^(const Integer&, const Integer&);
inline binder_arguments<Integer, Integer, Integer_not_tag>
 operator~(const Integer&);
inline binder_arguments<Integer, Natural, Integer_plus_tag>
 operator+(const Natural&, const Integer&);
inline binder_arguments<Integer, Natural, Integer_plus_tag>
 operator+(const Integer&, const Natural&);
inline binder_arguments<Natural, Integer, Integer_minus_tag>
 operator-(const Natural&, const Integer&);
inline binder_arguments<Integer, Natural, Integer_minus_tag>
 operator-(const Integer&, const Natural&);
inline binder_arguments<Integer, Natural, Integer_multiplies_tag>
 operator*(const Natural&, const Integer&);
```

```
inline binder_arguments<Integer, Natural, Integer_multiplies_tag>
 operator*(const Integer&, const Natural&);
inline binder_arguments<Natural, Integer, Integer_divides_tag>
 operator/(const Natural&, const Integer&);
inline binder_arguments<Integer, Natural, Integer_divides_tag>
 operator/(const Integer&, const Natural&);
inline binder_arguments<Integer, SignDigit, Integer_plus_tag>
 operator+(const SignDigit&, const Integer&);
inline binder_arguments<Integer, SignDigit, Integer_plus_tag>
 operator+(const Integer&, const SignDigit&);
inline binder_arguments<Integer, SignDigit, Integer_multiplies_tag>
 operator*(const SignDigit&, const Integer&);
inline binder_arguments<Integer, SignDigit, Integer_multiplies_tag>
 operator*(const Integer&, const SignDigit&);

inline Integer    operator/(const Integer&, const SignDigit);
inline SignDigit  operator%(const Integer&, const SignDigit);
inline Digit      operator&(const Integer&, const Digit);
inline Integer    operator|(const Integer&, const Digit);


inline bool       operator==(const Integer&, const Integer&);
inline bool       operator!=(const Integer&, const Integer&);
inline bool       operator<(const Integer&, const Integer&);
inline bool       operator<=(const Integer&, const Integer&);
inline bool       operator>(const Integer&, const Integer&);
inline bool       operator>=(const Integer&, const Integer&);

inline bool       operator==(const Integer&, const SignDigit);
inline bool       operator!=(const Integer&, const SignDigit);
inline bool       operator<(const Integer&, const SignDigit);
inline bool       operator<=(const Integer&, const SignDigit);
inline bool       operator>(const Integer&, const SignDigit);
inline bool       operator>=(const Integer&, const SignDigit);

inline binder_arguments<Integer, Integer, Integer_square_root_tag>
 sqrt(const Integer&);


inline const Natural& abs(const Integer&);
inline Digit          log2(const Integer&);
Integer               root(const Integer&, const SignDigit);
inline int            units(Integer&);
void                  gcd(Integer, Integer, Integer&, Integer&, Integer&);

inline Integer        atoI(const char*, const Digit = 10);
char*                 Itoa(const Integer&, char*, const Digit = 10);

inline Integer::rep   print(const Integer&, bool = false);
inline OSTREAM&       operator<<(OSTREAM&, const Integer::rep&);
inline OSTREAM&       operator<<(OSTREAM&, const Integer&);


////////////////////////// Inline-Implementation ////////////////////
```

⟨ absolute value of an Integer 189a ⟩

⟨ sign of an Integer 189b ⟩
⟨ negation of an Integer 199b, ... ⟩
⟨ protected constructor Integer without the initialization of the elements 187b ⟩
⟨ default constructor Integer 187c ⟩
⟨ overloaded constructor Integer for Natural 188a ⟩
⟨ copy constructor Integer 188b ⟩
⟨ overloaded constructor Integer for ASCII-string conversion 188c ⟩
⟨ destructor Integer 188d ⟩
⟨ comparison operator== of an Integer with a SignDigit 196b ⟩
⟨ comparison operator!= of an Integer with a SignDigit 196c ⟩
⟨ comparison operator< of an Integer with a SignDigit 196d ⟩
⟨ comparison operator<= of an Integer with a SignDigit 197a ⟩
⟨ comparison operator> of an Integer with a SignDigit 197b ⟩
⟨ comparison operator>= of an Integer with a SignDigit 197c ⟩
⟨ assign operator+= for Integers 204a ⟩
⟨ assign operator-= for Integers 209a ⟩
⟨ assign operator*= for Integers 226 ⟩
⟨ assign operator/= for Integers 234c ⟩
⟨ assign operator%= for Integers 235a ⟩
⟨ assign operator&= for Integers 215c ⟩
⟨ assign operator|= for Integers 219a ⟩
⟨ assign operator^= for Integers 220b ⟩
⟨ assign operator+= of an Integer with a Natural 204b ⟩
⟨ assign operator-= of an Integer with a Natural 209b ⟩
⟨ assign operator*= of an Integer with a Natural 227a ⟩
⟨ assign operator/= of an Integer with a Natural 235b ⟩
⟨ assign operator%= of an Integer with a Natural 235c ⟩
⟨ splits an Integer 236c ⟩
⟨ comparison operator== for Integers 194b ⟩
⟨ comparison operator!= for Integers 195a ⟩
⟨ comparison operator< for Integers 195b ⟩
⟨ comparison operator<= for Integers 195c ⟩
⟨ comparison operator> for Integers 195d ⟩
⟨ comparison operator>= for Integers 196a ⟩
⟨ negation operator- of an Integer 199d ⟩
⟨ additive operator+ for Integers 201b ⟩
⟨ additive operator- for Integers 207a ⟩
⟨ additive operator+ of an Integer with a Natural 202 ⟩
⟨ additive operator- of an Integer with a Natural 207b ⟩
⟨ multiplication of an Integer by a SignDigit 223b ⟩
⟨ multiplication of an Integer by a Natural 223c ⟩
⟨ multiplication of two Integers 224a ⟩
⟨ multiplicative operator* for Integers 224b ⟩
⟨ multiplicative operator* of an Integer with a Natural 224c ⟩
⟨ bitwise not of an Integer 221a ⟩
⟨ multiplication of an Integer by a power of 2 210b ⟩
⟨ division of an Integer by a power of 2 212a ⟩
⟨ assign operator*= of an Integer with a SignDigit 227b ⟩
⟨ assign operator/= of an Integer with a SignDigit 236a ⟩
⟨ assign operator%= of an Integer with a SignDigit 236b ⟩
⟨ assign operator>>= of an Integer 212c ⟩
⟨ assign operator<<= of an Integer 211 ⟩
⟨ shift operator>> of an Integer 212b ⟩
⟨ shift operator<< of an Integer 210c ⟩
⟨ assign operator= for Integers 190b ⟩
⟨ assign operator= for an Integer with a Natural 190c ⟩

⟨ assign operator= of an Integer with a SignDigit 191a ⟩
⟨ assign operator|= of an Integer with a Digit 219b ⟩
⟨ postfix incrementation of an Integer 198a ⟩
⟨ postfix decrementation of an Integer 199a ⟩
⟨ multiplicative operator/ for Integers 230b ⟩
⟨ multiplicative operator% for Integers 231a ⟩
⟨ multiplicative operator/ of an Integer with a Natural 232b ⟩
⟨ bitwise operator~ of an Integer 221b ⟩
⟨ bitwise operator& for Integers 215a ⟩
⟨ bitwise operator| for Integers 218b ⟩
⟨ bitwise operator^ for Integers 220a ⟩
⟨ additive operator+ of an Integer with a SignDigit 203 ⟩
⟨ additive operator- of an Integer with a SignDigit 208 ⟩
⟨ multiplicative operator* of an Integer with a SignDigit 225 ⟩
⟨ multiplicative operator/ of an Integer with a SignDigit 234a ⟩
⟨ multiplicative operator% of an Integer with a SignDigit 234b ⟩
⟨ bitwise operator& of an Integer with a Digit 215b ⟩
⟨ bitwise operator| of an Integer with a Digit 218c ⟩
⟨ binary logarithm for Integers 228b ⟩
⟨ units of an Integer 189c ⟩
⟨ simple call for square root of an Integer 237c ⟩
⟨ converts a string to an Integer by function call 194a ⟩
⟨ function print of an Integer 192b ⟩
⟨ puts internal representation of an Integer on output stream 191c ⟩
⟨ puts an Integer on output stream 191b ⟩

```
#endif
```
◇

⟨ private memberfunctions of Integer 423 ⟩ ≡

```
Natural&  abs();
void      neg();
void      neg(const Integer&);
void      add(const Integer&, const Integer&);
void      sub(const Integer&, const Integer&);
void      mul(const Integer&, const Integer&);
void      div(const Integer&, const Integer&);
void      sqrt(const Integer&);

void      lshift(const Integer&, const size_t);
void      rshift(const Integer&, const size_t);
void      bitwise_and(const Integer&, const Integer&);
void      bitwise_or(const Integer&, const Integer&);
void      bitwise_xor(const Integer&, const Integer&);
void      bitwise_not(const Integer&);

void      add(const Integer&, const Natural&);
void      sub(const Integer&, const Natural&);
void      sub(const Natural&, const Integer&);
void      mul(const Integer&, const Natural&);
void      div(const Integer&, const Natural&);
void      div(const Natural&, const Integer&);

void      add(const Integer&, const SignDigit);
void      sub(const Integer&, const SignDigit);
void      mul(const Integer&, const SignDigit);
void      muladd(const Integer&, const SignDigit);
```

```
void      mulsub(const Integer&, const SignDigit);
◇
```

Macro referenced in 186b.

⟨ public memberfunctions and friends of Integer 424 ⟩ ≡

```
inline Integer&   operator=(const Integer&);
inline Integer&   operator=(const Natural&);
Integer&          operator+=(const Integer&);
Integer&          operator-=(const Integer&);
Integer&          operator*=(const Integer&);
Integer&          operator/=(const Integer&);
Integer&          operator%=(const Integer&);
Integer&          operator&=(const Integer&);
Integer&          operator|=(const Integer&);
Integer&          operator^=(const Integer&);

Integer&          operator+=(const Natural&);
Integer&          operator-=(const Natural&);
Integer&          operator*=(const Natural&);
Integer&          operator/=(const Natural&);
Integer&          operator%=(const Natural&);

Integer&          operator+=(const SignDigit);
Integer&          operator-=(const SignDigit);
Integer&          operator*=(const SignDigit);
Integer&          operator/=(const SignDigit);
Integer&          operator%=(const SignDigit);
Integer&          operator>>=(const size_t);
Integer&          operator<<=(const size_t);
inline SignDigit  operator=(const SignDigit);
Digit             operator&=(const Digit);
Integer&          operator|=(const Digit);

const Integer&    operator++();
const Integer&    operator--();
Integer           operator++(int);
Integer           operator--(int);


Integer(const binder_arguments<Integer, Integer, Integer_negate_tag>&);
Integer& operator=(const binder_arguments<Integer, Integer, Integer_negate_tag>&);

Integer(const binder_arguments<Integer, Integer, Integer_plus_tag>&);
Integer& operator=(const binder_arguments<Integer, Integer, Integer_plus_tag>&);

Integer(const binder_arguments<Integer, Integer, Integer_minus_tag>&);
Integer& operator=(const binder_arguments<Integer, Integer, Integer_minus_tag>&);

Integer(const binder_arguments<Integer, Integer, Integer_multiplies_tag>&);
Integer& operator=(const binder_arguments<Integer, Integer, Integer_multiplies_tag>&);

Integer(const binder_arguments<Integer, Integer, Integer_divides_tag>&);
Integer& operator=(const binder_arguments<Integer, Integer, Integer_divides_tag>&);

Integer(const binder_arguments<Integer, Integer, Integer_modulus_tag>&);
Integer& operator=(const binder_arguments<Integer, Integer, Integer_modulus_tag>&);
```

```
Integer(const binder_arguments<Integer, size_t, Integer_lshift_tag>&);
Integer& operator=(const binder_arguments<Integer, size_t, Integer_lshift_tag>&);

Integer(const binder_arguments<Integer, size_t, Integer_rshift_tag>&);
Integer& operator=(const binder_arguments<Integer, size_t, Integer_rshift_tag>&);

Integer(const binder_arguments<Integer, Integer, Integer_and_tag>&);
Integer& operator=(const binder_arguments<Integer, Integer, Integer_and_tag>&);

Integer(const binder_arguments<Integer, Integer, Integer_or_tag>&);
Integer& operator=(const binder_arguments<Integer, Integer, Integer_or_tag>&);

Integer(const binder_arguments<Integer, Integer, Integer_xor_tag>&);
Integer& operator=(const binder_arguments<Integer, Integer, Integer_xor_tag>&);

Integer(const binder_arguments<Integer, Integer, Integer_not_tag>&);
Integer& operator=(const binder_arguments<Integer, Integer, Integer_not_tag>&);


Integer(const binder_arguments<Integer, Natural, Integer_plus_tag>&);
Integer& operator=(const binder_arguments<Integer, Natural, Integer_plus_tag>&);

Integer(const binder_arguments<Integer, Natural, Integer_minus_tag>&);
Integer& operator=(const binder_arguments<Integer, Natural, Integer_minus_tag>&);

Integer(const binder_arguments<Natural, Integer, Integer_minus_tag>&);
Integer& operator=(const binder_arguments<Natural, Integer, Integer_minus_tag>&);

Integer(const binder_arguments<Integer, Natural, Integer_multiplies_tag>&);
Integer& operator=(const binder_arguments<Integer, Natural, Integer_multiplies_tag>&);

Integer(const binder_arguments<Integer, Natural, Integer_divides_tag>&);
Integer& operator=(const binder_arguments<Integer, Natural, Integer_divides_tag>&);

Integer(const binder_arguments<Natural, Integer, Integer_divides_tag>&);
Integer& operator=(const binder_arguments<Natural, Integer, Integer_divides_tag>&);


Integer(const binder_arguments<Integer, SignDigit, Integer_minus_tag>&);
Integer& operator=(const binder_arguments<Integer, SignDigit, Integer_minus_tag>&);
Integer(const binder_arguments<SignDigit, Integer, Integer_minus_tag>&);
Integer& operator=(const binder_arguments<SignDigit, Integer, Integer_minus_tag>&);

Integer(const binder_arguments<Integer, SignDigit, Integer_plus_tag>&);
Integer& operator=(const binder_arguments<Integer, SignDigit, Integer_plus_tag>&);

Integer(const binder_arguments<Integer, SignDigit, Integer_multiplies_tag>&);
Integer& operator=(const binder_arguments<Integer, SignDigit, Integer_multiplies_tag>&);
Integer& operator+=(const binder_arguments<Integer, SignDigit, Integer_multiplies_tag>&);
Integer& operator-=(const binder_arguments<Integer, SignDigit, Integer_multiplies_tag>&);

Integer(const binder_arguments<Integer, Integer, Integer_square_root_tag>&);
Integer& operator=(const binder_arguments<Integer, Integer, Integer_square_root_tag>&);
```

```
friend void       div(const Integer&, const Integer&, Integer&, Integer&);
friend void       div(const Integer&, const SignDigit, Integer&, SignDigit&);
friend void       swap(Integer&, Integer&);
friend void       sqrt(const Integer&, Integer&, Integer&);
inline friend int sign(const Integer&);
friend Integer    pow(const Integer&, const SignDigit);
friend Integer    pow(const Integer&, const Integer&);
friend ISTREAM&   operator>>(ISTREAM&, Integer&);


void        split(const size_t, Integer&, Integer&) const;
void        setbit(const size_t);
void        clearbit(const size_t);
bool        testbit(const size_t) const;
void        rand(const size_t);
const char* atoI(const char*, const Digit = 10);
bool        scan(ISTREAM&);
◇
```

Macro referenced in 186b.

"integer.cpp" 426 ≡

```
////////////////////////////////
//
// Piologie V 1.3.2
// multi-precision arithmetic
// Integer
//
// (c) 1996-2001 HiPiLib
//               www.hipilib.de
//
// Sebastian Wedeniwski
// 10/21/2001
//

#include "integer.h"


⟨ macro definitions for internal conditions 6b ⟩


////////////////// Integer Arithmetic ////////////////////////

⟨ addition of two Integers 200a ⟩
⟨ addition of an Integer with a Natural 200b ⟩
⟨ subtraction of two Integers 205a ⟩
⟨ subtraction of an Integer with a Natural 205b ⟩
⟨ subtraction of a Natural with an Integer 206a ⟩
⟨ division of two Integers 230a ⟩
⟨ square root of an Integer 237a ⟩
⟨ assign operator+= of an Integer with a SignDigit 204c ⟩
⟨ assign operator-= of an Integer with a SignDigit 210a ⟩
⟨ assign operator&= of an Integer with a Digit 216a ⟩
⟨ prefix incrementation of an Integer 197d ⟩
⟨ prefix decrementation of an Integer 198b ⟩
⟨ addition of an Integer with a SignDigit 201a ⟩
⟨ subtraction of an Integer with a SignDigit 206b ⟩
```

⟨ multiplication and addition of an Integer by a SignDigit 227c ⟩
⟨ multiplication and subtraction of an Integer by a SignDigit 228a ⟩
⟨ division of two Integers with remainder 229c ⟩
⟨ division of an Integer by a Natural 231b ⟩
⟨ division of a Natural by an Integer 232a ⟩
⟨ division of an Integer by a SignDigit 233 ⟩
⟨ function swap for Integers 190a ⟩
⟨ positive square root and remainder of an Integer 237b ⟩
⟨ converts a string to an Integer 193b ⟩
⟨ converts an Integer to a string 193a ⟩
⟨ gets internal representation of an Integer from input stream 192d ⟩
⟨ gets an Integer from input stream 192c ⟩
⟨ bitwise and of two Integers 213 ⟩
⟨ bitwise inclusive or of two Integers 216b ⟩
⟨ bitwise exclusive or of two Integers 219c ⟩
⟨ sets a bit in an Integer 221c ⟩
⟨ clears a bit in an Integer 222 ⟩
⟨ tests a bit in an Integer 223a ⟩
⟨ calculates an Integer random number 238b ⟩
⟨ power of an Integer by a SignDigit 229a ⟩
⟨ power of an Integer by an Integer 229b ⟩
⟨ calculates the root of an Integer 238a ⟩
⟨ extented greatest common divisor of two Integers 275a ⟩
◇

D.5 Rational

"rational.h" 427 ≡

```
////////////////////////////////
//
// Piologie V 1.3.2
// multi-precision arithmetic
// Rational
//
// (c) 1996-2001 HiPiLib
//               www.hipilib.de
//
// Sebastian Wedeniwski
// 10/21/2001
//

#ifndef _Include_Rational_H_
#define _Include_Rational_H_

#include "integer.h"

struct Rational_plus_tag {};
struct Rational_negate_tag {};
struct Rational_minus_tag {};
struct Rational_multiplies_tag {};
struct Rational_divides_tag {};
struct Rational_modulus_tag {};
struct Rational_square_root_tag {};
struct Rational_lshift_tag {};
struct Rational_rshift_tag {};
```

```
//#ifdef _Old_STD_
//template <class I, class N>
//#else
//template <class I, class N = I>
//#endif
class Rational {
public:
  ⟨ output variables for representation of a Rational 245d ⟩

private:
  Integer num;
  Natural den;

  void add(const Rational&, const Rational&);
  void sub(const Rational&, const Rational&);
  void mul(const Rational&, const Rational&);
  void sqr(const Rational&);
  void div(const Rational&, const Rational&);
  void lshift(const Rational&, size_t);
  void rshift(const Rational&, size_t);

public:
  Rational(const SignDigit = 0);
  Rational(const Integer&);
  Rational(const Integer&, const Natural&);
  Rational(const Rational&);
  ~Rational();

  const Integer&   numerator() const;
  const Natural&   denominator() const;

  SignDigit        operator=(const SignDigit);
  Rational&        operator=(const Integer&);
  Rational&        operator=(const Rational&);
  Rational&        operator+=(const Rational&);
  Rational&        operator-=(const Rational&);
  Rational&        operator*=(const Rational&);
  Rational&        operator/=(const Rational&);
  Rational&        operator<<=(const size_t);
  Rational&        operator>>=(const size_t);

  const Rational&  operator++();
  const Rational&  operator--();
  Rational         operator++(int);
  Rational         operator--(int);


  Rational(const binder_arguments<Rational, Rational, Rational_negate_tag>&);
  Rational& operator=(const binder_arguments<Rational, Rational, Rational_negate_tag>&);

  Rational(const binder_arguments<Rational, Rational, Rational_plus_tag>&);
  Rational& operator=(const binder_arguments<Rational, Rational, Rational_plus_tag>&);

  Rational(const binder_arguments<Rational, Rational, Rational_minus_tag>&);
  Rational& operator=(const binder_arguments<Rational, Rational, Rational_minus_tag>&);
```

```
  Rational(const binder_arguments<Rational, Rational, Rational_multiplies_tag>&);
  Rational& operator=(const binder_arguments<Rational, Rational, Rational_multiplies_tag>&);

  Rational(const binder_arguments<Rational, Rational, Rational_divides_tag>&);
  Rational& operator=(const binder_arguments<Rational, Rational, Rational_divides_tag>&);

  Rational(const binder_arguments<Rational, size_t, Rational_lshift_tag>&);
  Rational& operator=(const binder_arguments<Rational, size_t, Rational_lshift_tag>&);

  Rational(const binder_arguments<Rational, size_t, Rational_rshift_tag>&);
  Rational& operator=(const binder_arguments<Rational, size_t, Rational_rshift_tag>&);

  void        rand(const size_t);
  inline void swap(Rational&);
  bool        scan(ISTREAM&);
  const char* atoR(const char*, const Digit = 10);

  friend ISTREAM&     operator>>(ISTREAM&, Rational&);
  friend Rational     inv(const Rational&);
};

inline binder_arguments<Rational, Rational, Rational_negate_tag>
 operator-(const Rational&);
inline binder_arguments<Rational, Rational, Rational_plus_tag>
 operator+(const Rational&, const Rational&);
inline binder_arguments<Rational, Rational, Rational_minus_tag>
 operator-(const Rational&, const Rational&);
inline binder_arguments<Rational, Rational, Rational_multiplies_tag>
 operator*(const Rational&, const Rational&);
inline binder_arguments<Rational, Rational, Rational_divides_tag>
 operator/(const Rational&, const Rational&);
inline binder_arguments<Rational, size_t, Rational_lshift_tag>
 operator<<(const Rational&, const size_t&);
inline binder_arguments<Rational, size_t, Rational_rshift_tag>
 operator>>(const Rational&, const size_t&);


inline bool operator==(const Rational&, const Rational&);
inline bool operator!=(const Rational&, const Rational&);
inline bool operator<(const Rational&, const Rational&);
inline bool operator<=(const Rational&, const Rational&);
inline bool operator>(const Rational&, const Rational&);
inline bool operator>=(const Rational&, const Rational&);

inline bool operator==(const Rational&, const SignDigit);
inline bool operator!=(const Rational&, const SignDigit);
inline bool operator<(const Rational&, const SignDigit);
inline bool operator<=(const Rational&, const SignDigit);
inline bool operator>(const Rational&, const SignDigit);
inline bool operator>=(const Rational&, const SignDigit);

inline void           swap(Rational&, Rational&);
inline const Integer& numerator(const Rational&);
inline const Natural& denominator(const Rational&);
inline Rational       abs(const Rational&);
```

```
inline int            sign(const Rational&);
inline Rational       atoR(const char*, const Digit = 10);
char*                 Rtoa(const Rational&, char*, const Digit = 10);
inline Rational::rep  print(const Rational&, bool = false);
ISTREAM&              operator>>(ISTREAM&, Rational&);
inline OSTREAM&       operator<<(OSTREAM&, const Rational::rep&);
inline OSTREAM&       operator<<(OSTREAM&, const Rational&);
Integer               ceil(const Rational&);
Integer               floor(const Rational&);
inline Integer        round(const Rational&);
Integer               trunc(const Rational&);
Rational              pow(const Rational&, const SignDigit);
Rational              pow(const Rational&, const Integer&);
```

⟨ class Rational 241a ⟩

```
////////////////////////// Inline-Implementation ////////////////////
```

⟨ squaring of a Rational 257b ⟩
⟨ default constructor Rational 241b ⟩
⟨ overloaded constructor Rational for Integer 241c ⟩
⟨ copy constructor Rational 242a ⟩
⟨ destructor Rational 242b ⟩
⟨ quotient field numerator (memberfunction) 242c ⟩
⟨ quotient field denominator (memberfunction) 243a ⟩
⟨ assign operator= for a Rational with a SignDigit 244b ⟩
⟨ assign operator= for a Rational with a Integer 245a ⟩
⟨ assign operator= for Rationals 244a ⟩
⟨ assign operator+= for Rationals 253a ⟩
⟨ assign operator-= for Rationals 255a ⟩
⟨ assign operator*= for Rationals 259c ⟩
⟨ assign operator/= for Rationals 262a ⟩
⟨ assign operator<<= of a Rational 256a ⟩
⟨ assign operator>>= of a Rational 257a ⟩
⟨ prefix incrementation of a Rational 250c ⟩
⟨ prefix decrementation of a Rational 250e ⟩
⟨ postfix incrementation of a Rational 250d ⟩
⟨ postfix decrementation of a Rational 251a ⟩
⟨ negation operator- of a Rational 251b ⟩
⟨ additive operator+ for Rationals 252c ⟩
⟨ additive operator- for Rationals 254c ⟩
⟨ multiplicative operator* for Rationals 259b ⟩
⟨ multiplicative operator/ for Rationals 261c ⟩
⟨ shift operator<< of a Rational 255c ⟩
⟨ shift operator>> of a Rational 256c ⟩
⟨ calculates a Rational random number 264c ⟩
⟨ function swap for Rationals 243e ⟩
⟨ comparison operator== for Rationals 248a ⟩
⟨ comparison operator!= for Rationals 248b ⟩
⟨ comparison operator< for Rationals 248c ⟩
⟨ comparison operator<= for Rationals 248d ⟩
⟨ comparison operator> for Rationals 248e ⟩
⟨ comparison operator>= for Rationals 249a ⟩
⟨ comparison operator== of a Rational with a SignDigit 249b ⟩

⟨ comparison operator!= of a Rational with a SignDigit 249c ⟩
⟨ comparison operator< of a Rational with a SignDigit 249d ⟩
⟨ comparison operator<= of a Rational with a SignDigit 249e ⟩
⟨ comparison operator> of a Rational with a SignDigit 250a ⟩
⟨ comparison operator>= of a Rational with a SignDigit 250b ⟩
⟨ quotient field numerator 242d ⟩
⟨ quotient field denominator 243b ⟩
⟨ absolute value of a Rational 243c ⟩
⟨ sign of a Rational 243d ⟩
⟨ converts a string to an Rational by function call 247c ⟩
⟨ function print of a Rational 246a ⟩
⟨ puts internal representation of a Rational on output stream 245c ⟩
⟨ puts a Rational on output stream 245b ⟩
⟨ function round of a Rational 264a ⟩

```
#endif
```
◇

"rational.cpp" 431 ≡

```
////////////////////////////////
//
// Piologie V 1.3.2
// multi-precision arithmetic
// Rational
//
// (c) 1996-2001 HiPiLib
//               www.hipilib.de
//
// Sebastian Wedeniwski
// 10/21/2001
//

#include "rational.h"
#include <string.h>


////////////////// Rational Arithmetic ///////////////////////
```

⟨ addition of two Rationals 251c ⟩
⟨ subtraction of two Rationals 253b ⟩
⟨ multiplication of two Rationals 257c ⟩
⟨ division of two Rationals 259d ⟩
⟨ multiplication of a Rational by a power of 2 255b ⟩
⟨ division of a Rational by a power of 2 256b ⟩
⟨ overloaded constructor Rational for Integer and Natural 241d ⟩
⟨ gets internal representation of a Rational from input stream 246c ⟩
⟨ gets a Rational from input stream 246b ⟩
⟨ converts a Rational to a string 247a ⟩
⟨ converts a string to a Rational 247b ⟩

⟨ inversion of a Rational 262b ⟩
⟨ function ceil of a Rational 263a ⟩
⟨ function floor of a Rational 263b ⟩
⟨ function trunc of a Rational 264b ⟩
⟨ power of a Rational by a SignDigit 262c ⟩
⟨ power of a Rational by an Integer 262d ⟩
◇

D.6 Modulare Arithmetik

"modulo.h" 432a ≡

```
////////////////////////////////
//
// modulo arithmetic for Piologie
// multi-precision arithmetic
//
// Sebastian Wedeniwski
// 01/02/1999
//


#ifndef _Include_Modulo_H_
#define _Include_Modulo_H_

⟨ modular power algorithm 277 ⟩
⟨ modular inverse 278 ⟩
⟨ modular square root 282 ⟩
⟨ chinese remainder theorem 279 ⟩

#endif
```

◇

D.7 Number Theory

"nmbrthry.h" 432b ≡

```
////////////////////////////////
//
// Piologie V 1.3.2
// multi-precision arithmetic
// Number Theory Package
//
// (c) 1996-2001 HiPiLib
//               www.hipilib.de
//
// Sebastian Wedeniwski
// 10/21/2001
//

#ifndef _Include_NumberTheory_H_
#define _Include_NumberTheory_H_

#include "natural.h"
#ifndef _Old_STD_
# include "modulo.h"
#endif

Natural factorial(const Digit a);
Natural binomial(const Digit a, const Digit b);
Natural fibonacci(Digit n);
void    pell(const Digit x, Natural& u, Natural& v);

#ifndef _Old_STD_
```

```
bool    MillerRabin(unsigned int i, const Natural& n);
bool    isprime(const Natural& n);
Natural euler(Natural a);
#endif


class Primes : public NumberBase {
private:
  static unsigned char* primes;
  unsigned char* primPos;
  Digit          primNumber;
  unsigned char  idx;

public:
  Primes();
  ~Primes();

  void   destroy();
  Digit  firstPrime();
  bool   nextPrime(Digit&);
  Digit  lastPrime() const;
  size_t numberOfPrimes(Digit) const;
};

inline Primes::~Primes()
{
}
```

⟨ getting first prime 286b ⟩

```
#ifndef _Old_STD_
```

⟨ strong pseudoprime test 289a ⟩

⟨ prime factorization of a Natural 296 ⟩

⟨ jacobi symbol 281 ⟩

```
#endif

#endif
```

◇

"nmbrthry.cpp" 433 ≡

```
////////////////////////////////
//
// Piologie V 1.3.2
// multi-precision arithmetic
// Number Theory Package
//
// (c) 1996-2001 HiPiLib
//               www.hipilib.de
//
// Sebastian Wedeniwski
// 10/21/2001
//

#include "nmbrthry.h"

#ifndef _Old_STD_
# include <list>
#endif
```

```
⟨ new operator 396e ⟩

const size_t     SieveSize = (size_t(1) << ((BETA > 32)? 16 : BETA/2)) - 1;

unsigned char*  Primes::primes     = 0;

Primes::Primes()
{
  if (primes) { firstPrime(); return; }
  // Generating prime numbers
  ⟨ generating prime numbers 286a ⟩
}

void Primes::destroy()
{
  delete[] primes;
  primPos = primes = 0;
}

⟨ getting next prime 287a ⟩
⟨ getting last prime 287b ⟩
⟨ getting the number of primes 288 ⟩
⟨ calculates the function factorial 306 ⟩
⟨ calculates the function binomial coefficient 308 ⟩
⟨ calculates the n-th Fibonacci number 22a ⟩
⟨ solving Pell's equation 276 ⟩
#ifndef _Old_STD_
⟨ Miller-Rabin test 289b ⟩
⟨ primality test 290 ⟩
⟨ euler function 304 ⟩
#endif
◇
```

D.8 Pi

"pi.h" 434 ≡

```
////////////////////////////////
//
// Piologie V 1.3.2
// multi-precision arithmetic
// Calculation of Constants
//
// (c) 1996-2001 HiPiLib
//               www.hipilib.de
//
// Sebastian Wedeniwski
// 10/21/2001
//

#ifndef _Include_Pi_H_
#define _Include_Pi_H_

#include "integer.h"
#include "nmbrthry.h"
```

⟨ class Fixed 311a ⟩

```
inline OSTREAM& operator<<(OSTREAM&, Fixed);

class Pi {
private:
  Fixed pi;

  void stoermer(const size_t, Natural&);
  void schoenhage(const size_t, Natural&);
  void chudnovsky(Digit, const Digit, Integer&, Integer&, Integer&) const;
  void chudnovsky2(const Digit, const Digit, Integer&, Integer&) const;
  void sqrt_series(Digit, const Digit, Integer&, Integer&, Integer&) const;
  void sqrt_series2(const Digit, const Digit, Integer&, Integer&) const;

public:
  Pi(const size_t);

  inline friend OSTREAM& operator<<(OSTREAM&, const Pi&);
};

class Sqrt {
private:
  Digit d;
  Fixed sqrt;
  Natural u,v;

  void series(Digit, const Digit, Natural&, Natural&, Integer&) const;
  void series2(const Digit, const Digit, Natural&, Integer&) const;

public:
  Sqrt(const Digit, const size_t);

  inline friend OSTREAM& operator<<(OSTREAM&, const Sqrt&);
};

class Zeta3 {
private:
  Fixed zeta;

  void linear(const size_t, Natural&) const;
  void series(Digit, const Digit, Integer&, Integer&, Integer&) const;
  void series2(const Digit, const Digit, Integer&, Integer&) const;

public:
  Zeta3(const size_t);

  inline friend OSTREAM& operator<<(OSTREAM&, const Zeta3&);
};

class Exp1 : protected Natural {
private:
  Fixed exp;

  void linear(const size_t, Natural&) const;
```

```
  void series(Digit, const Digit, Natural&, Natural&) const;

public:
  Exp1(const size_t);

  inline friend OSTREAM& operator<<(OSTREAM&, const Exp1&);
};

class Ln : protected Natural {
private:
  Fixed ln;
  Digit u,v,x;

  Natural atanh_inv_linear(const Digit, const size_t) const;
  void    atanh_inv_series(Digit, const Digit, Natural&, Natural&, Natural&) const;
  Natural atanh_inv_series(const Digit);
  Natural ln2(const size_t);

  void    linear(const size_t, Natural&) const;
  void    series(Digit, const Digit, Natural&, Integer&, Digit&, Integer&) const;

public:
  Ln(const Digit, const size_t);

  inline const Natural& value() const;

  inline friend OSTREAM& operator<<(OSTREAM&, const Ln&);
};

class EulerGamma : protected Natural {
private:
  Fixed euler;

  void linear(const size_t, Natural&);
  void series(Digit, const Digit,
              Natural&, Natural&, Natural&, Natural&, Natural&, Natural&) const;
  void series2(Digit, const Digit, Natural&, Natural&, Natural&, Natural&) const;

public:
  EulerGamma(const size_t);

  inline friend OSTREAM& operator<<(OSTREAM&, const EulerGamma&);
};

////////////////// Inline-Implementation //////////////////////
```

⟨ inline implementation of class Fixed 311b ⟩
⟨ puts Pi on output stream 325 ⟩
⟨ puts Sqrt-class on output stream 329 ⟩
⟨ puts Zeta3-class on output stream 333a ⟩
⟨ puts Exp1-class on output stream 336a ⟩

```
inline const Natural& Ln::value() const
{
  return ln.value();
}
```

⟨ puts Ln-class on output stream 340a ⟩

⟨ puts EulerGamma-class on output stream 345a ⟩

```
#endif
```
◇

"pi.cpp" 437 ≡

```
////////////////////////////////
//
// Piologie V 1.3.2
// multi-precision arithmetic
// Calculation of Constants
//
// (c) 1996-2001 HiPiLib
//               www.hipilib.de
//
// Sebastian Wedeniwski
// 10/21/2001
//

#include "pi.h"
```

⟨ macro definitions for internal conditions 6b ⟩

⟨ puts Fixed on output stream 312 ⟩
⟨ Pi-Calculation with Stoermer-Algorithms 317 ⟩
⟨ Pi-Calculation with Schönhage-Algorithms 324a ⟩
⟨ Pi-Calculation with Chudnovsky-Algorithms 318, ... ⟩
⟨ Pi-Sqrt-Series-Calculation 321b, ... ⟩
⟨ select π-Algorithm 324b ⟩

```
#ifdef _Old_STD_
#include <math.h>
#else
#include <cmath>
#endif
#ifdef log2
# undef log2
#endif
```

⟨ Sqrt-Series-Calculation 326, ... ⟩
⟨ select Sqrt-Algorithmus 328 ⟩
⟨ Zeta(3)-Series-Calculation 331, ... ⟩
⟨ Zeta(3)-Calculation 330 ⟩
⟨ select $\zeta(3)$-Algorithm 332b ⟩
⟨ Exp(1)-Calculation 333b ⟩
⟨ Exp(1)-Series-Calculation 334 ⟩
⟨ select Exp(1)-Algorithm 335 ⟩
⟨ Arctanh-Calculation 336b ⟩
⟨ Ln(2)-Calculation 337 ⟩
⟨ Ln-Calculation 338a ⟩
⟨ Ln-Series-Calculation 338b ⟩
⟨ select Ln-Algorithm 339 ⟩
⟨ EulerGamma-Calculation 340b ⟩
⟨ EulerGamma-Series-Calculation 341 ⟩
⟨ select EulerGamma-Algorithm 344 ⟩
◇

D.9 Makefile

"makefile" 438 ≡

```
#################################
##
## Piologie V 1.3.2
## multi-precision arithmetic
## Makefile
##
## (c) 1996-2001 HiPiLib
##               www.hipilib.de
##
## Sebastian Wedeniwski
## 10/21/2001
##

#
# include the configuration file
#

include config

#
# delete a file:
#
# DELETE = del       # Windows / DOS / OS/2
DELETE = rm -f     # UNIX / MKS-Tools / GNU-Tools

#
# copy a file:
#
# COPY = copy       # Windows / DOS / OS/2
COPY = cp         # UNIX / MKS-Tools / GNU-Tools


PIOLOGIE_OBJS   = digit.$(OBJ) natural.$(OBJ) integer.$(OBJ) rational.$(OBJ)
SUPPLEMENT_OBJS = nmbrthry.$(OBJ) pi.$(OBJ)
ALL_OBJS        = $(PIOLOGIE_OBJS) $(SUPPLEMENT_OBJS)

all: lib

objs: $(ALL_OBJS)

lib: $(PIOLOGIE_OBJS)
   $(DELETE) piologie.$(LIB_SUFFIX)
   $(MAKE_KEYLIB)

keylib: digit.$(SUFFIX) digit.h test.$(SUFFIX)
   $(COMPILE_ONLY) $(CFLAGS) $(CFLAGS_KEY) digit.$(SUFFIX)
   $(MAKE) objs
   $(DELETE) piologie.$(LIB_SUFFIX)
   $(MAKE_KEYLIB)
   $(COMPILE_ONLY) $(CFLAGS) $(CFLAGS_KEY) test.$(SUFFIX)
   $(LINK_TEST)

piologie.$(LIB_SUFFIX): $(ALL_OBJS)
```

```
	$(DELETE) $@
	$(MAKE_LIB)

digit.$(SUFFIX):
	$(COPY) digit.cpp $@

digit.$(OBJ): digit.$(SUFFIX) digit.h
	$(COMPILE_ONLY) $(CFLAGS) digit.$(SUFFIX)

natural.$(SUFFIX):
	$(COPY) natural.cpp $@

natural.$(OBJ): natural.$(SUFFIX) natural.h digit.h
	$(COMPILE_ONLY) $(CFLAGS) natural.$(SUFFIX)

integer.$(SUFFIX):
	$(COPY) integer.cpp $@

integer.$(OBJ): integer.$(SUFFIX) integer.h natural.h digit.h
	$(COMPILE_ONLY) $(CFLAGS) integer.$(SUFFIX)

rational.$(SUFFIX):
	$(COPY) rational.cpp $@

rational.$(OBJ): rational.$(SUFFIX) rational.h integer.h natural.h digit.h
	$(COMPILE_ONLY) $(CFLAGS) rational.$(SUFFIX)

nmbrthry.$(SUFFIX):
	$(COPY) nmbrthry.cpp $@

nmbrthry.$(OBJ): nmbrthry.$(SUFFIX) modulo.h natural.h digit.h
	$(COMPILE_ONLY) $(CFLAGS) nmbrthry.$(SUFFIX)

pi.$(SUFFIX):
	$(COPY) pi.cpp $@

pi.$(OBJ): pi.$(SUFFIX) pi.h nmbrthry.h integer.h natural.h digit.h
	$(COMPILE_ONLY) $(CFLAGS) pi.$(SUFFIX)


test.$(SUFFIX):
	$(COPY) test.cpp $@

test: test.$(SUFFIX) piologie.$(LIB_SUFFIX)
	$(COMPILE_ONLY) $(CFLAGS) $@.$(SUFFIX)
	$(LINK)

cfrac.$(SUFFIX):
	$(COPY) cfrac.cpp $@

cfrac: cfrac.$(SUFFIX) piologie.$(LIB_SUFFIX)
	$(COMPILE_ONLY) $(CFLAGS) $@.$(SUFFIX)
	$(LINK)

check.$(SUFFIX):
	$(COPY) check.cpp $@
```

```
check: check.$(SUFFIX) piologie.$(LIB_SUFFIX)
   $(COMPILE_ONLY) $@.$(SUFFIX)
   $(LINK)

constant.$(SUFFIX):
   $(COPY) constant.cpp $@

constant: constant.$(SUFFIX) piologie.$(LIB_SUFFIX)
   $(COMPILE_ONLY) $(CFLAGS) $@.$(SUFFIX)
   $(LINK)

zeta.$(SUFFIX):
   $(COPY) zeta.cpp $@

zeta: zeta.$(SUFFIX) piologie.$(LIB_SUFFIX)
   $(COMPILE_ONLY) $(CFLAGS) $@.$(SUFFIX)
   $(LINK)
   $(MAKE) makesh

makesh.$(SUFFIX):
   $(COPY) makesh.cpp $@

makesh: makesh.$(SUFFIX)
   $(COMPILE_ONLY) $(CFLAGS) $@.$(SUFFIX)
   $(LINK)


dvi: piologie.dvi manual.dvi
   cd doc
   cd src
   nuweb -o piologie.w
   latex piologie.tex
   nuweb -o piologie.w
   makeindex piologie.idx
   latex piologie.tex
   latex manual.tex
   latex manual.tex
   cd ..
   cd ..

ps: piologie.ps manual.ps dvi
   cd doc
   cd src
   dvips piologie.dvi
   dvips manual.dvi
   cd ..
   cd ..

clean:
   $(DELETE) *.$(OBJ)


#
# Supported Compiler
#
```

```
ap: clean
   $(COPY) config.ap config; make

borland: clean
   $(COPY) config.i386_bc config; make

dec: clean
   $(COPY) config.dec config; make

edg: clean
   $(COPY) config.edg config; make

gnu: clean
   $(COPY) config.gnu config; make

gnu28: clean
   $(COPY) config.gnu28 config; make

gnu_mips4: clean
   $(COPY) config.gnu_mips4 config; make

gnu_sparc: clean
   $(COPY) config.gnu_sparc config; make

gnu28_sparc: clean
   $(COPY) config.gnu28_sparc config; make

hp: clean
   $(COPY) config.hp config; make

hpa: clean
   $(COPY) config.hpa config; make

i386_ibm: clean
   $(COPY) config.i386_ibm config
   make

kai: clean
   $(COPY) config.kai config; make

os390: clean
   $(COPY) config.os390 config
   make

ppc_ibm: clean
   $(COPY) config.ppc_ibm config
   make

sgi: clean
   $(COPY) config.sgi config; make

sgi_8000: clean
   $(COPY) config.sgi_8000 config; make

sun: clean
```

```
   $(COPY) config.sun config; make

visual: clean
   $(COPY) config.i386_vc config; make

watcom: clean
   $(COPY) config.i386_wc config; make
◇
```

D.9.1 Standardeinstellung

"config" 442a ≡

```
#
# Default: GNU C++ 2.8.0:
#

# The C++ compiler:
CC           = g++

# The flags for the C++ compiler:
CFLAGS       = -O3
CFLAGS_KEY   = -D_NEEDS_PIOLOGIE_KEY_

# Suffix of the object file names:
OBJ          = o

# Suffix of the library file name:
LIB_SUFFIX   = a

# Command to compile only:
COMPILE_ONLY = $(CC) -c

# Flag to specify the output file name:
OUTPUT       = -o $@

# Command to make the library:
MAKE_LIB     = ar cr $@ $(ALL_OBJS); ranlib $@
MAKE_KEYLIB  = ar cr piologie.$(LIB_SUFFIX) $(PIOLOGIE_OBJS); ranlib piologie.$(LIB_SUFFIX)

# Command to call the linker:
LINK         = $(CC) $(CFLAGS) $@.$(OBJ) piologie.$(LIB_SUFFIX) -o $@
LINK_TEST    = $(CC) $(CFLAGS) test.$(OBJ) $(SUPPLEMENT_OBJS) piologie.$(LIB_SUFFIX) -o test

# Suffix of the C++ file names:
SUFFIX       = cpp
◇
```

D.9.2 Apogee C++

"config.ap" 442b ≡

```
#
# Apogee C++ Release AC3.0:
#
CC           = apCC
CFLAGS       = -fast -O5 -D_Old_STD_ -D_Unknown_Apogee_Bug_
```

```
CFLAGS_KEY   = -D_NEEDS_PIOLOGIE_KEY_
OBJ          = o
LIB_SUFFIX   = a
COMPILE_ONLY = $(CC) -c
OUTPUT       = -o $@
MAKE_LIB     = ar cr $@ $(ALL_OBJS); ranlib $@
MAKE_KEYLIB  = ar cr piologie.$(LIB_SUFFIX) $(PIOLOGIE_OBJS); ranlib piologie.$(LIB_SUFFIX)
LINK         = $(CC) $(CFLAGS) $@.$(OBJ) piologie.$(LIB_SUFFIX) -o $@
LINK_TEST    = $(CC) $(CFLAGS) test.$(OBJ) $(SUPPLEMENT_OBJS) piologie.$(LIB_SUFFIX) -o test
SUFFIX       = cpp
◇
```

D.9.3 Borland C++

"config.i386_bc" 443a ≡

```
#
# Pentium, Borland C++ 5.0:
#
CC           = bcc32
CFLAGS       = -O2 -OS -Oc -w- -D_Old_STD_
CFLAGS_KEY   = -D_NEEDS_PIOLOGIE_KEY_
OBJ          = obj
LIB_SUFFIX   = lib
COMPILE_ONLY = $(CC) -c
OUTPUT       =
MAKE_LIB     = tlib $@ -+digit.obj -+natural.obj -+integer.obj -+rational.obj \
               -+nmbrthry.obj -+pi.obj
MAKE_KEYLIB  = tlib piologie -+digit.obj -+natural.obj -+integer.obj -+rational.obj
LINK         = $(CC) $@.$(OBJ) piologie.$(LIB_SUFFIX)
LINK_TEST    = $(CC) test.$(OBJ) $(SUPPLEMENT_OBJS) piologie.$(LIB_SUFFIX)
SUFFIX       = cpp
◇
```

D.9.4 DEC C++

"config.dec" 443b ≡

```
#
# DEC Alpha 150 Mhz, DEC C++ 5.0:
#
CC           = cxx
CFLAGS       = -O2 -D_Old_STD_
CFLAGS_KEY   = -D_NEEDS_PIOLOGIE_KEY_
OBJ          = o
LIB_SUFFIX   = a
COMPILE_ONLY = $(CC) -c
OUTPUT       = -o $@
MAKE_LIB     = ar cr $@ $(ALL_OBJS); ranlib $@
MAKE_KEYLIB  = ar cr piologie.$(LIB_SUFFIX) $(PIOLOGIE_OBJS); ranlib piologie.$(LIB_SUFFIX)
LINK         = $(CC) $(CFLAGS) $@.$(OBJ) piologie.$(LIB_SUFFIX) -o $@
LINK_TEST    = $(CC) $(CFLAGS) test.$(OBJ) $(SUPPLEMENT_OBJS) piologie.$(LIB_SUFFIX) -o test
SUFFIX       = cc
◇
```

D.9.5 Edison Design Group C++ front end

"config.edg" 444a ≡

```
#
# EDG C++ front end 2.33:
#
CC           = eccp
CFLAGS       = -O2 -DSTATIC_VS_INLINE=static -D_Old_STD_
CFLAGS_KEY   = -D_NEEDS_PIOLOGIE_KEY_
OBJ          = o
LIB_SUFFIX   = a
COMPILE_ONLY = $(CC) -c
OUTPUT       = -o $@
MAKE_LIB     = ar cr $@ $(ALL_OBJS); ranlib $@
MAKE_KEYLIB  = ar cr piologie.$(LIB_SUFFIX) $(PIOLOGIE_OBJS); ranlib piologie.$(LIB_SUFFIX)
LINK         = $(CC) $(CFLAGS) $@.$(OBJ) piologie.$(LIB_SUFFIX) -o $@
LINK_TEST    = $(CC) $(CFLAGS) test.$(OBJ) $(SUPPLEMENT_OBJS) piologie.$(LIB_SUFFIX) -o test
SUFFIX       = cpp
◇
```

D.9.6 GNU C++

"config.gnu" 444b ≡

```
#
# GNU C++ 2.6.1 - 2.7.2:
#
CC           = g++
CFLAGS       = -O3 -D_Old_STD_
CFLAGS_KEY   = -D_NEEDS_PIOLOGIE_KEY_
OBJ          = o
LIB_SUFFIX   = a
COMPILE_ONLY = $(CC) -c
OUTPUT       = -o $@
MAKE_LIB     = ar cr $@ $(ALL_OBJS); ranlib $@
MAKE_KEYLIB  = ar cr piologie.$(LIB_SUFFIX) $(PIOLOGIE_OBJS); ranlib piologie.$(LIB_SUFFIX)
LINK         = $(CC) $(CFLAGS) $@.$(OBJ) piologie.$(LIB_SUFFIX) -o $@
LINK_TEST    = $(CC) $(CFLAGS) test.$(OBJ) $(SUPPLEMENT_OBJS) piologie.$(LIB_SUFFIX) -o test
SUFFIX       = cpp
◇
```

"config.gnu28" 444c ≡

```
#
# GNU C++ 2.8.0:
#
CC           = g++
CFLAGS       = -O3
CFLAGS_KEY   = -D_NEEDS_PIOLOGIE_KEY_
OBJ          = o
LIB_SUFFIX   = a
COMPILE_ONLY = $(CC) -c
OUTPUT       = -o $@
MAKE_LIB     = ar cr $@ $(ALL_OBJS); ranlib $@
MAKE_KEYLIB  = ar cr piologie.$(LIB_SUFFIX) $(PIOLOGIE_OBJS); ranlib piologie.$(LIB_SUFFIX)
LINK         = $(CC) $(CFLAGS) $@.$(OBJ) piologie.$(LIB_SUFFIX) -o $@
LINK_TEST    = $(CC) $(CFLAGS) test.$(OBJ) $(SUPPLEMENT_OBJS) piologie.$(LIB_SUFFIX) -o test
SUFFIX       = cpp
◇
```

"config.gnu_mips4" 445a ≡

```
#
# MIPS R8000, GNU C++ 2.7.2:
#
CC           = g++
CFLAGS       = -O3 -mips4 -D_Old_STD_
CFLAGS_KEY   = -D_NEEDS_PIOLOGIE_KEY_
OBJ          = o
LIB_SUFFIX   = a
COMPILE_ONLY = $(CC) -c
OUTPUT       = -o $@
MAKE_LIB     = ar cr $@ $(ALL_OBJS); ranlib $@
MAKE_KEYLIB  = ar cr piologie.$(LIB_SUFFIX) $(PIOLOGIE_OBJS); ranlib piologie.$(LIB_SUFFIX)
LINK         = $(CC) $(CFLAGS) $@.$(OBJ) piologie.$(LIB_SUFFIX) -o $@
LINK_TEST    = $(CC) $(CFLAGS) test.$(OBJ) $(SUPPLEMENT_OBJS) piologie.$(LIB_SUFFIX) -o test
SUFFIX       = cpp
◇
```

"config.gnu_sparc" 445b ≡

```
#
# SuperSPARC, GNU C++ 2.7.2:
#
CC           = g++
CFLAGS       = -O3 -msupersparc -D_Old_STD_
CFLAGS_KEY   = -D_NEEDS_PIOLOGIE_KEY_
OBJ          = o
LIB_SUFFIX   = a
COMPILE_ONLY = $(CC) -c
OUTPUT       = -o $@
MAKE_LIB     = ar cr $@ $(ALL_OBJS); ranlib $@
MAKE_KEYLIB  = ar cr piologie.$(LIB_SUFFIX) $(PIOLOGIE_OBJS); ranlib piologie.$(LIB_SUFFIX)
LINK         = $(CC) $(CFLAGS) $@.$(OBJ) piologie.$(LIB_SUFFIX) -o $@
LINK_TEST    = $(CC) $(CFLAGS) test.$(OBJ) $(SUPPLEMENT_OBJS) piologie.$(LIB_SUFFIX) -o test
SUFFIX       = cpp
◇
```

"config.gnu28_sparc" 445c ≡

```
#
# SuperSPARC, GNU C++ 2.8.0:
#
CC           = g++
CFLAGS       = -O3 -msupersparc
CFLAGS_KEY   = -D_NEEDS_PIOLOGIE_KEY_
OBJ          = o
LIB_SUFFIX   = a
COMPILE_ONLY = $(CC) -c
OUTPUT       = -o $@
MAKE_LIB     = ar cr $@ $(ALL_OBJS); ranlib $@
MAKE_KEYLIB  = ar cr piologie.$(LIB_SUFFIX) $(PIOLOGIE_OBJS); ranlib piologie.$(LIB_SUFFIX)
LINK         = $(CC) $(CFLAGS) $@.$(OBJ) piologie.$(LIB_SUFFIX) -o $@
LINK_TEST    = $(CC) $(CFLAGS) test.$(OBJ) $(SUPPLEMENT_OBJS) piologie.$(LIB_SUFFIX) -o test
SUFFIX       = cpp
◇
```

D.9.7 HP C++

"config.hp" 446a ≡

```
#
# HP PA, HP C++ A.10.22:
#
CC           = CC
CFLAGS       = +O4 +a1 -DSTATIC_VS_INLINE=static -D_Old_STD_
CFLAGS_KEY   = -D_NEEDS_PIOLOGIE_KEY_
OBJ          = o
LIB_SUFFIX   = a
COMPILE_ONLY = $(CC) -c
OUTPUT       = -o $@
MAKE_LIB     = ar cr $@ $(ALL_OBJS); ranlib $@
MAKE_KEYLIB  = ar cr piologie.$(LIB_SUFFIX) $(PIOLOGIE_OBJS); ranlib piologie.$(LIB_SUFFIX)
LINK         = $(CC) $(CFLAGS) $@.$(OBJ) piologie.$(LIB_SUFFIX) -o $@
LINK_TEST    = $(CC) $(CFLAGS) test.$(OBJ) $(SUPPLEMENT_OBJS) piologie.$(LIB_SUFFIX) -o test
SUFFIX       = cpp
◇
```

"config.hpa" 446b ≡

```
#
# HP PA-RISC 2.0, HP aC++ A.01.00:
#
CC           = aCC
CFLAGS       = +O4 +DA2.0 -D_Old_STD_
CFLAGS_KEY   = -D_NEEDS_PIOLOGIE_KEY_
OBJ          = o
LIB_SUFFIX   = a
COMPILE_ONLY = $(CC) -c
OUTPUT       = -o $@
MAKE_LIB     = ar cr $@ $(ALL_OBJS); ranlib $@
MAKE_KEYLIB  = ar cr piologie.$(LIB_SUFFIX) $(PIOLOGIE_OBJS); ranlib piologie.$(LIB_SUFFIX)
LINK         = $(CC) $(CFLAGS) $@.$(OBJ) piologie.$(LIB_SUFFIX) -o $@
LINK_TEST    = $(CC) $(CFLAGS) test.$(OBJ) $(SUPPLEMENT_OBJS) piologie.$(LIB_SUFFIX) -o test
SUFFIX       = cpp
◇
```

D.9.8 IBM C++ for OS/390 V 2.6

"config.os390" 446c ≡

```
#
# S/390, IBM C++ for OS/390
#
CC           = cxx
CFLAGS       = -O  -D_Old_STD_ -DDEFINE_VS_CONST
CFLAGS_KEY   = -D_NEEDS_PIOLOGIE_KEY_
OBJ          = o
LIB_SUFFIX   = a
OUTPUT       = -o $@
COMPILE_ONLY = $(CC) $(CFLAGS) -c
OUTPUT       = -o $@
MAKE_LIB     = ar cr $@ $(ALL_OBJS)
MAKE_KEYLIB  = ar cr piologie.$(LIB_SUFFIX) $(PIOLOGIE_OBJS)
LINK         = $(CC) -o $@ $@.$(OBJ) piologie.$(LIB_SUFFIX)
```

```
LINK_TEST    = $(CC) -o test test.$(OBJ) $(SUPPLEMENT_OBJS) piologie.$(LIB_SUFFIX)
SUFFIX       = C
◇
```

D.9.9 IBM CSet++ for AIX 3.1.5

"config.ppc_ibm" 447a ≡

```
#
# PowerPC, IBM CSet++:
#
CC           = icc
CFLAGS       = -O3 -D_Old_STD_
CFLAGS_KEY   = -D_NEEDS_PIOLOGIE_KEY_
OBJ          = o
LIB_SUFFIX   = a
COMPILE_ONLY = $(CC) -c
OUTPUT       = -o $@
MAKE_LIB     = ar cr $@ $(ALL_OBJS); ranlib $@
MAKE_KEYLIB  = ar cr piologie.$(LIB_SUFFIX) $(PIOLOGIE_OBJS); ranlib piologie.$(LIB_SUFFIX)
LINK         = $(CC) $(CFLAGS) $@.$(OBJ) piologie.$(LIB_SUFFIX) -o $@
LINK_TEST    = $(CC) $(CFLAGS) test.$(OBJ) $(SUPPLEMENT_OBJS) piologie.$(LIB_SUFFIX) -o test
SUFFIX       = cpp
◇
```

D.9.10 IBM Visual Age C++

"config.i386_ibm" 447b ≡

```
#
# Pentium, IBM C/C++ 3.6:
#
CC           = icc
CFLAGS       = /Q+ /G5 /Gf+ /Gi+ /O+ /Om- /D_Old_STD_
CFLAGS_KEY   = /D_NEEDS_PIOLOGIE_KEY_
OBJ          = obj
LIB_SUFFIX   = lib
COMPILE_ONLY = $(CC) /c
OUTPUT       =
MAKE_LIB     = ilib /nologo $(ALL_OBJS) /out:piologie.$(LIB_SUFFIX)
MAKE_KEYLIB  = ilib /nologo $(PIOLOGIE_OBJS) /out:piologie.$(LIB_SUFFIX)
LINK         = $(CC) $@.$(OBJ) piologie.$(LIB_SUFFIX)
LINK_TEST    = $(CC) test.$(OBJ) $(SUPPLEMENT_OBJS) piologie.$(LIB_SUFFIX)
SUFFIX       = cpp
◇
```

D.9.11 KAI C++

"config.kai" 447c ≡

```
#
# KAI C++ 3.3.3 (Windows NT):
#
CC           = KCC
CFLAGS       = -O3 -DSTATIC_VS_INLINE=static
CFLAGS_KEY   = -D_NEEDS_PIOLOGIE_KEY_
OBJ          = obj
```

```
LIB_SUFFIX   = lib
COMPILE_ONLY = $(CC) -c
OUTPUT       = -o $@
MAKE_LIB     = ar cr $@ $(ALL_OBJS); ranlib $@
MAKE_KEYLIB  = ar cr piologie.$(LIB_SUFFIX) $(PIOLOGIE_OBJS); ranlib piologie.$(LIB_SUFFIX)
LINK         = $(CC) $(CFLAGS) $@.$(OBJ) nmbrthry.$(OBJ) piologie.$(LIB_SUFFIX) -o $@
LINK_TEST    = $(CC) $(CFLAGS) test.$(OBJ) $(SUPPLEMENT_OBJS) piologie.$(LIB_SUFFIX) -o test
SUFFIX       = cpp
◇
```

D.9.12 Microsoft Visual C++

"config.i386_vc" 448a ≡

```
#
# Pentium, Microsoft Visual C++ 6.0:
#
CC           = cl
CFLAGS       = /nologo /ML /W3 /GX /O2
CFLAGS_KEY   = /D_NEEDS_PIOLOGIE_KEY_
OBJ          = obj
LIB_SUFFIX   = lib
COMPILE_ONLY = $(CC) /c
OUTPUT       =
MAKE_LIB     = link -lib /nologo $(ALL_OBJS) /out:$@
MAKE_KEYLIB  = link -lib /nologo $(PIOLOGIE_OBJS) /out:piologie.$(LIB_SUFFIX)
LINK         = link /nologo $@ piologie.$(LIB_SUFFIX)
LINK_TEST    = link /nologo test.$(OBJ) $(SUPPLEMENT_OBJS) piologie.$(LIB_SUFFIX)
SUFFIX       = cpp
◇
```

D.9.13 SGI C++

"config.sgi" 448b ≡

```
#
# MIPS R4000, SGI MIPSpro C++ 7.1:
#
CC           = CC
CFLAGS       = -O2 -mips4 -D_Old_STD_
CFLAGS_KEY   = -D_NEEDS_PIOLOGIE_KEY_
OBJ          = o
LIB_SUFFIX   = a
COMPILE_ONLY = $(CC) -c
OUTPUT       = -o $@
MAKE_LIB     = ar cr $@ $(ALL_OBJS)
MAKE_KEYLIB  = ar cr piologie.$(LIB_SUFFIX) $(PIOLOGIE_OBJS)
LINK         = $(CC) $(CFLAGS) $@.$(OBJ) piologie.$(LIB_SUFFIX) -o $@
LINK_TEST    = $(CC) $(CFLAGS) test.$(OBJ) $(SUPPLEMENT_OBJS) piologie.$(LIB_SUFFIX) -o test
SUFFIX       = cpp
◇
```

"config.sgi_8000" 448c ≡

```
#
# MIPS R8000, SGI MIPSpro C++ 7.1:
#
CC           = CC
```

```
CFLAGS       = -O2 -mips4 -64 -D_Old_STD_
CFLAGS_KEY   = -D_NEEDS_PIOLOGIE_KEY_
OBJ          = o
LIB_SUFFIX   = a
COMPILE_ONLY = $(CC) -c
OUTPUT       = -o $@
MAKE_LIB     = ar cr $@ $(ALL_OBJS)
MAKE_KEYLIB  = ar cr piologie.$(LIB_SUFFIX) $(PIOLOGIE_OBJS)
LINK         = $(CC) $(CFLAGS) $@.$(OBJ) piologie.$(LIB_SUFFIX) -o $@
LINK_TEST    = $(CC) $(CFLAGS) test.$(OBJ) $(SUPPLEMENT_OBJS) piologie.$(LIB_SUFFIX) -o test
SUFFIX       = cpp
◇
```

D.9.14 SUN WorkShop C++

"config.sun" 449a ≡

```
#
# SUN WorkShop C++ 4.2:
#
CC           = CC
CFLAGS       = -fast -O5 -native -DDEFINE_VS_CONST -D_Old_STD_
CFLAGS_KEY   = -D_NEEDS_PIOLOGIE_KEY_
OBJ          = o
LIB_SUFFIX   = a
COMPILE_ONLY = $(CC) -c
OUTPUT       = -o $@
MAKE_LIB     = ar cr $@ $(ALL_OBJS); ranlib $@
MAKE_KEYLIB  = ar cr piologie.$(LIB_SUFFIX) $(PIOLOGIE_OBJS); ranlib piologie.$(LIB_SUFFIX)
LINK         = $(CC) $(CFLAGS) $@.$(OBJ) piologie.$(LIB_SUFFIX) -o $@
LINK_TEST    = $(CC) $(CFLAGS) test.$(OBJ) $(SUPPLEMENT_OBJS) piologie.$(LIB_SUFFIX) -o test
SUFFIX       = cpp
◇
```

D.9.15 Watcom C++

"config.i386_wc" 449b ≡

```
#
# Pentium, Watcom C++ 11.0:
#
CC           = wpp386
CFLAGS       = /5r /oneatx /zp4 /D_Old_STD_
CFLAGS_KEY   = /D_NEEDS_PIOLOGIE_KEY_
OBJ          = obj
LIB_SUFFIX   = lib
COMPILE_ONLY = $(CC)
OUTPUT       =
MAKE_LIB     = wlib -b -c -n -q $@ +$(ALL_OBJS)
MAKE_KEYLIB  = wlib -b -c -n -q piologie.$(LIB_SUFFIX) +$(PIOLOGIE_OBJS)
LINK         = wlink f $@ l piologie.$(LIB_SUFFIX)
LINK_TEST    = wlink f test.$(OBJ) f nmbrthry.$(OBJ) f pi.$(OBJ) l piologie.$(LIB_SUFFIX)
SUFFIX       = cpp
◇
```

Anhang E

Formeln zur π-Berechnung

$$\frac{\pi}{4} = \lim_{n\to\infty} \frac{2^{4n}(n!)^4}{2n((2n)!)^2} = \prod_{n\geq 1} \frac{2n^2}{4n^2-1} \qquad \text{(Wallis, 1665)}$$

$$\pi = 2i \log \frac{1-i}{1+i} \qquad \text{(Formule de Fagnano)}$$

$$\pi = 2 \cdot \frac{2}{\sqrt{2}} \cdot \frac{2}{\sqrt{2+\sqrt{2}}} \cdot \frac{2}{\sqrt{2+\sqrt{2+\sqrt{2}}}} \cdots \qquad \text{(Viète)}$$

$$\pi = \sum_{n=0}^{\infty} \frac{1}{16^n} \left(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6} \right) \qquad \text{(Bailey, Borwein, Plouffe, 1995)}$$

E.1 Kettenbrüche

$$\frac{\pi}{4} = \cfrac{1}{1 + \cfrac{1^2}{2 + \cfrac{3^2}{2 + \cfrac{5^2}{2 + \cfrac{7^2}{2 + \cdots}}}}} \qquad \text{(Lord Brouncker, 1620-1684)}$$

$$\frac{\pi}{2} = 1 - \cfrac{1}{3 - \cfrac{2 \cdot 3}{1 - \cfrac{1 \cdot 2}{3 - \cfrac{4 \cdot 5}{1 - \cfrac{3 \cdot 4}{3 - \cfrac{6 \cdot 7}{1 - \cfrac{5 \cdot 6}{3 - \cdots}}}}}}} \qquad \text{(Stern, 1833)}$$

E.2 Arcustangens-Reihen

$$\frac{\pi}{4} = \arctan(1) \qquad \text{(Leibniz, 1673)}$$

$$\frac{\pi}{4} = \arctan\left(\tfrac{1}{2}\right) + \arctan\left(\tfrac{1}{3}\right) \qquad \text{(Euler, 1738)}$$

$$\frac{\pi}{4} = \arctan\left(\tfrac{1}{2}\right) + \arctan\left(\tfrac{1}{5}\right) + \arctan\left(\tfrac{1}{8}\right) \qquad \text{(Strassnitzky, Daze, 1840)}$$

$$\frac{\pi}{4} = 2\arctan\left(\tfrac{1}{2}\right) - \arctan\left(\tfrac{1}{7}\right) \qquad \text{(Hermann, 1706)}$$

$\frac{\pi}{4} = 2\arctan\left(\frac{1}{3}\right) + \arctan\left(\frac{1}{7}\right)$ (Clausen, Hutton, 1776)

$\frac{\pi}{4} = 3\arctan\left(\frac{1}{4}\right) + \arctan\left(\frac{1}{20}\right) + \arctan\left(\frac{1}{1985}\right)$ (Störmer, 1896)

$\frac{\pi}{4} = 4\arctan\left(\frac{1}{5}\right) - \arctan\left(\frac{1}{239}\right)$ (Machin, 1706)

$\frac{\pi}{4} = 5\arctan\left(\frac{1}{7}\right) + 2\arctan\left(\frac{3}{79}\right)$ (Hutton, Euler, 1775)

$\frac{\pi}{4} = 4\arctan\left(\frac{1}{5}\right) - \arctan\left(\frac{1}{70}\right) + \arctan\left(\frac{1}{99}\right)$ (Euler, Rutherford)

$\frac{\pi}{4} = 6\arctan\left(\frac{1}{8}\right) + 2\arctan\left(\frac{1}{57}\right) + \arctan\left(\frac{1}{239}\right)$ (Störmer, 1896)

$\frac{\pi}{4} = 8\arctan\left(\frac{1}{10}\right) - \arctan\left(\frac{1}{239}\right) - 4\arctan\left(\frac{1}{515}\right)$ (Meissel, Klingenstierna)

$\frac{\pi}{4} = 12\arctan\left(\frac{1}{18}\right) + 8\arctan\left(\frac{1}{57}\right) - 5\arctan\left(\frac{1}{239}\right)$ (Gauß)

$\frac{\pi}{4} = 12\arctan\left(\frac{1}{18}\right) + 3\arctan\left(\frac{1}{70}\right) + 5\arctan\left(\frac{1}{99}\right) + 8\arctan\left(\frac{1}{307}\right)$ (Bennett)

$\frac{\pi}{4} = 22\arctan\left(\frac{1}{28}\right) + 2\arctan\left(\frac{1}{443}\right) - 5\arctan\left(\frac{1}{1393}\right) - 10\arctan\left(\frac{1}{11018}\right)$ (Escott)

$\frac{\pi}{4} = 12\arctan\left(\frac{1}{38}\right) + 20\arctan\left(\frac{1}{57}\right) + 7\arctan\left(\frac{1}{239}\right) + 24\arctan\left(\frac{1}{268}\right)$ (Gauß)

$\frac{\pi}{4} = 44\arctan\left(\frac{1}{57}\right) + 7\arctan\left(\frac{1}{239}\right) - 12\arctan\left(\frac{1}{682}\right) + 24\arctan\left(\frac{1}{12943}\right)$ (Wrench)

$\frac{\pi}{4} = 2805\arctan\left(\frac{1}{5257}\right) - 398\arctan\left(\frac{1}{9466}\right) + 1950\arctan\left(\frac{1}{12943}\right)$
$+1850\arctan\left(\frac{1}{34208}\right) + 2021\arctan\left(\frac{1}{44179}\right) + 2097\arctan\left(\frac{1}{85353}\right)$
$+1484\arctan\left(\frac{1}{114669}\right) + 1389\arctan\left(\frac{1}{330182}\right) + 808\arctan\left(\frac{1}{485298}\right)$ (Gauß)

E.3 Ramanujan-Reihen

$$\frac{1}{\pi} = \frac{\sqrt{8}}{9}\sum_{n=0}^{\infty}\frac{(4n)!}{(n!)^4}\frac{1+10n}{144^{2n}}$$

$$\frac{1}{\pi} = \frac{\sqrt{27}}{49}\sum_{n=0}^{\infty}\frac{(4n)!}{(n!)^4}\frac{3+40n}{784^{2n}}$$

$$\frac{1}{\pi} = \frac{\sqrt{11}}{198}\sum_{n=0}^{\infty}\frac{(4n)!}{(n!)^4}\frac{19+280n}{1584^{2n}}$$

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801}\sum_{n=0}^{\infty}\frac{(4n)!}{(n!)^4}\frac{1103+26390n}{396^{4n}}$$

$$\frac{1}{\pi} = \frac{1}{\sqrt{12}}\sum_{n=0}^{\infty}\frac{(4n)!}{(n!)^4}\frac{1+8n}{2304^{n}}$$

$$\frac{1}{\pi} = \frac{1}{3528}\sum_{n=0}^{\infty}(-1)^n\frac{(4n)!}{(n!)^4}\frac{1123+21460n}{14112^{2n}}$$

$$\frac{1}{\pi} = \frac{\sqrt{5}}{288}\sum_{n=0}^{\infty}(-1)^n\frac{(4n)!}{(n!)^4}\frac{41+644n}{6635520^{n}}$$

$$\frac{1}{\pi} = \frac{1}{72}\sum_{n=0}^{\infty}(-1)^n\frac{(4n)!}{(n!)^4}\frac{23+260n}{288^{2n}}$$

$$\frac{1}{\pi} = \frac{1}{8}\sum_{n=0}^{\infty}(-1)^n\frac{(4n)!}{(n!)^4}\frac{3+20n}{32^{2n}}$$

$$\frac{1}{\pi} = \frac{2\sqrt{3}}{5\sqrt{5}}\sum_{n=0}^{\infty}\frac{(6n)!}{(3n)!(n!)^3}\frac{1+11n}{54000^n}$$

$$\frac{1}{\pi} = \frac{2}{15\sqrt{3}}\sum_{n=0}^{\infty}\frac{(2n)!(3n)!}{(n!)^5}\frac{4+33n}{15^{3n}}$$

$$\frac{1}{\pi} = \frac{4}{27}\sum_{n=0}^{\infty}\frac{(2n)!(3n)!}{(n!)^5}\frac{2+15n}{1458^n}$$

$$\frac{1}{\pi} = \frac{1}{4}\sum_{n=0}^{\infty}\frac{((2n)!)^3}{(n!)^6}\frac{1+6n}{2^{8n}}$$

$$\frac{1}{\pi} = \frac{1}{16}\sum_{n=0}^{\infty}\frac{((2n)!)^3}{(n!)^6}\frac{5+42n}{2^{12n}}$$

$$\frac{1}{\pi} = \frac{1}{53360\sqrt{640320}}\sum_{n=0}^{\infty}(-1)^n\frac{(6n)!}{(3n)!(n!)^3}\frac{13591409+545140134n}{640320^{3n}} \qquad \text{(Chudnovsky, 1989)}$$

E.4 Beziehungen zu den Fibonacci-Zahlen

$$\frac{\pi}{4} = \sum_{n=1}^{\infty}\arctan\left(\frac{1}{F_{2n+1}}\right)$$

$$\pi = \lim_{n\to\infty}\sqrt{\frac{6\log\prod_{i=1}^{n}F_i}{\log\operatorname{lcm}(F_i)_{i=1}^{n}}} \qquad \text{(Matiyasevich, 1985)}$$

E.5 Beziehungen zu der Riemanschen Zetafunktion

$$\frac{\pi^2}{6} = \zeta(2) \qquad \text{(Euler, 1736)}$$

$$\frac{(2\pi)^n}{n!} = \frac{2\zeta(n)}{|B_n|}, \qquad \text{mit Bernoulli-Zahlen } B_n \text{ und } n\in\mathbb{N},\ 2\mid n \qquad \text{(Euler)}$$

$$\pi = \sum_{n\geq 1}\frac{3^n-1}{4^n}\zeta(n+1) \qquad \text{(Flajolet, Vardi, 1996)}$$

Literaturverzeichnis

[1] T. Amdeberhan und D. Zeilberger: Hypergeometric Series Acceleration via the WZ Method, Electronic Journal of Combinatorics (Wilf Festschrift Volume) 4 (1997).

[2] A. V. Aho, J. E. Hopcroft, J. D. Ullman: The Design and Analysis of Computer Algorithmus, 1975.

[3] E. Bach, J. Shallit: Algorithmic Number Theory – Volume 1: Efficient Algorithms, 1996.

[4] D. H. Bailey: FFTs in External or Hierarchical Memory, Journal of Supercomputing 4 (1990), 23-35.

[5] A. Binstock, J. Rex: Practical Algorithms for Programmers, 1995.

[6] J. M. Borwein, P. B. Borwein: Pi and the AGM, 1987.

[7] G. Brassard, P. Bratley: Algorithmics - Theory and Practice, 1993.

[8] D. M. Bressoud: Factorization and Primality Testing, 1989.

[9] J. Brillhart, J. L. Selfridge Some factorizations of $2^n \pm 1$ and related results, Mathematics of Computation 21 (1967), 87-96.

[10] H. Cohen, C. Batut, D. Bernardi, M. Olivier: GP/PARI calculator, Version 1.39.03, available by anonymous ftp from `ftp://megrez.math.u-bordeaux.fr`, 1995.

[11] G. E. Collins et alliis: SACLIB User's Guide, Version 1.0, available by anonymous ftp from `ftp://ftp.risc.uni-linz.ac.at/pub/saclib/`, March 1993.

[12] G. E. Collins, R. Loos: Specification and Index of SAC-2 Algorithms, Technical Report WSI-90-4, Wilhelm-Schickard-Institut für Informatik Tübingen, 1990.

[13] J. W. Cooley, J. W. Tukey: An Algorithm for the Machine Calculation of Complex Fourier Series, Mathematics of Computation 19 (1965), 297-301.

[14] J. D. Dixon: Asymptotically Fast Factorization of Integers, Mathematics of Computation 36 (1981), 255-260.

[15] M. A. Ellis, B. Stroustrup: The Annotated C++ Reference Manual, 1990.

[16] M. R. Fellows, N. Koblitz: Self-Witnessing Polynomial-Time Complexity and Prime Factorization, Designs, Codes and Cryptography 2 (1992), 231-235.

[17] O. Forster: Algorithmische Zahlentheorie, 1996.

[18] R. L. Graham, D. E. Knuth, O. Patashnik: Concrete Mathematics, A Foundation for Computer Science, Second Edition, 1989.

[19] T. Granlund: GNU MP, The GNU Multiple Precision Arithmetic Library, Edition 1.3.2, available by anonymous ftp from `ftp://prep.ai.mit.edu/pub/gnu/`, May 1993.

[20] T. Granlund: GNU MP, The GNU Multiple Precision Arithmetic Library, Edition 2.0.2, available by anonymous ftp from `ftp://prep.ai.mit.edu/pub/gnu/`, June 1996.

[21] B. Haible, T. Papanikolaou: Fast multiprecision evaluation of series of rational numbers, Technical Report TI-97-7, Darmstadt University of Technology, April 1997.

[22] B. Haible: CLN, A Class Library for Numbers, available by anonymous ftp from `ftp://ma2s2.mathematik.uni-karlsruhe.de/pub/gnu/`, 1996.

[23] D. Herrmann: Algorithmen Arbeitsbuch, 1992.

[24] M. Hüttenhofer, M. Lesch, N. Peyerimhoff: Mathematik in Anwendung mit C++, 1994.

[25] G. Jaeschke: On Strong Pseudoprimes to Several Bases, Mathematics of Computation 61 (1993), 915–926.

[26] M. Jeger: Computer Streifzüge, 1986.

[27] D. E. Knuth: The Art of Computer Programming, Seminumerical Algorithms, Third Edition, 1998.

[28] S. Konyagin, C. Pomerance: On Primes Recognizable in Deterministic Polynomial Time, in R. L. Graham, J. Nešetřil (Eds.): The Mathematics of Paul Erdös I, 1997.

[29] R. S. Lehman: Factoring Large Integers, Mathematics of Computation 28 (1974), 637-646.

[30] D. H. Lehmer: On Arccotangent Realtions for π, American Mathematical Monthly 45 (1938), 657-664.

[31] A. Lenstra: Long Integer Package, Version 1.1, available by anonymous ftp from `ftp://ftp.ox.ac.uk/pub/math/freelip/`, July 1997.

[32] LiDIA-Group: LiDIA Manual, Version 1.3, Fachbereich 14, Universität des Saarlandes, available by anonymous ftp from `ftp://ftp.informatik.th-darmstadt.de/pub/TI/systems/LiDIA/`, February 1997.

[33] S. B. Lippman: C++ Primer, Second Edition, 1995.

[34] X. Luo: A Practical Sieve Algorithm for Finding Prime Numbers, Communications of the ACM 32 (1989), 344-346.

[35] Ch. Meinel: Effiziente Algorithmen, 1991.

[36] S. Meyers: More Effective C++, 35 New Ways to Improve Your Programs ans Designs, 1996.

[37] G. L. Miller: Riemann's Hypothesis and Tests for Primality, Journal of Computer and System Sciences 13 (1976), 300-317.

[38] M. A. Morrison, J. Brillhart: A Method of Factoring and the Factorization of F_7, Mathematics of Computation 29 (1975), 183-205.

[39] D. R. Musser, A. Saini: STL Tutorial and Reference Guide, C++ Programming with the Standard Template Library, 1996.

[40] W. H. Press, S. A. Teukolsky, W. T. Vetterling, B. P. Flannery: Numerical Recipes in C, Second Edition, 1992.

[41] H. Riesel: Prime Numbers and Computer Methods for Faktorization, 1985.

[42] A. Schönhage, A. F. W. Grotefeld, E. Vetter: Fast Algorithms, 1994.

[43] S. Schupp: Generic programming – SuchThat one can build an algebraic library, Dissertation Universität Tübingen, 1996.

[44] R. Sedgewick: Algorithmen, 1991.

[45] J. O. Shallit, J. Sorenson: A Binary Algorithm for the Jacobi Symbol, ACM SIGSAM Bulletin 27 (1993), 4-11.

[46] R. M. Stallman: Using and Porting GNU CC, 1993.

[47] J. Stein: Computational Problems Associated with Racah Algebra, Journal of Computational Physics 1 (1967), 397-405.

[48] A. A. Stepanov, M. Lee: The Standard Template Library, Technical Report HPL-94-34, 1994.

[49] J. Stoer: Numerische Mathematik, 7. Auflage, 1994.

[50] B. Stroustrup: The C++ Programming Language, Third Edition, 1997.

[51] M. Tommila: `apfloat`, A C++ High Performance Arbitrary Precision Arithmetic Package, Version 1.40, available at `http://www.hut.fi/~mtommila/apfloat/`, July 1997.

[52] K. Weber: The Accelerated Integer GCD Algorithm, ACM Transactions on Mathematical Software 21 (1995), 111-122.

[53] J. Wegener: Effiziente Algorithmen für grundlegende Funktionen, 1989.

Index

Dateien

Makros

⟨ squaring of a Natural 108 ⟩ Referenced in 409.
⟨ squaring of a Rational 257b ⟩ Referenced in 427.
⟨ standard system include file <limits.h> 3c ⟩ Referenced in 395.
⟨ standard system include file <stdlib.h> 4c ⟩ Referenced in 395.
⟨ static variables for memory management of Naturals 20b ⟩ Referenced in 402b.
⟨ strong pseudoprime test 289a ⟩ Referenced in 432b.
⟨ subtraction of a Natural with a Digit 61b ⟩ Referenced in 409.
⟨ subtraction of a Natural with an Integer 206a ⟩ Referenced in 426.
⟨ subtraction of an Integer with a Natural 205b ⟩ Referenced in 426.
⟨ subtraction of an Integer with a SignDigit 206b ⟩ Referenced in 426.
⟨ subtraction of two Integers 205a ⟩ Referenced in 426.
⟨ subtraction of two Naturals 59 ⟩ Referenced in 409.
⟨ subtraction of two Rationals 253b ⟩ Referenced in 431.
⟨ successive division for greater factors 294 ⟩ Referenced in 296.
⟨ successive division for less factors 293 ⟩ Referenced in 296.
⟨ testing size of Digit and SignDigit 186a ⟩ Referenced in 398b.
⟨ testing the bits for low ending 68a ⟩ Referenced in 398b.
⟨ tests a bit in a Natural 85b ⟩ Referenced in 409.
⟨ tests a bit in an Integer 223a ⟩ Referenced in 426.
⟨ the function swap for Digits 5 ⟩ Referenced in 397d.
⟨ trade off point for the Natural to string conversion 42a ⟩ Referenced in 409.
⟨ trade off point for the string to Natural conversion 45a ⟩ Referenced in 409.
⟨ trade off points for the FFT algorithm 112 ⟩ Referenced in 409.
⟨ trade off points for the Karatsuba algorithm 103b ⟩ Referenced in 409.
⟨ trade off points for the newton-iteration of the division algorithm 165a ⟩ Referenced in 409.
⟨ trade off points for the newton-iteration of the square root algorithm 179a ⟩ Referenced in 409.
⟨ transpose and copy a matrix 136 ⟩ Referenced in 409.
⟨ type declaration of Digit 3a ⟩ Referenced in 2.
⟨ type declaration of SignDigit 185 ⟩ Referenced in 2.
⟨ units of an Integer 189c ⟩ Referenced in 419.
⟨ variables for representation of a Natural 16 ⟩ Referenced in 402b.

Funktionsnamen