2 Die Grundrechnungs-Arten

Nachdem wir im vorigen Paragraphen gesehen haben, dass die sich aus den Peano-Axiomen ergebenden rekursiven Algorithmen für Addition, Multiplikation und Potenzierung sehr ineffizient sind, besprechen wir jetzt bessere Algorithmen, die mit der Binär-Darstellung ganzer Zahlen arbeiten. Bemerkenswert ist dabei der Potenzierungs-Algorithmus. Um eine Zahl in die n-te Potenz zu erheben, sind nicht, wie beim naiven Verfahren, $n-1$ Multiplikationen nötig, sondern höchstens $2k$, wobei k die Anzahl der Binär-Stellen von n ist.

Boolesche Operationen

Die möglichen Ziffern 0 und 1 in der Binär-Darstellung natürlicher Zahlen lassen sich auch als die logischen Konstanten falsch (0) und wahr (1) interpretieren. Die wichtigsten logischen (oder booleschen) Operationen auf der Menge $\{0,1\}$ sind die *Verneinung* (`not`, in Zeichen $\neg$) und die Verknüpfungen *Oder* (`or`, in Zeichen $\vee$), *Und* (`and`, in Zeichen $\wedge$) und *Exklusives Oder* (`xor`, in Zeichen $\oplus$). Sie sind wie folgt definiert:

x	$\neg x$
0	1
1	0

$\vee$	0	1
0	0	1
1	1	1

$\wedge$	0	1
0	0	0
1	0	1

$\oplus$	0	1
0	0	1
1	1	0

Alle diese Operationen lassen sich einfach durch die heutige Computer-Hardware realisieren.

Addition

Die Addition einstelliger Binärzahlen $a, b \in \{0,1\}$ lässt sich mit den booleschen Operationen so ausdrücken:

$$a + b = c + 2d, \quad \text{mit} \quad c = a \oplus b,\ d = a \wedge b$$

Dies kann man leicht durch Nachprüfen aller 4 möglichen Fälle für (a, b) bestätigen. Seien jetzt

$$x = \sum_{\nu=0}^{n} a_\nu\, 2^\nu, \quad y = \sum_{\nu=0}^{m} b_\nu\, 2^\nu, \quad a_\nu, b_\nu \in \{0,1\}$$

beliebig große natürliche Zahlen in Binär-Darstellung. Wir können annehmen, dass $m = n$, indem wir nötigenfalls führende Nullen ergänzen. Dann gilt

$$x + y = x_1 + y_1, \quad x_1 = \sum_{\nu=0}^{n} c_\nu\, 2^\nu, \quad y_1 = \sum_{\nu=0}^{n} d_\nu\, 2^{\nu+1}$$

O. Forster, *Algorithmische Zahlentheorie*, DOI 10.1007/978-3-658-06540-9_2

mit

$$c_\nu = a_\nu \oplus b_\nu, \quad d_\nu = a_\nu \wedge b_\nu.$$

Zu beachten ist, dass alle c_ν parallel berechnet werden können, da keine gegenseitigen Abhängigkeiten bestehen. Das Gleiche gilt für die d_ν. Auf die Summe $x_1 + y_1$ können wir die gleiche Konstruktion anwenden, $x_1 + y_1 = x_2 + y_2$. So fortfahrend erhalten wir eine Gleichungs-Kette

$$x + y = x_1 + y_1 = x_2 + y_2 = \dots$$

Nach endlich vielen Schritten muss sich $y_k = 0$ ergeben, denn für alle $i \geqslant 1$ gilt $y_i \leqslant x + y \leqslant 2^{n+1}$ und man zeigt durch Induktion, dass die Binär-Darstellung von y_i die Gestalt

$$y_i = \sum_{\nu \geqslant i} y_{i,\nu}\, 2^\nu$$

hat. Wenn $y_k = 0$, folgt $x + y = x_k$, also haben wir dann die Summe von x und y berechnet. Dies ist der von Neumann'sche Additions-Algorithmus. Er lässt sich leicht in Aribas implementieren.

```
function add(x,y: integer): integer;
var
    x0: integer;
begin
    while y > 0 do
        x0 := x;
        x := bit_xor(x,y);
        y := bit_shift(bit_and(x0,y),1);
    end;
    return x;
end.
```

Die eingebaute Aribas-Funktion `bit_xor(x,y)` berechnet das bitweise `xor` ihrer Argumente, d.h. für $x = \sum x_\nu\, 2^\nu$ und $y = \sum y_\nu\, 2^\nu$ ist das Ergebnis $\sum (x_\nu \oplus y_\nu)\, 2^\nu$. Analoges gilt für die Funktion `bit_and`. Außerdem wird noch die Funktion `bit_shift(x,k)` benutzt. Ist $x = \sum_{\nu=0}^{n} x_\nu\, 2^\nu$ und $k \geqslant 0$, so liefert `bit_shift` das Resultat $\sum_{\nu=0}^{n} x_\nu\, 2^{\nu+k}$. (Mit negativem Argument $k = -\ell < 0$ entsteht $\sum_{\nu=\ell}^{n} x_\nu\, 2^{\nu-\ell}$.)

Um die Arbeitsweise des Algorithmus genauer verfolgen zu können, schreiben wir noch eine Version von `add`, welche die Zwischenergebnisse ausdruckt.

```
function add_verbose(x,y: integer): integer;
var
    x0, N: integer;
begin
    N := max(bit_length(x),bit_length(y)) + 1;
```

```
    while y > 0 do
        writeln(x:base(2):digits(N));
        writeln(y:base(2):digits(N));
        writeln();
        x0 := x;
        x := bit_xor(x,y);
        y := bit_shift(bit_and(x0,y),1);
    end;
    writeln(x:base(2):digits(N));
    return x;
end.
```

In ARIBAS hat die Funktion `writeln` gegenüber PASCAL erweiterte Möglichkeiten der Formatierung. Durch `writeln(x:base(2):digits(N))` wird die Zahl x in Binär-Darstellung (Basis 2) mit insgesamt N Ziffern geschrieben (wobei evtl. führende Nullen ergänzt werden). N ist hier um eins größer als das Maximum der Binärstellen-Anzahl von x und y gewählt, so dass $x + y$ höchstens N Binärstellen hat. Ein kleines Testbeispiel:

```
==> add_verbose(1996,873).
0111_11001100
0011_01101001

0100_10100101
0110_10010000

0010_00110101
1001_00000000

1011_00110101
-: 2869
```

In diesem Beispiel hat sich schon nach 3 Schleifen-Durchgängen das Endergebnis eingestellt, obwohl die Argumente 11 bzw. 10 Binärstellen haben. Es lässt sich zeigen, dass der durchschnittliche Wert der benötigten Zyklen bei der Addition n-stelliger Binärzahlen etwa gleich $\log(n)$ ist. Im ungünstigsten Fall bei der Addition von $2^n - 1$ und 1 braucht man aber $n + 1$ Zyklen. Für eine eingehendere Diskussion dieses und vieler anderer Algorithmen verweisen wir auf [Weg].

Die Subtraktion natürlicher Zahlen kann man mit Hilfe der sog. *Zweierkomplement-Darstellung* auf die Addition zurückführen. Seien zwei ganze Zahlen $x \geqslant y \geqslant 0$ gegeben. Man wähle ein n, so dass $2^n > x$. Statt $x - y$ wird $z := 2^n + (x - y)$ berechnet. Daraus kann man $x - y$ einfach durch Streichen des Bits mit der Wertigkeit 2^n gewinnen. Nun ist

$$z = 2^n + (x - y) = x + (2^n - y),$$

es ist also das "Zweierkomplement" $2^n - y$ zu bilden und eine Addition durchzuführen. Da $y < 2^n$, lässt sich y darstellen als $y = \sum_{\nu=0}^{n-1} y_\nu\, 2^\nu$, $y_\nu \in \{0, 1\}$. Andrerseits gilt $2^n - 1 = \sum_{\nu=0}^{n-1} 2^\nu$, woraus folgt

$$y' := (2^n - 1) - y = \sum_{\nu=0}^{n-1} (\neg y_\nu)\, 2^\nu$$

Das sog. *Einerkomplement* y' entsteht also aus y durch Umkehrung aller Bits und für das Zweierkomplement folgt $2^n - y = y' + 1$.

Die russische Bauernregel der Multiplikation

Ein interessanter Algorithmus zur Multiplikation natürlicher Zahlen, der früher in Russland benutzt worden ist, ist unter dem Namen russische Bauernregel bekannt (er ist aber auch schon im alten Ägypten verwendet worden). Der Algorithmus führt die Multiplikation auf Verdoppeln und Halbieren sowie Additionen zurück. Um z.B. $x = 83$ mit $y = 57$ zu multiplizieren, wird eine Tabelle angefertigt, in deren erster Zeile die Zahlen x und y stehen. In der nächsten Zeile wird der x-Wert verdoppelt und der y-Wert halbiert (dabei wird von der Hälfte nur der ganzzahlige Anteil genommen). Diese Prozedur wird solange wiederholt, bis man beim y-Wert 1 angelangt ist.

+	83	57
	166	28
	332	14
+	664	7
+	1328	3
+	2656	1
	4731	

Die Zeilen, deren y-Wert ungerade ist (wo also die Halbierung nicht aufgeht), werden markiert und am Ende werden die x-Werte aller markierten Zeilen addiert. Das Resultat ist das gesuchte Produkt. Diese Multiplikations-Regel lässt sich leicht durch eine ARIBAS-Funktion realisieren.

```
function mult(x,y: integer): integer;
var
    z: integer;
begin
    z := 0;
    while y > 0 do
        if odd(y) then z := z + x; end;
        x := bit_shift(x,1);
        y := bit_shift(y,-1);
    end;
    return z;
end.
```

Man beachte, dass sich Verdopplung und Halbierung natürlicher Zahlen in Binär-Darstellung durch `bit_shift`'s um eine Stelle nach links bzw. rechts darstellen lassen. Die Anzahl der Durchlaufungen der `while`-Schleife ist gleich der Anzahl der Binärstellen von y. Um uns von der Korrektheit des Algorithmus zu überzeugen, zeigen wir, dass $xy + z$ eine Invariante der `while`-Schleife ist. Seien x_a, y_a, z_a die Werte von x, y, z zu Beginn eines Schleifen-Durchlaufs und x_e, y_e, z_e die entsprechenden Werte am Ende eines Durchlaufs. Ist y_a gerade, so gilt $x_e = 2x_a$, $y_e = y_a/2$ und $z_e = z_a$. Ist aber y_a ungerade, so ist $x_e = 2x_a$, $y_e = (y_a - 1)/2$ und $z_e = z_a + x_a$. In beiden Fällen erhält man $x_a y_a + z_a = x_e y_e + z_e$. Da vor dem 1. Schleifen-Duchlauf $z_a = 0$ ist und nach dem letzten Schleifen-Durchlauf $y_e = 0$, also auch $x_e y_e = 0$ gilt, folgt, dass am Schluss die Variable z das gesuchte Produkt enthält.

Betrachtet man den Multiplikations-Algorithmus genauer, so sieht man, dass er nichts anderes als die Schulmethode der Multiplikation im Binärsystem ist. Er ist deshalb besonders einfach, da das kleine 1×1 im Binärsystem trivial ist. Wir werden später in §19 Multiplikations-Algorithmen kennenlernen, die für große Zahlen schneller als die Schulmethode sind.

Potenzierung

Die naive Methode zur Berechnung einer Potenz x^n besteht darin, durch wiederholte Multiplikation mit x sukzessive $x^2, x^3, \ldots, x^n$ zu berechnen. Dabei werden offenbar $n - 1$ Multiplikationen benötigt. Dass dies im Allgemeinen nicht optimal ist, wird am Beispiel x^{16} deutlich. Hier kann man durch wiederholtes Quadrieren der Reihe nach x^2, x^4, x^8 und x^{16} berechnen, man kommt also schon mit 4 statt 15 Multiplikationen aus. Auch wenn der Exponent n keine Zweierpotenz ist, kann man durch Quadrieren Multiplikationen einsparen. Sei

$$n = \sum_{i=0}^{\ell-1} b_i \, 2^i, \quad b_i \in \{0, 1\},\ b_{\ell-1} = 1$$

die Binär-Darstellung des Exponenten n und

$$n_k = \sum_{i \geqslant k} b_i \, 2^i.$$

Damit ist $n_{\ell-1} = 1$, $n_0 = n$ und

$$n_k = \begin{cases} 2n_{k+1}, & \text{falls } b_k = 0, \\ 2n_{k+1} + 1, & \text{falls } b_k = 1. \end{cases}$$

Mit $z_k := x^{n_k}$ gilt daher $z_{\ell-1} = x$, $z_0 = x^n$ und

$$z_k = \begin{cases} z_{k+1}^2, & \text{falls } b_k = 0, \\ z_{k+1}^2 x, & \text{falls } b_k = 1. \end{cases}$$

Damit kann man $z_0 = x^n$ aus $z_{\ell-1} = x$ in $\ell - 1$ Schritten berechnen, wobei in jedem Schritt höchstens zwei Multiplikationen nötig sind. Die folgende ARIBAS-Funktion `power` führt dies durch.

```
function power(x,n: integer): integer;
var
    k, pow: integer;
begin
    if n = 0 then return 1; end;
    pow := x;
    for k := bit_length(n)-2 to 0 by -1 do
        pow := pow * pow;
        if bit_test(n,k) then
            pow := pow * x;
        end;
    end;
    return pow;
end.
```

Darin ist $\ell := \texttt{bit\_length}(n)$ die Anzahl der Binärstellen von n und die `for`-Schleife wird der Reihe nach für $k = \ell-2, \ell-3, \ldots, 1, 0$ durchgeführt. Die Funktion `bit_test`(n,k) testet, ob $b_k = 1$ in der Binär-Darstellung $n = \sum_{i=0}^{\ell-1} b_i\, 2^i$ von n.

Z.B. ergibt die Berechnung von 3^{100}

```
==> power(3,100).
-: 515_37752_07320_11331_03646_11297_65621_27270_21075_22001
```

Die Potenzierung ist auch als eingebauter ARIBAS-Operator vorhanden, der (wie in FORTRAN) durch das Symbol `**` dargestellt wird; das obige Resultat hätte man also bequemer mit dem Befehl `3**100` erhalten.

AUFGABEN

2.1. Man beweise, dass die folgende ARIBAS-Funktion

```
function eucl_div(x,y: integer): array[2] of integer;
var
    quot, b: integer;
begin
    quot := 0; b := 1;
    while y < x do
        y := bit_shift(y,1);
        b := bit_shift(b,1);
    end;
    while b > 0 do
        if x >= y then
            x := x - y;
            quot := quot + b;
```

```
            end;
            y := bit_shift(y,-1);
            b := bit_shift(b,-1);
        end;
        return (quot, x);
    end.
```

mit ganzzahligen Argumenten $x \geqslant 0, y > 0$ ein Paar (q, r) ganzer Zahlen zurückgibt mit $x = qy + r, \quad 0 \leqslant r < y$.

2.2. Man beweise, dass die folgende ARIBAS-Funktion

```
function rt(a: integer): integer;
var
    x,y: integer;
begin
    x := a; y := 1;
    while x > y do
        x := (x+y) div 2;
        y := a div x;
    end;
    return x;
end.
```

mit ganzzahligem Argument $a \geqslant 0$ die größte ganze Zahl x mit $x^2 \leqslant a$ zurückgibt. Dabei ist `div` der ARIBAS-Operator für die ganzzahlige Division, d.h. a `div` b ist für $b > 0$ die größte ganze Zahl q mit $qb \leqslant a$.

2.3. Mit dem in diesem Paragraphen besprochenen Potenzierungs-Algorithmus braucht man zur Berechnung von x^{15} und x^{63} insgesamt 6 bzw. 10 Multiplikationen. Man zeige, dass man x^{15} schon mit 5 und x^{63} mit 8 Multiplikationen berechnen kann.

2.4. Man zeige, dass die Menge $\{0, 1\}$ mit den Verknüpfungen $\oplus$ als Addition und $\wedge$ als Multiplikation einen Körper bildet.

2.5. Man beweise: Für alle $x, y \in \{0, 1\}$ gilt

$$x \oplus y = (x \wedge \neg y) \vee (\neg x \wedge y) = (x \vee y) \wedge \neg(x \wedge y).$$

http://www.springer.com/978-3-658-06539-3

Algorithmische Zahlentheorie
Forster, O.
2015, VIII, 314 S. 7 Abb., Softcover
ISBN: 978-3-658-06539-3