

▶ Exponentiation: das Problem

- ▶ Gegeben:
(multiplikative) Halbgruppe $(H, *)$, Element $a \in H, n \in \mathbb{N}$
- ▶ Aufgabe: berechne das Element

$$a^{*n} = \underbrace{a * a * a * \dots * a}_n \in H$$

(schreiben ab jetzt a^n statt a^{*n})

- ▶ Hinweis: der wesentliche "Trick" funktioniert immer, wenn $*$ eine assoziative (aber nicht notwendig kommutative) Operation ist — also nicht nur für Zahlen, sondern z.B. auch für Polynome, Matrizen, Funktionen, ...

▶ Banale Idee: iterierte Multiplikation

▶ Algorithmus:

Require: $a \in H, n \in \mathbb{N}$

Ensure: $x = a^n$

if $n = 0$ **then**

 Return(e)

end if

$x \leftarrow a$

for $i = 1$ to $n - 1$ **do**

$x \leftarrow x * a$

end for

Return(x)

- ▶ Dies erfordert $n - 1$ Multiplikationen in H .

Bessere Idee:

- ▶ In jeder Halbgruppe $(H, *)$ kann man zur Berechnung der Exponentiation

$$H \times \mathbb{N} \rightarrow H : (a, n) \mapsto a^n$$

die Binärdarstellung des Exponenten n verwenden

- ▶ Rekursive Struktur:

$$a^n = \left\{ \begin{array}{ll} \left(a^{\frac{n}{2}}\right)^2 & \text{falls } n \text{ gerade} \\ a * \left(a^{\frac{n-1}{2}}\right)^2 & \text{falls } n \text{ ungerade} \end{array} \right\} = a^{n \bmod 2} * \left(a^{\frac{n}{2}}\right)^2$$

→ "schnelle" oder "binäre" Exponentiation, Exponentiation mittels "square-and-multiply"

- ▶ Man kann die Binärdarstellung von n von links nach rechts (LR) oder von rechts nach links (RL) abarbeiten

Notation:

- ▶ $(n)_2$: Binärdarstellung von n (ohne führende Nullen)
- ▶ $\ell(n)$: Länge der Binärdarstellung von $n = \lfloor \log(n) \rfloor + 1$
- ▶ $\nu(n) = \#_1(n)_2$: Anzahl der Einsen in $(n)_2$

Rekursionen:

- ▶ $\ell(2n) = \ell(n) + 1$
 $\ell(2n + 1) = \ell(n) + 1$
 $\ell(1) = 1$
- ▶ $\nu(2n) = \nu(n)$
 $\nu(2n + 1) = \nu(n) + 1$
 $\nu(1) = 1$

Beispiel $n = 155$ mit $(155)_2 = 10011011$, $\ell(155) = 8$, $\nu(155) = 5$

► LR-Methode

$(1)_2$	$= 1$	$a^1 = a$
$(2)_2$	$= 10$	$a^2 = (a^1)^2$
$(4)_2$	$= 100$	$a^4 = (a^2)^2$
$(9)_2$	$= 1001$	$a^9 = (a^4)^2 * a$
$(19)_2$	$= 10011$	$a^{19} = (a^9)^2 * a$
$(38)_2$	$= 100110$	$a^{38} = (a^{19})^2$
$(77)_2$	$= 1001101$	$a^{77} = (a^{38})^2 * a$
$(155)_2$	$= 10011011$	$a^{155} = (a^{77})^2 * a$

erfordert $7 = \ell(155) - 1$ Quadrierungen und $4 = \nu(155) - 1$ zusätzliche Multiplikationen, also insgesamt $\ell(155) + \nu(n) - 2$ Multiplikationen

► RL-Methode

$(1)_2$	$= 1$	$a^1 = a$
		$a^2 = (a^1)^2$
$(3)_2$	$= 11$	$a^3 = a^2 * a$
		$a^4 = (a^2)^2$
		$a^8 = (a^4)^2$
$(11)_2$	$= 1011$	$a^{11} = a^8 * a^3$
		$a^{16} = (a^8)^2$
$(27)_2$	$= 11011$	$a^{27} = a^{16} * a^{11}$
		$a^{32} = (a^{16})^2$
		$a^{64} = (a^{32})^2$
		$a^{128} = (a^{64})^2$
$(155)_2$	$= 10011011$	$a^{155} = a^{128} * a^{27}$

erfordert $7 = \ell(155) - 1$ Quadrierungen und $4 = \nu(155) - 1$ zusätzliche Multiplikationen, also insgesamt $\ell(155) + \nu(n) - 2$ Multiplikationen

Aufwand:

- $M(n)$: Anzahl der Multiplikationen (incl. Quadrierungen) in H zur Berechnung von a^n mittels LR- bzw. RL-Methode

- Rekursion:

$$\begin{aligned} M(2n) &= M(n) + 1 \\ M(2n+1) &= M(n) + 2 \end{aligned} \quad M(1) = 0$$

- Folgerung: $M(n) = \ell(n) + \nu(n) - 2$

- Beweis (Induktion)

$$M(1) = 0 = 1 + 1 - 2 = \ell(1) + \nu(1) - 2$$

$$M(2n) = M(n) + 1 = \ell(n) + \nu(n) - 1 = \ell(2n) + \nu(2n) - 2$$

$$M(2n+1) = M(n) + 2 = \ell(n) + \nu(n) = \ell(2n+1) + \nu(2n+1) - 2$$

- Also gilt insbesondere:

$$\lfloor \log n \rfloor \leq M(n) \leq 2 \lfloor \log n \rfloor \quad \text{d.h. } M(n) \in \Theta(\log n)$$

This binary method is easily justified by a consideration of the sequence of exponents in the calculation: If we reinterpret "S" as the operation of multiplying by 2 and "X" as the operation of adding 1, and if we start with 1 instead of x , the rule will lead to a computation of n because of the properties of the binary number system. The method is quite ancient; it appeared before 200 B.C. in Pingala's Hindu classic *Chandaḥ-sūtra* [see B. Datta and A. N. Singh, *History of Hindu Mathematics 2* (Lahore: Motilal Banarsi Das, 1935), 76]. There seem to be no other references to this method outside of India during the next 1000 years, but a clear discussion of how to compute 2^n efficiently for arbitrary n was given by al-Uqlīdisī of Damascus in A.D. 952; see *The Arithmetic of al-Uqlīdisī* by A. S. Saidan (Dordrecht: D. Reidel, 1975), 341–342, where the general ideas are illustrated for $n = 51$. See also al-Birūnī's *Chronology of Ancient Nations*, edited and translated by E. Sachau (London: 1879), 132–136; this eleventh-century Arabic work had great influence.

Abbildung: Knuth, TAOCP vol.2, ch. 4.6.3 (LR-Methode)

The S-and-X binary method for obtaining x^n requires no temporary storage except for x and the current partial result, so it is well suited for incorporation in the hardware of a binary computer. The method can also be readily programmed; but it requires that the binary representation of n be scanned from left to right. Computer programs generally prefer to go the other way, because the available operations of division by 2 and remainder mod 2 will deduce the binary representation from right to left. Therefore the following algorithm, based on a right-to-left scan of the number, is often more convenient:

Abbildung: Knuth, TAOCP vol.2, ch. 4.6.3 (LR vs. RL)

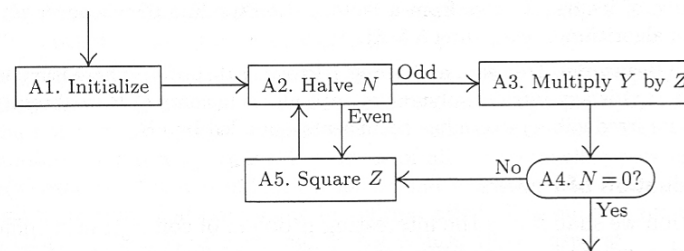


Fig. 13. Evaluation of x^n , based on a right-to-left scan of the binary notation for n .

Abbildung: Knuth, TAOCP vol.2, ch. 4.6.3 (RL)

Algorithm A (*Right-to-left binary method for exponentiation*). This algorithm evaluates x^n , where n is a positive integer. (Here x belongs to any algebraic system in which an associative multiplication, with identity element 1, has been defined.)

- A1.** [Initialize.] Set $N \leftarrow n$, $Y \leftarrow 1$, $Z \leftarrow x$.
- A2.** [Halve N .] (At this point, $x^n = YZ^N$.) Set $N \leftarrow \lfloor N/2 \rfloor$, and at the same time determine whether N was even or odd. If N was even, skip to step A5.
- A3.** [Multiply Y by Z .] Set $Y \leftarrow Z$ times Y .
- A4.** [$N = 0$?] If $N = 0$, the algorithm terminates, with Y as the answer.
- A5.** [Square Z .] Set $Z \leftarrow Z$ times Z , and return to step A2. ■

Abbildung: Knuth, TAOCP vol.2, ch. 4.6.3 (RL)

The great calculator al-Kāshī stated Algorithm A in A.D. 1427 [*Istoriko-Mat. Issledovaniā* 7 (1954), 256–257]. The method is closely related to a procedure for multiplication that was actually used by Egyptian mathematicians as early as 2000 B.C.; for if we change step A3 to “ $Y \leftarrow Y + Z$ ” and step A5 to “ $Z \leftarrow Z + Z$ ”, and if we set Y to zero instead of unity in step A1, the algorithm terminates with $Y = nx$. [See A. B. Chace, *The Rhind Mathematical Papyrus* (1927); W. W. Struve, *Quellen und Studien zur Geschichte der Mathematik* A1 (1930).] This is a practical method for multiplication by hand, since it involves only the simple operations of doubling, halving, and adding. It is often called the “Russian peasant method” of multiplication, since Western visitors to Russia in the nineteenth century found the method in wide use there.

Abbildung: Knuth, TAOCP vol.2, ch. 4.6.3 (RL)

Require: $a \in H, n \in \mathbb{N}$

Ensure: $x = a^n$

$x \leftarrow e$

$A \leftarrow a$

$N \leftarrow n$

while $N \neq 0$ **do**

if N is even **then**

$A \leftarrow A * A$

$N \leftarrow N/2$

else { N is odd}

$x \leftarrow x * A$

$N \leftarrow N - 1$

end if

end while

Return(x)

► Maple-Programm (rekursiv)

```
power := proc (a,n::nonnegint)
  if n=0 then RETURN(1)
  else t := power(a,iquo(n/2))^2;      (Q)
  if odd(e) then t := t*a fi;        (M)
  RETURN(t)
fi;
end;
```

Zur Anwendung:

- KNUTH (TAOCP, vol2., ch. 4.6.3) diskutiert, wann man "binäre" ("schnelle") Exponentiation verwenden sollte und wann nicht!
- Zitat: *The point of these remarks is that binary methods are nice, but not a panacea. They are most applicable when the time to multiply $x^j \cdot x^k$ is essentially independent of j and k (for example, when we are doing floating point multiplication, or multiplication modulo m); in such cases the running time is reduced from order n to order $\log n$.*
- Vorsicht! wenn der Aufwand für die Multiplikation $x^j \cdot x^k$ proportional zu $j \cdot k$, als ist (Integer- und Polynommultiplikation!), also quadratisch mit der Anzahl der Ziffern- bzw. Koeffizientenoperationen wächst, kann der Aufwand für die "banale" Methode von der gleichen Grössenordnung wie für die "schnelle" Methode sein — oder sogar schlechter!

► Anwendungsszenario: Exponentiation in \mathbb{Z}_m

$$(a, n, m) \mapsto a^n \bmod m$$

► Maple-Programm (rekursiv)

```
modpower := proc (a,n,m)
  if n=0 then RETURN(1)
  else t := modpower(a,iquo(n/2),m)^2; (Q)
  if odd(n) then t := t*a mod m fi;    (M)
  RETURN(t mod n)
fi;
end;
```

- benötigt (etwa) $\log n$ -maliges Quadrieren und höchstens $\log n$ weitere Multiplikationen von $\log m$ -bit Zahlen und $\log n$ Reduktionen modulo m von $2 \log m$ -bit-Zahlen, also insgesamt einen Aufwand $O(\log n \cdot (\log m)^2)$ gemessen in bit-Operationen.

- Umkehrabbildung: *diskreter Logarithmus*

$$(a, a^n \bmod m, m) \mapsto n$$

- Hierfür ist bis heute kein effizienter Algorithmus bekannt! Die besten bekannten Algorithmen haben gleiche Komplexität wie die besten Algorithmen für die Faktorisierung von m
- Zahlenbeispiel: a, n, m in der Größenordnung 10^{200}
 - schnelle Exponentiation erfordert etwa 3000 Multiplikationen von 200-digit-Zahlen und 3000 Reduktionen modulo m von 400-digit-Zahlen
 - die Berechnung des diskreten Logarithmus "brute-force" würde etwa 10^{200} Multiplikationen und Reduktionen modulo m erfordern

- Anwendungsszenario: schnelle Berechnung C-rekursiver Folgen
- Kanonisches Beispiel: Fibonacci-Zahlen

- Idee: mit $F = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ gilt

$$\begin{pmatrix} f_n \\ f_{n-1} \end{pmatrix} = \begin{pmatrix} f_{n-1} + f_{n-2} \\ f_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f_{n-1} \\ f_{n-2} \end{pmatrix} = F \begin{pmatrix} f_{n-1} \\ f_{n-2} \end{pmatrix}$$

also

$$\begin{pmatrix} f_n \\ f_{n-1} \end{pmatrix} = F^{n-2} \begin{pmatrix} f_2 \\ f_1 \end{pmatrix} = F^{n-2} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

- Die Berechnung von f_n mittels iteriertem Quadrieren der Matrix F benötigt $13 \cdot \lfloor \log(n-2) \rfloor + 12 \cdot \nu(n-2) - 10$ arithmetische Operationen (Additionen, Subtraktionen, Multiplikationen, Divisionen durch 2, siehe HEUN, GA)
- Dies kann noch etwas verbessert werden, wenn man ausnützt, dass die Potenzen von F symmetrische Matrizen sind

- Gofer-Programm zur Berechnung der Fibonacci-Zahlen mittels schneller Exponentiation von Matrizen (HEUN, GA)

```

h :: Int -> (Int,Int,Int,Int)
h n | n==1 = (1, 1,
             1, 0)
    | even n = let (a,b,c,d)=h(n/2)
                 in ((a*a)+(b*c), (a*b)+(b*d),
                    (c*a)+(d*c), (c*b)+(d*d))
    | odd n = let (a,b,c,d)=h(n/2)
              in ((a*a)+(b*c)+(a*b)+(b*d), (a*a)+(b*c),
                 (c*a)+(d*c)+(c*b)+(d*d), (c*a)+(d*c))
fib3 :: Int -> Int
fib3 n | n==1 || n==2 = 1
       | n>2          = a+c where (a,b,c,d) = h (n-2)
  
```

- Siehe Materialien zur Vorlesung für Laufzeitanalysen und gemessene Laufzeiten verschiedener Algorithmen zur Berechnung von Fibonacci-Zahlen
- Die Methode lässt sich generell für C-rekursive Folgen anwenden und liefert Verfahren logarithmischer (in n) Komplexität

Hinweis:

- Das Problem, a^n in einer Halbgruppe möglichst effizient zu berechnen, hat etwas mit dem Problem der sog. *Additionsketten* zu tun.

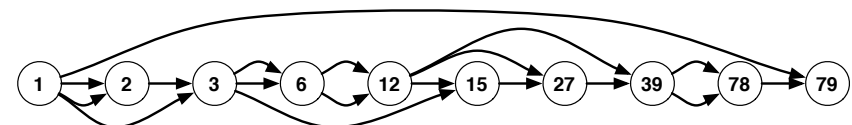
- Eine Additionskette für $n \in \mathbb{N}$ ist eine Folge

$$1 = a_0, a_1, a_2, \dots, a_r = n$$

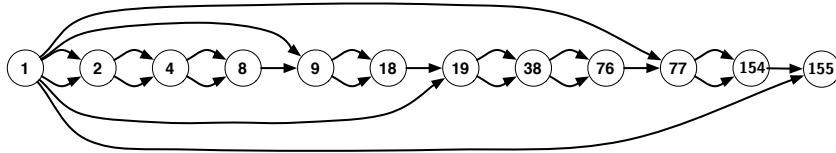
von ganzen Zahlen mit der Eigenschaft

$$a_i = a_j + a_k \text{ wobei } j \leq k < i \text{ (} 1 \leq i \leq r \text{)}$$

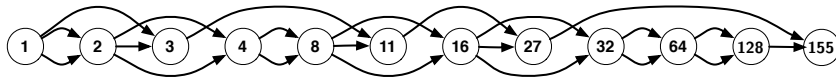
(straight-line-Programm mit Addition als einziger Operation)



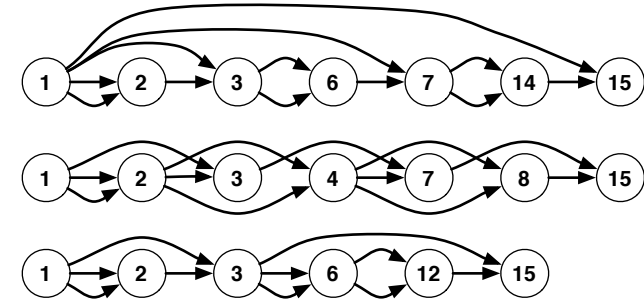
- ▶ LR-Additionskette für $n = 155$



- ▶ RL-Additionskette für $n = 155$



- ▶ Das Problem, die Länge $L(n)$ kürzester Additionsketten für gegebenes n zu bestimmen, ist sehr schwierig!
- ▶ LR- und RL-Additionsketten sind nicht immer optimal!



- ▶ Berühmte Vermutung (SCHOLZ, BRAUER, 1937,1939):

$$L(2^n - 1) \leq n - 1 + L(n)$$

- ▶ Mehr darüber in ch. 4.6.3 von TAOCP!