

basic algorithms for integers

1

implementation of base- $\beta$  arithmetic requires that certain operations on base- $\beta$ -digits are available

a common scenario is this:

$\beta$  fits into one word of the computer, so that the following can be considered as “machine operations” of constant complexity

- comparison of two one-word integers
- addition and subtraction of two one-word integers, with a one-word answer plus carry
- multiplication of two one-word integers giving a two-word answer
- division of a two-word integer by a one-word-integer giving a one-word quotient and a one-word remainder

2

- base conversion

let  $\beta, \gamma$  be bases ( $> 1$ ) and

$$a = \langle a_{k-1}a_{k-2} \dots a_1a_0 \rangle_\beta = \langle a'_{\ell-1}a'_{\ell-2} \dots a'_1a'_0 \rangle_\gamma$$

how to compute  $\langle a'_{\ell-1}a'_{\ell-2} \dots a'_1a'_0 \rangle_\gamma$  from  $\langle a_{k-1}a_{k-2} \dots a_1a_0 \rangle_\beta$ ?

$conv_{\beta,\gamma}$ : a routine which converts  $\beta$ -numbers  $0 \leq b < \max(\beta, \gamma)$  to  $\gamma$ -numbers

3

- first algorithm [uses base- $\beta$ -arithmetic]

**procedure**  $convert_{\beta,\gamma}^\beta$

**input** :  $a = \langle a_{k-1} \dots a_0 \rangle_\beta, \gamma = \langle g_{s-1} \dots g_0 \rangle_\beta$

**output**  $a = \langle a'_{\ell-1} \dots a'_0 \rangle_\gamma$

$i := 0$

**while**  $a \neq 0$  **do**

$a'_i := conv_{\beta,\gamma}(a \bmod \gamma)$

$a := a \text{ div } \gamma$

$i := i + 1$

**end while**

RETURN( $\langle \dots a'_1a'_0 \rangle_\gamma$ )

4

- second algorithm [uses base- $\gamma$ -arithmetic]

**procedure** *convert* $_{\beta,\gamma}^{\gamma}$

**input** :  $a = \langle a_{k-1} \dots a_0 \rangle_{\beta}$ ,  $\beta = \langle g_{s-1} \dots g_0 \rangle_{\gamma}$

**output**  $a = \langle a'_{\ell-1} \dots a'_0 \rangle_{\gamma}$

$m := \langle 0 \rangle_{\gamma}$

**for**  $i := k - 1$  **to** 0 **do**

$m := m * \beta + conv_{\beta,\gamma}(a_i)$

**end for**

**RETURN**( $m$ )

5

remarks

- operations *mod* and *div* will be seen later
- the first algorithm needs zero test, *mod* and *div* in base- $\beta$ -arithmetic
- the second algorithm needs addition and multiplication in base- $\gamma$ -arithmetic (see later)
- when  $\beta > \gamma$ , then  $conv_{\beta,\gamma}$  is not needed in the first algorithm; instead, in this situation the first algorithm can be used to implement  $conv_{\beta,\gamma}$
- when  $\beta < \gamma$ , then  $conv_{\beta,\gamma}$  is not needed in the second algorithm; instead, in this situation the second algorithm can be used to implement  $conv_{\beta,\gamma}$
- for small  $\beta, \gamma$ ,  $conv_{\beta,\gamma}$  can be implemented by table lookup

6

- comparison

**procedure** *compare*

**input**:  $a = \langle a_{m-1} \dots a_1 a_0 \rangle_{\beta}$ ,  $b = \langle b_{n-1} \dots b_1 b_0 \rangle_{\beta}$

$(a_{m-1} \neq 0 \neq b_{n-1})$

**output**:  $sgn(a - b)$

**if**  $m \neq n$  **then**

$sgn(m - n)$

**else**

$i := m - 1$

**while**  $a_i = b_i$  and  $i > 0$  **do**

$i := i - 1$

**end while**

$sgn(a_i - b_i)$

**end if**

7

- addition

for adding  $\langle a_{m-1} \dots a_1 a_0 \rangle_{\beta}$  and  $\langle b_{n-1} \dots b_1 b_0 \rangle_{\beta}$  one may assume that  $m \geq n$

**procedure** *addition* (of positive base- $\beta$ -numbers)

**input**:  $a = \langle a_{m-1} \dots a_1 a_0 \rangle_{\beta}$ ,  $b = \langle b_{n-1} \dots b_1 b_0 \rangle_{\beta}$

**output**: base- $\beta$  representation of  $c = a + b$

$carry := 0$

**for**  $i := 0$  **to**  $n - 1$  **do**

$s := a_i + b_i + carry$

**if**  $s < \beta$  **then**

$c_i := s$ ;  $carry := 0$

**else**

$c_i := s - \beta$ ;  $carry := 1$

**end if**

**end for**

$i := n$

**while**  $i < m$  and  $carry = 1$  **do**

$s := a_i + carry$

8

```

if  $s < \beta$  then
   $c_i := s; \text{carry} := 0$ 
else
   $c_i := s - \beta; \text{carry} := 1$ 
end if
 $i := i + 1$ 
end while
if  $i < m$  then
  for  $j := i$  to  $m - 1$  do
     $c_j := a_j$ 
  end for
else
  if  $\text{carry} = 1$  then
     $c_m := 1$ 
  end if
end if

```

9

- addition of signed numbers (subtraction)

general:

```

if  $\text{sgn}(a) = \text{sgn}(b)$  then
   $a + b = \text{sgn}(a)(|a| + |b|)$ 
else
  if  $|a| \geq |b|$  then
     $\text{sgn}(a)(|a| - |b|)$ 
  else
     $\text{sgn}(b)(|b| - |a|)$ 
  end if
end if

```

thus one need only subtraction  $a - b$  for  $a \geq b > 0$

10

- subtraction of positive numbers

**procedure** *subtraction* (of positive base- $\beta$ -numbers)

**input:**  $a = \langle a_{m-1} \dots a_1 a_0 \rangle_\beta$ ,  $b = \langle b_{n-1} \dots b_1 b_0 \rangle_\beta$ ,  $a \geq b > 0$

**output:** base- $\beta$  representation of  $c = a - b$

```

 $\text{carry} := 0$ 
for  $i := 0$  to  $n - 1$  do
   $s := a_i - b_i + \text{carry}$ 
  if  $s \geq 0$  then
     $c_i := s; \text{carry} := 0$ 
  else
     $c_i := s + \beta; \text{carry} := -1$ 
  end if
end for
 $i := n$ 
while  $i < m$  and  $\text{carry} := -1$  do
   $s := a_i + \text{carry}$ 

```

11

```

if  $s \geq 0$  then
   $c_i := s; \text{carry} := 0$ 
else
   $c_i := s + \beta; \text{carry} := -1$ 
end if
 $i := i + 1$ 
end while
for  $j := i$  to  $m - 1$  do
   $c_j := a_j$ 
end for

```

12

- multiplication

**procedure** *multiplication* (of positive base- $\beta$ -numbers)

**input:**  $a = \langle a_{m-1} \dots a_1 a_0 \rangle_\beta$ ,  $b = \langle b_{n-1} \dots b_1 b_0 \rangle_\beta$

**output:** base- $\beta$  representation of  $c = a \cdot b$

$carry := 0$

**for**  $h := 0$  to  $m + n - 2$  **do**

$t := carry$

**for**  $i := \max(0, h - n + 1)$  to  $\min(h, m - 1)$  **do**

$t := t + a_i \cdot b_{h-i}$

**end for**

$c_h := t \bmod \beta$

$carry := t \operatorname{div} \beta$

**end for**

**if**  $carry > 0$  **then**

$c_{m+n-1} := carry$

**end if**

13

improved multiplication algorithm

**procedure** *multiplication* (of positive base- $\beta$ -numbers)

**input:**  $a = \langle a_{m-1} \dots a_1 a_0 \rangle_\beta$ ,  $b = \langle b_{n-1} \dots b_1 b_0 \rangle_\beta$

**output:** base- $\beta$  representation of  $c = a \cdot b$

**for**  $i := 0$  to  $n - 1$  **do**

$c_i := 0$

**end for**

**for**  $j := 0$  to  $n - 1$  **do**

$carry := 0$

**for**  $i := 0$  to  $m - 1$  **do**

$t := a_i \cdot b_j + c_{i+j} + carry$

$carry := t \operatorname{div} \beta$

**end for**

$c_{m+j} := carry$

**end for**

14

– the “standard” algorithm has the serious disadvantage that the value of  $t$  can become bigger than  $\min(m, n) \cdot (\beta - 1)^2$  !

its memory requirement is  $\mathcal{O}(m \cdot n)$

– in the improved algorithm one always has  $carry < \beta$  and  $t < (\beta - 1)^2$

memory requirement is  $\mathcal{O}(m + n)$

for the proof of correctness note that if  $j = j_0$  then (by induction)

$$\langle a_{m-1} \dots a_0 \rangle_\beta \cdot \langle b_{j_0} \dots b_0 \rangle_\beta = \langle c_{m+j_0} \dots c_0 \rangle_\beta$$

15

- long division

the standard algorithm for long division of base- $\beta$ -numbers mimics the “school method”, estimating the digits of the quotient using only the leading two digits of the dividend and the leading digit of the divisor. Precisely:

let

$$u := \langle u_{n+1} u_n \dots u_0 \rangle_\beta \quad \text{and} \quad v := \langle v_n v_{n-1} \dots v_0 \rangle_\beta$$

with  $v_n \neq 0$  and  $u/v < \beta$ . Let  $q = u \operatorname{div} v$  be the true quotient and

$$\hat{q} = \min \left( \left\lfloor \frac{u_{n+1}\beta + u_n}{v_n} \right\rfloor, \beta - 1 \right)$$

as an estimate.

Then  $q \leq \hat{q}$  and  $v_n \geq \lfloor \beta/2 \rfloor \Rightarrow \hat{q} \leq q + 2$

16

**procedure** *(long)division* (of positive base- $\beta$ -numbers)

**input:**  $a = \langle a_{m-1} \dots a_1 a_0 \rangle_\beta$ ,  $b = \langle b_{n-1} \dots b_1 b_0 \rangle_\beta$ ,  $b \neq 0$

**output:** base- $\beta$  representation *quot* of  $a$  div  $b$  and *rem* of  $a$  mod  $b$

**if**  $a < b$  **then**

$quot := \langle 0 \rangle_\beta$ ;  $rem := a$

**else**

**if**  $b_{n-1} < \lfloor \beta/2 \rfloor$  **then**

$d := \lfloor \beta / (b_{n-1} + 1) \rfloor$

**else**

$d := 1$

**end if**

$u := d \cdot a$

$v := d \cdot b$   $\{v = \langle v_{n-1} \dots v_1 v_0 \rangle_\beta, u = \langle u_{s-1} \dots u_1 u_0 \rangle_\beta, v_{n-1} \neq 0\}$

**for**  $i := s - n + 1$  **to** 0 **by** -1 **do**

$u' := \lfloor u / \beta^i \rfloor$

$q := \min(\lfloor \lfloor u' / \beta^{n-1} \rfloor / v_{n-1} \rfloor, \beta - 1)$

**if**  $q \neq 0$  **then**

**while**  $u' - q \cdot v < 0$  **do**

$q := q - 1$

**end while**

$u := u - q \cdot v \cdot \beta^i$

**end if**

$quot := \langle quot, q \rangle_\beta$

**end for**

$rem := u/d$

**end if**

17

18

- exponentiation

here  $x$  is an element from some semigroup (group, ring, field)  $S$ ,  $n$  is a positive integer; the goal is to compute  $x^n$

the simple algorithm

**procedure** *power1*

**input:**  $x \in S$ ,  $n$  a positive integer

**output:**  $z = x^n$

$z := x$

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

$z := z \cdot x$

**end for**

a better algorithm uses recursion along the base-2-representation of the exponent

let  $n = \langle d_{k-1} \dots d_1 d_0 \rangle_2$ , then

$$x^n = x^{d_0 + d_1 2 + \dots + d_{k-1} 2^{k-1}} = x^{d_0} (x^{d_1 + d_2 2 + \dots + d_{k-1} 2^{k-2}})^2$$

**procedure** *power2*

**input:**  $x \in S$ ,  $n$  a positive integer

**output:**  $z = x^n$

**if**  $n = 1$  **then**

$z := x$

**else if**  $n$  is even **then**

$z := (\text{power2}(x, n/2))^2$

**else**

$z := x \cdot (\text{power2}(x, (n-1)/2))^2$

**end if**

19

20

yet another algorithm uses the base-2-representation of the exponent in writing

$$x^n = x^{d_0 + d_1 2 + \dots + d_{k-1} 2^{k-1}} = x^{d_0} (x^2)^{d_1} \dots (x^{2^{k-1}})^{d_{k-1}}$$

**procedure** *power3*

**input:**  $x \in S$ ,  $n$  a positive integer

**output:**  $z = x^n$

$y := x; z := 1; m := n$

**while**  $m \geq 1$  **do**

**if**  $m$  is odd **then**

$z := z \cdot y$

**end if**

$y := y^2; m := m \text{ div } 2$

**end while**

21

- complexity of computing powers

– in the *naive* model one counts only the *number* of multiplications that are performed — this is realistic, if one computes in a finite domain (in a finite field, for example). Then one has

\*  $n - 1$  multiplications for *power1*

\* at most about  $\log_2 n$  multiplications for *power2* and *power3*

which is why *power2* and *power3* are known as “fast” exponentiation schemes

22

– if one computes in  $\mathbb{Z}_n$  say, then it is more realistic to take into account the *size* of the numbers that have to be multiplied at the various stages of the run of the algorithm

here the advantage of the apparently “fast” algorithm vanishes: if the number  $x$  has  $\ell = L_\beta(x)$  base- $\beta$  digits and if  $cmn$  is the number of base- $\beta$  operations for the multiplication of numbers of  $\beta$  length  $m$  and  $n$ , then all algorithms have a worst-case complexity proportional to  $c \ell^2 n^2$

23

- reciprocal

NEWTON’s familiar method for computing zeros of differentiable functions can be adapted for integer arithmetic in order to compute reciprocals and thus also for performing division — this is a *divide-and-conquer* approach in that the method repeatedly treats problems of half the size of the input

assume here that  $\beta = 2$

the *reciprocal* of a  $n$ -bit integer  $v = \langle v_{n-1} \dots v_1 v_0 \rangle_2$  (where  $v_n = 1$ , i.e.,  $2^{n-1} \leq v < 2^n$ ) is defined to be  $2^{2n-1} \text{ div } v = \lfloor 2^{2n-1}/v \rfloor$

24

the following property is crucial:

assume that  $n = 2m$  and write  $v$  as

$$v = 2^m x + y \text{ where } 2^{m-1} \leq x < 2^m \text{ and } y < 2^m$$

put

$$A = 2^{m+1} \cdot a - \frac{a^2 v}{2^{2m-1}} \text{ where } a = 2^{2m-1} \text{ div } x$$

then

$$0 \leq (2^{n-1} \text{ div } v) - A < 4$$

25

**algorithm** *reciprocal*

**input:**  $v = \langle v_{n-1} \dots v_1 v_0 \rangle_2$  with  $v_{n-1} = 1$

**output:**  $\text{recip} := 2^{2n-1} \text{ div } v$

**if**  $n=1$  **then**

$\text{return}(2)$

**else**

$s := 2^{\lceil \log_2 n \rceil}; v := 2^{s-n} v; m := s/2$

$x := \lfloor v/2^m \rfloor$

$a := \text{reciprocal}(x)$

$\text{guess} := 2^{m+1} a - (a^2 v \div 2^{2m-1}) - 1$

$\text{rem} := 2^{4m-1} - v \cdot \text{guess}$

**while**  $\text{rem} > v$  **do**

$\text{guess} := \text{guess} + 1; \text{rem} := \text{rem} - v$

**end while**

$\text{return}(\text{guess} \text{ div } 2^{s-n})$

**end if**

26

complexity of algorithm *reciprocal*

$M(n)$  : complexity of multiplying two  $n$ -bit integers

$R(n)$  : complexity of computing the reciprocal of an  $n$ -bit integer

algorithm *reciprocal* shows

$$R(n) \leq R(n/2) + 2M(n) + M(n/2) + cn$$

from which it can be deduced that

$$R(n) \in \mathcal{O}(M(n))$$

27

algorithm *reciprocal* can be used to perform long division in  $\mathcal{O}(M(n))$

bit-operations:

• let  $u = \langle u_{2n-1} \dots u_1 u_0 \rangle_2, v = \langle v_{n-1} \dots v_1 v_0 \rangle_2$  with  $v_{n-1} = 1$

• compute  $q := 2^{2n-1} \text{ div } v$  using *reciprocal*

• it can be shown that

$$0 \leq u - \lfloor \frac{u \cdot q}{2^{2n-1}} \rfloor \cdot v < 3v$$

• hence  $(u \cdot q) \text{ div } 2^{2n-1}$  is a good guess for  $u \text{ div } v$ : it is never too big and too small by at most 2; this can easily be detected and corrected

28

as a consequence, the bit-complexities of the following operations are same (up to a constant factor), i.e., they are all  $\in \times(M(n))$

- multiplication of two  $n$ -bit integers
- computing the reciprocal of an  $n$ -bit integer
- division with remainder of an  $2n$ -bit integer by an  $n$ -bit integer
- squaring an  $n$ -bit integer

note that squaring can be reduced to reciprocals by  $a^2 = 1/(1/a - 1/(a+1)) - a$  and multiplication can be easily reduced to squaring

29

- about the complexity of multiplication

for big numbers, there are faster methods of multiplication than the traditional method, we mention

- the divide-and-conquer method method by KARATSUBA, based on

$$(a\beta^m + b)(c\beta^m + d) = ac\beta^{2m} + ((a-b)(d-c) + ac + bd)\beta^m + bd$$

- a method by TOOM and COOK, which uses the evaluation-interpolation paradigm together with a divide-and-conquer subdivision scheme
- the FFT-based method by SCHÖNHAGE and STRASSEN

30

### complexity of long-integer operations (survey)

for positive integers  $a$  their *size* with respect to base  $\beta$  = the number of  $\beta$ -digits in their base- $\beta$ -representation (the  $\beta$ -length of  $a$ ) will be denoted by  $L_\beta(a)$

note:  $L_\beta(a) = \lfloor \log_\beta a \rfloor + 1$

conversion	$\sum a_i \alpha^i \leftrightarrow \sum b_j \beta^j$	$\mathcal{O}(L_\beta(a)^2)$
comparison	$(a, b) \mapsto \text{sign}(a - b)$	$\theta(\max(L_\beta(a), L_\beta(b)))$ , but $\theta(\min(L_\beta(a), L_\beta(b)))$ average
addition	$(a, b) \mapsto a + b$	$\theta(\max(L_\beta(a), L_\beta(b)))$ , but $\theta(\min(L_\beta(a), L_\beta(b)))$ average
subtraction	$(a, b) \mapsto a - b$	$\theta(\max(L_\beta(a), L_\beta(b)))$ , but $\theta(L_\beta(b))$ average

31

multiplication	$(a, b) \mapsto a * b$	trad. $\theta(L_\beta(a) \cdot L_\beta(b))$ Karatsuba: $\mathcal{O}(n^{\log_2 3})$ Toom: $\mathcal{O}(n^{1+\epsilon})$ with $\epsilon > 0$ SS: $\mathcal{O}(n \cdot \log(n) \cdot \log \log(n))$ where $n = \max(L_\beta(a), L_\beta(b))$
exponentiation	$(a, n) \mapsto a^n$	$M(n \cdot L_\beta(a))$ where $M(k)$ cost for multiplication in $\beta$ -length $k$
long division	$(a, b) \mapsto (\lfloor \frac{a}{b} \rfloor, \frac{a}{b} - \lfloor \frac{a}{b} \rfloor)$	$\mathcal{O}(L_\beta(b)(L_\beta(a) - L_\beta(b) + 1))$ where $a \geq b$
gcd (Euclid)	$(a, b) \mapsto \text{ggT}(a, b)$	$\mathcal{O}(L_\beta(b)(L_\beta(a) - L_\beta(d) + 1))$ where $a \geq b$ and $d = \text{gcd}(a, b)$

32