NADA
Numerisk analys och datalogi
Kungl Tekniska Högskolan
100 44 STOCKHOLM

Department of Numerical Analysis
and Computer Science
Royal Institute of Technology
SE-100 44 Stockholm, SWEDEN

Fast Division of Large Integers

A Comparison of Algorithms

Karl Hasselström
d98-kha@nada.kth.se

TRITA-InsertNumberHere

Master's Thesis in Computer Science (20 credits)
at the School of Computer Science and Engineering,
Royal Institute of Technology, February 2003
Supervisor at Nada was Stefan Nilsson
Examiner was Johan Håstad

Abstract

I have investigated, theoretically and experimentally, under what circumstances Newton division (inversion of the divisor with Newton's method, followed by division with Barrett's method) is the fastest algorithm for integer division. The competition mainly consists of a recursive algorithm by Burnikel and Ziegler.

For divisions where the dividend has twice as many bits as the divisor, Newton division is asymptotically fastest if multiplication of two n-bit integers can be done in time $\mathcal{O}(n^c)$ for $c < 1.297$, which is the case in both theory and practice. My implementation of Newton division (using subroutines from GMP, the GNU Multiple Precision Arithmetic Library) is faster than Burnikel and Ziegler's recursive algorithm (a part of GMP) for divisors larger than about five million bits (one and a half million decimal digits), on a standard PC.

Snabb division av stora heltal

En jämförelse av algoritmer

Sammanfattning

Jag har undersökt, ur både teoretiskt och praktiskt perspektiv, under vilka omständigheter Newtondivision (invertering av nämnaren med Newtons metod, följd av division med Barretts metod) är den snabbaste algoritmen för heltalsdivision. Den främsta konkurrenten är en rekursiv algoritm av Burnikel och Ziegler.

För divisioner där täljaren har dubbelt så många bitar som nämnaren är Newtondivision snabbast asymptotiskt om multiplikation av två n-bitars heltal kan göras på tid $\mathcal{O}(n^c)$ med $c < 1.297$, vilket är fallet både teoretiskt och i praktiken. Min implementation av Newtondivision (som använder subrutiner från GMP, GNU:s multiprecisionsaritmetikbibliotek) är snabbare än Burnikel och Zieglers rekursiva algoritm (en del av GMP) för nämnare större än cirka fem miljoner bitar (en och en halv miljon decimala siffror), på en vanlig PC.

Preface

This is my Master's thesis at the Department of Numerical Analysis and Computer Science (Nada) at the Royal Institute of Technology in Stockholm, Sweden.

The subject of this project was invented by myself and my supervisor Torbjörn Granlund of Swox AB, primary maintainer of the bignum library GMP (see section 1.2). He spent a great deal of time helping me, and just generally explaining various aspects of bignum programming and discussing new ideas with me. Hopefully, I was able to produce at least a modest amount of useful feedback.

My supervisor at Nada was Stefan Nilsson; it was his inviting Torbjörn Granlund as a guest lecturer on bignum arithmetic that gave me the idea of this project in the first place.

The code that I have written as a part of this project is not production quality yet, but my intention is to improve it until it is fit to be incorporated in GMP – not because I am paid to do that, but because GMP is free software (see section 1.3).

Contents

Chapter 1

Introduction

1.1 What Is the Problem?

Computers have special hardware to do arithmetic on limited-size integers (and often floating-point numbers as well). This hardware only handles small integers, typically no larger than 32 or 64 bits. However, since even the smaller of those is greater than a billion, they are sufficient for the needs of almost all applications. (The same is true for floating-point numbers; they are typically represented with a precision well in excess of ten decimal digits.)

But not all numbers are small. The popular RSA encryption algorithm (introduced in [8]) needs integers that are at least several hundred bits long. Practical experiments with number-theoretic conjectures will often require arithmetic on large numbers as well, with 'large' limited only by the number of CPU-years the algorithm will take at that precision. Some numerical algorithms need more precision for intermediate results than the hardware operations give, to compensate for rounding errors, cancellation and such.

Then, of course, there are people who want to compute π to billions of digits, or render a really tiny piece of the Mandelbrot fractal (so that coordinates with a hundred significant digits are required), and other such worthwhile projects.

To sum it up, there is a need for high-precision arithmetic. And just as the vast majority of applications do not need more precision than the hardware arithmetic gives, most of those who do need more do not need that *much* more. But some do.

1.2 GMP

GMP, the GNU Multiple Precision Arithmetic Library[3], is a software library that provides routines for doing arithmetic on arbitrarily large integers. (Furthermore, I should mention that it is also free software; see the terminology section below.) For many arithmetic operations, it uses different algorithms depending on the size of the operands, since for small operands, simple but asymptotically slow algorithms are often faster than complicated but asymptotically fast algorithms; most asymptotically fast algorithms are fast because they spend time doing various pre- and postprocessing stages that makes things easier for the heaviest parts, but for small inputs the overhead of this preprocessing makes them perform less than spectacularly. In practice, this overhead is often even greater than one would think after having studied the theory, because when programming a simple algorithm, the programmer can be more clever (and thus squeeze more useful operations per second out of the computer) without getting confused. For example, GMP implements many simple algorithms (and subroutines of not so simple algorithms) in assembly language for lots of processors, taking factors such as the processor pipeline into account.

Currently, GMP implements all the multiplication algorithms discussed in appendix A. During compilation, the speed of each algorithm is measured for different operand sizes to determine the points where one algorithm starts to be faster than another.

GMP implements the schoolbook (algorithm 3.2) and divide-and-conquer (algorithm 3.3) algorithms for division, but not Newton inversion (algorithm 4.2) or Barrett's algorithm (algorithm 3.5). However, since theorems 3.6, 3.10 and 4.3 imply that Barrett's Algorithm together with Newton inversion is asymptotically faster than divide-and-conquer division when one has fast enough multiplication, maybe it should.

1.2.1 My Contribution

I have implemented Newton inversion and Barrett's algorithm, and pitted them against GMP's divide-and-conquer division. The intent being, of course, to either submit them to the GMP maintainers for possible inclusion into future versions of the library, or to prove that the breakeven point is too high for Newton and Barrett to be of practical interest on today's computers[1].

When doing arithmetic, there is one other thing besides speed that is important: correctness. Dividing fast is of no use if the result is not correct. This makes for some interesting mathematical tightrope-walking when I prove that the algorithms I have implemented actually work, since both Newton inversion and Barrett's algorithm are at heart slightly inexact.

1.3 Terminology

Radix point refers to the point separating the integer and fraction part of a number (this is the same thing as 'decimal point,' but without implying that the base is ten). **Integer digit** means 'digit to the left of the radix point,' and **fraction digit** means 'digit to the right of the radix point.'

[1]However, in that case they may still become practical in the future; their asymptotic superiority guarantees that as long as computers keep getting faster, the upper limit of practical interest must eventually overtake the breakeven point.

The phrase 'x is k bits [long]' means that x is a number (not necessarily an integer) whose representation uses k binary digits.

Bignum is the synonym for 'arbitrary precision arithmetic' often used by those who frequently need to refer to that concept.

Free software is software that anyone is free to do exactly what they want with, except denying other people the same freedom. The GNU project defines free software like this[2]:

> 'Free software' is a matter of liberty, not price. To understand the concept, you should think of 'free' as in 'free speech,' not as in 'free beer.'
>
> Free software is a matter of the users' freedom to run, copy, distribute, study, change and improve the software. More precisely, it refers to four kinds of freedom for the users of the software:
>
> **Freedom 0** The freedom to run the program, for any purpose.
>
> **Freedom 1** The freedom to study how the program works, and adapt it to your needs. Access to the source code is a precondition for this.
>
> **Freedom 2** The freedom to redistribute copies so you can help your neighbor.
>
> **Freedom 3** The freedom to improve the program, and release your improvements to the public, so that the whole community benefits. Access to the source code is a precondition for this.

One way to make your program free is to release it under the GNU General Public License (GPL), or the Lesser General Public License (LGPL; this is the license GMP uses). You can find the text of these licenses, as well as everything else you always wanted to know about free software but were too shy to ask, at `http://www.gnu.org/`.

[2]The quote is from `http://www.gnu.org/philosophy/free-sw.html`.

Chapter 2

Simple Algorithms for Arithmetic

The first step on the way to arbitrary-precision arithmetic is to realize that we all (hopefully) know how to add, subtract, multiply and divide using pencil and paper. These four algorithms are known as the **schoolbook** algorithms (for addition, multiplication, etc.), because that is where they are taught.

We will only concern ourselves with positive integers from now on, because operations on other kinds of numbers, such as floating-point and fractions, reduce easily to operations on integers.

We represent our integers in base $\beta \in \mathbb{N}$, $\beta \geq 2$. An arbitrary number (less than β^k) can then be uniquely written $N = a_{k-1}\beta^{k-1} + a_{k-2}\beta^{k-2} + \ldots + a_1\beta^1 + a_0$ for some integers a_i, $0 \leq a_i < \beta$ (these are the digits). For example, if $\beta = 10$, we can write any number less than 10^k as $N = 10^{k-1}a_{k-1} + 10^{k-2}a_{k-2} + \ldots + 10a_1 + a_0$, where each of the a_i is one of the numbers 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. Ten is an unlikely base, however; in practice, β will always be a power of two, often 2^{32} or 2^{64}.

The hardware is assumed capable of the following arithmetic operations:

- Adding two single-digit numbers a and b, giving a single-digit result $a + b$ (or $a + b - \beta$ if $a + b \geq \beta$).
- Subtracting one single-digit number b from another single-digit number a, giving a single-digit result $a - b$ (or $a - b + \beta$ if $a - b < 0$).
- Multiplying two single-digit numbers a and b, giving the two-digit result ab.
- Dividing a two-digit number a by a single-digit number b, giving the single-digit quotient $\lfloor a/b \rfloor$ and the single-digit remainder $a \mod b$. (Since the quotient must fit in one digit, we require that $\lfloor a/b \rfloor < \beta$ for this operation to work.)

2.1 Schoolbook Addition

Algorithm 2.1 is the schoolbook algorithm for addition. For every pair of digits with the same index, starting with the least significant, it adds the digits to produce the corresponding digit of the result. If the result digit is β or more, it subtracts β from

Algorithm 2.1 Schoolbook addition

```
Input Two nonnegative integers $A$ and $B$, and an integer $k$ such that $A < \beta^k$ and
      $B < \beta^k$.

Output The sum $C = A + B$.

  1. Let $A = a_{k-1}\beta^{k-1} + a_{k-2}\beta^{k-2} + \ldots + a_1\beta^1 + a_0$ and $B = b_{k-1}\beta^{k-1} + b_{k-2}\beta^{k-2} +$
     $\ldots + b_1\beta + b_0$.
  2. $i \leftarrow 0$
  3. $q \leftarrow 0$
  4. (a) $c_i \leftarrow a_i + b_i + q \mod \beta$
     (b) $q \leftarrow \lfloor \frac{a_i+b_i+q}{\beta} \rfloor$.
     (c) $i \leftarrow i + 1$
     (d) If $i = k$, return $C = q\beta^k + c_{k-1}\beta^{k-1} + \ldots + c_1\beta + c_0$.
     (e) Goto step 4a.
```

the current digit and remembers (in the variable q) that it has to add 1 to the result of the position immediately to the left.

I will leave the proofs of the following two theorems to the reader:

Theorem 2.1. Algorithm 2.1 returns the sum C of A and B. The most significant digit of C (that is, the digit with index k) is 0 or 1.

Theorem 2.2. Algorithm 2.1 stops after k iterations of the loop, and thus adds two n-digit numbers in $\Theta(n)$ time.

2.2 Schoolbook Subtraction

Algorithm 2.2 is the schoolbook algorithm for subtraction. As you can see, it is very similar to the addition algorithm: it proceeds from the least to the most significant pair of digits, but it subtracts instead of adding, and if the result digit is less than zero, it adds β to that digit and remembers to add -1 to the result of the position immediately to the left.

Note what happens when $B > A$. The result should have been negative then, but the algorithm returns $\beta^k + A - B$ instead (which is a positive number). If this behavior is undesirable, you might want to make sure that $A \geq B$ before using this algorithm.

Again, I state some theorems without their (easy) proofs:

Theorem 2.3. Algorithm 2.2 returns $C = A - B$ and $q = 0$ if $A \geq B$, and $C = \beta^k + A - B$ and $q = -1$ otherwise. In both cases, $q\beta^k + C = A - B$.

Algorithm 2.2 Schoolbook subtraction

Input Two nonnegative integers A and B, and an integer k such that $A < \beta^k$ and $B < \beta^k$.

Output The difference $C = A - B$, if $A \geq B$, or $C = \beta^k + A - B$ otherwise. A borrow-out flag q, which is 0 in the first case and -1 in the second case.

```
1. Let $A = a_{k-1}\beta^{k-1} + a_{k-2}\beta^{k-2} + \ldots + a_1\beta^1 + a_0$ and $B = b_{k-1}\beta^{k-1} + b_{k-2}\beta^{k-2} + \ldots + b_1\beta + b_0$.
2. $i \leftarrow 0$
3. $q \leftarrow 0$
4. (a) $c_i \leftarrow a_i - b_i + q \mod \beta$
   (b) $q \leftarrow \lfloor \frac{a_i - b_i + q}{\beta} \rfloor$.
   (c) $i \leftarrow i + 1$
   (d) If $i = k$, return $C = c_{k-1}\beta^{k-1} + \ldots + c_1\beta + c_0$ and the borrow-out flag $q$.
   (e) Goto step 4a.
```

Theorem 2.4. Algorithm 2.2 stops after k iterations of the loop, and thus subtracts two n-digit numbers in $\Theta(n)$ time.

2.3 Schoolbook Multiplication

Algorithm 2.3 is almost, but not quite, identical to the algorithm one learns in school. In both algorithms, the problem of multiplying a k-digit number by an m-digit number is reduced to m multiplications of the k-digit number by the individual digits in the m-digit number, and then adding the results; the difference is that with the pencil-and-paper algorithm, one first computes all the partial products and then adds them, while algorithm 2.3 first computes the first product, then computes the second product and adds it to the first in one step, then computes the third product and adds it to the sum in one step, etc. The first method is preferable if one cannot overwrite previous partial results easily, while the second is preferable if overwrites are possible and one wishes to save space (either paper or memory).

Lemma 2.5. In algorithm 2.3, we have $0 \leq t_j < \beta^2$ and $0 \leq q_j < \beta$ for all j.

Proof. We proceed by induction on j. It is obviously true for q_{-1}, since it is zero. Now, provided that $0 \leq q_{j-1} < \beta$, we have $t_j = c_{i+j} + q_{j-1} + a_i b_j \geq 0 + 0 + 0 \cdot 0 = 0$ and $t_j = c_{i+j} + q_{j-1} + a_i b_j \leq (\beta - 1) + (\beta - 1) + (\beta - 1)(\beta - 1) = \beta^2 - 1 < \beta^2$. (Since the c_n are assigned modulo β, we have $0 \leq c_n < \beta$ for all n.) And now that we know that $0 \leq t_j < \beta^2$, we have that $q_j = \lfloor \frac{t_j}{\beta} \rfloor \geq 0$ and $q_j = \lfloor \frac{t_j}{\beta} \rfloor \leq \lfloor \frac{\beta^2 - 1}{\beta} \rfloor = \beta - 1$. ∎

Algorithm 2.3 Schoolbook multiplication

```
Input  Two nonnegative integers $A$ and $B$, and two integers $k$ and $m$ such that
       $A < \beta^k$ and $B < \beta^m$.

Output  The product $C = AB$.

1. Let $A = a_{k-1}\beta^{k-1} + a_{k-2}\beta^{k-2} + \ldots + a_1\beta^1 + a_0$ and $B = b_{m-1}\beta^{m-1} +$
   $b_{m-2}\beta^{m-2} + \ldots + b_1\beta + b_0$.
2. $c_n \leftarrow 0$ for all $0 \le n \le m-1$
3. $i \leftarrow 0$
4. (a) $j \leftarrow 0$
   (b) $q_{-1} \leftarrow 0$
   (c)   i. $t_j \leftarrow c_{i+j} + q_{j-1} + a_i b_j$
        ii. $c_{i+j} \leftarrow t_j \mod \beta$
       iii. $q_j \leftarrow \lfloor \frac{t_j}{\beta} \rfloor$
        iv. $j \leftarrow j+1$
         v. If $j < m$, goto step 4(c)i.
   (d) $c_{i+m} \leftarrow q_{j-1}$
   (e) $i \leftarrow i+1$
   (f) If $i < k$, goto step 4a.
5. Return $C = c_{k+m+1}\beta^{k+m+1} + c_{k+m}\beta^{k+m} + \ldots + c_1\beta + c_0$.
```

Theorem 2.6. Algorithm 2.3 returns the product C of A and B.

Proof. We will prove by induction that after step 4d, $C_i = c_{i+m}\beta^{i+m} + \ldots + c_1\beta + c_0$ is the product of B and $A = a_{i-1}\beta^{i-1} \ldots + a_1\beta^1 + a_0$, and that for $0 \le n \le i+m$, $0 \le c_n < \beta$.

It is true for $i = 0$, since for every digit b_j in B, we add $a_0 b_j \beta^j$ to C, and every digit c_n we write satisfies $0 \le c_n < \beta$, the most significant one because $0 \le q < \beta$ and the others because they are modulo β.

Assuming it is true for i, it is true for $i+1$ as well, since for every digit b_j in B, we add $a_{i+1} b_j \beta^{j+i+1}$ to C_i, and every digit c_n we write satisfies $0 \le c_n < \beta$, as before. ■

Theorem 2.7. Algorithm 2.3 stops after km iterations of the inner loop, and thus multiplies two n-digit numbers in $\Theta(n^2)$ time.

Proof. The outer loop is executed once for $i = 0$, once for $i = 1$, and so on, the last one being $i = k-1$. That makes a total of k times. The same reasoning applied to the inner loop says that it executes m times for every run of the outer loop. ∎

There is one thing of particular interest among these two results, besides the fact that the schoolbook algorithm is correct: it takes $\Theta(n^2)$ time to multiply two n-bit numbers. Addition and subtraction only take $\Theta(n)$ time (which is obviously optimal since we have to read all the input), and we can afford that much, *but for large n, quadratic time is hopelessly slow.* Essentially, if the algorithm is linear then the limiting factor is the computer's memory, whereas if the algorithm is quadratic, the limiting factor is the amount of time one is willing to wait.

Lucky for us, then, that there are faster multiplication algorithms. You will find a few in appendix A, but understanding them is not a prerequisite for the rest of this report; just noting how fast they are is enough.

Chapter 3

Division Algorithms

Integer addition, subtraction and multiplication all give results that are integers. This is not the case for division, so we need to specify a little more carefully exactly what kind of answer we want.

Consider two integers A and B, with $A \geq 0$ and $B > 0$. When we divide A (the dividend) by B (the divisor), we want an integer quotient Q and an integer remainder R such that $A = BQ + R$ and $0 \leq R < B$. These numbers have the following properties:

- Since $0 \leq R < B$ and $A \geq 0$, we must have $Q \geq 0$ and $R \leq A$.
- Since $Q = \frac{A-R}{B}$ is an integer, $A - R$ must be divisible by B. There is exactly one R between 0 and $B - 1$ that satisfies this, so R is uniquely defined. It follows that Q is uniquely defined as well.
- $Q = \frac{A-R}{B} = \frac{A}{B} - \frac{R}{B} \leq \frac{A}{B}$ and $Q = \frac{A-R}{B} = \frac{A}{B} - \frac{R}{B} > \frac{A}{B} - 1$, so $Q = \lfloor \frac{A}{B} \rfloor$.
- $R = A - BQ = A \mod B$.

3.1 Schoolbook Division

The essentials of this section are from Knuth[6]. For pedagogic reasons, I have split up the algorithm into one main part (algorithm 3.2) and one subroutine (algorithm 3.1).

The entire subject of schoolbook division rests on a rather neat little fact: when dividing an $(n+1)$-digit number by an n-digit number, we may disregard the $n-1$ least significant digits of both numbers, so that we get a 2-by-1 division; the quotient will be almost correct anyway. This is stated more precisely is theorem 3.1:

Theorem 3.1. The approximate quotient q computed in step 3 of algorithm 3.1 satisfies $q_c \leq q \leq q_c + 2$, where q_c is the correct quotient.

Proof. We consider the two inequalities separately:

Algorithm 3.1 Schoolbook division subroutine

Input Two integers A and B, such that $0 \leq A < \beta^{n+1}$ and $\beta^n/2 \leq B < \beta^n$.

Output The quotient $\lfloor A/B \rfloor$ and the remainder $A \mod B$.

```
1. If $A \geq B\beta$, compute the quotient $q$ and remainder $r$ of $(A - B\beta)/B$ recursively, and return $\beta + q$ and $r$.
2. Let $A = a_n\beta^n + a_{n-1}\beta^{n-1} + \ldots + a_1\beta + a_0$ and $B = b_{n-1}\beta^{n-1} + b_{n-2}\beta^{n-2} + \ldots + b_1\beta + b_0$.
3. $q \leftarrow \lfloor \frac{\beta a_n + a_{n-1}}{b_{n-1}} \rfloor$, or $\beta - 1$ if that cannot be computed (due to the result being greater than $\beta - 1$).
4. $T \leftarrow qB$
5. If $T > A$, $q \leftarrow q - 1$ and $T \leftarrow T - B$.
6. If $T > A$, $q \leftarrow q - 1$ and $T \leftarrow T - B$.
7. Return the quotient $q$ and the remainder $R = A - T$.
```

$q_c \leq q$: If $q = \beta - 1$, we are home free since $q_c \leq \beta - 1$. Otherwise we have $q = \lfloor (\beta a_n + a_{n-1})/b_{n-1} \rfloor > (\beta a_n + a_{n-1})/b_{n-1} - 1$, so that $qb_{n-1} > \beta a_n + a_{n-1} - b_{n-1}$, and since this is an integer inequality, $qb_{n-1} \geq \beta a_n + a_{n-1} - b_{n-1} + 1$. It follows that

$$\begin{aligned} A - qB &\leq A - qb_{n-1}\beta^{n-1} \\ &\leq (a_n\beta^n + \ldots + a_0) - (a_n\beta + a_{n-1} - b_{n-1} + 1)\beta^{n-1} \\ &= a_{n-2}\beta^{n-2} + \ldots + a_0 - \beta^{n-1} + b_{n-1}\beta^{n-1} \\ &< b_{n-1}\beta^{n-1} \\ &\leq B \end{aligned} \tag{3.1}$$

The same is true per definition for the correct quotient ($A - q_cB < B$), so we must have that $q \geq q_c$.

$q \leq q_c + 2$: We will prove this by contradiction. Assume to the contrary that $q \geq q_c + 3$. We have

$$q \leq \frac{a_n\beta + a_{n-1}}{b_{n-1}} = \frac{a_n\beta^n + a_{n-1}\beta^{n-1}}{b_{n-1}\beta^{n-1}} \leq \frac{A}{b_{n-1}\beta^{n-1}} < \frac{A}{B - \beta^{n-1}} \tag{3.2}$$

since $A \geq a_n\beta^n + a_{n-1}\beta^{n-1}$ and $B < (b_{n-1} + 1)\beta^{n-1}$.

Now, the relation $q_c > A/B - 1$ implies that

$$\begin{aligned} 3 \le q - q_c &< \frac{A}{B - \beta^{n-1}} - \frac{A}{B} + 1 \\ &= \frac{AB}{B^2 - B\beta^{n-1}} - \frac{AB - A\beta^{n-1}}{B^2 - B\beta^{n-1}} + 1 \\ &= \frac{A}{B}\left(\frac{\beta^{n-1}}{B - \beta^{n-1}}\right) + 1 \end{aligned} \tag{3.3}$$

so that

$$\frac{A}{B} > 2\frac{B - \beta^{n-1}}{\beta^{n-1}} \ge 2(b_{n-1} - 1) \tag{3.4}$$

Finally, since $\beta - 4 \ge q - 3 \ge q_c = \lfloor A/B \rfloor \ge 2(b_{n-1} - 1)$ (the last inequality is true since $2(b_{n-1} - 1)$ is an integer), we have $b_{n-1} \le \beta/2 - 1$, so that $B < \beta^n/2$. However, $B \ge \beta^n/2$, so we have a contradiction.

■

Corollary 3.2. Algorithm 3.1 returns the correct quotient and remainder.

Proof. If $A \ge B\beta$, we return the correct quotient and remainder by induction on A. Otherwise, theorem 3.1 guarantees that the approximate quotient is not too small, and at most two greater than the correct quotient, so the two correction steps of algorithm 3 are sufficient to make the quotient correct. The remainder must then be correct too, since it is $A - qB$. (The correction steps work as intended since $qB > A$ if and only if $q > \lfloor A/B \rfloor$, because $q > \lfloor A/B \rfloor \Leftrightarrow q \ge \lfloor A/B \rfloor + 1 > A/B \Leftrightarrow qB > A$ and $q \le \lfloor A/B \rfloor \le A/B \Leftrightarrow qB \le A$.) ■

Theorem 3.3. Algorithm 3.1 takes $\mathcal{O}(n)$ time.

Proof. Since $A < \beta^{n+1} \le 2B\beta$, this algorithm calls itself recursively at most once. The rest of the algorithm clearly runs in $\mathcal{O}(n)$ time, since it calls a constant number of subroutines that all run in $\mathcal{O}(n)$ time. ■

Algorithm 3.2 is schoolbook division. It is capable of dividing any two positive integers A and B, as long as the most significant digit of B is at least $\beta/2$. Since $B \neq 0$, we can always coerce B into this form without affecting the result by multiplying both A and B by some suitable (small, single-digit) constant[1].

Step 4 can be done by simply splitting the digits of A into an upper and a lower half, since we represent numbers as a sequence of digits in base β. For the same

[1]In a practical application, where β is a power of two, it is always possible to pick a small constant that is a power of two, so that the multiplication can be done by fast bit- operations.

Algorithm 3.2 Schoolbook division

Input Two integers A and B, such that $\beta^{m-1} \leq A < \beta^m$ and $\beta^n/2 \leq B < \beta^n$.

Output The quotient $\lfloor A/B \rfloor$ and the remainder $A \mod B$.

```
1. If $m < n$, return the quotient 0 and the remainder $A$.
2. If $m = n$, then if $A < B$, return the quotient 0 and the remainder $A$; if $A \geq B$,
   return the quotient 1 and the remainder $A - B$.
3. If $m = n + 1$, compute the quotient and remainder of $A/B$ using algorithm
   3.1 and return them.
4. $A' \leftarrow \lfloor A/\beta^{m-n-1} \rfloor$ and $s \leftarrow A \mod \beta^{m-n-1}$
5. Compute the quotient $q'$ and the remainder $r'$ of $A'/B$ using algorithm 3.1.
6. Compute the quotient $q$ and remainder $r$ of $\frac{\beta^{m-n-1}r'+s}{B}$ recursively.
7. Return the quotient $Q = \beta^{m-n-1}q' + q$ and remainder $R = r$.
```

reason, $\beta^{m-n-1}r' + s$ can be constructed simply by concatenating the digits of r' and s (since $0 \leq s < \beta^{m-n-1}$). The same goes for $\beta^{m-n-1}q' + q$, since

$$\begin{aligned} q &= \lfloor \frac{\beta^{m-n-1}r' + s}{B} \rfloor \\ &\leq \lfloor \frac{\beta^{m-n-1}(B-1) + (\beta^{m-n-1} - 1)}{B} \rfloor \\ &= \lfloor \frac{\beta^{m-n-1}B - 1}{B} \rfloor \\ &= \beta^{m-n-1} + \lfloor \frac{-1}{B} \rfloor \\ &< \beta^{m-n-1} \end{aligned} \tag{3.5}$$

This means that algorithm 3.2 can be informally stated as follows: 'Divide the $n + 1$ most significant digits of A by the n digits of B. If A only had $n + 1$ digits, we are done. If not, the quotient of that division is the most significant digit of the total quotient. Get the remaining digits of the quotient and the total remainder by dividing the remainder followed by the ignored digits of A.' This is at least somewhat reminiscent of what I was taught in school, although it still sounds more precise than I remember it.

Theorem 3.4. Algorithm 3.2 returns the correct quotient and remainder.

Proof. If $m \leq n$, the quotient and remainder are obviously correct. If $m = n + 1$, algorithm 3.2 returns the correct remainder and quotient by corollary 3.2.

Otherwise, $m > n + 1$ so that $m - n - 1 \geq 1$. $A' < \beta^{n+1}$ since $A < \beta^m$, so the conditions of algorithm 3.1 are met.

The quotient we return is

$$\begin{aligned} Q = \beta^{m-n-1}q' + q &= \beta^{m-n-1}\frac{A' - r'}{B} + \frac{\beta^{m-n-1}r' + s - r}{B} \\ &= \frac{\beta^{m-n-1}A' + s - r}{B} \\ &= \frac{A - R}{B} \end{aligned} \tag{3.6}$$

so we have $A = BQ + R$. Since $0 \leq R < B$ by induction ($\beta^{m-n-1}r' + s < \beta^{m-1} \leq A$), Q and R are the correct quotient and remainder. ∎

Theorem 3.5. Algorithm 3.2 runs in $\mathcal{O}(mn)$ time if $m > n$.

Proof. Let the algorithm run in $T(m, n)$ time. The recursive call takes $T(m - 1, n)$ time, and the rest takes $\mathcal{O}(n)$ time[2]. We stop recursing when $m \leq n + 1$, so we run the body of the algorithm $\mathcal{O}(m - n)$ times. This means that $T(m, n) \in \mathcal{O}(mn)$. ∎

3.2 Divide-and-Conquer Division

Burnikel and Ziegler[2] describe a divide-and-conquer division algorithm, based on schoolbook division. There are two algorithms, algorithms 3.3 and 3.4, that call each other recursively; they both divide one integer by another and return the quotient and remainder.

I will not give the correctness proofs here; Burnikel and Ziegler spend a number of pages doing precisely that. However, the overall idea is quite simple:

Algorithm 3.3 is a description of how to divide a dividend of $2n$ digits by a divisor of n digits by seeing each group of $n/2$ digits as one large digit, so that we get a 4 digits by 2 digits division, and simply use schoolbook division (algorithm 3.2) on those huge digits.

Algorithm 3.4 divides $3n$ digits by $2n$ digits, by grouping n digits into one huge digit so that we have a 3 digits by 2 digits division. This division is computed just as in algorithm 3.1, by ignoring all but the most significant (huge) digit of the divisor, and the two most significant digits of the dividend, calling algorithm 3.2 to compute that approximate quotient, and then correcting it.

Let the time it takes to divide $2n$ digits by n digits using algorithm 3.3 be $T(n)$, and let $M(n)$ be the time it takes to multiply two n-digit numbers. Then $T(n) = 2T'(n/2) + K$, where $T'(n)$ is the time it takes to divide $3n$ digits by $2n$ digits using algorithm 3.4, and K is the constant time we spend juggling pointers and the like. Similarly, $T'(n) \leq T(n) + M(n) + Ln$, since we may or may not call algorithm

[2]If one does not implement the body of the function very carefully, it will take $\mathcal{O}(max(m-n, n))$ time. If $m \notin \mathcal{O}(n)$, this is not in $\mathcal{O}(n)$.

Algorithm 3.3 Divide-and-conquer division (2 by 1)

```
Input Two nonnegative integers $A$ and $B$, such that $A < \beta^n B$ and $\beta^n/2 \le B < \beta^n$.
    $n$ must be even.

Output The quotient $\lfloor A/B \rfloor$ and the remainder $A \mod B$.

1. Let $A = A_3\beta^{3n/2} + A_2\beta^n + A_1\beta^{n/2} + A_0$ and $B = B_1\beta^{n/2} + B_0$, with $0 \le A_i <$
   $\beta^{n/2}$ and $0 \le B_i < \beta^{n/2}$.
2. Compute the high half $Q_1$ of the quotient as $Q_1 = \frac{A_3\beta^n + A_2\beta^{n/2} + A_1}{B}$ with re-
   mainder $R_1$ using algorithm 3.4.
3. Compute the low half $Q_0$ of the quotient as $Q_0 = \frac{R_1\beta^{n/2} + A_4}{B}$ with remainder
   $R_0$ using algorithm 3.4.
4. Return the quotient $Q = Q_1\beta^{n/2} + Q_0$ and the remainder $R = R_0$.
```

Algorithm 3.4 Divide-and-conquer division (3 by 2)

```
Input Two nonnegative integers $A$ and $B$, such that $A < \beta^n B$ and $\beta^{2n}/2 \le B <$
    $\beta^{2n}$. $n$ must be even.

Output The quotient $\lfloor A/B \rfloor$ and the remainder $A \mod B$.

1. Let $A = A_2\beta^{2n} + A_1\beta^n + A_0$ and $B = B_1\beta^n + B_0$, with $0 \le A_i < \beta^n$ and
   $0 \le B_i < \beta^n$.
2. If $A_2 < B_1$, compute $\widehat{Q} = \lfloor \frac{A_2\beta^n + A_1}{B_1} \rfloor$ with remainder $R_1$ using algorithm 3.3;
   otherwise, let $\widehat{Q} = \beta^n - 1$ and $R_1 = (A_2 - B_1)\beta^n + A_1 + B_1$.
3. $\widehat{R} \leftarrow R_1\beta^n + A_4 - \widehat{Q}B_0$
4. If $\widehat{R} < 0$, $\widehat{R} \leftarrow \widehat{R} + B$ and $\widehat{Q} \leftarrow \widehat{Q} - 1$.
5. If $\widehat{R} < 0$, $\widehat{R} \leftarrow \widehat{R} + B$ and $\widehat{Q} \leftarrow \widehat{Q} - 1$.
6. Return $Q = \widehat{Q}$ and $R = \widehat{R}$.
```

3.3, and apart from that we do one n-by-n multiplication and spend an additional amount of time linear in n.

So we have $T(n) = 2T'(n/2)+K \leq 2(T(n/2)+M(n/2)+Ln/2)+K = 2T(n/2)+2M(n/2)+Ln+K$, and expanding T again we get $T(n) \leq 2(2T(n/4)+2M(n/4)+Ln/2+K)+2M(n/2)+Ln+K = 4T(n/4)+4M(n/4)+2M(n/2)+2Ln+2K$. Expanding it $\log_2 n$ times, so that we reach the bottom of the recursion, we get

$$\begin{aligned} T(n) &\leq nT(1) + \sum_{i=1}^{\log_2 n} 2^i M(n/2^i) + Ln\log_2 n + K\log_2 n \\ &= \sum_{i=1}^{\log_2 n} 2^i M(n/2^i) + \mathcal{O}(n\log n) \end{aligned} \tag{3.7}$$

If $M(n) \leq Dn^c$ for some $c > 1$, so that $M(n/2^i) \leq M(n)/2^{ic}$, we get

$$\begin{aligned} \sum_{i=1}^{\log_2 n} 2^i M(n/2^i) &= M(n)\sum_{i=1}^{\log_2 n}(2^{1-c})^i \\ &= \frac{2^{1-c}-(2^{1-c})^{\log_2 n+1}}{1-2^{1-c}}M(n) \\ &= \frac{1-n^{1-c}}{2^{c-1}-1}M(n) \\ &< \frac{1}{2^{c-1}-1}M(n) \end{aligned} \tag{3.8}$$

(Note that the term n^{1-c} that we discard grows insignificant when n is moderately large, unless c is very close to 1.)

For example, $M(n) = Dn^2$ (schoolbook multiplication) gives $T(n) < M(n) + \mathcal{O}(n\log n)$, and $M(n) = Dn^{log_2 3} \approx Dn^{1.585}$ (Karatsuba; see section A.1) gives $T(n) < 2M(n) + \mathcal{O}(n\log n)$. $M(n) = Dn^{log_3 5} \approx Dn^{1.465}$ (Toom-3; see section A.2) gives $T(n) < (2^{\log_3 5-1}-1)^{-1}M(n) + \mathcal{O}(n\log n) \approx 2.630M(n) + \mathcal{O}(n\log n)$. Figure 3.1 illustrates how the number of multiplications per division changes with the exponent; note that division is pretty much exactly as fast as multiplication when $c = 2$, and that the number of multiplications required for one division seems to grow towards infinity when c approaches 1.

If, on the other hand, $M(n) \leq Dn\log^k n$ or $M(n) \leq Dn\log^k n\log\log n$, we get (setting $f(n) = \log^k n$ or $f(n) = \log^k n\log\log n$)

$$\sum_{i=1}^{\log_2 n} 2^i M(n/2^i) = \sum_{i=1}^{\log_2 n} nf(n/2^i) = n\sum_{i=1}^{\log_2 n} f(n/2^i) \tag{3.9}$$

Theorems B.3 and B.4 tell us that the remaining sum is in $\Theta(f(n)\log n)$, so in both cases we have that $T(n) \in \Theta(M(n)\log n)$.

We state these results as a theorem:

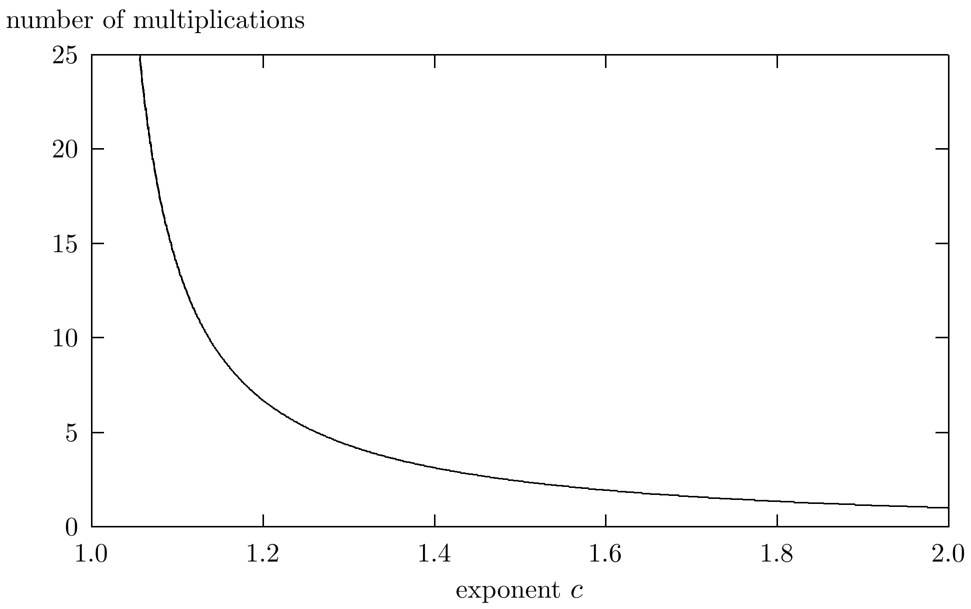

Figure 3.1. The relation between multiplication time and division time (with algorithm 3.3) when the multiplication time is $\Theta(n^c)$.

Theorem 3.6. If the time it takes to multiply two n-digit numbers is $M(n) = Dn^c$ for some $c > 1$, then algorithm 3.3 takes at most $\frac{1}{2^{c-1}-1}M(n) + \mathcal{O}(n \log n)$ time.

If $M(n) = Dn \log^k n$ or $M(n) = Dn \log^k n \log\log n$, then algorithm 3.3 runs in time $\Theta(M(n) \log n)$.

3.3 Division With Preinverted Divisor

If integer division were exact, we could compute A/B by first computing $1/B$, and then multiplying (this is the way division is done with rational numbers, by the way). Now, integer division is *not* exact, but it is possible to compute an approximation to $1/B$ (or to $\lfloor \beta^n/B \rfloor$ for some n, to be precise), and then multiply with A to get an approximation to the quotient and remainder (for a cost of about one multiplication each). These approximations are good enough that they can be corrected in linear time.

Algorithm 3.5 (originally published by Barrett[1]) is taken from Burnikel and Ziegler[2], who describe it very briefly. I have modified the algorithm to require a less precise inverse (after a suggestion from Torbjörn Granlund) and to correct the approximate quotient and divisor using addition and subtraction instead of division. The proofs are mine.

Lemma 3.7. At step 5a in algorithm 3.5, $R_k = A - BQ_k$.

Algorithm 3.5 Barrett's method

Input Two nonnegative integers A and B such that $\beta^{n-1} \leq B < \beta^n$ and $A < \beta^m$ for some positive integers n and m, with $1 \leq n \leq m \leq 2n$. A nonnegative integer μ such that $\mu - a \leq \frac{\beta^m}{B} \leq \mu + b$ for some nonnegative numbers a and b.

Output The quotient $Q = \lfloor A/B \rfloor$ and the remainder $R = A - BQ$.

```
1. $A_1 \leftarrow \lfloor A\beta^{-(n-1)} \rfloor$
2. $Q_1 \leftarrow \lfloor A_1 \mu \beta^{-(m-n+1)} \rfloor$
3. $R_1 \leftarrow A - BQ_1$
4. $k \leftarrow 1$
5. (a) If $0 \leq R_k < B$,
           i. return $Q = Q_k$ and $R = R_k$.
   (b) If $R_k < 0$,
           i. $R_{k+1} \leftarrow R_k + B$
          ii. $Q_{k+1} \leftarrow Q_k - 1$
   (c) else,
           i. $R_{k+1} \leftarrow R_k - B$
          ii. $Q_{k+1} \leftarrow Q_k + 1$
   (d) $k \leftarrow k + 1$
   (e) Goto step 5a.
```

Proof. It is trivially true the first time we reach step 5a. And if it was true for $k = n$, it will be true for $k = n + 1$, since both sides of the equality are either increased or decreased by B in each iteration of the loop. The lemma now follows by induction. ∎

Theorem 3.8. If algorithm 3.5 returns Q and R, then $R = A - BQ$ and $0 \leq R < B$, so Q and R are the quotient and remainder of the division A/B.

Proof. Lemma 3.7 proves the first part of the theorem. The second part follows from the fact that the algorithm never halts unless the condition in step 5a is fulfilled. ∎

Theorem 3.9. Algorithm 3.5 halts after at most $a + 2$ or $b + 1$ iterations of the loop (whichever is greater).

Proof. We required that $\mu \in \left[\frac{\beta^m}{B} - a, \frac{\beta^m}{B} + b\right]$. Step 1 guarantees that $A_1 \in \left(\frac{A}{\beta^{n-1}} - 1, \frac{A}{\beta^{n-1}}\right]$, and step 2 says that $Q_1 \in \left(\frac{A_1\mu}{\beta^{m-n+1}} - 1, \frac{A_1\mu}{\beta^{m-n+1}}\right]$.

Now,

$$A_1\mu \in \left(\left(\frac{A}{\beta^{n-1}} - 1\right)\left(\frac{\beta^m}{B} - a\right), \frac{A}{\beta^{n-1}}\left(\frac{\beta^m}{B} + b\right)\right] \tag{3.10}$$

So

$$\begin{aligned}
\frac{A_1\mu}{\beta^{m-n+1}} &\in \left(\beta^{-(m-n+1)}\left(\frac{A}{\beta^{n-1}} - 1\right)\left(\frac{\beta^m}{B} - a\right), \beta^{-(m-n+1)}\frac{A}{\beta^{n-1}}\left(\frac{\beta^m}{B} + b\right)\right] \\
&= \left(\left(\frac{A}{\beta^m} - \frac{1}{\beta^{m-n+1}}\right)\left(\frac{\beta^m}{B} - a\right), \frac{A}{\beta^m}\left(\frac{\beta^m}{B} + b\right)\right] \\
&= \left(\frac{A}{B} - \frac{Aa}{\beta^m} - \frac{\beta^{n-1}}{B} + \frac{a}{\beta^{m-n+1}}, \frac{A}{B} + \frac{Ab}{\beta^m}\right] \\
&\subset \left(\frac{A}{B} - (a+1), \frac{A}{B} + b\right] \qquad (3.11)
\end{aligned}$$

since $A < \beta^m$ and $B \geq \beta^{n-1}$.

It is now easy to see that

$$\begin{aligned}
Q_1 &\in \left(\frac{A_1\mu}{\beta^{n+1}} - 1, \frac{A_1\mu}{\beta^{n+1}}\right] \\
&\subset \big(A/B - (a+2), A/B + b\big] \qquad (3.12)
\end{aligned}$$

and since

$$A/B \in \big[\lfloor A/B \rfloor, \lfloor A/B \rfloor + 1\big) \tag{3.13}$$

we have that

$$Q_1 \in \big(\lfloor A/B \rfloor - (a+2), \lfloor A/B \rfloor + (b+1)\big] \tag{3.14}$$

All that is left to note is that every iteration of the loop takes Q_k one step closer to $\lfloor A/B \rfloor$, and according to equation 3.14 we can make at most $a+2$ or $b+1$ steps (whichever is greater) before $Q_k = \lfloor A/B \rfloor$. ∎

Theorem 3.10. If the time it takes to multiply two n-digit numbers is $M(n)$, then algorithm 3.5 runs in $2M(n) + \mathcal{O}(n)$ time.

Proof. We do two multiplications, $A_1\mu$ and BQ_1, that are not mere shifts. Other than that, all we do is obvious $\mathcal{O}(n)$-operations like addition, subtraction, truncation, shifting and comparisons. Theorem 3.9 guarantees that we do only a constant number of them. ∎

Chapter 4

Inversion Algorithms

Barrett's method (algorithm 3.5) requires as input an integer approximation of β^n divided by a large integer N. This problem is exactly the same as computing a limited-precision approximation to $1/N$, except that we have to move the radix point around a bit.

I will present two ways of computing inverses, one obvious and one slightly more far-fetched.

4.1 Inverting with General Division

Given an integer N and a (small) integer n, we can trivially find $\lfloor \frac{\beta^n}{N} \rfloor$ by explicitly constructing the number β^n and then using a general division routine such as algorithm 3.2 or 3.3 (using Barrett's algorithm is not an option, since it needs the inverse we are trying to compute).

Obviously, as long as we can disregard the $\mathcal{O}(n)$ time it takes to construct the number β^n, this inversion algorithm takes exactly as long as the division algorithm we use.

4.2 Newton Inversion

Newton's method is a well-known way to find roots of differentiable functions. Basically, to find a root to $f(x) = 0$, you start with some approximation to the root x_0, and then compute successively better approximations using the rule

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{4.1}$$

Newton's method is somewhat picky about the initial approximations – finding one is usually the hardest part of the entire algorithm – but if it converges at all, it converges quadratically, that is, the number of correct digits doubles with every iteration[1].

[1]With one exception: if the derivative is zero at the root, the convergence is only linear.

By setting $f(x) = 1/x - b$, and thus $f'(x) = -1/x^2$, we get a series that converges to $1/b$:

$$x_{i+1} = x_i + \frac{1/x_i - b}{1/x_i^2} = x_i + x_i - bx_i^2 = 2x_i - bx_i^2 \tag{4.2}$$

Note that there is not any division in the equation any longer; because of this, it can actually be useful.

Equation 4.2 is the foundation of Newton inversion. The trick is to write this in a way that works with integers, and then to prove that the result is always correct. Not surprisingly, the second of these is the more formidable task.

Algorithm 4.1 and its correctness proof are from Knuth[6]. I have changed the algorithm slightly, and made the proof twice as long by spelling it out in a more comfortable level of detail.

Note that this algorithm assumes a base of $\beta = 2$. It is trivial to adapt to any base that is a power of two (which should cover all actual implementations), but for any other base one would have to do some real work to prove that the algorithm still works.

Algorithm 4.1 Newton inversion

Input A real number v such that $\frac{1}{2} \leq v < 1$, and a (small) integer n.

Output An approximation z to $1/v$ such that $|z - 1/v| < 2^{-2^n}$.

```
1. $z_0 \leftarrow \frac{1}{4}\lfloor \frac{32}{4v_1+2v_2+v_3} \rfloor$, where $v_i$ is the $i$th most significant fraction digit of $v$.
2. $k \leftarrow 0$.
3. (a) $s_k \leftarrow z_k^2$.
   (b) $t_k \leftarrow v$ truncated so that exactly $2^{k+1} + 3$ fraction digits still remain.
   (c) $u_k \leftarrow t_k s_k$, truncated so that exactly $2^{k+1} + 1$ fraction digits still remain.
   (d) $w_k \leftarrow 2z_k$.
   (e) $z_{k+1} \leftarrow w_k - u_k$.
   (f) $k \leftarrow k + 1$.
   (g) If $k < n$, goto step 3a. Otherwise, return $z_k$ and terminate.
```

Theorem 4.1. In algorithm 4.1, we have

$$z_i \leq 2 \qquad \text{and} \qquad |z_i - 1/v| < 2^{-2^i} \tag{4.3}$$

for all i.

Proof. Let $\delta_k = 1/v - z_k$; what we want to prove is that $|\delta_k| < 2^{-2^k}$.

Base case Let $v' = 4v_1 + 2v_2 + v_3 = \lfloor 8v \rfloor$. We have

$$\begin{aligned}
\delta_0 = 1/v - z_0 &= 1/v - \frac{1}{4}\lfloor\frac{32}{v'}\rfloor \\
&= 1/v - 8/v' + \frac{32/v' - \lfloor 32/v' \rfloor}{4} \\
&= \frac{v' - 8v}{vv'} + \frac{32/v' - \lfloor 32/v' \rfloor}{4} \qquad (4.4)
\end{aligned}$$

We make the following observations:

- $0 \le \frac{32/v' - \lfloor 32/v' \rfloor}{4} < \frac{1}{4}$ since $32/v' - \lfloor 32/v' \rfloor$ is the fraction part of a positive number.
- $-\frac{1}{2} < \frac{v'-8v}{vv'} \le 0$, since $v' - 8v = \lfloor 8v \rfloor - 8v \in (-1, 0]$ and $vv' = v\lfloor 8v \rfloor \ge \frac{1}{2} \times 4 = 2$.

It follows that $-\frac{1}{2} < \delta_0 < \frac{1}{4}$, so $|\delta_0| < \frac{1}{2} = 2^{-2^0}$ which is what we wanted to make sure.

Inductive case Now assume that $|\delta_k| < 2^{-2^k}$ and $z_k \le 2$. Let $A_k = t_k s_k - u_k$ be the amount truncated in step 3c; clearly, $0 \le A_k < 2^{-2^{k+1}-1}$.

$$\begin{aligned}
\delta_{k+1} = 1/v - z_{k+1} &= 1/v - w_k + u_k \\
&= 1/v - 2z_k + t_k z_k^2 - A_k \\
&= \delta_k - z_k + z_k^2 v - z_k^2 v + t_k z_k^2 - A_k \\
&= \delta_k - z_k(1 - z_k v) - z_k^2(v - t_k) - A_k \\
&= \delta_k - (1/v - \delta_k)v\delta_k - z_k^2(v - t_k) - A_k \\
&= v\delta_k^2 - z_k^2(v - t_k) - A_k \qquad (4.5)
\end{aligned}$$

Since $0 \le v\delta_k^2 \le \delta_k^2 < (2^{-2^k})^2 = 2^{-2^{k+1}}$ and $0 \le z_k^2(v - t_k) + A_k < 4(2^{-2^{k+1}-3}) + 2^{-2^{k+1}-1} = 2^{-2^{k+1}}$, we have that $|\delta_{k+1}| < 2^{-2^{k+1}}$.

The only thing left to verify is that $z_k \le 2$ for all k. It is trivially true for $k = 0$. So assume it is true for z_k, and that $|\delta_k| < 2^{-2^k}$. Then we have three cases:

1. If $t_k = \frac{1}{2}$, then $t_i = \frac{1}{2}$ for $i \le k$. And since $z_0 = 2$, the recursion $z_{i+1} = 2z_i - \frac{1}{2}z_i^2$ says that $z_i = 2$ for $0 \le i \le k+1$.

2. If $t_k > \frac{1}{2}$, but $t_{k-1} = \frac{1}{2}$ so that $z_k = 2$ and $t_k \ge \frac{1}{2} + 2^{-2^{k+1}-3}$, then $2z_k - t_k z_k^2 = 4(1 - t_k) \le 4(\frac{1}{2} - 2^{-2^{k+1}-3}) = 2 - 2^{-2^{k+1}-1}$. Now let A_k be the amount truncated in step 3c, as before. Since $A_k < 2^{-2^{k+1}-1}$, we have $z_{k+1} = 2z_k - t_k z_k^2 + A_k < 2$.

3. If $t_{k-1} > \frac{1}{2}$, then $v \geq \frac{1}{2} + 2^{-2^k-3}$ so that $1/v < 2 - 2^{-2^k-1} + 2^{-2^{k+1}-3}$ (see theorem B.2). Since $|\delta_{k+1}| < 2^{-2^{k+1}}$ means that $\delta_{k+1} > -2^{-2^{k+1}}$, we have

$$\begin{aligned} z_{k+1} = 1/v - \delta_{k+1} &< 2 - 2^{-2^k-1} + 2^{-2^{k+1}-3} + 2^{-2^{k+1}} \\ &= 2 - 2^{-2^k-1} + 9 \cdot 2^{-2^{k+1}-3} \\ &= 2 - 2^{-1}2^{-2^k} + 9 \cdot 2^{-3}\left(2^{-2^k}\right)^2 \\ &= 2 - \frac{1}{2}\left(2^{-2^k} - \frac{9}{4}\left(2^{-2^k}\right)^2\right) \\ &= 2 - \frac{1}{2}2^{-2^k}\left(1 - \frac{9}{4}2^{-2^k}\right) \\ &< 2 \end{aligned} \tag{4.6}$$

for $k \geq 1$.

The observant reader will have noticed that these are not in fact two unrelated induction proofs; as stated in the 'Inductive case' steps, the dependencies are as follows:

- $|\delta_k| < 2^{-2^k}$ and $z_k \leq 2$ together imply $|\delta_{k+1}| < 2^{-2^{k+1}}$.
- $|\delta_{k+1}| < 2^{-2^{k+1}}$ and $z_k \leq 2$ together imply $z_{k+1} \leq 2$.

(The base cases do not have any such dependencies.) Clearly, if we prove that $|\delta_{k+1}| < 2^{-2^{k+1}}$ before we prove that $z_{k+1} \leq 2$, we will be OK. ■

4.3 Newton Inversion, Again

Algorithm 4.2 is basically the same as algorithm 4.1, but I have generalized it slightly to make it more suited for translation to program code. In particular, there are three major improvements (in order of decreasing importance and increasing difficulty to prove correct):

1. The base case does not involve only the three most significant bits of the operand, but a much larger and variable number of bits.
2. There is the option of increasing the number of fraction digits by one less in each iteration of the loop.
3. As long as certain rules are obeyed, there is no harm in keeping a few more fraction digits around than is strictly necessary.

Before the algorithm starts, one has to determine the constants k_i. The only restrictions are that for all i, either $k_{i+1} = 2k_i$ or $k_{i+1} = 2k_i - 1$, and that $k_i = n$ for some i. The idea is that they be computed as follows: $k_I = n$; then for every

Algorithm 4.2 Newton inversion, again

```
Input  A real number $v$ with $m$ fraction digits, such that $\frac{1}{2} \le v < 1$, and a (small)
       integer $n$.

Output  An approximation $z$ to $1/v$ such that $|z - 1/v| \le 2^{-n}$.

1. $i \leftarrow 0$
2. $z_0 \leftarrow 1/v$, truncated so that at least $k_0$ fraction digits remain. (Use some other
   inversion algorithm, such as schoolbook or divide-and-conquer division.)
3. (a) If $k_i = n$, return $z_i$ and terminate.
   (b) $s_i \leftarrow z_i^2$.
   (c) $t_i \leftarrow v$ truncated so that exactly $2k_i + 3 + h_i$ fraction digits still remain,
       $0 \le h_i \le 2k_i$.
   (d) $u_i \leftarrow t_i s_i$, truncated so that at least $2k_i + 1 + h_i$ fraction digits still remain.
   (e) $w_i \leftarrow 2z_i$.
   (f) $z_{i+1} \leftarrow w_i - u_i$.
   (g) $i \leftarrow i + 1$
   (h) Goto step 3a.
```

$i \ge 0$, $k_i = \lceil k_{i+1}/2 \rceil$. The number I is chosen such that k_0 is a suitable size for the basecase algorithm.

This scheme is used in place of simply doubling k, so that we minimize operand length at every iteration of the loop instead of simply doubling it until we have at least as much precision as was asked for, then truncate.

For example, say the required precision is 41000 bits and the breakeven point below which the basecase algorithm is faster is 1000 bits. The simple approach is to do a basecase of 1000 bits, then double the precision in each iteration of the loop until it reaches 64000 bits, and last of all truncate to 41000 bits. The smart approach is to calculate the k_i as above; see table 4.1. In both cases we have the base case and six iterations of the loop, but the precision required in each step is significantly less in the second case.

The reason for allowing h_i extra fraction digits in the truncation steps is convenience; if one machine word holds g bits, then it is possible to always round the number of fraction digits up to an integer multiple of g when truncating, which makes life simpler.

Theorem 4.2. In algorithm 4.2, we have

$$z_i \le 2 \qquad \text{and} \qquad |z_i - 1/v| < 2^{-k_i} \tag{4.7}$$

for all i.

	Simple	Smart
k_0	1000	641
k_1	2000	1282
k_2	4000	2563
k_3	8000	5125
k_4	16000	10250
k_5	32000	20500
k_6	64000	41000

Table 4.1. Precision in each iteration of the Newton loop using the simple and smart values of k_i.

Proof. Let $\delta_i = 1/v - z_i$; what we want to prove is that $|\delta_i| < 2^{-k_i}$.

Base case For $i = 0$, we have $1/v - 2^{-k_0} < z_0 \leq 1/v$ after step 2. This means that $\delta_0 = 1/v - z_0 < 2^{-k_0}$, and $\delta_0 \geq 0$. So $|\delta_0| \leq 2^{-k_0}$, as required.

Inductive case Assume that $|\delta_i| < 2^{-k_i}$. Let $A_i = t_i s_i - u_i$ be the amount truncated in step 3d; clearly, $0 \leq A_i < 2^{-2k_i-1}$. From equation 4.5 (modified slightly), we have that $\delta_{i+1} = v\delta_i^2 - z_i^2(v - t_i) - A_i$.

Since $0 \leq v\delta_i^2 \leq \delta_i^2 < (2^{-k_i})^2 = 2^{-2k_i}$ and $0 \leq z_i^2(v - t_i) + A_i < 4 \cdot 2^{-2k_i-3} + 2^{-2k_i-1} = 2^{-2k_i}$, we have that $|\delta_{i+1}| < 2^{-2k_i} \leq 2^{-k_{i+1}}$.

The only thing left to verify is that $z_i \leq 2$ for all i. It is trivially true for $i = 0$ since $v \geq \frac{1}{2}$. So assume it is true for z_i. Then we have three cases:

1. If $t_i = \frac{1}{2}$, then $t_j = \frac{1}{2}$ for all $j \leq i$. And since $z_0 = 2$, the recursion $z_{j+1} = 2z_j - \frac{1}{2}z_j^2$ says that $z_j = 2$ for $0 \leq j \leq i+1$, so $z_{i+1} = 2$.

2. If $t_i > \frac{1}{2}$, but $t_{i-1} = \frac{1}{2}$ so that $z_i = 2$ and $t_i \geq \frac{1}{2} + 2^{-2k_i-3-h_i}$, then $2z_i - t_i z_i^2 = 4(1 - t_k) < 4(\frac{1}{2} - 2^{-2k_i-3-h_i}) = 2 - 2^{-2k_i-1-h_i}$. Now let $A_i = t_i s_i - u_i$, as before. Since $A_i < 2^{-2k_i-1-h_i}$, we have $z_{i+1} = 2z_i - t_i z_i^2 + A_i < 2$.

3. If $t_{i-1} > \frac{1}{2}$, then $v \geq \frac{1}{2} + 2^{-2k_{i-1}-3-h_{i-1}}$ so that $1/v < 2 - 2^{-2k_{i-1}-1-h_{i-1}} + 2^{-4k_{i-1}-3-2h_{i-1}}$ (see theorem B.2). Since $|\delta_{i+1}| < 2^{-k_{i+1}}$ means that $\delta_{i+1} > -2^{-k_{i+1}}$, we have

$$\begin{aligned} z_{i+1} &= 1/v - \delta_{i+1} \\ &< 2 - 2^{-2k_{i-1}-1-h_{i-1}} + 2^{-4k_{i-1}-3-2h_{i-1}} + 2^{-k_{i+1}} \\ &\leq 2 - 2^{-2k_{i-1}-1-h_{i-1}} + 2^{-4k_{i-1}-3-2h_{i-1}} + 2^{-4k_{i-1}-3} \\ &= 2 - 2^{-2k_{i-1}-1-h_{i-1}}(1 - 2^{-2k_{i-1}-2-h_{i-1}} - 2^{-2k_{i-1}-2+h_{i-1}}) \\ &< 2 \end{aligned} \tag{4.8}$$

when $h_i \leq 2k_i$.

■

Theorem 4.3. If the time it takes to multiply two n-digit numbers is $M(n) \in \Theta(n^c)$ for some $c > 1$, then algorithm 4.2 takes at most $\left(1 + \frac{2}{2^c-1}\right) M(n) + \mathcal{O}(n)$ time.

If $M(n) \in \mathcal{O}(n^{1+\epsilon})$ for arbitrarily small ϵ, then algorithm 4.2 takes at most $3M(n) + \mathcal{O}(n)$ time.

Proof. We make $\log_2 n$ iterations of the loop. Each of them takes $M(k/2) + M(k) + \mathcal{O}(k)$ time. This sums to

$$\begin{aligned}
\sum_{i=1}^{\log_2 n} \left(M(2^{i-1}) + M(2^i) + \mathcal{O}(2^i)\right) &= \mathcal{O}(n) + \sum_{i=1}^{\log_2 n} \left(M(2^{i-1}) + M(2^i)\right) \\
&= \mathcal{O}(n) + M(1) + M(n) + 2 \sum_{i=1}^{\log_2 n-1} M(2^i) \\
&= \mathcal{O}(n) + M(n) + 2 \sum_{i=1}^{\log_2 n-1} M(2^i) \qquad (4.9)
\end{aligned}$$

If $M(n) = Cn^c$, the sum is

$$\begin{aligned}
\sum_{i=1}^{\log_2 n-1} M(2^i) &= \sum_{i=1}^{\log_2 n-1} C(2^i)^c \\
&= 2C \sum_{i=1}^{\log_2 n-1} (2^c)^i \\
&= 2C \frac{2^c(1-(n/2)^c)}{1-2^c} \\
&= M(n)\frac{2}{2^c-1} - 2C\frac{2^c}{2^c-1} \qquad (4.10)
\end{aligned}$$

so the time is

$$\mathcal{O}(n) + \left(1 + \frac{2}{2^c-1}\right) M(n) \qquad (4.11)$$

Table 4.2 lists how many times slower than multiplication Newton inversion is asymptotically for some particularly interesting values of c. Figure 4.1 shows the same thing for all interesting values of c.

Note that in the degenerate case $c = 1$, $M(n) \in \mathcal{O}(n)$ and thus the factor 3 is not important since we have disregarded other linear factors (however, if C is sufficiently large compared to the factor hidden in the $\mathcal{O}(n)$ term, 3 will not be such a bad guess).

If $M(n) \in \mathcal{O}(n^{1+\epsilon})$ for arbitrarily small ϵ (the examples $n \log^k n$ and $n \log^k n \log\log n$ come to mind), then it can be approximated by $Cn^{1+\epsilon(n)}$, where $\epsilon(n) \to 0$ as $n \to \infty$. Asymptotically, the behavior will be identical to the case $M(n) = \Theta(n)$, but the

c	Number of multiplications
1	3.000
$\log_3 5$	2.136
$\log_2 3$	2.000
2	1.667

Table 4.2. The execution time of algorithm 4.2 relative to that of multiplication when the multiplication time is $\Theta(n^c)$.

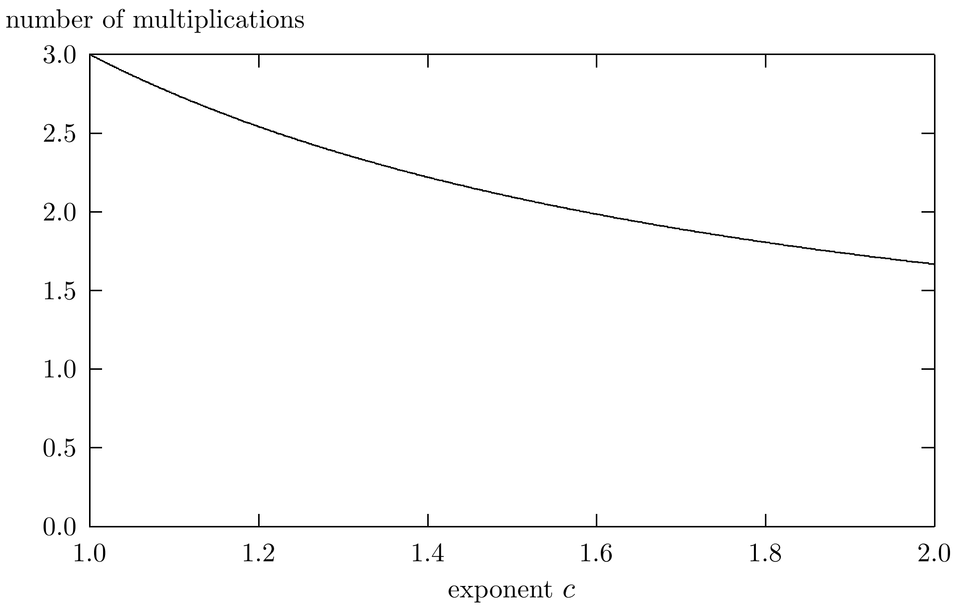

Figure 4.1. The relation between multiplication time and inversion time (with algorithm 4.2) when the multiplication time is $\Theta(n^c)$.

convergence may be much slower. Ironically, in this case we do not have to worry about the $\mathcal{O}(n)$ term, since we only approach linear time, without ever getting there.

■

4.3.1 Newton Division

Combining Newton inversion with Barrett's method gives a division algorithm that takes precisely two multiplications extra time compared to Newton inversion alone.

Figure 4.2 is a combination of figures 4.1 (with two multiplications added to the time for Barrett's algorithm) and 3.1. It indicates that for multiplication time $M(n) \in \mathcal{O}(n^c)$, with c less than about 1.3, Newton division is asymptotically superior to divide-and-conquer division. This figure is actually $c = \log_2(3/2+\sqrt{11/12}) \approx$

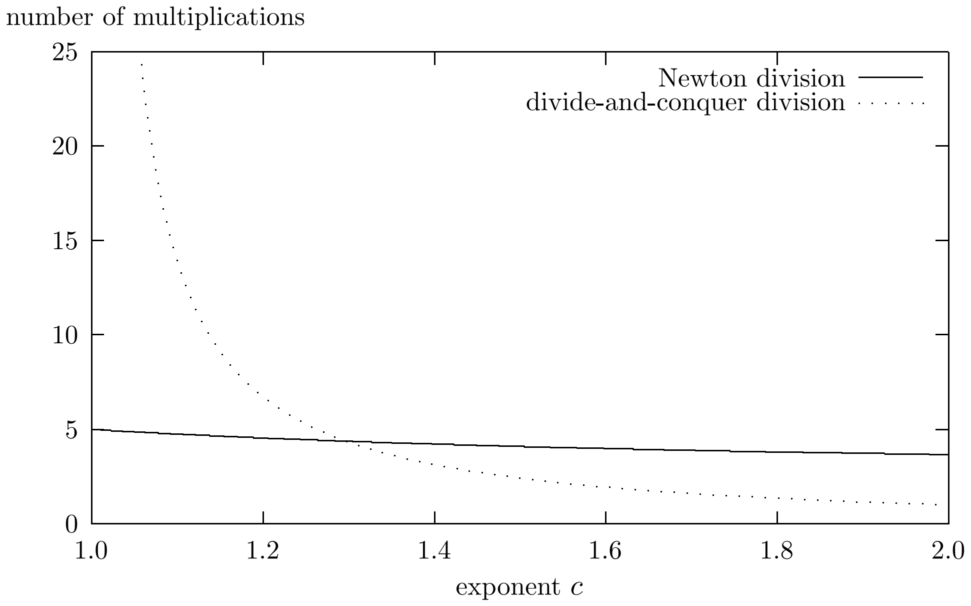

Figure 4.2. The relation between multiplication time and division time when the multiplication time is $\Theta(n^c)$.

1.297; it is the solution of the equation

$$\frac{1}{2^{c-1}-1} = 3 + \frac{2}{2^c - 1} \tag{4.12}$$

(The left hand side is from equation 3.8, the right hand side is from equation 4.11 plus 2 for Barrett's algorithm.)

If we want to do k divisions using the same divisor (but with different dividends), we can get away with less work than simply k times one Newton division, since we need only invert the divisor once. This moves the breakpoint to

$$c = \log_2 \frac{8k+1+\sqrt{16k^2+8k+9}}{4k+2} \tag{4.13}$$

which is the solution to the equation

$$\frac{k}{2^{c-1}-1} = 2k + 1 + \frac{2}{2^c - 1} \tag{4.14}$$

When k increases from one, c decreases quite rapidly at first, but already at $k = 100$ it is almost $\log_2 3$, the value on which it converges as $k \to \infty$ (see table 4.3). It is no coincidence that c converges to the same exponent we have in Karatsuba multiplication, since that same exponent makes the divide-and-conquer algorithm require exactly two multiplications per division; Barrett's algorithm always takes

k	c
1	1.297
2	1.402
3	1.452
4	1.481
5	1.499
10	1.540
100	1.580
1000	1.584
10000	1.585

Table 4.3. For k divisions by the same divisor, the constant c such that multiplication in time $\mathcal{O}(n^c)$ makes Newton and divide-and-conquer division equally fast asymptotically.

precisely two multiplications per division, and the inversion can be neglected since we only do it once.

Using the same divisor several times is not just a toy problem. When evaluating $a^b \bmod c$ for large integers a, b and c, one efficient method is repeated squaring and modular reduction; the modular reduction consists of finding the remainder of different numbers divided by c. Such powers occur in RSA cryptography (see, for example, [8] where it was first introduced). Similarly, when dividing kn digits by n digits, we essentially do k divisions with different $2n$-digit dividends and a constant n-digit divisor.

Newton division is the fastest known division algorithm, given that we use a fast enough multiplication algorithm, such as FFT (or Toom-9 or higher; see table A.1); in particular, it is asymptotically faster than divide-and-conquer division by more than just a constant factor, and asymptotically 5 times slower than multiplication. If we do several divisions by the same divisor, Newton division is even more favorable; for example, using Toom-3 multiplication, Newton division is asymptotically superior for four or more divisions. Divide-and-conquer division always wins if we use Karatsuba multiplication (or slower), though.

Chapter 5

Which Division Algorithm Is Fastest?

Are Newton inversion and Barrett's method superior to Burnikel and Ziegler's recursive algorithm in practice? I have built rather optimized Newton and Barrett algorithms on top of the low-level interface to GMP. The Burnikel-Ziegler algorithm is a part of GMP (built and tuned by far better programmers than myself), so I did not have to make that as well.

5.1 Implementing Newton Inversion and Barrett's Algorithm

I implemented both algorithms in plain C, a rather low-level high-level language. My approach to making them run fast is nothing revolutionary, just a few rules of thumb:

- Never allocate memory dynamically; I demand a pointer to one big chunk of scratch space as input to the algorithm, and then use that exclusively, being careful to reuse space when two temporary variables are not live simultaneously.
- Do magic with pointers to avoid copying (parts of) operands. Numbers are represented with a pointer to the least significant digit and a small integer that says how many digits there are; for example, truncating can be done by decreasing the integer and advancing the pointer.
- The non-integer numbers in Newton inversion are implemented with integer operations just like everything else; all I have to do is keep track of where the radix point is, and multiply or divide numbers with powers of β to align their radix points before I add or subtract them, but this is all just more pointer juggling.
- Call optimized GMP subroutines for all my arithmetic needs; most of these work on operands in-place, avoiding the need to copy big operands.

Bits	Algorithm
– 800	Schoolbook
801 – 6016	Karatsuba
6017 – 237536	Toom-3
237537 –	FFT

Table 5.1. Ranges for different multiplication algorithms on the test machine.

- Do *not* use assembly language or obscure tricks to make fast constant-time operations go even faster, since the GMP subroutines hide all the performance-critical inner loops where such extreme measures would be called for.

The idea is, simply put, to make sure that all the time is spent in calls to GMP subroutines, by not doing expensive things that are not necessary. This works very well for Newton inversion and Barrett's algorithm because, as I wrote above, the inner loops are hidden inside the GMP subroutines. This is clearly true in Barrett's algorithm, since it deals exclusively with numbers as large as its inputs; as for Newton inversion, it does start small, but virtually all time is spent in the last few iterations where numbers are almost as large as the input, so the first few iterations, which are the only ones possibly not totally dominated by subroutine calls, can be neglected.

5.2 Test Runs

The computer used for these test runs was a 1467 MHz AMD Athlon. For multiplication of two n-bit numbers, table 5.1 shows which algorithm was used for which range of n.

5.2.1 Comparison of Newton Inversion, Barrett's Algorithm and Divide-and-Conquer Division

Figures 5.1 through 5.3 show the execution time for Newton inversion, Barrett's algorithm and divide-and-conquer division[1] for a wide range of operand sizes[2] (multiplication is also plotted for comparison). Note that when the operand size is given as n bits, that means a division of $2n$ by n bits, or a multiplication of n by n bits. For small operands, all three algorithms seem to take about twice as long as multiplication, but past about 10^4 bits (where Toom-3 kicks in) divide-and-conquer division starts taking noticeably more time, and past a few million bits (where FFT multiplication kicks in) Newton inversion starts taking a little bit more time than Barrett's method.

[1]Below 2817 bits, GMP actually uses schoolbook division for this machine, since it is faster.

[2]Actually, they show almost the *entire* range of reasonable operand sizes for this particular computer. The limiting factor is the amount of available memory, most of which is eaten by inefficient memory use in the FFT multiplication routine.

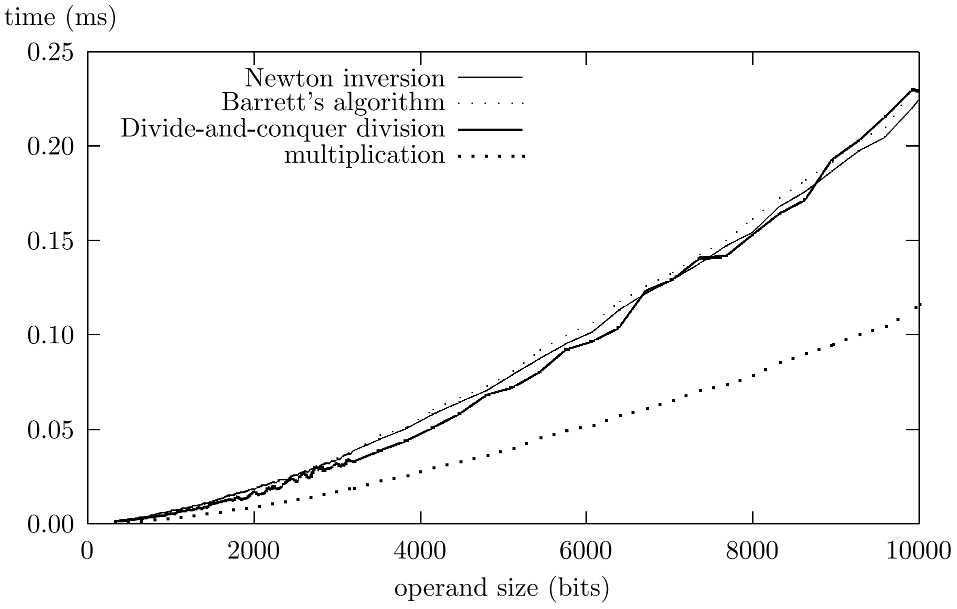

Figure 5.1. Execution time in seconds, small operands.

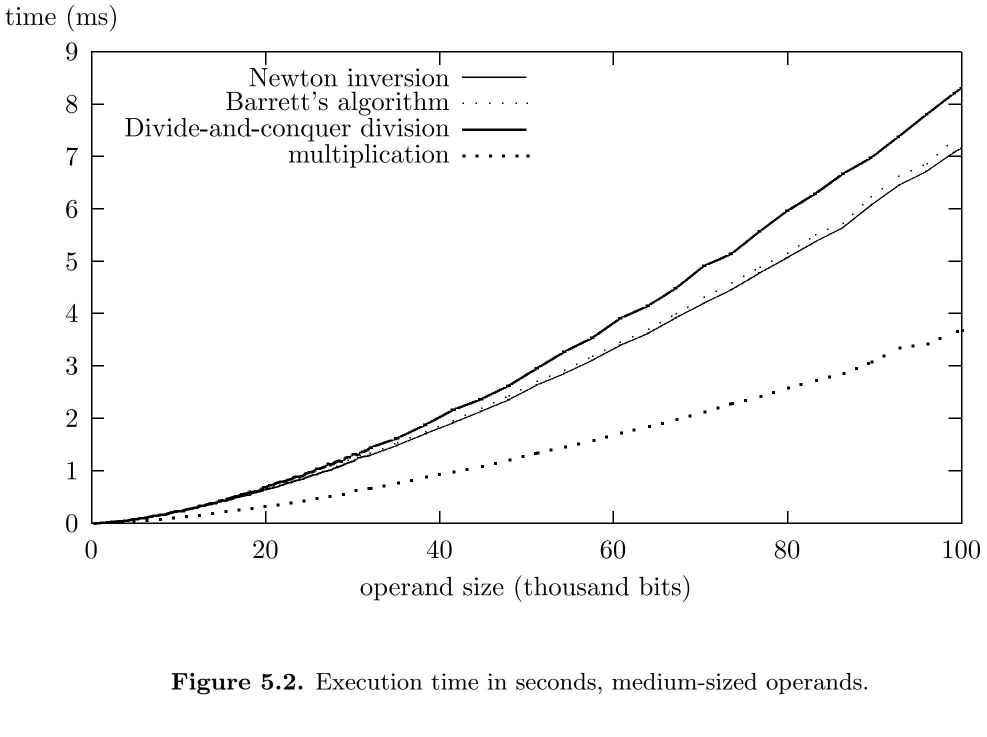

Figure 5.2. Execution time in seconds, medium-sized operands.

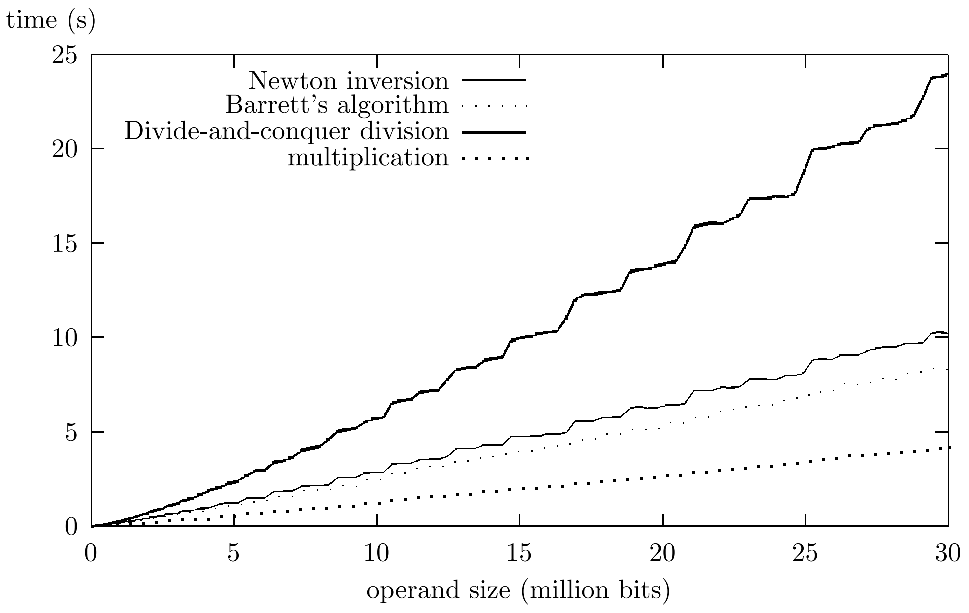

Figure 5.3. Execution time in seconds, large operands.

Figures 5.4 through 5.6 try to make things a bit more clear by giving execution times as a multiple of the multiplication time for that operand size, instead of in seconds. This makes a lot of sense since all three algorithms spend virtually all their time in the multiplication routine.

For small operands, we see that divide-and-conquer division is slightly faster than the others, but the difference is diminishing. From about 7000 bits on, when the multiplication routine switches to Toom-3, it is slower.

Barrett's algorithm takes twice as long as multiplication as long as the operands are not very small. The same goes for Newton inversion, until a few million bits. After that, it increases slowly and then seems to stabilize (more or less) at 2.5 multiplications.

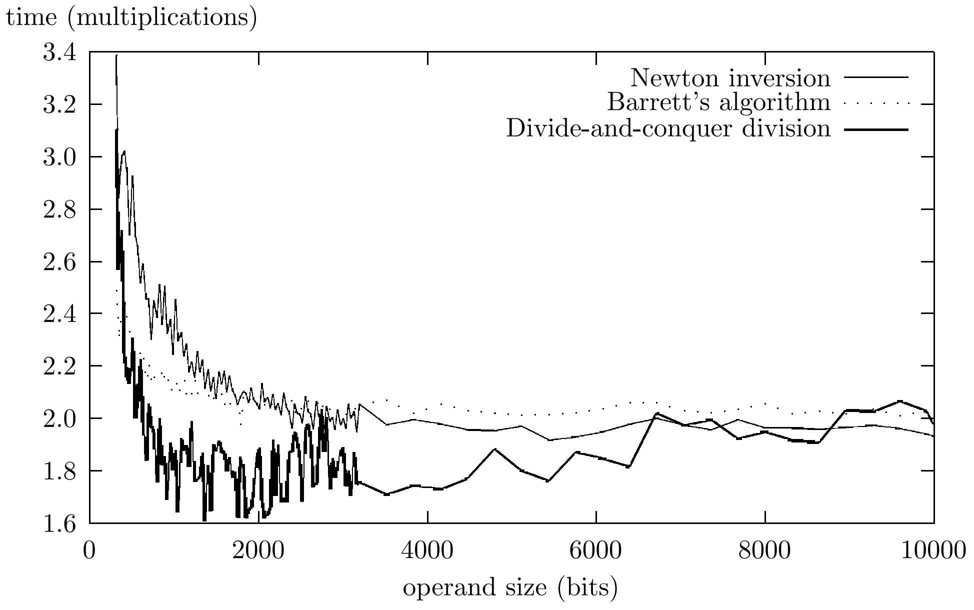

Figure 5.4. Execution time in multiples of multiplication time, small operands.

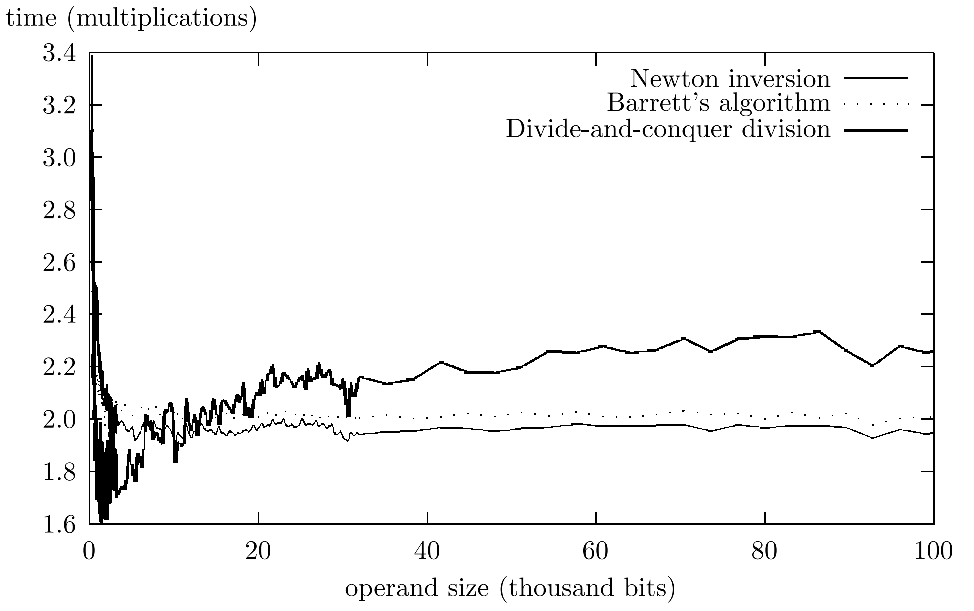

Figure 5.5. Execution time in multiples of multiplication time, medium-sized operands.

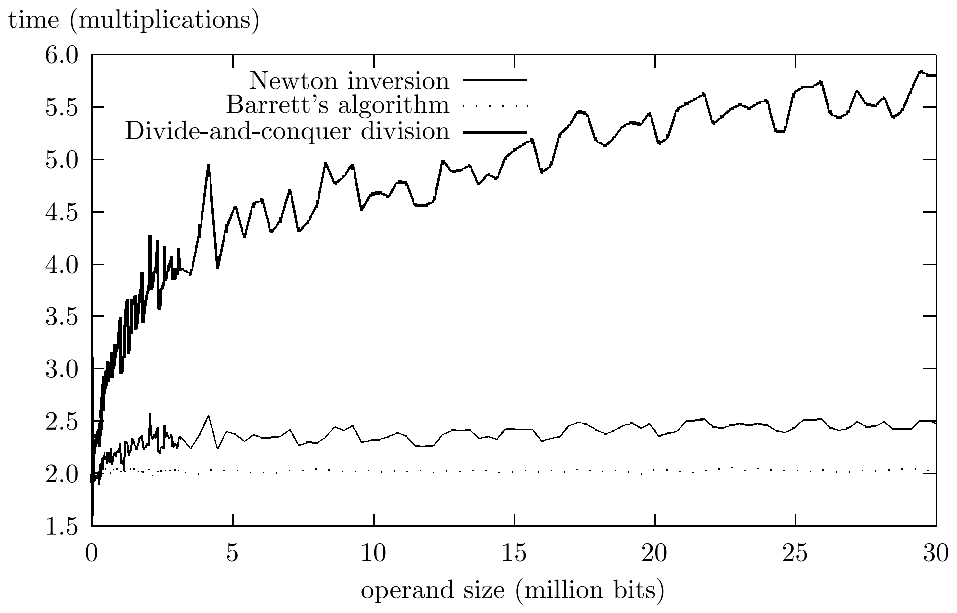

Figure 5.6. Execution time in multiples of multiplication time, large operands.

5.2.2 Divide-and-Conquer versus Newton Division

The comparisons in the previous section are interesting and all, but there is one question they do not answer: can Newton division beat divide-and-conquer division? I ran some tests; the result is shown in figures 5.7 through 5.9, still with times relative to multiplication time.

As one would expect from figures 5.4 through 5.6, Newton division takes four times the multiplication time for small operands, and about 4.5 times for large operands. Divide-and-conquer division is the same as before, steadily increasing with operand size. The breakeven point seems to be at around $5 \cdot 10^6$ bits, or 1.5 million decimal digits.

There is one feature of Newton division that we have not yet exploited, however: if we want to use the same divisor for several divisions, we need only invert it once. So for k divisions, we use Newton inversion once and Barrett's algorithm k times; this is compared with k runs of divide-and-conquer division for $k = 2$ (figure 5.10), $k = 10$ (figure 5.11) and $k = 100$ (figure 5.12).

The breakpoint moves steadily downwards as the number of divisions increases; for $k = 100$, it seems to lie at about 15000 bits, but just one extra division was enough to bring it down from five to just below one million bits. When $k \to \infty$, the asymptotic behavior can be seen by comparing divide-and-conquer division and Barrett's algorithm directly (as in figures 5.4 through 5.6), since the cost of inverting the divisor is negligible.

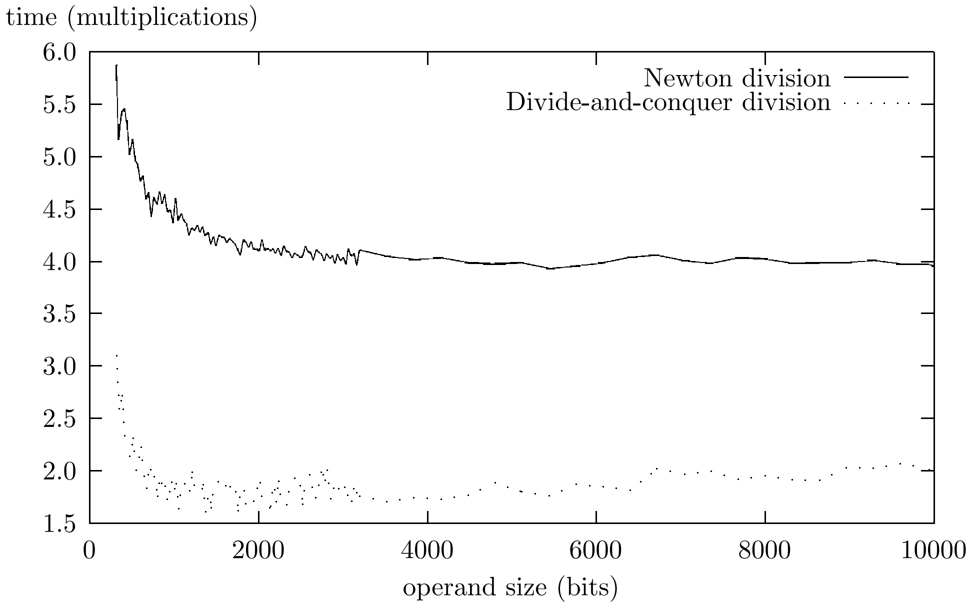

Figure 5.7. Execution time in multiples of multiplication time, small operands.

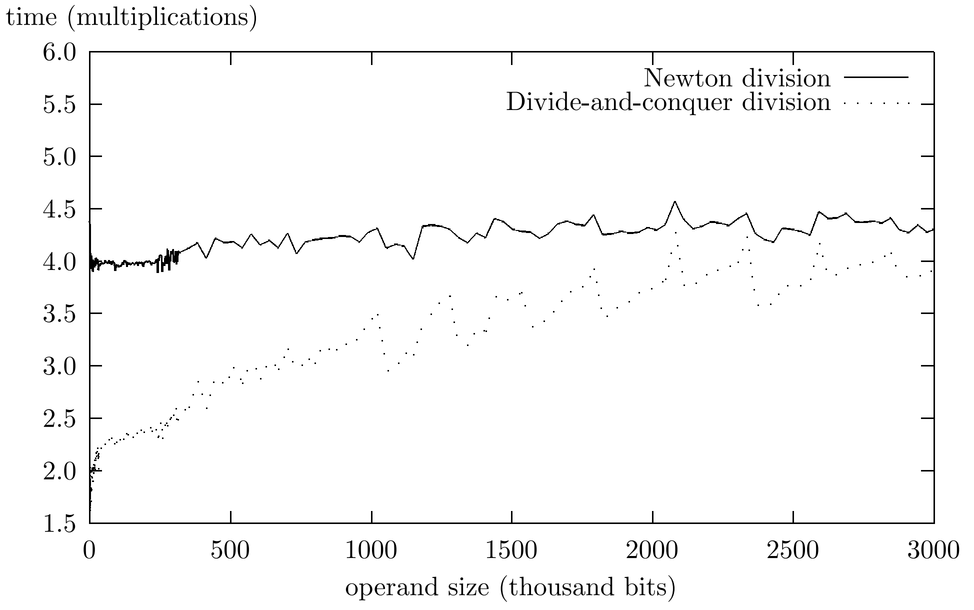

Figure 5.8. Execution time in multiples of multiplication time, medium-sized operands.

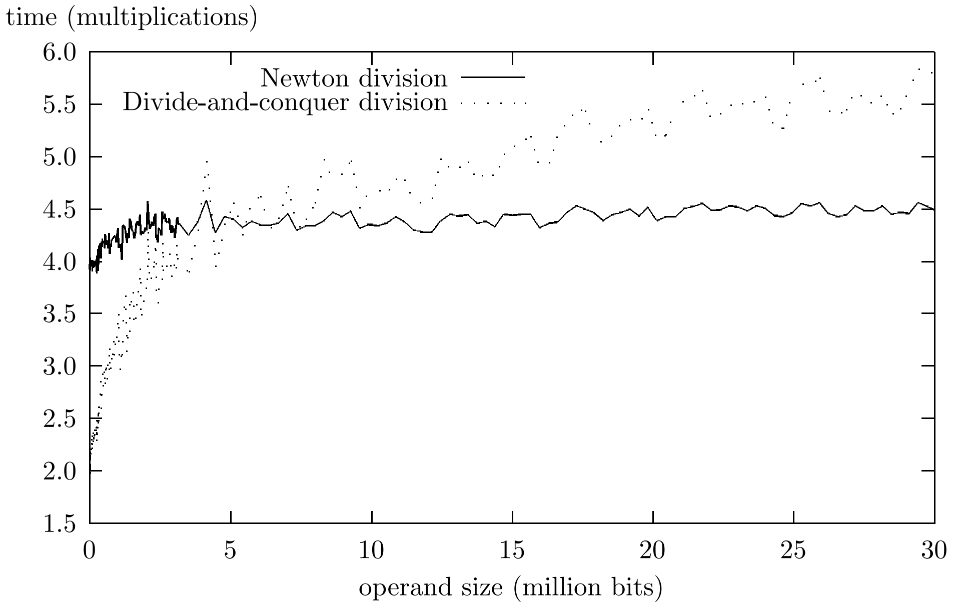

Figure 5.9. Execution time in multiples of multiplication time, large operands.

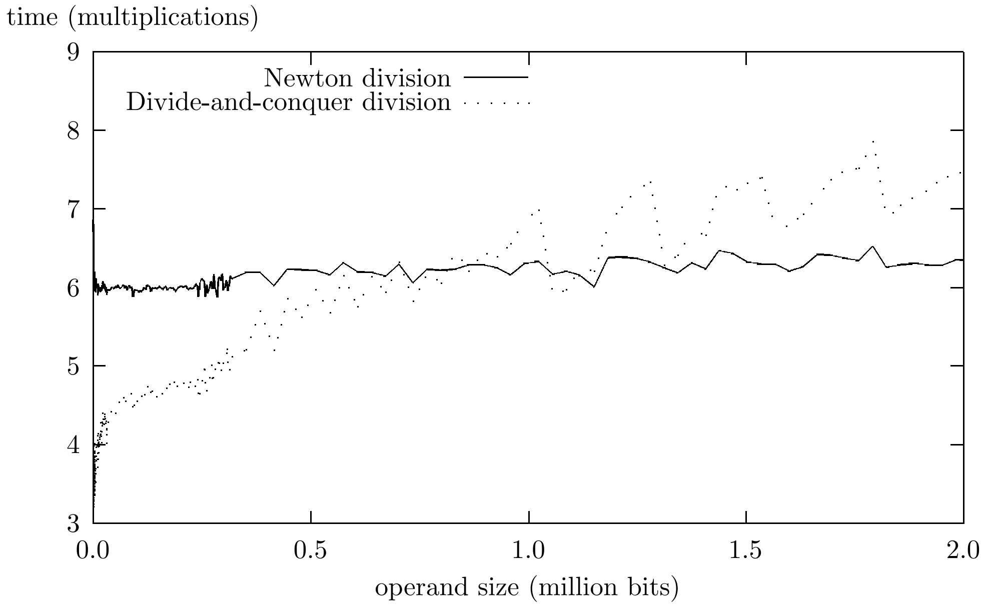

Figure 5.10. Two divisions with same divisor.

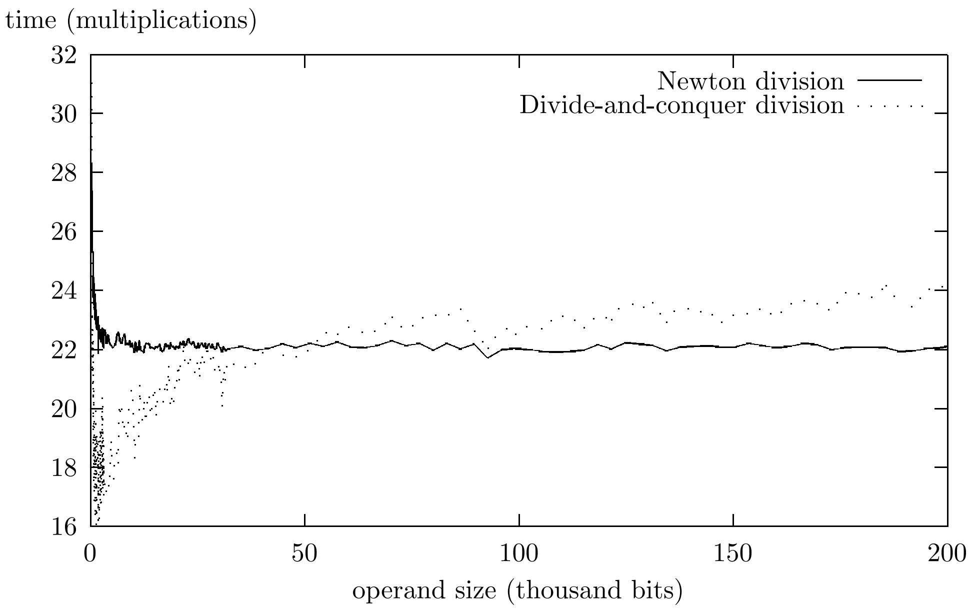

Figure 5.11. Ten divisions with same divisor.

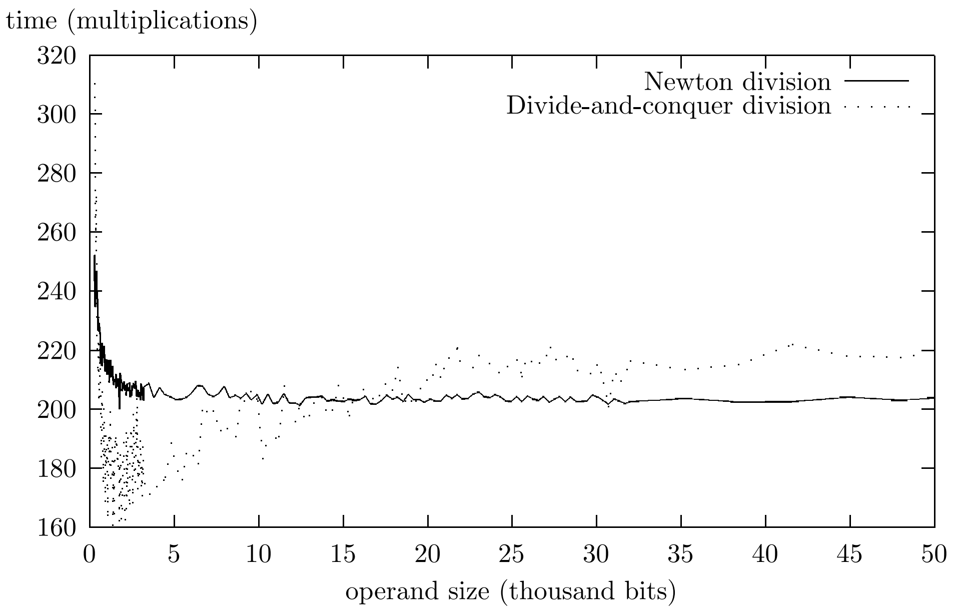

Figure 5.12. A hundred divisions with same divisor.

5.2.3 Newton versus Divide-and-Conquer Inversion

Since repeated division with the same divisor forces the breakeven point between divide-and conquer and Newton division as low as maybe 10^4 bits, we cannot be sure that Newton inversion is faster than computing the inverse using divide-and-conquer division (see section 4.1). Indeed, for operands small enough that schoolbook multiplication is the algorithm of choice, theorem 3.6 says that inversion should take about $M(n)$ time, which is almost certain to be faster than Newton inversion.

Figures 5.13 through 5.15 show the situation. The breakpoint is at about 10^4 bits, which is pretty much the same as the asymptotic breakpoint between Newton and divide-and-conquer division. This means that even though inversion by division is not useful on this computer with this exact software, a small perturbation of the relative speed of things is enough to make it so.

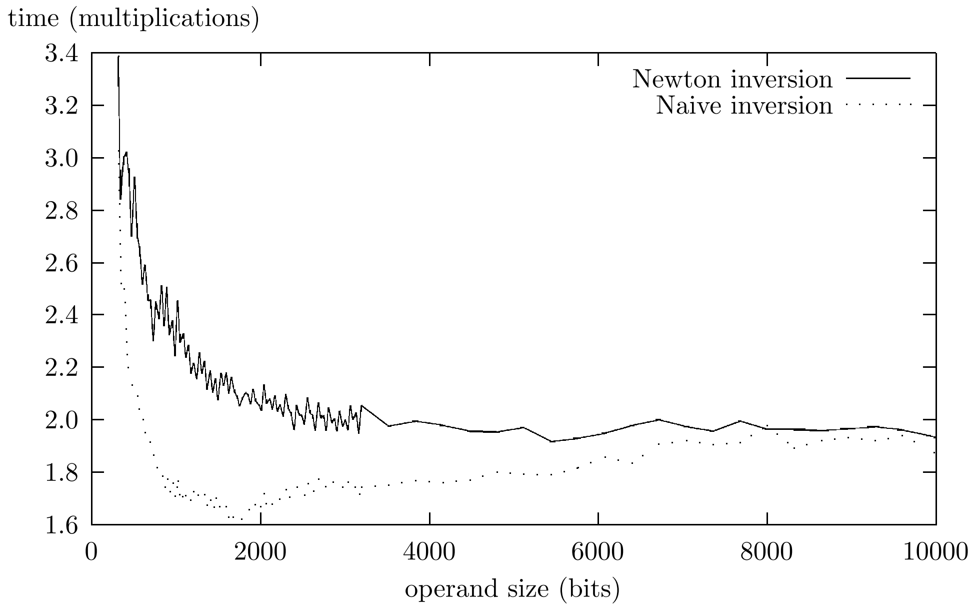

Figure 5.13. Execution time in multiples of multiplication time, small operands.

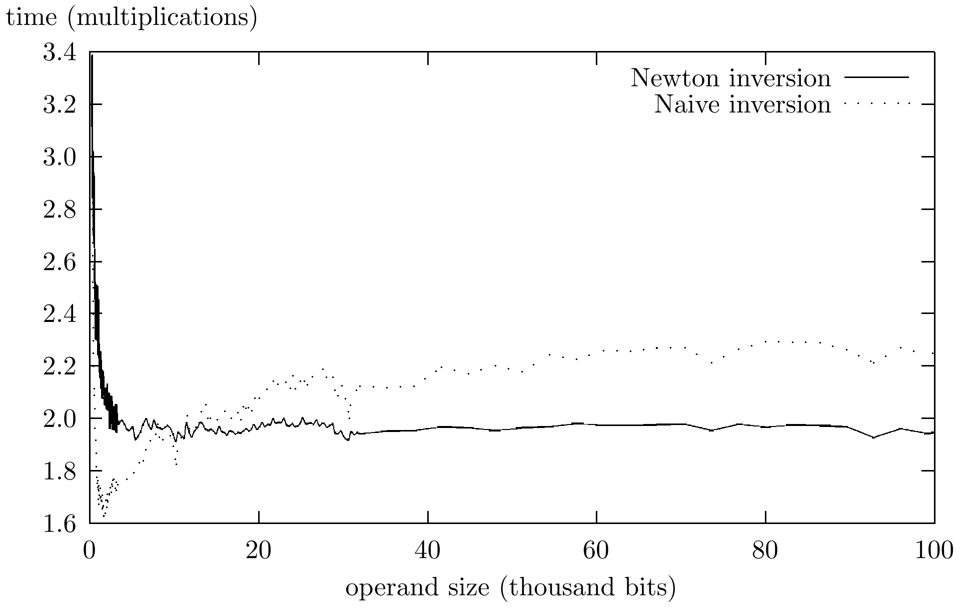

Figure 5.14. Execution time in multiples of multiplication time, medium-sized operands.

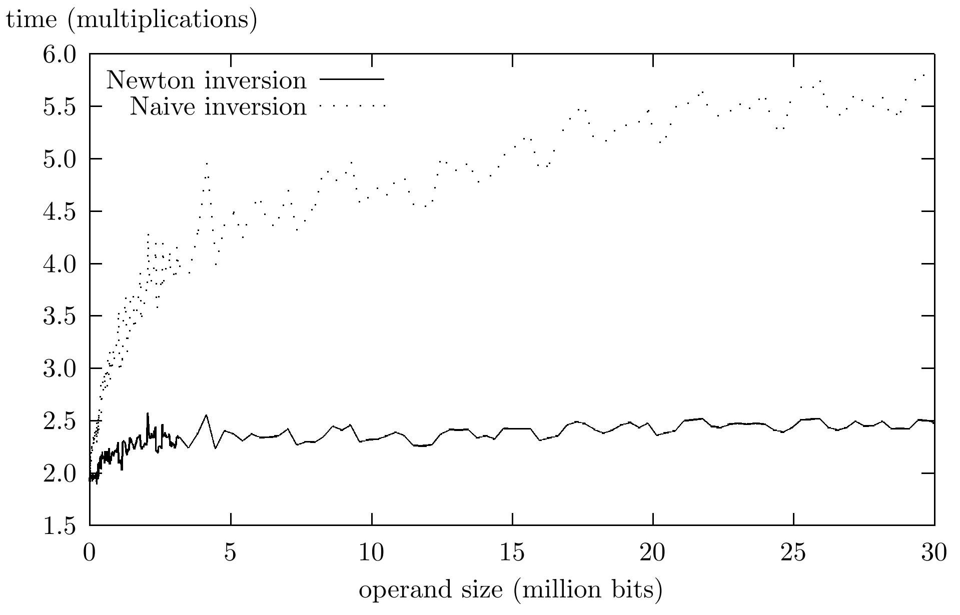

Figure 5.15. Execution time in multiples of multiplication time, large operands.

5.3 Conclusions

I think I may safely say that theory and experiments did not clash too badly.

The observed behavior of divide-and-conquer division (algorithm 3.3) in figures 5.4 through 5.6 coincides remarkably well with the behavior predicted by theorem 3.6; the same is true for Barrett's algorithm (algorithm 3.5, figures 5.4 through 5.6, theorem 3.10). Newton inversion (algorithm 4.2) can also be seen (in figures 5.4 through 5.6) to be conforming to theory (theorem 4.3), although at first glance one would guess that it was going to converge to something like 2.5 multiplications, whereas the theory says 3 (it certainly fooled me, for one).

Especially when looking at figures 5.1 and 5.4, it seems that the difference in running time between Newton inversion, Barrett's algorithm and divide-and-conquer division is almost too small; they are all supposed to run in time $2M(n)+\mathcal{O}(n)$ since Karatsuba multiplication is the multiplication algorithm of choice for that operand size, true, but ... *that* small a difference? They are totally different algorithms, after all, so why do not implementation differences disrupt the predicted similarity in running time?

The likely explanation is that although they are different, what I wrote above about Newton inversion and Barrett's algorithm is largely true for divide-and-conquer division as well: virtually all time is spent in GMP subroutines, specifically the multiplication subroutine, so the parts of the algorithms that take time are actually nearly identical.

On the practical side, Barrett's algorithm is certainly useful for those cases where we have several divisions with the same divisor; and Newton inversion is the inversion method of choice for all but the smallest numbers. Newton division when we cannot reuse the inverted divisor is a more dubious bet, though; five million bits is awfully large. However, given that computers keep getting faster[3], five million bits will not be as much in a few years as it is today. Moreover, seemingly minor tinkering with the program code can move the breakeven points between different algorithms by miles; one modification that is sure to invalidate my measurements is the total rewrite of GMP's FFT multiplication that is being planned.

[3]And that algorithm complexity is almost linear – if not for that, hardware improvements would not mean nearly as much.

References

[1] Paul Barrett. Implementing the Rivest, Shamir and Adleman public-key encryption algorithm on a standard digital signal processor. In *Advances in cryptology: CRYPTO '86: proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer-Verlag, 1987.

[2] Christoph Burnikel and Joachim Ziegler. *Fast Recursive Division.* Research Report MPI-I-98-1-022, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, October 1998.

[3] Torbjörn Granlund et al. *GNU Multiple Precision Arithmetic Library 4.1.2*, December 2002. `http://swox.com/gmp/`.

[4] Anatolii Karatsuba and Yu Ofman. Multiplication of multidigit numbers on automata. *Doklady Akademii Nauk SSSR*, 145(2):293–294, 1962.

[5] Anatolii Karatsuba and Yu Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics-Doklady*, 7(7):595–596, 1963. Translated from [4].

[6] Donald E. Knuth. *The Art of Computer Programming II: Seminumerical Algorithms.* Addison–Wesley, Reading, Massachusetts, second edition, 1981.

[7] Colin Percival. Rapid multiplication modulo the sum and difference of highly composite numbers. *Math. Comp.*, 72:387–395, 2003.

[8] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[9] Arnold Schönhage and Volker Strassen. Schnelle Multiplikation großer Zahlen. (German) [Fast multiplication of large numbers]. *Computing*, 7(3–4):281–292, 1971.

[10] Chee Yap and Chen Li. *QuickMul: Practical FFT-based Integer Multiplication.* Technical report, Department of Computer Science, Courant Institute, New York University, October 2000.

Appendix A

Multiplication Algorithms

The text in this chapter is largely based on Knuth[6].

A.1 Karatsuba's Algorithm

We want to compute the product of the two n-digit nonnegative integers x and y. Assume that n is a power of two, and write $x = a + \beta^{n/2}b$ and $y = c + \beta^{n/2}d$, where a, b, c and d are four nonnegative $n/2$-digit numbers. (We have simply divided the digits of x and y into an upper and a lower half.)

Now the product can be written as $xy = (a+\beta^{n/2}b)(c+\beta^{n/2}d) = ac+\beta^{n/2}(ad+bc) + \beta^{n}bd$. Using the identity $ad + bc = (a + b)(c + d) - ac - bd$, we can compute xy with only three $n/2$-by-$n/2$-digit multiplications and a few additions[1].

Algorithm A.1 is Karatsuba's algorithm (first published in [4], English translation in [5]). It calls itself recursively three times to compute the three products, and does a few additions and shifts[2].

If Karatsuba's algorithm on two n-digit numbers takes $M_K(n)$ time, then $M_K(n) = 3M_K(n/2) + \mathcal{O}(n)$, because of the three recursive calls and the linear-time schoolbook additions. This recurrence relation has the solution $M_K(n) \in \mathcal{O}(n^{\log_2 3})$. Since $\log_2 3 \approx 1.585$, Karatsuba's algorithm beats schoolbook multiplication for big enough n.

Note that Karatsuba's algorithm is easily extended to handle cases where n is not a power of two, by not splitting the factors exactly in half. The time will still be $\mathcal{O}(n^{\log_2 3})$ as long as the ratio of the split is bounded by a constant (proof left as an exercise to the reader), but the more even the split, the faster the algorithm.

[1]Well, actually the product $(a+b)(c+d)$ might be $n/2+1$ by $n/2+1$ digits, but that does not matter – it takes at most $\mathcal{O}(n)$ extra time compared to an $n/2$-by-$n/2$-digit multiplication. Just let $a + b = e + \beta^{n/2}f$ and $c + d = g + \beta^{n/2}h$, where f and h are single-digit numbers, so that $(a+b)(c+d) = eg+\beta^{n/2}(eh+fg)+\beta^{n}fh$. eg is $n/2$ by $n/2$ digits, fh is just 1 by 1 digit, and eh and fg are $n/2$ by 1 digits and can be computed in $\mathcal{O}(n)$ time using schoolbook multiplication.

[2]Multiplication by powers of the base β are accomplished in constant time by simply offsetting one number relative to another when doing a subsequent addition.

Algorithm A.1 Karatsuba's algorithm

Input Two nonnegative integers x and y, each represented by n digits, where n is a power of two.

Output The product xy.

```
1. If $n = 1$, just return the product $xy$.
2. Let $x = a + \beta^{n/2} b$ and $y = c + \beta^{n/2} d$.
3. Compute $t_1 = ac$ using Karatsuba's algorithm.
4. Compute $t_2 = bd$ using Karatsuba's algorithm.
5. Compute $t_3 = (a+b)(c+d)$ using Karatsuba's algorithm.
6. $t_4 \leftarrow t_3 - t_2 - t_1$
7. Return the product $t_1 + \beta^{n/2} t_4 + t_2$.
```

(Of course, you could pad x and y with enough zeros to make the number of digits a power of two, but this way is generally faster.)

A.2 Toom's Algorithm

Karatsuba's algorithm is a special case (for $r = 2$) of an algorithm that splits each factor in r parts and then computes the original product using $2r-1$ multiplications of numbers the size of these parts. This is Toom's algorithm (algorithm A.2).

It works as follows: Consider both n-digit operands to be $(r-1)$-degree polynomials, $X(t)$ and $Y(t)$, with n/r-digit coefficients. Evaluate them at $2r-1$ points; this can be done in linear time, using only additions and multiplication by constant-size numbers (since we only need $2r-1$ different points, and r is a constant). Multiply those values pairwise so as to obtain the values of the product polynomial $Z(t)$ at the same points; this takes $2r-1$ multiplications of numbers with n/r digits. Calculate the $2r-1$ $2n/r$-digit coefficients of $Z(t)$, using only addition, subtraction, and multiplication and division by constant-size numbers (this is a linear equation system with $2r-1$ equations and $2r-1$ unknowns). Now, the original product is $Z(n/r)$, which is easily evaluated using additions and shifts.

The only part of this algorithm that does not run in linear time are the $2r-1$ recursive calls; if Toom's algorithm on two n-digit numbers takes $M_T(n)$ time, then $M_T(n) = (2r-1)M_T(n/r) + \mathcal{O}(n)$. This recurrence relation has the solution $M_T(n) \in \mathcal{O}(n^{\log_r(2r-1)})$. Table A.1 shows this for some values of r (it also indicates the fact that when one has a specific r in mind, one can refer to the algorithm as 'Toom-r').

Algorithm A.2 Toom's algorithm

```
Input  Two nonnegative integers $x$ and $y$, each represented by $n$ digits. A (small)
       integer $r \geq 2$.

Output  The product $xy$.

  1. If $n < r$, compute the product $xy$ using schoolbook multiplication.
  2. $s \leftarrow \lceil n/r \rceil$
  3. Let $x = x_{r-1}s^{r-1} + \ldots + x_1 s + x_0$ and $y = y_{r-1}s^{r-1} + \ldots + y_1 s + y_0$.
  4. Let $X(t) = x_{r-1}t^{r-1} + \ldots + x_1 t + x_0$ and $Y(t) = y_{r-1}t^{r-1} + \ldots + y_1 t + y_0$.
  5. Evaluate $X(t)$ and $Y(t)$ for $2r - 1$ (small) values of $t$, and compute the value
     of $Z(t) = X(t)Y(t)$ for those $t$, using $2r - 1$ recursive calls to this algorithm.
  6. Compute the $2r - 1$ coefficients of $Z(t)$.
  7. Return the product $Z(s)$.
```

By setting r large enough, we can get a multiplication algorithm that runs in time $\mathcal{O}(n^{1+\epsilon})$ for any $\epsilon > 0$.

When implementing Toom's algorithm, one can either implement it for arbitrary r or for just a few particular values. The bigger r is, the faster it will be for sufficiently large operands, but the overhead (everything except the recursive calls) grows with r, so for small operands a small r is better.

GMP implements only Toom-3 (and Karatsuba's algorithm of course), for the simple reason that FFT multiplication is faster than Toom's algorithm for *all* operand sizes for large enough r, where 'large enough' is not very large at all; GMP maintainer Torbjörn Granlund expects that a suitably optimized FFT might be faster than all Toom algorithms except Toom-2 (which is equivalent to Karatsuba's algorithm) on some machines, and there are certainly no plans to implement a Toom-4.

A.3 FFT Multiplication

The overhead in Toom's algorithm is in $\mathcal{O}(n)$ for any fixed r. However, it grows rather maliciously with r (more than quadratically), so that if we let r be a function of n in order to automatically choose larger r as n grows, we end up with $\mathcal{O}(n2^{\sqrt{2 \log n}} \log n)$ if we choose $r(n)$ optimally[6].

There is, however, a clever way to do it faster. If, instead of choosing evaluation points more or less arbitrarily, we pick roots of unity, the evaluation and interpolation steps are in fact a Discrete Fourier Transform (DFT) and its inverse (which is almost the same thing). This is just a name change, however; we still have to do the exact

Algorithm	Exponent	
Toom-2 (Karatsuba)	$\log_2 3$	≈ 1.585
Toom-3	$\log_3 5$	≈ 1.465
Toom-4	$\log_4 7$	≈ 1.404
Toom-5	$\log_5 9$	≈ 1.365
Toom-6	$\log_6 11$	≈ 1.338
Toom-7	$\log_7 13$	≈ 1.318
Toom-8	$\log_8 15$	≈ 1.302
Toom-9	$\log_9 17$	≈ 1.289
Toom-10	$\log_{10} 19$	≈ 1.279
Toom-100	$\log_{100} 199$	≈ 1.149

Table A.1. Running time of Toom's algorithm.

same amount of work, $\mathcal{O}(r^2)$ arithmetic operations on numbers with $\mathcal{O}(\log r)$ digits. But, if r is a power of two we may compute the DFT using the Fast Fourier Transform (FFT) algorithm, reducing the complexity to $\mathcal{O}(r \log r)$ operations.

There is a catch, though: $\mathbb{Z}$ (and $\mathbb{Q}$ and $\mathbb{R}$) have only two roots of unity, 1 and -1. This forces us to pretend that our numbers live in a ring that has enough roots of unity, such as $\mathbb{C}$ or $\mathbb{Z}_p$ for some prime p; both of these are used in practice.

- Computing in $\mathbb{C}$ is convenient since there are infinitely many roots of unity ($e^{x\pi i}$ is a root of unity for every $x \in \mathbb{Q}$). The problem is that floating point computations are inherently inexact; see Percival[7] for more details.

- Computing in $\mathbb{Z}_p$ gives provably exact results, but making it as fast as computing in $\mathbb{C}$ is difficult, mostly because processor manufacturers feel that floating point multiplication performance is more important than integer multiplication performance (and because the math involved is trickier). One variant of this is to compute modulo several small primes, and then at the end reconstruct the result modulo their product using the Chinese Remainder Theorem; this might prove very fast in practice since the several small primes can be chosen to fit in machine words.

FFT multiplication can be done in a variety of ways, all of which involve a trade-off between simplicity, asymptotic efficiency and low overhead. The asymptotically fastest known algorithm, by Schönhage and Strassen[9], runs in time $\mathcal{O}(n \log n \log \log n)$, but as Yap and Li[10] point out, it may not be preferable in practice.

A.4 Multiplication In Practice

In practice, the multiplication algorithms are not as disjunct as this appendix suggests; not only does a practical implementation choose which multiplication algorithm to start with based on the size of the operands (see table 5.1), *but that choice* *is made every time a multiplication algorithm makes a recursive call.* Both algorithms A.1 and A.2 explicitly call themselves recursively, but in practice they should call whichever multiplication algorithm is fastest. Karatsuba could call the schoolbook algorithm, and Toom-n might call schoolbook, Karatsuba, or any Toom-m for $m \leq n$.

This will always make things faster (measured in seconds), since the breakpoints where we switch between algorithms are based on measurements, but will not affect the asymptotic complexity.

Appendix B

Tedious Math

B.1 Inequalities From Taylor Series

Lemma B.1. For $0 < x < 1$,

$$1 - x < \frac{1}{1+x} < 1 - x + x^2 \tag{B.1}$$

Proof. Taylor series expansion of $\frac{1}{1-x}$ around $x = 0$ gives $\frac{1}{1-x} = 1+x+x^2+x^3+\ldots$, for $|x| < 1$. Changing the sign of x, we get $\frac{1}{1+x} = 1-x+x^2-x^3+\ldots$. The inequalities follow from the fact that, for $0 < x < 1$,

$$\sum_{i=n}^{\infty}(-1)^i x^i = (-1)^n x^n \sum_{i=0}^{\infty}(-1)^i x^i = \frac{(-1)^n x^n}{1+x} \tag{B.2}$$

which is strictly less than 0 if n is odd, and strictly greater than 0 if n is even. ∎

Theorem B.2. For $k \leq -2$,

$$\frac{1}{\frac{1}{2} + 2^k} < 2 - 2^{k+2} + 2^{2k+3} \tag{B.3}$$

Proof. Using lemma B.1, we get

$$\frac{1}{\frac{1}{2} + 2^k} = \frac{2}{1 + 2^{k+1}} < 2(1 - 2^{k+1} + 2^{2k+2}) = 2 - 2^{k+2} + 2^{2k+3} \tag{B.4}$$

The fact that $k \leq -2$ ensures that $0 < 2^{k+1} < 1$, so that the conditions for the lemma are fulfilled. ∎

B.2 Asymptotic Growth Rate of Sums

Theorem B.3.

$$\sum_{i=1}^{\log n} \log^k \frac{n}{2^i} \in \Theta(\log^{k+1} n) \tag{B.5}$$

Proof. Let $n = 2^s$. We have

$$\begin{aligned}
\sum_{i=1}^{\log n} \log^k \frac{n}{2^i} &= \sum_{i=1}^{s} \log^k \frac{2^s}{2^i} \\
&= \sum_{i=1}^{s} (s-i)^k \\
&= \sum_{j=0}^{s-1} j^k \qquad \text{(B.6)}
\end{aligned}$$

Now, this sum is bounded by

$$\int_0^{s-1} j^k dj < \sum_{j=0}^{s-1} j^k < \int_0^s j^k dj \qquad \text{(B.7)}$$

and since

$$\int x^k = \frac{x^{k+1}}{k+1} + C \qquad \text{(B.8)}$$

we conclude that

$$\begin{aligned}
\sum_{i=1}^{\log n} \log^k \frac{n}{2^i} &= \sum_{j=0}^{s-1} j^k \\
&\in \Theta(s^{k+1}) \\
&= \Theta(\log^{k+1} n) \qquad \text{(B.9)}
\end{aligned}$$

■

Theorem B.4.

$$\sum_{i=1}^{\log n} \log^k \frac{n}{2^i} \log\log \frac{n}{2^i} \in \Theta(\log^{k+1} n \log\log n) \qquad \text{(B.10)}$$

Proof. This proof is more or less a copy of that for theorem B.3. Let $n = 2^s$, as before. We have

$$\begin{aligned}
\sum_{i=1}^{\log n} \log^k \frac{n}{2^i} \log\log \frac{n}{2^i} &= \sum_{i=1}^{s} \log^k \frac{2^s}{2^i} \log\log \frac{2^s}{2^i} \\
&= \sum_{i=1}^{s} (s-i)^k \log(s-i) \\
&= \sum_{j=0}^{s-1} j^k \log j \qquad \text{(B.11)}
\end{aligned}$$

Now, this sum is bounded by

$$\int_0^{s-1} j^k \log j dj < \sum_{j=0}^{s-1} j^k \log j < \int_0^s j^k \log j dj \tag{B.12}$$

and since

$$\begin{aligned} \int x^k \log x dx &= -\int \frac{x^{k+1}}{k+1}\frac{1}{x} dx + \frac{x^{k+1}}{k+1}\log x \\ &= \frac{x^{k+1}\log x}{k+1} - \frac{x^{k+1}}{(k+1)^2} + C \end{aligned} \tag{B.13}$$

we conclude that

$$\begin{aligned} \sum_{i=1}^{\log n} \log^k \frac{n}{2^i} \log\log\frac{n}{2^i} &= \sum_{j=0}^{s-1} j^k \log j \\ &\in \Theta(s^{k+1}\log s) \\ &= \Theta(\log^{k+1} n \log\log n) \end{aligned} \tag{B.14}$$

■