Contents

1 Introduction: Integer Arithmetic **1**

1.1 Addition 1
1.2 Multiplication: The School Method 2
1.3 Checking 5
1.4 A Recursive Version of the School Method 10
1.5 Karatsuba Multiplication 14
1.6 Exact Analysis of the Karatsuba Algorithm 17
1.7 Engineering 19
1.8 Even Faster Integer Multiplication 21
1.9 Corrections and Remarks 24

Chapter 1

Introduction: Integer Arithmetic

We introduce our readers into the design, analysis, and implementation of algorithms by studying algorithms for basic arithmetic operations on large integers. We treat addition and multiplication in the text and leave division and square roots for the exercises.

Integer arithmetic is interesting for many reasons:

- Arithmetic on long integers is needed in applications like cryptography, geometric computing, and computer algebra.
- We are familiar with the problem and know algorithms for addition and multiplication. We will see that the high school algorithm for integer multiplication is far from optimal and that much better algorithms exist.
- We will learn basic analysis techniques in a simple setting.
- We will learn basic algorithm engineering techniques in a simple setting.
- We will see the interplay between theory and experiment in a simple setting.

We assume that integers are represented as digit-strings (digits zero and one in our theoretical analysis and larger digits in our programs) and that two primitive operations are available: the addition of three digits with a two digit result (this is sometimes called a full adder) and the multiplication of two digits with a one digit result. We will measure the efficiency of our algorithms by the number of primitive operations executed.

We assume throughout this section that a and b are n-bit integers. We refer to the bits of a as a_{n-1} to a_0 with a_{n-1} being the most significant (also called leading) bit and a_0 being the least significant bit.

1.1 Addition

We all know how to add two integers a and b. We simply write them on top of each other with the least significant bits aligned and sum digit-wise, carrying a single bit from one position to the next.

```
digit c = 0;
for (int i = 0; i < n; i++)
```

```
{ add a_i, b_i and c to form s_i and a new carry c; }
s_n = c;
```

We need one primitive operation for each position and hence a total of n primitive operations.

Lemma 1 *Two n-bit integers can be added with n primitive operations.*

1.2 Multiplication: The School Method

We all know how to multiply two integers. In this section we will review the method familiar to all of us, in later sections we will get to know a method which is significantly faster for large integers.

The *school method* for integer multiplication works as follows: We first form partial products p_i by multiplying a with the i-th digit b_i of b and then sum the suitably aligned products $p_i \cdot 2^i$ to obtain the product of a and b.

```
prod = 0;
for (int i = 0; i < n; i++) prod = a * b_i * 2^i + prod;
```

Let us analyze the number of primitive operations required by the school method. We need n primitive multiplications to multiply a by b_i and hence a total of $n \cdot n = n^2$ primitive multiplications. All intermediate sums are at most $2n$-bit integers and hence each iterations needs at most $2n$ primitive additions. Thus there are at most $2n^2$ primitive additions.

Lemma 2 *The school method multiplies two n-bit integers with no more than* $3n^2 = O(n^2)$ *primitive operations.*

The following procedure implements the method. The number type *integer* of LEDA realizes arbitrary precision integers. It represents integers by a sign bit and a vector of *unsigned longs*. Depending on the system an *unsigned long* holds either 32 or 64 bits[1], i.e., each digit is an integer in the range $[0..2^{32}-1]$ or $[0..2^{64}-1]$, respectively. . Primitive operations on digits map onto hardware adds and multiplies.

⟨*school method*⟩≡

```
#include <LEDA/integer.h>

integer school_mult(const integer& a, const integer& b)
{ int n        = b.used_words();   // number of digits in b
  integer prod = 0;
  for (int i = 0; i < n; i++)
  { digit d = b.contents(i);       // i-th digit of b
    prod += a * d << ( i * 8 * sizeof(digit));
  }
  return sign(b) * prod;
}
```

[1] The C++ function *sizeof* returns the size of its arguments in bytes and hence the amount of shift in the i-th iteration is $i * 8 * sizeof(digit)$.

n	T
40000	0.3
80000	1.18
160000	4.8
320000	20.34

Table 1.1: The running time of the school method for the multiplication of n-bit integers. The running time grows quadratically.

Let us see how this algorithm performs. The following program generates two random n-bit numbers a and b and computes their product for different values of n. It starts with $n = 40,000$ and increases n by factors of 2. In each iteration n and the running time is reported. The program runs in two modes (see [MN99, Section 3.8]) depending on whether it is compiled with the flag `-DBOOK` or not. In the former case it produces a table in LaTeX-format. In the latter case it produces ASCII-output on cout. There is one subtle point in the program. Without the test `(c < 0)` the result of the multiplication is not used and hence a clever compiler might optimize the multiplication away. This would defeat the purpose of the experiment.

⟨*school_method.c*⟩≡

```
#include <LEDA/IO_interface.h>
⟨school method⟩
int main(){
IO_interface I;
int N = 40000;
for (int n = N; n <= 8*N; n *= 2)
{ integer a = integer::random(n), b = integer::random(n), c;
  float T = used_time();
  I.write_demo("\nnumber of digits              = ",n);
  I.write_table("\n ", n);
  c = school_mult(a,b);
  float UT = used_time(T);
  I.write_demo("time for school multiplication = ",UT);
  I.write_table(" & ",UT, "\\\\ \\hline");
  if (c < 0) break; // so that the compiler cannot optimize
}
return 0;
}
```

Table 1.1 shows the table produced by the program above. The quadratic growth of the running time is clearly visible: Doubling n leads to a four-fold increase in running time. We can interpret the table in different ways:

(1) *We can take the table to confirm our theoretical analysis*. Our analysis predicts quadratic growth and we are measuring quadratic growth. However, we analyzed the number of primitive operations and we measured running time on a vintage 1999 computer (either a 300 MHz SUN ULTRA-SPARC or a 266 MHz Pentium depending on

whether the experiments were made in the author's office or at his home). Our analysis concentrates on primitive operations on digits and completely ignores all book keeping operations and all questions of storage. The experiments show that this abstraction is a useful one. We will frequently only analyze the number of "representative operations". Of course, the choice of representative operations requires insight and knowledge. In Section **??** we will introduce a realistic computer model to have a basis for abstraction. We will develop tools to analyze running time of algorithms on this model. We will also connect our model to real machines, so that we can take our analysis as a predictor of actual performance. We will investigate the limits of our theory. Under what circumstances are we going to concede that an experiment contradicts theoretical analysis?

(2) *We can use the table to strengthen our theoretical analysis*. Our theoretical analysis tells us that the running time grows quadratically in n. From our table we may conclude that the running time on a 300 MHz SUN ULTRA-SPARC is approximately $1.84 \cdot 10^{-10} \cdot n^2$ seconds. We can use this knowledge to *predict* the running time for our program on other inputs. The following code snippet performs the experiment. It asks for the number of digits and then performs K multiplications of n-bit integers. We determine K such that the total predicted time for the multiplications is at least one second. This is to exclude that inaccuracies of the time measurement spoils the experiment. We report the running time for a single multiplication and the ratio between the measured running time and the prediction.

⟨*school_method_prediction*⟩≡

```
int n = I.read_int("number of digits = ");
int K = 1 + (int) (1000000000.0/(1.84*n*n));
integer a = integer::random(n), b = integer::random(n), c;
float T = used_time();
for (int k = 0; k < K; k++)
{
  c = school_mult(a,b);
  if (c < 0) break; // so that the compiler cannot optimize
}
float UT = used_time(T)/K;
I.write_demo("running time            = ",UT);
I.write_demo("ratio to predicted time = ",UT/(1.84*n*n/10000000000.0));
```

Here are three sample outputs. For $n = 100,000$, the running time is 1.85 seconds and the ratio is 1.005, for $n = 1000$, the running time is 0.0005147 seconds and the ratio is 2.797, for $n = 200$, the running time is $3.3 \cdot 10^{-5}$ seconds and the ratio is 4.5, and for $n = 1,000,000$, the running time is 263.6 seconds and the ratio is 1.433. We see that our predictions can be far off. We simply were too careless. We started with the assumption that the running is cn^2 for some constant c, estimated c, and then predicted. Starting from the assumption that the running time is $cn^2 + dn + e$ would have lead us to different conclusions. We need to do our theory more carefully[2] in order to be able to predict. Also, when we made the prediction that the running time is approximately $1.84 \cdot 10^{-10} \cdot n^2$ seconds, we did not make any restrictions on n.

[2] Maybe this is not what you wanted to read, but it is the truth.

However, my computer has a finite memory (albeit large) and this memory is organized into a complex hierarchy of registers, first and second level cache, main memory, and disk memory. The access times to the different levels of the memory differ widely and this will have an effect on running times. None of the experiments reported so far requires our program to use disk memory. We should analyze the space requirement of our program in order to be able to predict for how large a value of n the program is able "to run in core[3]".

In our example, we ran into both traps. The prediction is off for small n because we ignored the linear and constant terms in the running time and the prediction is off for large n because our prediction ignores the effects of the memory hierarchy.

(3) *We can use the table to conjecture quadratic growth of the running time of our algorithm*. Again, we need to be careful. What are we actually conjecturing? That the running time grows quadratically on random numbers? After all, we ran the algorithm only on random numbers (and even on very few of them). That the running time grows quadratically in the worst case, i.e., that there are no instances that lead to higher than quadratic growth? That the running grows quadratically in the best case, i.e., that there are no instances that lead to less than quadratic growth? We see that we need to develop more concepts and a richer language.

1.3 Checking

Our implementation *school␣mult* of the school method for integer multiplication is a short, but nevertheless non-trivial program. *We better make sure that we did not blunder*[4]. What can we do? We discuss two approaches: testing and program checking.

Testing means to run the program on inputs for which the result is known by other means. For example, we could prepare a list of triples (a_i, b_i, c_i) with $c_i = a_i \cdot b_i$ by hand and run our program on this list. It is wise to include boundary cases into this list, e.g., multiplication by 0 and ± 1. It is important to keep in mind that testing can never prove the absence of error, it only establishes that the program is correct on the test cases. This is not ideal, but a lot better than no assurance at all.

Program checking aims at a complete check of the post-condition of the program. A program checker for a program implementing a function f is a program that takes inputs x and y (and maybe additional input, that eases checking) and verifies whether $y = f(x)$. Program checking exploits the fact that verifying whether $y = f(x)$ is frequently much easier than computing $f(x)$ (in particular, if the person or program that computed $f(x)$ also computed additional information I that eases checking). We all know this fact from daily life. Checking an answer to a question is frequently much simpler than coming up with the answer. Program checking is used extensively in the LEDA system and has contributed considerably to its reliability, cf. [MN99, Section 2.14]. Two-version programming is a special version of program checking. Here, the function f is computed by two independent programs and the outputs of the two programs are compared.

[3]Main memory was called core memory in ancient times (when the author studied computer science).

[4]The first version of *school␣mult* was incorrect, because the author erroneously believed that *sizeof* returns the size in bits.

We describe a checker for integer multiplication that generalizes the so-called "Neuner-Probe". We have

$$123 \cdot 456 \neq 56188 \quad \text{since} \quad 123 \cdot 456 \not\equiv 56188 \pmod 9 .$$

The Neuner-Probe is taught in school since the remainder of a decimal number modulo 9 is particularly easy to compute: it is simply the digit sum modulo 9. The Neuner-Probe does not catch all errors in multiplications, e.g., the error in $9 \cdot 9 = 72$ is not caught. In fact, whenever $c - a \cdot b$ is divisible by 9, c is accepted as the product of a and b. However, checking $9 \cdot 9 = 72$ module 5 will catch the error.

We generalize the Neuner-Probe in two ways:

- We take random k-bit primes for a suitable k and check whether $a \cdot b \equiv c \pmod p$ by checking whether $((a \bmod p) \cdot (b \bmod p)) \bmod p = c \bmod p$. This check is computationally simple, as we will see, and discovers an error in a multiplication if $D = c - a \cdot b$ is not divisible by p. We choose k sufficiently large, such that at most one-tenth of the k-bit primes divide D. In this way a random prime will witness $c \neq a \cdot b$ with probability at least $1 - 1/10$.

- We repeat the test 10 times. This will guarantee that one of the random primes witnesses $c \neq a \cdot b$ with probability at least $1 - 10^{-10}$.

The checker has has running time $O(n \log n)$ and hence the time for checking is negligible compared to the time required for multiplication, at least for large numbers. Let $c = \mathit{school\_mult}(a,b)$. The following program[5] verifies that c is indeed the product of a and b; $\mathit{random\_prime}(k)$ returns a random prime p with $p < 2^k$.

⟨*check multiplication*⟩≡

```
#include <math.h>
bool is_large_enough(int k, int n)
{ // log denotes natural logarithm in the C math library
  double f1 = k * log(2) - log( 10 * k * log(2));
  double f2 = log(  k   - log( 10 * k * log(2) ) - 1 )/log(2);
  return ( log(2*n+1) < f1 + log(f2) );
}
bool check_mult(const integer& a, const integer& b, const integer& c)
{ int n = leda_max(a.length(),b.length());
  if ( c.length() > 2*n ) return false;
  int k = 1;
  while ( !is_large_enough(k,n) ) k++;
  for (int i = 0; i < 10; i++)
  { integer p = random_prime(k);
    integer amod = a % p, bmod = b % p, cmod = c % p;
    if ( (amod * bmod) % p  != cmod ) return false;
  }
  return true;
}
```

[5] For integers a and p, $a \bmod p$ is the unique integer r such that $0 \leq r < p$ and $a - r$ is divisible by p. The C++ notation for mod is %.

Theorem 1 *Let a and b be n-bit integers. If $c = a \cdot b$, the program above returns true. If $c \neq a \cdot b$, the program returns true with probability less than 10^{-10}.*

The integer k chosen by the program is bounded above by $\lceil k_0 \rceil$, where $k_0 = \max(20, \log((20 \ln 2)) + \log(2n+1)) = \log n + O(1)$.

Proof: Assume first that $c = a \cdot b$. Then c is a $2n$-bit number and hence the test `c.length() > 2*n` does not fire. Also $c = a \cdot b$ implies $c \equiv a \cdot b \pmod{p}$ for any prime p and hence $c \bmod p = ((a \bmod p) \cdot (b \bmod p)) \bmod p$. Thus the program returns *true*.

Assume next that $c \neq a \cdot b$. If c is not a $2n$-bit integer, the program returns *false*. So assume that c is a $2n$-bit integer and let $D = c - a \cdot b$. Then $D \neq 0$ and $|D| \leq |c| + |ab| \leq 2^{2n+1}$.

Let p be a prime returned by *random_prime*(k). If p does not divide D, $D \bmod p \neq 0$ and hence $((a \bmod p) \cdot (b \bmod p)) \bmod p \neq c \bmod p$, i.e., the program will return *false*.

Let q be the number of prime divisors of $|D|$. Since the product of any q distinct primes is at least $q!$ we have $q! \leq |D|$. We will show that our choice of k guarantees that there are at least $10q$ primes less than 2^k. If this is true, the probability that the prime chosen in any iteration divides D is less than $1/10$ and hence the probability that all ten primes chosen divide D is less than 10^{-10}.

Prime numbers are abundant. The distribution of prime numbers is a well-studied problem in number theory. We use $\pi(x)$ to denote the number of primes less than x. Then $\pi(x) \geq x/\ln x$ ([NZ, Vol. 2, Theorem 8.2]).

We want to choose k such that

$$\pi(2^k) \geq 10q \ .$$

Since $q! \leq 2^{2n+1}$, it suffices to have

$$(\pi(2^k)/10)! \geq 2^{2n+1} \ .$$

Since $x! \geq (x/e)^x$ (Stirling's inequality[6], see Section **??**) and hence $x \ln(x/e) \geq \ln(x!)$, it suffices to have

$$\frac{\pi(2^k)}{10} \cdot \log \frac{\pi(2^k)}{10e)} \geq 2n+1 \ .$$

Since $\pi(2^k) \geq 2^k/(k \ln 2)$ it suffices to have

$$2n+1 \leq \frac{2^k}{10k \ln 2)} \cdot \log \frac{2^k}{10ek \ln 2)}$$

or after taking natural logarithms

$$\ln(2n+1) \leq k \ln 2 - \ln(10k \ln 2) + \ln(\frac{k - \ln(10ek \ln 2)}{\ln 2} \ .$$

The program chooses the smallest integer k for which this condition holds.

It remains to verify that $k \leq \lceil k_0 \rceil$. For this is suffices to verify that the inequality above holds true when k is replaced by k_0. Since $k_0 \geq 20$, we have $\log(2^{k_0}/(10ek_0 \ln 2)) = k_0 - \log(10ek_0 \ln 2) \geq k_0/2$ (this is certainly true for $k_0 = 20$ and holds for larger k_0 since $k_0/2$ grows faster than $\log(10ek_0 \ln 2)$) and hence it suffices to verify that $2n+1 \leq 2^{k_0}/(20 \ln 2)$ or $\log(20 \ln 2) + \log(2n+1) \leq k_0$. This is true by the definition of k_0. ∎

The theorem above is truly remarkable. In order to verify the multiplication of two one-million-bit numbers ($n = 2^{20}$) with error probability less than 10^{-10} we only need to check the multiplication modulo ten 27-bit primes.

The multiplication checker obtains much of its power from the use of randomization; it is a so-called *randomized algorithm*. Many randomized algorithms with surprising properties have been discovered in the last fifteen years. We will see a few in this book; many more can be found in [MR95].

What is the running time of our checker? The school method requires 1.86 seconds to multiply two $100,000$-bit numbers and the checker requires 0.29 seconds to check it, i.e., the time required for the check is so small compared to the time for the multiplication that we should always (!!!) do the check.

⟨*checked school method*⟩≡

```
integer checked_school_mult(const integer& a, const integer& b)
{ integer prod = school_mult(a,b);
  leda_assert(check_multiplication(a,b,prod),"error in school mult",1);
  return prod;
}
```

What is the difference between *school_mult* and *checked_school_mult*? The second program is slightly slower but much to be prefered. Neither of the programs has been proved correct in a formal sense. We coded both programs carefully and for the second program we also programmed a checker. We have invested a significant amount of energy into the theory underlying the checker and proved that the checker will catch errors with high probability. Of course, there is the possibility that we made an error in the implementation of the checker. In fact, in our case the checking program is more complex than the program to be checked. Usually, it is the other way around. EXPAND.

Our checker is efficient. We show that its running time is $O(n \log n)$.

Lemma 3 *Let a be a n-bit integer and let p be a k-bit integer. Then $a \bmod p$ can be computed with $O(nk)$ primitive operations. In the checker we have $k = O(\log n)$.*

Proof: Divide a into $z = n/k$ chunks $a_{z-1}, \ldots, a_0$ of k bits each and compute $a \bmod p$ iteratively according to the formula:

$$r_{z-1} = a_{z-1} \bmod p \qquad \text{and} \qquad r_{i-1} = (r_i \cdot 2^k + a_{i-1}) \bmod p\ .$$

Each iteration takes $O(k^2)$ primitive operations (see the exercises) and there are n/k iterations. The bound follows. ∎

We still need to discuss how to find random k-bit primes. Since primes are abundant, the easiest way to generate a random prime is to generate a random number and to check it for primality. Since the number of primes less than 2^k is at least $\pi(2^k) \geq 2^k/(k\ln 2)$, the expected number of iterations required to find a prime is $k\ln 2$ and hence the expected time to find a k-bit prime is $O(k)$ times the time to test primality of a k-bit integer.

⟨*random_prime*⟩≡

⟨*is_prime*⟩

```
integer random_prime(int k)
{ integer p;
  do { p = integer::random(k); }  while ( !is_prime(p) );
  return p;
}
```

In our application we are interested in primes with a small number of digits (less than 30) and hence we could probably obtain a significant speed-up if we would program the case $k \leq 30$ separately using native machine arithmetic instead of *integer*-arithmetic. We leave this to the reader.

How do we test primality of a number p? The naive way[7] of doing it, is to go through all integers i with $2 \leq i \leq \sqrt{p}$ and to check whether one of them divides p. If none does, p is prime.

⟨*is_prime*⟩≡

```
bool is_prime(const integer& p)
{ if (p < 2) return false;
  integer i = 2;
  while ( i * i <= p )
  { if ( p % i == 0 ) return false; i++; }
  return true;
}
```

If p is a k-bit prime, the running time of this algorithm is $O(\sqrt{p}k^2)$, since the loop is executed at most $\sqrt{p}$ times and since each iteration of the loop requires the multiplication and the division of k-bit numbers. Since $p \leq 2^k$, the running time is $O(\sqrt{2^k}k^2) = O(2^{k/2}k^2)$. We summarize.

Theorem 2 *The procedure check_multiplication has running time $O(n\log n)$.*

Proof: We have $k = \log n + O(1)$ and hence the time required to check a k-bit number for primality is $O(2^{(\log n + O(1))/2} \cdot \log^2 n) = O(\sqrt{n}\log^2 n)$. The expected time to generate a random k-bit prime is $O(k)$ times the time to check primality and hence is $O(\sqrt{n}\log^3 n)$. The time to carry out the reduction modulo p is $O(n\log n)$. The total time is thus $O(\sqrt{n}\log^3 n + n\log n) = O(n\log n)$. ∎

This concludes our first encounter of program checking. Let us summarize. Testing and checking are integral parts of software development. This is true for software

[7] There are much faster algorithms for testing primality [SS77, Rab80, MR95, Sip98].

in general and is particularly true for the sophisticated algorithms presented in this book. Program checking is to be prefered over program testing as it does a complete check of the post-condition. The design of a checker may require additional theory; it is time well spent. Always have checking and testing in mind, when you design a program. In his own software development, the author frequently writes the checker first.

1.4 A Recursive Version of the School Method

We derive a recursive version of the school method. This will be our first encounter of the *divide-and-conquer paradigm*, one of the fundamental paradigms in algorithm design. We will also learn in this section that constant factors can make a significant difference.

Let a and b be our two n bit-integers which we want to multiply. Let $k = \lfloor n/2 \rfloor$. We split a into two numbers a_1 and a_0; a_0 consists of the k least significant bits and a_1 consists of the $n-k$ most significant bits. Then

$$a = a_1 \cdot 2^k + a_0 \quad \text{and} \quad b = b_1 \cdot 2^k + b_0$$

and hence

$$a \cdot b = a_1 \cdot b_1 \cdot 2^{2k} + (a_1 \cdot b_0 + a_0 \cdot b_1) \cdot 2^k + a_0 \cdot b_0 \ .$$

This formula suggests the following algorithm for computing $a \cdot b$:

1. Split a and b into a_1, a_0, b_1, and b_0.

2. Compute the four products $a_1 \cdot b_1$, $a_1 \cdot b_0$, $a_0 \cdot b_1$, and $a_0 \cdot b_0$.

3. Add the suitably aligned products to obtain $a \cdot b$.

Observe that the numbers a_1, a_0, b_1, and b_0 are $\lceil n/2 \rceil$-bit numbers and hence the multiplications in step (2) are simpler than the original multiplication if $\lceil n/2 \rceil < n$, i.e., $n > 1$. The complete algorithm is now as follows: To multiply 1-bit numbers, use our multiplication primitive, and to multiply n-bit numbers for $n \geq 2$, use the three step approach above.

It is clear why this approach is called divide-and-conquer. We reduce the problem of multiplying $a \cdot b$ into some number of *simpler* problems of the same kind. A divide and conquer algorithm always consists of three parts: In the first part, we split the original problem into simpler problems of the same kind (our step (1)), in the second part we solve the simpler problems using the same method (our step (2)), and in the third part, we obtain the solution to the original problem from the solutions to the subproblems. The following program implements the divide-and-conquer approach to integer multiplication.

⟨*school method, recursive version*⟩≡

```
void split(const integer& a, integer& a1, integer& a0, int k)
{
  a1 = a >> k;
```

TODO

Figure 1.1:

```
  a0 = a - ( a1 << k );
}
integer school_mult_rec(const integer& a, const integer& b)
{ int n = leda_max(a.length(),b.length());
  if ( n <= 32 ) return a*b;
  int k = n/2;
  integer a1, a0, b1, b0;
  split(a,a1,a0,k); split(b,b1,b0,k);
  integer prod2 = school_mult_rec(a1,b1);
  integer prod1 = school_mult_rec(a1,b0) + school_mult_rec(a0,b1);
  integer prod0 = school_mult_rec(a0,b0);
  return (prod2 << 2*k) + (prod1 << k ) + prod0;
}
```

The procedure *split* splits a into a_1 and a_0, where a_0 has k bits. We use *split* in *school_mult_rec* to split a and b into their constituents, then perform the four recursive multiplications, and finally compute $a \cdot b$ from the partial results.

What is the connection of our recursive integer multiplication to the school method? It is really the same. Figure 1.1 shows that the products $a_1 \cdot b_1$, $a_1 \cdot b_0$, $a_0 \cdot b_1$, and $a_0 \cdot b_0$ are also computed by the school method.

Knowing that our recursive integer multiplication is just the school method in disguise tells us that the recursive algorithms uses a quadratic number of primitive operations. Let us also derive this from first principles. This will allow us to introduce recurrence relations, a powerful concept for the analysis of recursive algorithm.

Lemma 4 *Let $T(n)$ be the maximal number of primitive operations required by our recursive multiplication algorithm when applied to n-bit integers. Then*

$$T(n) \leq \begin{cases} 1 & \text{if } n = 1, \\ 4 \cdot T(\lceil n/2 \rceil) + 3 \cdot 2 \cdot n & \text{if } n \geq 2. \end{cases}$$

Proof: Multiplying two 1-bit numbers requires one primitive multiplication. This justifies the case $n = 1$. So assume $n \geq 2$. Splitting a and b into the four pieces a_1, a_0, b_1, and b_0 requires no primitive operations[8]. Each piece has at most $\lceil n/2 \rceil$ bits and hence the four recursive multiplications require at most $4 \cdot T(\lceil n/2 \rceil)$ primitive operations. Finally, we need three additions to assemble the final result. Each addition involves two numbers of at most $2n$ bits and hence requires at most $2n$ primitive operations. This justifies the inequality for $n \geq 2$. ∎

[8]It will require work, but it is work that we do not account for in our analysis.

We give two methods to solve the recurrence above. We will learn general techniques for solving recurrence relations in Section **??**. The general techniques generalize the methods that we learn on this simple example.

The simplest way of proving a bound on $T(n)$ is use induction to verify a bound known by other means. We know that $T(n)$ is bounded by a quadratic function by its connection to the school method.

Lemma 5 *$T(n) \le c \cdot n^2 + d \cdot n$ for n being a power of two provided that $1 \le c + d$ and $d \le -6$. In particular, $T(n) \le 7 \cdot n^2 - 6 \cdot n$ for n a power of two.*

Proof: We have $T(1) = 1 \le c + d$ provided that $1 \le c + d$.

We turn to the induction step. If n is a power of two, $\lceil n/2 \rceil = n/2$ and $n/2$ is a power of two. Thus

$$\begin{aligned} T(n) &\le 4 \cdot T(n/2) + 3 \cdot 2 \cdot n \\ &\le 4 \cdot (c \cdot (n/2)^2 + d \cdot n/2) + 6n \qquad \text{by I.H} \\ &= cn^2 + (2d + 6)n \\ &\le cn^2 + dn \end{aligned}$$

provided that $2d + 6 \le d$ or $d \le -6$.

We conclude that the induction base and the induction step hold provided that $1 \le c + d$, $d \le -6$. Both conditions are satisfied for $c = 7$ and $d = -6$. ∎

An alternative method is repeated substitution. Again, we restrict ourselves to powers of two. We repeatedly substitute the right hand side in the recurrence for T for the left until all occurrences of T have disappeared.

$$\begin{aligned} T(n) &\le 6 \cdot n + 4 \cdot T(n/2) \\ &\le 6 \cdot n + 4 \cdot (6 \cdot n/2 + 4 \cdot T(n/4)) \\ &\le \ldots \\ &\le 6 \cdot n + 4 \cdot (6 \cdot n/2 + 4 \cdot (6 \cdot n/4 + 4 \cdot (6 \cdot n/8 + 4 \cdot (\ldots \\ &\quad + 4 \cdot (6 \cdot n/2^{k-1} + 4 \cdot T(1)) \ldots)))) \\ &= 6 \cdot 4^0 n/2^0 + 6 \cdot 4^1 n/2^1 + 6 \cdot 4^2 n/2^2 + 6 \cdot 4^3 n/2^3 + \ldots + 4^k \cdot T(1) \\ &= 6 \cdot \sum_{i=0}^{k-1} 4^i n/2^i + 4^k \;. \end{aligned}$$

The many dots in the derivation above are a possible source of error. We should therefore take the derivation not as proof but as suggestion of what could be true. We verify the derivation by induction.

Lemma 6 *For $n = 2^k$ we have $T(n) \le 6 \cdot \sum_{i=0}^{k-1} 4^i n/2^i + 4^k = 7n^2 - 6n$.*

Proof: We use induction on k. For $k = 0$ we have $n = 1$ and $T(1) = 1 \le 4^0$. For $k > 0$ we have $n/2 = 2^{k-1}$ and hence $T(n/2) \le \sum_{i=0}^{k-2} 4^i (n/2)/2^i + 4^{k-1}$ by induction hypothesis. Thus

$$\begin{aligned}
T(n) &\leq 6n + 4 \cdot T(n/2) \\
&\leq 6n + 4 \cdot (6 \cdot \sum_{i=0}^{k-2} 4^i (n/2)/2^i + 4^{k-1}) \\
&= 6 \cdot 4^0 n/2^0 + 6 \cdot \sum_{i=0}^{k-2} 4^{i+1} n/2^{i+1} + 4^k \\
&= 6 \cdot 4^0 n/2^0 + 6 \cdot \sum_{i=1}^{k-1} 4^i n/2^i + 4^k \\
&= 6 \cdot \sum_{i=0}^{k-1} 4^i n/2^i + 4^k .
\end{aligned}$$

We next observe that $4^i/2^i = 2^i$ and $\sum_{0 \leq i \leq k-1} 2^i = 2^k - 1 = n-1$. Also $4^k = n^2$. Thus $6 \cdot \sum_{0 \leq i \leq k-1} 4^i n/2^i + 4^k = 6n(n-1) + n^2 = 7n^2 - 6n$. ∎

How can one generalize our bound for $T(n)$ to arbitrary n? The simplest way is semantic reasoning. It is clear[9] that it is more difficult to multiply longer number than shorter numbers and hence the number of primitive operations required to multiply n-bit numbers will be less than the number of primitive operations required to multiply $2^{\lceil \log n \rceil}$-bit numbers. Thus

$$T(n) \leq T(2^{\lceil \log n \rceil}) = O((2^{\lceil \log n \rceil})^2) = O((2n)^2) = O(n^2) .$$

The following lemma makes this reasoning precise. Mathematically less inclined readers may skip it.

Lemma 7 *Let $U : N \to N$ be defined by*

$$U(n) = \begin{cases} 1 & \text{if } n = 1, \\ 4 \cdot U(\lceil n/2 \rceil) + 3 \cdot 2 \cdot n & \text{if } n \geq 2. \end{cases}$$

Then

$$T(n) \leq U(n) \qquad \text{and} \qquad U(n) \leq U(2^{\lceil \log n \rceil}).$$

Proof: We use induction on n. We have $T(1) \leq 1 = U(1)$. For $n \geq 2$ we have

$$T(n) \leq 4T(\lceil n/2 \rceil) + 6n \leq 4U(\lceil n/2 \rceil) + 6n = U(n),$$

where the second inequality follows from the induction hypothesis. This proves the first claim.

For the second claim we also use induction on n. The claim is trivial for $n = 1$. So assume $n > 1$. The induction hypothesis states that $U(\lceil n/2 \rceil) \leq U(2^{\lceil \log \lceil n/2 \rceil \rceil})$. Let $k = \lceil \log n \rceil$. Then[10] $k - 1 = \lceil \log \lceil n/2 \rceil \rceil$ and hence

$$U(n) = 4U(\lceil n/2 \rceil) + 6n \leq 4U(2^{\lceil \log \lceil n/2 \rceil \rceil}) + 6n \leq 4U(2^{k-1}) + 62^k = U(2^k) .$$

[9]Be aware that most errors in mathematical arguments are near occurrences of the word clearly.

[10]Observe that $n \leq 2^k$ implies $n/2 \leq 2^{k-1}$ and hence $\lceil n/2 \rceil \leq 2^{k-1}$. Thus $\log \lceil n/2 \rceil \leq k-1$ and hence $\lceil \log \lceil n/2 \rceil \rceil \leq k-1$. Also $2^{k-1} < n$ implies $2^{k-2} < n/2 \leq \lceil n/2 \rceil$. Thus $k-2 < \log \lceil n/2 \rceil \leq \lceil \log \lceil n/2 \rceil \rceil$.

n	T_r	T_n	T_r/T_n
8000	0.44	0.01	44
16000	1.76	0.05	35.2
32000	7.1	0.19	37.37
64000	28.75	0.75	38.33
128000	114.4	3.1	36.9

Table 1.2: The running time of two quadratic methods for integer multiplication: T_r is the running time of the recursive version and T_n is the running time of the non-recursive version.

∎

At this point we have two algorithms for integer multiplication whose running time grows as a quadratic function in the number of bits of the number to be multiplied. We have two algorithm with running time $O(n^2)$.

Can two algorithms with running time $O(n^2)$ differ in their actual running times? They certainly can and Table 1.2 illustrates the point. Our first implementation of the school method is significantly faster than the second. *The example shows that constant factors matter*. Although the running time of both implementations grows quadratically in the size of the numbers involved, the constant factor in front of the n^2 is significantly larger for the second method. In fact, we can infer from the table that the constant factor is about 37 times larger and hence the running time of the second implementation is about 37 times the running time of the first implementation. This is due to the fact that splitting numbers into parts by shifting requires a significant amount of work. If we had access to the internal representation of *integer*s, we could implement splitting much more efficiently.

Please do not conclude from this example that recursion is a bad implementation strategy, on the contrary, it is an extremely powerful algorithmic paradigm. You will see a convincing example in the next section. We want you to take home from this example, that constant factors matter and can make a tremendous difference.

1.5 Karatsuba Multiplication

In 1962 the Soviet mathematician Karatsuba [KO63] discovered a faster way of multiplying large integers. The running time of his algorithm grows like $n^{\log 3} \approx n^{1.58}$. The method is surprisingly simple. Karatsuba observed that a simple algebraic identity allows one to save one multiplication in the divide-and-conquer implementation, i.e., one can multiply n-bit numbers using only three(!!) multiplications of integers half the size.

The details are as follows. Let a and b be our two n bit-integers which we want to multiply. Let $k = \lfloor n/2 \rfloor$. We split a into two numbers a_1 and a_0; a_0 consists of the k least significant bits and a_1 consists of the $n-k$ most significant bits. Then

$$a = a_1 \cdot 2^k + a_0 \quad \text{and} \quad b = b_1 \cdot 2^k + b_0$$

and hence (the magic is in the second equality)

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

At first sight, we have only made things more complicated. A second look shows that the last formula can be evaluated with only three multiplications, namely, $a_1 \cdot b_1$, $a_1 \cdot b_0$, and $(a_1 + a_0) \cdot (b_1 + b_0)$. We also need six additions. That is three more than in the recursive implementation of the school method. The key is that additions are cheap compared to multiplications and hence saving a multiplication more than outweighs three additional additions. We obtain the following algorithm for computing $a \cdot b$:

1. Split a and b into a_1, a_0, b_1, and b_0.

2. Compute the three products $p_2 = a_1 \cdot b_1$, $p_0 = a_0 \cdot b_0$, and $p_1 = (a_1 + a_0) \cdot (b_1 + b_0)$.

3. Add the suitably aligned products to obtain $a \cdot b$, i.e., compute $a \cdot b$ according to the formula $a \cdot b = p_2 \cdot 2^{2k} + (p_1 - p_2 - p_0) \cdot 2^k + p_0$.

The numbers a_1, a_0, b_1, b_0, $a_1 + a_0$, and $b_1 + b_0$ are $\lceil n/2 \rceil + 1$-bit numbers and hence the multiplications in step (2) are simpler as the original multiplication if $\lceil n/2 \rceil + 1 < n$, i.e., $n \geq 4$. The complete algorithm is now as follows: To multiply 3-bit numbers, use the school method, and to multiply n-bit numbers for $n \geq 4$, use the three step approach above.

The following program shows an implementation (where we made the switch to the built-in method for integer multiplication at $n = 32$).

⟨*Karatsuba*⟩≡

```
integer fast_mult(const integer& a, const integer& b)
{ int n = leda_max(a.length(),b.length());
  if ( n <= 32 ) return a*b;
  n = n/2;
  integer a1, a0, b1, b0;
  split(a,a1,a0,n); split(b,b1,b0,n);
  integer prod2 = fast_mult(a1,b1);
  integer prod1 = fast_mult(a0 + a1,b0 + b1);
  integer prod0 = fast_mult(a0,b0);
  return (prod2 << 2*n) + (( prod1 - prod0 - prod2) << n  ) + prod0;
}
```

Table 1.3 shows the running time of the Karatsuba method in comparison with the school method. We also show the ratio of the running times (last column) and the ratio to the running time in the preceding iteration, i.e., the ratio between the running time for $2n$-bit integers and the running time on n-bit integers. We see that the ratio is around three in case of the Karatsuba method (doubling the size of the numbers about triples the running time) and is about four in case of the school method (doubling the size of the numbers about quadruples the running time). The latter statement requires some phantasy. We also see that the Karatsuba-method looses on short integers but wins on very large integers.

n	T_K	T_K/T_K'	T_S	T_S/T_S'	T_K/T_S
80000	5.85	–	1.19	–	4.916
160000	17.51	2.993	4.73	3.975	3.702
320000	52.54	3.001	19.77	4.18	2.658
640000	161	3.065	99.97	5.057	1.611
1280000	494	3.068	469.6	4.698	1.052
2560000	1457	2.95	1907	4.061	0.7641
5120000	4310	2.957	7803	4.091	0.5523
80000	6.36	–	0.83	–	7.663
160000	19.04	2.994	3.89	4.687	4.895
320000	57.45	3.017	15.95	4.1	3.602
640000	171.3	2.981	81.63	5.118	2.098
1280000	515.9	3.012	364	4.46	1.417
2560000	1542	2.988	1532	4.207	1.006
5120000	4626	3.001	6142	4.01	0.7532

Table 1.3: The running time of the Karatsuba and the school method for integer multiplication: T_K is the running time of the Karatsuba method and T_S is the running time of the school method. T_K' and T_S' denote the running time of the preceding iteration. For an algorithm with running time $T(n) = cn^\alpha$ we have $T/T' = T(2n)/T(n) = 2^\alpha$. For $\alpha = \log 3$ we have $2^\alpha = 3$ and for $\alpha = 2$ we have $2^\alpha = 4$. The top half of the table was produced on a 300 MHz SUN ULTRA-SPARC, the bottom half was produced on a 266 MHz Pentium.

The lessons to remember are:

- Better asymptotics ultimately wins. However, it may loose on small inputs.
- An asymptotically slower algorithm can be faster on small inputs. You may only be interested in small inputs.

It is time to derive the asymptotics of the Karatsuba method.

Lemma 8 *Let $T(n)$ be the maximal number of primitive operations required by the Karatsuba algorithm when applied to n-bit integers. Then*

$$T(n) \le \begin{cases} 3n^2 & \text{if } n \le 3, \\ 3 \cdot T(\lceil n/2 \rceil + 1) + 6 \cdot 2 \cdot n & \text{if } n \ge 4. \end{cases}$$

Proof: Multiplying two n-bit numbers with the school method requires no more than $3n^2$ primitive operations by Lemma 2. This justifies the first line. So assume $n \ge 4$. Splitting a and b into the four pieces a_1, a_0, b_1, and b_0 requires no primitive operations[11]. Each piece and the sums $a_0 + a_1$ and $b_0 + b_1$ have at most $\lceil n/2 \rceil + 1$ bits and hence the three recursive multiplications require at most $3 \cdot T(\lceil n/2 \rceil + 1)$ primitive operations. Finally, we need two additions to form $a_0 + a_1$ and $b_0 + b_1$ and four additions

[11] It will require work, but it is work that we do not account for in our analysis.

to assemble the final result. Each addition involves two numbers of at most $2n$ bits and hence requires at most $2n$ primitive operations. This justifies the inequality for $n \geq 4$. ∎

We will next argue that $T(n) = O(n^{\log 3})$, a complete proof will be given in the next section.

Lemma 9 *Let a and b be non-negative constants and let $U(n)$ be defined by*

$$U(n) = \begin{cases} a & \text{if } n = 1, \\ 3 \cdot U(n/2) + b \cdot n & \text{if } n > 1. \end{cases}$$

Then $U(n) \leq (2a+b)n^{\log 3} = O(n^{\log 3})$ for n a power of two.

Proof: Let $k = \log n$. Then

$$\begin{aligned} U(n) &= bn + 3U(n/2) \\ &= bn + 3(bn/2 + 3U(n/4)) \\ &= \ldots \\ &= bn \cdot \sum_{0 \leq i \leq k-1} \frac{3^i}{2^i} + 3^k a \\ &\leq (2b+a)n^{\log 3}, \end{aligned}$$

since $\sum_{0 \leq i \leq k-1} (3/2)^i = ((3/2)^k - 1)/(3/2 - 1) \leq 2 \cdot 3^k/2^k$, $2^k = n$, and $3^k = 2^{k \cdot \log 3} = n^{\log 3}$. ∎

The lemma above does not prove $T(n) = O(n^{\log 3})$, it only suggests it. Table 1.3 gives additional evidence for $T(n) = O(n^{\log 3})$. It is a rule of thumb that replacing $\lceil n/2 \rceil + O(1)$ by $n/2$ in the analysis of a recurrence does not change the asymptotics of the solution. We give a full proof that the sloppiness makes no difference in the next section. The full proof may be skipped by mathematically less inclined readers.

1.6 Exact Analysis of the Karatsuba Algorithm

We will prove $T(n) = O(n^{\log 3})$. Mathematically less inclined readers may skip this section.

Theorem 3 *Let $T(n)$ be the maximal number of primitive operations required by the Karatsuba algorithm when applied to n-bit integers. Then $T(n) = O(n^{\log 3})$.*

Proof: We use a method which is known as bootstrapping or factoring out known factors. We want to show that $T(n)$ grows like $n^{\log 3}$. We therefore define a function $U(n)$ by $U(n) = T(n)/n^{\log 3}$. The recurrence relation for $T(n)$ yields the following recurrence for $U(n)$.

$$n^{\log 3} U(n) \leq \begin{cases} 3n^2 & \text{if } n \leq 3, \\ 3 \cdot (\lceil n/2 \rceil + 1)^{\log 3} U(\lceil n/2 \rceil + 1) + 12n & \text{if } n \geq 4. \end{cases}$$

or

$$U(n) \le \begin{cases} 3n^{2-\log 3} & \text{if } n \le 3, \\ 3((\lceil n/2 \rceil + 1)/n)^{\log 3} U(\lceil n/2 \rceil + 1) + 12n^{1-\log 3} & \text{if } n \ge 4. \end{cases}$$

Our goal is to show that $U(n)$ is bounded by a constant. We use the substitution method as formulated in Theorem **??** with $n_0 = 3$, $r(n) = \lceil n/2 \rceil + 1$, $g(n) = 12n^{1-\log 3}$ and $f(n) = 3((\lceil n/2 \rceil + 1)/n)^{\log 3}$. Then

$$U(n) = \sum_{j=0}^{k(n)} g(r^{(j)}(n)) \prod_{i=0}^{j-1} f(r^{(i)}(n)) \ .$$

This expression is intimidating at first sight. But actually, it simplifies quickly. We first observe that $3 = 2^{\log 3}$ and hence $f(n) = 3((\lceil n/2 \rceil + 1)/n)^{\log 3} \le (2(n/2+2)/n)^{\log 3} \le (1+4/n)^{\log 3} \le e^{\log 3 \cdot \ln(1+4/n)} \le e^{8/n}$, where the last inequality uses the facts that $\ln(1+x) \le x$ and $4 \log 3 \le 8$. We next observe that $r(n) = \lceil n/2 \rceil + 1 \ge n/2$ and hence $r^{(i)}(n) \ge n/2^i$. Thus $f(r^{(i)}(n)) \le f(n/2^i)$ since f is a decreasing function and hence

$$\begin{aligned} \prod_{i=0}^{j-1} f(r^{(i)}(n)) &\le \prod_{i=0}^{j-1} f(n/2^i) \\ &\le \prod_{i=0}^{j-1} e^{82^i/n} \\ &= e^{(8/n)\sum_{0 \le i \le j-1} 2^i} \\ &\le e^{8 \cdot 2^j/n} \ . \end{aligned}$$

Thus

$$U(n) \le \sum_{j=0}^{k(n)} g(r^{(j)}(n)) e^{8 \cdot 2^j/n} \le e^{8 \cdot 2^{k(n)}/n} \sum_{j=0}^{k(n)} g(r^{(j)}(n))$$

which is already much less intimidating. Observe next that g is also a decreasing function of n and hence, by an analogous argument, $g(r^{(j)}(n)) \le g(n/2^j) = 12(2^j/n)^{\log 3 - 1} = 12 \cdot 2^{\alpha j}/n^{\alpha}$ with $\alpha = \log 3 - 1$. Observe that $2^{\alpha} = 3/2$. Thus

$$\begin{aligned} U(n) &\le 12 e^{8 \cdot 2^{k(n)}/n} (1/n)^{\alpha} \sum_{j=0}^{k(n)} 2^{\alpha j} \\ &\le 12 e^{8 \cdot 2^{k(n)}/n} (1/n)^{\alpha} 2^{\alpha(k(n)+1)}/(2^{\alpha} - 1) \\ &\le 24 e^{8 \cdot 2^{k(n)}/n} 2^{\alpha(k(n)+1-\log n)} \end{aligned}$$

It remains to estimate $k(n)$. We will show that $k(n) \le \log n + C$ for some constant C. Then

$$U(n) \le 24 e^{8 \cdot 2^C} 2^{\alpha(C+1)} = O(1)$$

and we are done.

For $i \ge 1$ let a_i be the smallest integer such that $k(a_i) = i$. Then $a_1 = 4$ and $r(a_i) = a_{i-1}$ for $i \ge 2$. The condition $r(a_i) = a_{i-1}$ is equivalent to $\lceil a_i/2 \rceil + 1 = a_{i-1}$ or $a_i =$

$2a_{i-1}-3$. The recurrence $a_1 = 4$ and $a_i = 2a_{i-1} - 3$ for $i \geq 2$ is again easily solved by substitution. We obtain

$$a_i = 2^{i-1} \cdot 4 - 3 \sum_{0 \leq j < i-2} 2^j = 2^{i+1} - 3(2^{i-1} - 1) = 2^{i-1} + 3 \ .$$

Readers which do not trust the substitution method should verify the expression $a_i = 2^{i-1} + 3$ by induction on i (the author did).

At this point we know that $a_i = 2^{i-1} + 3$ is the smallest integer with $k(a_i) = i$. Rewriting $a_i = 2^{i-1} + 3$ yields $i = \log(a_i - 3) + 1$. Thus $k(n) \leq \log(n-3) + 1$ and we are done. ∎

1.7 Engineering

Karatsuba integer multiplication is asymptotically superior to the school method. In our implementation the superiority only shows for integers with more than several million bits. In this section we show that how to combine the Karatsuba and the school method into a method superior to both. The idea is simple. Since the school method is superior to Karatsuba for short integers we should stop the recursion earlier and switch to the school method for numbers which have at n_0 bits for some yet to be determined n_0. In this way we obtain a method which is never worse than the school method and also has the $O(n^{\log 3})$ asymptotics. We ask the reader to redo the proof of Lemma 9, when the recursion stops at n_0 instead of 1.

⟨*refined Karatsuba*⟩≡

```
integer imp_fast_mult(const integer& a, const integer& b, int n0)
{ int n = leda_max(a.length(),b.length());
  if ( n <= n0 ) return school_mult(a,b);
  n = n/2;
  integer a1, a0, b1, b0;
  split(a,a1,a0,n); split(b,b1,b0,n);
  integer prod2 = imp_fast_mult(a1,b1,n0);
  integer prod1 = imp_fast_mult(a0 + a1,b0 + b1,n0);
  integer prod0 = imp_fast_mult(a0,b0,n0);
  return (prod2 << 2*n) + ((prod1 - prod0 - prod2) << n) + prod0;
}
```

What is a good choice for *n0*? We will answer this question analytically and experimentally.

We first approach the question analytically. We make the following assumptions:

- The running time of the school method is $T_S = a \cdot n^2$ for some constant a.
- The Karatsuba method requires time $c \cdot n$ to decompose the two arguments and to combine the results of the three recursive multiplications.

We now ask ourselves the question whether we should stop the recursion at n_0 and use the school method for numbers of this length or do one further Karatsuba step and use

$n0$	64	128	256	512	1024	2048	4096	8192	16384	32768
T	3.64	1.91	1.07	0.72	0.81	0.85	0.77	0.76	0.87	1.1
	32.78	16.93	9.65	6.46	7.37	7.67	6.83	7.02	7.85	9.9
n_0	64	128	256	512	1024	2048	4096	8192	16384	32768
T	4.17	2.09	1.15	0.69	0.57	0.51	0.44	0.43	0.49	0.63
	37.63	18.97	10.11	6.34	5.26	4.63	4.05	3.96	4.48	5.67

Table 1.4: The time to multiply 100,000-bit and 400,000-bit numbers with the refined Karatsuba method. Numbers with at most n_0 bits are multiplied with the school method. The top half shows the results for a SUN ULTRA-SPARC and the bottom half shows the outcome for an INTEL PENTIUM.

the school method for the three recursive multiplications. It is better to make a further Karatsuba step iff

$$cn_0 + 3a(\lceil n_0/2 \rceil + 1)^2 < an_0^2 \; .$$

Ignoring lower order terms this is equivalent to $cn_0 < an_0^2/4$ or $n_0 > 4c/a$. For $n > 4c/a$ is pays off to use a Karatsuba step and for $n \leq 4c/a$ it is better to resort directly to the school method. *We should therefore set n0 to* $4c/a$.

With respect to primitive operations we have $a = 3$ and $c = 12$. We should therefore set $n0 = 16$. With respect to running time we can read off a from Table 1.1, we should time the body of the Karatsuba without the recursive calls to determine c, and then compute $n0$ as $4c/a$.

Alternatively, we can simply time the revised Karatsuba algorithm for different values of $n0$ and then take the value which gives the smallest running time. Table **??** shows the time to multiply two n-bit numbers with the refined Karatsuba method for different values of $n0$ and $n = 10^5$ and $n = 4 \cdot 10^5$. The table suggests to use $n0 = 512$ for the ULTRA-SPARC and a somewhat larger value for the PENTIUM. We settled for $n0 = 512$ (because the author knows of no elegant way of making $n0$ depend on the machine on which he is running his experiments. Suggestions are welcome). The author expected to see a unimodal function in Table **??**, first decreasing and then increasing. Why are we seeing a function with at least two minima in the case of the ULTRA-SPARC.

Table 1.5 compares the running times of our refined Karatsuba implementation ($N0 = 512$) and the school method. We see that the refined implementation always wins. Table **??** compares the running times of our refined Karatsuba implementation ($n0 = 512$) and the LEDA integer multiplication. LEDA integer multiplication is also based on the Karatsuba method. It is about five times faster than the implementation given here. Where does the improved speed come from?

- The LEDA implementation has access to the internal representation of integers and hence can implement the school method more efficiently than we could. On some architectures the innermost loop is implemented in assembler code.

- The LEDA implementation has access to the internal representation and therefore can compute *a1*, *a0*, *b1*, and *b0* more efficiently than we can.

n	T_K	T_K/T'_K	T_S	T_S/T'_S	T_K/T_S
80000	0.57	–	1.22	–	0.4672
160000	1.67	2.93	4.74	3.885	0.3523
320000	5.04	3.018	19.32	4.076	0.2609
640000	15.6	3.095	100.6	5.205	0.1551

Table 1.5: The running time of the refined Karatsuba and the school method for integer multiplication: T_K is the running time of the refined Karatsuba method and T_S is the running time of the school method. T'_K and T'_S denote the running time of the preceding iteration. The table was produced on a 300 MHz SUN ULTRA-SPARC. Compare this table with Table 1.3. Observe that the refined Karatsuba method is always better than the school method.

n	T_K	T_K/T'_K	T_L	T_L/T'_L	T_K/T_L
80000	0.55	–	0.11	–	5
160000	1.7	3.091	0.32	2.909	5.313
320000	4.97	2.924	0.94	2.937	5.287
640000	15.03	3.024	2.86	3.043	5.255
1280000	45.3	3.014	8.6	3.007	5.267
2560000	–	–	26.13	3.038	–

Table 1.6: The time to multiply two n-bit numbers with our refined Karatsuba method ($n0 = 512$) and the multiplication algorithm of LEDA. The LEDA implementation is also based on the Karatsuba method. Observe the values in the next to last column.

- The LEDA implementation can optimize on the additions of the partial products by exploiting the fact that certain bits are known to be zero.

1.8 Even Faster Integer Multiplication

Is the Karatsuba method the fastest known method for integer multiplication? Much faster methods are known. The asymptotically most efficient algorithm is due to Schönhage and Strassen [SS71]. It multiplies n-bit integers in time $O(n \log n \log\log n)$. Their method is beyond the scope of this book.

We briefly sketch a generalization of the Karatsuba method which gives a running time of $O(n^{1+\varepsilon})$ for any positive ε. We split our n-bit numbers a and b into k parts $a_{k-1}, \ldots, a_0$ and $b_{k-1}, \ldots, b_0$ of $r = n/k$ bits each. Consider the polynomials

$$A(x) = \sum_{0 \le l \le k-1} a_l x^l \qquad \text{and} \qquad B(x) = \sum_{0 \le l \le k-1} b_l x^l$$

and their product

$$C(x) = A(x)B(x) = \sum_{0 \le l \le 2k-2} c_l x^l \ .$$

Since $a = A(2^r)$ and $b = B(2^r)$, we have $ab = C(2^r)$. We show how to compute the coefficients of C with $2k-1$ multiplications of r-bit integers and linear additional work. Having achieved this, we obtain the following recurrence for the number of primitive operations:

$$T(n) \leq (2k-1)T(n/k) + cn \ .$$

Theorem **??** implies that $T(n) = O(n^{\log_k(2k-1)}) = O(n^{1+\log_k(2-1/k)})$. For $k = 2$, the exponent is $\log 3 \approx 1.58$ as in Karatsuba, for $k = 4$, the exponent is $(\log 7)/2 \approx 1.403$, and for $k = 8$, the exponent is $((\log 15)/3 \approx 1.302$. The exponent can be made arbitrarily small by choosing k sufficiently large. More precisely, for $k = 2^x$, we have $1 + \log_k(2-1/k) \leq 1 + \log_k 2 = 1 + (\log 2)/\log k = 1 + 1/x$.

How do we compute the coefficients of C? Recall that a polynomial of degree $2k-2$ is uniquely determined by its values at $2k-1$ points. The values at $2k-1$ points are readily determined as

$$C(0) = A(0)B(0) \quad C(1) = A(1)B(1) \quad \ldots \quad C(2k-2) = A(2k-2)B(2k-2) \ .$$

The computation of each value essentially amounts to a multiplication of two r bit integers. The coefficients of C are obtained from the values of C by standard interpolation formulae. Their evaluation takes a linear amount of work. More precisely, consider the matrix $(2k-1) \times (2k-1)$ matrix V with $V_{ij} = i^j$ for $0 \leq i \leq 2k-2$ and $0 \leq j \leq 2k-2$. Then

$$\begin{pmatrix} C(0) \\ C(1) \\ \vdots \\ C(2k-2) \end{pmatrix} = V \cdot \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{2k-2} \end{pmatrix}$$

and hence

$$\begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{2k-2} \end{pmatrix} = V^{-1} \cdot \begin{pmatrix} C(0) \\ C(1) \\ \vdots \\ C(2k-2) \end{pmatrix} \ .$$

The entries of V^{-1} are fixed size rational numbers and hence the c's can be computed from the C's in $O(n)$ time.

Exercises:

1. Recapitulate the basics of C++.

2. Familiarize yourself with LEDA.

3. Describe an algorithm for the division of two integers (the algorithm is supposed to compute $a \div b$ and a mod b). What is its complexity?

4. (Neuner Probe): For an integer a let $neun(a)$ be the sum of the digits in the decimal notation of a modulo 9. Show: If $c = a \cdot b$ then $neun(c) = neun(a) \cdot neun(b)$ mod 9.

5. Write a program that generates all primes p with $2^{15} \leq p < 2^{16}$. Estimate the running time of your program before you run it. Will is run for a second or for an hour?

6. Formulate an algorithm for the multiplication of square matrices. What is the number of arithmetic operations required to multiply $n \times n$ matrices. If your answer is less than $O(n^3)$, check.

7. Solve the recurrences $T(1) = 1$ and $T(n) = 8T(n/2) + c \cdot n^2$ for $n \geq 2$ by repeated substitution.

8. Formulate the multiplication of square matrices as a recursive program, i.e., in order to multiply matrices A and B decompose both matrices into matrices of half the size and then express $A \cdot B$ in the products of the smaller matrices. Derive a recurrence for the number of arithmetic operations. Is the recurrence similar to the one studied in the preceding exercise?

9. (Continuation) Suppose somebody told you how to multiply two 2×2 matrices with 7 multiplications and some number of additions. What would this mean for the multiplication of matrices.

1.9 Corrections and Remarks

page 6, line 15: the error in $9 \cdot 9 = 72$ is not caught. In fact, whenever $c - a \cdot b$ is divisible by 9, c is accepted as the product of a and b. However, checking $9 \cdot 9 = 72$ module 5 will catch the error.

insert at the end of Section 1.3:

Our checker can be simplified somewhat by exploiting randomization even more. The simplification will not change the asymptotic running time of the checker, but will make it somewhat slower in practice.

In the checker above we selected 10 random k-bit primes p and then checked the equation $c \equiv a \cdot b \pmod{p}$. In order to select a random k-bit prime, we generated random k-bit integers until a prime was generated. This required us to program a primality check.

We now change the program as follows: we generate R random numbers p and check the equation $c \equiv a \cdot b \pmod{p}$; the appropriate value of R will be determined below.

⟨*simplified check multiplication*⟩≡

```
bool simplified_check_mult(const integer& a, const integer& b, const integer& c)
{ int n = leda_max(a.length(),b.length());
  if ( c.length() > 2*n ) return false;
  int k = 1;
  while ( !is_large_enough(k,n) ) k++;
  int R;
  ⟨determine R⟩
  for (int i = 0; i < R; i++)
  { integer p = integer::random(k);
    integer amod = a % p, bmod = b % p, cmod = c % p;
    if ( (amod * bmod) % p  != cmod ) return false;
  }
  return true;
}
```

Clearly, if the equation does not hold for one of the numbers chosen, c is not the product of a and b. The question is now what we can conclude if the equation is satisfied for R random numbers p. Our proof of Theorem 1 does not apply, since it crucially relies on the fact that we check the equations modulo primes.

But isn't checking modulo a non-prime even better than checking modulo a prime, since checking modulo a number p checks the equation concurrently modulo all prime factors of p. It seems to be difficult to make this reasoning precise, since the prime factors of random numbers are not random primes. For example, half of the numbers are even and hence the number two is very frequent as a prime factor.

We argue differently. We choose R such that there are at least ten primes among the chosen numbers. It is tempting to choose $R = 10k\ln 2$. Since the probability that a random k-bit number is prime is at least $1/(k\ln 2)$, this choice guarantees that the expected number of primes among the chosen numbers is at least $(10k\ln 2)/(k\ln 2) = 10$. However, it does not guarantee that we have ten primes. The right question to ask is the following: Which value of R guarantees that the probability of having less than 10 primes among R random k-bit numbers is less than 10^{-10}. We will learn in Section **??**, how to determine an R with this property. Assume we have determined R appropriately. Then the failure probability of our simplified checker is at most $2 \cdot 10^{-10}$. It might either fail because less than 10 of the chosen numbers are prime or because all chosen primes divide $D = c - a \cdot b$.

Bibliography

[KO63] A. Karatsuba and Yu. Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics—Doklady*, 7(7):595–596, January 1963.

[MN99] K. Mehlhorn and S. Näher. *The LEDA Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999. 1018 pages.

[MR95] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[NZ] I. Niven and H.S. Zuckermann. *Introduction to the Theory of Numbers*.

[Rab80] M.O. Rabin. A probabilistic algorithm for testing primality. *Journal of Number Theory*, 12, 1980.

[Sip98] M. Sipser. *Introduction to the Theory of Computation*. MIT Press, 1998.

[SS71] A. Schönhage and V. Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7:281–292, 1971.

[SS77] R. Solovay and V. Strassen. A fast Monte-Carlo test for primality. *SIAM Journal of Computing*, 6(1):84–85, 1977.