Complexity and Cryptography

Thomas Zeugmann

Hokkaido University
Laboratory for Algorithmics

https://www-alg.ist.hokudai.ac.jp/~thomas/COCRB/

Lecture 1: Addition, Multiplication

Motivation I

This course mainly deals with *sequential* computations. That is, within our underlying model, one and only one bit operation can be performed in one time step. So, we identify the time complexity with the number of bit operations to be performed.

Motivation I

This course mainly deals with *sequential* computations. That is, within our underlying model, one and only one bit operation can be performed in one time step. So, we identify the time complexity with the number of bit operations to be performed.

For many problems often quite different algorithms are known that solve them. Thus, one has to compare these algorithms in order to make a qualified choice which algorithm to use.

Motivation II

We distinguish between the *best-case*, *worst-case* and *average-case* analysis of algorithms.
The best-case deals with those inputs on which the algorithm achieves its fastest running time. The worst-case analysis provides an upper bound which is never exceeded independently of the inputs the algorithm is run with. So a worst-case analysis is of particular importance for all applications that are safety critical.

Motivation II

We distinguish between the *best-case*, *worst-case* and *average-case* analysis of algorithms.
The best-case deals with those inputs on which the algorithm achieves its fastest running time. The worst-case analysis provides an upper bound which is never exceeded independently of the inputs the algorithm is run with. So a worst-case analysis is of particular importance for all applications that are safety critical.

On the other hand, it may well be that those inputs, on which the considered algorithm achieves its worst-case, occur very seldom in practice. Provided the application is not safety critical, one should prefer the algorithm achieving the better average-case behavior.

Motivation III

Last but not least, for several applications one also needs guarantees that there is no algorithm solving a problem quickly on any input. We shall study such applications in the second part of our course when dealing with cryptography. But even without going into any detail here, it should be clear that deciphering a *password* must be as difficult on average as in the worst-case.

Motivation IV

We aim to express the complexity as a functions of the *size* of the input. This assumes that inputs are made by using any reasonable representation for the data on hand. For doing this, it is advantageous to neglect constant factors. Intuitively, this means that we are aiming at results saying that the running time of an algorithm is proportional to some function.

Motivation IV

We aim to express the complexity as a functions of the *size* of the input. This assumes that inputs are made by using any reasonable representation for the data on hand. For doing this, it is advantageous to neglect constant factors. Intuitively, this means that we are aiming at results saying that the running time of an algorithm is proportional to some function.

To formalize this approach, we need the following:
$\mathbb{N} = \{0, 1, 2, \ldots\}$ is the set of all natural numbers.
We set $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$.
$\mathbb{Z}$ is the set of all integers.
We use $\mathbb{Q}$ and $\mathbb{R}$ for the set of all rational numbers and real numbers, respectively.

Definitions I

The *non-negative real numbers* are denoted by $\mathbb{R}_{\geqslant 0}$.
For all real numbers y we define $\lfloor y \rfloor$, the *floor function*, to be the greatest integer less than or equal to y.

Similarly, $\lceil y \rceil$ denotes the smallest integer greater than or equal to y, i.e., the *ceiling function*.

Definitions I

The *non-negative real numbers* are denoted by $\mathbb{R}_{\geqslant 0}$.
For all real numbers y we define $\lfloor y \rfloor$, the *floor function*, to be the greatest integer less than or equal to y.

Similarly, $\lceil y \rceil$ denotes the smallest integer greater than or equal to y, i.e., the *ceiling function*.

Furthermore, for all numbers $y \in \mathbb{R}$ we write $|y|$ to denote the *absolute value* of y.

Order Notations I

Definition 1

Let $g\colon \mathbb{N} \to \mathbb{R}_{\geqslant 0}$ be any function. We define the following sets:

(1) $O(g(n)) =_{df} \{f \mid f\colon \mathbb{N} \to \mathbb{R}_{\geqslant 0}$ there are constants $c, n_0 > 0$ such that $0 \leqslant f(n) \leqslant cg(n)$ for all $n \geqslant n_0\}$,

Order Notations I

Definition 1

Let $g\colon \mathbb{N} \to \mathbb{R}_{\geqslant 0}$ be any function. We define the following sets:

(1) $O(g(n)) =_{df} \{f \mid f\colon \mathbb{N} \to \mathbb{R}_{\geqslant 0}$ there are constants $c, n_0 > 0$ such that $0 \leqslant f(n) \leqslant cg(n)$ for all $n \geqslant n_0\}$,

(2) $\Omega(g(n)) =_{df} \{f \mid f\colon \mathbb{N} \to \mathbb{R}_{\geqslant 0},$ there are constants $c, n_0 > 0$ such that $0 \leqslant cg(n) \leqslant f(n)$ for all $n \geqslant n_0\}$,

Order Notations I

Definition 1

Let $g\colon \mathbb{N} \to \mathbb{R}_{\geqslant 0}$ be any function. We define the following sets:

(1) $O(g(n)) =_{df} \{f \mid f\colon \mathbb{N} \to \mathbb{R}_{\geqslant 0}$ there are constants $c, n_0 > 0$ such that $0 \leqslant f(n) \leqslant cg(n)$ for all $n \geqslant n_0\}$,

(2) $\Omega(g(n)) =_{df} \{f \mid f\colon \mathbb{N} \to \mathbb{R}_{\geqslant 0},$ there are constants $c, n_0 > 0$ such that $0 \leqslant cg(n) \leqslant f(n)$ for all $n \geqslant n_0\}$,

(3) $o(g(n)) =_{df} \{f \mid f\colon \mathbb{N} \to \mathbb{R}_{\geqslant 0},$ for every constant $c > 0$ there exists a constant $n_0 > 0$ such that $0 \leqslant f(n) \leqslant cg(n)$ for all $n \geqslant n_0\}$,

Order Notations I

Definition 1

Let $g\colon \mathbb{N} \to \mathbb{R}_{\geqslant 0}$ be any function. We define the following sets:

(1) $O(g(n)) =_{df} \{f \mid f\colon \mathbb{N} \to \mathbb{R}_{\geqslant 0}$ there are constants $c, n_0 > 0$ such that $0 \leqslant f(n) \leqslant cg(n)$ for all $n \geqslant n_0\}$,

(2) $\Omega(g(n)) =_{df} \{f \mid f\colon \mathbb{N} \to \mathbb{R}_{\geqslant 0},$ there are constants $c, n_0 > 0$ such that $0 \leqslant cg(n) \leqslant f(n)$ for all $n \geqslant n_0\}$,

(3) $o(g(n)) =_{df} \{f \mid f\colon \mathbb{N} \to \mathbb{R}_{\geqslant 0},$ for every constant $c > 0$ there exists a constant $n_0 > 0$ such that $0 \leqslant f(n) \leqslant cg(n)$ for all $n \geqslant n_0\}$,

(4) $\Theta(g(n)) =_{df} \{f \mid f\colon \mathbb{N} \to \mathbb{R}_{\geqslant 0},$ there exist constants $c_1, c_2,$ and $n_0 > 0$ such that $0 \leqslant c_1 g(n) \leqslant f(n) \leqslant c_2 g(n)$ for all $n \geqslant n_0\}$.

Order Notations II

To indicate that a function f is a member of $O(g(n))$, we write $f(n) = O(g(n))$.
We adopt this convention to $\Omega(g(n))$, $o(g(n))$ and $\Theta(g(n))$.

Order Notations II

To indicate that a function f is a member of $O(g(n))$, we write $f(n) = O(g(n))$.
We adopt this convention to $\Omega(g(n))$, $o(g(n))$ and $\Theta(g(n))$.

Note that the O-notation expresses an *asymptotic upper bound* while the Ω-notation expresses an *asymptotic lower bound*. Looking at the definition above, we see that the Θ-notation establishes an *asymptotic tight bound*.

More Notations

Throughout this course, we write $\log n$ to denote the logarithm to the base 2,
$\ln n$ to denote the logarithm to the base e (where e is the Euler number),
and $\log_c n$ to denote the logarithm to the base c.

More Notations

Throughout this course, we write $\log n$ to denote the logarithm to the base 2,
$\ln n$ to denote the logarithm to the base e (where e is the Euler number),
and $\log_c n$ to denote the logarithm to the base c.

Now, we are ready to study the first algorithms. The basic arithmetic operations, i.e., *addition, subtraction, multiplication* and *division* are of fundamental importance. Therefore, we start with them.

Addition I

For measuring the complexity of addition, we use the length of the numbers to be added as the basic complexity parameter. Thus, in the following we assume as input two n-bit numbers

$$a = a_{n-1} \cdots a_0 \quad \text{and} \quad b = b_{n-1} \cdots b_0,$$

where $a_i, b_i \in \{0, 1\}$ for all $i = 0, \ldots, n-1$.

Addition I

For measuring the complexity of addition, we use the length of the numbers to be added as the basic complexity parameter. Thus, in the following we assume as input two n-bit numbers

$a = a_{n-1} \cdots a_0$ and $b = b_{n-1} \cdots b_0$,

where $a_i, b_i \in \{0, 1\}$ for all $i = 0, \ldots, n-1$.

The semantics of these numbers a and b is then

$$a = \sum_{i=0}^{n-1} a_i 2^i \text{ and}$$

$$b = \sum_{i=0}^{n-1} b_i 2^i.$$

Addition II

We have to compute the sum

$$s = a + b = \sum_{i=0}^{n} s_i 2^i \,, \quad s_i \in \{0,1\} \text{ for all } i = 0, \ldots, n \,.$$

Addition II

We have to compute the sum

$$s = a + b = \sum_{i=0}^{n} s_i 2^i \,, \quad s_i \in \{0,1\} \text{ for all } i = 0, \ldots, n \,.$$

Note that s is a number having at most $n+1$ bits.

Addition II

We have to compute the sum

$$s = a + b = \sum_{i=0}^{n} s_i 2^i \,, \quad s_i \in \{0,1\} \text{ for all } i = 0, \ldots, n \,.$$

Note that s is a number having at most $n+1$ bits.

We use $\wedge$ and $\vee$ to denote the logical *AND* and *OR* operation, respectively. By $\oplus$ we denote the Boolean $EX-OR$ function, i.e.,

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

Addition III

Using essentially the well-known school method for addition, we can express the s_i and the carry bits c_0 and c_{i+1}, $i = 0, \ldots, n-1$ as follows:

$$\begin{aligned} s_n &= c_n\,, \\ s_i &= a_i \oplus b_i \oplus c_i\,, \quad \text{where} \\ c_0 &= 0\,, \\ c_{i+1} &= (a_i \wedge b_i) \vee (a_i \wedge c_i) \vee (b_i \wedge c_i)\,. \end{aligned}$$

Addition III

Using essentially the well-known school method for addition, we can express the s_i and the carry bits c_0 and c_{i+1}, $i = 0, \ldots, n-1$ as follows:

$$\begin{aligned} s_n &= c_n\,, \\ s_i &= a_i \oplus b_i \oplus c_i\,, \quad \text{where} \\ c_0 &= 0\,, \\ c_{i+1} &= (a_i \wedge b_i) \vee (a_i \wedge c_i) \vee (b_i \wedge c_i)\,. \end{aligned}$$

Since we want to count the number of binary operations to be performed, we have to make a decision here. Either we can consider $\oplus$ also as a basic binary operation or we restrict ourselves to allow exclusively the logical *AND*, *OR* and the logical negation as basic binary operations.

Addition IV

In the latter case, we have to express $\oplus$ as

$$x \oplus y = (x \wedge \bar{y}) \vee (\bar{x} \wedge y)\,, \quad \text{where } x, y \in \{0, 1\}\,,$$

i.e., it takes 5 bit basic operations (2 negations, 2 times *AND* and one time *OR*) to express $\oplus$. So, the choice we make will only affect the constant and can thus be neglected here.

Addition IV

In the latter case, we have to express $\oplus$ as

$$x \oplus y = (x \wedge \bar{y}) \vee (\bar{x} \wedge y)\,, \quad \text{where } x, y \in \{0, 1\}\,,$$

i.e., it takes 5 bit basic operations (2 negations, 2 times *AND* and one time *OR*) to express $\oplus$. So, the choice we make will only affect the constant and can thus be neglected here.
We thus obtain the following theorem:

Theorem 1

The addition of two numbers a and b each having at most n bits can be performed in time $O(n)$.

Subtraction I

Subtraction can be handled within the scope of integer addition. For representing integers, we need an extra bit for representing the sign. We shall call such representations *AS*-numbers, where A stands for "absolute value" and S for "sign."

Definition 2

The *value* of an *AS*-number $x = (x_{n-1}, \ldots, x_0) \in \{0,1\}^n$ is defined as

$$V_{AS}(x) =_{df} (-1)^{x_{n-1}}(x_{n-2}2^{n-2} + \cdots + x_1 2 + x_0)\,.$$

Subtraction II

When dealing with addition of two *AS*-numbers x, y having at most n bits each, we distinguish the following cases:

Case 1. $x_{n-1} = y_{n-1}$.

That means x and y have the same sign. Hence, the sign of the sum is the same as the common sign of x and y. Moreover, the absolute value of the sum equals the sum of the absolute values of x and y. Thus, we can directly use addition algorithm presented above.

Subtraction II

When dealing with addition of two *AS*-numbers x, y having at most n bits each, we distinguish the following cases:

Case 1. $x_{n-1} = y_{n-1}$.

That means x and y have the same sign. Hence, the sign of the sum is the same as the common sign of x and y. Moreover, the absolute value of the sum equals the sum of the absolute values of x and y. Thus, we can directly use addition algorithm presented above.

Case 2. $x_{n-1} \neq y_{n-1}$.

Now, we have to compare the absolute values of x and y. The sign of the sum is equal to sign of the number having the bigger absolute value. Without loss of generality, let x' be the bigger absolute value and y' the smaller one. Hence, the absolute value of the sum is $x' - y'$.

Subtraction III

Consequently, we have to deal with two additional problems.
First, we have to show that subtraction of two n-bit numbers can be performed in time $O(n)$.

Subtraction III

Consequently, we have to deal with two additional problems.
First, we have to show that subtraction of two n-bit numbers can be performed in time $O(n)$.
Second, we have to prove that comparison of absolute values of two n-bit numbers can be done in time $O(n)$, too. We leave it as an exercise to show these two results, since we should have already some familiarity with these subjects (from technical computer science and/or electrical engineering).

Subtraction III

Consequently, we have to deal with two additional problems.
First, we have to show that subtraction of two n-bit numbers can be performed in time $O(n)$.
Second, we have to prove that comparison of absolute values of two n-bit numbers can be done in time $O(n)$, too. We leave it as an exercise to show these two results, since we should have already some familiarity with these subjects (from technical computer science and/or electrical engineering).

Thus, we leave it as an exercise to prove the following theorem:

Theorem 2

The subtraction of two AS-numbers a and b each having at most n bits can be performed in time $O(n)$.

Multiplication I

We deal here only with the multiplication of natural numbers. Given are two n-bit numbers $a = a_{n-1} \cdots a_0$ and $b = b_{n-1} \cdots b_0$, where $a_i, b_i \in \{0, 1\}$ for all $i = 0, \ldots, n-1$. Again, the semantics of these numbers a and b is then

$$a = \sum_{i=0}^{n-1} a_i 2^i \quad \text{and} \quad b = \sum_{i=0}^{n-1} b_i 2^i.$$

We have to compute the product ab. Taking into account that $a, b < 2^n$, one easily estimates $ab < 2^n \cdot 2^n = 2^{n+n} = 2^{2n}$. So, the product ab has at most $2n$ bits.

Multiplication II

First, we apply the usual school method for multiplication. Multiplication of two bits can be realized by the *AND* function. Thus, first we need n^2 many applications of *AND* to form the n summands and n costless shifts. Next, we have to add these n numbers. Using the same ideas as above, one easily verifies that this iterated sum takes another $O(n^2)$ many bit operations. Thus we have the following theorem:

Multiplication II

First, we apply the usual school method for multiplication. Multiplication of two bits can be realized by the *AND* function. Thus, first we need n^2 many applications of *AND* to form the n summands and n costless shifts. Next, we have to add these n numbers. Using the same ideas as above, one easily verifies that this iterated sum takes another $O(n^2)$ many bit operations. Thus we have the following theorem:

Theorem 3

The usual school algorithm for multiplying two numbers a and b each having at most n bits can be performed in time $O(n^2)$.

Multiplication III

Question

Can we do any better?

Multiplication III

Question

Can we do any better?

The affirmative answer will be provided by our next theorem which goes back to Anatoly Karatsuba.
We apply the method of *divide et impera* (*divide and conquer* in English) for sequential computations.

Multiplication III

Question

Can we do any better?

The affirmative answer will be provided by our next theorem which goes back to Anatoly Karatsuba.

We apply the method of *divide et impera* (*divide and conquer* in English) for sequential computations.

The general idea of the method *divide et impera* is as follows: The problem of size n is divided into a certain number of independent subproblems having the same type but having lower size. The solution of the original problem is obtained by combining the solutions of the subproblems in an appropriate manner. If this technique is applied recursively to the subproblems until problems of sufficiently small size arise, then the best effect will result.

Multiplication IV

Theorem 4

There is an algorithm for multiplying two numbers a *and* b *each having at most* n *bits that can be performed in time* $O(n^{\log 3})$.

Multiplication IV

Theorem 4

There is an algorithm for multiplying two numbers a and b each having at most n bits that can be performed in time $O(n^{\log 3})$.

Proof. Without loss of generality, we let $n = 2^k$. Then there exist numbers a_0, a_1 and b_0, b_1 such that

$$a = a_0 2^{n/2} + a_1 \quad \text{and } b = b_0 2^{n/2} + b_1 \, .$$

Then, it holds

$$\begin{aligned} ab &= (a_0 2^{n/2} + a_1)(b_0 2^{n/2} + b_1) \\ &= a_0 b_0 2^n + (a_0 b_1 + a_1 b_0) 2^{n/2} + a_1 b_1 \, . \end{aligned} \tag{1}$$

Multiplication V

Equation (1) reduces the problem of multiplying two n-bit numbers to *four* multiplications of numbers having at most $n/2$ bits and three additions as well as two shift operations. So, it does not help.

Multiplication V

Equation (1) reduces the problem of multiplying two n-bit numbers to *four* multiplications of numbers having at most $n/2$ bits and three additions as well as two shift operations. So, it does not help.

Karatsuba found a way to reduce the number of multiplications of numbers having at most $n/2$ bits from four to *three* by observing that

$$(a_0b_1 + a_1b_0) = (a_0 + a_1)(b_0 + b_1) - (a_0b_0 + a_1b_1)\,. \tag{2}$$

Using Equation (2), it is immediately clear that we only have to compute the three products a_0b_0, a_1b_1, and $(a_0 + a_1)(b_0 + b_1)$.

Multiplication VI

Therefore, we directly arrive at the following algorithm: For each of the Steps (1) through (6) perform the computation sequentially.

Karatsuba Multiplication

(1) $s_0 = a_0 + a_1$, $s_1 = b_0 + b_1$;

(2) $p_0 = a_0 b_0$, $p_1 = a_1 b_1$, $p_2 = s_0 s_1$;

(3) $t = p_0 + p_1$;

(4) $u = p_2 - t$, $\hat{u} = u 2^{n/2}$;

(5) $v = a_0 b_0 2^n + a_1 b_1$;

(6) $p = v + \hat{u}$.

Multiplication VII

The correctness of the algorithm above is obvious by the arguments provided before displaying it.

Multiplication VII

The correctness of the algorithm above is obvious by the arguments provided before displaying it.

We have to estimate the complexity of the Karatsuba multiplication.

Let $A(n)$ be the time for adding two n-bit numbers, and let $M(n)$ be the time for multiplying two n-bit numbers.

Multiplication VIII

Then, we can estimate the time complexity as follows:

(1) $2A(n/2)$;

(2) $2M(n/2) + M(n/2+1)$;

(3) $A(n)$;

(4) $A(n+2)$ and a costless shift.

(5) This step is costless, since it is only a concatenation of bits.

(6) $A\left(\frac{3}{2}n\right)$, since the $n/2$ lower bits of $\hat{u}$ are all 0.

Multiplication IX

Now, the main idea is to apply the Karatsuba multiplication algorithm recursively to itself. The only disturbing point here is that we have a subproblem (computing $s_0 s_1$) which is the multiplication of two numbers having $n/2 + 1$ bits (and not only $n/2$ as desired). We resolve it as follows:

Multiplication IX

Now, the main idea is to apply the Karatsuba multiplication algorithm recursively to itself. The only disturbing point here is that we have a subproblem (computing $s_0 s_1$) which is the multiplication of two numbers having $n/2 + 1$ bits (and not only $n/2$ as desired). We resolve it as follows:

Let $s_0 = x_0 2 + x_1$ and $s_1 = y_0 2 + y_1$, where $x_1,\ y_1 \in \{0, 1\}$ and $x_0,\ y_0$ are $n/2$-bit numbers. Then

$$\begin{aligned} s_0 s_1 &= (x_0 2 + x_1)(y_0 2 + y_1) \\ &= 4x_0 y_0 + 2(x_0 y_1 + x_1 y_0) + x_1 y_1 \,. \end{aligned} \tag{3}$$

Multiplication IX

Now, the main idea is to apply the Karatsuba multiplication algorithm recursively to itself. The only disturbing point here is that we have a subproblem (computing $s_0 s_1$) which is the multiplication of two numbers having $n/2 + 1$ bits (and not only $n/2$ as desired). We resolve it as follows:

Let $s_0 = x_0 2 + x_1$ and $s_1 = y_0 2 + y_1$, where $x_1,\ y_1 \in \{0, 1\}$ and $x_0,\ y_0$ are $n/2$-bit numbers. Then

$$\begin{aligned} s_0 s_1 &= (x_0 2 + x_1)(y_0 2 + y_1) \\ &= 4x_0 y_0 + 2(x_0 y_1 + x_1 y_0) + x_1 y_1 \,. \end{aligned} \tag{3}$$

Now, the multiplication of $x_0 y_0$ is a multiplication of $n/2$-bit numbers, i.e., it has costs $M(n/2)$.

Multiplication X

The remaining multiplications, i.e., x_0y_1, x_1y_0 and x_1y_1 can be directly realized by using $n+1$ AND gates, since $x_1,\ y_1 \in \{0,1\}$. Additionally, we have to include the costs for addition, i.e.,

$$A(n) + A(n/2) + 1\,. \tag{4}$$

Therefore, by using (3) and (4), we arrive at

$$M(n/2+1) \leqslant M(n/2) + A(n) + A(n/2) + 1\,.$$

Multiplication X

The remaining multiplications, i.e., x_0y_1, x_1y_0 and x_1y_1 can be directly realized by using $n+1$ AND gates, since $x_1,\ y_1 \in \{0,1\}$. Additionally, we have to include the costs for addition, i.e.,

$$A(n) + A(n/2) + 1\,. \tag{4}$$

Therefore, by using (3) and (4), we arrive at

$$M(n/2+1) \leqslant M(n/2) + A(n) + A(n/2) + 1\,.$$

Since addition can be realized by a sequential algorithm taking time $O(n)$, we conclude that there is a constant $\hat{c} > 0$ such that

$$M(n/2+1) \leqslant M(n/2) + \hat{c}n\,.$$

Consequently, there is a constant $c > 0$ such that

$$M(n) = \begin{cases} c, & \text{if } n = 1\,; \\ 3M(n/2) + cn, & \text{if } n > 1\,. \end{cases} \tag{5}$$

Multiplication XI

Now, it suffices to show that $M(n) = 3cn^{\log 3} - 2cn$ is a solution of the recursive Equation (5).

Multiplication XI

Now, it suffices to show that $M(n) = 3cn^{\log 3} - 2cn$ is a solution of the recursive Equation (5).

This is shown inductively. For the induction base, we get

$$M(1) = 3c1^{\log 3} - 2c = c\,.$$

Multiplication XI

Now, it suffices to show that $M(n) = 3cn^{\log 3} - 2cn$ is a solution of the recursive Equation (5).

This is shown inductively. For the induction base, we get

$$M(1) = 3c1^{\log 3} - 2c = c\,.$$

Assume the induction hypothesis for m. The induction step is from m to $2m$. Recall that $n = 2^k$, thus this induction step is justified. We have to show that $M(2m) = 3c(2m)^{\log 3} - 2c(2m)$.

$$\begin{aligned} M(2m) &= 3M(m) + 2cm \\ &= 3\left(3cm^{\log 3} - 2cm\right) + 2cm \\ &= 9cm^{\log 3} - 4cm = 9cm^{\log 3} - 2c(2m) \\ &= 3c2^{\log 3}m^{\log 3} - 2c(2m) = 3c(2m)^{\log 3} - 2c(2m)\,. \end{aligned}$$

Consequently, $M(n) = O(n^{\log 3})$. This proves the theorem. ∎

Discussion

Recall that $\log 3 \approx 1.59$. Thus, the Karatsuba multiplication is indeed much faster than the usual school method. Intuitively, the improvement is due to the fact that multiplication is more complex than addition. Hence, performing three instead of four multiplications results in roughly 25% saving of time.

Discussion

Recall that $\log 3 \approx 1.59$. Thus, the Karatsuba multiplication is indeed much faster than the usual school method. Intuitively, the improvement is due to the fact that multiplication is more complex than addition. Hence, performing three instead of four multiplications results in roughly 25% saving of time.

One can do even better. Schönhage and Strassen (1971) found a multiplication algorithm that takes time $O(n \log n \log\log n)$ for computing the product of two n-bit numbers. Very roughly speaking, their algorithm works via the fast Fourier transformation. However, their algorithm is only asymptotically faster, and n must be very large for achieving an improvement in practice. We thus omit this algorithm here.

And 2007 Fürer presented an even faster algorithm for integer multiplication.

Discussion

Recall that $\log 3 \approx 1.59$. Thus, the Karatsuba multiplication is indeed much faster than the usual school method. Intuitively, the improvement is due to the fact that multiplication is more complex than addition. Hence, performing three instead of four multiplications results in roughly 25% saving of time.

One can do even better. Schönhage and Strassen (1971) found a multiplication algorithm that takes time $O(n \log n \log\log n)$ for computing the product of two n-bit numbers. Very roughly speaking, their algorithm works via the fast Fourier transformation. However, their algorithm is only asymptotically faster, and n must be very large for achieving an improvement in practice. We thus omit this algorithm here.

And 2007 Fürer presented an even faster algorithm for integer multiplication.

Try to implement Karatsuba's multiplication.

Solutions of Recursive Equations

The technique of *divide et impera* yields recursive equations.
Can we say something about the general solvability of them?

Solutions of Recursive Equations

The technique of *divide et impera* yields recursive equations.
Can we say something about the general solvability of them?

Theorem 5

Let $a, b, c \in \mathbb{N}^+$. *Then the recursive equation*

$$T(n) = \begin{cases} b, & \text{if } n = 1\,; \\ aT\left(\frac{n}{c}\right) + bn, & \text{for all } n > 1\,, \end{cases}$$

where n *is a power of* c, *has the following solution:*

$$T(n) = \begin{cases} O(n), & \text{if } a < c\,; \\ O(n \log n), & \text{for all } a = c\,; \\ O(n^{\log_c a}), & \text{for all } a > c\,. \end{cases}$$

Proof I

Let n be a power of c, i.e., $n = c^k$. First, we show that

$$T(n) = bn \cdot \sum_{i=0}^{\log_c n} \left(\frac{a}{c}\right)^i$$

is a solution of the recursive equation given in Theorem 5. This is done inductively. For the induction basis let $n = 1$. Then, $\log_c 1 = 0$ and

$$\begin{aligned} T(1) &= b \quad \text{(by definition)} \\ &= b \cdot \left(\frac{a}{c}\right)^0 \quad \text{(multiplying by 1)} \\ &= b \cdot \sum_{i=0}^{0} \left(\frac{a}{c}\right)^0 . \end{aligned}$$

This shows the induction basis.

Proof II

Next, assume the induction hypothesis (abbr. IH) that

$$T(m) = bm \cdot \sum_{i=0}^{\log_c m} \left(\frac{a}{c}\right)^i$$

is a solution of the recursive equation for $n = m$. The induction step has to be done from m to cm, i.e., we have to show that

$$T(cm) = bcm \cdot \sum_{i=0}^{\log_c (cm)} \left(\frac{a}{c}\right)^i .$$

Proof III

$$
\begin{aligned}
T(cm) &= aT(m) + bcm \quad \text{(by definition)} \\
&= abm \cdot \sum_{i=0}^{\log_c m} \left(\frac{a}{c}\right)^i + bcm \quad \text{(by the IH)} \\
&= bm \cdot \sum_{i=0}^{\log_c m} \frac{a^{i+1}}{c^i} + bcm \\
&= bcm \cdot \sum_{i=0}^{\log_c m} \left(\frac{a}{c}\right)^{i+1} + bcm = bcm \cdot \sum_{i=0}^{(\log_c m)+1} \left(\frac{a}{c}\right)^i .
\end{aligned}
$$

Finally, $(\log_c m) + 1 = \log_c m + \log_c c = \log_c(cm)$. Thus,

$$
T(cm) = bcm \cdot \sum_{i=0}^{\log_c(cm)} \left(\frac{a}{c}\right)^i .
$$

Proof IV

Next, we distinguish the following cases:

Case 1. $a < c$

Then $a/c < 1$ and thus $\sum_{i=0}^{\infty} \left(\frac{a}{c}\right)^i$ is a convergent series. Hence, we have $T(n) = O(n)$.

Proof IV

Next, we distinguish the following cases:

Case 1. $a < c$

Then $a/c < 1$ and thus $\sum_{i=0}^{\infty} \left(\frac{a}{c}\right)^i$ is a convergent series. Hence, we have $T(n) = O(n)$.

Case 2. $a = c$

Consequently, $a/c = 1$, and thus

$$\sum_{i=0}^{\log_c n} \left(\frac{a}{c}\right)^i = 1 + \log_c n\,.$$

Therefore, $T(n) = O(n \log_c n) = O(n \log n)$.

Proof V

Case 3. $a > c$

$$
\begin{aligned}
T(n) &= bn \cdot \sum_{i=0}^{\log_c n} \left(\frac{a}{c}\right)^i = bn \cdot \frac{\left(\frac{a}{c}\right)^{1+\log_c n} - 1}{\frac{a}{c} - 1} \\
&\leqslant \hat{k}bn \cdot \left(\left(\frac{a}{c}\right)^{1+\log_c n} - 1\right) \quad \text{(note that } \hat{k} \text{ is a constant)} \\
&\leqslant \hat{k}bn \cdot \frac{a^{1+\log_c n}}{c^{1+\log_c n}} = \hat{k}b\frac{a}{c}a^{\log_c n} \quad \text{(recall that } n = c^{\log_c n}\text{)} \\
&= k' a^{\log_c n}, \quad \text{where } k' = \hat{k}ba/c \\
&= k' e^{\log_c n \ln a} = k' e^{\ln n \ln a \cdot (1/\ln c)}, \quad \left(\text{since } \log_c n = \frac{\ln n}{\ln c}\right) \\
&= k' e^{\ln n \log_c a}, \quad (\text{since } (\ln a)/\ln c = \log_c a) \\
&= k' n^{\log_c a} = O(n^{\log_c a}). \qquad \blacksquare
\end{aligned}
$$

Thank you!

Anatolii Alexeevich Karatsuba