Part IV: Complexity of problems and algorithms

Fundamental issues of computation

A successful search for better and better algorithms naturally leads to the question "Is there a best algorithm?", whereas an unsuccessful search leads one to ask apprehensively: "Is there *any* algorithm (of a certain type) to solve this problem?" These questions turned out to be difficult and fertile. Historically, the question about the *existence* of an algorithm came first, and led to the concepts of computability and decidability in the 1930s. The question about a "best" algorithm led to the development of complexity theory in the 1960s.

The study of these fundamental issues of computation requires a mathematical arsenal that includes mathematical logic, discrete mathematics, probability theory, and certain parts of analysis, in particular asymptotics. We introduce a few of these topics, mostly by example, and illustrate the use of mathematical techniques of algorithm analysis on the important problem of sorting.

15. Computability and complexity

Learning objectives:

- algorithm
- computability
- RISC: Reduced Instruction Set Computer
- Almost nothing is computable.
- The halting problem is undecidable.
- complexity of algorithms and problems
- Strassen's matrix multiplication

Models of computation: the ultimate RISC

Algorithm and *computability* are originally intuitive concepts. They can remain intuitive as long as we only want to show that some specific result can be computed by following a specific algorithm. Almost always an informal explanation suffices to convince someone with the requisite background that a given algorithm computes a specified result. We have illustrated this informal approach throughout Part III. Everything changes if we wish to show that a desired result is *not computable*. The question arises immediately: "What tools are we allowed to use?" Everything is computable with the help of an oracle that knows the answers to all questions. The attempt to prove negative results about the nonexistence of certain algorithms forces us to agree on a rigorous definition of *algorithm*.

The question "What can be computed by an algorithm, and what cannot?" was studied intensively during the 1930s by Emil Post (1897–1954), Alan Turing (1912–1954), Alonzo Church (1903), and other logicians. They defined various formal models of computation, such as production systems, Turing machines, and recursive functions, to capture the intuitive concept of "computation by the application of precise rules". All these different formal models of computation turned out to be equivalent. This fact greatly strengthens Church's thesis that the intuitive concept of algorithm is formalized correctly by any one of these mathematical systems.

We will not define any of these standard models of computation. They all share the trait that they were designed to be conceptually simple: their primitive operations are chosen to be as weak as possible, as long as they retain their property of being universal computing systems in the sense that they can simulate any computation performed on any other machine. It usually comes as a surprise to novices that the set of primitives of a universal computing machine can be so simple as long as these machines possess two essential ingredients: *unbounded memory* and *unbounded time*.

Most simulations of a powerful computer on a simple one share three characteristics: it is straightforward in principle, it involves laborious coding in practice, and it explodes the space and time requirements of a computation. The weakness of the primitives, desirable from a theoretical point of view, has the consequence that as simple an operation as integer addition becomes an exercise in programming.

The model of computation used most often in algorithm analysis is significantly more powerful than a Turing machine in two respects: (1) its memory is not a tape, but an array, and (2) in one primitive operation it can deal with numbers of arbitrary size. This model of computation is called *random access machine*, abbreviated as RAM. A RAM is essentially a *random access memory*, also abbreviated as RAM, of unbounded capacity, as suggested in Exhibit 15.1. The memory consists of an infinite array of memory cells, addressed 0, 1, 2, ... Each cell can hold a number, say an integer, of arbitrary size, as the arrow pointing to the right suggests.

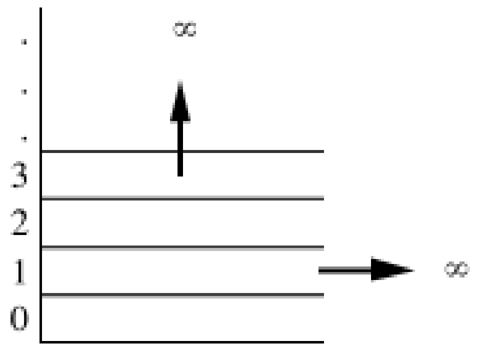

Exhibit 15.1: RAM - unbounded address space, unbounded cell size.

A RAM has an arithmetic unit and is driven by a program. The meaning of the word *random* is that any memory cell can be accessed in unit time (as opposed to a tape memory, say, where access time depends on distance). A further crucial assumption in the RAM model is that an arithmetic operation (+, –, ·, /) also takes unit time, regardless of the size of the numbers involved. This assumption is unrealistic in a computation where numbers may grow very large, but often is a useful assumption. As is the case with all models, the responsibility for using them properly lies with the user. To give the reader the flavor of a model of computation, we define a RAM whose architecture is rather similar to real computers, but is unrealistically simple.

The ultimate RISC

RISC stands for *Reduced Instruction Set Computer*, a machine that has only a few types of instructions built into the hardware. What is the minimum number of instructions a computer needs to be universal? In theory, one.

Consider a stored-program computer of the "von Neumann type" where data and program are stored in the same memory (John von Neumann, 1903–1957). Let the random access memory (RAM) be "doubly infinite": There is a *countable infinity* of memory cells addressed 0, 1, ... , each of which can hold an integer of arbitrary size, or an instruction. We assume that the constant 1 is hardwired into memory cell 1; from 1 any other integer can be constructed. There is a single type of "three-address instruction" which we call "subtract, test and jump", abbreviated as

STJ x, y, z

where x, y, and z are addresses. Its semantics is equivalent to

```
STJ  x, y, z   ⇔   x := x – y;  if  x ≤ 0  then  goto z;
```

x, y, and z refer to cells Cx, Cy, and Cz. The contents of Cx and Cy are treated as data (an integer); the contents of Cz, as an instruction (Exhibit 15.2).

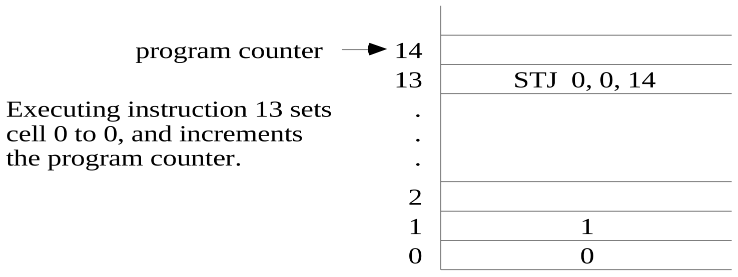
program counter
14
13
STJ 0, 0, 14
Executing instruction 13 sets cell 0 to 0, and increments the program counter.
.
.
.
2
1
1
0
0

Exhibit 15.2: Stored program computer: data and instructions share the memory.

Since this RISC has just one type of instruction, we waste no space on an op-code field. An instruction contains three addresses, each of which is an unbounded integer. In theory, fortunately, three unbounded integers can be packed into the same space required for a single unbounded integer. In the following exercise, this simple idea leads to a well-known technique introduced into mathematical logic by Kurt Gödel (1906 – 1978).

Exercise: Gödel numbering

(a) Motel Infinity has a countable infinity of rooms numbered 0, 1, 2, ... Every room is occupied, so the sign claims "No Vacancy". Convince the manager that there is room for one more person.

(b) Assume that a memory cell in our RISC stores an integer as a sign bit followed by a sequence d_0, d_1, d_2, ... of decimal digits, least significant first. Devise a scheme for storing three addresses in one cell.

(c) Show how a sequence of positive integers i_1, i_2, ... , i_n of arbitrary length n can be encoded in a single natural number j: Given j, the sequence can be uniquely reconstructed. Gödel's solution:

$$j = 2^{i_1} \cdot 3^{i_2} \cdot 5^{i_3} \cdot \ldots \cdot (n\text{-th prime})^{i_n}.$$

Basic program fragments

This computer is best understood by considering program fragments for simple tasks. These fragments implement simple operations, such as setting a variable to a given constant, or the assignment operator, that are given as primitives in most programming languages. Programming these fragments naturally leads us to introduce basic concepts of assembly language, in particular symbolic and relative addressing.

Set the content of cell 0 to 0:

```
STJ  0, 0, .+1
```

Whatever the current content of cell 0, subtract it from itself to obtain the integer 0. This instruction resides at some address in memory, which we abbreviate as '.', read as "the current value of the program counter". '.+1' is the next address, so regardless of the outcome of the test, control flows to the next instruction in memory.

a := b, where a and b are symbolic addresses. Use a temporary variable t:

```
STJ  t, t, .+1 { t := 0 }
STJ  t, b, .+1 { t := –b }
STJ  a, a, .+1 { a := 0 }
STJ  a, t, .+1 { a := –t, so now a = b }
```

Exercise: a program library

(a) Write RISC programs for a:= b + c, a := b · c, a := b div c, a := b mod c, a := |b|, a : = min(b, c), a := gcd(b, c).

(b) Show how this RISC can compute with rational numbers represented by a pair [a, b] of integers denoting numerator and denominator.

(c) (Advanced) Show that this RISC is universal, in the sense that it can simulate any computation done by any other computer.

The exercise of building up a RISC program library for elementary functions provides the same experience as the equivalent exercise for Turing machines, but leads to the goal much faster, since the primitive STJ is much more powerful than the primitives of a Turing machine.

The purpose of this section is to introduce the idea that conceptually simple models of computation are as powerful, in theory, as much more complex models, such as a high-level programming language. The next two sections demonstrate results of an opposite nature: Universal computers, in the sense we have just introduced, are subject to striking limitations, even if we remove any limit on the memory and time they may use. We prove the existence of noncomputable functions and show that the "halting problem" is undecidable.

The theory of computability was developed in the 1930s, and greatly expanded in the 1950s and 1960s. Its basic ideas have become part of the foundation that any computer scientist is expected to know. Computability theory is not directly useful. It is based on the concept "computable in principle" but offers no concept of a "feasible computation". Feasibility, rather than "possible in principle", is the touchstone of computer science. Since the 1960s, a theory of the complexity of computation is being developed, with the goal of partitioning the range of computability into complexity classes according to time and space requirements. This theory is still in full development and breaking new ground, in particular in the area of concurrent computation. We have used some of its concepts throughout Part III and continue to illustrate these ideas with simple examples and surprising results.

Almost nothing is computable

Consider as a model of computation any programming language, with the fictitious feature that it is implemented on a machine with infinite memory and no operational time limits. Nevertheless we reach the conclusion that "almost nothing is computable". This follows simply from the observation that there are fewer programs than problems to be solved (functions to be computed). Both the number of programs and the number of functions are infinite, but the latter is an infinity of higher cardinality.

A programming language L is defined over an alphabet $A = \{a_1, a_2, \ldots, a_k\}$ of k characters. The set of programs in L is a subset of the set A^* of all strings over A. A^* is *countable*, and so is its subset L, as it is in one-to-one correspondence with the natural numbers under the following mapping:

1. Generate all strings in A^* in order of increasing length and, in case of equal length, in lexicographic order.
2. Erase all strings that do not represent a program according to the syntax rules of L.
3. Enumerate the remaining strings in the originally given order.

Among all programs in L we consider only those which compute a (partial) function from the set $N = \{1, 2, 3, \ldots\}$ of natural numbers into N. This can be recognized by their heading; for example,

```
function f(x: N): N;
```

As this is a subset of L, there exist only countably many such programs.

However, there are uncountably many functions $f: N \to N$, as Georg Cantor (1845–1918) proved by his famous diagonalization argument. It starts by assuming the opposite, that the set $\{f \mid f: N \to N\}$ is countable, then derives a contradiction. If there were only a countable number of such functions, we could enumerate all of them according to the following scheme:

$$\begin{array}{c|ccccc}
 & 1 & 2 & 3 & 4 & \dots \\
\hline
f_1 & f_1(1) & f_1(2) & f_1(3) & f_1(4) & \\
f_2 & f_2(1) & f_2(2) & f_2(3) & f_2(4) & \\
f_3 & f_3(1) & f_3(2) & f_3(3) & f_3(4) & \\
f_4 & f_4(1) & f_4(2) & f_4(3) & f_4(4) & \\
\vdots & & & & &
\end{array}$$

Construct a function $g: N \to N$, $g(i) = fi(i) + 1$, which is obtained by adding 1 to the diagonal elements in the scheme above. Hence g is different from each f_i, at least for the argument i: $g(i) \neq fi(i)$. Therefore, our assumption that we have enumerated all functions $f: N \to N$ is wrong. Since there exists only a countable infinity of programs, but an uncountable infinity of functions, almost all functions are noncomputable.

The halting problem is undecidable

If we could predict, for any program P executed on any data set D, whether P terminates or not (i.e. whether it will get into an infinite loop), we would have an interesting and useful technique. If this prediction were based on rules that prescribe exactly how the pair (P, D) is to be tested, we could write a program H for it. A fundamental result of computability theory states that under reasonable assumptions about the model of computation, such a *halting program* H cannot exist.

Consider a programming language L that contains the constructs we will use: mainly recursive procedures and procedure parameters. Consider all procedures P in L that have no parameters, a property that can be recognized from the heading

```
procedure P;
```

This simplifies the problem by avoiding any data dependency of termination.

Assume that there exists a program H in L that takes as argument any parameterless procedure P in L and decides whether P halts or loops (i.e. runs indefinitely):

$$H(P) = \begin{cases} \text{true} & \text{if P halts,} \\ \text{false} & \text{if P loops.} \end{cases}$$

Consider the behavior of the following parameterless procedure X:

```
procedure X;
begin  while  H(X)  do;  end;
```

Consider the reference of X to itself; this trick corresponds to the diagonalization in the previous example. Consider further the loop

```
while  H(X)  do;
```

which is infinite if H(X) returns true (i.e. exactly when X should halt) and terminates if H(X) returns false (i.e. exactly when X should run forever). This trick corresponds to the change of the diagonal $g(i) = f_i(i) + 1$. We obtain:

By definition of X:

$$H(X) = \begin{cases} \text{true} & \text{if X halts,} \\ \text{false} & \text{if X loops.} \end{cases}$$

By construction of X:

$$H(X) = \begin{cases} \text{true} & \text{if X loops,} \\ \text{false} & \text{if X halts.} \end{cases}$$

The fiendishly crafted program X traps H in a web of contradictions. We blame the weakest link in the chain of reasoning that leads to this contradiction, namely the unsupported assumption of the existence of a halting program H. This proves that the halting problem is undecidable.

Computable, yet unknown

In the preceding two sections we have illustrated the limitations of computability: clearly stated questions, such as the halting problem, are undecidable. This means that the halting question cannot be answered, *in general*, by any computation no matter how extensive in time and space. There are, of course, lots of individual halting questions that can be answered, asserting that a particular program running on a particular data set terminates, or fails to do so. To illuminate this key concept of theoretical computer science further, the following examples will highlight a different type of practical limitation of computability.

Computable or decidable is a concept that naturally involves *one* algorithm and a *denumerably infinite* set of problems, indexed by a parameter, say n. Is there a uniform procedure that will solve any one problem in the infinite set? For example, the "question" (really a denumerable infinity of questions) "Can a given integer n > 2 be expressed as the sum of two primes?" is decidable because there exists the algorithm 's2p' that will answer any single instance of this question:

```
procedure s2p(n: integer): boolean;
  { for n>2, s2p(n) returns true if n is the sum of two primes,
    false otherwise }

  function p(k: integer): integer;
    { for k>0, p(k) returns the k-th prime: p(1) = 2, p(2) = 3, p(3)
= 5, … }
  end;

begin
  for all i, j such that p(i) < n and p(j )< n do
    if  p(i) + p(j) = n  then  return(true);
  return(false);
end;  { s2p }
```

So the general question "Is any given integer the sum of two primes?" is solved readily by a simple program. A *single* related question, however, is much harder: "Is every even integer >2 the sum of two primes?" Let's try:

4 = 2 + 2, 6 = 3 + 3, 8 = 5 + 3, 10 = 7 + 3 = 5 + 5, 12 = 7 + 5,

14 = 11 + 3 = 7 + 7, 16 = 13 + 3 = 11 + 5, 18 = 13 + 5 = 11 + 7,

20 = 17 + 3 = 13 + 7, 22 = 19 + 3 = 17 + 5 = 11 + 11,

24 = 19 + 5 = 17 + 7 = 13 + 11, 26 = 23 + 3 = 21 + 5 = 19 + 7 = 13 + 13,

28 = 23 + 5 = 17 + 11, 30 = 23 + 7 = 19 + 11 = 17 + 13,

32 = 29 + 3 = 19 + 13, 34 = 31 + 3 = 29 + 5 = 23 + 11 = 17 + 17,

36 = 33 + 3 = 31 + 5 = 29 + 7 = 23 + 13 = 19 + 17.

A bit of experimentation suggests that the number of distinct representations as a sum of two primes increases as the target integer grows. Christian Goldbach (1690–1764) had the good fortune of stating the plausible conjecture "yes" to a problem so hard that it has defied proof or counterexample for three centuries.

One might ask: Is the Goldbach conjecture decidable? The straight answer is that the concept of decidability does not apply to a single yes/no question such as Goldbach's conjecture. Asking for an algorithm that tells us whether the conjecture is true or false is meaninglessly trivial. Of course, there is such an algorithm! If the Goldbach conjecture is true, the algorithm that says 'yes' decides. If the conjecture is false, the algorithm that says 'no' will do the job. The problem that we *don't know* which algorithm is the right one is quite compatible with saying that *one of those two* is the right algorithm. If we package two trivial algorithms into one, we get the following trivial algorithm for deciding Goldbach's conjecture:

```
function GoldbachOracle(): boolean:
begin  return(GoldbachIsRight)  end;
```

Notice that 'GoldbachOracle' is a function without arguments, and 'GoldbachIsRight' is a boolean constant, either true or false. Occasionally, the stark triviality of the argument above is concealed so cleverly under technical jargon as to sound profound. Watch out to see through the following plot.

Let us call an even integer > 2 that is *not* a sum of two primes a *counterexample*. None have been found as yet, but we can certainly reason about them, whether they exist or not. Define the

```
function G(k: cardinal): boolean;
```

as follows:

$$G(k) = \begin{cases} \text{true} & \text{if the number of counterexamples is } \le k, \\ \text{false} & \text{otherwise.} \end{cases}$$

Goldbach's conjecture is equivalent to G(0) = true. The (implausible) rival conjecture that there is exactly one counterexample is equivalent to G(0) = false, G(1) = true. Although we do not know the value of G(k) for any single k, the definition of G tells us a lot about this artificial function, namely:

if G(i) = true for any i, then G(k) = true for all k > i.

With such a strong monotonicity property, how can G look?

1. If Goldbach is right, then G is a constant: G(k) = true for all k.
2. If there are a finite number i of exceptions, then G is a step function:

   G(k) = false for k < i, G(k) = true for k ≥ i.
3. If there is an infinite number of exceptions, then G is again a constant:

   G(k) = false for all k.

Each of the infinitely many functions listed above is obviously computable. Hence G is computable. The value of G(0) determines truth or falsity of Goldbach's conjecture. Does that help us settle this time-honored mathematical puzzle? Obviously not. All we have done is to rephrase the honest statement with which we started this section, "The answer is yes or no, but I don't know which" by the circuitous "The answer can be obtained by evaluating a computable function, but I don't know which one".

Multiplication of complex numbers

Let us turn our attention from noncomputable functions and undecidable problems to very simple functions that are obviously computable, and ask about their complexity: How many primitive operations must be executed in evaluating a specific function? As an example, consider arithmetic operations on real numbers to be primitive, and consider the product z of two complex numbers x and y:

$x = x_1 + i \cdot x_2$ and $y = y_1 + i \cdot y_2$,

$x \cdot y = z = z_1 + i \cdot z_2$.

The complex product is defined in terms of operations on real numbers as follows:

$z_1 = x_1 \cdot y_1 - x_2 \cdot y_2$,

$z_2 = x_1 \cdot y_2 + x_2 \cdot y_1$.

It appears that one complex multiplication requires four real multiplications and two real additions/subtractions. Surprisingly, it turns out that multiplications can be traded for additions. We first compute three intermediate variables using one multiplication for each, and then obtain z by additions and subtractions:

$p_1 = (x_1 + x_2) \cdot (y_1 + y_2)$,

$p_2 = x_1 \cdot y_1$,

$p_3 = x_2 \cdot y_2$,

$z_1 = p_2 - p_3$, $z_2 = p_1 - p_2 - p_3$.

This evaluation of the complex product requires only 3 real multiplications, but 5 real additions / subtractions. This trade of one multiplication for three additions may not look like a good deal in practice, because many computers have arithmetic chips with fast multiplication circuitry. In theory, however, the trade is clearly favorable. The cost of an addition grows linearly in the number of digits, whereas the cost of a multiplication using the standard method grows quadratically. The key idea behind this algorithm is that "linear combinations of k products of sums can generate more than k products of simple terms". Let us exploit this idea in a context where it makes a real difference.

Complexity of matrix multiplication

The *complexity of an algorithm* is given by its time and space requirements. Time is usually measured by the number of operations executed, space by the number of variables needed at any one time (for input, intermediate results, and output). For a given algorithm it is often easy to count the number of operations performed in the worst and in the best case; it is usually difficult to determine the average number of operations performed (i.e. averaged over all possible input data). Practical algorithms often have time complexities of the order O(log n), O(n^2), O(n · log n), O(n^2), and space complexity of the order O(n), where n measures the size of the input data.

The *complexity of a problem* is defined as the minimal complexity of all algorithms that solve this problem. It is almost always difficult to determine the complexity of a problem, since all possible algorithms must be considered, including those yet unknown. This may lead to surprising results that disprove obvious assumptions.

The complexity of an algorithm is an upper bound for the complexity of the problem solved by this algorithm. An algorithm is a witness for the assertion: You need at most this many operations to solve this problem. A specific algorithm never provides a *lower bound* on the complexity of a problem— it cannot extinguish the hope for a more efficient algorithm. Occasionally, algorithm designers engage in races lasting decades that result in (theoretically) faster and faster algorithms for solving a given problem. Volker Strassen started such a race with his 1969 paper

"Gaussian Elimination Is Not Optimal" [Str 69], where he showed that matrix multiplication requires fewer operations than had commonly been assumed necessary. The race has not yet ended.

The obvious way to multiply two n × n matrices uses three nested loops, each of which is iterated n times, as we saw in a transitive hull algorithm in the chapter, “Matrices and graphs: transitive closure”. The fact that the obvious algorithm for matrix multiplication is of time complexity $\Theta(n^3)$, however, does not imply that the matrix multiplication problem is of the same complexity.

Strassen's matrix multiplication

The standard algorithm for multiplying two n × n matrices needs n^3 scalar multiplications and $n^2 \cdot (n - 1)$ additions; for the case of 2 × 2 matrices, eight multiplications and four additions. Seven scalar multiplications suffice if we accept 18 additions/subtractions.

$$\begin{bmatrix} r_{11} & r_{12} \\ r_{21} & r_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}.$$

Evaluate seven expressions, each of which is a product of sums:

$p_1 = (a_{11} + a_{22}) \cdot (b_{11} + b_{22})$,

$p_2 = (a_{21} + a_{22}) \cdot b_{11}$

$p_3 = a_{11} \cdot (b_{12} - b_{22})$

$p_4 = a_{22} \cdot (-b_{11} + b_{21})$ $p_5 = (a_{11} + a_{12}) \cdot b_{22}$

$p_6 = (-a_{11} + a_{21}) \cdot (b_{11} + b_{12}) p_7 = (a_{12} - a_{22}) \cdot (b_{21} + b_{22})$.

The elements of the product matrix are computed as follows:

$r_{11} = p_1 + p_4 - p_5 + p_7$,

$r_{12} = p_3 + p_5$,

$r_{21} = p_2 + p_4$, $r_{22} = p_1 - p_2 + p_3 + p_6$.

This algorithm does not rely on the commutativity of scalar multiplication. Hence it can be generalized to n × n matrices using the divide-and-conquer principle. For reasons of simplicity consider n to be a power of 2 (i.e. $n = 2^k$); for other values of n, imagine padding the matrices with rows and columns of zeros up to the next power of 2. An n × n matrix is partitioned into four n/2 × n/2 matrices:

$$\begin{bmatrix} R_{11} & R_{12} \\ R_{21} & R_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}.$$

The product of two n × n matrices by Strassen's method requires seven (not eight) multiplications and 18 additions/subtractions of n/2 × n/2 matrices. For large n, the work required for the 18 additions is negligible compared to the work required for even a single multiplication (why?); thus we have saved one multiplication out of eight, asymptotically at no cost.

Each n/2 × n/2 matrix is again partitioned recursively into four n/4 × n/4 matrices; after $\log_2 n$ partitioning steps we arrive at 1 × 1 matrices for which matrix multiplication is the primitive scalar multiplication. Let T(n) denote the number of scalar arithmetic operations used by Strassen's method for multiplying two n × n matrices. For n > 1, T(n) obeys the recursive equation

$$T(n) = 7 \cdot T\left(\frac{n}{2}\right) + 18 \cdot \left(\frac{n}{2}\right)^2 .$$

If we are only interested in the leading term of the solution, the constants 7 and 2 justify omitting the quadratic term, thus obtaining

$$\begin{aligned} T(n) &= 7 \cdot T\left(\frac{n}{2}\right) = 7 \cdot 7 \cdot T\left(\frac{n}{4}\right) = 7 \cdot 7 \cdot 7 \cdot T\left(\frac{n}{8}\right) = \ldots \\ &= 7^{\log_2 n} \cdot T(1) = n^{\log_2 7} \cdot T(1) \approx n^{2.81} \cdot T(1). \end{aligned}$$

Thus the number of primitive operations required to multiply two n × n matrices using Strassen's method is proportional to $n^{2.81}$, a statement that we abbreviate as "Strassen's matrix multiplication takes time $\Theta(n^{2.81})$".

Does this asymptotic improvement lead to a more efficient program in practice? Probably not, as the ratio

$$\frac{n^3}{n^{2.81}} \approx n^{0.2} = \sqrt[5]{n}$$

grows too slowly to be of practical importance: For n ≈ 1000, for example, we have $\sqrt[5]{1024}$ = 4 (remember: 2^{10} = 1024). A factor of 4 is not to be disdained, but there are many ways to win or lose a factor of 4. Trading an algorithm with simple code, such as straightforward matrix multiplication, for another that requires more elaborate bookkeeping, such as Strassen's, can easily result in a fourfold increase of the constant factor that measures the time it takes to execute the body of the innermost loop.

Exercises

1. Prove that the set of all ordered pairs of integers is countably infinite.
2. A *recursive function* is defined by a finite set of rules that specify the function in terms of variables, nonnegative integer constants, increment ('+1'), the function itself, or an expression built from these by composition of functions. As an example, consider *Ackermann's function* defined as $A(n) = A_n(n)$ for n ≥ 1, where $A_k(n)$ is determined by

   $A_k(1) = 2$ for k ≥ 1

   $A_1(n) = A_1(n-1) + 2$ for n ≥ 2

   $A_k(n) = A_{k-1}(A_k(n-1))$ for k ≥ 2

   (a) Calculate A(1) , A(2) , A(3), A(4).

   (b) Prove that

   $A_k(2) = 4$ for k ≥ 1,

   $A_1(n) = 2 \cdot n$ for n ≥ 1,

   $A_2(n) = 2^n$ for n ≥ 1,

   $A_3(n) = 2^{A_3(n-1)}$ for n ≥ 2.

   (c) Define the inverse of Ackermann's function as

   $\alpha(n) = \min\{m: A(m) \geq n\}$.

   Show that $\alpha(n) \leq 3$ for $n \leq 16$, that $\alpha(n) \leq 4$ for n at most a "tower" of 65536 2's, and that $\alpha(n) \to \infty$ as $n \to \infty$.

3. Complete Strassen's algorithm by showing how to multiply $n \times n$ matrices when n is not an exact power of 2.
4. Assume that you can multiply 3×3 matrices using k multiplications. What is the largest k that will lead to an asymptotic improvement over Strassen's algorithm?
5. A permutation matrix P is an $n \times n$ matrix that has exactly one '1' in each row and each column; all other entries are '0'. A permutation matrix can be represented by an array

   ```
   var a: array[1 .. n] of integer;
   ```

   as follows: a[i] = j if the i-th row of P contains a '1' in the j-th column.
6. Prove that the product of two permutation matrices is again a permutation matrix.
7. Design an algorithm that multiplies in time $\Theta(n)$ two permutation matrices given in the array representation above, and stores the result in this same array representation.