CHAPTER 14

APPLICATIONS AND EXAMPLES

In the present chapter we illustrate how the various methods we have introduced in this book can be applied to

1. study complexity problems for algorithms and
2. prove correctness of programs.

There are several criteria by which to evaluate the complexity of an algorithm:

- Space complexity: for instance, the number of variables or instructions, the size of the variables, the space allowed in the memory, etc.
- Time complexity: the length of time required to execute the program; this will usually depend on the input data, and several notions of complexity may be of interest:
  - average case complexity,
  - worst-case complexity (namely, the complexity for the input data resulting in the longest possible computation),
  - best-case complexity (namely, the complexity for the input data resulting in the shortest possible computation).

One of the goals of a 'discrete mathematics' course is to build tools for evaluating these various notions of complexity. We illustrate these tools by studying the average complexity of Quicksort and the worst-case complexity of a simple algorithm: Euclid's algorithm. We will only sketch the various tools in the present chapter; for further details the reader is refferred to the chapters in which each tool is defined. This chapter owes much to the class handouts by Jean Claude Raoult on the same subject; we heartily thank him.

On the other hand, to prove the correctness of programs, most existing techniques boil down to proofs by induction and searches for loop invariants. We illustrate these techniques in the remainder of the chapter.

14.1 Quicksort

Intuitively, the complexity of an algorithm corresponds to the size $t(i_n)$ of the computations on a size n input i_n. However, $t(i_n)$ is usually not the same for all size n inputs. For i_n ranging over size n inputs:

- The worst-case complexity is the maximum value of all the $t(i_n)$s.
- The average complexity is the average value of the $t(i_n)$s.
- The best-case complexity is the least value of all the $t(i_n)$s.

We will now study a more complex example: the worst-case complexity and the average complexity of Quicksort, which is considered one of the best sorting algorithms.

14.1.1 Quicksort

Briefly recall the idea of Quicksort: for sorting a length n list, we choose a pivot p in this list and we pairwise permute the elements of the list in order to put together all the elements $\leq p$ at the beginning of the list, and all the elements $> p$ at the end of the list; the pivot is then put in its proper place (between the elements $\leq p$, and the elements $> p$), and we repeat this operation with the two sublists of elements $\leq p$, and of elements $> p$ (see Figure 14.1). Here again we have a 'divide and conquer' strategy.

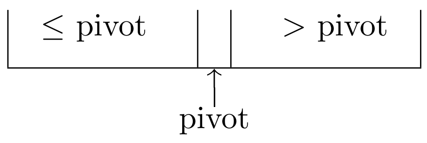

Figure 14.1

Let $T[i \dots j]$ be the list of the elements to be sorted; the algorithm just described can be written:

```
PROGRAM Quicksort
VAR i,j,k:  integer
VAR T: integer list
BEGIN
READ i,j,T
IF i < j THEN
   pivot(T,i,j; k)
   Quicksort(T, i, k-1)
   Quicksort(T, k+1, j)
ENDIF
PRINT T
END
```

`pivot` is a procedure choosing $T(i)$ as pivot and permuting the elements of $T[i \ldots j]$ until $T(i)$ is put in its final place, whose index is k. All the $T(j) \le T(i)$ are before $T(i)$ and all the $T(j) > T(i)$ are after $T(i)$. The `pivot` procedure uses two counters for ranging over the $T(j)$s, one of them increasing, and the other one decreasing until they meet in k, which is the final place of the pivot, then $T(i)$ and $T(k)$ are interchanged and $T(k)$ sits in its final place (Figure 14.2).

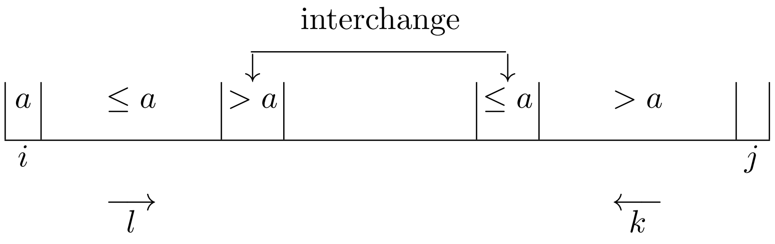

Figure 14.2

```
PROGRAM pivot
VAR i, j, k, l:  integer
VAR T: integer list
BEGIN
READ i,j,T
l:= i+1
k:= j
p:= T(i)
WHILE l $\le$ k DO
   WHILE T(k) > p DO k:= k-1 ENDWHILE
   WHILE T(l) $\le$ p DO l:= l+1 ENDWHILE
   IF l < k THEN
      interchange (T(l),T(k))
      k:= k-1
      l:= l+1
   ENDIF
ENDWHILE
interchange (T(i),T(k))
RETURN (k)
END
```

k stops as soon as $T(k) \le$ pivot, l stops as soon as $T(l) >$ pivot and $T(k)$ and $T(l)$ are interchanged; then, when k and l meet, namely, k becomes greater than or equal to l, we can stop and place the pivot, which will no longer move. The following example for sorting the list $100, 202, 22, 143, 53, 78, 177$, can be studied:

$\underline{\textbf{100}}$	**202**	**22**	**143**	**53**	**78**	**177**
$\underline{\textbf{53}}$	**78**	**22**	100	**143**	**202**	**177**
22	53	**78**	100	$\underline{\textbf{143}}$	**202**	**177**
22	53	78	100	143	$\underline{\textbf{202}}$	**177**
22	53	78	100	143	**177**	202

The pivots have been underlined and the sublists of the elements $\leq k$ and of the elements $> k$ which are left for sorting, are in bold-face.

We will study the *complexity* of `Quicksort` in terms of the *number of comparisons performed* on the list which we want to sort, T: if there are n elements in this list, the `pivot` procedure will compare the pivot to all the other elements; however, depending upon how the two counters k and l meet, we will have $n-1$, n or $n+1$ comparisons to perform.

EXERCISE 14.1 Study the various possible cases for the termination of the `pivot` procedure in order to check the above statement. ♢

Let $C_n(T[1,\ldots,n])$ be the complexity of `Quicksort` for a list $T[1,\ldots,n]$ of length n; if the pivot is put in the kth place we will have

$$C_n(T[1,\ldots,n]) = C_{k-1}(T[1,\ldots,k-1]) + C_{n-k}(T[k,\ldots,n]) \\ + \pi_n(T[1,\ldots,n])\,,$$

where $\pi_n(T[1,\ldots,n])$ denotes the complexity of placing the pivot for the list $(T[1,\ldots,n])$.

EXERCISE 14.2 Check that

1. $\pi_2(T[1,2]) = 2$ for $T[1,2] = (11,22)$ or $T[1,2] = (22,11)$.
2. $\pi_5(T[1,\ldots,5]) = 6$ for $T[1,\ldots,5] = (30,51,23,42,14)$. ♢

14.1.2 Worst-case complexity of Quicksort in number of comparisons

We can see that the worst-case complexity will occur when the initial list is already sorted and each call to pivot simply verifies that the first element is the least one. Let p_n be the worst-case complexity for sorting a length n list; we thus have

$$p_n = n + 1 + p_{n-1} \tag{14.1}$$

with $p_2 = 3$.

EXERCISE 14.3 Solve this recurrence relation, and evaluate p_n. ♢

14.1.3 Average complexity of Quicksort in number of comparisons

Let piv_n be the average number of comparisons performed by the procedure `pivot` on a size n list, let $c_n(k)$ be the average number of comparisons performed by the procedure `Quicksort` on a size n list assuming that the result of the `pivot` procedure was the pivot in the kth position and let c_n be the average number of comparisons performed by `Quicksort` on a size n list; we then have

$$n - 1 \leq piv_n \leq n + 1 ,$$

$$n - 1 + c_{k-1} + c_{n-k} \leq c_n(k) \leq n + 1 + c_{k-1} + c_{n-k} .$$

Assume a random repartition of the numbers of the list, and assume that the positions $1, \ldots, n$ are equally probable for the pivot with probability $1/n$; thus $c_n = 1/n\big(c_n(1) + \cdots + c_n(n)\big)$ and, finally, we obtain the following squeeze for c_n:

$$\forall n \geq 2 , \quad n-1+1/n\left(\sum_{k=1}^{n}(c_{k-1}+c_{n-k})\right) \leq c_n \leq n+1+1/n\left(\sum_{k=1}^{n}(c_{k-1}+c_{n-k})\right) .$$

Hence, noting that

$$\sum_{k=1}^{n} c_{k-1} = \sum_{k=1}^{n} c_{n-k} = \sum_{k=1}^{n-1} c_k \quad (\text{ since } c_0 = 0) ,$$

$$\forall n \geq 2 , \quad n - 1 + 2/n \sum_{k=1}^{n-1} c_k \leq c_n \leq n + 1 + 2/n \sum_{k=1}^{n-1} c_k ,$$

or also, letting

$$a_n = n - 1 + 2/n \sum_{k=1}^{n-1} a_k \quad \text{and} \quad b_n = n + 1 + 2/n \sum_{k=1}^{n-1} b_k , \tag{14.2}$$

$$\forall n \geq 2 , \quad a_n \leq c_n \leq b_n .$$

Note, moreover, that $c_0 = c_1 = 0$, and thus we also have $a_0 = b_0 = a_1 = b_1 = 0$.

Consider the recurrence relation defining b_n. It will be evaluated by forming linear combinations of suitably chosen instances of the recurrence relation. Note, first, that

$$n(b_n - (n + 1)) = 2 \sum_{k=1}^{n-1} b_k,$$

hence for $n-1$: $(n-1)(b_{n-1}-n) = 2\sum_{k=1}^{n-2} b_k$ and, by subtraction,

$$nb_n - (n-1)b_{n-1} - 2n = 2b_{n-1}\,, \qquad \text{for } n \geq 3,$$

namely, $nb_n = (n+1)b_{n-1} + 2n$, and dividing by $n(n+1)$,

$$\frac{b_n}{n+1} = \frac{b_{n-1}}{n} + \frac{2}{n+1}\,, \qquad n \geq 3,$$

with $b_0 = b_1 = 0$ and $b_2 = 3$ (computed by the initial recurrence relation). Letting $v_n = \dfrac{b_n}{n+1}$, we obtain $v_n = v_{n-1} + \dfrac{2}{n+1}$, which immediately yields, by the summation factors method,

$$v_n = 1 + 2\sum_{k=4}^{n+1} \frac{1}{k}\,,$$

i.e. letting $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}$,

$$\begin{aligned} b_n &= n+1+2(n+1)\left(H_{n+1} - 1 - \frac{1}{2} - \frac{1}{3}\right) = 2(n+1)\left(H_{n+1} - \frac{4}{3}\right) \\ &= 2(n+1)\left(H_n + \frac{1}{n+1} - \frac{4}{3}\right) = 2(n+1)H_n - \frac{8n+2}{3}\,. \end{aligned}$$

Exercise 14.4 Prove that, similarly, $a_n = 2(n+1)H_n - 4n$. ◇

Since $H_n \sim \log n$ (see Example 7.26, Definition 9.13 and Section 9.2.4), we deduce that $a_n \sim b_n \sim 2n \log n$, and thus $c_n \sim 2n \log n$.

The average complexity of Quicksort for sorting a length n list, assuming that all the possible permutations of the list are equally probable, is thus of the order of magnitude $2n \log n$. This gives us an example where c_n was in fact given by two recurrence inequations, and not by a single equation. When a recurrence c_n defining a cost is defined not by an equality but by a squeeze, we will proceed as above, and try to evaluate the order of magnitude of each part of the squeeze. If we obtain the same order of magnitude $f(n)$ for both parts, then we will have proved that c_n is also of the same order of magnitude $f(n)$.

The average complexity of Quicksort can also be computed by using generating series. Consider the recurrence relation defining a_n:

$$a_n = n - 1 + \frac{2}{n}\sum_{k=0}^{n-1} a_k \quad \text{for} \quad n \geq 2, \qquad \text{and} \qquad a_0 = a_1 = 0.$$

Letting $a(x) = \sum_{n\geq 0} a_n x^n$, multiplying the recurrence equation by x^n and summing for $n \geq 2$:

$$\sum_{n\geq 2} a_n x^n = \sum_{n\geq 2}(n-1)x^n + 2\sum_{n\geq 2}\left(\sum_{k=0}^{n-1} a_k\right)\frac{x^n}{n}.$$

Note that

$$\sum_{n\geq 2}(n-1)x^n = x^2\sum_{n\geq 2}(n-1)x^{n-2} = \frac{x^2}{(1-x)^2}$$

and

$$\begin{aligned}\sum_{n\geq 2}\left(\sum_{k=1}^{n-1} a_k\right)\frac{x^n}{n} &= \sum_{n\geq 1}\left(\sum_{k=0}^{n} a_k\right)\frac{x^{n+1}}{n+1} \qquad \text{(since } a_0 = 0\text{)}\\ &= \int_0^x \sum_{n\geq 0}\left(\sum_{k=0}^{n} a_k\right) s^n ds\\ &= \int_0^x a(s)\times\frac{1}{1-s}ds\ .\end{aligned}$$

Hence,

$$a(x) = \frac{x^2}{(1-x)^2} + 2\int_0^x \frac{a(s)}{1-s}ds\,,$$

and, taking the derivative,

$$a'(x) = \frac{2x}{(1-x)^2} + \frac{2x^2}{(1-x)^3} + \frac{2a(x)}{1-x}\,. \tag{14.3}$$

Equation (14.3) is a differential equation which can be solved by the usual methods:

- First solve the associated homogeneous equation

$$\frac{a'(x)}{a(x)} = \frac{2}{1-x}\ ,$$

hence

$$a(x) = \frac{\lambda}{(1-x)^2}\ .$$

- Then reporting the result in (14.3) and applying the method of variation of parameters, we have

$$\frac{\lambda'(x)}{(1-x)^2} + \frac{2\lambda(x)}{(1-x)^3} = \frac{2x}{(1-x)^2} + \frac{2x^2}{(1-x)^3} + \frac{2\lambda(x)}{(1-x)^3}\ ,$$

and, after simplifications,

$$\begin{aligned}\lambda'(x) &= 2x + \frac{2x^2}{1-x} = 2x + \frac{2(1-x)^2 - 4(1-x) + 2}{1-x} \\ &= -2 + \frac{2}{1-x}\,.\end{aligned}$$

(We computed the partial fraction expansion of the rational function $\frac{2x^2}{1-x}$.) Integrating we obtain

$$\lambda(x) = -2x + 2\log\frac{1}{1-x} + c$$

and, since $\lambda(0) = a_0 = 0$, we have $c = 0$. Hence, finally,

$$a(x) = \frac{2}{(1-x)^2}(\log\frac{1}{1-x} - x)\,.$$

But

$$\begin{aligned}\frac{1}{(1-x)^2}\log\frac{1}{1-x} &= \sum_{n\geq 0}(n+1)H_{n+1}x^n - \frac{1}{(1-x)^2} \qquad \text{(see Example 8.10)} \\ &= \sum_{n\geq 0}(n+1)H_{n+1}x^n - \sum_{n\geq 0}(n+1)x^n\end{aligned}$$

and

$$\frac{x}{(1-x)^2} = \sum_{n\geq 0}(n+1)x^{n+1},$$

hence,

$$a_n = 2(n+1)H_{n+1} - 2(n+1) - 2n = 2(n+1)H_n - 4n - 2.$$

We thus have

$$a_n \sim 2nH_n = \theta(n\log n)$$

since the order of magnitude of H_n is $\log n$.

EXERCISE 14.5 Verify by the same method that the upper bound b_n of the inequalities giving the average complexity of Quicksort, which is defined by the recurrence equation

$$b_n = n + 1 + \frac{2}{n}\sum_{k=1}^{n-1} b_k, \qquad n \geq 2, \qquad b_0 = b_1 = 0,$$

is equal to

$$b_n = 2(n+1)H_n - \frac{8n+2}{3}\,, \qquad n \geq 2.$$ ♢

14.2 Euclid's algorithm

14.2.1 Euclid's algorithm

This algorithm takes as input two integers $u, v \in \mathbb{N}$ and gives as output their *gcd*, namely, the greatest integer d such that:

$$d \mid u \ , \ d \mid v \quad \text{and} \quad \forall d' \quad (d' \mid u \text{ and } d' \mid v \Longrightarrow d' \mid d),$$

where $d \mid u$ is an abbreviation for 'd divides u'.

The idea of the algorithm is as follows: we assume $0 \leq v \leq u$. If $v = 0$ then $gcd(u, v) = d = u$. Otherwise, we use the property given below.

Lemma 14.1 *If $u = vq + r$ with $0 \leq r < v$, then $gcd(u, v) = gcd(v, r)$.*

Proof. By induction. See Exercise 14.6. □

We thus inductively substitute the computation of $gcd(v, r)$ for the computation of $gcd(u, v)$, where r is the remainder of the division of u by v. Formally, the algorithm is:

```
PROGRAM Euclid
VAR u,v,r:  integer
BEGIN
READ u,v
IF u < 0 THEN u:= -u ENDIF
IF v < 0 THEN v:= -v ENDIF
IF u = 0 and v = 0 THEN
   WRITE gcd undefined
OTHERWISE
   WHILE v ≠ 0 DO
      r:= remainder of the division of u by v
      u:= v
      v:= r
   ENDWHILE
   WRITE u
ENDIF
END
```

Exercise 14.6 Show that Euclid's algorithm terminates and computes the greatest common divisor of its arguments. Hint: an induction on the value of the variable v in the loop will do the job. (See Chapter 3 and Section 14.3.) ♢

The space complexity of this algorithm is quite simple: only three variables. The time complexity can be measured according to various criteria: number of operations performed, possibly multiplied by a factor measuring the cost of the operation, etc. We will choose as the complexity measure the number n of times we go through the '`WHILE`' loop; this implicitly assumes that the costs of the assignments and the divisions performed in a loop are integrated in the cost of executing that loop, and we count only the 'number of divisions performed' before obtaining the *gcd*.

The best-case complexity (case when $(v = 0)$) is $n = 0$. We will now study the worst-case complexity. This complexity was first studied in the eighteenth century using Fibonacci numbers.

14.2.2 Complexity of Euclid's algorithm in the worst-case

In the preceding section we studied the time complexity, and we characterized it by the number of divisions to be performed in order to obtain the *gcd*. It thus suffices to evaluate this number of divisions.

Proposition 14.2 *Let $0 < v \leq u$. Then*
(i) $\frac{v}{u}$ *can be represented by a continued fraction, namely, an expression of the form*

$$\frac{v}{u} = \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\cdots + \cfrac{1}{a_{n-1} + \cfrac{1}{a_n}}}}}$$

where all the a_is are positive integers and $a_n \neq 1$.
(ii) *The number of divisions performed by Euclid's algorithm for obtaining the gcd of u and v is equal to the number of coefficients a_i of the continued fraction representing $\frac{v}{u}$.*

Before sketching the proof, note that in order to obtain the worst-case complexity it is enough to minimize u and v with respect to n, thus obtaining the least possible u and v for each n, and to this end we will see that it will suffice to minimize the a_is. We only sketch the proofs and invite the reader to fill in the details as an exercise.

Lemma 14.3 *The continued fraction*

$$e = \frac{v}{u} = \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\cdots + \cfrac{1}{a_{n-1} + \cfrac{1}{a_n}}}}}$$

can be rewritten as $\dfrac{Q_{n-1}(a_2,\ldots,a_n)}{Q_n(a_1,\ldots,a_n)}$ *where the* Q_i*s are polynomials in the* a_i*s, with positive integer coefficients, and where the* a_i*s are the successive quotients obtained by Euclid's algorithm applied to the pair* (u,v)*, with* $u = Q_n(a_1,\ldots,a_n)$ *and* $v = Q_{n-1}(a_2,\ldots,a_n)$.

Proof. By induction on n.
Basis: If $n = 2$, $\dfrac{1}{a_1 + \dfrac{1}{a_2}} = \dfrac{a_2}{a_1a_2+1}$ and

$$Q_1(a_2) = a_2\,,\quad Q_2(a_1,a_2) = a_1a_2+1\,;$$

moreover, letting $u = a_1v+1$ and $v = a_2$, we have $gcd(u,v) = 1$.
Inductive step: Assume the result holds for p, and let us prove that it holds for $n = p+1$. Let $e = \dfrac{1}{a_1+e'}$, where

$$e' = \cfrac{1}{a_2 + \cfrac{1}{\cdots + \cfrac{1}{a_{n-1} + \cfrac{1}{a_n}}}}$$

then, by the induction hypothesis,

$$e' = \frac{v'}{u'} = \frac{Q_{p-1}(a_3,\ldots,a_n)}{Q_p(a_2,\ldots,a_n)}\,,$$

where Q_{p-1} and Q_p are polynomials in the a_is, with positive integral coefficients, $v' < u'$, and $a_2,\ldots,a_n$ are the successive quotients obtained in the computation of $gcd(u',v')$. Then,

$$e = \frac{1}{a_1 + \dfrac{v'}{u'}} = \frac{u'}{a_1u'+v'}\,.$$

The integers

$$\begin{aligned} u &= a_1u'+v' = Q_n(a_1,\ldots,a_n) = a_1Q_p(a_2,\ldots,a_n) + Q_{p-1}(a_3,\ldots,a_n)\,,\\ v &= u' = Q_{n-1}(a_2,\ldots,a_n) = Q_p(a_2,\ldots,a_n) \end{aligned}$$

have the required form, and $u' < a_1u'+v'$. The quotient of $a_1u'+v'$ by u' is a_1 , and $gcd(a_1u'+v',u') = gcd(u',v')$. Hence the result. □

Remark 14.4
1. In the preceding proof we could have started the induction with $n = 1$ instead of $n = 2$.
2. The preceding proof, moreover, shows that the recurrence defining the polynomials Q_n is given by

$$Q_n(a_1, \ldots, a_n) = a_1 Q_{n-1}(a_2, \ldots, a_n) + Q_{n-2}(a_3, \ldots, a_n).$$

The proposition immediately follows from Lemma 14.3.

Because all the coefficients of the polynomials Q_n are positive integers, the worst-case complexity (namely, the greatest possible n for the least possible u and v) will be obtained when all the a_is with $i < n$ are equal to 1 and $a_n = 2$.

So, for a given n, the least possible u and v whose *gcd* requires n divisions are determined by the recurrence

$$u_n = Q_n(1, \ldots, 1, 2) = 1 \times Q_{n-1}(1, \ldots, 1, 2) + Q_{n-2}(1, \ldots, 1, 2)\,,$$

and

$$v_n = Q_{n-1}(1, \ldots, 1, 2)\,.$$

u_n and v_n are obtained by solving: $u_n = u_{n-1} + u_{n-2}$ and $v_n = u_{n-1}$. (The verification is left to the reader.)

We thus have to solve the recurrence $u_n = u_{n-1} + u_{n-2}$, with $u_0 = 1$, $u_1 = 2$.

Shortly, we sketch various methods for so doing, and refer the reader to Chapter 7 for more details about recurrence relations.

14.2.3 Fibonacci numbers

The u_ns such that $u_n = u_{n-1} + u_{n-2}$ are called the Fibonacci numbers, and, following tradition, we will denote them by F_n. We thus want to determine F_n given that

$$F_n = F_{n-1} + F_{n-2} \qquad \text{and} \qquad F_0 = 0 \quad F_1 = 1\,. \tag{14.4}$$

(We will have: $u_n = F_{n+2}\,, \quad u_{n-1} = F_{n+1}$.)

First method: the characteristic polynomial

We look for a solution of the form $F_n = r^n$. We deduce $r^n = r^{n-1} + r^{n-2}$, and hence $r^2 = r + 1$, which has two roots $r = \dfrac{1 \pm \sqrt{5}}{2}$. Then the general solution of (14.4) is of the form: $\lambda r_1^n + \mu r_2^n$, where $r_1 = \dfrac{1 + \sqrt{5}}{2}$ and $r_2 = \dfrac{1 - \sqrt{5}}{2}$; moreover, $\lambda + \mu = F_0 = 0$ and $\lambda r_1 + \mu r_2 = 1$, and hence $\lambda = -\mu = 1/\sqrt{5}$, and $F_n = 1/\sqrt{5} \times (r_1^n - r_2^n)$.

For evaluating the complexity, we are interested in large values of n. When n tends to infinity, $\lim_{n\to\infty} r_2^n = 0$ and $\lim_{n\to\infty} r_1^n = \infty$, and thus F_n is equivalent to $r_1^n/\sqrt{5}$ when $n \to \infty$ (see Chapter 9 on asymptotic behaviours). Consequently, if u_n is the Fibonacci number F_{n+2}, and v_n is the Fibonacci number F_{n+1}, the number of divisions performed by Euclid's algorithm will be equal to n, and by the asymptotic behaviour of F_n we will have:

$$u_n = F_{n+2} \sim \frac{r_1^{n+2}}{\sqrt{5}} ,$$

and thus $n + 2 \sim \log_{r_1}(\sqrt{5}u_n)$ and finally $n \sim \log_{r_1}(\sqrt{5}u_n)$.

The worst-case complexity is thus at most $\log_{r_1}(\sqrt{5}u)$. This example shows us that, in order to evaluate the complexity of a very simple algorithm such as Euclid's algorithm, we already need tools as elaborate as inductive proofs, methods for solving recurrences and methods for studying asymptotic behaviours, to which chapters of this book have been devoted. Before concluding the present study, we briefly sketch another method for solving recurrences.

Second method: matrix method

$F_{n+1} = F_n + F_{n-1}$ can be written in the form of a linear system:

$$\begin{aligned} F_{n+1} &= F_n + F_{n-1} \\ F_n &= F_n \end{aligned} \qquad \Longleftrightarrow \qquad \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} .$$

Let M be the matrix

$$M = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} .$$

We have

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = M \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix} = \cdots = M^n \begin{pmatrix} F_1 \\ F_0 \end{pmatrix} = M^n \begin{pmatrix} 1 \\ 0 \end{pmatrix} .$$

It thus suffices to compute the nth power of M. To this end, we compute the eigenvalues (we find r_1 and r_2) and diagonalize M. Hence,

$$N = \begin{pmatrix} r_1 & 0 \\ 0 & r_2 \end{pmatrix} = A^{-1}MA \quad \text{and} \quad M^n = AN^nA^{-1} = A \begin{pmatrix} r_1^n & 0 \\ 0 & r_2^n \end{pmatrix} A^{-1}.$$

(A is the transition matrix from M to N.)

Third method: generating series

See Example 8.2.

Exercise 14.7 Compute the continued fraction corresponding to v/u with $v = 8$ and $u = 29$. What is the number n of divisions needed for obtaining $gcd(8, 29)$? ♢

14.3 Proofs of program properties and termination

To conclude the present chapter, we will show how to use inductive proofs to verify properties of programs. Our goal is *not* to build a program prover but simply to show, with some examples, how to use proofs by induction in that area. Essentially, proofs by induction are useful for iterative loops as well as recursive processes (recursive procedures, functions, clauses, etc.).

Consider the following program:

```
PROGRAM square
VAR a,b,c:  integer
BEGIN
READ a
IF a < 0 THEN
   a:= -a
ENDIF
b:= a
c:= 0
WHILE b ≠ 0 DO
   c:= c + a
   b:= b - 1
ENDWHILE
WRITE c
END
```

We will show that this program computes and writes a^2. The first point to be verified is that we will eventually exit the loop. We proceed as follows:

- Before entering the loop, the value of `b` is a positive integer.
- The value of `b` is decreased at each execution of the loop. This value will thus eventually become zero.
- When the value of `b` is zero, we exit the loop.

Once we have proved that the loop terminates, we must check that the result is correct. For so doing, we show that the property $I(a,b,c)$: '$a^2 = c + b \times a$' holds at each execution of the loop. Let b_n and c_n be the values of the variables `b` and `c` after the nth execution of the loop. Let $P(n)$ be the property '$a^2 = c_n + b_n \times a$'. We must now verify that $\forall n \in \mathbb{N}, P(n)$. The proof is by induction:

- $b_0 = a$ and $c_0 = 0$, and thus $P(0)$ is true.
- Let $n \in \mathbb{N}$, and assume $P(n)$. We have $b_{n+1} = b_n - 1$ and $c_{n+1} = c_n + a$. We deduce $c_{n+1} + b_{n+1} \times a = c_n + a + (b_n - 1) \times a = c_n + b_n \times a = a^2$, which proves that $P(n+1)$ is true.

As a result, on exit of the loop both $b = 0$ and $I(a,b,c)$ hold. Thus $a^2 = c + 0 \times a$ $= c$, which shows that the result of the program is a^2.

The basis for proving the correctness of a program is formalized by Hoare's *assertion method.* Let p, q be two first order formulas, and let `S` be a program; `S` is said to be *partially correct with respect to initial assertion p and final assertion* q if and only if whenever p is true for the input values of `S`, then q is true for the output values of `S`. This is denoted by: $p\{\mathtt{S}\}q$.

`S` is said to be *totally correct with respect to initial assertion p and final assertion q* if and only if whenever p is true for the input values of `S`, then `S` terminates *and* `S` is partially correct with respect to p and q.

We sketch a deductive system for proving partial correctness of iterative program.

The *axioms* are:

for any assignment `x:=t`, and any formula p, $p(t)\{\mathtt{x} := \mathtt{t}\}p(x)$,

i.e. if $p(t)$ holds and we assign t to x then $p(x)$ holds.

The *inference rules* are:

composition
$$\frac{p\{\mathtt{S}_1\}q \quad q\{\mathtt{S}_2\}r}{p\{\mathtt{S}_1;\mathtt{S}_2\}r}$$

conditional
$$\frac{(p \wedge b)\ \{\mathtt{S}\}\ q \qquad \big((p \wedge \neg b) \Longrightarrow q\big)}{p\ \{\mathtt{IF}\ b\ \mathtt{THEN}\ \ \mathtt{S}\ \mathtt{ENDIF}\}\ q}$$

$$\frac{(p \wedge b)\ \{\mathtt{S}_1\}\ q \qquad (p \wedge \neg b)\ \{\mathtt{S}_2\}\ q}{p\ \{\mathtt{IF}\ b\ \mathtt{THEN}\ \ \mathtt{S}_1\ \mathtt{OTHERWISE}\ \ \mathtt{S}_2\ \mathtt{ENDIF}\}\ q}$$

loop invariant
$$\frac{(p \wedge b)\ \{\mathtt{S}\}\ p}{p\ \{\mathtt{WHILE}\ b\ \mathtt{DO}\ \ \mathtt{S}\ \ \mathtt{ENDWHILE}\}\ (p \wedge \neg b)}$$

consequence
$$\frac{(p \Longrightarrow q) \qquad q\{\mathtt{S}\}r \qquad (r \Longrightarrow s)}{p\{\mathtt{S}\}s}$$

The rules are to be read as: if the formula(s) above the horizontal line hold, then the formula below the horizontal line also holds. For instance, the composition rule states that: assume that if p is true and $\mathtt{S}_1$ executes and terminates, then q is true and, moreover, if q is true and $\mathtt{S}_2$ executes and terminates, then r is true; with those assumptions, if p is true and $\mathtt{S} = \mathtt{S}_1;\mathtt{S}_2$ executes and terminates, where ';' denotes sequential composition, then r is true.

Example 14.5

1. Consider the program segment `S` defined by

```
c':= c + a
b':= b - 1
```

and let p be $(a^2 = ab + c)$; then, by the consequence rule and the axioms for assignment, we have that

$$p \; \{\texttt{c':= c + a}\} \; (a^2 = ab + c' - a).$$

Applying the consequence rule

$$p \; \{\texttt{c':= c + a}\} \; (a^2 = a(b-1) + c').$$

Let q be $(a^2 = a(b-1) + c')$; then by the axioms for assignment

$$q \; \{\texttt{b':= b-1}\} \; (a^2 = ab' + c').$$

Applying now the composition rule, we obtain

$$(a^2 = ab + c) \; \{\texttt{S}\} \; (a^2 = ab' + c').$$

2. Consider the program segment `S'` defined by

```
IF $a < 0$ THEN $a := -a$ ENDIF
```

Applying the conditional rule and the assignment axioms, we can deduce that: $p \; \{\texttt{S'}\} \; q$ holds, where p is 'a is an integer' and q is $(a \geq 0)$.

3. First, we give the intuition behind the loop invariant rule. Assertion p is said to be a loop invariant if p holds before entering the loop and if p remains true at each execution of the loop. Because the loop is executed until condition b becomes false, then, when exiting the loop (if this occurs), $\neg b \wedge p$ must hold.

Consider the program segment `S''` defined by

```
WHILE B $\neq$ 0 DO
c:= c + a
b:= b - 1
ENDWHILE
```

and let p be $(a^2 = ab + c)$, then, by the loop invariant rule and part 1 of the present example, we have that: $p \; \{\texttt{S''}\} \; \big(p \wedge (b = 0)\big)$.

Combining parts 1, 2 and 3 of Example 14.5, with the composition rule, we conclude that program `square` is partially correct with respect to the initial assertion *integer*(a) and the final assertion $(a^2 = ab + c) \wedge (b = 0)$. This shows that program `square` indeed computes a^2. In order to picture the whole proof at a glance, we annotate program `square` with the final intermediate assertions; each assertion is written to the right of the instruction after which it is true; we thus have

```
PROGRAM square
VAR a,b,c:  integer
BEGIN
READ a                              $integer(a)$
IF a < 0 THEN
   a:= -a
ENDIF                               $a \geq 0$
b:= a
c:= 0                               $a^2 = ab + c$
WHILE b $\neq$ 0 DO
   c:= c + a
   b:= b - 1
ENDWHILE                            $(a^2 = ab + c) \wedge (b = 0)$
WRITE c                             $(a^2 = c)$
END
```

We can show the axioms and rules of our deductive system for proving partial correctness of iterative program are sound, so that any statement $p\{\mathtt{S}\}q$ obtained by this deductive system will be valid. This system is, however, not complete, and it can be shown that there does not exist a complete deductive system for proving all valid partial correctness assertions, even for a toy programming language. (Hard.)

EXERCISE 14.8 Show that the program power terminates and writes the result a^k (with the convention $0^0 = 1$). ♢

```
PROGRAM power
VAR a,k,r:  integer
BEGIN
READ a,k
n:= k
IF n < 0 and a = 0 THEN
   WRITE undefined result
OTHERWISE
   r:= 1
   WHILE n < 0 DO
      r:= r / a
      n:= n+1
   ENDWHILE
   WHILE n > 0 DO
      r:= r * a
      n:= n-1
   ENDWHILE
   WRITE r
ENDIF
END
```

The case of recursive programs is slightly more complex because several recursive calls can occur in the same program and several cases can also occur when the result is obtained without any recursive call. Recursive programs still can, however, be studied in a similar way. Consider the following procedure listing the inorder traversal of a binary tree (see Example 3.24):

```
PROCEDURE inorder(x:  BT)
BEGIN
IF x $\neq \emptyset$ THEN
   inorder(LeftChild(x))
   WRITE root(x)
   inorder(RightChild(x))
ENDIF
END
```

In order to prove termination, we can consider the mapping $h\colon BT \longrightarrow \mathbb{N}$ giving the height of a binary tree. The value of $h(x)$ strictly decreases at each recursive call, namely, $h(\mathit{LeftChild}(x)) < h(x)$ and $h(\mathit{RightChild}(x)) < h(x)$. Since there can be no strictly decreasing infinite sequence in $\mathbb{N}$, we deduce that there are a finite number of recursive calls. Consequently, the procedure `inorder` always terminates.

Note that the choice of the mapping h is arbitrary. We could have chosen the mapping $n\colon BT \longrightarrow \mathbb{N}$ giving the number of nodes of a tree or even the mapping $\mathit{id}\colon BT \longrightarrow BT$. Indeed, the relation 'to be a subtree of' is a well-founded ordering on the set BT of binary trees. It is formally defined as the reflexive and transitive closure of the relation

$$\forall a \in A, \forall l, r \in BT, \quad l < (a, l, r) \text{ and } r < (a, l, r)\ .$$

Moreover, it verifies $\mathit{LeftChild}(x) < x$ and $\mathit{RightChild}(x) < x$, which proves that the procedure `inorder` terminates.

Usually, to prove the termination of a recursive program we associate it with an expression that is given its values in a well-founded ordering. Most often, this expression depends only on the arguments of the recursive program. Let V be the value of the expression applied to the arguments of the program. ($V = h(x)$ in the above example.) We must then verify that, for each recursive call, the value of this expression for the parameters of the call ($h(\mathit{LeftChild}(x))$ and $h(\mathit{RightChild}(x))$ in the above example) is strictly less than V. As a result we can conclude that the program terminates.

Similarly, in order to prove a property of a recursive program, we associate with it a property P connecting the arguments of the program with its result. We then directly show that P holds in all the cases when the program terminates without recursive calls. Assuming that the property holds for all the recursive calls of the program, we prove that it still holds when the program terminates. Thus we have another inductive proof that P holds on exit of the program regardless of the values of its initial arguments. Consider, for instance, the following function:

```
FUNCTION power(a,n: integer): y: integer
BEGIN
IF n = 0 THEN
   y:=1
OTHERWISE
   IF n = 1 THEN
      y:=a
   OTHERWISE
      IF n is even THEN
         y:=power(a * a, n / 2)
      OTHERWISE
         y:=a * power(a * a, (n - 1) / 2)
      ENDIF
   ENDIF
ENDIF
RETURN(y)
END
```

Let us show that, $\forall n \geq 0$, this function computes a^n. We thus consider the property P: '$\texttt{power}(a, n) = a^n$' and we prove that it holds when the function returns its value y:

- There are two cases where the function directly returns a value: if $n = 0$, the result is $1 = a^0$, and if $n = 1$, the result is $a = a^1$. The property P is thus verified in both cases.
- If n is even, the result is $\texttt{power}(a \times a, n/2)$. By the induction hypothesis we have $\texttt{power}(a \times a, n/2) = (a \times a)^{n/2} = a^n$. The property P still holds.
- Similarly, if n is odd, the result is $a \times \texttt{power}(a \times a, (n-1)/2)$. By the induction hypothesis we have

$$\texttt{power}(a \times a, (n-1)/2) = (a \times a)^{(n-1)/2} = a^{n-1}.$$

We deduce that the result is $a \times a^{n-1} = a^n$ and property P still holds.

This can be formalized as an Hoare inference rule for recursion. Let

$$f(x : y) \quad \texttt{body}$$

be a recursive procedure with argument x, result y and defined by body; let p (resp. q) be an initial (resp. final) assertion; we extend Hoare's inference rules for iterative programs by an inference rule for recursion:

recursion $$\frac{\big(p\{f(x:y)\}q\big) \Longrightarrow \big(p\{\texttt{body}\}q\big)}{p\ \{f(x:y)\ \texttt{body}\}\ q},$$

which means: if, when we assume the partial correctness of all internal calls with respect to p and q we can prove the partial correctness of `body` with respect to p and q, then the recursive procedure $f(x:y)$ `body` is indeed partially correct with respect to p and q.

In order to apply this rule to the preceding program `power`, note that p is: *integer*(a, n), and q is: $y = a^n$. `power` annotated with the final assertions is given below:

```
FUNCTION power(a,n:  integer):  y:  integer
BEGIN
IF n = 0 THEN
   y:= 1                                 $(n = 0) \wedge (y = 1)$
OTHERWISE
   IF n = 1 THEN
      y:= a                              $(n = 1) \wedge (y = a)$
   OTHERWISE
      IF n is even THEN
         y:= power(a*a, n/2)            $\big(even(n)\big) \wedge \big(y = (a \times a)^{n/2}\big)$
      OTHERWISE
         y:= a*power(a*a, (n-1)/2)      $\big(odd(n)\big) \wedge \big(y = a\times(a\times a)^{\frac{n-1}{2}}\big)$
      ENDIF
   ENDIF
ENDIF                                   $(y = a^n)$
RETURN(y)                               $(y = a^n)$
END
```

EXERCISE 14.9 Show that $\forall n \in \mathbb{N}$, the call `Fact`(n) of the `Fact` function defined below terminates and computes $n!$ ♢

```
FUNCTION Fact(n:  integer):  y:  integer
BEGIN
IF n = 0 THEN y:= 1
OTHERWISE y:= n*Fact(n-1)
ENDIF
RETURN(y)
END
```

Let the Ackermann function be defined by

```
FUNCTION Ackermann(n,m:  integer):  y:  integer
BEGIN
IF n = 0 THEN
   y:= (m + 1)
OTHERWISE
   IF m = 0 THEN
      y:= (Ackermann(n - 1,1))
   OTHERWISE
      y:= (Ackermann(n - 1, Ackermann(n, m - 1))
   ENDIF
ENDIF
RETURN(y)
END
```

EXERCISE 14.10 Show that $\forall n, m \in \mathbb{N}$, the call `Ackermann`$(n, m)$ terminates. Hint: use the lexicographic ordering on $\mathbb{N}^2$ (see Example 2.30). ♢

EXERCISE 14.11 Show that in the PROLOG program given below, the call `Q(x)` terminates with the result *true*, $\forall x \in \mathbb{N}$. ♢

```
Q(x) ⟵ (x = 0)
Q(x) ⟵ (y = x - 1) ∧ Q(y)
```

EXERCISE 14.12
1. Prove that `Quicksort` and `pivot` terminate.
2. Prove that `Quicksort` is partially correct with respect to the final assertion q $i \leq k \leq l \leq j \Longrightarrow \big(T(k) \leq T(l)\big)$. ♢