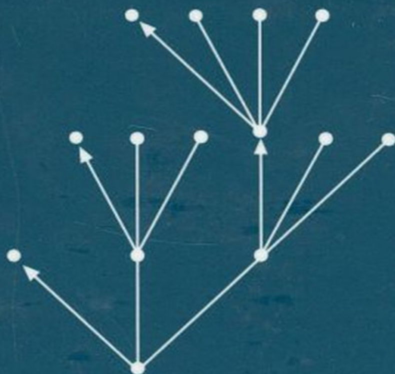
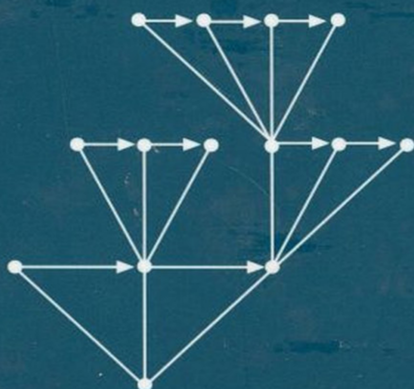
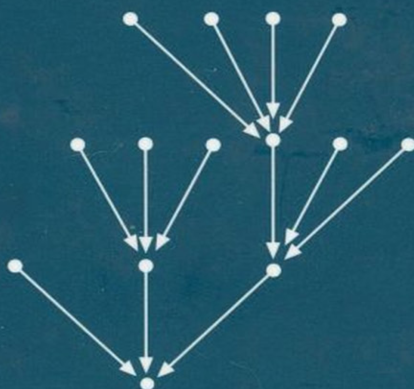


Alexander Shen



Algorithms and Programming
Problems and Solutions



Birkhäuser

Modern Birkhäuser Classics

Many of the original research and survey monographs in pure and applied mathematics published by Birkhäuser in recent decades have been groundbreaking and have come to be regarded as foundational to the subject. Through the MBC Series, a select number of these modern classics, entirely uncorrected, are being re-released in paperback (and as eBooks) to ensure that these treasures remain accessible to new generations of students, scholars, and researchers.

Algorithms and Programming

Problems and Solutions

Alexander Shen

Reprint of the 1997 Edition

Birkhäuser
Boston • Basel • Berlin

Alexander Shen
Institute of Problems
of Information Transmission
Moscow 103051
Russia

ISBN-13: 978-0-8176-4760-5 e-ISBN-13: 978-0-8176-4761-2
DOI: 10.1007/978-0-8176-4761-2

Library of Congress Control Number: 2007940260

©2008 Birkhäuser Boston
All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Birkhäuser Boston, c/o Springer Science+Business Media LLC, 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.
The use in this publication of trade names, trademarks, service marks and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Cover design by Alex Gerasev.

Printed on acid-free paper.

9 8 7 6 5 4 3 2 1

www.birkhauser.com

Alexander Shen

Algorithms and Programming

Problems and Solutions

Birkhäuser
Boston • Basel • Berlin

Alexander Shen
Institute of Problems
of Information Transmission
Moscow 103051
Russia

Library of Congress Cataloging-in-Publication Data

Shen, A. (Alexander), 1958-
Algorithms and programming : problems and solutions / Alexander Shen.
p. cm.
Includes bibliographical references and index.
ISBN 0-8176-3847-4 (alk. paper). -- ISBN 3-7643-3847-4 (alk. paper)
1. Computer algorithms. 2. Electronic digital computers--Programming. I. Title.
QA76.9.A43S47 1996 96-30975
005.1--dc 20 CIP

Printed on acid-free paper
Original published 1995 in Russian

© 1997 Birkhäuser Boston

Copyright is not claimed for works of U.S. Government employees.
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior permission of the copyright owner.

Permission to photocopy for internal or personal use of specific clients is granted by Birkhäuser Boston for libraries and other users registered with the Copyright Clearance Center (CCC), provided that the base fee of $6.00 per copy, plus $0.20 per page is paid directly to CCC, 222 Rosewood Drive, Danvers, MA 01923, U.S.A. Special requests should be addressed directly to Birkhäuser Boston, 675 Massachusetts Avenue, Cambridge, MA 02139, U.S.A.

ISBN 0-8176-3847-4
ISBN 3-7643-3847-4

Typeset in Latex by the Author.
Printed and bound by Quinn-Woodbine, Woodbine, NJ.
Printed in the U.S.A.

9 8 7 6 5 4 3 2 1

Contents

1 Variables, expressions, assignments 1
- 1.1 Problems without arrays 1
- 1.2 Arrays 15
- 1.3 Inductive functions 29

2 Generation of combinatorial objects 33
- 2.1 Sequences 33
- 2.2 Permutations 34
- 2.3 Subsets 35
- 2.4 Partitions 37
- 2.5 Gray codes and similar problems 38
- 2.6 Some remarks 44
- 2.7 Counting 46

3 Tree traversal (backtracking) 49
- 3.1 Queens not attacking each other: position tree traversal 49
- 3.2 Backtracking in other problems 58

4 Sorting 60
- 4.1 Quadratic algorithms 60
- 4.2 Sorting in $n \log n$ operations 61
- 4.3 Applications of sorting 67
- 4.4 Lower bound for the number of comparisons 69
- 4.5 Problems related to sorting 70

5 Finite-state algorithms in text processing 73
- 5.1 Compound symbols, comments, etc. 73
- 5.2 Numbers input 74

6 Data types 79
- 6.1 Stacks 79
- 6.2 Queues 85
- 6.3 Sets 93
- 6.4 Priority queues 96

7 Recursion 98
- 7.1 Examples 98
- 7.2 Trees: recursive processing 101
- 7.3 The generation of combinatorial objects; search 103
- 7.4 Other applications of recursion 107

8 Recursive and nonrecursive programs 114
8.1 Table of values (dynamic programming) 114
8.2 Stack of postponed tasks 118
8.3 Difficult cases 121

9 Graph algorithms 124
9.1 Shortest paths 124
9.2 Connected components, breadth and depth search 127

10 Pattern matching 133
10.1 Simple example 133
10.2 Repetitions in the pattern 135
10.3 Auxiliary lemmas 136
10.4 Knuth–Morris–Pratt algorithm 137
10.5 Boyer–Moore algorithm 140
10.6 Rabin–Karp algorithm 142
10.7 Automata and more complicated patterns 143

11 Set representation. Hashing 151
11.1 Hashing with open addressing 151
11.2 Hashing using lists 153

12 Sets, trees, and balanced trees 158
12.1 Set representation using trees 158
12.2 Balanced trees 165

13 Context-free grammars 176
13.1 General parsing algorithm 176
13.2 Recursive-descent parsing 181
13.3 Parsing algorithm for LL(1)-grammars 191

14 Left-to-right parsing (LR) 198
14.1 LR-processes 198
14.2 LR(0)-grammars 204
14.3 SLR(1)-grammars 207
14.4 LR(1)-grammars, LALR(1)-grammars 208
14.5 General remarks about parsing algorithms 211

Further reading 212

Preface

Somebody once said that one may prove the correctness of an algorithm, but not of a program. One of the main goals of this book is to convince the reader that things are not so bad.

A well-known programmer, C.A.R. Hoare, said that the beauty of a program is not an additional benefit but a criterion that separates success from failure. If, while solving problems in this book, you come to appreciate the beauty of a well-written program with each part in its correct place, the author's goal will have been reached.

We have utilized the problem-solution format. Some sections are collections of problems having a common topic, while others are devoted to one specific algorithm (e.g., Section 14 covers LR(1)-parsing). The sections are more or less independent, but the concluding sections are more difficult. Sections 1–7 cover material usually included in undergradute courses while Sections 13–14 are more appropriate for a graduate compiler course. In each section we have tried to give problems at different levels starting with easy exercises.

Problems are usually provided with solutions, answers or hints. However, we strongly recommend that the reader look at the solutions only after making a good faith attempt to solve the problems independently.

Pascal is used as a programming language to write program examples; however, readers familiar with some other procedural language (C, Modula, Oberon, etc.) will encounter no difficulties.

Most of the problems, of course, are well known. References are rare, but this does not mean that the problem or algorithm is new. However, we hope that in some cases the algorithm or the proof is explained better than what is found in other sources.

This book is addressed both to the ambitious student who wants to test and improve his/her skills and to the instructor looking for problems for his/her class.

I thank all the people I met while teaching programming (first of all, my former students from 57th school and A.G. Kushnirenko, who was my programming teacher) and all readers who sent me corrections for the preliminary versions of this book (especially Yu.V. Matijasevich).

I also thank the American Mathematical Society (former Soviet Union aid fund), International Science Foundation, Open Society Foundation, MIT, University of Bordeaux, Bonn University, the Rosenbaum Foundation, INTAS, the

Russian government and the Institute of Problems of Information Transmission for support during the writing of this book.

I thank Ann Kostant and the other nice people at Birkhäuser Publishing house for their help. Tom Scavo did a great job correcting my English (as well as several other errors) but in no case should he be blamed for the remaining mistakes.

The Russian version of this book is freely distributed in ASCII, TeX and PostScript formats; please contact the author (`shen@landau.ac.ru`, `shen@ium.ips.ras.ru`, `shen@sch57.mcn.msk.su`) for details. I'd be grateful if bug reports will be sent to the same addresses.

To the memory of Anna Pogossiants

Algorithms and Programming

Problems and Solutions

1 Variables, expressions, assignments

1.1 Problems without arrays

1.1.1. Consider two integer variables a and b. Write a program block that exchanges the values of a and b (i.e., the value of a becomes the value of b and vice versa).

Solution. We use an auxiliary integer variable t.

```
t := a;
a := b;
b := t;
```

■

If we try to eliminate this auxiliary variable by writing

```
a := b;
b := a;
```

we get an incorrect program (the value of a is lost after the first assignment).

1.1.2. Solve the preceding problem without an auxiliary variable. (Assume all variables accept arbitrary integer values.)

Solution. (By a0 and b0 we denote the initial values of a and b.)

```
a := a + b; {a = a0 + b0, b = b0}
b := a - b; {a = a0 + b0, b = a0}
a := a - b; {a = b0, b = a0}
```

■

1.1.3. Let a be an integer and n be a nonnegative integer. Compute a^n. In other words, we ask for a program that does not change the values of a and n and assigns the value a^n to another variable (say, b). (The program may use other variables as well.)

Solution. Consider an integer variable k, whose range is 0..n. (We maintain the property: $b = a^k$.)

```
k := 0; b := 1;
{b = a^k}
while k <> n do begin
| k := k + 1;
| b := b * a;
end;
```

Another solution:

```
k := n; b := 1;
{a^n = b * (a^k)}
while k <> 0 do begin
| k := k - 1;
| b := b * a;
end;
```

■

1.1.4. Solve the preceding problem with the additional requirement that the number of execution steps should be of order $\log n$ (i.e., it should not exceed $C \log n$ for some constant C).

Solution. Let us make some changes in the second solution of the preceding problem:

```
k := n;  b := 1;  c := a;
{a^n = b * (c^k)}
while k <> 0 do begin
| if k mod 2 = 0 then begin
| | k := k div 2;
| | c := c*c;
| end else begin
| | k := k - 1;
| | b := b * c;
| end;
end;
```

In both cases (even k and odd k) the value of k decreases; if k is even, it is divided by 2; if k is odd, after `k := k - 1` it becomes even and is divided by 2 during the next iteration. Therefore, after any two iterations k becomes twice smaller (or even less). ■

1.1.5. Two nonnegative integers a and b are given. Compute the product a*b (only +, −, =, <> are allowed).

Solution.

```
k := 0; c := 0;
{invariant relation: c = a * k}
while k <> b do begin
| k := k + 1;
| c := c + a;
end;
{c = a * k and k = b, therefore, c = a * b}
```

■

1.1.6. Two nonnegative integers a and b are given. Compute a + b. Only assignments of the form

$$\langle\text{variable1}\rangle := \langle\text{variable2}\rangle;$$
$$\langle\text{variable}\rangle := \langle\text{number}\rangle;$$
$$\langle\text{variable1}\rangle := \langle\text{variable2}\rangle + 1;$$

are allowed.

[Hint. Use the invariant relation `c = a + k`] ■

1.1.7. A nonnegative integer a and positive integer d are given. Compute the quotient q and the remainder r when a is divided by d. Do not use the operations `div` or `mod`.

Solution. By definition, $a = q*d + r$ and $0 \le r < d$.

```
{a >= 0; d > 0}
r := a; q := 0;
{invariant relation: a = q * d + r, 0 <= r}
while not (r < d) do begin
| {r >= d}
| r := r - d; {r >= 0}
| q := q + 1;
end;
```

■

1.1.8. For a given nonnegative integer n, compute n! (n! is the product $1 \cdot 2 \cdot 3 \cdots n$; we assume that $0! = 1$). ■

1.1.9. The Fibonacci sequence is defined as follows: $a_0 = 0$, $a_1 = 1$, $a_k = a_{k-1} + a_{k-2}$ for $k \ge 2$. For a given n, compute a_n. ■

1.1.10. Repeat the preceding problem with the additional requirement that the number of operations should be proportional to $\log n$. (Use only integer variables.)

[Hint. Any pair of consecutive Fibonacci numbers is the product of the matrix

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

and the preceding pair. Therefore, it is enough to compute the n-th power of this matrix. It can be done in $C \log n$ steps in the same manner as problem 1.1.4 (for integers).] ■

1.1.11. For a nonnegative integer n, compute

$$\frac{1}{0!} + \frac{1}{1!} + \cdots + \frac{1}{n!}.$$

■

1.1.12. Repeat the preceding problem with the additional requirement that the number of steps (i.e., the number of assignments performed during the execution) should be of order n (i.e., not greater than Cn for some constant C).

Solution. The invariant relation: sum $= 1/1! + \cdots + 1/k!$, last $= 1/k!$ (it is important not to compute k! each time from scratch). ■

1.1.13. Two nonnegative integers a and b are not both zero. Compute GCD(a,b), the greatest common divisor of a and b.

Solution (version 1).

```
if a > b then begin
| k := a;
end else begin
| k := b;
end;
{k = max (a,b)}
{invariant relation: no numbers greater than k are
  common divisors}
while not ((a mod k = 0) and (b mod k = 0)) do begin
| k := k - 1;
end;
{k is a common divisor, all larger k are not}
```

(version 2 — Euclid's algorithm). We assume that GCD(0,0)=0. Then GCD(a,b) = GCD(a-b,b) = GCD(a,b-a) with GCD(a,0) = GCD(0,a) = a for all a, b $\geq$ 0. This property allows us to decrease a and b without changing GCD(a,b).

```
 m := a; n := b;
{invariant relation: GCD(a,b) = GCD(m,n); m,n >= 0 }
while not ((m=0) or (n=0)) do begin
| if m >= n then begin
| | m := m - n;
| end else begin
| | n := n - m;
| end;
end;
{m = 0 or n = 0}
if m = 0 then begin
| k := n;
end else begin {n = 0}
| k := m;
end;
```
■

1.1.14. Write down a modified version of Euclid's algorithm using the identities GCD(a,b) = GCD(a mod b, b) for a $\geq$ b and GCD(a,b) = GCD(a, b mod a) for b $\geq$ a. ■

1.1.15. Nonnegative integers a and b are given, at least one of which is not zero. Find d = GCD(a,b) and integers x and y such that d = a*x + b*y.

Solution. Add the auxiliary variables p, q, r, s to Euclid's algorithm and add the requirements m = p*a+q*b and n = r*a+s*b to the invariant relation:

```
m:=a; n:=b; p:=1; q:=0; r:=0; s:=1;
{invariant relation:
    GCD(a,b) = GCD(m,n);
    m,n >= 0
    m = p*a + q*b;
    n = r*a + s*b.}
while not ((m=0) or (n=0)) do begin
| if m >= n then begin
| | m := m - n;
| | p := p - r;
| | q := q - s;
| end else begin
| | n := n - m;
| | r := r - p;
| | s := s - q;
| end;
end;
if m = 0 then begin
| k := n; x := r; y := s;
end else begin
| k := m; x := p; y := q;
end;
```

■

1.1.16. Solve the preceding problem using the mod operator. ■

1.1.17. (E. Dijkstra) Let us add three variables u, v, z to Euclid's algorithm:

```
 m := a; n := b; u := b; v := a;
{invariant relation: GCD (a,b) = GCD (m,n); m,n >= 0 }
while not ((m=0) or (n=0)) do begin
| if m >= n then begin
| | m := m - n; v := v + u;
| end else begin
| | n := n - m; u := u + v;
| end;
end;
```

```
if m = 0 then begin
| z:= v;
end else begin {n=0}
| z:= u;
end;
```

Prove that after execution the value of z is twice as large as the least common multiple of a and b, that is, $z = 2 \cdot \mathrm{LCM}(a,b)$.

Solution. Look at the value of $m \cdot u + n \cdot v$, which remains unchanged during program execution. Initially it is equal to 2ab; therefore, this expression has the same value at the end. Now apply the identity $\mathrm{GCD}(a, b) \cdot \mathrm{LCM}(a, b) = ab$. ■

1.1.18. Write a version of Euclid's algorithm using the identities

$$\mathrm{GCD}(2a, 2b) = 2 \cdot \mathrm{GCD}(a, b); \quad \mathrm{GCD}(2a, b) = \mathrm{GCD}(a, b) \quad \text{for odd } b$$

The algorithm should avoid division (div and mod operations); only division by 2 and the test "to be even" are allowed. (The number of operations should be of order $\log k$ if both numbers do not exceed k.)

Solution.

```
m:=a; n:=b; d:=1;
{GCD(a,b) = d * GCD(m,n)}
while not ((m=0) or (n=0)) do begin
| if (m mod 2 = 0) and (n mod 2 = 0) then begin
| | d:= d*2; m:= m div 2; n:= n div 2;
| end else if (m mod 2 = 0) and (n mod 2 = 1) then begin
| | m:= m div 2;
| end else if(m mod 2 = 1) and (n mod 2 = 0) then begin
| | n:= n div 2;
| end else if (m mod 2=1) and (n mod 2=1) then begin
| | if m >=n then begin
| | | m:= m-n;
| | end else begin {m < n}
| | | n:= n-m;
| | end;
| end;
end;
{m=0 => answer=d*n; n=0 => answer=d*m}
```

If both numbers m and n do not exceed k, the number of operations does not exceed $C \log k$; indeed, each other operation makes at least one of the numbers m and n twice smaller. ■

1.1.19. Modify the solution of the preceding problem to find x and y such that $ax + by = \mathrm{GCD}(a, b)$.

Solution. (The idea was communicated by D. Zvonkin.) Assume that both a and b are even. In this case we divide both of them by 2; the values of x and y we are looking for remain unchanged. Therefore, without loss of generality, we may assume that at least one of the numbers a and b is odd. (This property will remain true.)

As before, we wish to maintain the numbers p, q, r, s such that

$$m = ap + bq$$
$$n = ar + bs$$

The problem, however, is that if we divide m by 2 (say), then we should at the same time divide p and q by 2. In this case p and q are no longer integers but become finite binary fractions, that is, numbers of the type $r/2^s$. Such a number can be represented by a pair $\langle r, s \rangle$. As a result, we get d as a linear combination of a and b with coefficients being finite binary fractions. In other words, we have

$$2^i d = ax + by$$

for some integers x, y and nonnegative integer i. What should we do if $i > 0$? If both x and y are even, we may divide them by 2 (and decrease i by 1). If not, we apply the transformations:

$$x := x + b$$
$$y := y - a$$

(this transformation leaves $ax + by$ unchanged). Let us see why this works. Recall that one of the numbers a and b is odd (according to our assumption). Let a be odd. If y is even, then x is even as well (otherwise $ax + by$ is odd); this case is considered above. If a and y are odd, then y becomes even after executing the statement $y := y - a$. ■

1.1.20. Write a program that prints the squares of the natural numbers 0, 1, . . . , n.

Solution.

```
k:=0;
writeln (k*k);
{invariant relation: k<=n, all the squares
   up to (k*k) are printed}
while not (k=n) do begin
| k := k + 1;
| writeln (k*k);
end;
```

■

1.1.21. Repeat the preceding problem, but only addition and subtraction are allowed. The number of steps should be of order n.

Solution. We use the variable k_square, and maintain the relation $\mathtt{k_square} = \mathtt{k}^2$:

```
k := 0; k_square := 0;
writeln (k_square);
while not (k = n) do begin
| k := k + 1;
| {k_square = (k-1) * (k-1) = k*k - 2*k + 1}
| k_square := k_square + k + k - 1;
| writeln (k_square);
end;
```

■

Remark. We can avoid subtraction by the following trick:

```
while not (k = n) do begin
| k_square := k_square + k;
| {k_square = k*k + k}
| k := k + 1;
| {k_square = (k-1)*(k-1)+(k-1)=k*k-k}
| k_square := k_square + k;
end;
```

1.1.22. Write a program that prints the factorization of a given integer $\mathtt{n} > 0$. (In other words, it should print prime numbers whose product is equal to n; if $\mathtt{n} = 1$, nothing should be printed.)

Solution (version 1).

```
k := n;
{invariant relation: the product of k and all numbers
 printed is equal to n; only prime numbers are printed}
while not (k = 1) do begin
| t := 2;
| {invariant relation: k has no divisors in (1,t)}
| while k mod t <> 0 do begin
| | t := t + 1;
| end;
| {t is the smallest divisor of k greater than 1;
|  therefore, t is prime}
| writeln (t);
| k:=k div t;
end;
```

(version 2)

```
k := n; t := 2;
{the product of k and all number printed is equal
 to n; only prime numbers are printed;
 k has no divisors in (1,t)}
while not (k = 1) do begin
| if k mod t = 0  then begin
| | {k is a multiple of t and has no divisors
| |   less than t; therefore, t is prime}
| | k := k div t;
| | writeln (t);
| end else begin
| | {k is not a multiple of t}
| | t := t+1;
| end;
end;
```

■

1.1.23. Solve the preceding problem taking into account the following fact: any composite number N has a factor not exceeding $\sqrt{N}$.

Solution. In version 2 of the above solution, replace `t:=t+1` by

```
if t*t > k then begin
| t:=k;
end else begin
| t:=t+1;
end;
```

■

1.1.24. Check whether a given number n > 1 is prime. ■

1.1.25. (This problem requires some algebra) A Gaussian integer $n+mi \in Z[i]$ is given. (a) Check whether it is a prime element in $Z[i]$; (b) print its factorization as a product of prime factors in $Z[i]$. ■

1.1.26. Assume the command `write(i)` is allowed for i = 0, 1, 2,. . .,9. Write a program that prints the decimal representation of a given positive integer n.

Solution.

```
base:=1;
{base is an integer power of 10 not exceeding n}
while 10 * base <= n do begin
| base:= base * 10;
end;
{base is a maximal power of 10 not exceeding n}
```

```
k:=n;
{invariant relation: it remains to print k with the same
 number of digits as in base; base = 100..00}
while base <> 1 do begin
| write(k div base);
| k:= k mod base;
| base:= base div 10;
end;
{base=1; it remains to write one digit k}
write(k);
```

Please note that this program assumes that $n > 0$. ■

(A typical mistake while solving this problem is that the numbers with zeros in the middle are printed incorrectly. The invariant relation mentioned above allows the case k < base; in this case, the decimal representation of k begins with zero.)

1.1.27. Write a program that prints the decimal representation in reverse. (For $n = 173$, the program should print 371.)

Solution.

```
k:= n;
{invariant relation: it remains to print k reversed}
while k <> 0 do begin
| write (k mod 10);
| k:= k div 10;
end;
```

■

1.1.28. A nonnegative integer n is given. Count all the solutions of the inequality $x^2 + y^2 < n$ where x and y are nonnegative integers. The program should not use operations with real numbers (such as $\sqrt{\ }$, etc.).

Solution.

```
a := 0; s := 0;
{invariant relation: s = the number of all pairs
 <x,y> such that x*x + y*y < n and x < a}
while a*a < n do begin
| ...
| {t = the number of nonnegative integers y such that
|  a*a + y*y < n (for fixed a)}
| a := a + 1;
| s := s + t;
end;
{a*a >= n, therefore s is the total number of solutions}
```

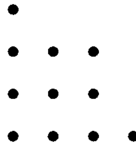
Here the ellipsis represents part of the program that is still to be written. Here it is:

```
b := 0; t:= 0;
{invariant relation: t is the number of integers y
  such that a*a + y*y < n and 0<=y<b }
while a*a + b*b < n do begin
| b := b + 1;
| t := t + 1;
end;
{a*a + b*b >= n,  so t is the number of nonnegative
   integers y such that a*a + y*y < n}
```

■

1.1.29. The same problem with the additional restriction that the total number of operations should be of order $\sqrt{n}$. (The previous solution requires about n operations.)

Solution. We have to count all the integer grid points in the first quadrant that lie inside the circle of radius $\sqrt{n}$. The set in question (call it X) is a union of columns of points having width 1 and non-increasing height.



The idea is to trace the boundary of this set, which resembles a staircase that goes down as we move from left to right. The current position is <a,b>. We use one more variable s and maintain the following invariant relation:

<a,b> is on the top of a-th column;
s is the number of points in the preceding columns.

Formally,

- b is minimal among all $b \geq 0$ such that <a,b> is not in X;
- s is the number of all pairs <x,y> of nonnegative integers such that $x < a$ and $\langle x,y \rangle \in X$.

These conditions will be denoted by (I).

```
a := 0; b := 0;
while <0,b> is in X do begin
| b := b + 1;
end;
```

```
{a = 0, b is minimal among all b >= 0
 such that <a,b> is not in X }
s := 0;
{invariant relation: (I)}
while not (b = 0) do begin
| s := s + b;
| {s is the number of points in columns 0..a}
| a := a + 1;
| {point <a,b> is outside X, it should be moved down to
|    restore (I) unless (I) is already true}
| while (b <> 0) and (<a, b-1> is not in X) do begin
| | b := b - 1;
| end;
end;
{(I), b = 0, therefore the a-th column and all subsequent
  columns are empty; s is the required number}
```

An estimate for the number of steps is evident. First we move up performing not more than $\sqrt{n}$ steps. Then we move right and down in not more than $\sqrt{n}$ steps in each direction. ∎

1.1.30. Nonnegative integers n and k are given, with $n > 1$. Print k digits of the decimal representation of the number $1/n$. (If two decimal representations exist, such as $0.499\ldots = 0.500\ldots$, print the latter.) The program should use integer variables only.

Solution. Moving the decimal point of the number $1/n$, k positions to the right, we get the number $10^k/n$. We wish to print its integer part, that is, we must compute 10^k div n. We do not want to compute 10^k because of the possibility of integer overflow. Instead, we perform ordinary division. Here is the program:

```
m := 0;
r := 1;
{m digits of 1/n  are printed; it remains to print
  k - m digits of the decimal expansion of r/n}
while m <> k do begin
| write ( (10 * r) div n);
| r := (10 * r) mod n;
| m := m + 1;
end;
```

∎

1.1.31. A natural number $n > 1$ is given. Find the length of the period of the decimal number $1/n$.

Solution. The period of a decimal fraction is equal to the period of the sequence of "remainders" r (see the solution of the preceding problem). [Prove this fact;

do not forget to prove that the period of the fraction cannot be less than the period of the sequence of remainders.] In the sequence of remainders all terms that form the period are distinct and the length of the nonperiodic initial segment does not exceed n. Therefore, it is enough to find the (n + 1)-th term of the sequence, and then to find the minimal k such that the (n+1+k)-th term is equal to the (n+1)-th term.

```
m := 0;
r := 1;
{r/n = what remains from 1/n after the decimal point
  is moved m positions to the right and the integral
  part is discarded}
while m <> n+1 do begin
| r := (10 * r) mod n;
| m := m + 1;
end;
c := r;
{c = (n+1)-th term of the sequence of remainders}
r := (10 * r) mod n;
k := 1;
{r = (n+k+1)-th term of the same sequence}
while r <> c do begin
| r := (10 * r) mod n;
| k := k + 1;
end;
```
■

1.1.32. (R.W. Floyd, communicated by Yu.V. Matijasevich) A function f : {1..N} → {1..N} is given. Find the period of the sequence 1, f(1), f(f(1)), ... The number of operations should be proportional to the length of the smallest initial segment that includes the period (this length may be significantly less than N).

Solution. After discarding the initial segment, we have a periodic sequence, and all terms in the period are different.

```
{Notation: f[n,1]=f(f(...f(1)...)) (n times)}
k := 1;
a := f(1);
b := f(f(1));
{a = f[k,1]; b = f[2k,1]}
while a <> b do begin
| k:=k+1; a:=f(a); b:=f(f(b));
end;
{a = f[k,1] = f[2k,1]; f[k,1] is in the periodic part}
m := 1;  b := f(a);
{b = f[k+m,1]; f[k,1],...,f[k+m-1,1] are different}
```

```
while a <> b do begin
| m:=m+1; b:=f(b);
end;
{period = m}
```
■

1.1.33. (E. Dijkstra). A function f whose arguments and values are nonnegative integers is defined as follows: $f(0) = 0$, $f(1) = 1$, $f(2n) = f(n)$, $f(2n+1) = f(n) + f(n+1)$. Write a program that computes $f(n)$ for a given n; the number of operations should be of order $\log n$.

Solution.

```
k := n; a := 1; b := 0;
{invariant relation: 0<=k, f(n) = a*f(k) + b*f(k+1)}
while k <> 0 do begin
| if k mod 2 = 0  then begin
| | l := k div 2;
| | {k=2l, f(k)=f(l), f(k+1) = f(2l+1) = f(l) + f(l+1),
| |  f (n) = a*f(k) + b*f(k+1) = (a+b)*f(l) + b*f(l+1)}
| | a := a + b; k := l;
| end else begin
| | l := k div 2;
| | {k = 2l + 1, f(k) = f(l) + f(l+1),
| |  f(k+1) = f(2l+2) = f(l+1),
| |  f(n) = a*f(k) + b*f(k+1) = a*f(l) + (a+b)*f(l+1)}
| | b := a + b; k := l;
| end;
end;
{k = 0, f(n) = a * f(0) + b * f(1) = b,
 b is the answer}
```
■

1.1.34. The same problem for $f(0) = 13$, $f(1) = 17$, $f(2n) = 43\,f(n) + 57\,f(n+1)$ and $f(2n+1) = 91\,f(n) + 179\,f(n+1)$ for $n \geq 1$.

[Hint. The program stores k, a, b, c such that $f(n) = a \cdot f(k) + b \cdot f(k+1) + c \cdot f(k+2)$.] ■

1.1.35. Two nonnegative integers a and b are given, with $b > 0$. Find a mod b and a div b using only integer variables and avoiding explicit div and mod operations (the only exception: an even number may be divided by 2). The number of operations should not exceed $C_1 \log(a/b) + C_2$ for some constants C_1 and C_2.

Solution.

```
b1 := b;
while b1 <= a do begin
| b1 := b1 * 2;
end;
{b1 > a, b1 = b * (integer power of 2)}
q:=0; r:=a;
{invariant relation: q, r are quotient and remainder when
 a is divided by b1; b1 = b * (some integer power of 2)}
while b1 <> b do begin
| b1 := b1 div 2 ; q := q * 2;
| { a = b1 * q + r, 0 <= r < 2 * b1}
| if r >= b1 then begin
| | r := r - b1;
| | q := q + 1;
| end;
end;
{q, r are quotient and remainder when a is divided by b}
```
■

1.2 Arrays

We assume in the sequel that x, y, z are defined as `array[1..n] of integer` (here n is a fixed positive integer constant) unless otherwise stated.

1.2.1. Fill the array x with zeros. (Write a program fragment whose execution guarantees that all values `x[1]..x[n]` are zero independent of the initial value of x.)

Solution.

```
i := 0;
{invariant relation: x[1],..,x[i] = 0}
while i <> n do begin
| i := i + 1;
| {x[1]..x[i-1] = 0}
| x[i] := 0;
end;
```
■

1.2.2. Count the number of zeros in an array x. (Write a program fragment that does not change the value of x and guarantees that the integer variable k contains the number of zeros among `x[1]..x[n]`.)

Solution.

```
...
{invariant: k = number of zeros among x[1]..x[i] }
...
```
■

1.2.3. Not using assignment statement for arrays, write a program that is equivalent to the statement `x:=y`.

Solution.

```
i := 0;
{invariant: y is unchanged, x[t]=y[t] for all t<=i}
while i <> n do begin
| i := i + 1;
| x[i] := y[i];
end;
```

■

1.2.4. Find the maximum value among `x[1]..x[n]`.

Solution.

```
i := 1; max := x[1];
{invariant relation: max = maximum(x[1]..x[i])}
while i <> n do begin
| i := i + 1;
| {max = maximum(x[1]..x[i-1])}
| if x[i] > max then begin
| | max := x[i];
| end;
end;
```

■

1.2.5. An array `x: array[1..n] of integer` is given such that $x[1] \leq x[2] \leq \cdots \leq x[n]$. Find the number of different elements among `x[1]..x[n]`.

Solution (version 1).

```
i := 1; k := 1;
{invariant relation: k = the number of
 different elements among x[1]..x[i]}

while i <> n do begin
| i := i + 1;
| if x[i] <> x[i-1] then begin
| | k := k + 1;
| end;
end;
```

(version 2) The number in question is one unit larger than the number of `i` in `1..n-1` such that `x[i]` is not equal to `x[i+1]`, plus one.

```
k := 1;
for i := 1 to n-1 do begin
| if x[i]<> x[i+1] then begin
| | k := k + 1;
| end;
end;
```

■

1.2.6. An array `x: array[1..n] of integer` is given. Compute the number of different elements among `x[1]..x[n]`. (The number of operations should be of order n^2.) ■

1.2.7. The same problem with an additional requirement: the number of operations should be of order $n \log n$.

[Hint. See chapter 4 on sorting.] ■

1.2.8. The same problem where all elements are integers in `1..k` and the number of operations should be of order $n + k$. ■

1.2.9. (Communicated by A.L. Brudno.) A rectangular field $m \times n$ contains mn squares. Some squares are marked as black. It is known that black squares are grouped into several disjoint rectangles that are at least one apart from each other. Assuming that the colors of squares are represented as

```
array [1..m] of array [1..n] of Boolean;
```

count the number of rectangles. The number of operations should be of order mn.

Solution. The number of rectangles is equal to the number of their upper left corners. It is easy to check whether a square is in the upper left corner. Just check the color of the cell as well as the colors of its upper and left neighbors. (Don't forget the case when the cell is on the left or upper boundary of a given $m \times n$ rectangle.) ■

1.2.10. An array `x[1]..x[n]` is given. Without using other arrays, put its elements in reverse order.

Solution. We should exchange `x[i]` and `x[n+1-i]` for all `i` such that $i < n + 1 - i$, i.e., $2i < n + 1 \Leftrightarrow 2i \le n \Leftrightarrow i \le n \text{ div } 2$:

```
for i := 1 to n div 2 do begin
| ...exchange x[i] and x[n+1-i];
end;
```

■

1.2.11. (From D. Gries' book [6]) An array `x[1]..x[m+n]` is considered as a concatenation of two segments: a prefix `x[1]..x[m]` of length `m` and a suffix `x[m+1]..x[m+n]` of length `n`. Without using other arrays, exchange these prefix and suffix segments. (The number of operations should be of order $m + n$.)

Solution. (version 1) Reverse the prefix segment (see the preceding problem), then the suffix segment, and finally the whole array.

(version 2, A.G. Kushnirenko) Imagine that the array is written down along a circle. Then the required transformation is a rotation. Recall that rotation may be represented as the composition of two axial symmetries. Each symmetry can be performed by exchanges without extra memory.

(version 3) Consider the more general problem: Exchange two adjacent segments `x[p+1]..x[q]` and `x[q+1]..x[r]` in an array. Assume that the length of the left segment (called A in the sequel) does not exceed the length of the right segment (called B). Split B into two segments B_1 and B_2, where B_1 is an initial segment of B of the same length as A. (So, $B = B_1 + B_2$, where $+$ stands for concatenation.) We need to transform $A+B_1+B_2$ into B_1+B_2+A. We can easily exchange A and B_1 because they have equal lengths. After that we get B_1+A+B_2 and it remains to exchange A and B_2. Therefore, we have reduced our problem to a similar problem for shorter segments. Here is the outline of the program:

```
p := 0; q := m; r := m + n;
{invariant relation: it remains
  to exchange x[p+1..q], x[q+1..r]}
while (p <> q) and (q <> r) do begin
| {both segments are nonempty}
| if (q - p) <= (r - q) then begin
| | ..exchange x[p+1]..x[q] and x[q+1]..x[q+(q-p)]
| | pnew := q; qnew := q + (q - p);
| | p := pnew; q := qnew;
| end else begin
| | ..exchange x[q-(r-q)+1]..x[q] and x[q+1]..x[r]
| | qnew := q - (r - q); rnew := q;
| | q := qnew; r := rnew;
| end;
end;
```

The number of operations may be estimated as follows. At each step the part of the array that should be processed becomes shorter by the length of A. The number of operations required is also proportional to the length of A. ■

1.2.12. An array `a: array[0..n] of integer` contains the coefficients of a polynomial of degree n. Compute the value of this polynomial at the point x, that is, $a[n]\,x^n + \cdots + a[1]\,x + a[0]$.

Solution. (The algorithm described below is called Horner's rule)

```
k := 0; y := a[n];
{invariant relation: 0 <= k <= n,
 y= a[n]*(x ** k)+...+a[n-1]*(x ** (k-1))+...+
                     + a[n-k]*(x ** 0)}
```

```
while k <> n do begin
| k := k + 1;
| y := y * x + a [n-k];
end;
```

■

1.2.13. (Requires some calculus; communicated by A.G. Kushnirenko) Extend Horner's rule to compute not only the value of a polynomial at some point, but also the value of the derivative of the same polynomial at the same point.

Solution. When a new coefficient is added, the polynomial changes from $P(x)$ to $Q(x) = xP(x)+c$. The derivative $Q'(x)$ is equal to $xP'(x)+P(x)$. Therefore we can easily compute $Q(x)$ and $Q'(x)$ if we know x, c, $P(x)$ and $P'(x)$. ■

This solution has a unexpected feature: we do not need to know in advance the degree of the polynomial. If we add this requirement and ask to compute the value of the derivative only (not mentioning the polynomial itself), we get a rather confusing problem.

There is a general statement about the computation of derivatives:

1.2.14. (W. Baur, V. Strassen) Assume that a "straight-line" program (containing only assignment statements) computes the value of some polynomial $P(x_1,\ldots,x_n)$ given the variables $x_1,\ldots,x_n$. We assume that the right-hand sides of the assignment statements are expressions that contain only addition, multiplication, constants, variables $x_1,\ldots,x_n$ and the variables that appear on the left-hand side of previous assignment statements. Prove that there exists a program of the same type that computes all n derivatives $\partial P/\partial x_1,\ldots,\partial P/\partial x_n$, where the number of arithmetic operations is only C times larger than in the original program. (Here the constant C does not depend on n.)

[Hint. We may assume that each assignment consists of addition, multiplication by a constant, or multiplication of two variables. Use induction on the number of statements, applying the inductive assumption to the program obtained by deleting the *first* command of the program.] ■

1.2.15. Two arrays `a: array[0..k] of integer` and `b: array[0..l] of integer` contain the coefficients of two polynomials of degrees `k` and `l` respectively. Put into `c: array[0..m] of integer` the coefficients of their product. (Here `k`, `l`, `m` are nonnegative integers such that `m` = `k` + `l`; the array element indexed by `i` contains the coefficient of $\mathtt{x}^{\mathtt{i}}$.)

Solution.

```
for i:=0 to m do begin
| c[i]:=0;
end;
```

```
for i:=0 to k do begin
| for j:=0 to 1 do begin
| | c[i+j] := c[i+j] + a[i]*b[j];
| end;
end;
```
■

1.2.16. The multiplication algorithm for polynomials given above uses about n^2 operations to compute the product of two polynomials of degree n. Find an (asymptotically) more effective algorithm that uses only $O(n^{\log 4/\log 3})$ operations.

[Hint. Suppose we want to multiply two polynomials of degree $2k$. Represent these polynomials as

$$A(x)\,x^k + B(x) \quad \text{and} \quad C(x)\,x^k + D(x)$$

where A, B, C, D are polynomials of degree k. The product in question is equal to

$$A(x)C(x)\,x^{2k} + (A(x)D(x) + B(x)C(x))\,x^k + B(x)D(x).$$

The natural way to compute AC, $AD + BC$, BD requires four multiplications of degree k polynomials. However, the following trick requires only three multiplications: compute AC, BD and $(A + B)(C + D)$, then use the identity $AD + BC = (A + B)(C + D) - AC - BD$.] ■

1.2.17. Two arrays `x: array[1..k] of integer` and `y: array[1..l] of integer` are sorted (`x[1]` $< \cdots <$ `x[k]`, `y[1]` $< \cdots <$ `y[l]`). Find the number of common elements in both arrays, that is, the number of integers `t` such that `t` $=$ `x[i]` $=$ `y[j]` for some `i` and `j`. (The number of operations should be of order k + l.)

Solution.

```
k1:=0; l1:=0; n:=0;
{invariant relation: 0<=k1<=k; 0<=l1<=l;
 the number in question is n plus the number of common
 elements in x[k1+1],...,x[k] and y[l1+1],...,y[l]}
while (k1 <> k) and (l1 <> l) do begin
| if x[k1+1] < y[l1+1] then begin
| | k1 := k1 + 1;
| end else if x[k1+1] > y[l1+1] then begin
| | l1 := l1 + 1;
| end else begin {x[k1+1] = y[l1+1]}
| | k1 := k1 + 1;
| | l1 := l1 + 1;
| | n := n + 1;
| end;
end;
```

```
{k1 = k or l1 = l; therefore, one of the sets
 mentioned in the invariant relation is empty
 and n is the number in question}
```

Remark. In the last alternative it is enough to increase only one of the variables k1 and l1 (though the symmetry would be broken if we did that). ■

1.2.18. Solve the preceding problem with the assumption that x[1] $\le \cdots \le$ x[k] and y[1] $\le \cdots \le$ y[l] (arrays are nondecreasing but not necessarily increasing).

Solution. In the third alternative of the previous solution, when increasing k1 and l1 by 1, we decreased (by 1) the number of common elements in x[k1+1]...x[k] and x[l1+1]...x[l]. For nondecreasing arrays, this is not enough since the same element may appear many times. A more complicated procedure is required:

```
...
end else begin {x[k1+1] = y[l1+1]}
| t := x [k1+1];
| while (k1<k) and (x[k1+1]=t) do begin
| | k1 := k1 + 1;
| end;
| while (l1<l) and (x[l1+1]=t) do begin
| | l1 := l1 + 1;
| end;
| n := n+1;
end;
```

Remark. This program has a bug, however. If in the condition

```
(k1<k) and (x[k1+1]=t)
```

(or in the similar second condition) the first expression (k1<k) is false, the second one is meaningless (index out of bounds) and an error may occur. Some versions of Pascal use "short circuit evaluation" of Boolean expressions: when evaluating A and B the evaluation of B is "short circuited" when A is false. (This is the default behaviour of Turbo Pascal 5.0, but not 3.0.) In this case, the problem disappears.

Rather than rely on implementation-dependent features (short-circuit evaluation is not prescribed by the Pascal's author, N. Wirth), we can do the following. Introduce an additional variable b: Boolean and write:

```
if k1 < k  then b := (x[k1+1]=t)  else  b:=false;
{b = (k1<k) and (x[k1+1] = t)}
while  b  do  begin
| k1 := k1+1;
| if k1 < k then b := (x[k1+1]=t)  else  b:=false;
end;
```

Another possibility (which is shorter, but less symmetric):

```
end else begin {x[k1+1] = y[l1+1]}
  if k1 + 1 = k then begin
    k1 := k1 + 1;
    n := n + 1;
  end else if x[k1+1] = x [k1+2] then begin
    k1 := k1 + 1;
  end else begin
    k1 := k1 + 1;
    n := n + 1;
  end;
end;
```

Alternatively, we can increase the constant in the array declaration and reserve a spare memory location. ■

1.2.19. Two arrays `x: array[1..k] of integer` and `y: array[1..l] of integer` satisfying $x[1] \le \cdots \le x[k]$, $y[1] \le \cdots \le y[l]$ are given. Find the number of different elements among $x[1], \ldots, x[k], y[1], \ldots, y[l]$. (The number of operations should be of order $k + l$.) ■

1.2.20. Two arrays $x[1] \le \cdots \le x[k]$ and $y[1] \le \cdots \le y[l]$ are given. Merge them into one array $z[1] \le \cdots \le z[m]$ $(m = k + l)$. Any element should appear is z as many times as it appears in x and y together. The number of operations should be of order m.

Solution.

```
k1 := 0; l1 := 0;
{invariant relation: the answer is the concatenation
  of z[1]..z[k1+l1] and the merge of
  x[k1+1]..x[k] and y[l1+1]..y[l]}
while (k1 <> k) or (l1 <> l) do begin
  if k1 = k then begin
    {l1 < l}
    l1 := l1 + 1;
    z[k1+l1] := y[l1];
  end else if l1 = l then begin
    {k1 < k}
    k1 := k1 + 1;
    z[k1+l1] := x[k1];
  end else if x[k1+1] <= y[l1+1] then begin
    k1 := k1 + 1;
    z[k1+l1] := x[k1];
  end else if x[k1+1] >= y[l1+1] then begin
```

```
| | l1 := l1 + 1;
| | z[k1+l1] := y[l1];
| end else begin
| | {this cannot happen}
| end;
end;
{k1 = k, l1 = l, arrays are merged}
```

This process can be illustrated as follows. Assume we have two piles of cards with a word on each card, and each pile is alphabetically sorted. We merge them into one pile as follows. At every step we compare the first cards of both piles and take the one which is alphabetically first. If one pile is already empty, we take the remaining cards from the other pile. ■

1.2.21. Two arrays x[1] $\le \cdots \le$ x[k] and y[1] $\le \cdots \le$ y[l] are given. Find their "intersection", i.e., an array z[1] $\le \cdots \le$ z[m] that contains their common elements. The multiplicity of each element in z should be equal to the smaller of its multiplicities in x and y. The number of operations should be of order k + l. ■

1.2.22. Two arrays x[1] $\le \cdots \le$ x[k] and y[1] $\le \cdots \le$ y[l] and a number q are given. Find i and j such that x[i] + y[j] is as close to q as possible. (The number of operations should be of order k+l. You may use a fixed number of auxiliary integer variables; the arrays x and y are read-only.)

[Hint. We need to find the minimal distance between x[1] $\le \cdots \le$ x[k] and q − y[l] $\le \cdots \le$ q − y[1]. This is easily done while merging these numbers into one (imaginary) array.] ■

1.2.23. (from D. Gries' book [6]) There is a number that is present in all three nondecreasing arrays x[1] $\le \cdots \le$ x[p], y[1] $\le \cdots \le$ y[q], z[1] $\le \cdots \le$ z[r]. Find this number (or one of them, if there is more than one). The number of operations should be of order p + q + r.

Solution.

```
p1:=1; q1=1; r1:=1;
{invariant relation: x[p1]..x[p], y[q1]..y[q],
 z[r1]..z[r] have an element in common}
while not ((x[p1]=y[q1]) and (y[q1]=z[r1])) do begin
| if x[p1]<y[q1] then begin
| | p1:=p1+1;
| end else if y[q1]<z[r1] then begin
| | q1:=q1+1;
| end else if z[r1]<x[p1] then begin
| | r1:=r1+1;
| end else begin
```

```
| | {this cannot happen}
|  end;
end;
{x[p1] = y[q1] = z[r1]}
writeln (x[p1]);
```
■

1.2.24. Repeat the previous problem assuming that we do not know in advance if such a common element exist. Determine whether or not it exists and locate it if it does. ■

1.2.25. The array `a[1..n]` consists of arrays `[1..m]` of integers:

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

$$a[1][1] \le \cdots \le a[1][m],\ \ldots,\ a[n][1] \le \cdots \le a[n][m].$$

It is known that there is a common number present in all `a[i]` (that is, there exists an x such that for all i in `1..n` there exists a j in `1..m` such that `a[i][j]` $=$ x). Find such a number x.

Solution. We use an array `b[1]..b[n]` whose elements mark the start of the "non-scanned" portions of arrays `a[1],...,a[n]`.

```
for k:=1 to n do begin
|  b[k]:=1;
end;
eq := true;
for k := 2 to n do begin
| eq := eq and (a[1][b[1]] = a[k][b[k]]);
end;
{invariant relation: non-scanned parts have nonempty
 intersection, i.e., there is x such that for any i in
 [1..n] there is j in [b[i]..m] such that a[i][j] = x;
 eq <=> first non-scanned elements are all equal}
while not eq do begin
| s := 1; k := 1;
| {a[s][b[s]] is minimal among a[1][b[1]]..a[k][b[k]]}
| while k <> n do begin
| | k := k + 1;
| | if a[k][b[k]] < a[s][b[s]] then begin
| | | s := k;
| | end;
| end;
| {a[s][b[s]] is minimal among a[1][b[1]]..a[n][b[n]]}
| b [s] := b [s] + 1;
```

```
  | for k := 2 to n do begin
  | | eq := eq and (a[1][b[1]] = a[k][b[k]]);
  | end;
end;
writeln (a[1][b[1]]);
```

■

1.2.26. Our solution of the preceding problem requires mn^2 operations. Find an algorithm that needs only $O(mn)$ operations (i.e., not more than Cmn operations for some C).

[Hint. We have to break the symmetry and choose one of the rows as a "principal" row. We move along the principal row maintaining the following relation: in all other rows the maximal element not exceeding the current element of the principal row is located.] ■

1.2.27. (Binary search) An array x[1] ≤ ··· ≤ x[n] of integers and an integer a are given. Determine if a is present in x, that is, if there exists an i in 1..n such that x[i] = a. (The number of operations should be of order log n.)

Solution. (We assume that n > 0.)

```
l := 1; r := n+1;
{r > l, if a is present, it is present among x[l]..x[r-1]}
while r - l <> 1 do begin
| m := l + (r-l) div 2 ;
| {l < m < r }
| if x[m] <= a then begin
| | l := m;
| end else begin {x[m] > a}
| | r := m;
| end;
end;
```

(Check that the invariant relation is maintained even if x[m] = a.)

At each step the difference r − l is halved, so we get the required bound for the number of operations.

Program can be simplified using the equality

$$l + (r-l)\,\mathrm{div}\,2 = (2l + (r - l))\,\mathrm{div}\,2 = (r + l)\,\mathrm{div}\,2.$$

■

Remark. It is very important that the array x[1]..x[n] is sorted; otherwise we have to test all n elements x[1]..x[n] to be sure that a given element is not in the array ("sequential search").

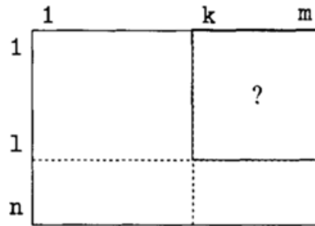
1.2.28. (From D. Gries' book [6]) An array x: array[1..n] of array[1..m] of integer is sorted both row-wise and column-wise:

$$x[i][j] \le x[i][j+1],$$

x[i][j] ≤ x[i+1][j],

Determine if a given number a is present among the array elements x[i][j].

Solution. Represent x as a rectangular matrix. Choose a rectangle that contains a (assuming that a is present at all) and then make this rectangle smaller and smaller. This rectangle contains x[i][j] such that 1 ≤ i ≤ l and k ≤ j ≤ m.



(The rectangle is empty if l = 0 or k = m + 1.)

```
l:=n; k:=1;
{l>=0, k<=m+1, if a is present at all, it is present
 inside the rectangle}
while (l > 0) and (k < m+1) and (x[l][k] <> a) do begin
| if x[l][k] < a then begin
| | k := k + 1; {left column cannot contain a, delete it}
| end else begin {x[l][k] > a}
| | l := l - 1; {last row cannot contain a, delete it}
| end;
end;
{x[l][k] = a or the rectangle is empty}
answer:= (l > 0) and (k < m+1) ;
```

Remark. Here the same error as in problem 1.2.18 appears: x[l][k] may be undefined. (We leave its correction to the reader.) ■

1.2.29. (Moscow programming contest) A nondecreasing integer array a[1] ≤ a[2] ≤ ··· ≤ a[n] contains positive numbers only. Find the minimal positive integer that cannot be represented as a sum of several elements of this array (no element may be used not more than once). The number of operations should be of order n.

Solution. Assume all numbers that can be represented as sums of subsets of {a[1], ..., a[k]} form the set {1, 2, ..., N}. If a[k+1] > N+1, then N+1 is the smallest number that cannot be represented as the sum of some subset of {a[1], ..., a[n]}. If a[k+1] ≤ N+1, then all numbers that can be represented as sums of subsets of {a[1] ... a[k+1]} form the set {1, 2, ..., N+a[k+1]}.

```
k := 0; N := 0;
{invariant relation: all the numbers that can be
 represented as sums of subsets of {a[1],..,a[k]},
 form the set {1,2,...,N}}
while (k <> n) and (a[k+1] <= N+1) do begin
| N := N + a[k+1];
| k := k + 1;
end;
{(k = n) or (a[k+1] > N+1); the answer is N+1
 in both cases}
writeln (N+1);
```

(Error: when the first condition in the `while`-construct is false, the second is undefined.) ■

1.2.30. (Requires some algebra) An integer array `a[1]..a[n]` contains some permutation of `1..n` (each of numbers `1..n` appears exactly once).

(a) Determine if the permutation is even.

(b) Without using other arrays, replace the permutation by its inverse permutation (i.e., if `a[i]` $=$ `j` was true before execution, then `a[j]` $=$ `i` is true after execution).

(In both (a) and (b), the number of operations should be of order `n`.)

[Hint. (a) The number of cycles determines whether a permutation is even or odd. To mark an already counted cycle, we can (for example) change the sign of its elements. (b) The inverse permutation is computed cycle by cycle.] ■

1.2.31. An array `a[1..n]` and a threshold `b` are given. Rearrange the elements of the array in such a way that all elements on the left of some boundary do not exceed `b` whereas all elements on the right of the boundary are greater than or equal to `b`. The number of operations should be proportional to `n`.

Solution.

```
l:=0; r:=n;
{invariant relation: a[1]..a[l]<=b; a[r+1]..a[n]>=b}
while l <> r do begin
| if a[l+1] <= b then begin
| | l:=l+1;
| end else if a[r] >=b then begin
| | r:=r-1;
| end else begin {a[l+1]>b; a[r]<b}
| | ..exchange a[l+1] and a[r]
| | l:=l+1; r:=r-1;
| end;
end;
```

■

1.2.32. Repeat the previous problem with the additional restriction that the elements smaller than b should precede elements equal to b which themselves should precede elements greater than b.

Solution. We need three boundaries to divide our segment into four parts. The first part contains elements smaller than b; the second part contains only elements equal to b; the third part may contain anything; and the fourth part contains only elements greater than b. (We can get a more symmetric solution using a fourth boundary, but that's not important.) At each step we consider the left element of the third part (just to the right of the second boundary).

```
l:=0; m:=0; r:=n;
{invariant relation:
   a[1..l]<b; a[l+1..m]=b; a[r+1]..a[n]>b}
while m <> r do begin
| if a[m+1]=b then begin
| | m:=m+1;
| end else if a[m+1]>b then begin
| | ..exchange a[m+1] and a[r]
| | r:=r-1;
| end else begin {a[m+1]<b}
| | ..exchange a[m+1] and a[l+1]
| | l:=l+1; m:=m+1;
end;
```

■

1.2.33. (This version of the preceding problem is called the "Dutch flag" problem in E. Dijkstra's book [4].) The array contains n elements; each element is equal to 0, 1, or 2. Sort the array if the only allowed operation (besides reading its elements) is the exchange of two elements of the array. The number of operations should be proportional to n. ■

1.2.34. An array a[1..n] and a number $m \leq n$ are given. For any segment formed by m adjacent elements (there are $n-m+1$ segments of this type) compute its sum. The total number of operations should be of order n.

Solution. When moving the segment to the right, add one element and subtract another. ■

1.2.35. A square matrix a[1..n][1..n] and a number $m \leq n$ are given. For any $m \times m$-subsquare, compute the sum of its elements. The total number of operations should be of order n^2.

Solution. First compute the sum for all horizontal rectangles of size $m \times 1$. (When such a rectangle is shifted to the right, one element is added and one is subtracted.) After computing all these sums, we compute the sums for squares. (When a square is shifted down, one rectangle is added and another rectangle is subtracted.) ■

1.2.36. An array a[1]..a[n] contains all integers in [0..n] except one. Find this omitted integer with fixed additional memory. Number of operations should be proportional to n.

[Hint. Add all the numbers.] ∎

1.3 Inductive functions (following A.G. Kushnirenko)

Let M be a set. Let f be a function whose arguments are finite sequences of elements of M and whose values belong to some other set N. The function f is called *inductive* if its value on the sequence x[1]..x[n] is uniquely determined by its value on the sequence x[1]..x[n-1] and by x[n], that is, if there is a function $F : N \times M \to N$ such that

$$f(\langle x[1],\dots,x[n]\rangle) = F(f(\langle x[1],\dots,x[n-1]\rangle), x[n]).$$

For example, the sum $x[1] + \cdots + x[n]$ is an inductive function (it is enough to know the sum $x[1] + \cdots + x[n-1]$ and the value of x[n] to compute $x[1] + \cdots + x[n]$). At the same time, the average value is not an inductive function; if we know x[n] and the average of $x[1],\dots,x[n-1]$, but have no information about n, we cannot compute the average of $x[1],\dots,x[n]$.

An inductive function can be computed as follows:

```
k := 0; f := f0;
{invariant relation:
  f is a value of the function on <x[1],...,x[k]>}
while  k<>n do begin
 | k := k + 1;
 | f := F (f, x[k]);
end;
```

Here f0 is the value of the function on the empty sequence (sequence of length 0). If f is defined only on nonempty sequences, the first line should be replaced by

```
k:=1; f:=f(<x[1]>);
```

If a given function f is not inductive, it is instructive to look for its *inductive extension*. By an inductive extension of f we mean an inductive function g whose values determine uniquely the values of f (i.e., there exists a function t such that

$$f(\langle x[1]\dots x[n]\rangle) = t(g(\langle x[1]\dots x[n]\rangle))$$

for all $\langle x[1]\dots x[n]\rangle$). One can prove that there exists a minimal extension F among all inductive extensions of a given function f. Here the word "minimal" means that for any other inductive extension g the values of F are determined uniquely by the values of g, that is, $F(x) = u(g(x))$ for some function u.

1.3.1. Find an inductive extension for the following functions:

(a) the average value of a sequence of real numbers;

(b) the number of elements in a sequence that are equal to its maximal element;

(c) the second largest element of the sequence (second from the top after the sequence is sorted in nondescending order);

(d) the maximal number of consecutive equal elements;

(e) the maximal length of a monotone (nonincreasing or nondecreasing) fragment composed of consecutive elements of a sequence;

(f) the number of groups of ones separated by zeros (in a 0-1-sequence).

Solution.

(a) As we have seen, the average value is not an inductive function. However, the average value is a ratio of two inductive functions. The first one is the sum of all the terms; the second one is the number of terms. Therefore, the combination ⟨the sum of all elements; the length⟩ is an inductive extension.

(b) ⟨the maximal element; the number of elements equal to the maximal element⟩;

(c) ⟨the maximal element; the second maximal element⟩;

(d) ⟨the maximal number of adjacent equal elements; the maximal number of adjacent equal elements at the end of the sequence; the last element⟩;

(e) ⟨the maximal length of a monotone fragment; the maximal length of a non-decreasing fragment at the end of the sequence; maximal length of a nonincreasing fragment at the end of the sequence; the last term of the sequence⟩;

(f) ⟨the number of 1-groups; the last term⟩. ■

1.3.2. (Communicated by D.V. Varsonofiev) Two sequences `x[1]..x[n]` and `y[1]..y[k]` of integers are given. Determine if the second sequence is a subsequence of the first one, that is, if it is possible to delete some terms of the first sequence to obtain the second one. The number of operations should be of order $\mathtt{n} + \mathtt{k}$.

Solution. (version 1) Reduce the problem to the same problem involving shorter sequences.

```
n1:=n;
k1:=k;
{invariant relation: the answer is TRUE <=>
  it is possible to get y[1]..y[k1] out of x[1]..x[n1]}
while (n1 > 0) and (k1 > 0) do begin
| if x[n1] = y[k1] then begin
| | n1 := n1 - 1;
| | k1 := k1 - 1;
| end else begin
| | n1 := n1 - 1;
| end;
end;
```

```
{n1 = 0 or k1 = 0; if k1 = 0, the answer is positive;
 if k1<>0 (and n1 = 0), the answer is negative}
answer := (k1 = 0);
```

We use the following fact: If x[n1] = y[k1] and y[1]..y[k1] is a subsequence of x[1]..x[n1], then y[1]..y[k1-1] is a subsequence of x[1]..x[n1-1].

(version 2) The function ⟨x[1]..x[n1]⟩ $\mapsto$ [the maximal k1 such that y[1]..y[k1] is a subsequence of x[1]..x[n1]] is inductive. ∎

1.3.3. Two sequences x[1]..x[n] and y[1]..y[k] of integers are given. Find the maximal length of a sequence that is a subsequence of both given sequences. The number of operations should be of order $n \cdot k$.

Solution (communicated by M.N. Weinzweig and A.M. Dimentman). Denote the maximal length of a common subsequence of sequences x[1]..x[p] and y[1]..y[q] by f(p, q). Then

$$x[p] \neq y[q] \Rightarrow f(p,q) = \max(f(p,q-1), f(p-1,q));$$
$$x[p] = y[q] \Rightarrow f(p,q) = \max(f(p,q-1), f(p-1,q), f(p-1,q-1)+1)$$

(In the second case, the maximum of three numbers is in fact equal to the third number because $f(p-1,q-1)+1 \geq f(p,q-1), f(p-1,q)$.)

Therefore we can construct a table of f-values. This table is of size $n \cdot k$. We can even proceed using only k (or n) memory locations if we compute (for $p = 1, 2, \ldots$) the array ⟨f(p,0), ..., f(p,k)⟩ (it is an inductive function of p). ∎

1.3.4. (from D. Gries' book [6]) A sequence of integers x[1], ..., x[n] is given. Find the maximum length of an increasing subsequence. (The number of operations should be of order $n \log n$).

Solution. The function in question is not inductive. However, it has the following inductive extension: it consists of the maximal length of the increasing subsequences (denoted by k in the sequel) and the numbers u[1], ..., u[k], where u[i] is the minimal last term of all increasing subsequences of length i. Evidently, $u[1] \leq \cdots \leq u[k]$. When a new term is appended to x, the values of u and k should be updated.

```
n1 := 1; k := 1; u[1] := x[1];
{invariant: k and u satisfy the description above}
while n1 <> n do begin
 | n1 := n1 + 1;
 | ...
 | {i is the maximal number in 1..k such that
 |   u[i] < x[n1]; i=0 if there is no such numbers}
```

```
  if i = k then begin
  | k := k + 1;
  | u[k+1] := x[n1];
  end else begin {i < k, u[i] < x[n1] <= u[i+1] }
  | u[i+1] := x[n1];
  end;
end;
```

The omitted fragment employs binary search (see 1.2.27, p. 1.2) In the invariant relation we assume that u[0] $= -\infty$ and u[k+1] $= +\infty$. The goal is u[i] $<$ x[n1] $\leq$ u[i+1].

```
i:=0; j:=k+1;
{u[i] < x[n1] <= u[j], j > i}
while (j - i) <> 1 do begin
| s := i + (j-i) div 2;     {i < s < j}
| if x[n1] <= u[s] then begin
| | j := s;
| end else begin {u[s] < x[n1]}
| | i := s;
| end;
end;
{u[i] < x[n1] <= u[j], j-i = 1}
```

Remark. We get a simpler (but not minimal) inductive extension if, for any i, we keep the maximal length of an increasing sequence whose last term is x[i]. This extension leads to an algorithm requiring n^2 operations. ∎

1.3.5. What changes are needed in the solution of the previous problem if we are looking for a maximal *nondecreasing* sequence? ∎

2 Generation of combinatorial objects

In this section, we deal with problems that require us to generate all the elements of some finite set one-by-one.

2.1 Sequences

2.1.1. Print all the sequences of length k composed of the numbers 1..n.

Solution. Let us print them in alphabetic order (a sequence a precedes sequence b if for some s their initial segments of length s are equal and the (s+1)-th term of a is smaller.) The first sequence in this ordering is <1,1,...,1>; the last one is <n,n,...,n>. We use an array x[1]..x[k] to store the last sequence printed.

```
..make x[1]...x[k] equal to 1
..print x
..make last[1]...last[k] make equal to n
{all sequences up to x (including x) are printed}
while x <> last do begin
| ...x := the successor of x
| ...print x
end;
```

Let us explain how to get the successor of x. By definition, the successor should have the same first s terms and larger (s+1)-th term. This is possible only if x[s+1] < n. To get the immediate successor, we find the maximal s with this property and increase the corresponding element by 1. In other words, we move along the sequence from right to left and find the rightmost term that is smaller than n (it does exist, because x<>last by assumption). Then we increase it by 1 and make all the subsequent terms equal to 1.

```
p := k;
while not (x[p] < n) do begin
| p := p-1;
end;
{x[p] < n, x[p+1] =...= x[k] = n}
x[p] := x[p] + 1;
for i := p+1 to k do begin
| x[i] := 1;
end;
```

Remark. If we use the numbers 0..n-1 instead of 1..n, then finding the successor corresponds to adding 1 in n-ary notation. ∎

2.1.2. The program above uses comparisons for arrays (x <> last). Eliminate this step by using a Boolean variable l and adding the requirement

$$1 \Leftrightarrow x = last$$

to the invariant relation. ■

2.1.3. Print all subsets of the set 1..k.

Solution. These subsets are in one-to-one correspondence with all sequences of 0s and 1s of length k. ■

2.1.4. Print all sequences of length k of positive integers such that the i-th term does not exceed i for all i. ■

2.2 Permutations

2.2.1. Print all permutations of 1..n (i.e., all sequences of length n that contain all the numbers 1...n).

Solution. We store the current permutation in an array x[1]..x[n]. Permutations are printed in lexicographic order. The first permutation (in this order) is ⟨1 2..n⟩. The last one is ⟨n..2 1⟩. How do we find the next permutation (in the lexicographic order)? When is it possible to increase the k-th term in a permutation without changing all preceding terms? The answer is: When the term is smaller than one of the next terms (i.e., terms with larger indices). Therefore, to find the next permutation we should find the maximum k for which it is possible, that is, a k such that

$$x[k] < x[k+1] > \cdots > x[n]$$

Next we increase x[k] but keep the increase as small as possible. This means that we must find the minimal number among x[k+1]..x[n] that is larger than x[k]. After we exchange x[k] with the number found, we have to rearrange x[k+1]..x[n] to make the permutation as small as possible. To achieve this goal, we put x[k+1]..x[n] in increasing order. (Fortunately, they are already arranged in *decreasing* order.)

Here's how to get the next permutation:

```
{<x[1]...x[n]> <> <n...2,1>}
k:=n-1;
{after x[k] terms go in decreasing order: x[k+1]>...>x[n]}
while x[k] > x[k+1] do begin
| k:=k-1;
end;
{x[k] < x[k+1] > ... >  x[n]}
t:=k+1;
{t <=n, all terms x[k+1] > ... > x[t] are bigger than x[k]}
while (t < n) and (x[t+1] > x[k]) do begin
| t:=t+1;
end;
```

```
{x[k+1] > ... > x[t] > x[k] > x[t+1] > ... > x[n]}
..exchange x[k] and x[t]
{x[k+1] > ... > x[n]}
..put x[k+1]...x[n] in reversed order
```

Remark. This program suffers from the usual problem: x[t+1] is undefined when t = n. ∎

2.3 Subsets

2.3.1. Generate all subsets of the set 1..n having k elements.

Solution. Each subset may be represented by a bit string of length n that contains exactly k 1s. (We'll consider another representation later.) We generate these bit strings in alphabetic order. There is a natural way to do this: Generate all bit strings and select those that contain exactly k 1s. However, this solution is considered as inefficient, because bit strings with k 1s form a tiny fraction of all bit strings of length n. In the program below the generation of each subsequent string requires not more than $C{\cdot}$n operations (for some constant C).

When is it possible to increase the s-th term of a bit string with k 1s without changing the preceding terms? If x[s] is changed from 0 to 1, we should replace 1 by 0 somewhere to keep the total number of 1s fixed. Therefore, it is necessary to have 1s on the right of x[s].

Conclusion: If we want to find the next bit string with k 1s, we need x[s] to be the rightmost 0 that has some 1s on the right. In this case we have x[s+1]=1 (otherwise, x[s] is not the rightmost one). Therefore, we should look for the maximal s such that x[s]=0 and x[s+1]=1:

```
x [            |0|1..1|0..0]
                ↑
                s
```

The term x[s+1] may be followed by several 1s and then several 0s. After we replace x[s] by 1 the next terms should be chosen to get the alphabetically first string, that is, 0s should precede 1s. Here is what we get:

first string: $\underbrace{0\ldots0}_{n-k}\underbrace{1\ldots1}_{k}$

last string: $\underbrace{1\ldots1}_{k}\underbrace{0\ldots0}_{n-k}$

How to find the next string after x[1]..x[n] (assuming it exists):

```
s := n - 1;
while not ((x[s]=0) and (x[s+1]=1)) do begin
| s := s - 1;
end;
```

```
{x[s] should be changed from 0 to 1}
num:=0;
for k := s to n do begin
| num := num + x[k];
end;
{num is the number of 1s among x[s]...x[n], the number
 of 0s is (length - number of 1s), that is, (n-s+1)-num}
x[s]:=1;
for k := s+1 to n-num+1 do begin
| x[k] := 0;
end;
{it remains to put num-1 1s at the end}
for k := n-num+2 to n do begin
| x[k]:=1;
end;
```

■

We can also represent a subset by a list of its elements. To obtain the unique representation we require that elements should be listed in increasing order. Now we come to the following problem:

2.3.2. Generate (in alphabetic order) all increasing sequences of length k consisting of the numbers 1..n. (Example: for n=5, k=2 we get ⟨12 13 14 15 23 24 25 34 35 45⟩.)

Solution. The first sequence is ⟨1 2..k⟩; the last one is ⟨(n-k+1)..(n-1) n⟩. When is it possible to increase the s-th element of the sequence? Answer: If it is less than n-k+s. After the s-th element is increased, all subsequent elements should form an arithmetic sequence with difference 1. Here is the algorithm:

```
s:=n;
while not (x[s] < n-k+s) do begin
| s:=s-1;
end;
{s-th element should be increased};
x[s] := x[s]+1;
for i := s+1 to n do begin
| x[i] := x[i-1]+1;
end;
```

■

2.3.3. Suppose we represent subsets of 1..n of cardinality k by *decreasing* sequences of length k. (Example: ⟨21 31 32 41 42 43 51 52 53 54⟩.) How do we generate these sequences in alphabetical order?

[Hint. Find the maximal s such that x[s+1]+1 < x[s]. (If it does not exist, let s=0.) Now increase x[s+1] by 1 and let all subsequent elements be as small as possible (x[t]=k+1-t for t>s).] ■

2.3.4. Solve the two preceding problems if alphabetic order is replaced by reversed alphabetic order. ■

2.3.5. Generate all injective mappings of the set 1..k into 1..n (assume that k $\le$ n). A mapping is injective if no two elements of 1..k are mapped to the same element of 1..n. Generation of each mapping should require no more that $C \cdot$ k operations.

[Hint. This problem can be reduced to generation of permutations and generation of subsets.] ■

2.4 Partitions

2.4.1. Generate all partitions of a given positive integer n, that is, all the representations of n as a sum of positive integers. We do not take the order of the summands into account. (Example: For n=4, partitions are 1+1+1+1, 2+1+1, 2+2, 3+1 and 4.)

Solution. Let us agree that (i) the summands are written in nonincreasing order; and (ii) the partitions are generated in alphabetic order. We store a partition in the initial part of an array x[1]..x[n]; the length of the partition is k, and the summands are x[1]..x[k]. At the beginning, k = n and all x[1]...x[n] are equal to 1. At the end, x[1] = n and k = 1.

When can we increase x[s] leaving all preceding elements unchanged? This is possible only if x[s-1] > x[s] or if s = 1. Moreover, x[s] may not be the last element of the partition (because an increase in x[s] should be compensated by a decrease in the subsequent elements). After x[s] is increased, all subsequent elements should be chosen as small as possible.

```
s := k - 1;
while not ((s=1) or (x[s-1] > x[s])) do begin
| s := s-1;
end;
{x[s] should be increased}
x[s] := x[s] + 1;
sum := 0;
for i := s+1 to k do begin
| sum := sum + x[i];
end;
{sum = the sum of terms after x[s]}
for i := 1 to sum-1 do begin
| x [s+i] := 1;
end;
k := s+sum-1;
```
■

2.4.2. In this problem, partitions are still represented as nonincreasing sequences, but now we want to generate them in reversed alphabetic order (e.g., for n=4, we would generate `4, 3+1, 2+2, 2+1+1, 1+1+1+1`).

[Hint. The rightmost term that may be decreased is the rightmost term not equal to 1. Find it and decrease it by 1. All subsequent terms should be taken as large as possible (equal to the selected term when possible; the last one may be smaller).] ■

2.4.3. Partitions are represented as nondecreasing sequences; generate them in alphabetic order. For example, when n = 4, we would generate `1+1+1+1, 1+1+2, 1+3, 2+2, 4.`

[Hint. The last term `x[k]` cannot be increased, but the term `x[k-1]` can. (Of course, the last one should be decreased to maintain the sum.) If the sequence is no longer nondecreasing, we combine two terms into one. If the sequence is still nondecreasing, then `x[k]` should be split into several terms equal to `x[k-1]` (except for the last one, which may be larger).] ■

2.4.4. Partitions are represented as nondecreasing sequences. Generate them in reversed alphabetic order. (For n = 4 we have `4, 2+2, 1+3, 1+1+2, 1+1+1+1.`)

[Hint. The element `x[s]` can be decreased only if `s=1` or `x[s-1]<x[s]`. If `x[s]` is not the last term, these conditions are sufficient. If it is the last one, then we must also have `x[s-1]` $\le$ $\lfloor$`x[s]/2`$\rfloor$ or `s=1`. (Here $\lfloor\alpha\rfloor$ stands for the integer part of α, that is, the greatest integer not exceeding α.)] ■

2.5 Gray codes and similar problems

Sometimes it is useful to generate objects in an order such that the next object is only a small modification of the preceding one. In this section, we consider several problems of this type.

Consider 2^n strings of length n containing only 0s and 1's.

2.5.1. Prove that it is possible to list all of them in an order such that two neighboring strings differ only in one bit.

Solution. Use induction on n. Assume that $x_1, \dots, x_k$ is such a sequence of n-bit strings (here $k = 2^n$; for any i, strings x_i and x_{i+1} differ only in one bit). Then the following sequence includes all $(n+1)$-bit strings and satisfies the desired condition:

$$0x_1, 0x_2, \dots, 0x_k, 1x_k, 1x_{k-1}, \dots, 1x_1$$

In geometric terms, the problem states that we can traverse the n-dimensional Boolean cube visiting each vertex exactly once. The solutions considers n-dimensional Boolean cube as composed of two $n-1$-dimensional Boolean

cubes; we traverse one of them (using the inductive assumption) and then switch to another one. ■

We'll return to this problem later.

2.5.2. Generate all sequences of length n composed of the numbers 1..k in such an order that neighboring sequences differ only in one place, and the numbers at this place differ by 1.

Solution. Consider a rectangular chess board of width n and height k. Place a piece in each column of the chess board. The position is represented by a sequence of n integers (each between 1 and k); the i-th number represents a position of the piece in the i-th column. At each piece we draw a small arrow that points up or down. Initially, all the pieces are in the first row and all the arrows point up. We move pieces according to the following rule: Find the rightmost piece that can be moved in the direction of the arrow on it, and move it. At the same time all the pieces on the right (they cannot move in the direction of their arrows) are turned over.

It is evident that at each step only one piece is moving, therefore only one term in the corresponding sequence is changed by 1. Let us prove by induction on n that all sequences of length n composed of the numbers 1..k will appear. The case n=1 is evident, so assume that $n > 1$. Divide all moves into two categories. The first category is formed by moves where the last (rightmost) piece is moving. The second category is formed by moves where the moving piece is not the last one. In this case the rightmost piece is near the border and is turned over. Therefore, each move of the second category is followed by k-1 moves of the first category; during this period the rightmost piece visits all the cells. Let us forget now about the rightmost piece. Then the first n-1 pieces are moving according to the prescribed rules. Therefore, by the induction assumption, all sequences of length n-1 appear exactly once. The movements of the last piece make k sequences of length n out of each sequence of length n-1.

The program keeps an array x[1]..x[n] (positions of pieces) and an array d[1]..d[n] composed of numbers +1 and -1 (+1 denotes up-arrow; -1 denotes down-arrow).

Initial state: $x[1] = \cdots x[n] = 1; d[1] = \cdots = d[n] = 1$.

The following algorithm produces the next position according to the description above. At the same time, it checks whether the next position exists; the answer is stored in a Boolean variable p.

```
{if possible, make a move and let p := true;
 otherwise, p := false }
i := n;
```

```
while (i > 1) and
 | (((d[i]=1) and (x[i]=n)) or ((d[i]=-1) and (x[i]=1)))
 |   do begin
 | i:=i-1;
end;
if (d[i]=1 and x[i]=n) or (d[i]=-1 and x[i]=1)
 |    then begin
 | p:=false;
end else begin
 | p:=true;
 | x[i] := x[i] + d[i];
 | for j := i+1 to n do begin
 | | d[j] := - d[j];
 | end;
end;
```

Remark. For the case k $= 2$ there is another solution that uses the binary system. (It is this solution that is usually associated with the name "Gray code".)

Let us write down all the numbers $0, \ldots, 2^n - 1$ in binary notation. For example, for $n = 3$ we have:

000 001 010 011 100 101 110 111

Each number is transformed according to the following rule: each digit (except the first one) is replaced by its sum (modulo 2) with the preceding (untransformed) digit. In other words, the number with binary digits $a_1, a_2, \ldots, a_n$ is transformed into the number with binary digits $a_1, a_1 + a_2, a_2 + a_3, \ldots, a_{n-1} + a_n$ (addition modulo 2). For $n = 3$, we get the following list:

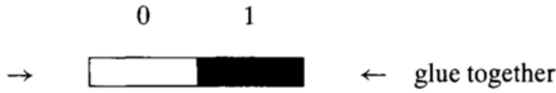
000 001 011 010 110 111 101 100

It is easy to check that the transformation described (which can be applied to any sequence of n binary digits, giving another sequence of the same length) is invertible. Therefore, the list obtained contains all sequences of length n.

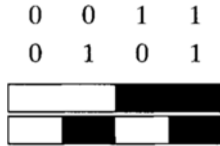
On the other hand, adding 1 to a number in binary notation means replacement of the suffix 011...1 by 100...0. This change leads to a change of exactly one digit after the transformation is applied. ∎

An application of Gray codes. Assume that some mechanical device has a rotating drum and we wish to get information about the position of this drum. If we make half the drum white, the remaining half black, and use a light sensor, we can measure the position of the drum up to 180°.

Drum cover:



If we make another track with black and white parts, and use a second light sensor, we can measure the position angle up to 90°:



With a third track,

the precision becomes 45°, etc. However, there is a problem with this scheme. When two light sensors change their state from black to white, this change may not happen at exactly the same time, and for a while the data are senseless.

We can use Gray codes to overcome this difficulty: we arrange the black and white sectors in such a way that only one track changes color each time. (This is also true for the last change after a complete rotation is performed.)

The above formula allows us to convert the sensor data into the corresponding rotation angle easily.

2.5.3. Generate all permutations of the numbers `1..n` in such a way that each permutation is obtained from the preceding one by an exchange (transposition) of two adjacent numbers. For example, for `n=3`, one of the possible answers is

`3.2 1` → `2 3.1` → `2.1 3` → `1 2.3` → `1.3 2` → `3 1 2`

(the dots indicate which numbers are exchanged at each step)

Solution. Put the set of all permutations into one-to-one correspondence with another set. This latter set contains all sequences y[1]..y[n] of nonnegative integers such that y[1] $\le$ 0, ..., y[n] $\le$ n-1. It has the same cardinality as the set of all permutations. The one-to-one correspondence is established as follows: Each permutation corresponds to the sequence y[1]..y[n], where y[i] is the number of j's such that both (a) j $<$ i and (b) j is located to the left of i in this permutation. Why is it an one-to-one correspondence? Any permutation of 1..n can be obtained from a permutation of 1..n-1 by inserting n into one of the n places (before the first term, between the first and the second terms,..., after the last term). What means this insertion for the corresponding sequence of integers? A number that ranges from 0 to n-1 is appended to the end while the other terms remain unchanged.

This one-to-one correspondence can be explained by the following metaphor. Consider n cards with numbers 1..n written on the cards, and a growing pile made of the cards. Initially the pile has only one card with number 1 written on it. At the next step we add the card with number 2. There are two possible positions for that card (either before the first card or after it). Then we add the card with number 3 on it; there are three posiible positions, etc. After we add the last card (there are n possible positions), we get a permutation of the numbers 1..n. This permutation is determined by positions chosen at steps 1..n; if we denote by y[i] the number of cards before the inserted card at step i, we get the one-to-one correspondence defined above.

We make one more remark about this correspondence. Assume that we increase or decrease y[i] by 1 for some i (leaving the other y[j]'s unchanged). Assume also that all subsequent y[j]'s have maximal or minimal values. In this case two adjacent numbers in our permutation are exchanged. Namely, an increase in y[i] means that i is exchanged with its right neighbor, while a decrease means that i is exchanged with its left neighbor.

Now recall how we generated all sequences of numbers 1..k in such a way that each sequence differs from the preceding sequence in one and only one place by using n $\times$ k rectangle. Now replace it by a board that resembles a staircase (the i-th column is a rectangle of width 1 and height i). Moving pieces according to the rules described above (using arrows on pieces), we traverse all the sequences, and the property mentioned above (that the i-th term changes only if all subsequent terms are maximal or minimal) holds.

To implement this scheme we need to modify the permutation according to the changes on the board. An obvious approach is to search for a given number i at each step. We can save ourselves some work if we keep (in addition to the permutation itself) the function

$$i \mapsto \text{position of } i \text{ in the permutation}$$

that is, the inverse mapping, and update both the permutation and its inverse. Here is the program:

```
program test;
  const n = ...;
  var
    x: array [1..n] of 1..n; {permutation}
    inv_x: array [1..n] of 1..n; {inverse permutation}
    y: array [1..n] of integer; {y[i] < i}
    d: array [1..n] of -1..1; {arrows}
    b: Boolean;

  procedure print_x;
    var i: integer;
  begin
    for i := 1 to n do begin
      write (x[i], ' ');
    end;
    writeln;
  end;

  procedure set_first; {first: y[i]=0 for all i}
    var i : integer;
  begin
    for i := 1 to n do begin
      x[i] := n + 1 - i;
      inv_x[i] := n + 1 - i;
      y[i] := 0;
      d[i] := 1;
    end;
  end;

  procedure move (var done : Boolean);
    var i, j, pos1, pos2, val1, val2, tmp : integer;
  begin
    i := n;
    while (i > 1) and (((d[i]=1) and (y[i]=i-1)) or
                 ((d[i]=-1) and (y[i]=0))) do begin
      i := i-1;
    end;
    done := (i > 1);
    {simplification: the first term cannot be changed}
    if done then begin
      y[i] := y[i] + d[i];
```

```
      for j := i+1 to n do begin
      | d[j] := -d[j];
      end;
      pos1 := inv_x[i];
      val1 := i;
      pos2 := pos1 + d[i];
      val2 := x[pos2];
      {pos1, pos2 are positions of elements to be
        exchanged; val1, val2 are its values; val2 < val1}
      tmp := x[pos1];
      x[pos1] := x[pos2];
      x[pos2] := tmp;
      tmp := inv_x[val1];
      inv_x[val1] := inv_x[val2];
      inv_x[val2] := tmp;
    end;
  end;

begin
  set_first;
  print_x;
  b := true;
  {all permutations up to the current one (including it)
   are printed;
   if b is false, the current one is the last one}
  while b do begin
  | move (b);
  | if b then print_x;
  end;
end.
```

■

2.6 Some remarks

Let us review the approach we've been using. We introduce some order on the objects to be generated and write a procedure that obtains the next object (in this order). In the Gray code problems, we were forced to maintain some additional information (directions of arrows). Finally, when generating permutations in such a way that only two numbers are exchanged at a time, we establish a one-to-one correspondence between the set to be generated and some other (presumable simpler) set. There are some cases where this trick is useful. In this section, we consider several problems of this type connected with the so-called Catalan numbers.

2.6.1. Generate all sequences of length 2n, composed of 1s and -1s satisfying the following conditions: (a) the sum of all terms is 0; (b) the sum of any prefix is nonnegative, that is, the number of -1s does not exceed the number of 1s. (The number of such sequences is called the *Catalan number*; see the formula for Catalan numbers on p. 48, problem 2.7.3.)

Solution. Represent 1 by a vector (1,1) and represent -1 by (1,-1). In terms of vectors, we are looking for all paths from (0,0) to (2n,0) that never go below the x-axis.

Let us generate the sequences in alphabetic order (assuming that -1 precedes 1). The first sequence is the "zig-zag"

```
1, -1, 1, -1, ...
```

The last sequence will be the sequence

```
1, 1, 1,..., 1, -1, -1,..., -1.
```

But how do we generate the next sequence? It should coincide with the current sequence up to some point where they differ and -1 is replaced by 1. This place should be as close to the end as possible. But there is a restriction; -1 may be replaced by 1 only if there is 1 on the right of it (which can be replaced by -1). After we replace -1 by 1, we are faced with the following problem: A prefix of the sequence is fixed; find the minimal sequence with that prefix. The solution: extend the given prefix step by step; at each step append -1 if possible (the sum must be nonnegative); otherwise, append 1. Here is the resulting program:

```
...
type array2n = array [1..2n] of integer;
...
procedure get_next (var a: array2n; var last: Boolean);
 | {a is replaced by the next sequence if it exists
 |  (and last:=false), otherwise last:=true}
 | var k, i, sum: integer;
begin
 | k:=2*n;
 | {invariant: a[k+1..2n] contains only -1s}
 | while a[k] = -1 do begin k:=k-1; end;
 | {k is maximal among all k such that a[k]=1}
 | while (k>0) and (a[k] = 1) do begin k:=k-1; end;
 | {a[k] is the rightmost -1 preceding some 1;
 |  k=0 if there is no -1 on the left of 1}
 | if k = 0 then begin
 | | last := true;
 | end else begin
 | | last := false;
```

```
    i:=0; sum:=0;
    {sum = a[1]+...+a[i]}
    while i<>k do begin
    | i:=i+1; sum:= sum+a[i];
    end;
    {sum = a[1]+...+a[k], a[k]=-1}
     a[k]:= 1; sum:= sum+2;
    {all a[1]..a[k] have their final values,
      sum=a[1]+...+a[k]}
    while k <> 2*n do begin
    | k:=k+1;
    | if sum > 0 then begin
    | | a[k]:=-1
    | end else begin
    | | a[k]:=1;
    | end;
    | sum:= sum+a[k];
    end;
    {k=2n, sum=a[1]+...a[2n]=0}
  end;
end;
```

■

2.6.2. Find all possible ways to compute the product of n factors. (The order of the factors remains unchanged.) Each multiplication should be indicated by parentheses. For example, for n = 4, the following five expressions should be generated:

```
((ab)c)d, (a(bc))d, (ab)(cd), a((bc)d), a(b(cd)).
```

[Hint. Each order of operations corresponds to a sequence of commands of the stack calculator described on p. 122.] ■

2.6.3. There are 2n points on a circle numbered (along the circle) by the numbers 1..2n. Generate all possible ways to draw n non-intersecting segments having those 2n points as endpoints. ■

2.6.4. Generate all ways to cut a convex polygon with n vertices into triangles using n-2 diagonals. ■

(We will discuss polygon triangulations in the section 8 on dynamic programming on p. 116.)

2.7 Counting

In this section we considered several methods that may be used to generate all the elements of a given finite set. One more approach will be considered below (under

the name of "backtracking") in section 3. But sometimes it is much easier to count all the objects with some property than it is to generate them. The classic example is $\binom{n}{k}$, which is the number of k-element subsets of an n-element set. These numbers form the "Pascal triangle" and can be computed using the identities

$$\begin{aligned} \binom{n}{0} &= \binom{n}{n} = 1 && (n \geq 1) \\ \binom{n}{k} &= \binom{n-1}{k-1} + \binom{n-1}{k} && (n > 1,\ 0 < k < n) \end{aligned}$$

or the formula

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}.$$

(The first method is more efficient when many values of $\binom{n}{k}$ for different n and k are needed.)

Let us give some other examples.

2.7.1. (Number of partitions) Let $P(n)$ be the number of representations of a nonnegative integer n as a sum of positive integer summands (order is insignificant; that is, the representations $1+2$ and $2+1$ are identical). We assume that $P(0) = 1$ (the only representation has no summands at all). Write a program that finds $P(n)$ for a given n.

Solution. One can prove the following (nontrivial) formula for $P(n)$:

$$P(n) = P(n-1) + P(n-2) - P(n-5) - P(n-7) + P(n-12) + P(n-15) + \cdots$$

(terms are grouped in pairs, the signs before the pairs alternate, arguments in q-th pair are $n - (3q^2 - q)/2$ and $n - (3q^2 + q)/2$). We assume $P(k) = 0$ for $k \leq 0$, so the sum is finite.

Even if we did not know this formula, there is a way to compute $P(n)$ that is much more efficient than counting all the partitions one-by-one.

By $R(n, k)$ (defined for $n \geq 0, k \geq 0$) we denote the number of representations of n as a sum of positive integers not exceeding k. (Let $R(0, k)$ be equal to 1 for all $k \geq 0$.) Evidently, $P(n) = R(n, n)$. All the representations of n are classified according to the maximal summand (which is denoted by i in the sequel). The number $R(n, k)$ is the sum over all i in $\{1, \ldots, k\}$ of the number of partitions with elements not exceeding k and maximal element i. The partitions of n into a sum where all terms do not exceed k and maximal term is equal to i are in one-to-one correspondence with the partitions of $n - i$ into terms not exceeding i (assuming that $i \leq k$). Therefore,

$$\begin{aligned} R(n, k) &= \sum_{i=1}^{k} R(n-i, i) \quad \text{for } k \leq n; \\ R(n, k) &= R(n, n) \quad \text{for } k > n. \end{aligned}$$

These equations allows us to construct a table of values of the function R. ■

2.7.2. (Lucky numbers) A sequence of $2n$ digits (each digit is in the $0, \ldots, 9$ range) is called "lucky" if the sum of the first n digits is equal to the sum of the last n digits. Find the number of all lucky sequences of a given length.

Solution. Let us generalize the problem and find the number $T(n, k)$ of sequences of length $2n$ where the difference between the sum of first n digits and the sum of the last n digits is equal to k (where $-9n \leq k \leq 9n$).

We divide all these sequences into classes according to the difference between the first and last digit. If this difference is equal to t, the difference between the remaining sums of $n-1$ digits is $k-t$. Recall that there are $10-|t|$ pairs of decimal digits with difference t, so we get the formula:

$$T(n, k) = \sum_{t=-9}^{9} (10 - |t|) T(n-1, k-t).$$

(Some terms may be missing if $k - t$ is too large.) ■

In some cases, the answer may be given by an explicit formula. For example, this is the case for Catalan numbers

2.7.3. Prove that the Catalan number, that is, the number of sequences of length $2n$ composed of n ones and n minus ones such that each initial segment has a nonnegative sum, is equal to $\binom{2n}{n}/(n+1)$.

[Hint. The Catalan number is the number of polygonal paths going from $(0, 0)$ to $(2n, 0)$ formed by vectors $(1, 1)$ and $(1, -1)$ that do not intersect the half-plane $y < 0$. Therefore, this number is the difference between the number of all polygonal paths of the type described (which is $\binom{2n}{n}$) and the number of paths that intersect the half-plane $y < 0$. All paths of the type described that intersect the half-plane $y < 0$ intersect the line $y = -1$. If we reflect the part of the polygonal path that is on the right of the rightmost intersection point, we get a one-to-one correspondence between the polygonal paths in question and all polygonal paths from $(0, 0)$ to $(2n, -2)$. It remains to check that $\binom{2n}{n} - \binom{2n}{n+1} = \binom{2n}{n}/(n+1)$.] ■

3 Tree traversal (backtracking)

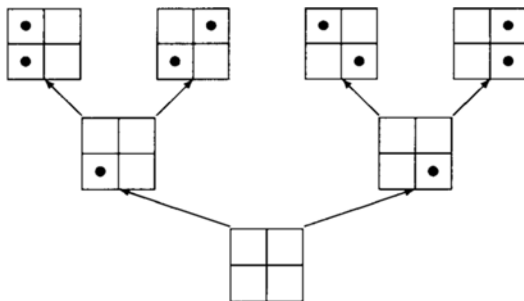
3.1 Queens not attacking each other: position tree traversal

In the preceding section we considered several problems of a similar type: "generate all the elements of some set A". The scheme used to solve these problems was the following one: A linear ordering on A was imposed and a procedure to generate the next element of A (according to that order) was described. Sometimes this scheme cannot be applied directly. In this chapter, we consider another useful approach that allows us to generate all elements of some set. It is called "backtracking" or "tree traversal".

This approach is fairly general; however, we prefer to start with a specific example.

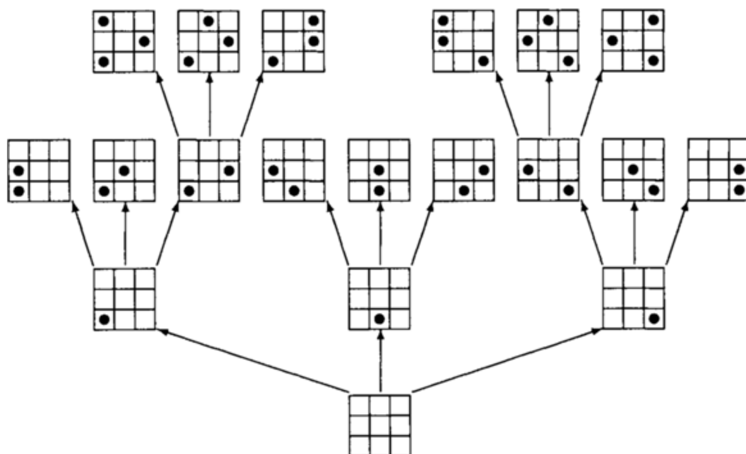
3.1.1. Generate all the positions of n chess queens on an $n \times n$ board such that the queens are not attacking each other.

Solution. Evidently, each of n rows should contain exactly one queen. By k-position we mean a position where k queens occupy k rows (starting from the bottom of the chessboard) containing exactly one queen each. We do not impose any restrictions as of yet and allow positions where some queens are attacking other queens. Arrange all positions into a tree, whose root is the empty position ($k = 0$). Each k-position has exactly n descendants, which have an additional queen in the $(k + 1)$-th row (in one of the columns $1, \ldots, n$). These n descendants are ordered from left to right according to the position of the last (i.e., the uppermost) queen.



We are to select (among the vertices of this tree) those n-positions where queens are not attacking each other. To find them, our program will traverse the positions tree. To avoid unnecessary work, we make use of the following fact: If some tree vertex corresponds to a position where queens are attacking each other, all descendants of this vertex have the same property and therefore may be ignored safely. Therefore, this part of the position tree may be discarded.

Let us give some relevant definitions. A k-position is called "admissible" if *after the k-th queen is removed*, the remaining queens are not attacking each other. Our program will consider only admissible positions.



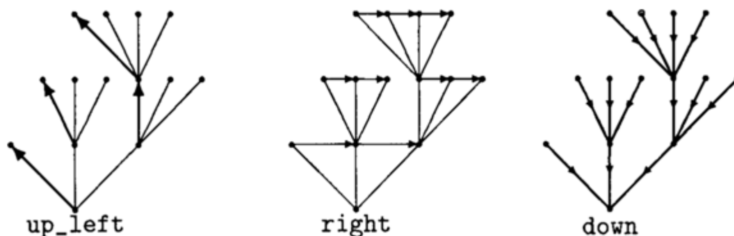
The tree of admissible positions for $n = 3$

Now the queens problem can be divided in two parts: (1) how to traverse all the vertices of a given tree; (2) how to represent the tree of admissible positions for the queens problem using Pascal constructs.

Let us formulate the general problem of visiting all the vertices of a given tree. Imagine there is a robot that can be placed at any vertex of a tree. (Vertices are shown as small circles in our pictures.) The repertoire of the robot consists of the following commands:

- `up_left` ("move along the up-left arrow")
- `right` ("move to the right neighbor")
- `down` ("move down one level")

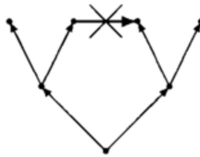
(The pictures below show which movements correspond to these commands.)



Moreover, the robot's repertoire includes tests that check whether each command can be executed:

- is_up;
- is_right;
- is_down;

(the last test returns True everywhere except at the root). Please note that the right command allows a move from the vertex to its "brother" but not to its "cousin" having only a grandfather in common.

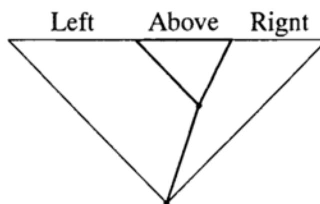


This is **not** a valid right move!

Finally, we assume that the robot is able to perform a command process. Our goal is to process (that is, to execute the command process) all leaves of the tree (a *leaf* is a vertex such that is_up is false, that is, a vertex with no descendants). In our chess problem, process means to check the position and to print it (if it contains n queens not attacking each other).

Remark. Our trees have root at the bottom. Please note that in most computer science books trees are drawn with the root at the *top*. While it seems to be nonintuitive, it is the de facto standard.

The proof of the program below uses the following conventions. Assume that the position of the robot is fixed. Then all the leaves of the tree are divided into three categories: (1) leaves *above* the robot; (2) leaves *on the left* of the robot and (3) leaves *on the right* of the robot. Indeed, the (unique) path from the root to a given leaf (a) may go through the robot's position; (b) may turn to the left before the robot's position, or (c) may go to the right before it. By (LP) we denote the condition "all the leaves on the left of the robot are processed"; by (LAP) we denote the condition "all the leaves on the left of the robot and above it are processed". (In both cases we require that no other leaves are processed.)



We will use the following procedure:

```
procedure go_up_and_process;
| {before: (LP), after: (LAP)}
begin
| {invariant: LP}
| while is_up do begin
| | up_left;
| end
| {LP, current position is a leaf}
| process;
| {LAP}
end;
```

Here is the main program:

```
before: robot is in the root, no leaves are processed
after: robot is in the root, all leaves are processed

{LP}
go_up_and_process;
{invariant: LAP}
while is_down do begin
| if is_right then begin {LAP, is_right}
| | right;
| | {LP}
| | go_up_and_process;
| end else begin
| | {LAP, not is_right, is_down}
| | down;
| end;
end;
{LAP, current position is root => all leaves are processed}
```

Correctness now follows from the properties of the robot's commands. They are presented below in the format:

{precondition} command {postcondition};

The postcondition is guaranteed after execution of the command, assuming that the precondition was true before.

(1) {LP, not `is_up`} `process` {LAP}
(2) {LP} `up_left` {LP}
(3) {`is_right`, LAP} `right` {LP}
(4) {not `is_right`, `is_down`, LAP} `down` {LAP}

3.1.2. Prove that the program shown above terminates for any finite tree.

Solution. The procedure `go_up_and_process` terminates (since the height of the robot cannot increase indefinitely). Assume that the program as a whole does not terminate. Leaves are never processed twice and the number of leaves is finite. Therefore, there is a moment after which leaves are not processed. This is possible only if the robot goes down at each step, but this is a contradiction. (The estimate for the number of operations will be given below.) ∎

3.1.3. Prove that the following program also processes all the leaves of a tree (one time each):

```
var state: (LP, LAP);
state := LP;
while is_down or (state <> LAP) do begin
 | if (state = LP) and is_up then begin
 | | up_left;
 | end else if (state = LP) and not is_up then begin
 | | process; state := LAP;
 | end else if (state = LAP) and is_right then begin
 | |  right; state := LP;
 | end else begin {state = LAP, not is_right, is_down}
 | |  down;
 | end;
end;
```

Solution. The invariant relation: The value stored in the variable `state` is correct, that is,

$$\texttt{state} = \text{LP} \;\Rightarrow\; \text{LP is true}$$
$$\texttt{state} = \text{LAP} \;\Rightarrow\; \text{LAP is true}$$

The proof of termination: the change from LP to LAP is possible only when a vertex is processed. Therefore, if the program does not terminate, the variable `state` achieves its final value and does not change further, which is impossible. ∎

3.1.4. Write a program that traverses the tree and processes all vertices (not only leaves).

Solution. Let x be a vertex. Then all vertices of the tree can be divided into four categories. Indeed, let y be some other vertex. Consider the path from the root to y. Four cases are possible:

(a) this path is a prefix of the path from the root to x (y is *below* x);
(b) this path turns to the left before reaching x (y is *on the left* of x);
(c) this path goes through x (y is *above* x);
(d) this path turns to the right before reaching x (y is *on the right* of x).

In particular, the vertex x belongs to class (c). Now the following conditions are used in our program:

(ULP) all vertices under the current position and on the left of it are processed;

(ULAP) all vertices under the current position, on the left of it, and above it are processed.

Here is the program:

```
procedure go_up_and_process;
| {before: (ULP), after: (ULAP)}
begin
| {invariant: ULP}
| while is_up do begin
| | process;
| | up_left;
| end
| {ULP, the current position is a leaf}
| process;
| {ULAP}
end;
```

The main algorithm:

```
before: robot is in the root, no vertices are processed
after: robot is in the root, all vertices are processed

{ULP}
go_up_and_process;
{invariant: ULAP}
while is_down do begin
| if is_right then begin {ULAP, is_right}
| | right;
| | {ULP}
| | go_up_and_process;
| end else begin
| | {ULP, not is_right, is_down}
| | down;
| end;
end;
{ULAP, robot is in the root => all vertices are processed} ■
```

3.1.5. The program given in the solution of the preceding problem processes any vertex before its descendants. Modify the program in such a way that any vertex will be processed twice, once *before* and once *after* its descendants. (The leaves should be processed once.)

Solution. In the program below, by "Under-Left-Processed" (ULP) we mean "all the vertices under the current position of the robot are processed once; all the vertices on the left are processed completely (that is, leaves are processed once, all other vertices are processed twice: once before and once after their descendants)". By "Under-Left-Above-Processed" (ULAP) we mean "all the vertices under the current position of the robot are processed once; all vertices on the left of and above the current position are processed completely".

Here is the auxiliary procedure:

```
procedure go_up_and_process;
| {before: (ULP), after: (ULAP)}
begin
| {invariant: ULP}
| while is_up do begin
| | process;
| | up_left;
| end
| {ULP, the current position is a leaf}
| process;
| {ULAP}
end;
```

The main program:

```
before: robot is in the root, no vertices are processed
after: robot is in the root, all vertices are processed

{ULP}
go_up_and_process;
{invariant: ULAP}
while is_down do begin
| if is_right then begin {ULAP, is_right}
| | right;
| | {ULP}
| | go_up_and_process;
| end else begin
| | {ULP, not is_right, is_down}
| | down;
| | process;
| end;
end;
{ULAP, robot is in the root =>
   all vertices are processed completely}
```

■

3.1.6. Prove that the number of operations in this program is proportional to the number of vertices. (Therefore, the same is true for the programs given above

that differ from the last one only because some `process` commands have been omitted.)

[Hint. Roughly speaking, each second operation is processing some vertex, and any vertex is processed at most twice.] ∎

Let us return to the queens problem. In this problem, we use only the first and simplest of our tree traversal programs, which processes each leaf once.

We implement all the operations for the case of the positions tree. Each position is represented by a variable `k:0..n` (the number of queens) and an array `c: array[1..n] of 1..n`. Here `c[i]` is the horizontal coordinate of the `i`-th queen (whose vertical coordinate is `i`). If `i` > `k`, the value of `c[i]` is insignificant. Only the admissible positions are included in the tree. (According to our definition, a position is admissible if after the uppermost queen is removed, no queens are attacking each other.)

Now we are ready to present the program that solves queens' problem:

```
program queens;
| const n = ...;
| var
|   k: 0..n;
|   c: array [1..n] of 1..n;
|
| procedure begin_work; {initialize}
| begin
| | k := 0;
| end;
|
| function danger: Boolean;
| | {the uppermost queen is under attack}
| | var b: Boolean; i: integer;
| begin
| | if k <= 1 then begin
| | | danger := false;
| | end else begin
| | | b := false; i := 1;
| | | {b <=> the uppermost queen is under attack of
| | |   some queen with y-coordinate < i}
| | | while i <> k do begin
| | | | b := b or (c[i]=c[k]) {vertical}
| | | |     or (abs(c[i]-c[k])=abs(i-k)); {diagonal}
| | | | i := i+1;
| | | end;
```

```
    danger := b;
  end;
end;

function is_up: Boolean;
begin
  is_up := (k < n) and not danger;
end;

function is_right: Boolean;
begin
  is_right := (k > 0) and (c[k] < n);
end;
{danger: when k=0, the value c[k] is undefined}

function is_down: Boolean;
begin
  is_down := (k > 0);
end;

procedure up_left;
begin {k < n, not danger}
  k := k + 1;
  c [k] := 1;
end;

procedure right;
begin {k > 0,  c[k] < n}
  c [k] := c [k] + 1;
end;

procedure down;
begin {k > 0}
  k := k - 1;
end;

procedure process;
  var i: integer;
begin
  if (k = n) and not danger then begin
    for i := 1 to n do begin
      write ('<', i, ',' , c[i], '> ');
    end;
```

```
      writeln;
    end;
  end;

  procedure go_up_and_process;
  begin
    while is_up do begin
      up_left;
    end
    process;
  end;

begin
  begin_work;
  go_up_and_process;
  while is_down do begin
    if is_right then begin
      right;
      go_up_and_process;
    end else begin
      down;
    end;
  end;
end.
```

■

3.1.7. The program above spends a lot of time inside the procedure `is_up` (to check if the uppermost queen is under attack, we need $O(\mathrm{n})$ operations). Modify the implementation of the positions tree in such a way that all three tests `is_up/right/down` and the corresponding three commands require only $O(1)$ operations (that is, the number of operations for any of them should be limited by a constant that does not depend on n).

Solution. For any vertical and for any diagonal line (there are two types of diagonal lines — ascending and descending ones) there is a Boolean variable that indicates if this line is occupied by some queen (except the uppermost one, which is ignored). Recall that any of those lines may be occupied by at most one queen (because the position is assumed to be admissible). ■

3.2 Backtracking in other problems

3.2.1. Use backtracking in the following problem: An array of n positive integers `a[1]..a[n]` and a positive integer s are given. Determine if s can be represented as a sum of some of the elements of the array a. (Each element may be used at most once.)

Solution. Construct the position tree as follows: The k-*position* is a sequence of k Boolean values that determines which of the elements a[1]..a[k] are used as summands. The position is *admissible* if the sum of the corresponding elements does not exceed s. ■

Remark. This approach is better than exhaustive search (that considers all 2^n subsets). We may also sort the array a in descending order. Also, we can change the definition of an admissible position to exclude positions where the sum of rejected elements is larger than the difference between s and the sum of all accepted elements. However, this does not lead to a fundamental improvement; this problem belongs to the category of the so-called "NP-complete problems". See the book by A. Aho, J. Hopcroft, and J. Ullman [1] and the book by M.R. Garey and D.S. Johnson [5].

This problem is traditionally called "the knapsack problem": A knapsack that is capable of carrying s pounds should be filled completely using only objects of weights a[1]..a[n]. See section 8, p. 118, where a "dynamic programming" algorithm is given whose running time is polynomial in n + s.

3.2.2. Generate all sequences of n digits 0, 1 and 2 that do not contain a substring of type XX. (E.g., the sequence 210102 is prohibited because it contains 1010.) ■

3.2.3. Repeat the previous problem for binary strings of length n that do not contain a substring of type XXX. ■

Another problem of the same category: "Is it possible to compose a given polygon of 'pentamino' blocks?" The crucial component of an effective algorithm for such a problem is a good criterion that can (in some cases) guarantee that a given position cannot be extended to a solution of the problem and therefore may be discarded.

4 Sorting

4.1 Quadratic algorithms

4.1.1. Let `a[1]`, ..., `a[n]` be an array of numbers (say, integers). Construct the array `b[1]`, ..., `b[n]` that contains the same numbers in increasing order: `b[1]` $\le \cdots \le$ `b[n]`.

Remark. The elements `a[1]..a[n]` need not to be distinct. In this case we require that the multiplicity of each number in `b[1]..b[n]` should be equal to its multiplicity in `a[1]..a[n]`.

Solution. It is convenient to consider `a[1]..a[n]` and `b[1]..b[n]` as the initial and final values of some array x. The requirement "a and b contain the same numbers" will be guaranteed if the only operation permitted on x is the exchange of two its elements. (Of course, we are also allowed to read elements of x, too.)

```
k := 0;
{k minimal elements of x are in their places}
while k <> n do begin
 | s := k + 1; t := k + 1;
 | {x[s] is minimal among x[k+1]...x[t] }
 | while t<>n do begin
 | | t := t + 1;
 | | if x[t] < x[s] then begin
 | | | s := t;
 | | end;
 | end;
 | {x[s] is minimal among x[k+1]..x[n] }
 | ... exchange x[s] and x[k+1];
 | k := k + 1;
end;
```

■

4.1.2. Give another sorting algorithm which uses the following invariant relation: "first k elements are sorted" (`x[1]` $\le \cdots \le$ `x[k]`).

Solution. (This algorithm is called *insertion sort.*)

```
k:=1;
{first k elements are sorted}
while k <> n do begin
 | t := k + 1;
 | {k+1-th element moves to the left until it finds its
 |   place; t is its current position}
```

```
   while (t > 1) and (x[t] < x[t-1]) do begin
     ... exchange x[t-1] and x[t];
     t := t - 1;
   end;
end;
```

Remark. Danger: When `(t > 1)` is false, the test `x[t] < x[t-1]` refers to a non-existing value `x[0]`. ■

Both of the above solutions require a number of operations proportional to n^2. There are more efficient algorithms, however, as we shall see.

4.2 Sorting in $n \log n$ operations

4.2.1. Find a sorting algorithm that requires only $O(n \log n)$ operations. (In other words, the number of operations should not exceed $Cn \log n$ for some constant C that does not depend on n.)

We give two solutions.

Solution 1 (merge sort).

Let k be a positive integer, and split the array `x[1]..x[n]` into segments of length k. (The first segment is `x[1]..x[k]`, the next is the segment `x[k+1]..x[2k]`, etc.) The last segment is incomplete if n is not a multiple of k. We say that the array x is k-*sorted* if each of these segments (considered separately) is sorted. Of course, any array is 1-sorted. If an array of length n is k-sorted for k $\geq$ n, it is sorted.

Assume there is a procedure that transforms any k-sorted array into a 2k-sorted array (containing the same elements). Using this procedure, we write down an algorithm as follows:

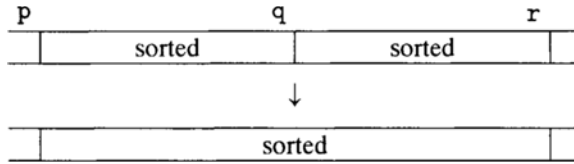
```
k:=1;
{the array x is k-sorted}
while k < n do begin
  ..transform the k-sorted array into a 2k-sorted array;
  k := 2 * k;
end;
```

How do we construct such a procedure? It repeats the following step: two sorted segments of length at most k are merged into one sorted segment. Assume that the procedure

```
merge (p,q,r:  integer)
```

called with p $\leq$ q $\leq$ r merges the sorted segments `x[p+1]..x[q]` and `x[q+1]..x[r]` into a sorted segment `x[p+1]..x[r]` (without changing other

parts of the array x):



The transformation of a k-sorted array into a 2k-sorted array is as follows:

```
t:=0;
{t is a multiple of 2k or t = n, x[1]..x[t] is
 2k-sorted; the rest of x is unchanged}
while t + k < n do begin
 | p := t;
 | q := t+k;
 | ...r := min (t+2*k, n);
 |        {min(a,b) is the minimum of a and b}
 | merge (p,q,r);
 | t := r;
end;
```

The merge procedure uses an auxiliary array as temporary storage for the result. This auxiliary array will be denoted by b. Let p0 and q0 be the indices of the last elements merged; s0 is the index of the last element written to b. At each step, one of the two following actions is performed:

```
b[s0+1]:=x[p0+1];
p0:=p0+1;
s0:=s0+1;
```

or

```
b[s0+1]:=x[q0+1];
q0:=q0+1;
s0:=s0+1;
```

(C fans will enjoy the shorthands `b[++s0]=x[++p0]` and `b[++s0]=x[++q0]` here.)

The first action (where the element is taken from the first segment) may be performed if the following two conditions are fulfilled:

(1) the first segment is not empty (p0 < q); and

(2) the second segment is empty (q0 = r) or its first element is greater than or equal to the first element of the first segment [(q0 < r) and (x[p0+1] ≤ x[q0+1])].

The conditions that make the second action possible are similar. We obtain the following program:

```
p0 := p; q0 := q; s0 := p;
while (p0 <> q) or (q0 <> r) do begin
| if (p0 < q) and ((q0 = r) or ((q0 < r) and
| |                 (x[p0+1] <= x[q0+1]))) then begin
| | b [s0+1] := x [p0+1];
| | p0 := p0+1;
| | s0 := s0+1;
| end else begin
| | {(q0 < r) and ((p0 = q) or ((p0<q) and
| |   (x[p0+1] >= x[q0+1])))}
| | b [s0+1] := x [q0+1];
| | q0 := q0 + 1;
| | s0 := s0 + 1;
| end;
end;
```

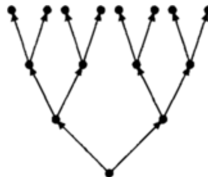
(If both segments are nonempty and have equal first elements, both actions are legal. In this case, the program chooses the first one.)

The only thing left to do is copy the merged array back into x. (Warning: If you decide to perform copying outside the merge procedure, please note that the last segment should be copied even it was not merged with anything.)

The program has a standard deficiency: the array index can be out of bounds when the Boolean expressions are evaluated (if "short-circuit evaluation" is not used).

Solution 2 (heap sort).

Draw a "complete binary tree". The root of this tree is drawn as a small circle at the bottom; two arrows go from the root node to the two nodes above it, two arrows go from each of them, etc.



We say that arrows connect a "father" to its two "sons". Each node has two sons and one father unless it is the the "root" (a node at the bottom) or the "leaf"

(a node at the top). For simplicity, we assume that the length of the array to be sorted is a power of 2 and the elements fill completely some level of the tree. Fill the part of the tree below them using the following rule:

$$\text{father} = \min(\text{son}_1, \text{son}_2)$$

According to this rule, the value at the root of the tree will be the minimal element of the whole array.

Take the minimal element out of the array. To do that, we first locate it. It can be traced going from bottom to top, traversing the son that has the same value as its father. After the minimal element is removed, we replace it by the symbol ∞ and modify its ancestors going from top to bottom. (We assume that $\min(t, \infty) = t$.) Consequently, the root of the tree contains the second minimal element. We locate it, take it out (replacing it by ∞) and modify the tree. This procedure is repeated until all the elements are taken out and the root of the tree is occupied by ∞.

To write down the corresponding program the following agreement is useful. Assume that the vertices of the tree are numbered by 1, 2, ... in such a way that position n has sons 2n and 2n + 1. We do not give the details, because we will write down a more efficient algorithm that does not use any additional memory (except for a fixed number of variables and the array itself). Here it is:

The elements to be sorted are placed at all levels of the tree, not just the upper level. Suppose we want to sort the array `x[1]..x[n]`. The tree has numbers `1..n` as vertices. We assume that `x[i]` is placed at vertex `i`. During execution, the number of vertices in the tree will decrease. The current number of vertices is stored in `k`. Therefore, at any time the array `x[1]..x[n]` is divided into two parts. Its initial segment `x[1]..x[k]` represents a tree. The remaining part `x[k+1]..x[n]` contains the already sorted part of the array; those elements have already reached their final destination.

At each step, the algorithm extracts the maximal element from the tree and puts it into the sorted part (using the position freed when the tree becomes smaller).

Let us specify some terminology. The *vertices* of the tree are numbers from 1 up to the current value of k. Each vertex s may have *sons* 2s and 2s + 1. If both numbers are larger than k, the vertex s has no sons. Such a vertex is called a *leaf*. If 2s = k, the vertex s has exactly one son (2s).

For any s in `1..k`, we consider a *subtree rooted at* s (or s-*subtree*). It contains the vertex s and all its descendants (sons, grandsons, etc. — until we leave the segment `1..k`). The vertex s is called *regular* if the element placed in it is the maximal element of the s-subtree; the s-subtree is called *regular* if all its vertices are regular. (In particular, any leaf is a regular singleton subtree.) Please note that the validity of the statement "s-subtree is regular" depends not only upon s but also upon the current value of k.)

Remark. Modern textbooks (see, e.g., [3]) use terms "child" ("parent", "siebling", etc.) instead of "son" ("father", "brother", etc.) that are used in older textbooks (see, e.g., [1]). When using this new terminology, you should have in

mind that each vertex has only one parent, only one grandparent, etc.

The general structure of the algorithm is as follows:

```
k:= n
.. Make the 1-subtree regular;
{x[1],..,x[k] <= x[k+1] <=..<= x[n]; 1-subtree is regular,
 therefore, x[1] is maximal among x[1]..x[k]}
while k <> 1 do begin
| .. exchange x[1] and x[k];
| k := k - 1;
| {x[1]..x[k-1] <= x[k] <=...<= x[n]; 1-subtree is
|   regular everywhere except the root (may be)}
| .. restore the regularity of 1-subtree
end;
```

As a tool, we use a procedure that restores the regularity of the subtree which is regular everywhere except its root. Here it is:

```
{s-subtree is regular everywhere except perhaps its root}
t := s;
{s-subtree is regular everywhere except perhaps t}
while ((2*t+1 <= k) and (x[2*t+1] > x[t])) or
|       ((2*t <= k) and (x[2*t] > x[t])) do begin
| if (2*t+1 <= k) and (x[2*t+1] >= x[2*t]) then begin
| | ... exchange x[t] and x[2*t+1];
| | t := 2*t + 1;
| end else begin
| | ... exchange x[t] and x[2*t];
| | t := 2*t;
| end;
end;
```

Let us look closely at this procedure to check its correctness. Assume that all vertices of the s-subtree are regular except perhaps the vertex t. Consider the sons of t. They are regular and therefore contain maximal elements of their subtrees. Therefore, we have only three possibilities for the maximal element of the t-subtree, namely, the vertex t and its sons. If the vertex t contains the maximal element, this vertex is regular, and we are done. The while-construct can be rewritten as follows:

```
while the maximal element is not t but one of its sons
|   do begin
| if it is the right son then begin
| | exchange t and its right son; t:= right son;
| end else begin {the maximal element is the left son of t}
```

```
| | exchange t and its left son; t:= left son;
| end
end
```

After the exchange, the vertex t becomes regular (since it contains the maximal element of the t-subtree). The son that does not take part in the exchange is still regular. The other son may become non-regular. Any other vertex u of the s-subtree remains regular because the value placed in u is unchanged, as well as the u-subtree (though elements of the subtree may be permuted).

The same procedure may be used at the first stage of our algorithm to make the 1-subtree regular:

```
k := n;
u := n;
{s-subtrees are regular for all s>u}
while u<>0 do begin
| {u-subtree is regular everywhere
|             except the root (may be)}
| .. restore the regularity of u-subtree;
| u:=u-1;
end;
```

This algorithm is called *heap sort.*

Now we are ready to write down the heapsort program in Pascal. We assume that n is a constant and x is a variable of type arr = array [1..n] of integer).

```
procedure sort (var x: arr);
| var u, k: integer;
| procedure exchange(i, j: integer);
| | var tmp: integer;
| | begin
| | tmp  := x[i];
| | x[i] := x[j];
| | x[j] := tmp;
| end;
| procedure restore (s: integer);
| | var t: integer;
| | begin
| | t:=s;
| | while ((2*t+1 <= k) and (x[2*t+1] > x[t])) or
| | |     ((2*t <= k) and (x[2*t] > x[t])) do begin
| | | if (2*t+1 <= k) and (x[2*t+1] >= x[2*t]) then begin
| | | | exchange (t, 2*t+1);
| | | | t := 2*t+1;
| | | end else begin
```

```
            exchange (t, 2*t);
            t := 2*t;
          end;
        end;
      end;
begin
  k:=n;
  u:=n;
  while u <> 0 do begin
    restore (u);
    u := u - 1;
  end;
  while k <> 1 do begin
    exchange (1, k);
    k := k - 1;
    restore (1);
  end;
end;
```

Several remarks:

- The method used in the heapsort algorithm has other applications. One example is the priority queue implementation, see section 6.4, p. 96.
- The advantage of the merge sorting algorithm is that it does not require the entire array to be placed into RAM. We can sort portions of the array that fit into available RAM (say, using heapsort) and then merge the files obtained.
- Another important sorting algorithm ("Hoare quicksort") uses the following approach. To sort an array, choose a random element b and split the array into three parts: elements smaller than b, equal to b and greater than b. (This problem is discussed in section 1.) Now it remains to sort the first and the third parts, which can be done recursively using the same method. The number of steps of this algorithm is a random variable. One can prove that its expectation does not exceed $Cn \log n$. It is one of the fastest algorithms in practice. (We shall discuss its recursive and non-recursive implementations later.)
- Finally, let us mention that sorting in $Cn \log n$ operations may be performed using the technique of balanced trees (see section 12), but the programs are rather complicated and the constant C is large enough to make this method impractical. ∎

4.3 Applications of sorting

4.3.1. Find the number of different elements in an integer array. The number of

operations should be of order $n \log n$. (This problem was already mentioned in section 1.)

Solution. Sort the array and then count the different elements going from left to right. ■

4.3.2. Suppose that n closed intervals [a[i], b[i]] on the real line are given (i = 1..n). Find the maximal k such that there exists a point covered by k intervals (the "maximal thickness" of covering). The number of operations should be of order n log n.

[Hint. Sort all the left and right endpoints of the intervals together. While sorting, assume that the left endpoint precedes the right endpoint located at the same point of the real line. Then move from left to right counting the number of layers. When we cross the left endpoint, increase the number of layers by 1; when we cross the right endpoint, decrease the number of layers by 1. Please note that two adjacent intervals are processed correctly; that is, the left endpoint precedes the right endpoint according to our convention.] ■

4.3.3. Assume that n points in the plane are given. Find a polygonal arc with n − 1 sides whose vertices are the given points, and whose sides do not intersect. (Adjacent sides may form a 180° angle.) The number of operations should be of order n log n.

Solution. Sort all the points with respect to the x-coordinate; when x-coordinates are equal, take the y-coordinate into account, then connect all vertices by line segments (in that order). ■

4.3.4. The same problem for a polygon: for a given set of points in the plane find a polygon having these points as vertices.

Solution. Take the leftmost point (the point whose x-coordinate is minimal). Consider all the rays starting from this point and going through all other points. Sort these rays according to their slopes, and sort the points that are on the same ray according to their distance from the initial point. Do this for all rays except the rays with maximal and minimal slopes. The polygon goes from the initial point along the ray with minimal slope, then visits all the points in the order chosen, returning via the ray with maximal slope. ■

4.3.5. Assume that n points in the plane are given. Find their *convex hull*, that is, the minimal convex polygon that contains all the points. (A rubber band put on the nails is the convex hull of the nails inside it.) The number of operations should be of order n log n.

[Hint. Order all the points according to one of the orderings mentioned in the two preceding problems. Then construct the convex hull considering the points one by one. (To maintain information about the current convex hull, it is useful to use a deque; see section 6, page 87. It is not necessary, however, when the points are ordered according to their slopes.)] ■

4.4 Lower bound for the number of comparisons

Suppose we have n objects (say, stones) of different weights and a balance that can be used to find which of any two given stones is heavier. In programming terms, we have access to a Boolean function `heavier(i,j:1..n)`. Our goal is to sort all the stones in increasing order using the balance as few times as possible (making the fewest calls to the function `heavier`).

Of course, the number of comparisons depends not only on the algorithm we choose but also on the initial order of the stones. By *complexity* of the algorithm we mean the number of comparisons *in the worst case.*

4.4.1. Prove that any sorting algorithm for n stones has complexity at least $\log_2 n!$. (Here $n! = 1 \cdot 2 \cdots n$.)

Solution. Assume that we have an algorithm of complexity d, that is, an algorithm that makes at most d comparisons (in all cases). For any of $n!$ possible orderings of the stones let us write down the results of all the comparisons (calls to the function `heavier`). The protocol is a binary string of length at most d. If necessary, pad it with trailing zeros to get a string of length d. Now we have $n!$ binary strings of length d (corresponding to $n!$ permutations of input stones). All those strings are different, otherwise our algorithm gives the same answer for two different orderings (and at least one of the answers is incorrect). Therefore, $2^d \geq n!$.

Another way to say the same thing is to consider the tree of possibilities that appear during the execution of the algorithm. Indeed, a tree of height d has no more than 2^d leaves.

This argument shows that any algorithm that relies upon comparisons and exchange operations only, requires at least $\log_2 n!$ comparisons. A simple calculation shows that $\log_2 n! \geq \log_2 (n/2)^{n/2}$ (we omit the first half of the factors and replace the remaining factors by $n/2$). Now $\log_2 (n/2)^{n/2} = (n/2)(\log_2 n - 1) \geq Cn \log_2 n$ for some C. Therefore, our sorting algorithms are close to optimal (improvement is limited to a constant factor). ∎

However, a sorting algorithm that uses not only comparisons (but also the internal structure of the sorted objects) may be faster. Here is an example:

4.4.2. An integer array `a[1]..a[n]` is given; all the integers are non-negative and do not exceed `m`. Sort this array using no more than $C(\mathtt{m} + \mathtt{n})$ operations (C is a constant that does not depend on `m` and `n`).

Solution. For each number in `0..m`, count how many times it appears in the array. (These data can be collected during one pass through the array.) Then erase the array and write down its elements in increasing order using the information about the multiplicity of each number. ∎

Note that this algorithm does not exchange elements of the array but puts "fresh" sorted numbers into the array.

There exists another sorting method that sequentially performs several "partial sorts" with respect to fixed bits. Let us start with the following problem:

4.4.3. Rearrange the array `a[1]..a[n]` in such a way that all even elements precede all odd elements (not changing the order inside each of the two groups).

Solution. Copy all the even elements into an auxiliary array. Then append all the odd elements to that array and copy all elements back. ■

4.4.4. An array of n integers in the range $0, \ldots, 2^k - 1$ is given. Each integer is written as a binary string of length k. Using the tests "i-th bit is 0" and "i-th bit is 1" instead of comparisons, sort all the integers. The number of operations should be of order nk.

Solution. Sort all the numbers with respect to the last bit as in the preceding problem. Then sort them with respect to the bit next the last one, etc. After k stages, the numbers will be sorted. Indeed, by induction over i, we can easily prove the following statement: "after i steps, any two numbers that differ only in the last i bits, are in the correct order". (Or prove by induction the following statement: "after i steps the suffixes of length i are in the right order".) ■

A similar algorithm can be constructed using m-ary notation instead of binary. The following problem is useful in this regard.

4.4.5. Assume that an array of n elements and a function f defined on those elements are given. Assume that the possible values of f are $1, \ldots, m$. Rearrange the array in such a way that the values of f are in nondecreasing order and the elements with equal values of f are in the same order as before. The number of operations should be of order $m + n$.

[Hint. Create m lists of total length n using "pointer implementation" (see section 6, p. 83). Put an element into the i-th list if the value of f is equal to i. Another possibility: count how many elements have a given value of f (for all m possible values); thereafter, we know where the elements of any given f-value should be placed in the array.] ■

4.5 Problems related to sorting

4.5.1. Find the minimal complexity (= the number of comparisons in the worst case) for an algorithm that finds the stone with minimal weight.

Solution. The obvious algorithm with the invariant relation "the minimal among the first i stones is found" requires $n - 1$ comparisons. No algorithm can have smaller complexity. This is a corollary of a stronger statement, see the next problem. ■

4.5.2. An expert wants to convince a jury that a given stone has minimal weight among n given stones. The expert wants to do this using a balance less than $n - 1$

times. Prove that this is impossible. (The expert knows in advance the weights of all the stones; the jury does not.)

Solution. Consider stones as vertices of a graph. For any comparison, draw an edge between the corresponding pair of vertices. After $n-1$ measurements, the graph is not connected; it has more than 1 connected component, because each edge decreases the number of connected components by at most 1. Therefore, the jury knows nothing about the relation between weights of stones from different components and may assume that the stone with minimal weight is in any of the components. ∎

Let us stress the difference between this problem and the preceding one. In this problem, we have to show that $n-2$ comparisons are not enough to prove that a given stone has minimal weight even if we know the answer in advance — not to mention finding the answer. (The difference between the two settings is clear in the case of sorting. When a correct answer is known, it can be confirmed by $n-1$ comparisons (each stone should be compared with the next one), which is many fewer comparisons than what was needed to find the answer.)

4.5.3. Prove that it is possible to find the stones with minimal and maximal weights among $2n+1$ stones using only $3n$ comparisons.

[Hint. Divide all the stones into n pairs (one stone remains) and compare stones within each pair.] ∎

4.5.4. Assume that n stones of different weights are given. Let k be a number in the range $1, \ldots, k$. Find the k-th stone (in the order of increasing weights) making not more than Cn comparisons, where C is some constant that does not depend on k or n.

Remark. Using sorting, we can do this in $Cn \log n$ steps. See the section 7, p. 112, where a hint for this (rather difficult) problem is given. ∎

The following problem has a surprisingly simple solution.

4.5.5. There are n stones that look identical, but in fact, some of them have different weights. There is a device that can be applied to two stones and tells whether they are different or not (but it does not say which one is heavier). It is known in advance that most of the stones (more than 50%) are identical. Find one of those identical stones making no more than n comparisons. (Beware: it is possible that two stones are identical but do not belong to the majority of identical stones.)

[Hint. If two stones are different, they may be discarded, because one of them does not belong to the majority and the majority remains.]

Solution. The program processes the stones one-by-one and keeps the number of the stones processed in a variable `i`. (Assume that stones are numbered `1..n`). The program remembers the number of the "current candidate" `c` and its "multiplicity" `k`. The names are explained by the following invariant relation `(I)`:

If we add k copies of the c-th stone to the unprocessed stones (i+1..n), the majority stones in the initial array will remain the majority in the new array.

Here is the program:

```
k:=0; i:=0;
{(I)}
while i<>n do begin
| if k=0 then begin
| | k:=1; c:=i+1; i:=i+1;
| end else if (i+1-th stone is the same as c-th)
| |    then begin
| | i:=i+1; k:=k+1;
| |  {replace a physical stone by a virtual stone}
| end else begin
| | i:=i+1; k:=k-1;
| | {discard one physical and one virtual stone}
| end;
end;
c-th stone is the answer
```

Remark. All three branches of the if-block include the statement i:=i+1, so it can be moved to the next level. ∎

Let us mention that this program finds the most frequent stone only if it forms the majority (more than 50%).

This problem can be found as problem 4-7 on page 75 of the book [3] in a completely different setting ("VLSI chip testing") where a recursive solution is sketched.

At first glance, the following problem seems unrelated to sorting.

4.5.6. There is a square array a[1..n,1..n] filled by 0s and 1s. It is known that for some i the i-th row contains only 0s and at the same time the i-th column contains only 1s (except the main diagonal entry a[i,i], which may be arbitrary). Find this i (which is unique). The number of operations should be of order n. (Please note that the number of operations should be much smaller than the total number of elements in a.)

[Hint. Assume we get the Boolean value a[i][j] when comparing two virtual stones with numbers i and j. Recall that the maximal element among n elements may be found using n-1 comparisons. Take into account that the array may not be "transitive"; however, after two numbers are compared, one of them may be discarded.] ∎

5 Finite-state algorithms in text processing

5.1 Compound symbols, comments, etc.

5.1.1. Throughout a program text the operation x^y was denoted by `x**y`. It was decided that notation should be changed to `x^y`. How do we do that? The input text is read character-by-character; the output text should be printed in the same manner.

Solution. At any time, the program is in one of two states: "basic" state and "after" state (after an asterisk):

State	Next symbol	New state	Action
basic	$*$	after	none
basic	$x \neq *$	basic	print x
after	$*$	basic	print ^
after	$x \neq *$	basic	print $*, x$

If after reading all the text, the program is in the "after" state, it should print an asterisk (and quit). ∎

Remark. Our program replaces `***` by `^*` (and not by `*^`). We did not specify the behavior of the program in this case, assuming (as is often done) that some "reasonable" behavior is expected. In this example, the simplest way to describe the required behavior is to list the states and the corresponding actions.

Please note also that if two asterisks appear in other parts of the program (say, comments), they will be also replaced.

5.1.2. Write a program that deletes all occurrences of the substring `abc`. ∎

5.1.3. In Pascal, comments are surrounded by curly braces like this:

```
begin {here a block begins}
i:=i+1; {increase i by one}
```

Write a program that removes all comments and puts a space character in the place of a removed comment. (According to Pascal rules, `1{one}2` is equivalent to `1 2`, not `12`).

Solution. The program has two states: a "basic" state and an "inside" state (inside a comment).

State	Next symbol	New state	Action
basic	{	inside	none
basic	$x \neq \{$	basic	print x
inside	}	basic	print a space
inside	$x \neq \}$	inside	none

∎

This program cannot deal with nested comments: the string

```
{{comment inside a} comment}
```

is transformed into

```
  comment}
```

(the latter string starts with two spaces). It is impossible to deal with nested comments using a finite automaton (a program that has finite number of internal states); roughly speaking, we have to remember the number of opening braces and a finite automaton cannot do that.

Please note that after reading all the text, the program may still be in the "inside" state. Most probably, we would like to consider this as an error.

5.1.4. Pascal programs also contain quoted strings. If a curly brace appears inside a string, it does not mean the start of a comment. Similarly, a quote symbol inside a comment does not signify a string. How do we modify the above program to take this into account?

[Hint. We need three states: "basic", "inside a comment", "inside a string".] ■

(Note that actual Pascal conventions are more complicated allowing a quote appear inside a quoted string, etc.)

5.1.5. One more feature that exists in many Pascal implementations is a comment of the type

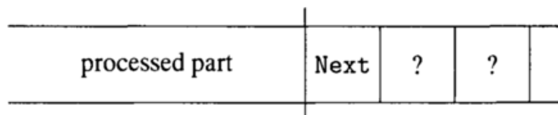
```
i := i+1;   (* here i is increased by 1 *)
```

A closing comment symbol must be paired with an opening comment symbol of the same type (e.g., `{...*)` is not permitted). How do we deal with these types of comments? ■

5.2 Numbers input

Assume that a program scans a decimal representation of some number from left to right. The program should "read" this number, that is, put its value into a variable of type `real`. Also, the program should complain if the input is incorrect.

Let us specify the problem in more detail. Assume that the input string is divided into two parts: the part that is already processed and the remaining part. We have access to a function `Next:char`, which returns the first symbol of the unprocessed part. Also, we have access to a procedure `Move`, which moves the first unprocessed symbol to the processed part.



By a decimal number we mean a character string of the type

⟨0 or more spaces⟩ ⟨1 or more digits⟩

or

⟨0 or more spaces⟩ ⟨1 or more digits⟩ . ⟨1 or more digits⟩

Please note that this definition does not allow the following strings:

`1.`   `.1`   `1.␣1`   `-1.1`

Let us now state the problem:

5.2.1. Read the maximal prefix of the input string that may be a prefix of a decimal number. Determine whether this prefix is a decimal number or not.

Solution. Let us write a program using Pascal's "enumeration type" for clarity. (The variable `state` may have one of the listed values.)

```
var state:
 (Accept, Error, Initial, IntPart, DecPoint, FracPart);

state := Initial;
while (state <> Accept) or (state <> Error) do begin
 | if state = Initial then begin
 | | if Next = ' ' then begin
 | | | state := Initial; Move;
 | | end else if Digit(Next) then begin
 | | | state := IntPart;
 | | |   {after the start of the integer part}
 | | | Move;
 | | end else begin
 | | | state := Error;
 | | end;
 | end else if state = IntPart then begin
 | | if Digit (Next) then begin
 | | | state := IntPart; Move;
 | | end else if Next = '.' then begin
 | | | state := DecPoint; {after the decimal point}
 | | | Move;
 | | end else begin
 | | | state := Accept;
 | | end;
```

```
  end else if state = DecPoint then begin
    if Digit (Next) then begin
      state := FracPart; Move;
    end else begin
      state := Error; {at least one digit is needed}
    end;
  end else if state = FracPart then begin
    if Digit (Next) then begin
      state := FracPart; Move;
    end else begin
      state := Accept;
    end;
  end else if
    {this cannot happen}
  end;
end;
```

Please note that the assignments `state := Accept` and `state := Error` are not accompanied by a call to procedure `Move`, so the symbol after the end of the decimal number is left unprocessed. ■

This program does not store the value of the number.

5.2.2. Add the following requirement to the preceding program: If a processed part is a decimal number, its value should be placed into the variable `val:real`.

Solution. While reading the fractional part, we use the variable `scale` which is a factor for the digit to come (0.1, 0.01 etc.).

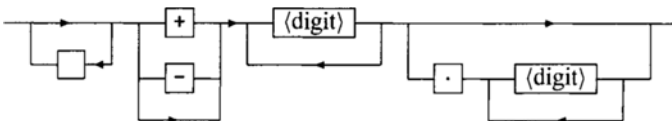
```
state := Initial; val:= 0;
while (state <> Accept) or (state <> Error) do begin
  if state = Initial then begin
    if Next = ' ' then begin
      state := Initial; Move;
    end else if Digit(Next) then begin
      state := IntPart;
        {after the start of the integer part}
      val := DigitVal(Next);
      Move;
    end else begin
      state := Error;
    end;
  end else if state = IntPart then begin
```

```
      if Digit (Next) then begin
        state := IntPart; val := 10*val + DigitVal(Next);
        Move;
      end else if Next = '.' then begin
        state := DecPoint; {after the decimal point}
        scale := 0.1;
        Move;
      end else begin
        state := Accept;
      end;
    end else if state = DecPoint then begin
      if Digit (Next) then begin
        state := FracPart;
        val := val + DigitVal(Next)*scale;
        scale := scale/10;
        Move;
      end else begin
        state := Error; {at least one digit is needed}
      end;
    end else if state = FracPart then begin
      if Digit (Next) then begin
        state := FracPart;
        val := val + DigitVal(Next)*scale;
        scale := scale/10;
        Move;
      end else begin
        state := Accept;
      end;
    end else if
      {this cannot happen}
    end;
  end;
```

■

5.2.3. Repeat the previous problem if the number may be optionally preceded by - or +. ■

The format of numbers in this problem can be represented as follows:



5.2.4. The same problem if the number may be followed by an integer exponent, as in `254E-4` (= 0.0254) or `0.123E+9` (= 123 000 000). Draw the corresponding picture. ∎

5.2.5. What changes in the above program above are necessary to allow empty integer or fractional parts like in `1.`, `.1` or even `.` (the latter number is considered to be equal to zero)? ∎

We return to finite-state algorithms (also called *finite automata*) in section 10.

6 Data types

6.1 Stacks

Let T be some type. Consider the data type "stack of elements of type T". Values of that type are sequences of values of type T.

Operations:

- Make_empty (`var s`: stack of elements of type T)
- Add (`t:T`; `var s`: stack of elements of type T)
- Take (`var t:T`; `var s`: stack of elements of type T)
- Is_empty (`s`: stack of elements of type T): `Boolean`
- Top (`s`: stack of elements of type T): `T`

(We use Pascal notation even though the stack type does not exist in Pascal.) The procedure "Make_empty" makes the stack empty. The procedure "Add" adds `t` to the end of the sequence `s` (i.e., the top of the stack). The procedure "Take" is applicable if the sequence `s` is not empty; it takes the last element away from `s` and puts it into the variable `t`. The expression "Is_empty(`s`)" is true when the sequence `s` is empty. The expression "Top(`s`)" is defined when `s` is not empty; its value is the last element of the sequence `s`.

Usually the operations "Add" and "Take" are called "Push" and "Pop" respectively; we use the names "Add" and "Take" to stress the similarity between stacks and queues (section 6.2).

Our goal is to show how stacks can be implemented in Pascal and what they can be used for.

Stack: array implementation

Assume that the number of elements in a stack never exceeds some constant `n`. Then the stack can be implemented using two variables:

```
Content: array [1..n] of T;
Length: integer;
```

We assume that our stack contains elements

```
Content [1],...,Content [Length]
```

- To make the stack empty, it is enough to perform the assignment

```
Length := 0
```

- Adding element t:

```
{Length < n}
Length := Length + 1;
Content [Length] :=t;
```

- Taking element into a variable t:

```
{Length > 0}
t := Content [Length];
Length := Length - 1;
```

- The stack is empty when Length = 0.

- The top of the stack is Content [Length], assuming Length > 0.

Therefore, a variable of type stack can be replaced in a Pascal program by two variables Content and Length. We can also define the type stack as follows:

```
const n = ...
type
  stack = record
  | Content: array [1..n] of T;
  | Length: integer;
  end;
```

We then define procedures dealing with stack variables. For example, we write

```
procedure Add (t: T; var s: stack);
begin
| {s.Length < n}
| s.Length := s.Length + 1;
| s.Content [s.Length] := t;
end;
```

The use of stacks

In the following problem, we consider sequences of opening and closing parentheses ( ) and square brackets [ ]. Some sequences are considered to be "correct". Namely, a sequence is correct if its correctness follows from the following rules:

- the empty sequence is correct;

- if A and B are correct, then AB is correct;

- if A is correct, then $[A]$ and (A) are correct.

Example. The sequences (), [[]], [()[]()][] are correct, while the sequences],)(, (], ([)] are not.

6.1.1. Check the correctness of a given sequence. The number of operations should be proportional to the length of the sequence. We assume that the sequence terms are encoded as follows:

(	1
[	2
)	−1
]	−2

Solution. Let a[1]..a[n] be a sequence of length n. Consider a stack whose elements are opening parentheses and brackets (i.e., the numbers 1 and 2).

Initially the stack is empty. We scan the sequence from left to right. When an opening parenthesis or bracket is found, we put it onto the stack. When a closing parenthesis or bracket is found, we check if the top of the stack is a complementary parenthesis or bracket. If not, we stop and reject the input. If so, we take the top of the stack away. The sequence is correct if it is not rejected while reading the input and if the stack is empty after the input is exhausted.

```
Make_empty (s);
i := 0; Error_found := false;
{i symbols are processed}
while (i < n) and not Error_found do begin
  i := i + 1;
  if (a[i] = 1) or (a[i] = 2) then begin
    Add (a[i], s);
  end else begin  {a[i] is either -1 or -2}
    if Is_empty (s) then begin
      Error_found := true;
    end else begin
      Take (t, s);
      Error_found := (t <> - a[i]);
    end;
  end;
end;
Correct := (not Error_found) and Is_empty (s);
```

Let us prove the correctness of our program.

(1) If the input sequence is correct, our program accepts it. This can be proved by induction. We need to prove that **(a)** our program accepts the empty sequence; **(b)** that it accepts the sequence AB (assuming that A and B are accepted); and **(c)** it accepts the sequences [A] and (A) assuming that A is accepted.

An empty sequence is accepted for obvious reasons. (Note: In this case, the while-loop is not executed.)

For AB our program works exactly as for A until all symbols of A are processed; therefore, the stack is empty at that moment. Then program processes B (and finishes with the empty stack, because B is accepted by assumption).

For [A] the program begins by putting an opening bracket onto the stack. Then the program processes A, the only difference is that there is an additional bracket at the bottom of the stack. When A is finished, the stack is empty except for the left bracket; at the next step, the stack becomes empty. A similar thing happens for (A).

(2) Let us now prove that if the program accepts some sequence, then the sequence is correct. This is proved by induction over the length of the sequence. Consider the length of the stack during execution. If the stack becomes empty at some point, then the sequence can be divided into two parts and each of the parts is accepted by the program. Therefore, each part is correct (inductive hypothesis) and the sequence as a whole is correct (definition of correctness). Now assume that the stack never becomes empty (except for the beginning and the end). This means that the bracket or parenthesis put onto the stack at the first step is removed at the last step. Therefore, the first and last symbols in our sequence are complementary, the sequence is of type (A) or [A], and the behavior of the program differs from its behavior on A only by the additional parenthesis or bracket at the bottom of the stack. Therefore, by the induction hypothesis, A is correct and the sequence is correct by definition. ■

6.1.2. The program can be simplified if the sequence contains only parentheses and no brackets. How?

Solution. In this case, the stack is reduced to its length, and we arrive at the following statement: A sequence of "(" and ")" is correct if and only if each prefix contains no more symbols ")" than "(", and the entire sequence has equal numbers of both symbols. ■

6.1.3. Implement two stacks using one array. The total number of elements in both stacks is limited by the array length; all stack operations should run in $O(1)$ time (i.e., running time should be bounded by a constant).

Solution. The stacks grow from the beginning and end of the array `Content[1..n]` in opposite directions. One stack occupies places

`Content[1]..Content[Length1]`,

while the other stack occupies places

`Content[n]..Content[n-Length2+1]`

(both stacks are listed from bottom to top). Stacks do not overlap if their total length does not exceed `n`. ■

6.1.4. Implement `k` stacks of elements of type `T` with a total of at most `n` elements using arrays with total length $C(\mathrm{n}+\mathrm{k})$. Each stack operation (except

initialization, which makes all stacks empty) should be performed in constant time (not depending on n and k). (In other words, the implementation should require space $O(\mathtt{n}+\mathtt{k})$ and run in time $O(1)$ for each operation.)

Solution. We use a "pointer implementation" of stacks. It uses three arrays:

```
Content: array [1..n] of T;
Next: array [1..n] of 0..n;
Top: array [1..k] of 0..n;
```

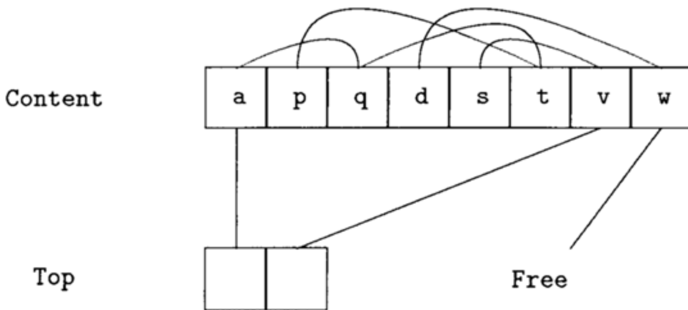
The array `Content` can be thought of as n cells numbered from 1 to n. Each of the cells is capable of holding one element of type T. The array `Next` is represented by arrows between elements: there is an arrow from i to j if `Next[i]=j`. (If `Next[i]=0`, there are no arrows from i.) The content of the s-th stack (s ∈ {1..k}) is determined as follows: the top element is `Content[Top[s]]` and other elements are read by following the arrow links (if they exist). Moreover,

(s-th stack is empty) ⇔ `Top[s] = 0`.

The "arrow trajectories" starting from

`Top[1], ..., Top[k]`

(those not equal to 0) are disjoint. Besides these, we need one more trajectory that traverses all locations that are currently free. Its starting point is stored in the variable `Free: 0..n` (where `Free` = 0 means that all places are occupied). Here is an example:



Content	a	p	q	d	s	t	v	w
Next	3	0	6	0	0	2	5	4
Top	1	7						

`Free` = 8

Stacks: the first one contains p, t, q, a (a is on the top); the second one contains s, v (v is on the top).

```
procedure Initialize; {Make all stacks empty}
| var i: integer;
begin
| for i := 1 to k do begin
| | Top [i]:=0;
| end;
| for i := 1 to n-1 do begin
| | Next [i] := i+1;
| end;
| Next [n] := 0;
| Free:=1;
end;

function Is_free: Boolean;
begin
| Is_free := (Free <> 0);
end;

procedure Add (t: T; s: integer);
| {Add t to the s-th stack}
| var i: 1..n;
begin
| {Is_free}
| i := Free;
| Free := Next [i];
| Next [i] := Top [s];
| Top [s] :=i;
| Content [i] := t;
end;

function Is_empty (s: integer): Boolean;
| {s-th stack is empty}
begin
| Is_empty := (Top [s] = 0);
end;

procedure Take (var t: T; s: integer);
| {Take the top of the s-th stack into t}
| var i: 1..n;
| begin
| {not Is_empty (s)}
| i := Top [s];
| t := Content [i];
| Top [s] := Next [i];
| Next [i] := Free;
```

```
| Free := i;
end;

function Top_element (s: integer): T
| {Top of the s-th stack}
begin
| Top_element := Content[Top[s]];
end;
```

■

6.2 Queues

Values of type "queue of elements of type T" are sequences values of type T. The same is true for stacks, but the difference is that of queue elements are added to the beginning of a sequence and are taken from the end of it. Therefore, an element that arrived first to a queue will be the first element taken from it. Hence the name First In First Out (FIFO), which is used for queues. The method used for stacks is called Last In First Out (LIFO).

Operations on queues:

- Make_empty (`var x`: queue of elements of type T);
- Add (`t:T`, `var x`: queue of elements of type T);
- Take (`var t:T`, `var x`: queue of elements of type T);
- Is_empty (`x`: queue of elements of type T): `Boolean`;
- First_element (`x`: queue of elements of type T): T.

The procedure "Add" adds the specified element to the end of the queue. The procedure "Take" is applicable if the queue is not empty; it puts the first element of the queue into a variable `t`, removing it from the queue. (The first element is the longest-waiting element.)

The procedures "Add" and "Take" are often called "Enqueue" and "Dequeue".

Queue: array implementation

6.2.1. Implement a queue of limited size in such a way that all operations run in $O(1)$ time (that is, in time not exceeding some constant, which does not depend on length of the queue).

Solution. Assume that queue elements are stored as consecutive elements in an array. The queue grows to the right and is taken from the left. A growing queue may reach the end of the array, so we assume the array is "wrapped around" in circular fashion.

Our implementation uses an array

```
Content: array [0..n-1] of T
```

and variables

```
First: 0..n-1
Length : 0..n
```

The queue is formed by elements

```
Content [First], Content [First + 1],...,
        Content [First+Length-1]
```

where addition is performed modulo n. (Warning: If you instead use variables `First` and `Last` whose values are residues modulo n, be careful not to mix the empty queue with the queue containing n elements.)

The queue operations are implemented as follows:

Make_empty:

```
Length := 0;
First := 0;
```

Add an element t:

```
{Length < n}
Content [(First + Length) mod n] := t;
Length := Length + 1;
```

Take element into variable t:

```
{Length > 0}
t := Content [First];
First := (First + 1) mod n;
Length := Length - 1;
```

Is_empty:

```
Length = 0
```

First_element:

```
Content [First]
```

■

6.2.2. (Communicated by A.G. Kushnirenko) Implement a queue using two stacks (and a fixed number of variables of type T). For n queue operations starting with an empty queue, the implementation should perform not more than Cn stack operations.

Solution. We maintain the following invariant relation: *stacks whose bottoms are put together, form the queue.* (In other words, listing all elements of one stack

from top to bottom and then of the other stack from bottom to top, we list all the queue elements in the proper order.) To add an element to the queue, it is enough to add it to one of the stacks. To check if the queue is empty, we must check that both stacks are empty. When taking the first element from the queue, we should distinguish between two cases. If the stack that contains the first element is not empty, there is no problem. If that stack is empty, the required element is buried under all the elements of the second stack. In this case, we move all the elements one-by-one onto the first stack (their ordering is reversed) and return to the first case.

The number of operations for this step is not limited by any constant. However, the requirement posed in the problem is still met. Indeed, any element of the queue can participate in such a process at most once during its presence in the queue. ∎

6.2.3. Deque (double-ended queue) is a structure that combines the properties of a queue and a stack: we can add and remove elements from both ends of a deque. Implement a deque using an array in such a way that each deque operation runs in $O(1)$ time. ∎

6.2.4. (Communicated by A.G. Kushnirenko.) A deque of elements of type T is given. The deque contains several elements. The program should detemine how many elements are in the deque. Program may use variables of type T and integer variables, but arrows are not allowed.

[Hint. (1) We can perform a cyclic shift on deque elements taking an element from one end and adding it to the other end. After n shifts in one direction, we return the deque to its initial state by n shifts in the other direction. (2) How do we know that the cycle is complete? If we know in advance that some element is guaranteed not to appear in the deque, this is easy. We put this element into the deque and wait until it appears at the other end. But we do not have such an element. Instead, we may perform (for any fixed n) a cyclic shift by n positions twice adding two different elements. If the elements that appear after the shift are different, we have made a complete cycle.] ∎

Queue applications

6.2.5. (see E.W. Dijkstra's book [4]) Print in increasing order the first n positive integers whose factorization contains only the factors 2, 3, and 5.

Solution. The program uses three queues x2, x3, x5. They are used to store elements which are 2, 3, and 5 times larger than already printed elements, but are not yet printed. We use the procedure

```
procedure Print_and_add (t: integer);
begin
| writeln (t);
| Add (2*t, x2);
```

```
      Add (3*t, x3);
      Add (5*t, x5);
    end;
```

The program is as follows:

```
.. make queues x2, x3, x5 empty
Print_and_add (1);
k := 1;
{invariant relation: k first elements of the required
  set are printed; the queues contain elements
  that are 2, 3 and 5 times bigger that the elements
  already printed, but are not printed yet
  (in increasing order)}
while k <> n do begin
  x := min (Next(x2), Next(x3), Next(x5));
  Print_and_add (x);
  k := k+1;
  .. take x from the queues where it was present;
end;
```

Let us check the correctness of the program. Assume that the invariant relation is valid and we perform the operations as prescribed. Let x be the smallest element of our set that is not printed. Then x is larger than 1, and it is divisible by 2, 3, or 5. The quotient belongs to the set, too. The quotient is smaller than x and is therefore printed. Thus x is present in one of the queues. It is the smallest element in any queue to which x belongs (because all the elements less than x are already printed and cannot appear in any queue). When x is printed, we must delete x from the queues and add the corresponding multiples of x to maintain the invariant.

It is easy to check that queue lengths do not exceed the number of elements printed. ∎

The next problem is related to graphs (see section 9 for other graph problems).

Let V be a finite set whose elements are called *vertices*. Let E be a subset of the set $V \times V$; the elements of E are called *edges*. The sets E and V define a *directed graph*. A pair $\langle p, q\rangle \in E$ is called an edge *going from p to q*. One says that this edge *leaves* p and *enters* q. Usually vertices are drawn as points and edges as arrows. According to the above definition, there is at most one edge from p to q; edges that are loops (from p to p) are allowed.

A (directed) *path* is a sequence of vertices connected by edges (for example, path $pqrs$ contains four vertices p, q, r, and s, connected by three edges $\langle p, q\rangle$, $\langle q, r\rangle$, and $\langle r, s\rangle$.

6.2.6. Suppose a directed graph satisfies two requirements: (1) it is connected, that is, there is a path from any given vertex to any other vertex; and (2) for any

vertex the number of incoming edges is equal to the number of outgoing edges. Prove there exists an edge cycle that traverses each edge exactly once. Give an algorithm to find this cycle.

Solution. A "worm" is a nonempty queue of vertices such that each pair of adjacent vertices is connected by a graph edge (going in the direction from the first element to the last element). The first element in the queue is the tail of the worm; the last element in the queue is the worm's head. The worm can be drawn as a chain of arrows; arrows lead from the tail to the head. When a vertex is added, the worm grows near the head; when a vertex is removed, the tail is cut off.

Initially, the worm consists of a single vertex and evolves according to the following rule:

```
while the worm includes not all the edges do begin
  if there is an unused edge leaving from the worm's head
      then begin
    add this edge to the worm
  end else begin
    {the head and tail of the worm are at the same vertex}
    cut a piece of tail and add it to the head
    {"the worm eats its own tail"}
  end;
end;
```

Let us prove that this algorithm terminates when the worm spans all edges with its head and tail at the same vertex.

(1) Traversing the worm from tail to head, we enter each vertex as many times as we leave it. We also know that each vertex has as many incoming edges as it has outgoing edges. Therefore, we fail to find an outgoing edge only if the head of the worm is located at the same vertex as its tail.

(2) The worm never becomes shorter. Therefore, it will eventually reach some maximal length and never grow again. In the latter case, the worm will slide over itself forever. This is possible only if all the vertices visited do not have outgoing edges. Since the graph is connected, this is possible only if all the edges are included in the worm.

Some remarks about the Pascal implementation. The vertices are numbered `1..n`. For each vertex `i`, we store the number `Out[i]` of outgoing edges, as well as the numbers `Num[i][1]`, ..., `Num[i][Out[i]]` of vertices receiving the outgoing edges. While constructing the worm, we always choose the first unused edge. In this case, it is enough to keep (for each vertex) only the *number* of used outgoing edges to find the first unused edge. ∎

6.2.7. Prove that for any n there exists a bit string x of length 2^n with the following property: any binary string of length n is a substring of the string $xxx\ldots$. Find an algorithm that constructs such a binary string in time C^n (for some constant C that does not depend on n).

[Hint. Consider a graph whose vertices are binary strings of length $n-1$. An edge leaving x and entering y exists if and only if there is a string z of length n such that x is a prefix of z and y is a suffix of z. (In other words, if x minus its first bit is equal to y minus its last bit.) This graph is connected; each vertex has two incoming and two outgoing edges. A cycle that traverses all edges provides a string satistying the desired property.] ■

6.2.8. Implement k queues with total length not exceeding n, using memory of size $O(n+k)$ (that is, not exceeding $C(n+k)$ for some constant C). Each operation (except for initialization, which makes all the queues empty) should run in time $O(1)$ (that is, limited by a constant that does not depend on n).

Solution. We use the same method as for the pointer implementation of stacks. For each queue, remember the element that is first to be served; each element of the queue remembers the next element (the one that came immediately after). The last element believes that the next one is a special element number 0. We also have to remember the last element of each queue (otherwise we would trace the queue each time when a new element is added). As for stacks, all the free places are linked into a chain. Please note that for an empty queue the information about the last element makes no sense and is not used when adding elements.

```
    Content: array [1..n] of T;
    Next: array [1..n] of 0..n;
    First: array [1..k] of 0..n;
    Last: array [1..k] of 0..n;
    Free: 0..n;

procedure Make_empty;
| var i: integer;
begin
| for i := 1 to n-1 do begin
| | Next [i] := i + 1;
| end;
| Next [n] := 0;
| Free := 1;
| for i := 1 to k do begin
| | First [i]:=0;
| end;
end;

function Is_space: Boolean;
begin
| Is_space := Free  <> 0;
end;
```

```
function Is_empty (queue_number: integer): Boolean;
begin
| Is_empty := First [queue_number] = 0;
end;

procedure Take (var t: T; queue_number: integer);
| var frst: integer;
begin
| {not Is_empty (queue_number)}
| frst := First [queue_number];
| t := Content [frst]
| First [queue_number] := Next [frst];
| Next [frst] := Free;
| Free := frst;
end;

procedure Add (t: T; queue_number: integer);
| var new, lst: 1..n;
begin
| {Is_space}
| new := Free; Free := Next [Free];
| {location new is removed from free space list}
| if Is_empty (queue__number) then begin
| | First [queue_number] := new;
| | Last [queue_number] := new;
| | Next [new] := 0;
| | Content [new] := t;
| end else begin
| | lst := Last [queue_number];
| | {Next [lst] = 0 }
| | Next [lst] := new;
| | Next [new] := 0;
| | Content [new] := t
| | Last [queue_number] := new;
| end;
end;

function First_element (queue_number: integer): T;
begin
| First_element := Contents [First [queue_number]];
end;
```

■

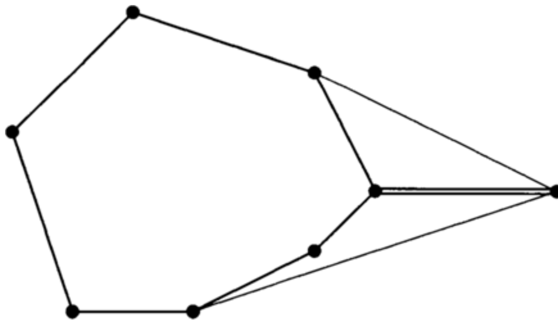
6.2.9. The same problem for deques.

[Hint. A deque is a symmetric structure, so we should keep pointers to both the next and preceding elements. It is convenient to tie the ends of each deque with a special element that forms a "ring". Another ring can be constructed from the free locations.] ∎

In the following problem, the deque is used to store the vertices of a convex polygon.

6.2.10. Assume that n points in the plane are numbered from left to right (and when the x-coordinates coincide, according to the order of the y-coordinates). Write a program that finds the convex hull of these n points in time $O(n)$ (that is, the number of operations should not exceed Cn for some constant C). The convex hull is a polygon, so the answer should be a list of all its vertices.

Solution. Consider the points one by one, each time adding a new point to the existing convex hull. The ordering guarantees that the new point becomes one of the vertices of the convex hull. We call this vertex of the convex hull a "marked" vertex. At the next step the marked vertex is visible from the point to be added. We extend our polygon by a "needle", which goes from the marked vertex to the new point and back. We obtain a degenerate polygon and eliminate "concavities" in that polygon.



The program stores the vertices of a polygon in a deque listed counter-clockwise from the "head" to the "tail". The marked vertex is both the head and the tail of the deque. Adding a "needle" means that new vertex is added to both ends of the deque. The elimination of concavities is more difficult. Let us call the elements nearest the head the "subhead" and "subsubhead", respectively. The elimination of concavities near the head is done as follows:

```
while going from the head to the subsubhead we turn
  |   to the right near the subhead  do begin
  | remove the subhead from the deque
end
```

The concavity near the tail is eliminated in a similar way.

Remark. Strictly speaking, operations involving the subhead and subsubhead of a deque are not allowed by definition. However, they may be reduced to a few legal operations (for example, we can take three elements, process them, and put back what remains).

Another remark. Two degenerate cases are possible. The first occurs when we do not turn at all near the subhead (in this case, the three vertices lie on the same line); the second occurs when we make a 180° turn (this happens when we have a "polygon" with two vertices). In the first case, the subhead should be removed (to eliminate the redundant vertices from the convex hull); in the second case, the deque is left unchanged. ■

6.3 Sets

Let `T` be a type. There are several methods to store (finite) sets of values of type `T`. There is no "best" method; the choice depends on type `T` and on the operations needed.

Subsets of $\{1, \dots, n\}$

6.3.1. Using $O(n)$ space (that is, space proportional to n), store a subset of $\{1, \dots, n\}$.

Operations	Time
Make empty	Cn
Test membership	C
Add	C
Delete	C
Minimal element	Cn
Test if the set is empty	Cn

Solution. Store the set as `array [1..n] of Boolean`. ■

6.3.2. The same problem with an additional requirement: test if the set is empty in constant (i.e., $O(1)$) time.

Solution. Store the number of elements in an additional variable. ■

6.3.3. The same problem with the following restrictions:

Operations	Time
Make empty	Cn
Test membership	C
Add	C
Delete	Cn
Minimal element	C
Test if the set is empty	C

Solution. Maintain also the minimal element of the set. ■

6.3.4. The same problem with the following restrictions:

Operations	Time
Make empty	Cn
Test membership	C
Add	Cn
Delete	C
Minimal element	C
Test if the set is empty	C

Solution. Store the minimal element of the set. Also, for each element we maintain pointers to the next and preceding elements (in order determined by value). ■

Sets of integers

In the following problems, elements of the set are integers (unbounded); the number of elements does not exceed n.

6.3.5. The memory size is limited by Cn.

Operations	Time
Make empty	C
Cardinality	C
Test membership	Cn
Add element (known to be absent)	C
Delete	Cn
Minimal element	Cn
Take some element	C

Solution. The set is represented by the variables

```
a:array [1..n] of integer, k: 0..n;
```

The set contains k (distinct) elements a[1], ..., a[k]. In a sense, we keep the elements of the set in a stack. (We require all elements in the stack to be different.) We may also use a queue instead of a stack. ■

6.3.6. The memory size is limited by Cn.

Operations	Time
Make empty	C
Test if the set is empty	C
Test membership	$C \log n$
Add	Cn
Delete	Cn
Minimal element	C

Solution. We use the same representation as in the preceding problem, with the additional restriction a[1] < ··· < a[k]. To test membership, we use binary search. ■

In the following problem, different methods are combined.

6.3.7. Find all the vertices of a directed graph that can be reached from a given vertex along the graph edges. The program should run in time Cm, where m is the total number of edges leaving the reachable vertices.

Solution. (A recursive solution is given in section 7.) Let num[i] be the number of outgoing edges for vertex i (assume that vertices are numbered 1..n). Let out[i][1], ..., out[i][num[i]] be the endpoints of the edges starting from vertex i.

```
procedure Print_Reachable (i: integer);
 |  {print all the vertices reachable from i,
 |      including the vertex i itself}
 | var  X: subset of 1..n;
 |      P: subset of 1..n;
 |      q, v, w: 1..n;
 |      k: integer;
begin
 | ...make X and P empty;
 | writeln (i);
 | ...add i to X, P;
 | {(1) P is the set of printed vertices; P contains i;
 |  (2) only vertices reachable from i are printed;
 |  (3) X is a subset of P;
 |  (4) all printed vertices which have an outgoing edge to
 |      a non-printed vertex, belong to X}
```

```
  while X is not empty do begin
    ...take some element of X into v;
    for k := 1 to num [v] do begin
      w := out [v][k];
      if w does not belong to P then begin
        writeln (w);
        add w to P;
        add w to X;
      end;
    end;
  end;
end;
```

Let us check that the requirements (1)–(4) mentioned in the program text, are satisfied. (1) We print a number and simultaneously add it to P. (2) Since v is in X, v is reachable; therefore, w is reachable. (3) Obvious. (4) We have deleted v from X, but all the endpoints of edges emanating from v are printed.

Let us prove the upper bound for the number of operations. If some element is removed from X, it never appears in X again. Indeed, it was present in P when removed, and only elements not in P can be added. Therefore the body of the while-loop is executed at most once for any reachable vertex; the number of iterations of the for-loop is equal to the number of outgoing edges.

For X we use a stack or queue representation (see above); for P we use a Boolean array. ■

6.3.8. Solve the preceding problem if all the reachable vertices are to be printed in the following order: first the given vertex, then its neighbors, then (unprinted) neighbors of its neighbors, etc.

[Hint. Use a queue for the representation of the set X in the program above. By induction over k we prove that at some point all the vertices having distance not exceeding k (and no others) are printed, and all the vertices having distance exactly k (and no others) are in the queue. For the detailed solution see section 9.2, p. 9.2] ■

More elaborate data structures for sets are considered in sections 11 (hash tables) and 12 (trees).

6.4 Priority queues

6.4.1. Implement a data structure that has the same set of operations as an array of length n, namely,

- initialize;
- put x in the i-th cell;

- find the contents of the i-th cell;

as well as the operation

- find the index of the minimal element

(or one of the minimal elements). Any operation should run in time $C \log n$ (except initialization, which should run in time Cn).

Solution. We use the trick from the heapsort algorithm. Assume that the array elements are positioned at the leaves of a binary tree and each non-leaf vertex contains the minimum of its two sons. To maintain this information and to trace the path from the root to the minimal element, we need only $C \log n$ operations. ■

6.4.2. A priority queue does not employ First In First Out (FIFO) rule; only an element's priority is important. An element is added to the priority queue with some priority (which is assumed to be an integer). When an element is taken from the queue, it is the element with the greatest priority (or one of the elements with greatest priority). Implement a priority queue in such a way that adding and removing elements requires logarithmic (in the size of the queue) time.

Solution. Here we follow the idea of the heapsort algorithm in its final form. We place queue elements in an array x[1..k] and maintain the following invariant relation: x[i] is higher (has larger priority) than its sons x[2i] and x[2i+1], if they exist. (Therefore, each element is higher than all its descendants.) The priority information is maintained along with the elements in the array, so we have an array of pairs ⟨element, priority⟩. From the heapsort algorithm, we know how to delete an element and maintain this relation. Another thing we need to do is restore this relation after adding some element to the end of the array. This is done as follows:

```
t:= the number of element added
{invariant: any element is higher than any its
   descendant if the descendant is not t}
while t is not root and t is higher than its father
|    do begin
|  exchange t and its father
end;
```

Suppose the priority queue is formed by people standing at the vertices of a tree (drawn on the ground); each person has one predecessor and at most two successors. The idea of the algorithm is this: A highly-ranked individual added to the queue begins to move toward the head of the queue. If a predecessor has lower rank, this new individual takes the predecessor's place. This continues until a higher-ranked predecessor is encountered. ■

Remark. The priority queue is an important data structure in simulation. Indeed, events are taken to be queue elements where the priority is determined by the time planned for the event.

7 Recursion

7.1 Examples

Up to now, we have not use recursion in our examples. Instead we devote a special section to this important programming technique. Recursion can be very useful and convenient, and in some cases recursive solutions are much shorter than nonrecursive ones.

When a recursive procedure is given, we have to show that:

(a) the procedure terminates;

(b) the procedure works properly (assuming it terminates).

Let us start with (b). Here it is enough to check that a procedure containing a recursive call works properly assuming that the called program (with the same name) works properly. Indeed, in this case, all the programs in the chain of recursive calls (from the end of the chain to its beginning) work properly. In other words, a recursive procedure is proved correct by induction.

To prove (a) we usually find a parameter that decreases as the recursion depth increases and prove that it cannot decrease indefinitely.

7.1.1. Write a recursive program that computes the factorial of a positive integer n (i.e., the product $n! = 1 \cdot 2 \cdots n$).

Solution. We apply the relations $1! = 1$ and $n! = (n-1)! \cdot n$ for $n > 1$.

```
procedure factorial (n: integer; var fact: integer);
| {fact := n!}
begin
| if n=1 then begin
| | fact:=1;
| end else begin {n>1}
| | factorial (n-1, fact);
| | {fact = (n-1)!}
| | fact:= fact*n;
| end;
end;
```

■

Using Pascal functions, we may write the above procedure as follows:

```
function factorial (n: integer): integer;
begin
| if n=1 then begin
| | factorial := 1;
| end else begin {n>1}
| | factorial := factorial(n-1)*n;
| end;
end;
```

Please note that the identifier `factorial` has two different meanings. It is a local variable as well as a function name. Fortunately, the difference is clear because the function name has parentheses after it. However, in the case of a function without parameters we have to be careful. (A common error occurs as the programmer sees a variable whereas the compiler sees a recursive call. This error is sometimes difficult to find.)

7.1.2. The factorial of 0 is defined as $0! = 1$. Modify the program accordingly. ■

7.1.3. Write a recursive program that computes the nonnegative integral power of a real number. ■

7.1.4. Repeat the previous problem with the requirement: the recursion depth (number of recursion levels) should not exceed $C \log n$, where n is the exponent.

Solution.

```
function power (a,n: integer): integer;
begin
| if n = 0 then begin
| | power:= 1;
| end else if n mod 2 = 0 then begin
| | power:= power(a*a, n div 2);
| end else begin
| | power:= power(a, n-1)*a;
| end;
end;
```

■

7.1.5. What happens if we replace the line

```
power:= power(a*a, n div 2)
```

in the above program by the line

```
power:= power(a, n div 2)* power(a, n div 2)
```

Solution. The program is still correct, but becomes much slower. In this case, one call of the function `power` generates *two* calls of the same function (with identical parameters). Thus, the number of calls grows exponentially as a function of the recursion depth. The program still has logarithmic recursion depth, but the number of steps is now linear instead of logarithmic. ■

This difficulty can be avoided by writing

```
t:= power(a, n div 2);
power:= t*t;
```

or by using Pascal's square function (sqr).

7.1.6. Using the procedure write(x) for x $= 0 \ldots 9$, write a recursive procedure that prints the decimal representation of a positive integer n.

Solution. The recursive solution allows us to produce digits from right to left but print them from left to right:

```
procedure print (n:integer); {n>0}
begin
| if n<10 then begin
| | write (n);
| end else begin
| | print (n div 10);
| | write (n mod 10);
| end;
end;
```

■

7.1.7. The "Towers of Hanoi" puzzle consists of three vertical sticks and N rings of different sizes. The rings are put on one of the sticks in such a way that larger rings are beneath smaller ones. We are to move this tower onto another stick one ring at a time. While moving the rings from one stick to another, we are not allowed to put a larger ring onto a smaller one. Write a program that shows the list of movements required to solve the problem.

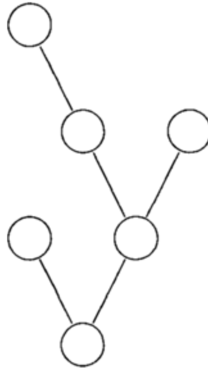
Solution. The following recursive procedure moves i upper rings from the m-th stick to the n-th stick (we assume that the remaining rings are larger and remain untouched).

```
procedure move(i,m,n: integer);
| var s: integer;
begin
| if i = 1 then begin
| | writeln ('move ', m, '->', n);
| end else begin
| | s:=6-m-n; {s is the third stick; 1+2+3 = 6}
| | move (i-1, m, s);
| | writeln ('move ', m, '->', n);
| | move (i-1, s, n);
| end;
end;
```

(The first recursive call moves a tower of i-1 rings onto the third stick. After that the i-th ring is free and is moved to the remaining stick. The second recursive call moves the tower onto the i-th ring.) ■

7.2 Trees: recursive processing

A binary tree is represented by a picture like this:



The vertex at the bottom of the tree is called the *root*. Two lines may go up from any vertex: one going up-left and one going up-right. These two vertices are called the *left* and *right sons* of the given vertex. Any given vertex may have either two sons, one son (which may be either the left son or the right son), or no sons at all. In the latter case, the vertex is called a *leaf*.

Let x be a vertex of tree. Consider this vertex together with its sons, grandsons, etc. This is a *subtree rooted at* x, the subtree of all *descendants* of the vertex x.

Please note that in most textbooks trees have root at the top and grow downwards. In some books terms "son" ("father", "brother", etc.) are replaced by "child" ("parent", "siebling", etc.).

In the following set of problems tree vertices are numbered by positive integers, and all numbers are different. The number assigned to the tree root is kept in the variable `root`. There exist two arrays

```
l,r: array [1..N] of integer
```

The left and right sons of the vertex number `i` have numbers `l[i]` and `r[i]`, respectively. If vertex x has no left (or right) son, the value of `l[i]` (resp., `r[i]`) is equal to 0. (Traditionally, we use the symbolic constant `nil` instead of the literal 0.) Numbers of all vertices do not exceed N.

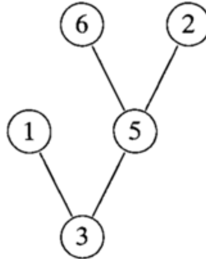
Let us stress that the vertex number has no connection with its position in a tree and that some integers in 1 ... N are not assigned to vertices at all. (Therefore, some data in the arrays `l` and `r` are irrelevant.)

7.2.1. Assume that N $= 7$, `root` $= 3$, and the arrays `l` and `r` are as follows:

i	1	2	3	4	5	6	7
l[i]	0	0	1	0	6	0	7
r[i]	0	0	5	3	2	0	7

Draw the corresponding tree.

Answer:



■

7.2.2. Write a program that counts all the vertices in a given tree.

Solution. Consider a function n(x), which is defined as the number of vertices in the subtree rooted at vertex number x. We agree that n(nil) = 0 (and the corresponding subtree is empty) and ignore the values n(s) for s not assigned to any vertex. Here is a recursive program that computes n(x):

```
function n(x:integer):integer;
begin
  if x = nil then begin
    n:= 0;
  end else begin
    n:= n(l[x]) + n(r[x]) + 1;
  end;
end;
```

(Vertices in the x-subtree are vertices in the subtrees rooted at its sons plus the vertex x itself.) The procedure terminates because the recursive calls refer to trees of smaller heights. ■

7.2.3. Write a program that counts the leaves in a tree.

Solution.

```
function n (x:integer):integer;
begin
  if x = nil then begin
    n:= 0;
  end else if (l[x]=nil) and (r[x]=nil) then begin {leaf}
```

```
  | n:= 1;
  end else begin
  | n:= n(l[x]) + n(r[x]);
  end;
end;
```

■

7.2.4. Write a program that finds the height of a tree. (The root of a tree has depth 0, its sons have depth 1, its grandsons have depth 2, etc. The height of a tree is the maximal depth of its vertices.

[Hint. Let `h(x)` be the height of the subtree rooted at `x`. The function `h` may be defined recursively.] ■

7.2.5. Write a program which for a given `n` counts all the vertices of depth `n` in a given tree. ■

Instead of counting vertices, we may ask to list them (in some order).

7.2.6. Write a program that prints all vertices (one time each).

Solution. The procedure `print_subtree(x)` prints all the vertices of the subtree rooted at `x` (one time each). The main program consists of the call `print_subtree(root)`.

```
procedure print_subtree (x:integer);
begin
| if x = nil then begin
| | {nothing to do}
| end else begin
| | writeln (x);
| | print_subtree (l[x]);
| | print_subtree (r[x]);
| end;
end;
```

This program uses the following ordering of tree vertices: first the root, then the left subtree, and then the right subtree. This order is determined by the order of the three lines in the `else`-part. Any of six possible permutations of these lines gives a specific order of tree traversal. ■

7.3 The generation of combinatorial objects; search

Recursion is a convenient tool to write programs that generate elements of some finite set. As an example, we now return to the problems of section 2.

7.3.1. Write a program that prints all sequences of length `n` composed of the numbers `1..k`. (Each sequence should be printed once, so the program prints k^n sequences.)

Solution. The program employs an array a[1]..a[n] and an integer variable t. The recursive procedure generate prints all sequences with prefix a[1]..a[t]; after it terminates, the value of t and a[1]..a[t] are the same as before the call.

```
procedure generate;
| var i,j : integer;
begin
  if t = n then begin
    for i:=1 to n do begin
    | write(a[i]);
    end;
    writeln;
  end else begin {t < n}
    for j:=1 to k do begin
      t:=t+1;
      a[t]:=j;
      generate;
      t:=t-1;
    end;
  end;
end;
```

The main program body now consists of two lines:

```
t:=0;
generate;
```

Remark. For efficiency reasons we may move the commands t:=t+1 and t:=t-1 out of the for-loop. ■

7.3.2. Write a program that prints all permutations of the numbers 1..n (each should be printed once).

Solution. The program utilizes an array a[1]..a[n] that contains a permutation of numbers 1..n. The recursive procedure generate prints all permutations that have the same first t elements as the permutation a. After the call, the values of t and a are the same as before the call. The main program is:

```
for i:=1 to n do begin a[i]:=i; end;
t:=0;
generate;
```

The procedure definition is as follows:

```
procedure generate;
| var i,j : integer;
```

```
begin
| if t = n then begin
| | for i:=1 to n do begin
| | | write(a[i]);
| | end;
| | writeln;
| end else begin {t < n}
| | for j:=t+1 to n do begin
| | | ..exchange a[t+1] and a[j]
| | | t:=t+1;
| | | generate;
| | | t:=t-1;
| | | ..exchange a[t+1] and a[j]
| | end;
| end;
end;
```

■

7.3.3. Print all increasing sequences of length k constructed from the natural numbers 1..n. (We assume that $k \le n$; otherwise the sequences do not exist.)

Solution. The program utilizes an array a[1]..a[k] and integer variable t. Assuming that a[1]..a[t] is an increasing sequence whose terms are numbers in 1..n, the recursive procedure generate prints all its increasing extensions of length k. (After the call, the values of t and a[1]..a[t] are the same as before the call.)

```
procedure generate;
| var i: integer;
begin
| if t = k then begin
| | ...print a[1]..a[k]
| end else begin
| | t:=t+1;
| | for i:=a[t-1]+1 to t-k+n do begin
| | | a[t]:=i;
| | | generate;
| | end;
| | t:=t-1;
| end;
end;
```

Remark. The for-loop may use n instead of $t - k + n$. The above version is more efficient; we use that the (k-1)-th term cannot exceed n-1, the (k-2)-th term cannot exceed n-2, etc.

The main program:

```
t:=1;
for j:=1 to 1-k+n do begin
| a[1]:=j;
| generate;
end;
```

(Another possibility is to add to a an auxiliary element a[0]:=0, then let t:=0 and call the procedure generate once.) ■

7.3.4. Generate all representations of a given positive integer n as the sum of a nonincreasing sequence of positive integers.

Solution. The program uses an array a[1..n] (the maximal number of summands is n) and an integer variable t. The procedure generate assumes that a[1] ... a[t] is a nonincreasing sequence of positive integers whose sum does not exceed n, and prints all the representations that extend this sequence. For efficiency reasons, the sum a[1] $+ \cdots +$ a[t] is kept in an auxiliary variable s.

```
procedure generate;
| var i: integer;
begin
| if s = n then begin
| | ...print a[1]..a[t]
| end else begin
| | for i:=1 to min(a[t], n-s) do begin
| | | t:=t+1;
| | | a[t]:=i;
| | | s:=s+i;
| | | generate;
| | | s:=s-i;
| | | t:=t-1;
| | end;
| end;
end;
```

The main program looks like

```
t:=1;
for j:=1 to n do begin
| a[1]:=j
| s:=j;
| generate;
end;
```

Remark. A small improvement is possible, since we may move the statements that increase and decrease t out of the loop. Also, instead of setting the value of

s each time (`s:=s+i`) and restoring it (`s:=s-i`) we may increase it by 1 at each time through the loop and restore the original value at the end of loop. Finally, we may add an auxiliary element a[0] = n and simplify the main program:

```
t:=0; s:=0; a[0]:=n; generate;
```

■

7.3.5. Write a recursive program that traverses a tree (using the same statements and conditions as in section 3).

Solution. The procedure process_above processes all the leaves above the robot's position and returns the robot to the start position. Here is the recursive definition:

```
procedure process_above;
begin
| if is_up then begin
| | up_left;
| | process_above;
| | while is_right do begin
| | | right;
| | | process_above;
| | end;
| | down;
| end else begin
| | process;
| end;
end;
```

■

7.4 Other applications of recursion

Topological sorting. Imagine n government officials, each of whom issues permissions of some type. We wish to obtain all the permissions (one from each official) according to the rules. The rules state (for each official) a list of permissions that should be collected before you can obtain this permission. There is no hope of solving the problem if the dependency graph has a cycle (we cannot get permission from A without having B's permission in advance, B without $C, \ldots,$ Y without Z, and Z without A). Assuming that such a cycle does not exist, we wish to find a plan that secures one of the permitted orders.

Let us represent officials by points and dependencies by arrows. (If permission B should be obtained before A, draw an arrow going from A to B.) We then have the following problem. There are n points numbered from 1 to n. From each point there are several (maybe zero) arrows that go to other points. (This picture is called a *directed graph.*) The graph has no cycles. We want to put the graph vertices in such an order that the end of any arrow precedes its beginning. This is the problem of *topological sorting.*

7.4.1. Prove that it is always possible to topologically sort a directed graph without cycles.

Solution. The absence of cycles implies that there exists a vertex with no outgoing edges (otherwise, we can follow edges until we come to the already visited vertex). This vertex with no outgoing edges gets number 1. After we discard all the edges entering vertex number 1, we reduce our problem to the same problem with a smaller number of edges. ■

7.4.2. Assume that a directed graph without cycles is stored in the following manner: Its vertices are numbered `1..n`. For any `i` in `1..n`, the value of `num[i]` is the number of edges leaving vertex `i`, and `adr[i][1]`, ..., `adr[i][num[i]]` are the numbers of vertices those edges enter. Write a (recursive) algorithm that performs a topological sort in time $C \cdot (\mathtt{n} + \mathtt{m})$, where `m` is the number of edges (arrows) in the graph.

Remark. The solution to the preceding problem does not provide such an algorithm directly; we need a more ingenious construction.

Solution. Our program prints the vertices in question (their numbers). It uses an array

```
printed: array[1..n] of Boolean;
```

such that `printed[i]` is true if and only if vertex `i` is already printed (this information is updated each time a vertex is printed). We say that a sequence of printed vertices is *correct* if **(a)** no vertex is printed twice, and **(b)** for any printed vertex `i` all the edges leaving `i` enter the vertices that are printed before `i`.

```
procedure add (i: 1..n);
| {before: the sequence of printed vertices is correct}
| {after: the sequence of printed vertices is correct
|         and includes i}
begin
| if printed [i] then begin {i is printed already}
| | {nothing to do}
| end else begin {printed sequence is correct}
| | for j:=1 to num[i] do begin
| | | add(adr[i][j]);
| | end;
| | {printed sequence is correct; all the edges going out
| |  of i are entering the vertices already printed; thus,
| |  we may print i correctly if it is not printed yet}
| | if not printed[i] then begin
| | | writeln(i); printed [i]:= TRUE;
| | end;
| end;
end;
```

The main program is:

```
for i:=1 to n do begin
| printed[i]:= FALSE;
end;
for i:=1 to n do begin
| add(i)
end;
```

The time bound will be proven shortly.

7.4.3. The program above can be simplified if we remove the test, replacing

```
if not printed[i] then begin
| writeln(i); printed [i]:= TRUE;
end;
```

by

```
writeln(i); printed [i]:= TRUE;
```

(Why?) How should we change the specification of the procedure?

Solution. The specification of the procedure is now as follows:

```
{before: the sequence of printed vertices is correct}
{after: the sequence of printed vertices is correct
        and includes i; all newly printed vertices
        can be reached from i}
```

■

7.4.4. The correctness of the program depends on the assumption about the absence of cycles. However, we did not mention this assumption in the solution of problem 7.4.2. Why?

Solution. We omitted the proof that the program terminates. Let us give it now. For any vertex we define its *level* as the maximal length of a path going out of it along the edges. The level is finite because there are no cycles. Vertices of level 0 have no outgoing edges. For any edge the level of its endpoint is smaller than the level of its starting point by at least 1. When `add(i)` is executed, all recursive calls refer to vertices whose levels are smaller. ■

Now we return to the time bound. How many calls `add(i)` are possible for some fixed `i`? The first call prints `i`; all others check that `i` is printed and exit immediately. It is also clear that all the calls `add(i)` are induced by the *first* calls of `add(j)` for all `j` such that the edge from `j` to `i` is present in the graph. Therefore, the number of calls `add(i)` is equal to the number of incoming edges for vertex `i`. All the calls except the first one require $O(1)$ time. The first requires time proportional to the number of outgoing edges (if we ignore the time needed

to perform add(j) for endpoints of outgoing edges). Therefore the total time is proportional to the total number of edges (plus the number of vertices). ■

Connected component of a graph. An *undirected graph* is a set of points (vertices) some of which are connected by lines (edges). An undirected graph is a special case of a directed graph where for each edge there is another edge going in the reverse direction.

The *connected component* of vertex i is the set of all vertices that are reachable from i via graph edges. Since the graph is undirected, the relation "j belongs to the connected component of i" is an equivalence relation.

7.4.5. Suppose an undirected graph is given (for each vertex its neighbors are listed; see the problem about topological sorting for details). Write an algorithm that for a given i prints all the vertices of the connected component of i (each vertex is printed once; no other vertices should be printed). The number of operations should be proportional to the total number of vertices and edges in the connected component.

Solution. The program will "blacken" vertices of the graph as they are printed. (Initially the vertices are assumed to be white.) By "white part" of the graph we mean that part of the graph that remains after we remove all black vertices and all edges adjacent to black vertices. The procedure add(i) blackens the connected component of i *in the white part of the graph* (and does nothing if i is already black).

```
procedure  add (i:1..n);
begin
| if i is black then begin
| | {nothing to do}
| end else begin
| | ..print i and mark i as black
| | for all j that are neighbors of i do begin
| | | add(j);
| | end;
| end;
end;
```

Let us prove that this procedure works properly (assuming that all recursive calls work properly). Indeed, it cannot blacken anything except the connected component of i. Let us check that all vertices in the connected component are blackened (and printed). Let k be a vertex that is reachable from x via path $i \to j \to \cdots \to k$, which includes only white vertices (and goes along graph edges). We may assume that this path does not visit vertex i again. Among all the paths, we consider the path with the smallest j (in the order they are considered in the for-loop). Then after the calls add(m) for preceding neighbors m, no one of the vertices in the path $j \to \cdots \to k$ becomes black (otherwise, j is not minimal). Therefore, k belongs to the connected component of the white part at the time when add(j) is called.

To prove that the algorithm terminates, it is enough to mention that the number of white vertices decreases at each recursion level.

Let us estimate the number of operations. Each vertex is blackened at most once, during the first call `add(i)` (for a given `i`). All subsequent calls occur when one of the neighbors is blackened. Therefore, the number of those calls is limited by the number of neighbors. And the only thing that happens during those calls is the check that `i` is already black. On the other hand, during the first call to `add(i)` all neighbors are considered and corresponding recursive calls are made. Therefore the total number of operations related to vertex `i` (not including the operations performed during the recursive calls `add(j)` for its neighbors `j`) is proportional to the number of neighbors of `i`. The upper bound stated in the problem follows. ■

7.4.6. Solve the same problem for a directed graph (that is, print all the vertices accessible from a given vertex). Note: the graph may contain cycles.

Solution. Essentially the same program can be used. The line "for all neighbors of a vertex" should be replaced by "for all endpoints of outgoing edges". ■

Hoare Quicksort. A well-known sorting algorithm called "quicksort" is a recursive algorithm considered to be one of the fastest algorithms available. Assume that an array `a[1]..a[n]` is given. The recursive procedure `sort(l,r:integer)` sorts an interval of the array `a` with indices in (l, r], that is, `a[l+1]..a[r]` (leaving the remaining part of the array unchanged).

```
procedure sort (l,r: integer);
begin
 | if l = r then begin
 | | {nothing to do - the interval is empty}
 | end else begin
 | | ..find a random number s in the interval (l,r]
 | | b := a[s]
 | | ..rearrange the elements of the interval (l,r]
 | |       into three parts:
 | |   the elements smaller than b  - the interval (l,ll]
 | |   the elements equal to b      - the interval (ll,rr]
 | |   then elements greater than b - the interval (rr,r]
 | | sort (l,ll);
 | | sort (rr,r);
 | end;
end;
```

How do we rearrange the elements of the interval according to the three categories listed in the above algorithm? As problem 1.2.32 (p. 28) shows, it can be done in time proportional to the length of the interval. Termination is guaranteed because the length of the interval decreases by at least 1 for each recursion level.

7.4.7. (For readers familiar with probability theory) Prove that the expected number of operations of the Hoare quicksort algorithm does not exceed $Cn\log n$ where the constant C does not depend on the array to be sorted.

[Hint. Let $T(n)$ be the maximal value of the expected number of operations (the maximal value is computed over all possible inputs of length n). The following inequality holds:

$$T(n) \le Cn + \frac{1}{n}\sum_{k+l=n-1}\bigl(T(k)+T(l)\bigr)$$

Indeed, the first summand corresponds to the stage where all elements are rearranged (divided into "less than", "equal to", or "greater than" parts). The second summand is an average value taken over all possible choices of a random number s. (To be precise, we should note that some of the elements may be equal to the threshold, so instead of $T(k)$ and $T(l)$ we should use the maximum of $T(x)$ over all x not exceeding k (or l), but this makes no difference.) Now, we prove by induction over n that $T(n) \le C'n\ln n$. To compute the average value of $x\ln x$ for all integer x such that $1 \le x \le n-1$, we integrate $\int_1^n x\ln x\,dx$ by parts as $\int \ln x\, d(x^2)$. When C' is large enough, the term Cn on the right-hand side is absorbed by the integral $\int x^2\, d\ln x$ and the inductive step is finished.] ∎

7.4.8. An array of n different integers and a number k is given. Find the k-th element of the array (in increasing order) using at most Cn operations (where C is some constant that does not depend on k and n).

Remark. Sorting algorithms can be used, but the number of operations ($Cn\log n$) is too big. The naïve algorithm (find the minimal element, then the next one, ..., then the k-th one) requires about kn operations (which is not allowed, because the constant C must not depend on k).

[Hint. An elegant method (though hardly practical since the constants are rather big) goes as follows:

A. Separate the array into $n/5$ groups each containing 5 elements. Sort each group.

B. Consider the median (third) elements of each group. This gives an array of $n/5$ elements. Calling our algorithm recursively, find the median element of this array. Call it M.

C. Compare all other elements of the initial array with M. They are divided into two groups (elements less than M and elements greater than M). Count the elements in both groups. Then we know which category the required (k-th) element belongs to and what its number is inside that part.

D. Apply the algorithm recursively to that part to find the required element.

Let $T(n)$ be the maximal possible number of operations when this algorithm is applied to arrays of length not exceeding n (the number k may be arbitrary). We have the following bound:

$$T(n) \le Cn + T(n/5) + T(0.7n).$$

The last term may be explained as follows. Each of the three categories contains at least $0.3n$ elements. Indeed, about half of all the median elements (in 5-element sets) are smaller than M. And if a median element of a 5-element set is smaller than M, then at least two more elements are smaller than M. Therefore, 3/5 of half of all elements are smaller than M.

Now the bound $T(n) \leq Cn$ can be proved by induction. The crucial point is that $1/5 + 0.7 < 1$.] ■

8 Recursive and nonrecursive programs

For a universal programming language (like Pascal) recursion is, in a sense, redundant: for any recursive program it is possible to write an equivalent program without recursion. Of course, this does not mean that recursion should be avoided, because it allows us to provide elegant solutions to otherwise complicated problems.

However, we want to show some methods that allow us to eliminate recursion in some cases and transform a recursive program into an equvalent nonrecursive program.

What for? A pragmatic answer is that many computers do not like recursion: recursive programs may be two or three times slower than equivalent nonrecursive programs. (Unfortunately, this is the case for some modern computers using so-called RISC processors.) Another problem is that some programming languages do not allow recursion at all. But the main reason is that elimination of recursion is sometimes very instructive.

8.1 Table of values (dynamic programming)

8.1.1. The following recursive procedure computes binomial coefficients. Write an equivalent program without recursion.

```
function C(n,k: integer):integer;
| {n >= 0; 0 <=  k <=n}
begin
| if (k = 0) or (k = n) then begin
| | C:=1;
| end else begin {0<k<n}
| | C:= C(n-1,k-1)+C(n-1,k)
| end;
end;
```

Remark. $C(n,k) = \binom{n}{k}$ is the number of k-element subsets of an n-element set. The identity

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

is proved as follows: Fix some element x of the n-element set. Then all k-element subsets are divided into two categories: those that contain x and those that do not. The elements of the first type are in one-to-one correspondence with the $(k-1)$-element subsets of a $(n-1)$-element set (just discard x); the elements of the second type are k-element subsets of a $(n-1)$-element set.

The table of $\binom{n}{k}$

$$
\begin{array}{ccccccccc}
 & & & & 1 & & & & \\
 & & & 1 & & 1 & & & \\
 & & 1 & & 2 & & 1 & & \\
 & 1 & & 3 & & 3 & & 1 & \\
1 & & 4 & & 6 & & 4 & & 1 \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{array}
$$

is called the *Pascal triangle* (the same Blaise Pascal who gave his name to a programming language). In this triangle, any element (except the 1s on the left and the right) is the sum of the two elements above it.

Solution. One may use the formula

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

We do not use it because we want to show more general methods to eliminate recursion. Our program fills the table of values `C(n,k)` $= \binom{n}{k}$ for $n = 0, 1, 2, \ldots$ until it reaches the element in question. ■

8.1.2. Compare the computation time for the recursive and nonrecursive versions of the binomial coefficient algorithm, and similarly for the amount of memory used.

Solution. The table used in the nonrecursive version occupies space of order n^2. We can reduce it to n if we recall that only one line of the Pascal triangle is needed to compute the next line. The time required is still n^2.

The recursive program requires much more time. Indeed, the call `C(n,k)` causes two calls of type `C(n-1,..)`, those two calls cause four calls of type `C(n-2,..)`, etc. Hence, the time is exponential (of order 2^n). The recursive procedure uses $O(n)$ memory (we have to multiply the recursion depth, that is n, by the amount of memory required by one copy of the procedure, that is $O(1)$. ■

The reason why the nonrecursive version is so much faster is the following. In the recursive version, the same computations are repeated many times. For example, the call `C(5,3)` causes *two* calls of `C(3,2)`:

```
            C(5,3)
            ↙    ↘
      C(4,2)      C(4,3)
      ↙    ↘      ↙    ↘
C(3,1)      C(3,2)      C(3,3)
```

When we fill the table, we compute the value for each cell only once, hence the economy of the nonrecursive method. This method is called *dynamic programming*, and is useful when the amount of information to be stored in the table is not too large.

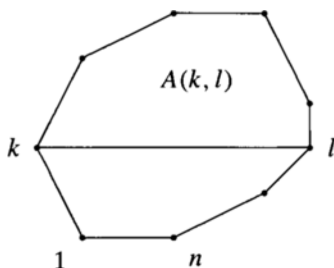
8.1.3. Compare the recursive and the (simplest) nonrecursive algorithm to compute the Fibonacci numbers defined as the sequence

$$\Phi_1 = \Phi_2 = 1; \quad \Phi_n = \Phi_{n-1} + \Phi_{n-2} \ (n > 2).$$

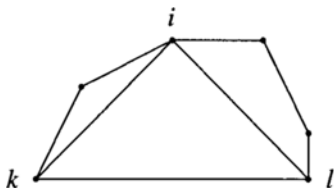
■

8.1.4. A convex polygon with n vertices is given (by a list of coordinates of its vertices). It is cut into triangles by non-intersecting diagonals; to do this, we need exactly $n - 3$ diagonals (proof by induction over n). The cost of the triangulation is defined as the total length of all the diagonals used. Find the minimal cost of the triangulation. The number of operations should be limited by a polynomial of n. (This requirement excludes exhaustive search, since the number of possibilities is not bounded by any polynomial.)

Solution. Assume that the vertices are numbered from 1 to n and the numbers increase in the clockwise direction. Let k and l be two numbered vertices and suppose $l > k$. By $A(k, l)$ we denote a polygon cut from the original polygon by segment k–l. (The segment k–l cuts the polygon into two polygons, one of which contains the 1–n side; $A(k, l)$ is the other one.) The initial polygon is denoted by $A(1, n)$. When $l = k + 1$, we have a degenerate polygon with only two vertices.



By $a(k, l)$ we denote the minimal cost of triangulation of $A(k, l)$. Let us write a recurrence formula for $a(k, l)$. When $l = k + 1$, we have a degenerate polygon with two vertices and let $a(k, l) = 0$. When $l = k + 2$, we have a triangle, and $a(k, l) = 0$. Assume that $l > k + 2$.



The chord k–l is a side of the polygon $A(k, l)$; therefore, it is a side of one of the triangles of the triangulation. The opposite vertex of this triangle has some number

i. It may be any of the vertices $k+1, \ldots, l-1$, and the minimal triangulation cost can be computed as

$$\min\{(\text{the length of } k\text{–}i) + (\text{the length of } i\text{–}l) + a(k,i) + a(i,l)\}$$

where the minimal value is taken over all $i = k+1, \ldots, l-1$. We should also take into account that for $q = p+1$, the segment p–q is one of the sides and its length should be counted as 0 for our purposes.

This formula allows us to fill the table of values $a(k,l)$ in order of increasing number of vertices (which is $l-k+1$). The corresponding program uses memory of order n^2 and time of order n^3 (one application of the recurrent formula requires a search for a minimal value among not more than n numbers). ■

8.1.5. An $m \times n$ matrix is a rectangular table with m rows and n columns filled with numbers. An $m \times n$ matrix may be multiplied by an $n \times k$ matrix (the width of the left factor must be equal to the height of the matrix on the right) giving $m \times k$ matrix as the result. The cost of such a multiplication is defined as mnk (this is the number of multiplications required by the standard multiplication algorithm, but this is not important). Matrix multiplication is associative, therefore the product of s matrices may be computed in any order. For each ordering, consider the total cost of all matrix multiplications. Find the minimal cost when the sizes of the matrices are given. The running time of the algorithm should be bounded by a polynomial over the number of factors (s).

Example. Matrices of size 2×3, 3×4, 4×5 can be multiplied in two different ways. The cost is either $2\cdot 3\cdot 4 + 2\cdot 4\cdot 5 = 64$ or $3\cdot 4\cdot 5 + 2\cdot 3\cdot 5 = 90$.

Solution. Suppose the first matrix is associated with an interval $[0,1]$, the second one is associated with $[1,2]$,. . ., and the s-th matrix is associated with $[s-1,s]$. Adjacent matrices (for segments $[i-1,i]$ and $[i,i+1]$) have a common dimension so we can multiply them. Let us denote this common dimension by $d[i]$. Therefore, the initial data of our problem is an array $d[0], \ldots, d[s]$.

Let $a(i,j)$ be the minimal cost of computation of the product of all the matrices in the interval $[i,j]$ (here $0 \le i < j \le s$). The cost in question is $a(0,s)$. The values of $a(i,i+1)$ are equal to 0 (we have only one matrix and nothing to multiply). The recurrence formula is as follows:

$$a(i,j) = \min\{a(i,k) + a(k,j) + d[i]d[k]d[j]\}$$

where the minimal value is computed over all possible places of the last multiplication, that is, over all $k = i+1, \ldots, j-1$. Indeed, the product of all matrices in the interval $[i,k]$ is a matrix of size $d[i] \times d[k]$, the product of all the matrices in the interval $[k,j]$ has size $d[k] \times d[j]$, and the cost of multiplication is $d[i]d[k]d[j]$. ■

Remark. The last two problems are rather similar. This is clear if we associate matrix factors with the sides 1–2, 2–3,. . .,($s-1$)–s of a polygon, and associate any chord i–j with the product of all matrices covered by this chord.

8.1.6. A one-way railway has n stops. We know the price of tickets from the i-th stop to the j-th stop (for $i < j$, since there is no traffic in the other direction). Find the minimal travel cost from stop 1 to stop n (taking into account possible savings due to intermediate stops). ■

We have seen that sometimes we get a more effective algorithm by replacing the recursion with a table that is filled cell by cell. A similar effect is achieved if we retain the recursive algorithm, but store the values of the function already computed and do not compute them again when the second request occurs. This trick is called *memoization.*

8.1.7. A finite set and a binary operation $\langle u, v\rangle \mapsto u \circ v$ defined on this set are given (the operation may be noncommutative and nonassociative). We have n elements $a_1, \ldots, a_n$ from the set and one more element x. Check if it is possible to insert parentheses in the expression $a_1 \circ \cdots \circ a_n$ in such a way that the result is equal to x. The number of operations should not exceed Cn^3 for some constant C (which depends on the cardinality of the set given).

Solution. Fill a table that contains (for any subexpression $a_i \circ \cdots \circ a_j$) the set of all possible values (for different placements of parentheses). ■

The same trick is used in the polynomial algorithm that tests whether a given word belongs to a context-free language (see section 13.1, p. 179).

The next problem (knapsack problem) was mentioned in section 3.2, p. 59.

8.1.8. An array $x_1, \ldots, x_n$ of n positive integers and an integer N are given. Check if N is equal to the sum of some subset of $\{x_1, \ldots, x_n\}$. The number of operations should be of order Nn.

[Hint. After i iterations, keep the set of all numbers in $0, \ldots, N$ that can be represented as a sum of some subset of $\{x_1 \ldots x_i\}$.] ■

8.2 Stack of postponed tasks

We illustrate another way to eliminate recursion using the Towers of Hanoi (p. 100) problem as an example.

8.2.1. Write a nonrecursive program that prints the sequence of moves for Towers of Hanoi problem.

Solution. Recall the following recursive program that moves `i` upper rings from stick `m` to stick `n`:

```
procedure move(i,m,n: integer);
| var s: integer;
begin
| if i = 1 then begin
| | writeln ('move ', m, '->', n);
```

```
  end else begin
    s:=6-m-n; {s is the unused stick; 1+2+3=6}
    move (i-1, m, s);
    writeln ('move ', m, '->', n);
    move (i-1, s, n);
  end;
end;
```

This program reduces the task "move i rings from m to n" to three tasks of the same type. Two of them deal with i-1 rings; one of them deals with 1 ring.

Try to execute this program manually. You'll see that it is rather difficult to remember which tasks are still to be done on different recursion levels.

The nonrecursive program uses a *stack of postponed tasks*, whose elements are triples ⟨i, m, n⟩. Each triple is interpreted as the request "move i (upper) rings from stick m to stick n". Tasks must be performed in the order they appear on the stack (the request on the top of the stack is the most urgent one). We obtain the following program:

```
procedure move(i,m,n: integer);
begin
  make the stack empty
  put <i,m,n> into the stack
  {invariant: it remains to process
                all the requests in the stack}
  while stack is not empty begin
    take the top of the stack into <j,p,q>
    if j = 1 then begin
      writeln ('move ', p, '->', q);
    end else begin
      s:=6-p-q; {s is the third stick; 1+2+3=6}
      put the triple <j-1,s,q> into the stack
      put the triple <1,p,q>   into the stack
      put the triple <j-1,p,s> into the stack
    end;
  end;
end;
```

(Please note that the triple put on the stack first will be the last request processed.) The stack of triples may be implemented as three separate stacks or one stack of records containing three integers (using a record type in Pascal). ∎

8.2.2. (Communicated via A.K. Zvonkin by Andrzei Lissowski.) There are other nonrecursive solutions of the Towers of Hanoi problem. Prove the correctness of the following solution: the *unused* stick (the stick that is neither the source nor the target of the move) should alternate cyclicly. (Another rule: alternately move the smallest ring and another ring, always moving the smallest one clockwise.) ∎

8.2.3. In the recursive program that prints a decimal number (7.1.6), replace the recursion by a stack.

Solution. The digits are generated from right to left and put onto the stack. They are taken from the stack (in the reverse order) and printed. ∎

8.2.4. Write a nonrecursive program that prints all the vertices of a binary tree.

Solution. In this case, the stack of postponed tasks will contain requests of two types: "print a vertex" and "print all the vertices of a subtree rooted at a given vertex". (We consider `nil` to be the root of an empty tree.) Therefore, the stack element is a pair ⟨request type, vertex number⟩.

When an element is taken off the stack, we either process it immediately (if it is a request of the first type) or put onto the stack the three requests caused by it (in one of six possible orderings). ∎

8.2.5. What if we only want to count the number of vertices but not print them?

Solution. Instead of printing a vertex, we add 1 to a counter. In other words, the invariant is the following: (total number of vertices) = (counter) + (the total number of vertices in the subtrees whose roots are in the stack). ∎

8.2.6. For some orderings (among six possible), the program that prints all vertices may be simplified. Show these simplifications.

Solution. If the order required is

root, left subtree, right subtree,

then a request to print the root may be processed immediately; thus we do not need to put it onto the stack.

A more complicated construction is necessary for the case

left subtree, root, right subtree.

In this case, all the requests in the stack (except the first one, which requests to print some subtree) are grouped into pairs

print vertex x, print "right subtree" of x

(that is, the subtree rooted at the right son of x). We can combine such pairs into requests of a special type and use an additional variable for the first request; in this way, all requests on the stack are homogenuous (have the same type).

For the symmetric case, similar simplifications are possible. Thus, for at least four of six possible orderings the program may be simplified. ∎

Remark. Another program that prints all the tree vertices is based on a program constructed in section 3. That program uses the command "down", which is not currently provided in the representation of trees. Therefore, we must keep a list of all vertices from the root to the current position (this list behaves like a stack).

8.2.7. Write a nonrecursive version of Hoare's quicksort algorithm. How do we guarantee that the size of the stack does not exceed $C \log n$, where n is the number of elements to be sorted?

Solution. The stack is filled with pairs $\langle i, j \rangle$, which are interpreted as requests to sort the corresponding intervals of the array. All such intervals are disjoint, therefore the size of the stack does not exceed n. To insure that the size of the stack is logarithmic, we follow the rule: "a larger request is pushed onto the stack first". Let $f(n)$ be the maximal size of the stack that may appear when sorting some array of length n using this rule. We desire an upper bound for $f(n)$. Indeed, after the array is split into two fragments, the shorter one is sorted first (whereas the request to sort the longer one is kept on the stack); then the longer fragment is sorted. At the first stage, the size of the stack does not exceed $f(n/2) + 1$, and at the second stage it does not exceed $f(n-1)$; therefore

$$f(n) \leq \max\{f(n/2) + 1, f(n-1)\}$$

A simple induction argument gives $f(n) = O(\log n)$. ■

8.3 Difficult cases

Let f be a function with nonnegative integer arguments and values defined by the equations

$$\begin{aligned} &f(0) = a, \\ &f(x) = h(x, f(l(x))) \quad (x > 0) \end{aligned}$$

Here a is some number while h and l are known functions. In other words, the value of f at x is determined by the value of f at $l(x)$. We assume that for any x, the sequence

$$x,\ l(x),\ l(l(x)), \ldots$$

reaches 0. If we know in addition that $l(x) < x$ for all x, the computation of f is trivial; just compute $f(0), f(1), f(2), \ldots$ sequentially.

8.3.1. Write a nonrecursive program to compute f in the general case.

Solution. To compute $f(x)$, compute the sequence

$$l(x),\ l(l(x)),\ l(l(l(x))), \ldots$$

until 0 appears. Now compute the values of f for all terms of this sequence, going from right to left. ■

The next example involves a more complicated case of recursion. (This example is hardly practical, and if it did appear in practice, it would probably be better to leave the recursion as is.)

Assume that a function f with nonnegative integer arguments and values is defined by the equations

$$f(0) = a,$$
$$f(x) = h(x, f(l(x)), f(r(x))) \quad (x > 0),$$

where a is a constant, and l, r, h are (known) functions. We assume that if one starts from any nonnegative integer and applies functions l and r in some arbitrary order, one eventually gets 0.

8.3.2. Write a nonrecursive program to compute f.

Solution. It is possible to construct a tree that has x at the root, and has $l(i)$ and $r(i)$ as sons of vertex i (unless i is equal to 0, in which case it is a leaf). Then we may compute the values of f from the leaves to the root. However, we'll use another approach.

By a *reverse Polish notation* (or *postfix notation*) we mean an expression where the function symbol is placed after all the arguments; parentheses are not used. Here are several examples:

$f(2)$	$2\ f$
$f(g(2))$	$2\ g\ f$
$s(2, t(7))$	$2\ 7\ t\ s$
$s(2, u(2, s(5, 3)))$	$2\ 2\ 5\ 3\ s\ u\ s$

Postfix notation allows us to compute the value of an expression easily using a so-called *stack calculator*. This calculator has a stack that we assume to be horizontal (the top of the stack is on the right), as well as number and function keys. When a number key is pressed, the number in question is put onto the stack. When a function key is pressed, the corresponding function is applied to the several arguments (according to its arity) taken from the stack. For example, if the stack contains the numbers

$$2\,3\,4\,5\,6$$

and the function key s is pressed (we assume that s is a function of two arguments), the new content of the stack is

$$2\,3\,4\ s(5, 6)$$

Now let us return to our problem. The program employs a stack whose elements are nonnegative integers. It also uses a sequence of numbers and the symbols `f`, `l`, `r`, `h` (which we consider a sequence of keys on a stack calculator). The invariant relation:

> If the number stack represents the current state of a stack calculator and we press all the keys in the sequence, the stack contains only one number that is the required answer.

Suppose we want to compute $f(x)$. Put the number x onto a stack and consider a sequence that contains only one symbol `f`. (The invariant relation is true.) Then the stack and the sequence are subjected to the following transformations:

old stack	old sequence	new stack	new sequence
X	$x\ P$	$X\ x$	P
$X\ x$	`l` P	$X\ l(x)$	P
$X\ x$	`r` P	$X\ r(x)$	P
$X\ x\ y\ z$	`h` P	$X\ h(x, y, z)$	P
$X\ 0$	`f` P	$X\ a$	P
$X\ x$	`f` P	X	$x\ x$ `l f` x `r f h` P

Here x, y, z are numbers, X is a sequence of numbers, and P is a sequence of numbers and the symbols `f`, `l`, `r`, `h`. In the last line, we assume that $x \neq 0$. This line corresponds to the equation

$$f(x) = h(x, f(l(x)), f(r(x)))$$

in postfix notation.

The transformations are performed until the sequence is empty. At that moment the invariant relation guarantees that the stack contains only one number, and this number is the answer required.

Remark. The sequence may be considered as a stack of delayed tasks (whose top is on the left). ∎

9 Graph algorithms

9.1 Shortest paths

This section is devoted to different versions of one problem. Suppose a country has n cities numbered 1..n. For each pair of cities i and j, an integer a[i][j] is given that is the cost of a (nonstop) plane ticket from i to j. We assume that flights exist between any two cities, and that a[k][k] $= 0$ for any k. In general, a[i][j] may be different from a[j][i]. Our goal is to find the minimal cost of a trip from one city (s) to another one (t) that takes into account all the possible travel plans (nonstop, one stop, two stops etc.). This minimal cost does not exceed a[s][t] but may be smaller. We allow a[i][j] to be negative for some i and j (you are paid if you agree to use some flight).

In the following problems, we compute the minimal cost for some pairs of cities, but first we have to check that our definition is correct.

9.1.1. Assume there is no cyclic travel plan with negative total cost. Prove that in this case a travel plan with minimal cost exists.

Solution. If a travel plan is long enough (includes more than n cities), it has a cycle, which may be omitted (because of our assumption). Now there are only a finite number of travel plans involving n or fewer cities. ■

Throughout the rest of this subsection, we assume that this condition (absence of negative cycles) is satisfied. (It is evident if all edge costs are nonnegative, but the latter condition is not always imposed.)

9.1.2. Find the minimal travel cost from the first city to all other cities in time $O(n^3)$.

Solution. By MinCost(1,s,k) we denote the minimal travel cost from 1 to s with less than k stops. It is easy to see that MinCost(1, s, k+1) is equal to

$$\min\left\{\texttt{MinCost(1,s,k)},\ \min_{\texttt{i=1..n}}\{\texttt{MinCost(1,i,k)} + \texttt{a[i][s]}\}\right\}$$

The minimum on the right-hand side is taken over all possible places of the last stop before the final destination.

As we have seen in the solution of the preceding problem, the cycles can be eliminated, so the answer in question is MinCost(1,i,n) for all i $=$ 1..n. We get the following program:

```
k:= 1;
for i := 1 to n do begin x[i] := a[1][i]; end;
{invariant: x[i] = MinCost(1,i,k)}
while k <> n do begin
| for s := 1 to n do begin
| | y[s] := x[s];
```

```
    for i := 1 to n do begin
      if y[s] > x[i]+a[i][s] then begin
        y[s] := x[i]+a[i][s];
      end;
    end
    {y[s] = MinCost(1,s,k+1)}
  end;
  for i := 1 to n do begin x[s] := y[s]; end;
  k := k + 1;
end;
```
■

This algorithm is called the dynamic programming algorithm, or Ford–Bellman algorithm.

9.1.3. Prove that the algorithm remains correct if the array y is not used, that is, if all changes are made in array x (just replace all y's by x's and delete redundant lines).

Solution. In this case the invariant is

$$\text{MinCost}(1,i,n) \le x[i] \le \text{MinCost}(1,i,k).$$ ■

This algorithm may be improved at least in two ways. First, with the same running time $O(n^3)$, we can find the minimal travel cost $i \to j$ for *all* pairs i,j (not just $i = 1$). Second, we can compute all travel costs from a given vertex in time $O(n^2)$. (In the latter case, however, we require all flight costs a[i][j] to be nonnegative.)

9.1.4. Find the minimal travel costs $i \to j$ for all i,j in time $O(n^3)$.

Solution. For any $k = 0..n$ consider the minimal travel cost from i to j *assuming intermediate stops are allowed only in cities* 1..k. This cost is denoted by A(i,j,k). Then

$$A(i,j,0) = a[i][j],$$
$$A(i,j,k+1) = \min\Big\{A(i,j,k),\ A(i,k+1,k) + A(k+1,j,k)\Big\}$$

(we either ignore city k+1 or use it as an intermediate stop; there is no reason to visit it twice). ■

This algorithm is called the Floyd algorithm.

9.1.5. Assume all costs a[i][j] are nonnegative. Find the minimal travel cost $1 \to i$ for all $i = 1..n$ in time $O(n^2)$.

Solution. Our algorithm will *mark* cities during its operation. Initially, only city number 1 is marked. Finally, all cities are marked. For all the cities, a "current cost" is maintained. This cost is a number whose meaning is explained by the following invariant relation:

- for any marked city `i`, the current cost is the minimal cost of travel `1` $\to$ `i`; it is guaranteed that this minimal cost is obtained via a path through marked cities only;
- for any non-marked city `i`, the current cost is the minimal cost among all travel plans `1` $\to$ `i` such that all intermediate stops are marked.

The set of marked cities is extended using the following observation:

> For a non-marked city with minimal current cost (among all non-marked cities), the current cost is the true cost and is reached via a path going through marked cities only.

Let us prove this. Assume that a shorter path exists. Consider the first non-marked city along this path. Even if we stop the trip in that city, the cost is already greater! (All costs are nonnegative.)

When a city is selected in this way, the algorithm marks it. To maintain the invariant, we update the current cost for non-marked cities. It is enough to take into account only those paths where the newly marked city is the last intermediate stop. This is easy to do since the minimal travel cost from the starting point to the newly marked city is already known.

If we store the set of marked cities in a Boolean array, we need $O(\mathtt{n})$ operations per city. ■

This algorithm is called the Dijkstra algorithm.

The problem of finding the shortest path has a natural interpretation in terms of matrices. Assume that A is the cost matrix for some carrier and B is the cost matrix for another carrier. Suppose we want to make one stop along the way, using the first carrier (with matrix A) for the first flight and the second carrier (B) for the second flight. How much should we pay for the trip from `i` to `j`?

9.1.6. Prove that the costs mentioned above form a matrix that can be computed using a formula similar to the standard formula for matrix multiplication. The only difference is that the sum is replaced by a min operation and the product is replaced by a sum:

$$C_{ij} = \min_{k=1,\ldots,n} \{A_{ik} + B_{kj}\}$$

■

9.1.7. Prove that matrix "multiplication" defined by the preceding formula is associative. ■

9.1.8. Prove that finding the shortest paths for all pairs of cities is equivalent to computation of A^{∞} for the cost matrix A in the following sense. For the sequence $A, A^2, A^3, \ldots$ there exists an N such that all elements A^N, A^{N+1}, etc. are equal to the matrix whose elements are minimal travel costs. (We assume that there are no cycles with negative cost.) ■

9.1.9. How large should N be in the preceding problem? ■

The usual (unmodified) matrix multiplication may also be applied, but in a different situation. Let us assume that only some flights exist and let `a[i][j]` be equal to `1` if there is a (direct) flight from `i` to `j`; otherwise, `a[i][j]=0`. Compute the `k`-th power of the matrix `a` (in the usual sense) and consider its `(i,j)`-th element.

9.1.10. What is the meaning of this element?

Solution. It is the number of different travel plans from `i` to `j` using `k` flights (and `k-1` intermediate stops). ■

Let us return to our original problem (finding the shortest path). We can easily extend our algorithms to the case where not all pairs of cities are connected by direct flights. Indeed, we may assume that nonexisting flights are infinitely expensive (or just very expensive), so our algorithms may be applied in this case too. However, a new question arises. The number of actual flights may be much smaller than n^2, so it is of interest to find algorithms that are more effective in this special case. First, we change the representation of the initial data: for each city we keep the number of outgoing flights and an array containing the destination points and costs.

9.1.11. Prove that the Dijkstra algorithm may be modified in such a way that if the number of cities is n and the total number of flights is m, then no more that $C(n+m)\log n$ operations are required.

[Hint. What should we do at each step? We must choose a non-marked city with minimal current cost and update the data for all cities that can be reached by direct flight from this city. If there were an oracle to inform us which of the unmarked cities has minimal current cost, $C(n+m)$ operations would be enough. And an additional $\log n$-factor in the running time allows us to maintain the information needed to find the minimal value in the array (see the problem on p. 97).] ■

9.2 Connected components, breadth and depth search

The simplest possible case of the shortest path problem is when all the flight costs are 0 or $+\infty$. This means that we want to know whether it is possible to travel from i to j, but do not worry about the price. In other words, we have a directed graph (a picture composed of points and arrows that connect some of the points) and we want to know which points are reachable from a given point via the arrows.

For this special case the algorithms given in the preceding section are not optimal. Indeed, a faster recursive program that solves this problem was given in section 7; its nonrecursive version was shown in section 6. Now we add the following additional requirement: We not only want to list all the points (vertices) that are reachable from a given vertex via arrows (edges), but we also want to list them in a specific order. Two of the most popular instances of this are the so-called "breadth-first" and "depth-first" search.

Breadth-first search

We are to list all the vertices of a directed graph that are reachable from a given vertex. The order is determined by the distance (minimal number of edges between a vertex and the given vertex). This list solves the answer to the problem of minimal travel cost when all the edges have cost 1 or $+\infty$.

9.2.1. Find an algorithm that performs breadth-first search in time Cm, where m is the total number of outgoing edges of all reachable vertices.

Solution. This problem was considered in section 6, p. 96. Here we give a detailed solution. Let `num[i]` be the number of outgoing edges for vertex `i`, and let `out[i][1]`, ..., `out[i][num[i]]` be the terminal vertices of the edges emanating from vertex `i`. Here is the program (as it was written before):

```
procedure Print_Reachable (i: integer);
   {print all the vertices reachable from i,
      including the vertex i itself}
  var  X: subset of 1..n;
       P: subset of 1..n;
       q, v, w: 1..n;
       k: integer;
begin
  ...make X and P empty;
  writeln (i);
  ...add i to X, P;
  {(1) P = is a set of printed vertices; P contains i;
   (2) only vertices reachable from i are printed;
   (3) X is a subset of P;
   (4) all printed vertices which have an outgoing edge to
       a non-printed vertex, belong to X}
  while X is not empty do begin
    ...take some element of X into v;
    for k := 1 to num [v] do begin
      w := out [v][k];
      if w does not belong to P then begin
        writeln (w);
        add w to P;
        add w to X;
      end;
    end;
  end;
end;
```

If we don't worry about the order in which the reachable vertices are printed, it doesn't matter which element of `X` is chosen by the algorithm. Now we assume

that `X` is a queue (first in, first out). In this case, the program prints all vertices reachable from `i` in order of increasing distance from `i` (distance is the number of vertices on the shortest path from `i`). Let us prove this assertion.

By $V(k)$ we denote the set of vertices whose distance from `i` (in the sense described above) is k. The set $V(k+1)$ is equal to the set

$$\text{(endpoints of edges whose startpoints are in } V(k)) \setminus (V(0) \cup \cdots \cup V(k))$$

Let us prove now that for a nonnegative integer $k = 0, 1, 2 \ldots$ there exists a point during the execution of the program (after one of the `while`-iterations) such that

- the queue contains all the elements of $V(k)$ and no other elements;
- all elements of $V(0), \ldots, V(k)$ and no others are printed.

For $k = 0$, it is the state before the first iteration. Now comes the induction step: Assume that at some point, the queue contains elements of $V(k)$. Those elements are processed one by one (the new elements are appended to the end of the queue and therefore cannot interfere). The endpoints of the edges emanating from the elements of $V(k)$ are printed and placed in the queue (unless they were printed earlier), exactly as in the equation for $V(k+1)$ shown above. Therefore, when all elements of $V(k)$ are processed, the queue is filled with all the elements of $V(k+1)$. ■

Depth-first search

When thinking about depth-first search, it is convenient to represent a given graph as the image of a tree. Let us explain what we mean by this. Suppose some vertex x of a directed graph is given. Assume that all vertices are reachable (via edges) from x. We construct a tree that may be called the "universal covering tree" of the graph. Its root is the point x, and it has the same outgoing edges as in the graph. The endpoints of those edges are sons of the root. Now consider any son y of x and all its outgoing edges. Their endpoints are the sons of y in the tree. The difference between the graph and the tree is that different paths from x to the same vertex of the graph now lead to different vertices of the tree. In other words, the vertex of the universal covering tree is a path in the graph starting from x. Its sons are paths that are one edge longer. Please note that the tree is infinite if the graph has (reachable) directed cycles.

There exists a natural mapping from the universal covering tree to the graph. For any vertex y in the graph, the number of preimages is the number of paths from x to y in the graph. Therefore, if we visit the tree vertices in some order, we at the same time visit the vertices of the graph (but some graph vertices may be visited many times).

Assume that for any graph vertex the outgoing edges are numbered. Then for any vertex of the universal covering tree its sons are numbered. Let us visit tree vertices in the following order: first the root, then the subtrees rooted at the

root's sons (in the given order of sons). An algorithm which traverses tree in that order was considered in section 7. This algorithm can be modified to traverse the graph avoiding visits to vertices already visited. Doing that, we get what is called "depth-first search".

Here is another description of depth-first search. Let us introduce a linear ordering on paths starting at a given vertex x. Any path precedes all its extensions. If two paths diverge at some vertex, they are ordered according to the ordering of the outgoing edges at that vertex. After that, vertices are ordered according to the minimal paths reaching them. This ordering is called *depth-first* ordering.

9.2.2. Write an algorithm for depth-first search.

[Hint. Take a program that traverses a tree (root → left subtree → right subtree) from section 7 or 8 and modify it. The main difference is that we do not want to revisit any visited vertex. Therefore, if we are at an already-visited vertex, we do nothing. (If a path is not minimal among all paths going to some vertex, all its extensions are not minimal as well, and can be safely ignored.)] ∎

Remark. Recall that in section 8 two possible nonrecursive algorithms for tree traversal were mentioned (p. 120). Both versions may be used for depth-first search.

Depth-first search is used in several graph algorithms (sometimes in a modified form).

9.2.3. An undirected graph is called a *bipartite graph* if its vertices may be colored in two colors in such a way that each edge connects vertices of different colors. Find an algorithm that checks whether a graph is a bipartite graph in time $C \cdot$ (number of edges + number of vertices).

[Hint. (a) Each connected component may be considered separately. (b) After we choose the color of some vertex, the colors of all other vertices of the same component are uniquely determined.] ∎

Remark. In this problem we may use breadth-first as well as depth-first search.

9.2.4. Write a nonrecursive algorithm for topological sorting of a directed graph without cycles. (For a recursive algorithm, see p. 108.)

Solution. Assume that the graph has vertices `1..n`. For any vertex `i`, we know the number `num[i]` of outgoing edges and the vertices `dest[i][1]`, ..., `dest[i][num[i]]` that the outgoing edges enter. We adopt the following terminology: the outgoing edges are listed "from left to right" (so `dest[i][1]` is "on the left" of `dest[i][2]`, etc.).

Our goal is to print all the vertices of the graph; the requirement is that the endpoint of any edge is printed before its starting point. We assume that the graph has no cycles (otherwise this is impossible).

Let us add to the graph an auxiliary vertex `0` that has `n` outgoing edges to `1`, ..., `n`. If it is printed and the requirement is fulfilled, then all other vertices are already printed.

Our algorithm maintains a path that starts at 0 (the auxiliary vertex) and traverses the graph edges. The length of this path is kept in an integer variable m. The path is formed by the vertices vert[1]..vert[m] and edges having numbers edge[1]..edge[m]. The number edge[s] refers to the numbering of all outgoing edges of the vertex vert[s]. Therefore, for all s, the following inequality holds:

$$\texttt{edge[s]} \leq \texttt{num[vert[s]]}$$

as well as the equality

$$\texttt{vert[s+1]} = \texttt{dest[vert[s]][edge[s]]}.$$

Please note that the end of the last edge in the path (i.e., the vertex dest[vert[m]][edge[m]]), is not included in the array vert. Moreover, we make an exception for the last edge and allow it to point "nowhere": edge[m] may be equal to num[vert[m]]+1.

The algorithm prints the vertices of the graph; a vertex is printed only after all the vertices where the outgoing edges go are printed. Moreover, the following requirement (I) is fulfilled:

> all vertices in the path, except the last one (i.e., the vertices vert[1]..vert[m]), are not printed, but if we turn to the left and leave our path, we immediately come to an already printed vertex.

Here is the algorithm in full:

```
m:=1; vert[1]:=0; edge[1]:=1;
{(I)}
while not( (m=1) and (edge[1]=n+1)) do begin
| if edge[m]=num[vert[m]]+1 then begin
| | {path leads to nowhere, therefore all vertices
| |     following vert[m] are printed and we may
| |     print vert[m]}
| | writeln (vert[m]);
| | m:=m-1; edge[m]:=edge[m]+1;
| end else begin
| |  {edge[m] <= num[vert[m]], path ends in a real
| |     vertex}
| |  lastvert:= dest[vert[m]][edge[m]];
| |  if lastvert is printed then begin
| |  | edge[m]:=edge[m]+1;
| |  end else begin
| |  | m:=m+1; vert[m]:=lastvert; edge[m]:=1;
| |  end;
| end;
end;
```

```
{the path immediately goes to nowhere, so all the
 vertices on the left (1..n) are printed}
```
■

9.2.5. Prove that if the graph has no cycles, this algorithm terminates.

Solution. Assume that this is not true. Any vertex may be printed at most once, so the vertices are not printed after some point. In a graph without cycles, the path length is limited (no vertex can appear in a path twice); therefore, after some point the path never becomes longer. After that, the only possibility is an increase in `edge[m]`, but this cannot happen infinitely many times. ■

9.2.6. Prove that the running time of the previous algorithm is O(number of vertices + number of edges). ■

9.2.7. Modify the algorithm in such a way that it can be applied to any graph. The algorithm should either find a cycle (if it exists) or perform a topological sort (if there are no cycles). ■

10 Pattern matching

10.1 Simple example

10.1.1. The character string x[1]..x[n] is given. Check if it contains the substring abcd.

Solution. There are approximately n (n-3, to be exact) positions where a substring of length 4 may be found. For each position, we can check whether the substring is in that position. This would require approximately $4n$ comparisons.

However, there is a more efficient approach. While reading the string x[1]..x[n] from left to right, we are looking for the character a. After it appears, we look for the character b (immediately after a), then for c, and finally d. If our expectations are met, the substring abcd is found. If one of the letters is not found where expected, we start from scratch looking for a again.

This simple algorithm can be described in different terms. In the framework of so-called *finite automata*, we say that while scanning x from left to right the algorithm is in one of the following "states": the initial state (0), the state "immediately after a" (1), "immediately after ab" (2), "immediately after abc" (3) and "immediately after abcd" (4). When reading the next character, we change the state according to the following rule:

Current state	Next character	New state
0	a	1
0	except a	0
1	b	2
1	a	1
1	except a,b	0
2	c	3
2	a	1
2	except a,c	0
3	d	4
3	a	1
3	except a,d	0

As soon as we come to state 4, or the input string is exhausted, the search is complete.

The corresponding program is straightforward (we indicate the new state even if it coincides with the old one, but those lines may be omitted):

```
i:=1; state:=0;
{i is the index of the first unread character;
 state is the current state}
```

```
while (i <> n+1) and (state <> 4) do begin
  if state = 0 then begin
    if x[i] = a then begin
      state:= 1;
    end else begin
      state:= 0;
    end;
  end else if state = 1 then begin
    if x[i] = b then begin
      state:= 2;
    end else if x[i] = a then begin
      state:= 1;
    end else begin
      state:= 0;
    end;
  end else if state = 2 then begin
    if x[i] = c then begin
      state:= 3;
    end else if x[i] = a then begin
      state:= 1;
    end else begin
      state:= 0;
    end;
  end else if state = 3 then begin
    if x[i] = d then begin
      state:= 4;
    end else if x[i] = a then begin
      state:= 1;
    end else begin
      state:= 0;
    end;
  end;
end;
answer := (state = 4);
```

In other words, at any point we have information about the maximal prefix of the pattern abcd which is a suffix of the substring already read. (Its length is the value of the variable `state`.) ■

Let us recall the terminology used. A *string* is an arbitrary finite sequence of elements of a set called an *alphabet*; its elements are called *letters*. If we discard some letters at the end of a string, we get another string, which is called a *prefix* of the first string. Any string is a prefix of itself. The *suffix* of a string is what remains after several initial letters of a string are discarded. Every string is a suffix of itself. A *substring* is obtained when we discard some letters both at the beginning and

the end of a string. (In other words, substrings are prefixes of suffixes, as well as suffixes of prefixes.)

In terms of "inductive functions" (see section 1.3) we can describe the situation as follows: Consider a function whose arguments are strings and whose values are either "True" or "False". The function has value "True" for all strings containing substring abcd. This function is not inductive, but it does have an inductive extension:

$$x \mapsto \text{the length of the maximal prefix of abcd that is a suffix of } x$$

10.2 Repetitions in the pattern

10.2.1. Can the previous algorithm be used for any other string instead of abcd?

Solution. A problem arises when the pattern contains repetitions. For example, suppose we are looking for substring ababc. Assume that a appears, then b, a, and b again. At this point, we are eagerly waiting for c. If the letter d appears instead, we should start from scratch. However, if the letter a appears, we still have a chance that b and c follow and the pattern is found.

Here is an illustration:

```
x y z a b a b a b c ...   <-  input string
      a b a b c           <-  pattern was expected here
          a b a b c       <-  but it is here
```

In other words, at the point

```
x y z a b a b |           <-  input string
      a b a b | c         <-  pattern was expected here
          a b | a b c     <-  but it is here
```

there are two possible pattern positions to be tested. ∎

However, a finite automaton that reads the input string letter-by-letter, changes its state according to some table, and says (after the input string is exhausted) whether the input string contains the given substring, is still possible.

10.2.2. Construct such an automaton. Show all its states and the transition table (which determines a new state as a function of an old state and an input character).

Solution. As before, the current state is the length of the maximal prefix of the pattern that is also a suffix of the currently read part of the string. There are six states: 0, 1 (a), 2 (ab), 3 (aba), 4 (abab), 5 (ababc). The transition table is as

follows:

Current state	Next character	New state
0	a	1 (a)
0	except a	0
1 (a)	b	2 (ab)
1 (a)	a	1 (a)
1 (a)	except a,b	0
2 (ab)	a	3 (aba)
2 (ab)	except a	0
3 (aba)	b	4 (abab)
3 (aba)	a	1 (a)
3 (aba)	except a,b	0
4 (abab)	c	5 (ababc)
4 (abab)	a	3 (aba)
4 (abab)	except a,c	0

Consider the second (from below) line in this table as an example. If the processed part is ended by abab and the next letter is a, the new processed part is ended by ababa. The maximal prefix of ababc, which is also a suffix of the processed part, is aba. ■

Question: as we said, the difficulty appears because there are several possible positions of the pattern; each position corresponds to some prefix of the pattern that is also a suffix of the input string. The finite automaton remembers only the longest one. What about others?

Answer. The longest prefix-suffix X determines all other prefix-suffixes. Namely, prefix-suffixes of the processed part are prefixes of X that are also suffixes of X.

It is easy to write a transition table and a program for any fixed pattern. However, we want to write a general program that will search for any given pattern in any given input string. The following approach may be used. Consider a program that has two stages. In the first stage, it examines the pattern and constructs a transition table for that pattern. In the second stage, it reads the input string and behaves according to the transition table. Such an approach is often used for more complicated patterns (see below), but for substring search there is more direct and efficient method called Knuth–Morris–Pratt algorithm. (A similar approach was suggested by Yu. Matijasevich.) We start with some auxiliary lemmas.

10.3 Auxiliary lemmas

For any string X, consider all the prefixes of X that are at the same time suffixes of X. Choose the longest one (not counting X itself), which is denoted by $l(X)$.

For example, l(aba) = a, l(abab) = ab, l(ababa) = aba, l(abc) = the empty string.

10.3.1. Prove that all strings $l(X)$, $l(l(X))$, $l(l(l(X)))$, etc. are prefixes of X.

Solution. Each of them is a prefix of the preceding one. (And any prefix of a prefix of X is also a prefix of X.) ■

For the same reason all such strings are suffixes of X as well.

10.3.2. Prove that the sequence in the preceding problem is finite (the last string is empty).

Solution. Each subsequent string is shorter than the preceding one (since $l(Y)$ is shorter than Y for any Y). ■

10.3.3. Prove that any string that is both a prefix and a suffix of X (except for X itself) is listed in the sequence $l(X), l(l(X)), \ldots$.

Solution. Let Y be both a prefix of X and a suffix of X. The string $l(X)$ is the longest string having this property, so Y is not longer than $l(X)$. Both Y and $l(X)$ are prefixes of X, and the shorter one is a prefix of the longer one. Thus Y is a prefix of $l(X)$. For the same reason, Y is a suffix of $l(X)$. Using an induction argument, we assume that the statement in question is true for all strings shorter than X. In particular, it is true for $l(X)$. So the string Y, being a prefix and a suffix of the string $l(X)$, is either equal to $l(X)$ or one of the strings $l(l(X)), l(l(l(X))), \ldots$. ■

10.4 Knuth–Morris–Pratt algorithm

The Knuth–Morris–Pratt (KMP) algorithm takes a string

$$\mathtt{X} = \mathtt{x[1]x[2]..x[n]}$$

as input and scans it from left to right. The output is the sequence of nonnegative integers `L[1]...L[n]` such that

$$\mathtt{L[i]} = \text{the length of } l(\mathtt{x[1]..x[i]})$$

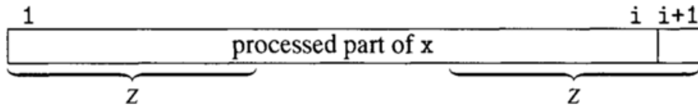
(the function L is defined in the preceding section). In other words, `L[i]` is the length of the maximal prefix of `x[1]..x[i]` that is simultaneously a suffix of `x[1]..x[i]`.

10.4.1. How can we use the KMP algorithm to check whether a given string `A` is a substring of a string `B`?

Solution. Apply the KMP algorithm to the string `A#B`, where `#` is a special character that does not appear in `A` or `B`. The string `A` is a substring of `B` if and only if the array `L` (which is the output of the KMP algorithm) contains a number equal to the length of `A`. ■

10.4.2. How do we fill the table `L[1]..L[n]`?

Solution. Assume that the first `i` values `L[1]..L[i]` are already known. We read the next input character (i.e., `x[i+1]`) and compute `L[i+1]`.



How do we find L[i+1]? It is the length of the longest prefix Z of the string x[1]..x[i+1] that is at the same time a suffix of this string. Any string Z having this property (except for the empty string) is obtained from some string Z' by adding the letter x[i+1]. The string Z' is both a prefix and a suffix of the string x[1]..x[i]. However, it is not the only requirement for Z'; another requirement is that Z' is followed (as a prefix of x[1]..x[i]) by x[i+1].

Therefore, the string Z may be found as follows. Consider all the prefixes Z' of the string x[1]...x[i] that are also the suffixes of this string. Then choose the longest one that is followed (as a prefix of x[1]...x[i]) by x[i+1]. Adding x[i+1] produces the string Z.

Now it is time to use the lemmas proved earlier. Recall that all strings that are both prefixes and suffixes may be obtained by applying the function l iteratively. Here is the program:

```
i:=1; L[1]:= 0;
{the table L[1]..L[i] is filled correctly}
while i <> n do begin
  len := L[i]
  {len is the length of a prefix  of x[1]..x[i] that is
    its suffix; all longer prefixes-suffixes were
    tested without success}
  while (x[len+1] <> x[i+1]) and (len > 0) do begin
    {this prefix does not fit also, we should apply l}
    len := L[len];
  end;
  {we either have found the longest prefix that
   fits our requirements (and its length is n)
   or have found that it does not exist (len=0)}
  if x[len+1] = x[i+1] do begin
    {x[1]..x[len] is the longest prefix that fits}
    L[i+1] := len+1;
  end else begin
    {there are no good prefixes}
    L[i+1] := 0;
  end;
  i := i+1;
end;
```

■

10.4.3. Prove that the number of operations in the above algorithm is limited by Cn for some constant C.

Solution. This is not obvious, because one input character may cause many iterations in the inner loop. However, each iteration in the inner loop decreases `len` by at least `1`, so in this case, `L[i+1]` will be significantly smaller than `L[i]`. On the other hand, while `i` is increased by `1`, the value of `L[i]` may increase by at most `1`, therefore the values of `i` that require many iterations in the inner loop are rare.

Formally, we use the inequality

$$\texttt{L[i+1]} \le \texttt{L[i]} - (\text{the number of iteration at step } \texttt{i}) + 1$$

or

$$(\text{the number of iterations at step } \texttt{i}) \le \texttt{L[i]} - \texttt{L[i+1]} + 1$$

Summing these inequalities over `i`, we get the required upper bound for the total number of iterations. ■

10.4.4. Imagine that we use this algorithm to determine whether a string `X` of length `n` is a substring of a string `Y` of length `m`. (We explained above how to do that using a "separator" `#`.) The algorithm runs in time $O(\texttt{n}+\texttt{m})$ and uses memory of size $O(\texttt{n}+\texttt{m})$. Find a way to do this using memory of size $O(\texttt{n})$ (which may be significantly less if the pattern is short and the string is long).

Solution. Start applying the KMP algorithm to the string `A#B`. Wait until the algorithm computes all the values `L[1]`, ..., `L[n]` for the word `X` of length `n`. All those values are stored. From then on, we keep only the value `L[i]` for the current `i`; we only need `L[i]` and the table `L[1]..L[n]` to compute `L[i+1]`. ■

In practice, the words `X` and `Y` are usually separated, so the scan of `X` and the scan of `Y` should be implemented as two different loops. (This also makes the separator `#` unnecessary.)

10.4.5. Write the program discussed in the last paragraph, which checks whether the string `X` = `x[1]..x[n]` is a substring of the string `Y` = `y[1]..y[m]`.

Solution. First we fill the table `L[1]...L[n]` as before. Then we execute the following program:

```
j:=0; len:=0;
{len is the length of a longest prefix  of X which is
  a suffix of y[1]..y[j]}
while (len <> n) and (j <> m) do begin
| while (x[len+1] <> y[j+1]) and (len > 0) do begin
| | {this prefix does not fit}
| | len := L[len];
| end;
```

```
  | {we have found the prefix that fits or
  |     have found that it does not exist}
  | if x[len+1] = y[j+1] do begin
  | | {x[1]..x[len] is the longest prefix that fits}
  | | len := len+1;
  | end else begin
  | | {no prefixes fit}
  | | len := 0;
  | end;
  | j := j+1;
end;
{if len=n, X is a substring of Y;
  otherwise we reached the end of Y not finding X}
```
■

10.5 Boyer–Moore algorithm

This algorithm attains a goal that seems impossible at first: In a typical situation, it reads only a tiny fraction of all the characters of a string in which the pattern is searched. How can this be done? The idea is rather simple. Suppose we are searching for the pattern abcd in a string X. Check the fourth character of X. If it is, say, e, there is no need to look at the first three characters, because our pattern does not contain e and may start only after fourth position.

We show below a simplified version of the Boyer–Moore algorithm that does not guarantee good running time in all cases.

Let X = x[1]..x[n] be the pattern we are searching for. For any character s, we find the rightmost occurrence of s in the string X, that is, the maximal k such that x[k] = s. This information is stored in an array pos[s]. If the character s does not appear in the pattern at all, it is convenient to put pos[s] :=0 (see below).

10.5.1. How do we fill the array pos?

Solution.

```
...let all pos[s] be equal to 0
for i:=1 to n do begin
| pos[x[i]]:=i;
end;
```
■

The program searches for the pattern x[1]..x[n] in the input string y[1]..y[m]. When searching, store in the variable last the number of the input character that corresponds to the last character of the pattern (in the current pattern position). Initially, last = n (the length of the pattern); then last increases gradually.

```
last:=n;
{all previous positions of the pattern are checked}
while last <= m do begin {the work is not finished}
| if x[n] <> y[last] then begin
| | {the last characters differ}
| | last := last + (n - pos[y[last]]);
| | {n - pos[y[last]] is the minimal shift of the
| |    pattern that makes the character y[last]
| |    match the corresponding character in the
| |    pattern. If y[last] does not appear in the
| |    pattern, the new pattern position starts
| |    immediately after y[last]}
| end else begin
| | {x[n] = y[last]}
| | check if the current position is okay, that is,
| | if x[1]..x[n] = y[last-n+1]..y[last].
| | If yes, inform about that.
| | last := last+1;
| end;
end;
```

It is recommended to start testing the condition (x[1]..x[n] = y[last-n+1]..y[last]) from right to left starting from the last position (where the coincidence is already tested). We also obtain a small optimization if we store n-pos[s] instead of pos[s] (avoiding subtraction at each step); n-pos[s] is the number of characters to the right of the rightmost occurrence of character s in the pattern.

Different versions of this algorithm exist. For example, we may replace the line last:=last+1 by last:=last+(n-u), where u is the position of the second (from the right) occurrence of the character x[n] in the pattern.

10.5.2. What modifications in the program are necessary?

Solution. To fill up the table pos, we use the line

```
for i:=1 to n-1 do...
```

(all other lines remain the same); in the main program we replace the line last:=last+1 by

```
last:= last+n-pos[y[last]];
```

■

This simplified version of the Boyer–Moore algorithm sometimes require significantly more that n operations (mn in the worst case), so the worst-case behavior of the Knuth–Morris–Pratt algorithm is much better.

10.5.3. Give an example where a pattern of length n is not a substring of a given string of length m, but the program above requires mn operations to determine this.

Solution. Assume that the pattern is baaa..aa and the string contains n letters a. Then at each step we need n comparisons to discover that the pattern is not a substring. ∎

The complete (not simplified) Boyer–Moore algorithm guarantees that the number of operations does not exceed $C(\mathtt{m}+\mathtt{n})$ in the worst case. It uses ideas similar to those in the KMP algorithm. Suppose we compare the pattern and the string from right to left. Assume that we find the coincident suffix Z of the pattern, but the characters before Z in the input string and in the pattern are different. What do we know about the input string at that point? We have found a fragment equal to Z that is preceded by a character that differs from the character in the pattern. This information may allow us to shift the pattern to the right several positions. These shifts should be computed in advance for all suffixes Z of the pattern. One can prove that all operations (the computation and use of the shift table) can be performed in time $C(\mathtt{m}+\mathtt{n})$.

10.6 Rabin–Karp algorithm

This algorithm is also based on a simple idea. Suppose we are looking for a pattern of length n in a string of length m. Let us make a sliding window and move it along the input string. Our goal is to check whether the substring in the window coincides with the given pattern.

We want to avoid character-by-character comparison and find a faster method. Let us consider some function defined on strings of length n. If this function takes on different values when applied to both the pattern and the substring in the window, we may be sure that there is no match. Only if the function values coincide, we have to compare strings character-by-character.

What do we gain? It seems that we have achieved nothing because to compute the function value for the substring in the window, we have to read all the characters in the window anyway. So why not just compare them with the pattern characters? Some gain, however, is still possible for the following reason. When we shift the window, the substring in it does not change completely; a single character is appended on the right and deleted on the left. If our function is well chosen, we may compute its new value quickly, knowing its old value and the added/deleted characters.

10.6.1. Find an example of such a "well chosen" function.

Solution. Replace all characters in the pattern by their codes, which are assumed to be integers. The sum of all codes is such a function. (Indeed, after the shift, we only have to add the numeric value of the new character and subtract the numeric value of the old character.) ∎

Given any function, most likely there are distinct strings that are mapped to the same value. For the same pair of strings another function may indeed produce distinct values. So let us have a pool of functions and begin the algorithm by

choosing one of the functions at random. Then an adversary who wants to choose the worst problem instance will not know which function it is working against.

10.6.2. Give an example of a family of easily computable functions (in the sense explained above). ■

Solution. Let us choose some number p (presumably prime; see below) and some residue x modulo p. Each string of length n is considered as a sequence of integers (characters are replaced by their numeric codes). Those integers are taken to be coefficients of a polynomial of degree $n-1$. We compute the value of this polynomial modulo p at the point x. This construction provides one function of the family (for each p and x we get another function). When the window is shifted by 1, we subtract the term of the highest degree (x^{n-1} should be computed in advance), multiply by x, and add the constant term.

The following arguments show that the coincidence of function values (for different arguments) is not very likely. Assume that p is fixed and is prime. Let X and Y be two different words of length n. Then the corresponding polynomials are different. (We assume that different characters have different codes modulo p, so we need p to be larger than the size of the alphabet.) The coincidence of function values on X and Y means that two different polynomials coincide at x, that is, x is a root of their difference. This difference is a nonzero polynomial of degree $n-1$ and can have at most $n-1$ roots. Therefore, if n is much smaller than p, the chances for the random x to be a root are negligible.

10.7 Automata and more complicated patterns

Rather than a specific string, we may search for a string of some type. For example, we may look for a substring of type `a?b` where `?` denotes any single character. In other words, we are looking for characters `a` and `b` with exactly one character in between.

10.7.1. Construct a finite automaton that checks if the pattern `a?b` is present in the input string.

Solution. While reading the input string, the automaton keeps track of whether the character `a` is present at the two last positions. The automaton has states 00, 01, 10, 11 with the following meanings:

00	no `a` in the last two positions
01	`a` is in the last position but not in the position immediately before it
10	`a` is in the position before the last one but not in the last position
11	the processed part of the input string ends with `aa`

Here is the transition table:

Current state	Next character	New state
00	a	01
00	not a	00
01	a	11
01	not a	10
10	a	01
10	b	found
10	not a and not b	00
11	a	11
11	b	found
11	not a and not b	10

■

Another widely used notation in a pattern is an asterisk (∗), which is matched by any string (including the empty string). For example, the pattern ab∗cd means that we are looking for any occurrence of ab followed by cd (the distance between ab and cd is arbitrary).

10.7.2. Construct a finite automaton that checks if the input string contains the pattern ab∗cd (in the sense just described).

Solution.

Current state	Next character	New state
initial	a	a
initial	not a	initial
a	b	ab
a	a	a
a	not a and not b	initial
ab	c	abc
ab	not c	ab
abc	d	found
abc	c	abc
abc	not c and not d	ab

■

Another type of search occurs when we are looking for a substring that belongs to a given finite set of strings.

10.7.3. Assume that strings $X_1, \ldots, X_k$ (patterns) and a string Y are given. Check if one of the strings X_i is a substring of the string Y. The number of

operations should not exceed the total length of all the strings (X_i and Y) multiplied by some constant which does not depend on k.

Solution. The obvious approach is to check all the X_i separately (using one of the algorithms given above). However, this method does not satisfy the speed requirements (since we have to read the string Y many, in fact, k times).

Let us look at another aspect of the problem. For each pattern X_i, there exists a finite automaton that tests for the presence of X_i. These automata may be combined into one automaton whose set of states is the product of the sets of states for all the automata. This set is very large. However, most of its elements are unreachable and may be discarded.

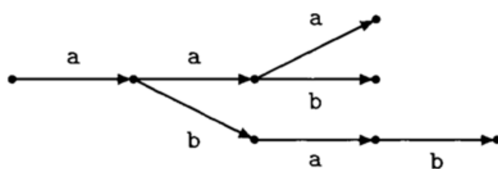
This idea is used below (in a modified form).

Let us recall the Knuth–Morris–Pratt algorithm. While reading the input string, the KMP algorithm keeps the maximal prefix of the pattern that is a suffix of the processed part of the input string. Now we need to keep this information (the longest prefix that is a suffix of the processed part) for all the patterns. The crucial remark is: It is enough to keep the longest one, because all others are uniquely determined by the longest one. Indeed, let X be the longest prefix of some pattern that is a suffix of the processed part of the input string. Then for any pattern P, the longest prefix of P being a suffix of the processed part is the longest prefix of P being a suffix of X.

All the patterns may be "glued" together to form a tree if we "splice" together equal prefixes. For example, the set of patterns

$$\{\texttt{aaa}, \texttt{aab}, \texttt{abab}\}$$

corresponds to the tree



Here is the formal definition: any prefix of any pattern is a tree vertex; a father of a vertex is obtained by deleting the last character.

While reading the input string, we traverse this tree. The current position is the maximal (rightmost) vertex that is a suffix of the processed part of the input string (that is, the longest suffix of the processed part being a prefix of one of the patterns).

Let us introduce a function l whose arguments and values are tree vertices, namely, $l(P)$ = maximal tree vertex that is a (proper) suffix of P. (Recall that tree vertices are strings.) The following result will be used:

10.7.4. Let P be a tree vertex. Prove that the set of all tree vertices that are (proper) suffixes of P is $\{l(P), l(l(P)), \dots\}$

Solution. See the proof of the similar assertion for the Knuth–Morris–Pratt algorithm. ∎

Now it is clear what the algorithm (or automaton) should do if it is at the vertex P and the next input character is z: It should consider sequentially the vertices P, $l(P)$, $l(l(P)), \dots$ until it finds the vertex that has an outgoing (to the right) edge labeled "z". The endpoint of that edge is the next position of the algorithm (next state of the automaton).

It remains to show how to compute the values of the function l for all tree vertices. This is done as before using the values of l for shorter strings to compute the next value of l. Therefore, we should consider all tree vertices in order of increasing length. It is easy to see that this can be done in the required time. (Please note that the constant in the upper bound for the running time depends on the cardinality of the alphabet.) For a discussion of the methods used to store the tree, see section 9. ∎

The general question arises: Which properties of strings can be tested using finite automata? It turns out that there is an easily defined class of patterns that correspond to finite automata. These patterns are called "regular expressions".

Definition. Let Γ be a finite alphabet. We assume that Γ does not contain six symbols Λ, ε, (,), $*$ and | (these symbols will be used for constructing regular expressions; therefore, we should not mix them with letters from Γ). *Regular expressions* are constructed according to the following rules:

(a) any letter from Γ is a regular expression;

(b) the symbols Λ, ε are regular expressions;

(c) if $A, B, C, \dots, E$ are regular expressions, then $(ABC\dots E)$ is a regular expression;

(d) if $A, B, C, \dots, E$ are regular expressions, then $(A \mid B \mid C \mid \dots \mid E)$ is a regular expression;

(e) if A is a regular expression, then $A*$ is a regular expression.

Each regular expression defines a set of strings (composed of characters from Γ) according to the following rules:

(a) A letter corresponds to a singleton whose element is a one-character string containing this letter;

(b) The symbol ε corresponds to the empty set; the symbol Λ corresponds to the singleton whose element is the empty string;

(c) the regular expression $(ABC \ldots E)$ corresponds to the set of all strings obtained as follows: take a string from the set that corresponds to A, a string from the set that corresponds to B, to $C, \ldots,$ and to E and concatenate all those strings in the given order (*concatenation* of sets);

(d) the regular expression $(A \mid B \mid C \mid \ldots \mid E)$ corresponds to the union of the sets that correspond to expressions $A, B, C, \ldots, E$;

(e) the regular expression $A*$ corresponds to the *iteration* of a set corresponding to A, that is, to the set of all strings that may be cut into pieces in such a way that each piece belongs to the set corresponding to A. (In particular, the set corresponding to $A*$ always contains the empty string.)

Sets that correspond to regular expressions are called *regular* sets. Here are several examples:

Expression	**Set**
(a|b)*	All strings composed of a and b
(aa)*	All strings of even length composed of as, including the empty string
(Λ|a|b|aa|ab|ba|bb)	all strings of length at most 2 composed of a and b

10.7.5. Find a regular expression corresponding to the set of all strings composed of a and b that contain an even number of as.

Solution. The expression b* defines the set of all strings without a; the expression (b* a b* a b*) defines the set of all words with exactly two as. It remains to take the union of these two sets and then to apply iteration:

((b* a b* a b*) | b*)*

Another possible answer:

((b* a b* a)* b*) ■

10.7.6. Write a regular expression that defines a set of strings composed of a, b, c having bac as a substring.

Solution. ((a|b|c)*bac (a|b|c)*) ■

Remark. A more difficult problem is to write the expression for the *complement* of this set, that is, the set of all strings composed of a, b, c that do *not* have bac as a substring. This is possible, however, as we'll see below.

Now the general pattern-matching problem may be stated as follows: check whether an input string belongs to the set corresponding to a given regular expression.

10.7.7. What regular expressions are equivalent to the patterns a?b and ab*cd used as examples earlier? (Please note that the symbol * in the pattern ab*cd has a completely different meaning compared to its use in regular expressions.) We assume that the alphabet is {a, b, c, d, e}.

Solution.

```
((a|b|c|d|e)*a(a|b|c|d|e)b(a|b|c|d|e)*)
```

```
((a|b|c|d|e)*ab(a|b|c|d|e)*cd(a|b|c|d|e)*)
```

■

10.7.8. Prove that for any regular expression there exists a finite automaton that recognizes the corresponding set of strings.

Solution. To prove this, we need the notion of a *nondeterministic finite automaton*. Consider a directed graph containing several points (vertices) and some arrows (edges) connecting those points. Assume that some of the edges are labeled by letters (from a given alphabet) and some edges remain unlabeled. Assume also that two vertices are selected; one is called the *initial* vertex I and the other is called the *final* vertex F. Such a labeled graph is called a nondeterministic finite automaton.

Let us consider all the paths from I to F. Going along a path, we read all the letters (on labeled edges). Therefore, each path from I to F determines a string. The automaton as a whole determines a set of strings, namely, the set of all strings that can be read along some path from I to F. We say that these strings are *accepted* by the automaton.

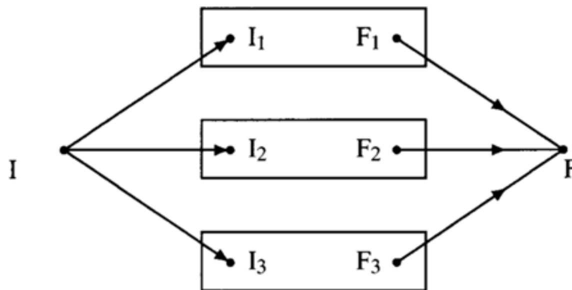
Remark. If we draw the states of a finite automaton as points and the transitions as labeled edges, it is clear that finite automata are special cases of nondeterministic finite automata. They are distinguished by the following requirements: (a) all edges are labeled except for the edges directed to the final vertex; (b) for each vertex and for each letter there is exactly one outgoing edge labeled by this letter.

We transform a regular expression into a finite automaton in two stages. First, we construct a nondeterministic finite automaton that corresponds to the same set. Then for any nondeterministic finite automaton we construct an equivalent deterministic finite automaton.

10.7.9. A regular expression is given. Construct a nondeterministic finite automaton that corresponds to the same set.

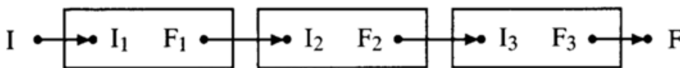
Solution. This automaton is constructed inductively, following the definition of a regular expression. If the regular expression is a letter or ε, the corresponding automaton has one edge. If the regular expression is Λ, the automaton has no

edges at all. A union is implemented as follows:

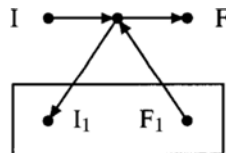


Here the picture for the union of three sets is drawn. The rectangles show the corresponding nondeterministic finite automata; their initial and final vertices are shown. New arrows (there are six of them) are unlabeled.

Concatenation corresponds to the following picture:



Finally, iteration corresponds to the picture



■

10.7.10. A nondeterministic finite automaton N is given. Construct an equivalent deterministic finite automaton (or a program with a finite number of states) that checks if an input string x is accepted by N (that is, if x can be read on a path from I to F).

Solution. The states of the deterministic automaton are *sets* of vertices of the nondeterministic automaton. After a prefix X of the input string is read, the state $s(X)$ of the deterministic automaton is the set of all vertices that are reachable from I along paths carrying the string X on it. In other words, consider all paths starting from I. Each path determines a string that can be read along it. If the string is X, include the end of the path into $s(X)$. ■

The two-stage construction of a finite automaton corresponding to a given regular expression, is finished. ■

It turns out that regular expressions, deterministic finite automata, and nondeterministic finite automata define the same class of sets. To prove this, it remains to solve the following problem:

10.7.11. A nondeterministic finite automaton is given. Construct a regular expression that defines the same set.

Solution. Assume that the nondeterministic automaton has vertices $1, \dots, k$, where 1 is its initial vertex and k is its final vertex. By $D(i, j, s)$ we denote the set of all strings read along all the paths from i to j if only $1, 2, \dots, s$ are allowed as intermediate path vertices. By definition, the automaton itself corresponds to the set $D(1, k, k)$.

We prove by induction over s that all sets $D(i, j, s)$ for all i and j are regular. For $s = 0$, this is obvious (intermediate vertices are not permitted, therefore each set is a finite set whose elements are strings of length not exceeding 1).

Which strings are elements of $D(i, j, s + 1)$? Let us consider a path from i to j and mark all the steps when it enters the $(s + 1)$-th vertex. The marked steps split our path into several paths that do not use $s + 1$ as an intermediate vertex. This argument leads to the equation

$$D(i, j, s + 1) = D(i, j, s) \mid (D(i, s + 1, s)\ D(s + 1, s + 1, s)*\ D(s + 1, j, s))$$

(here the notation for regular expressions is used for sets). It remains to apply the induction assumption. ∎

10.7.12. Where have you seen a similar argument?

Solution. In the Floyd algorithm for the shortest path (see section 9, p. 125). ∎

10.7.13. Prove that the class of sets corresponding to regular expressions remains the same if we agree to use not only set union but also complementation (and therefore set intersection, since it can be expressed using set union and complement).

Solution. For the deterministic finite automata the transition from a set to its complement is evident. ∎

Remark. From a practical point of view, things are not so easy. The problem is that the transition from a nondeterministic automaton to a deterministic one may exponentially increase the number of states. There are many theoretical and practical questions concerning this problem. See the book of Aho, Sethi, and Ullman on compilers [2].

11 Set representation. Hashing

11.1 Hashing with open addressing

In section 6 we considered several representations for sets whose elements are integers of arbitrary size. However, all those representations are rather inefficient: at least one of the operations (membership test, adding or deleting an element) runs in time proportional to the number of elements in the set. This is unacceptable in almost all practical applications.

It is possible to find a set representation where all three operations mentioned run in time $C \log n$ (in the worst case). One such representation is considered in the next section. In this section, we consider another set representation that may require n operations in the worst case but is very efficient "in a typical case". The method is called "hashing".

Suppose we want to store a set of elements of type `T`, where the number of elements is guaranteed to be less than `n`. Choose a function `h` that is defined on elements of type `T` and whose values are integers in the range `0..n-1`. It is desirable that this function have different values for different elements of the set we are trying to represent (the worst case is when all the function values are the same). This function is called a *hash function*.

Our representation uses two arrays

```
val:  array [0..n-1] of T;
used: array [0..n-1] of Boolean;
```

(we write `n-1` in the type definition, though it is not permitted in Pascal). The set consists of `val [i]` for all `i` such that `used [i]` is true. (The values `val [i]` are all different.) When possible, we store an element `t` at position `h(t)`, which is considered a "natural place" for `t`. However, it may happen that a new element `t` appears whose place `h(t)` is already used by another element (that is, `used[h(t)]` is true). In this case, we search to the right looking for the first unused place and put the element `t` there. (Here "to the right" means that the index increases; when we reach `n-1`, the index wraps around.) Recall that we assume that the number of elements is always less than the number of places, therefore free places do exist.

Formally speaking, the invariant relation that we maintain is the following: For any element, the interval between its natural place and its actual place is filled completely.

This invariant makes the membership test easy. Suppose we want to check if an element `t` is in the set. We find the natural place for `t` and then go to the right until we find an empty slot or `t`. In the first case, the element `t` is not in the set (a consequence of our invariant); in the second case, the element is in the set. If it is absent, we may add it (filling the unused place found). If not, we can delete it by putting `False` in the corresponding cell of the `used` array.

11.1.1. The last passage has a severe error. Find it and correct it.

Solution. The delete operation implemented as described can destroy the invariant and create an empty position between the natural and actual positions of some element. We should be more careful. After a gap appears, we move from left to right until we find either an element that is not at its natural place or another gap. If the gap appears first, we have nothing to worry about. If an element is found not at its natural place, we check whether it needs to be moved to the gap we've created. If not, we continue our search. If yes, we move the element found to the gap. A new gap appears which we deal with in the same way. ■

11.1.2. Write the programs for membership test, adding and deleting elements.

Solution.

```
function is_element (t: T): Boolean;
| var i: integer;
begin
| i := h (t);
| while used [i] and (val [i] <> t) do begin
| | i := (i + 1) mod n;
| end; {not used [i] or (val [i] = t)}
| is_element := used [i] and (val [i] = t);
end;

procedure add (t: T);
| var i: integer;
begin
| i := h (t);
| while used [i] and (val [i] <> t) do begin
| | i := (i + 1) mod n;
| end; {not used [i] or (val [i] = t)}
| if not used [i] then begin
| | used [i] := true;
| | val [i] := t;
| end;
end;

procedure delete (t: T);
| var i, gap: integer;
begin
| i := h (t);
| while used [i] and (val [i] <> t) do begin
| | i := (i + 1) mod n;
| end; {not used [i] or (val [i] = t)}
| if used [i] and (val [i] = t) then begin
| | used [i] := false;
| | gap := i;
```

```
    i := (i + 1) mod n;
    {gap may be filled by one of i,i+1,...}
    while used [i] do begin
      if i = h (val[i]) then begin
        {i is the natural place, nothing to do}
      end else if dist(h(val[i]),i) < dist(gap,i) then begin
        {gap...h(val[i])...i, nothing to do}
      end else begin
        used [gap] := true;
        val [gap] := val [i];
        used [i] := false;
        gap := i;
      end;
      i := (i + 1) mod n;
    end;
  end;
end;
```

Here `dist (a,b)` is the distance from a to b measured clockwise, that is,

$$\texttt{dist (a,b)} = (\texttt{b - a + n}) \bmod \texttt{n}.$$

(We add n, because mod works best when the dividend is positive.) ∎

11.1.3. There are many versions of hashing. For example, when we find that the natural place (say, i) is occupied, we look for a free place not among $i+1, i+2, \ldots$, but among $r(i), r(r(i)), r(r(r(i))), \ldots$ where r is some mapping of the set $\{0, \ldots, n-1\}$ into itself. What are the possible problems?

Answer. (1) We cannot guarantee that free space will be found even if we know it exists. (2) It is not clear how to fill gaps after deleting an element. (In many practical cases, deletion is not necessary, so this approach is sometimes used. The idea is that a careful choice of the function r will prevent the appearance of big "clusters" of occupied cells.) ∎

11.1.4. Suppose hashing is used to store the set of all English words (say, for a spelling checker). What should we add to the data to be able to find Russian translations of all English words?

Solution. The array `val` (whose elements are English words) should be extended by a parallel array `rval` of their translations: if `used[i]` is true, `rval[i]` is a translation of `val[i]` ∎

11.2 Hashing using lists

A hash function with k values is a tool that reduces the storage problem for one large set to a storage problem for k small sets. Indeed, after a hash function with

k values is chosen, any set is split into k subsets corresponding to the k different values of the hash function. (Some of them may be empty.) If we want to perform a membership test or an add/delete operation, we compute the hash function value and determine for which of the k sets the operation should be performed.

These smaller sets may be stored conveniently using references, because we know the total size of all the sets but not their individual sizes. The following problem suggests an implementation.

11.2.1. Suppose the values of hash function `h` are `1..k`. For any number `j` in `1..k`, consider a list of all set elements `z` such that `h(z)` = `j`. Let us store those `k` lists using the variables

```
Content: array [1..n] of T;
Next: array [1..n] of 1..n;
Free: 1..n;
Top: array [1..k] of 1..n;
```

in the same way as we did for `k` stacks of limited size (p. 83). Write the corresponding procedures. (Please note that deletion is now easier than in the open addressing case.)

Solution. We start with `Top[i]` = 0 for all `i` = `1..k`. All the positions are linked in a free list as follows: `Free` = 1; `Next[i]` = `i+1` for `i` = `1...n-1`; `Next[n]` = 0.

```
function is_element (t: T): Boolean;
| var i: integer;
begin
| i := Top[h(t)];
| {we should search in the list starting from i}
| while (i <> 0) and (Content[i] <> t) do begin
| | i := Next[i];
| end; {(i=0) or (Content [i] = t)}
| is_element := (i<>0) and (Content[i]=t);
end;

procedure add (t: T);
| var i: integer;
begin
| if not is_element (t) then begin
| | i := Free;
| | {Free<>0; we assume that the size limit is not reached}
| | Free := Next[Free];
| | Content[i]:=t;
```

```
    |  Next[i]:=Top[h(t)];
    |  Top[h(t)]:=i;
  end;
end;

procedure delete (t: T);
| var i, pred: integer;
begin
  i := Top[h(t)]; pred := 0;
  {we should search in the list starting from i;
    pred is a predecessor of i in the list
    (if exists; otherwise 0)}
  while (i <> 0) and (Content[i] <> t) do begin
  | pred := i; i := Next[i];
  end; {(i=0) or (Content[i] = t)}
  if i <> 0 then begin
    {Content[i]=t, the element exists
      and should be deleted}
    if pred = 0 then begin
    | {this is the first element in the list}
    | Top[h(t)] := Next[i];
    end else begin
    | Next[pred] := Next[i]
    end;
    {it remains to return i to the free list}
    Next[i] := Free;
    Free:=i;
  end;
end;
```

■

11.2.2. (Requires some probability theory) Suppose a hash function with k values is used to store a set of cardinality n. Prove that the expected number of operations in the preceding problem does not exceed $C(1 + n/k)$, if the element `t` is taken at random in such a way that all values of $h(t)$ are equiprobable (have probability $1/k$).

Solution. Let $l(i)$ be the length of the list corresponding to the hash value i. The number of operations does not exceed $C(1 + l(h(t)))$; the expectation does not exceed $C(1 + n/k)$, since $\sum_i l(i) = n$. ■

This estimate is based on the assumption that all values of $h(t)$ have the same probability. However, for a given input distribution and a given hash function this assumption may be false, and many elements of the set may share the same value of the hash function, so large clusters appear. A method that avoids this difficulty is called *universal hashing*.

The idea is to use a family of hash functions instead of just one and to choose a function from this family at random. The hope is that any fixed set behaves well for most of the functions in the family.

Let H be a family of functions. Each function maps the set T into a set of cardinality k (say, into $0, \ldots, k-1$). The family H is called a *universal family of hash functions* if for any two distinct elements $s, t \in T$, the probability of the event $h(s) = h(t)$ (for a random function $h \in H$) is equal to $1/k$. (In other words, the functions $h \in H$ satisfying $h(s) = h(t)$ are in proportion $1/k$ with all functions in H.)

Remark. A stronger requirement may be given, namely, we may require that for any two distinct elements $s, t \in H$, the values $h(s)$ and $h(t)$ (for a randomly chosen h) are independent random variables uniformly distributed among $0, \ldots, k-1$. This stronger requirement is fulfilled in the examples below.

11.2.3. Let $t_1, \ldots, t_n$ be any sequence of distinct elements of the set T. Consider the sequence of events that occurs when the elements $t_1, \ldots, t_n$ are added to a set stored using a hash function h from a universal family H. Prove that the expected number of operations (the mean value taken over all $h \in H$) does not exceed $Cn(1 + n/k)$.

Solution. By m_i we mean the number of elements among $t_1, \ldots, t_n$ with hash value i. (Of course, the numbers $m_0, \ldots, m_{k-1}$ depend on h.) The number of operations we are interested in is equal to $m_0^2 + m_1^2 + \cdots + m_{k-1}^2$ up to a constant factor. (Indeed, if s elements are placed in a list, the number of operations is approximately $1 + 2 + \cdots + s \sim s^2$.) The same sum of squares may be written as the number of pairs $\langle p, q \rangle$ satisfying $h(t_p) = h(t_q)$. For any fixed p and q the event $h(t_p) = h(t_q)$ has probability $1/k$ (assuming that $p \neq q$). Therefore, the expected value of the corresponding term is equal to $1/k$, and the expected value of the sum is roughly n^2/k. More precisely, we obtain $n + n^2/k$ since we need to count terms with $p = q$. ∎

This problem shows that the average number of operations per element is $C(1 + n/k)$. Here n/k may be called the "average load of a hash value".

11.2.4. Prove a similar assertion about the arbitrary sequence of additions, deletions, and membership tests (not only additions, as in the preceding problem).

[Hint. Let us imagine that while performing addition, search, or deletion, the element is a person that traverses the list of its colleagues with the same hash value until it finds its twin brother (an equal element; it this case, the element disappears) or reaches the end of the list. By i-j-meeting we mean the event when elements t_i and t_j meet each other. (It may or may not happen depending on h.) The total number of operations is (up to a constant factor) equal to the number of meetings plus the number of elements. When $t_i \neq t_j$, the probability of an i-j-meeting does not exceed $1/k$. It remains to count the meetings of equal elements. Let us fix some value $x \in T$ and consider all operations that refer to this value. They follow the pattern: tests, addition, tests, deletion, tests, addition, etc. The meetings

occur between an added element and tested elements that follow it (up to the next deletion, and including it), therefore the total number of meetings does not exceed the number of elements equal to x.] ■

Now we give several examples of universal families. For any two finite sets A and B, the family of all functions that map A into B is an universal family. However, from a practical viewpoint this family is useless, since to store a random function from this family, we need an array with $\#A$ elements ($\#A$ is the cardinality of A). If we can afford an array of that size, we do not need hashing at all!

More practical examples of universal families may be obtained using simple algebraic techniques. By Z_p we denote the set of all residues modulo p where p is a prime number, that is, the set $\{0, 1, \ldots, p - 1\}$. Arithmetic operations are performed on this set modulo p. An universal family is formed by all linear functionals defined on Z_p^n with values in Z_p. More precisely, let $a_1, \ldots, a_n$ be arbitrary elements of Z_p and consider the mapping

$$h : \langle x_1, \ldots, x_n \rangle \mapsto a_1 x_1 + \cdots + a_n x_n$$

We get a family of p^n mappings $Z_p^n \to Z_p$ indexed by n-tuples $\langle a_1, \ldots, a_n \rangle$.

11.2.5. Prove that this family is universal.

[Hint. Let x and y be distinct points of the space Z_p^n. What is the probability of the event "a random functional α has the same values for x and y"? In other words, what is the probability of the event "$\alpha(x - y) = 0$"? The answer is provided by the following statement. If u is a nonzero vector, all possible values of $\alpha(u)$ are equiprobable.] ■

In the following problem, the set $B = \{0, 1\}$ is taken to be the set of residues modulo 2.

11.2.6. Show that the family of all linear mappings of B^n into B^m is universal. ■

Hashing turns out to be useful in unexpected curcumstances. The following example was communicated to me by D.V. Varsonofiev. Suppose we want to contruct a spelling checker to find (most of) the typos in an English text. We do not want, however, to keep a list of all correct words (in all grammatic forms). We can use the following trick. Choose some positive integer N and functions $f_1, \ldots, f_k$ that map words to $1, \ldots, N$. Consider an array of N bits initially set to zero. Then for any (correctly spelled) word x, compute the values $f_1(x), \ldots, f_k(x)$ and make the corresponding bits equal to 1. (Some bits may correspond to several words.) Then the approximate test to check whether a string z is a correctly spelled word, is as follows. Compute all values $f_1(z), \ldots, f_k(z)$ and check that all the corresponding bits are 1s. This test may miss some errors, but all correct words will be allowed.

12 Sets, trees, and balanced trees

12.1 Set representation using trees

Full binary trees and T-trees

Draw a point. Now draw two arrows going up-left and up-right to two other points. From those two points also draw two arrows, etc. The resulting tree is called a *full binary tree* (the n-th level has 2^{n-1} points). The initial point (at the bottom of the tree) is called the *root*. Each vertex has two *sons* (arrows point to them), the *left* son and the *right* son. Each vertex (except for the root) has an unique *father*.

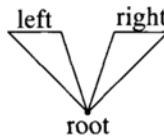
Please note that many textbooks draw trees with the root at the top and also use words "child" ("parent", "siebling", etc.) instead of "son" ("father", "brother", etc.).

Now choose some subset of the set of all vertices of the full binary tree. It should satisfy the following requirement: for each vertex of the subset, its father belongs to the subset, too. (Therefore, all vertices on a path from the root to some vertex from the subset belong to the subset.) Assume that each vertex in the subset has a label that is an element of some set T. (In other words, we assume that a mapping from the subset into the set T is given.) Such a subset with labels from T is called a T*-tree*. The set of all T-trees is denoted by $\mathrm{Tree}(T)$.

The notion of T-tree may be defined recursively. Any nonempty T-tree is divided into three parts: the root (which carries a label from T), the left sub-tree, and the right subtree (one or both of which may be empty). Therefore, there is an one-to-one correspondence between the set of nonempty T-trees and the product $T \times \mathrm{Tree}(T) \times \mathrm{Tree}(T)$. We get the following equality:

$$\mathrm{Tree}(T) = \{\mathbf{empty}\} + T \times \mathrm{Tree}(T) \times \mathrm{Tree}(T).$$

(here **empty** stands for the empty tree).

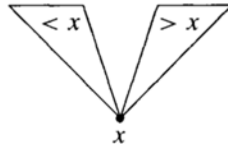


Subtrees and height

Assume that some T-tree is fixed. For any vertex x, the following objects are defined: the *left subtree* (the left son of x and all its descendants); the *right subtree* (the right son of x and all its descendants); and the *subtree rooted at* x (the vertex x and all its descendants). The left and right subtrees of x may be empty, but the subtree rooted at x may not (it always contains the vertex x). The *height* of a subtree is defined as the maximal length of the sequence $y_1, \ldots, y_n$ of its vertices where y_{i+1} is a son of y_i for all i, minus one. (The height of the empty tree is -1 by definition; the height of a tree containing only the root is 0.)

Ordered T-trees

Assume that a linear order is defined on the set T. A T-tree is *ordered* if the following requirement is fulfilled: for any vertex x, all labels in its left subtree are less than the label at x and all labels in its right subtree are greater than the label at x.



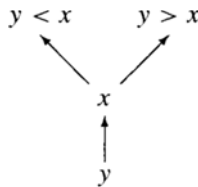
12.1.1. Prove that all labels in an ordered subtree are different.

[Hint. Induction over the height of the tree.] ■

Set representation using trees

Consider any tree as a representation of the set of labels of its vertices. (Of course, the same set may have different representations.)

If the set is ordered, each element can easily find its way to a place in the tree. It starts from the root; coming to a vertex, an element compares itself with the label at that vertex and decides whether to go to the left or to the right.



Using this rule, the element either finds the identical label already present in the tree or the place where it should stay to keep the tree ordered.

In this section, assume that the set T is a linearly ordered set. All T-trees we consider are ordered.

Tree representation

The simplest way to represent a tree is to identify the vertices of a full binary tree with integers $1, 2, 3, \ldots$ (the left son of n is $2n$, the right son of n is $2n + 1$) and store the labels in an array `val [1..N]` (for a large enough `N`). However, this approach wastes space because space is set aside for positions in the full binary tree that are not filled in a specific T-tree.

The following approach is more space efficient. We use three arrays

```
val: array [1..n] of T;
left, right: array [1..n] of 0..n;
```

(n is the maximal possible number of tree vertices for trees we want to store) and a variable `root:0..n`. Each vertex of the stored T-tree will have a number that is an integer in `1..n`. Different vertices have different numbers; some numbers may be unused. The label of the vertex with number `x` is stored in `val[x]`. The root has number `root`. If vertex `i` has sons, their numbers are `left[i]` and `right[i]`. Nonexistent sons are replaced by the number 0. Similarly, the condition `root` $= 0$ means that the tree is empty.

The tree vertices only occupy part of the array. For "free" values of `i` that are not used as vertex numbers, the values `val[i]` have no meaning. We want the free numbers to be "linked in a list"; the first free number is stored in a special variable `free:0..n`, while the free number that follows `i` in the list is `left[i]`. In other words, the list of all free numbers is

```
free, left[free], left[left[free]],...
```

For the last free number `i` in the list, the value `left[i]` equals 0. If `free` $= 0$, there are no free numbers. (Remark. We used the array `left` to link all free numbers in a list but of course, we may use the array `right` instead.)

We can use any other integer outside `1..n` to indicate the absence of a vertex (instead of 0). To stress this, we use a symbolic constant `null` $= 0$ instead of the numeral 0.

12.1.2. Write a procedure that checks if an element `t:T` is present in an ordered tree (as described above).

Solution.

```
if root = null then begin
| ..t is not in the tree
end else begin
|  x := root;
|  {invariant: it remains to check if t is present in
|    a nonempty subtree rooted at x}
|  while ((t < val [x]) and (left [x] <> null)) or
|  |     ((t > val [x]) and (right [x] <> null)) do begin
|  |  if t < val [x] then begin {left [x] <> null}
|  |  | x := left [x];
|  |  end else begin {t > val [x], right [x] <> null}
|  |  | x := right [x];
|  |  end;
|  end;
|  {either t = val [x] or t is not in the tree}
|  ..answer is (t = val [x])
end;
```

■

12.1.3. Simplify the procedure using the following trick. Extend the array `val`, adding a cell with index `null`. Let `val[null]` be `t`.

Solution.

```
val [null] := t;
x := root;
while t <> val [x] do begin
| if t < val [x] then begin
| | x := left [x];
| end else begin
| | x := right [x];
| end;
end;
..answer is (x <> null).
```

■

12.1.4. Write a procedure that adds an element t to a set represented as an (ordered) T-tree. (If t is already present, nothing should be done.)

Solution. The procedure get_free (var i:integer) produces a free integer i in 1..n (that is, an integer that is not a number of any vertex) and updates the free list. (For simplicity, we assume that free integers exist.)

```
procedure get_free (var i: integer);
begin
| {free <> null}
| i := free;
| free := left [free];
end;
```

Using this procedure, we write:

```
if root = null then begin
| get_free (root);
| left [root] := null; right [root] := null;
| val [root] := t;
end else begin
| x := root;
| {invariant: it remains to add t to a (nonempty) subtree
|  rooted at x}
| while ((t < val [x]) and (left [x] <> null)) or
| |     ((t > val [x]) and (right [x] <> null)) do begin
| | if t < val [x] then begin
| | | x := left [x];
| | end else begin {t > val [x]}
| | | x := right [x];
| | end;
| end;
```

```
  if t <> val [x] then begin {t is not in the tree}
    get_free (i);
    left [i] := null; right [i] := null;
    val [i] := t;
    if t < val [x] then begin
      left [x] := i;
    end else begin {t > val [x]}
      right [x] := i;
    end;
  end;
end;
```

∎

12.1.5. Write a procedure that deletes an element t from a set represented as an ordered tree. (If the element is not in the set, nothing should be done.)

Solution.

```
if root = null then begin
  {the tree is empty, there is nothing to do}
end else begin
  x := root;
  {it remains to delete t from the subtree rooted at x;
   since it may require changes in the father node,
   we introduce the variables  father: 1..n and
   direction: (l, r) with the following
   invariant: if x is not the root, then father
   is (the number of) x's father node, direction is
   equal to l/r if x is the left/right son of its father}
  while ((t < val [x]) and (left [x] <> null)) or
        ((t > val [x]) and (right [x] <> null)) do begin
    if t < val [x] then begin
      father := x; direction := l;
      x := left [x];
    end else begin {t > val [x]}
      father := x; direction := r;
      x := right [x];
    end;
  end;
  {t = val [x] or t is not in the tree}
  if t = val [x] then begin
    ..delete the node x with a known father and direction
  end;
end;
```

The deletion of a vertex uses the procedure

```
procedure make_free (i: integer);
begin
| left [i] := free;
| free := i;
end;
```

which adds the number i to the free list. While deleting a vertex, we should distinguish between four cases depending on whether the vertex has a left/right son or not.

```
if (left [x] = null) and (right [x] = null) then begin
| {x is a leaf, no sons}
| make_free (x);
| if x = root then begin
| | root := null;
| end else if direction = l then begin
| | left [father] := null;
| end else begin {direction = r}
| | right [father] := null;
| end;
end else if (left[x]=null) and (right[x] <> null) then begin
| {when x is deleted, right[x] occupies its place}
| make_free (x);
| if x = root then begin
| | root := right [x];
| end else if direction = l then begin
| | left [father] := right [x];
| end else begin {direction = r}
| | right [father] := right [x];
| end;
end else if (left[x] <> null) and (right[x]=null) then begin
| ..the symmetrical code
end else begin {left [x] <> null, right [x] <> null}
| ..delete a vertex with two sons
end;
```

The deletion of a vertex with two sons is the most difficult case. Here we should exchange it with an immediately following vertex (in the sense of label ordering).

```
y := right [x]; father := x; direction := r;
{now father and direction refer to vertex y}
while left [y] <> null do begin
| father := y; direction := l;
| y := left [y];
end;
```

```
{val[y] is minimal element of the set larger
 than val[x], y has no left son}
val [x] := val [y];
..delete the vertex y (we already know how to do
  that for a vertex without the left son)
```
■

12.1.6. Simplify the deletion procedure using the following observation: Some cases (say, the first two) may be combined into a single case. ■

12.1.7. Use an ordered tree to store a function whose domain is a finite subset of T and whose range is some set U. The operations are: find the value of the function for a given argument; change this value; delete an element from the domain; and add an element to the domain (the value is also provided).

Solution. We represent the domain using an ordered tree and add one more array

```
func_val: array [1..n] of U;
```

If `val[x] = t` and `func_val[x] = u`, then the function value on `t` equals `u`. ■

12.1.8. Assume that we want to find the k-th element of a set (according to the ordering on T) in time limited by $C \cdot$ (tree height). What additional information do we need to store at the tree vertices?

Solution. At each vertex, we store the number of its descendants. When a vertex is added or deleted, this information must be updated along a path from the root to the new/deleted vertex. While searching for the k-th vertex, we maintain the following invariant: the vertex in question is the s-th vertex (according to the T-ordering) of a subtree rooted at x (here s and x are variables). ■

Running time

All of the procedures discussed above (membership test, addition, and deletion) run in time $C \cdot$ (tree height). For a "well-balanced" tree where all leaves have approximately the same height, the tree height is close to the logarithm of the number of vertices. However, for a unbalanced tree the height may be much larger. In the worst case, the vertices may form a chain (if all vertices have no left son, for example) and the tree height is the number of vertices. This happens if we start with the empty set and add elements in increasing order. However, one can prove that if the elements are added in random order, then the expected height of the tree will not exceed C log(tree size). If this "average bound" is not good enough for our application, we must spend additional effort to keep the tree "balanced". This is explained in the next section.

12.2 Balanced trees

A tree is called *balanced* (or an AVL-tree, in the honor of the inventors of this algorithm, G.M. Adelson-Velsky and E.M. Landis) if for any vertex, the heights of the left and the right subtrees differ by at most 1. (In particular, the only son of a vertex is required to be a leaf, since the height of the other subtree is -1.)

12.2.1. Find the minimal and maximal number of vertices in a balanced tree of height n.

Solution. The maximal number of vertices is equal to $2^{n+1} - 1$. If m_n is the minimal number of vertices, then $m_{n+2} = 1 + m_n + m_{n+1}$. An easy induction argument gives $m_n = \Phi_{n+3} - 1$ (where Φ_n is the n-th Fibonacci number: $\Phi_1 = 1$, $\Phi_2 = 1$, and $\Phi_{n+2} = \Phi_n + \Phi_{n+1}$). ■

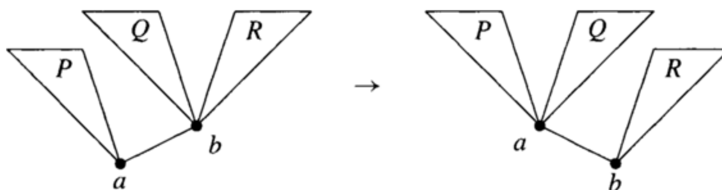
12.2.2. Prove that a balanced tree with $n > 1$ vertices has height at most $C \log n$ for some constant C that does not depend on n.

Solution. By induction over n, we prove that $\Phi_{n+2} \geq a^n$ where a is the larger root of the quadratic equation $a^2 = 1 + a$, that is, $a = (\sqrt{5} + 1)/2$. (This number is usually called "the golden mean".) It remains to apply the preceding problem.

■

Rotations

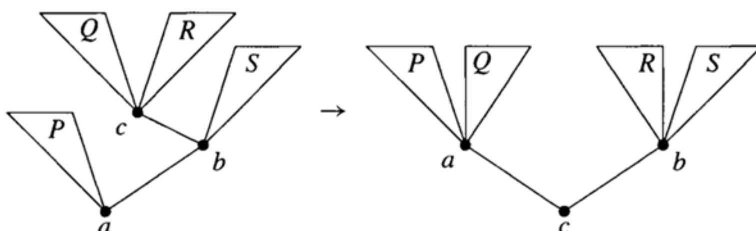
After an element is added or deleted, the tree may become unbalanced, and we have to restore the balance. Therefore, we need some tree transformations that preserve the set of labels and the ordering requirement, but help to balance the tree. Here are some of those transformations:



Assume that a vertex a has a right son b. Let P be the left subtree of a. Let Q and R be the left and right subtrees of b, respectively. The ordering requirement guarantees that $P < a < Q < b < R$. (This means that any label in P is smaller than a, that a is smaller than any label in Q, etc.) The same condition is imposed by the ordering requirements for another tree. The latter tree has root b; the left son a of the root has left subtree P and right subtree Q; the right subtree of the root is R. Therefore the first tree may be transformed to the second one without changing the set of labels or violating the ordering requirements. This transformation is called a

small right rotation. It is called "right" because there is a symmetric "left" rotation; it is called "small" because there exists a "big" rotation, which we describe now.

Let b be the right son of the root vertex a; let c be the left son of b; let P be the left subtree of a; let Q and R be the left and the right subtrees of c, respectively; and finally, let S be the right subtree of b. Then $P < a < Q < c < R < b < S$.

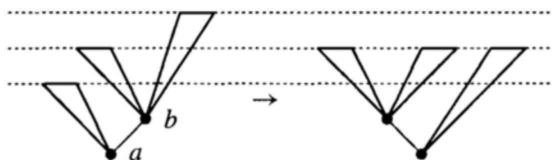


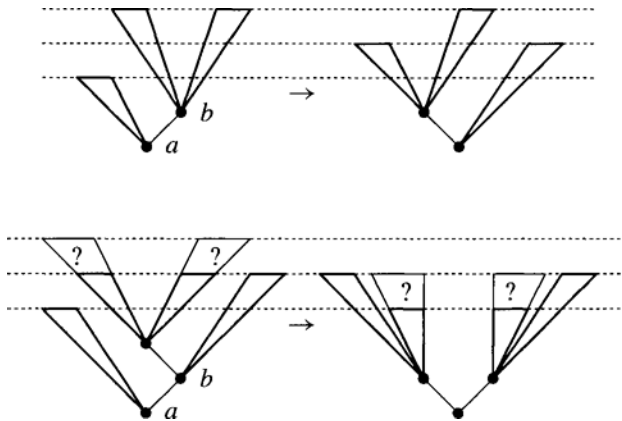
The same ordering conditions are imposed by a tree with root c, its left son a and right son b that have the left and the right subtrees P and Q (for a) and R and S (for b). The corresponding transformation is called a *big right rotation*. (A *big left rotation* is defined in a symmetric way.)

How to balance a tree using rotations

12.2.3. Suppose a tree is balanced everywhere except at the root where the difference of heights between the left and right subtrees equals 2 (that is, the left and right subtrees are balanced and their heights differs by 2). Prove that this tree may be transformed into a balanced tree using one of the four transformations mentioned above and that the height remains the same or decreases by 1 after the transformation.

Solution. Assume, for example, that the left subtree has smaller height, which we denote by k. Then the height of the right subtree is $k+2$. Denote the root of the tree by a. Let b be its right son (it does exist). Consider the left and right subtrees of the vertex b. One of them has height $k+1$, the other has height k or $k+1$. (Its height cannot be smaller than k because the right subtree of the root is balanced.) If the height of the left subtree of b is $k+1$, and the height of the right subtree of b is k, a big right rotation is needed; in all other cases, a small right rotation suffices. Here are the three possible cases:





■

12.2.4. A leaf is added to or deleted from a balanced tree. Prove that it is possible to make the tree balanced again using several rotations and that the number of rotations does not exceed the tree height.

Solution. We prove the more general statement:

Lemma. If a subtree Y of a balanced tree X is replaced by a balanced tree Z, and the heights of Y and Z differ by 1, then the resulting tree can be made balanced by several rotations. The number of rotations does not exceed the height where the change occurs (that is, where the root of Y and Z is located).

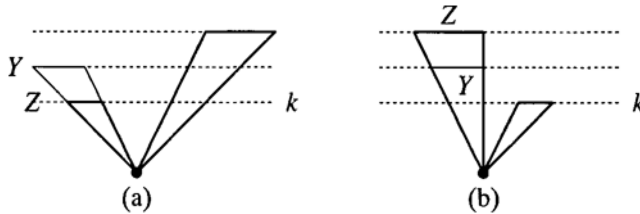
The addition/deletion of a leaf is a special case of the transformation mentioned in the lemma, therefore it is enough to prove this lemma.

Proof of the lemma. We use induction over the height where the change is made. If the change is made at the root, the entire tree is replaced; in this case, the lemma is evident because the tree Z is balanced. Assume that the replaced tree Y is, say, the left subtree of some vertex x. Two cases are possible:

1. After replacement, the balance condition at the vertex x is still valid. (However, the balance condition at the ancestors of x may be violated because the height of the subtree rooted at x may change.) In this case, we apply the induction hypothesis assuming that the replacement was done at the lower level and the whole tree rooted at x was replaced.

2. The balance condition at x is no longer valid. In this case, the height difference is 2 (it cannot be larger because the heights of Y and Z differ by at

most 1). Here two subcases are possible:



(a) The right subtree of x (the one that was not replaced) is higher. Assume that the height of the left subtree of x (i.e., Z) is k; then the height of the right subtree is $k+2$. The height of the old left subtree of X (i.e., Y) was $k+1$. The subtree of the initial tree rooted at x has height $k+3$ and its height does not change after replacement.

 By the preceding problem, a rotation can transform the subtree rooted at x into a balanced subtree of height $k+2$ or $k+3$. While doing this, the height of the subtree rooted at x (compared with its height before the transformation) did not change or was decreased by 1. Therefore, we apply the induction assumption.

(b) The left subtree of x is higher. Let the height of the left subtree (i.e., Z) be $k+2$; the right subtree has height k. The old left subtree of x (i.e., Y) was of height $k+1$. The subtree rooted at x (in the initial tree) has height $k+2$; after the replacement it has height $k+3$. After a suitable rotation (see the preceding problem), the subtree rooted at x becomes balanced and its height is $k+2$ or $k+3$; therefore, the change in height (compared with the height of the subtree of X rooted at x) does not exceed 1 and the induction assumption applies. ■

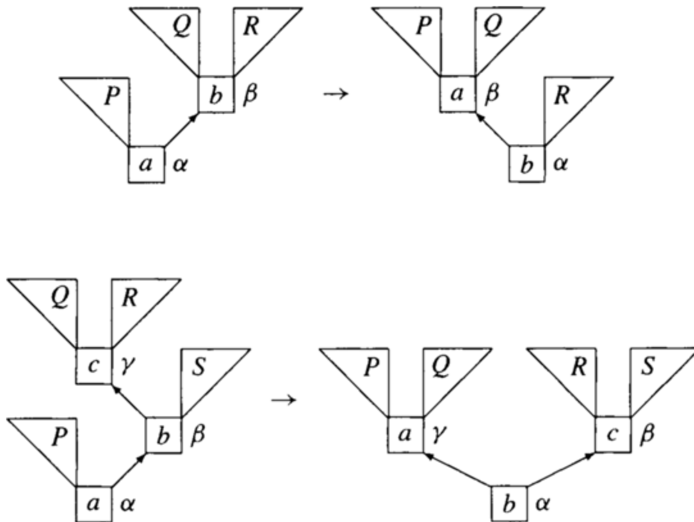
12.2.5. Write addition and deletion procedures that keep the tree balanced. The running time should not exceed $C\cdot$(tree height). It is allowed to store additional information (needed for balancing) at the vertices of the tree.

Solution. For each vertex we keep the difference between the heights of its right and left subtrees:

`diff [i]` = (the height of the right subtree of `i`) −
(the height of the left subtree of `i`).

We need four procedures that correspond to left/right, small/big rotations. Let us first make two remarks. (1) We want to keep the number of the tree root unchanged during the rotation. (Otherwise it would be necessary to update the pointer at the father vertex, which is inconvenient.) This can be done, because the numbers of tree vertices may be chosen independently of their content. (In our pictures, the

number is drawn near the vertex while the content is drawn inside it.)



(2) After the transformation, we should update values in the `diff` array. To do this, it is enough to know the heights of trees $P, Q, \ldots$ up to a constant (only differences are important), so we may assume that one of the heights is equal to 0.

Here are the rotation procedures:

```
procedure SR (a:integer); {small right rotation at a}
| var b: 1..n; val_a,val_b: T; h_P,h_Q,h_R: integer;
begin
| b := right [a]; {b <> null}
| val_a := val [a]; val_b := val [b];
| h_Q := 0; h_R := diff[b]; h_P := (max(h_Q,h_R)+1)-diff[a];
| val [a] := val_b; val [b] := val_a;
| right [a] := right [b] {subtree R}
| right [b] := left [b] {subtree Q}
| left [b] := left [a] {subtree P}
| left [a] := b;
| diff [b] := h_Q - h_P;
| diff [a] := h_R - (max (h_P, h_Q) + 1);
end;

procedure BR(a:integer);{big right rotation at a}
| var b,c: 1..n; val_a,val_b,val_c: T;
|     h_P,h_Q,h_R,h_S: integer;
```

```
begin
  b := right [a]; c := left [b]; {,c <> null}
  val_a := val [a]; val_b := val [b]; val_c := val [c];
  h_Q := 0; h_R := diff[c]; h_S := (max(h_Q,h_R)+1)+diff[b];
  h_P := 1 + max (h_S, h_S-diff[b]) - diff [a];
  val [a] := val_c; val [c] := val_a;
  left [b] := right [c] {subtree R}
  right [c] := left [c] {subtree Q}
  left [c] := left [a]  {subtree P}
  left [a] := c;
  diff [b] := h_S - h_R;
  diff [c] := h_Q - h_P;
  diff [a] := max (h_S, h_R) - max (h_P, h_Q);
end;
```

The (small and big) left rotations are similar. ■

The addition/deletion procedures are written as before, but now they have to update the `diff` array and restructure the tree to keep it balanced.

An auxiliary procedure with the following pre- and postconditions is used:

> **before:** the left and right subtrees of the vertex number a are balanced; the difference of heights at a is at most 2; the `diff` array is filled correctly for the subtree rooted at a;
>
> **after:** the subtree rooted at a is now balanced; the `diff` is updated (inside that subtree); the change in the height of the subtree rooted at a is stored in d and is equal to 0 or -1; the remaining part of the tree (including the `diff` array) remains unchanged.

```
procedure balance (a: integer; var d: integer);
begin {-2 <= diff[a] <= 2}
  if diff [a] = 2 then begin
    b := right [a];
    if diff [b] = -1 then begin
      BR (a); d := -1;
    end else if diff [b] = 0 then begin
      SR (a); d := 0;
    end else begin {diff [b] = 1}
      SR (a); d := - 1;
    end;
  end else if diff [a] = -2 then begin
    b := left [a];
    if diff [b] = 1 then begin
      BL (a); d := -1;
    end else if diff [b] = 0 then begin
```

```
|   |  | SL (a); d := 0;
|   |  end else begin {diff [b] = -1}
|   |  | SL (a); d := - 1;
|   |  end;
|  end else begin {-2 < diff [a] < 2, there is nothing to do}
|  | d := 0;
|  end;
end;
```

To restore the balance, we go downwards from a leaf to the root. To do that, we store the path from the root to the current vertex in a stack. The elements of the stack are pairs ⟨vertex, direction of move from the vertex⟩, that is, values of type

```
record
| vert: 1..n; {vertex}
| direction : (l, r); {l for left, r for right}
end;
```

The addition of an element t is now as follows:

```
if root = null then begin
| get_free (root);
| left[root] := null; right[root] := null; diff[root] := 0;
| val[root] := t;
end else begin
| x := root; ..make the stack empty
| {invariant: it remains to add t to the nonempty subtree
|  rooted at x; the stack contains the path to x}
| while ((t < val [x]) and (left [x] <> null)) or
| |     ((t > val [x]) and (right [x] <> null)) do begin
| | if t < val [x] then begin
| | | ..add <x, l> to the stack
| | | x := left [x];
| | end else begin {t > val [x]}
| | | ..add <x, r> to the stack
| | | x := right [x];
| | end;
| end;
| if t <> val [x] then begin {t is not in the tree}
| | get_free (i); val [i] := t;
| | left [i] := null; right [i] := null; diff [i] := 0;
| | if t < val [x] then begin
| | | ..add <x, l> to the stack
| | | left [x] := i;
| | end else begin {t > val [x]}
```

```
      ..add <x, r> to the stack
      right [x] := i;
    end;
    d := 1;
    {invariant: the stack contains the path to a changed
     subtree whose height has increased by d (= 0 or 1);
     this subtree is balanced; values of diff for its
     vertices are correct; in the remaining part of the
     tree everything is unchanged (including the values
     of diff)}
    while (d <> 0) and the stack is non-empty do begin
      {d = 1}
      ..take a pair from stack into <v, direct>
      if direct = l then begin
        if diff [v] = 1 then begin
          c := 0;
        end else begin
          c := 1;
        end;
        diff [v] := diff [v] - 1;
      end else begin {direct = r}
        if diff [v] = -1 then begin
          c := 0;
        end else begin
          c := 1;
        end;
        diff [v] := diff [v] + 1;
      end;
      {c = the change in the height of the subtree rooted
       at v (compared with the initial tree); the array
       diff has correct values inside that subtree; the
       balance condition at v may be violated}
      balance (v, d1); d := c + d1;
    end;
  end;
end;
```

It is easy to check that d may be equal to 0 or 1 (but not -1); indeed, if $c = 0$, then `diff [v]` $= 0$ and balancing is not performed.

The deletion procedure is similar. Its main part is:

```
  {invariant: the stack contains a path to the changed
   subtree whose height was changed by d (=0 or -1)
   compared with the initial tree; this subtree is
```

```
   balanced; the values of diff are correct for the
   vertices of that subtree; the remaining part of the
   tree is unchanged (including the values of diff)}
  while (d <> 0) and the stack is not empty do begin
  | {d = -1}
  | ..take a pair from the stack into <v, direct>
  | if direct = l then begin
  | | if diff [v] = -1 then begin
  | | | c := -1;
  | | end else begin
  | | | c := 0;
  | | end;
  | | diff [v] := diff [v] + 1;
  | end else begin {direct = r}
  | | if diff [v] = 1 then begin
  | | | c := -1;
  | | end else begin
  | | | c := 0;
  | | end;
  | | diff [v] := diff [v] - 1;
  | end;
  | {c = the change in the height of the subtree rooted
  |  at v (compared with the initial tree); the array diff
  |  has correct values inside that subtree; the balance
  |  condition at v may be violated}
  | balance (v, d1);
  | d := c + d1;
  end;
```

It is easy to check that d may be equal to 0 or -1 (but not -2); indeed, if c = -1, then `diff[v]` = 0 and balancing is not performed.

Let us mention that the existence of the stack makes the variables `father` and `direction` used in the deletion procedure (see above) redundant, because now the stack top contains the same information.

12.2.6. Prove that while the element is added,

(a) the second case of the balancing step (see picture on p. 166) is, in fact, impossible;

(b) the complete balancing of the entire tree requires only one rotation.

However, deletion may require many rotations to restore the balance. ∎

Remark. Addition and deletion procedures may be simplified if we do not want to make them similar.

Other versions of balanced trees

There are several other ways to represent sets using trees. Some of those methods also guarantee a running time of order $\log n$ for each operation. Let us sketch one of them, called *B-trees*. (It is often used for large databases stored on a hard disk.)

Up to now each vertex contained only one element of the set. This element was used as a threshold that separates the left and right subtrees. Now let the vertex store $k \geq 1$ elements of the set. The value of k may be different for different vertices and may change while adding or deleting elements (see below). The k elements stored at a vertex are used as separators between $k + 1$ subtrees (so a vertex with k elements may have up to $k + 1$ sons).

Assume that some number $t \geq 1$ is fixed. We consider trees that satisfy the following requirements:

1. Each vertex contains not less than t and not more than $2t$ elements. (The root is an exception; it may contain any number of elements not exceeding $2t$.)

2. Any vertex with k elements either has $k + 1$ sons or does not have any sons at all (that is, it is a *leaf*).

3. All leaves are on the same level.

The *addition* of an element proceeds as follows. If the leaf where this element goes is not full (that is, contains less than $2t$ elements), we simply add this element to that leaf. If that leaf is full, then we have $2t + 1$ elements ($2t$ old ones and the new one). We split them into two leaves with t elements and the median element between them. This median element should be added to a vertex at the preceding level. This is easy if that vertex has less than $2t$ elements. If it is full, then it is split into two vertices, a median is found, etc. Finally, if we need to add the new element to the root and the root is full, we split the root into two vertices and the tree height is increased by 1.

The *deletion* of an element that is placed not at a leaf may be reduced to the deletion of the next element of the set, which is in a leaf. Therefore, it is enough to delete elements from leaves. If the leaf becomes too small, we can borrow some elements from a neighboring leaf, unless it too has the minimal possible size t. If both leaves have size t, together they have $2t$ elements, or rather $2t + 1$ elements if we count the separator between them. After deleting one element, the remaining $2t$ elements may be placed onto one leaf. However, the vertex of the preceding level may now be too small. In that case, we have to do the same transformation at that level, etc.

12.2.7. Implement this scheme of set representation and check that it also performs additions, deletions, and membership tests in time $C \log n$, where n is the cardinality of the set. ■

12.2.8. Another definition of a balanced tree requires that for each vertex the number of vertices in its left and right subtrees do not differ too much. (The advantage of this definition is that a rotation performed at some vertex does not destroy the balance at the ancestors of that vertex.) Using this idea, find a set representation that guarantees a running time bound of $C \log n$ for additions, deletions and membership tests.

[Hint. This approach also uses small and big rotations. The details can be found in the book of Reingold, Nievergelt, and Deo [9].] ∎

13 Context-free grammars

13.1 General parsing algorithm

To define a *context-free grammar* we should:

- fix a finite set A, called an *alphabet*, whose elements are called *symbols* or *letters*; finite sequences of symbols are called *strings* or *words*;
- divide all symbols in A into two classes: *terminal* symbols and *nonterminal* symbols;
- choose a nonterminal symbol called the *initial* symbol, or *axiom*;
- fix a finite set of *productions*, or *production rules*; each production has the form $K \to X$, where K is some nonterminal and X is a string that may contain both terminal and nonterminal symbols.

Assume that a context-free grammar is given (we often omit the words "context-free" because we do not consider another types of grammars). A *derivation* in this grammar is a sequence of strings $A_0, A_1, \ldots, A_n$, where A_0 is a one-letter string consisting of the initial symbol; A_{i+1} is obtained from A_i by replacing some nonterminal K in A_i by a string X according to one of the production rules $K \to X$.

A string containing only terminals is *generated* by a grammar if there exists a derivation that ends in this string. The set of all strings generated by some grammar G is called the *context-free language generated by* G. A language (that is, a set of strings) is called *context-free* if it is generated by some context-free grammar.

In this section, as well as the following one, we are interested in the following question: A context-free grammar G is given; construct an algorithm that checks if an input string belongs to the language generated by G.

Example 1. Alphabet:

```
( ) [ ] E
```

(four terminals and one nonterminal `E`). Axiom: `E`. Productions:

```
E → (E)
E → [E]
E → EE
E →
```

(the last rule has the empty string on its right-hand side).

Examples of generated strings:

```
            (empty string)
()
```

```
([])
()[([])]
[()[]()[]]
```

Examples of strings not in the language:

```
(
)(
(]
([)]
```

This grammar was considered in section 6. An algorithm that checks whether an input string belongs to the corresponding language was considered; that algorithm used a stack.

Example 2. Another grammar that generates the same language:

Alphabet: `( ) [ ] T E`

Productions:

```
E →
E → TE
T → (E)
T → [E]
```

In all subsequent examples, the axiom will be the nonterminal on the left-hand side of the first rule unless stated otherwise (in this example, the axiom is E).

For any nonterminal K, consider the set of all strings composed of terminals that can be obtained from K by a derivation. (For the axiom, this set is a language generated by a grammar.) In a sense, each rule of the grammar is a statement about those sets. Let us explain what we mean using the grammar of example 2. Let T and E be the sets of all strings in the alphabet {(,), [,]} derivable from nonterminals T and E, respectively. The rules of the grammar correspond to the following properties:

E →	E contains an empty string
E → TE	if A is in T and B is in E, then AB is in E
T → [E]	if A is in E, then [A] is in T
T → (E)	if A is in E, then (A) is in T

These four properties of E and T do not determine those sets uniquely. For example, they are still true if $T = E =$ the set of all strings. However, one may prove (for an arbitrary context-free grammar) that the sets defined by the grammar are minimal among all the sets having those properties ("minimal" means "minimal up to inclusion").

13.1.1. Give the precise statement and proof of this claim. ■

13.1.2. Construct a context-free grammar that generates the following strings (and no others):

(a) $0^k 1^k$ (the numbers of zeros and ones are equal);

(b) $0^{2k} 1^k$ (the number of zeros is twice as large as the number of ones);

(c) $0^k 1^l$ (the number of zeros k is larger than the number of ones l).

(d) (communicated by M. Sipser) all the strings $X2Y$ where X and Y are composed of 0s and 1s and $X \neq Y$. ■

13.1.3. Prove that there is no context-free grammar that generates all strings of type $0^k 1^k 2^k$ (and no other strings).

[Hint. Prove the following lemma about an arbitrary context-free language: Any sufficiently long string F in the language can be represented as $F = ABCDE$ in such a way that any string AB^kCD^kE (where B^k is B repeated k times) belongs to the language. To prove this lemma, find a nonterminal that is a descendant of itself in the "derivation tree".] ■

A nonterminal may be considered a "class name" for all generated strings. In the next example, we use fragments of English words as nonterminals; each fragment is considered to be one nonterminal symbol of the grammar.

Example 3.

Terminals: + * ( ) x

Nonterminals: $\langle\text{expr}\rangle$ $\langle\text{restexpr}\rangle$ $\langle\text{summ}\rangle$ $\langle\text{restsumm}\rangle$ $\langle\text{fact}\rangle$

Production rules:

$$
\begin{aligned}
\langle\text{expr}\rangle &\to \langle\text{summ}\rangle\ \langle\text{restexpr}\rangle \\
\langle\text{restexpr}\rangle &\to +\ \langle\text{expr}\rangle \\
\langle\text{restexpr}\rangle &\to \\
\langle\text{summ}\rangle &\to \langle\text{fact}\rangle\ \langle\text{restsumm}\rangle \\
\langle\text{restsumm}\rangle &\to *\ \langle\text{summ}\rangle \\
\langle\text{restsumm}\rangle &\to \\
\langle\text{fact}\rangle &\to \text{x} \\
\langle\text{fact}\rangle &\to (\ \langle\text{expr}\rangle\ )
\end{aligned}
$$

According to this grammar, an **expr**ession is a sequence of **summ**ands separated by symbols +; a **summ**and is a sequence of **fact**ors, separated by symbols *; a **fact**or is either the letter x or an **expr**ession in parentheses.

13.1.4. Give another grammar that generates the same language.

Answer. Here is one possibility:

$$\langle\text{expr}\rangle \to \langle\text{expr}\rangle + \langle\text{expr}\rangle$$
$$\langle\text{expr}\rangle \to \langle\text{expr}\rangle * \langle\text{expr}\rangle$$
$$\langle\text{expr}\rangle \to \text{x}$$
$$\langle\text{expr}\rangle \to (\ \langle\text{expr}\rangle\ )$$

This grammar is simpler, but not quite as good (see below). ■

13.1.5. An arbitrary context-free grammar is given. Construct an algorithm that checks if an input string belongs to the language generated by the grammar. The algorithm should run in polynomial time: the number of operations should not exceed P(input length) for some polynomial P. (The polynomial may depend on the grammar.)

Solution. The required polynomial time bound rules out any solution based on exhaustive search. However, a polynomial algorithm for a general context-free language exists. We give an outline of that algorithm below. In fact, it has no practical value, because all context-free grammars used in practice have special properties that make more efficient algorithms possible.

(1) Let $K_1, \ldots, K_n$ be the nonterminals of the given grammar. Construct a new context-free grammar with nonterminals $K_1', \ldots, K_n'$. This grammar has the following property: a string S can be generated from K_i' (in the new grammar) if and only if S is nonempty and can be generated from K_i in the old grammar.

To do that, we must know which nonterminals of the given grammar generate the empty string. Then each rule is replaced by a set of rules obtained as follows: On the left-hand side we add the dash, and on the right-hand side we omit some of the nonterminals that generate the empty string and put dashes near the other non-terminals. For example, if the initial grammar has the rule

```
K → L M N
```

and the empty string may be generated from L and N but not from M, the new grammar contains rules

```
K′ → L′M′N′
K′ → M′N′
K′ → L′M′
K′ → M′
```

(2) Therefore, we have reduced our problem to the case of a grammar where no terminal generates an empty string. Now we eliminate "cycles" of the form

```
K → L
L → M
M → N
N → K
```

(each rule has one nonterminal and no terminals on the right-hand side; nonterminals form a cycle of any length). This is easy; we identify all the nonterminals that appear in the same cycle.

(3) Now the membership test for the language generated by a grammar can be performed as follows. For any nonterminal and for any substring of a given string, we determine whether this substring can be generated from this nonterminal. We consider all substrings in the order of increasing length. All nonterminals are considered in such an order that for any rule of the form $K \to L$, the nonterminal L is considered before the nonterminal K. (This is possible because there are no cycles.) Let us explain this process by an example.

Assume that the grammar has rules

```
K → L
K → M N L
```

and no other rules with `K` on the left-hand side. We want to know if a given word A may be derived from the nonterminal `K`. This happens:

- if A can be derived from `L`;
- if A can be split into $A = BCD$ where B, C, D are nonempty strings such that B can be derived from `M`, C can be derived from `N`, and D can be derived from `L`.

All this information is available because B, C, and D are shorter than A and the nonterminal `L` is considered before the nonterminal `K`.

It is easy to see that the running time of the algorithm is polynomial. The degree of the polynomial depends on the number of nonterminals on the right-hand side of the grammar rules. The degree can be made smaller if we convert the grammar into a form where right-hand sides of rules contain not more than two nonterminals. This can be done easily; for example, the rule `K → LMK` may be replaced by two rules `K → LN` and `N → MK` where `N` is a new nonterminal. ■

13.1.6. Consider a grammar with one nonterminal symbol `K`, terminals `1`, `2`, and `3`, and the rules

```
K → 0
K → 1 K
K → 2 K K
K → 3 K K K
```

How do we check whether a given string belongs to the corresponding language if the string is scanned from left to right? The number of operations per character should be limited by a constant.

Solution. An integer variable `n` is used along with the invariant relation: "the input string belongs to the language if and only if the non-processed part of the input string is a concatenation of `n` strings from the language". ■

13.1.7. Repeat the previous problem for the grammar

```
  K → 0
  K → K 1
  K → K K 2
K → K K K 3
```
■

13.2 Recursive-descent parsing

Unlike the algorithm of the preceding section (which is of mostly theoretical interest), the recursive-descent parsing algorithm is used quite often. However, it is not applicable to all grammars. (See below the requirements that allow us to apply this method.)

The idea is as follows. For any nonterminal `K` we construct a procedure `ReadK` (being applied to any input string x) that does two things:

- finds the maximal prefix z of the string x that may appear as a prefix of some string derivable from `K`;
- says if the string z is derivable from `K`.

Before we give a more detailed description of this method, we should agree how the procedures access the input string and how they communicate their results. We assume that the input string is read character-by-character. In other words, we assume that there is a separator between the "already read" (processed) part and "the unread" part. (The last name should not be taken literally, because the first symbol of the unread part may be already known to the procedure.)

We assume that there exists a function without parameters

```
Next: Symbol
```

which returns the first symbol of the unread part. Its values are terminals as well as the special symbol `EOI` that stands for "End Of Input"; this symbol means that the input string is ended. (In a sense, EOI is written after the last character of the input string.) A call to `Next` does not move the separator between the read and unread parts. There exists a special procedure `Move` that "reads" the next character, that is, moves the separator to the right, adding one character to the processed part. This procedure is applicable when `Next<>EOI`. Finally, we have also a Boolean variable `b`; its role is described below

Now we state our requirements for the procedure `ReadK`:

- `ReadK` reads the maximal prefix A of the input string (its unprocessed part) that may appear as a prefix of some string derivable from `K`;
- the value of `b` becomes true or false depending on whether A is derivable from K or is only a prefix of some derivable string (but is not derivable itself).

It is convenient to use the following notation: Any string that is derivable from some nonterminal K is called a K-*string*. Any string that is a prefix of a string derivable from K is called K-*prefix*. If the two requirements for ReadK stated above are fulfilled, we say that "ReadK is correct for K".

Let us begin with an example. Assume that the rule

$$K \to L\ M$$

is the only rule of the grammar that has K on the left-hand side. Assume that L, M are nonterminals and ReadL, ReadM are correct procedures for those nonterminals.

Consider the following procedure:

```
procedure ReadK;
begin
| ReadL;
| if b then begin
| | ReadM;
| end;
end;
```

13.2.1. Give an example where this procedure is not correct for K.

Answer. Assume that any string 000...000 is derivable from L and that only the string 01 is derivable from M. Then the string 00001 is derivable from K, but the procedure ReadK does not see this. ■

Let us give a sufficient condition for the correctness of the procedure ReadK given above. To do that, we need some notation. Assume that a context-free grammar is fixed and that N is some nonterminal of that grammar. Consider the N-string A that has a proper prefix B, which is also an N-string (assuming such A and B exist). For each pair of such A and B, consider the terminal that follows B in A (appears immediately after B in A). The set of all such symbols (for all A and B) is denoted by Foll(N). (If no N-string is a proper prefix of another N-string, the set Foll(N) is empty.)

13.2.2. Find (a) Foll(E) for the grammar given in example 1 (see p. 176); (b) Foll(E) and Foll(T) for the grammar give in example 2 (see p. 177); (c) Foll(⟨summ⟩) and Foll(⟨fact⟩) for the grammar given in example 3 (see p. 178).

Answer. (a) Foll(E) = {[, (}. (b) Foll(E) = {[, (}; Foll(T) is empty (no T-string is a prefix of another T-string). (c) Foll(⟨summ⟩) = {*}; Foll(⟨fact⟩) is empty. ■

For any nonterminal N, we denote the set of all terminals that are first characters of nonempty N-strings by First(N). Now we are ready to give a sufficient condition for the correctness of the procedure ReadK in the situation explained above.

13.2.3. Prove that if Foll(L) and First(M) are disjoint sets and the set of all M-words is not empty, then the procedure ReadK is correct for K.

Solution. Consider two cases.

(1) Suppose that after the call to ReadL the value of b is false. In this case, ReadL reads the maximal L-prefix A; this prefix is not an L-string. The string A is a K-prefix (here we use the fact that the set of strings derivable from M is not empty). Will A be the maximal prefix of the input string that is at the same time a K-prefix? The answer is "yes". Indeed, assume that A is not maximal and there exists a longer string X that is both a K-prefix and a prefix of the input string. Since ReadL is correct, X is not a K-prefix, and therefore, $X = BC$ where B is an L-string and C is a M-prefix.

If B is longer than A, then A is not the maximal prefix of the input string that is also a K-prefix, which contradicts the correctness of ReadL. If $B = A$, then A would be an L-string, which is not true. Therefore, B is a proper prefix of A, C is not empty, and the first character of C follows the last character of B in A. So the first character of C belongs both to Foll(L) and First(M), which contradicts our assumption.

This contradiction shows that A is a maximal prefix of the input string that is also a K-prefix. Moreover, the argument above shows that A is not a K-string. The correctness of the procedure ReadK is therefore established (see its code).

(2) Assume that after the call to ReadL, the value of b is true. Then the procedure ReadK reads some string of the form AB where A is an L-string and B is an M-prefix. Therefore, AB is a K-prefix. Let us check that it is maximal. Assume that C is a longer prefix, which is at the same time a K-prefix. Then either C is an L-prefix (which is impossible because A is the maximal L-prefix) or $C = A'B'$, where A' is an L-string and B' is an M-prefix. If A' is shorter than A, then B' is not empty and begins with a character that belongs both to First(M) and Foll(L), which is impossible. If A' is longer than A, then A is not the maximal L-prefix. Therefore, the only possibility is $A' = A$, but in this case B is a prefix of B', which contradicts the correctness of ReadM. Therefore, AB is the maximal prefix of the input string that is a K-prefix.

It remains to check that the value of b returned by ReadK is correct. If b is true, this is evident. If b is false, then B is not an M-string, and we have to check that AB is not a K-string. Indeed, if $AB = A'B'$ where A' is an L-string and B' is an M-string, then A' cannot be longer than A (since ReadL reads the maximal prefix), A' cannot be equal to A (since in this case B' would be equal to B and could not be an M-string), and A' cannot be shorter than A (since in this case the first character of B' would belong both to First(M) and Foll(L)). The correctness of ReadK is proved. ■

Now we consider another special case. Assume that a context-free grammar contains the rules

$$\text{K} \to \text{L}$$
$$\text{K} \to \text{M}$$
$$\text{K} \to \text{N}$$

and has no other rules with K on the left-hand side.

13.2.4. Assume that ReadL, ReadM, and ReadN are correct (for L, M, and N) and that First(L), First(M), and First(N) are disjoint. Write a procedure ReadK that is correct for K.

Solution. Here is the procedure:

```
procedure ReadK;
begin
| if (Next is in First(L)) then begin
| | ReadL;
| end else if (Next is in First(M)) then begin
| | ReadM;
| end else if (Next is in First(N)) then begin
| | ReadN;
| end else begin
| | b := true or false depending on whether an
| |      empty string is derivable from K or not
| end;
end;
```

Let us prove that ReadK is correct for K. If the symbol Next is not in the sets First(L), First(M), and First(N), then the empty string is the maximal prefix of the input string that is a K-prefix. If Next belongs to one of those sets (and, therefore, does not belong to the others), then the maximal prefix of the input string that is a K-prefix is nonempty and the corresponding procedure reads it. ∎

13.2.5. Using the methods discussed, write a procedure that recognizes expressions generated by the grammar of example 3 (p. 178):

$$\begin{aligned}
\langle\text{expr}\rangle &\to \langle\text{summ}\rangle\ \langle\text{restexpr}\rangle\\
\langle\text{restexpr}\rangle &\to +\ \langle\text{expr}\rangle\\
\langle\text{restexpr}\rangle &\to\\
\langle\text{summ}\rangle &\to \langle\text{fact}\rangle\ \langle\text{restsumm}\rangle\\
\langle\text{restsumm}\rangle &\to *\ \langle\text{summ}\rangle\\
\langle\text{restsumm}\rangle &\to\\
\langle\text{fact}\rangle &\to \text{x}\\
\langle\text{fact}\rangle &\to (\ \langle\text{expr}\rangle\ )
\end{aligned}$$

Solution. This grammar does not follow the patterns above: among the right-hand sides of its rules there are combinations of terminals and nonterminals such as

$$+\ \langle\text{expr}\rangle$$

as well as a group of three symbols

$$(\ \langle\text{expr}\rangle \ )$$

This grammar also contains several rules with the same left-hand side and right-hand sides of different types, such as

$$\langle\text{restexpr}\rangle \to + \langle\text{expr}\rangle$$
$$\langle\text{restexpr}\rangle \to$$

These problems are not fatal. For example, a rule of type `K` $\to$ `L M N` may be replaced by two rules `K` $\to$ `L Q` and `Q` $\to$ `M N`. The terminals on the right-hand side may be replaced by nonterminals (the only rule involving these nonterminals allows to replace them by the corresponding terminals). If several rules have the same left-hand side and different right-hand sides, such as

$$\texttt{K} \to \texttt{L M N}$$
$$\texttt{K} \to \texttt{P Q}$$
$$\texttt{K} \to$$

they can be replaced by rules

$$\texttt{K} \to \texttt{K}_1$$
$$\texttt{K} \to \texttt{K}_2$$
$$\texttt{K} \to \texttt{K}_3$$
$$\texttt{K}_1 \to \texttt{L M N}$$
$$\texttt{K}_2 \to \texttt{P Q}$$
$$\texttt{K}_3 \to$$

We will not, however, transform the grammar (example 3) explicitly. Instead, we imagine that this transformation is performed (new nonterminals added), then the procedures for all nonterminals (old and new) are written, and finally the procedures for the new nonterminals are eliminated (by in-line substitutions). For example, for the rule

$$\texttt{K} \to \texttt{L M N}$$

we get the procedure

```
procedure ReadK;
begin
  ReadL;
  if b then begin ReadM; end;
  if b then begin ReadN; end;
end;
```

Its correctness is guaranteed if (1) Foll(L) and First(MN) are disjoint (First(MN) is equal to First(M) if the empty string is not derivable from M; otherwise, it is equal to the union of First(M) and First(N)); (2) Foll (M) and First(N) are disjoint.

Similarly, the rules

```
K → L M N
K → P Q
K →
```

lead to the procedure

```
procedure ReadK;
begin
  if (Next is in First(LMN)) then begin
    ReadL;
    if b then begin ReadM; end;
    if b then begin ReadN; end;
  end else if (Next is in First(PQ)) then begin
    ReadP;
    if b then begin ReadQ; end;
  end else begin
    b := true;
  end;
end;
```

To prove its correctness, we require the sets First(LMN) and First(PQ) to be disjoint.

Now we apply these methods to the grammar of example 3:

```
procedure ReadSymb (c: Symbol);
  b := (Next = c);
  if b then begin Move; end;
end;

procedure ReadExpr;
  ReadSumm;
  if b then begin ReadRestExpr; end;
end;

procedure ReadRestExpr;
  if Next = '+' then begin
    ReadSymb ('+');
    if b then begin ReadExpr; end;
  end else begin
```

```
| | b := true;
| end;
end;

procedure ReadSumm;
| ReadFact;
| if b then begin ReadRestSumm; end;
end;

procedure ReadRestSumm;
| if Next = '*' then begin
| | ReadSymb ('*');
| | if b then begin ReadSumm; end;
| end else begin
| | b := true;
| end;
end;

procedure ReadFact;
| if Next = 'x' then begin
| | ReadSymb ('x');
| end else if Next = '(' then begin
| | ReadSymb ('(');
| | if b then begin ReadExpr; end;
| | if b then begin ReadSymb (')'); end;
| end else begin
| | b := false;
| end;
end;
```

These procedures are mutually recursive, that is, some procedure uses another one which in its turn uses the first one, etc. This is allowed in Pascal if we use the so-called `forward` definitions of the mutually recursive procedures. As usual, to prove the correctness of recursive procedures we need to prove that (1) each of them is correct, assuming all calls work correctly (here our method works: one needs only check that the corresponding sets are disjoint); (2) the procedure terminates. The second claim is not self-evident. For example, if the grammar has the rule $K \to KK$, then no strings are derivable from K, and the sets Foll(K) and First(K) are empty (and therefore disjoint), but the procedure

```
procedure ReadK;
begin
| ReadK;
```

```
  if b then begin
    ReadK;
  end;
end;
```

(written according to our guidelines) never terminates.

In the case in question, the procedures ReadRestExpr, ReadRestAdd, and ReadFact either terminate immediately or decrease the length of the unprocessed part of the input string. Since any cycle of the mutually recursive calls includes one of them, termination is guaranteed. Our problem is solved. ■

13.2.6. Assume that a grammar has two rules with nonterminal K on the left-hand side:

$$K \to L\ K$$
$$K \to$$

According to these rules, any K-string is a concatenation of several L-strings. Assume also that the sets Foll(L) and First(K) (which equals First(L) in this case) are disjoint. Assume that a procedure ReadL is correct for L. Write a *nonrecursive* procedure ReadK that is correct for K.

Solution. As we already know, the following recursive procedure is correct for K:

```
procedure ReadK;
begin
  if (Next is in First(L)) then begin
    ReadL;
    if b then begin ReadK; end;
  end else begin
    b := true;
  end;
end;
```

Termination is guaranteed because the length of the unprocessed part is decreased before the recursive call. This recursive procedure is equivalent to the following nonrecursive one:

```
procedure ReadK;
begin
  b := true;
  while b and (Next is in First(L)) do begin
    ReadL;
  end;
end;
```

Let us formally check this equivalence. Termination is guaranteed both for the recursive and nonrecursive procedures. Therefore, it is enough to check that the body of the recursive procedure becomes equivalent to the body of the nonrecursive one if the recursive call is replaced by the call of the nonrecursive procedure. Let us make this replacement:

```
if (Next is in First(L)) then begin
| ReadL;
| if b then begin
| | b := true;
| | while b and (Next is in First(L)) do begin
| | | ReadL;
| | end;
| end;
end else begin
| b := true;
end;
```

The first command `b:=true` may be deleted because at this point b is already true. The second command `b:=true` may be moved to the beginning:

```
b := true;
if (Next is in First(L) then begin
| ReadL;
| if b then begin
| | while b and (Next is in First(L)) do begin
| | | ReadL;
| | end;
| end;
end;
```

Now the second `if` may be removed (because if b is false, the `while`-loop does nothing). We may also add the condition b to the first `if` (because b is true at that point). Thus we get

```
b := true;
if b and (Next is in First(L)) then begin
| ReadL;
| while b and (Next is in First(L)) do begin
| | ReadL;
| end;
end;
```

which is equivalent to the body of the nonrecursive procedure above (the first iteration of the loop is unfolded). ■

13.2.7. Prove the correctness of the nonrecursive procedure shown above directly, without referring to the recursive version.

Solution. Consider the maximal prefix of the input string that is a K-prefix. It can be represented as a concatenation of several nonempty strings: all are L-strings except, maybe, the last one, which is an L-prefix. We call those strings (including the last one) "components".

The invariant relation: several components are read; b is true if and only if the last component is an L-string.

Let us check that this invariant relation remains true after the next iteration. If only the last component remains, it is evident. If several components remain, the first of the remaining components is followed by a character that belongs to First(L) and is therefore not in Foll(L); so the first remaining component is a maximal L-prefix that is also a prefix of the of the unprocessed part. ∎

In practice a shorthand notation for grammars is used. Namely, rules of the form

$$\mathtt{K} \to \mathtt{L}\ \mathtt{K}$$
$$\mathtt{K} \to$$

(we assume that no other rule has K on the left-hand side, so K-strings are concatenations of L-strings) are omitted, and K is replaced by L enclosed in curly braces (which denotes iteration in this case). Also, several rules with the same left-hand side are often written as one rule where alternatives are written one after other separated by bars.

For example, the grammar for **expressions** given above may be rewritten as follows:

$$\langle\text{expr}\rangle \to \langle\text{summ}\rangle\ \{\ +\ \langle\text{summ}\rangle\ \}$$
$$\langle\text{summ}\rangle \to \langle\text{fact}\rangle\ \{\ *\ \langle\text{fact}\rangle\ \}$$
$$\langle\text{fact}\rangle \to \ \text{x} \mid (\ \langle\text{expr}\rangle\ )$$

13.2.8. Write a procedure that is correct for ⟨expr⟩, following this grammar. Use iteration instead of recursion whenever possible.

Solution.

```
procedure ReadSymb (c: Symbol);
| b := (Next = c);
| if b then begin Move; end;
end;

procedure ReadExpr;
begin
| ReadSumm;
```

```
  while b and (Next = '+') do begin
  | Move; ReadSumm;
  end;
end;

procedure ReadSumm;
begin
  ReadFact;
  while b and (Next = '*') do begin
  | Move; ReadFact;
  end;
end;

procedure ReadFact;
begin
  if Next = 'x' do begin
  | Move; b := true;
  end else if Next = '(' then begin
  | Move; ReadExpr;
  | if b then begin ReadSymb (')'); end;
  end else begin
  | b := false;
  end;
end;
```
■

13.2.9. The assignment `b:=true` in the last procedure may be omitted. Why?

Solution. We may assume that all procedures are called only when `b=true`. ■

13.3 Parsing algorithm for LL(1)-grammars

In this section, we consider one more algorithm to check if a given string is generated by a given grammar. This algorithm is called LL(1)-*parsing*. Its main idea can be summed up in one sentence: we may assume that all the production rules are applied to the leftmost nonterminal only; if we are lucky, the applicable rule is determined uniquely by the first character of the string derivable from this nonterminal.

Now we give the details. To begin with, we have the following

Definition. A *leftmost derivation* (of a string in a grammar) is a derivation where the leftmost nonterminal is replaced at each step.

13.3.1. Each derivable word (which contains only terminals) has the leftmost derivation.

Solution. During the derivation process, different nonterminals in a string are replaced independently of each other. (That is why the grammar is called

"context-free".) In other words, if at some point of the derivation we have the string ... K ... L ... where K and L are nonterminals, then the substitutions for K and L may be performed in any order. Therefore, we can rearrange the derivation in such a way that the left nonterminal K is replaced first. ■

13.3.2. Consider the grammar with four production rules:

(1) E →
(2) E → TE
(3) T → (E)
(4) T → [E]

Find the leftmost derivation of the word A = [()([])] and prove that it is uniquely determined.

Solution. At the first step, only the rule (2) may be applied:

E → TE

What happens with T then? Since A starts with [, only the rule (4) can be applied:

E → TE → [E]E

The leftmost E is now replaced by TE (otherwise the second symbol of the input string would be]):

E → TE → [E]E → [TE]E

and T is replaced according to (3):

E → TE → [E]E → [TE]E → [(E)E]E

Now the leftmost E should be replaced by the empty string, otherwise the third character of the input string would be (or [(other characters cannot be the first character of a T-string):

E → TE → [E]E → [TE]E → [(E)E]E → [()E]E

We continue:

... → [()TE]E → [()(E)E]E → [()(TE)E]E → [()([E]E)E]E →

→ [()([]E)E]E → [()([])E]E → [()([])]E → [()([])]

Thus we see that the leftmost derivation is uniquely determined. ■

What are the requirements for a grammar that make this approach (finding the unique leftmost derivation) possible? Assume that at some point the leftmost nonterminal is K. In other words, we have the string AKU where A is a string

containing only terminals and U is a string that may contain both terminals and nonterminals. Suppose the grammar has the rules

$$K \to L\,M\,N$$
$$K \to P\,Q$$
$$K \to R$$

and we have to choose one of them. We make the choice based on the first symbol of the part of the input string that is derivable from $K\,U$.

Consider the set First($L\,M\,N$) of all terminals that are first symbols of nonempty strings of terminals derivable from LMN. This set is equal to the union of the set First(L), the set First(M) (if the empty string is derivable from L), and the set First(N) (if the empty string is derivable from both L and N). To make the choice (based on the first character) possible, we require that the sets First($L\,M\,N$), First($P\,Q$), and First(R) are disjoint. But this is not the only requirement. Indeed, it is possible, for example, that the empty string is derivable from LMN, and the string derived from U starts with a character in First($P\,Q$). The definitions below take this problem into account.

A language recognized by a context-free grammar was defined as the set of all strings *of terminals* derivable from the initial nonterminal (axiom). We will also speak about strings composed of terminals and nonterminals derivable from the axiom, or from any other nonterminal, or from any string composed of terminals and nonterminals. So the relation "derivable from" can be considered as a binary relation defined on the set of all strings composed of terminals and nonterminals. (However, if we say that some string is derivable and do not specify the starting point of the derivation, we always mean that the derivation starts from the axiom.)

For any string X composed of terminal and nonterminals, First(X) denotes the set of all terminals that are the first characters of nonempty strings of terminals derivable from X. If for any nonterminal there is at least one string of terminals derivable from it, then the phrase "of terminals" may be omitted in the definition. We assume in the sequel that this condition is satisfied.

For any nonterminal K, the notation Follow(K) is used for the set of all terminals that appear in the derivable (from the axiom) strings immediately after K. (Please do not confuse this set with Foll(K) defined in the preceding section.) We add the symbol `EOI` to Follow(K) if there exists a derivable string that ends with K.

For each rule

$$K \to V$$

(where K is a nonterminal and V is a string that contains terminals and nonterminals) we define the set of *leading terminals*, which is denoted by Lead($K \to V$). By definition, Lead($K \to V$) is equal to First(V) or the union of First(V) and Follow(K) if the empty string is derivable from V.

Definition. A context-free grammar is called an LL(1)-*grammar* if for any two rules $K \to V$ and $K \to W$ with the same left-hand sides, the sets Lead($K \to V$) and Lead($K \to W$) are disjoint.

13.3.3. Is the grammar

```
K → K #
K →
```

(derivable strings are sequences of #'s) an LL(1)-grammar?

Solution. No, because # is a leading terminal for both rules. (This is true for the second rule because # belongs to Follow(K).) ■

13.3.4. Write an equivalent LL(1)-grammar.

Solution.

```
K → # K
K →
```

We have replaced a "left-recursive" rule by a "right-recursive" one. ■

The next problem shows that for a LL(1)-grammar, the next step in the construction of a leftmost derivation is uniquely defined.

13.3.5. Assume that a string X is derivable in an LL(1)-grammar and K is the leftmost nonterminal in X, that is, $X = AKS$ where A is a string of terminals and S is a string of terminals and nonterminals. Assume that two different rules of the grammar have K on the left-hand side, and both of them were applied to the nonterminal K selected in X. Both derivations were continued and two strings of terminals (having prefix A) were obtained. Prove that this prefix is followed by different terminals. (Here we consider EOI as a terminal.)

Solution. Those terminals are leading terminals of two different rules. ■

13.3.6. Prove that if a string is derivable in an LL(1)-grammar, its leftmost derivation is unique.

Solution. The preceding problem shows that at each step there is only one possible continuation. ■

13.3.7. A grammar is called *left-recursive* grammar if there exists a nonterminal K and a string derivable from K that starts with K (but is not equal to K). Prove that if (1) a grammar G is left-recursive; (2) for each nonterminal K, there exists a nonempty string derivable from K; and (3) for each nonterminal K there exists a derivation starting from the axiom and including K, then G is not an LL(1)-grammar.

Solution. Consider the derivation of a string KU from a nonterminal K where U is a nonempty string. We may assume that it is a leftmost derivation (other nonterminals may remain untouched). Consider the derivation

$$K \rightsquigarrow KU \rightsquigarrow KUU \rightsquigarrow \cdots$$

(here $\rightsquigarrow$ stands for several derivation steps) and the derivation $K \rightsquigarrow A$ where A is a nonempty string of terminals. At some point these two derivations diverge; however, both derivations may lead to a string that starts with A (in the first derivation there is still the nonterminal K at the beginning, which may be transformed to A). This contradicts the fact that the next step of the leftmost derivation is determined uniquely by the first character of the derived string. (This uniqueness is valid for derivations that start from the axiom; recall that K may appear in such a derivation by assumption.) ∎

Therefore, the LL(1) approach cannot be applied to left-recursive grammars (except for trivial cases). We have to transform them to equivalent LL(1)-grammars first (or use other parsing algorithms).

13.3.8. For any LL(1)-grammar, construct an algorithm that checks if the input string belongs to the language generated by the grammar. Use the preceding results.

Solution. We follow the scheme outlined above and look for a leftmost derivation of the given string. At each point, we have an initial part of the leftmost derivation that ends with a string composed of terminals and nonterminals. This string has the processed part of the input string as a prefix. Our algorithm stores the remaining part. In other words, we keep a string `S` of terminals and nonterminals with the following properties (the processed part of the input string is denoted by `A`):

1. the string `AS` is derivable (in the grammar);
2. any leftmost derivation of the input string includes the string `AS`.

These properties are denoted by "(I)" in the sequel.

Initially, `A` is empty and `S` contains only one nonterminal (the axiom).

If at some point the string `S` begins with a terminal `t` and `t` = `Next`, then we may call the procedure `Move` and delete the initial terminal `t` from `S`. Indeed, this operation leaves `AS` unchanged.

If the string `S` starts with a terminal `t` and `t` ≠ `Next`, then the input string is not derivable at all, because (I) implies that any (leftmost) derivation goes through the stage `AS`. (The same is true if `Next` = `EOI`.)

If `S` is empty, the condition (I) implies that the input string is derivable if and only if `Next` = `EOI`.

The only remaining case is that `S` starts with some nonterminal `K`. As we have already shown, all the leftmost derivations that start with `S` and end with a string whose first character is `Next`, begin with the same production rule, that is, the

production rule whose set of leading terminals includes Next. If such a rule does not exist, the input string is not derivable at all. If such a rule exists, we apply it to the opening nonterminal K of the string S and property (I) remains valid. We arrive at the following algorithm:

```
S := empty string;
error := false;
{error => input string is not derivable}
{not error => (I)}
while (not error) and not ((Next=EOI) and (S is empty))
|     do begin
|  if (S starts with a terminal equal to Next) then begin
|  | Move; delete the first symbol from S;
|  end else if (S starts with a terminal different from Next)
|  |    then begin
|  | error := true;
|  end else if (S is empty) and (Next <> EOI) then begin
|  | error := true;
|  end else if (S starts with some nonterminal K and Next
|  |     belongs to the set of leading terminals for one of
|  |     the production rules for K) then begin
|  | apply this rule to K
|  end else if (S starts with some nonterminal K and Next
|  |     does not belong to the set of leading terminals
|  |     for all the production rules for K) then begin
|  | error := true;
|  end else begin
|  | {this cannot happen}
|  end;
end;
{the input string is derivable <=> not error}
```

This algorithm always terminates. Indeed, if a terminal appears as the first symbol in S, the algorithm stops or reads the next input character. If nonterminals alternate as first symbols of S in an infinite loop, then the grammar is left-recursive; we may assume that this is not the case. (This follows from the preceding problem; we may easily remove from the grammar all the nonterminals that do not appear in derivations beginning with the axiom; the same can be done for nonterminals from which only the empty string is derivable.) ∎

Remarks.

- This algorithm uses S as a stack (all operations are made near its left end).
- In either of the last two cases (in the if-construct), no input characters are read. Therefore, we can precompute the action for all nonterminals and all

possible values of `Next`. Doing that, we need only one iteration per input character.

- In practice, it is convenient to have a table that lists all actions for all pairs (input symbol, nonterminal), and a small program that interprets this table.

13.3.9. To check if a given grammar is an LL(1)-grammar, we need to compute Follow(T) and First(T) for all nonterminals T. How can we do that?

Solution. If the grammar includes, say, the rule $K \to L\ M\ N$, then (Λ denotes the empty string):

$$\begin{array}{rcll}
\text{First}\,(L) & \subset & \text{First}\,(K), & \\
\text{First}\,(M) & \subset & \text{First}\,(K), & \text{if } \Lambda \text{ is derivable from } L, \\
\text{First}\,(N) & \subset & \text{First}\,(K), & \text{if } \Lambda \text{ is derivable both from } L \text{ and } M, \\
\text{Follow}\,(K) & \subset & \text{Follow}\,(N), & \\
\text{Follow}\,(K) & \subset & \text{Follow}\,(M), & \text{if } \Lambda \text{ is derivable from } N, \\
\text{Follow}\,(K) & \subset & \text{Follow}\,(L), & \text{if } \Lambda \text{ is derivable both from } M \text{ and } N, \\
\text{First}\,(N) & \subset & \text{Follow}\,(M), & \\
\text{First}\,(M) & \subset & \text{Follow}\,(L), & \\
\text{First}\,(N) & \subset & \text{Follow}\,(L), & \text{if } \Lambda \text{ is derivable from } M.
\end{array}$$

These rules (written for all productions) allow us to generate the sets First(T), and thereafter Follow(T), for all terminals and nonterminals T. As a starting point we use

$$\mathtt{EOI} \in \text{Follow}\,(K)$$

for an initial nonterminal K (the axiom) and

$$z \in \text{First}\,(z)$$

for any terminal z. We stop the generation process when the repeated applications of the rules give no new elements of the sets First(T) and Follow(T). ∎

14 Left-to-right parsing (LR)

Here we consider another approach to parsing, called LR(1)-*parsing* algorithm, as well as some simplified versions of it.

14.1 LR-processes

There are two main differences between LR(1)-parsing and LL(1)-parsing. First, we seek a *rightmost* derivation, not a leftmost one. Second, we construct the derivation from the bottom (beginning with the input string) to the top (the axiom) and not vice versa (as in LL(1)-parsing).

A *rightmost* derivation is a derivation where the rightmost nonterminal is replaced at each step.

14.1.1. Prove that any derivable string of terminals has a rightmost derivation. ■

It is convenient to look at the rightmost derivation backwards, starting from the input string. Let us define the notion of an LR-*process* on the input string A. This process involves the string A and another string S that contains both terminals and nonterminals. Initially, the string S is empty. The LR-process includes two types of actions:

(1) the first character of A (called the next input symbol and denoted by `Next`) may be moved to the end of the string S (and deleted from A); this action is called a *shift* action;

(2) if the right-hand side of some production rule is a suffix of S, then it can be replaced by the nonterminal that is on the left-hand side of that rule; the string A remains unchanged. This action is called a *reduce* action.

Let us mention that the LR-process is not deterministic; there are situations where many different actions are possible.

We say that the LR-process on a string A is *successful* if the string A becomes empty and the string S contains only one nonterminal, and this nonterminal is the initial nonterminal (the axiom).

14.1.2. Prove that for any string A (of terminals) a successful LR-process exists if and only if A is derivable in the grammar. Find a one-to-one correspondence between rightmost derivations and successful LR-processes.

Solution. The shift action does not change the string SA. The reduce action changes SA and this change is a reversed step of a derivation. This derivation is a rightmost one because the reduction is done at the end of S and all symbols of A are terminals. Therefore, each LR-process corresponds to a rightmost derivation.

Conversely, assume that a rightmost derivation is given. Imagine a separator placed after the last nonterminal in the string. When a production rule is applied to

that nonterminal, we may need to move the separator to the left (if the right-hand side of the rule applied ends with a terminal). Splitting this move into steps (one symbol per step) we get a process that is exactly an inverted LR-process. ∎

All changes in the string S during an LR-process are made near its right end. This is why the string S is called the *stack* of the LR-process.

So the problem of finding the rightmost derivation of a given string is the problem of constructing a successful LR-process on this string. At each step we have to decide whether we want to apply a shift or reduce action, and choose a production rule if several reductions are possible. In the LR(1)-algorithm, the decision is made based on S and the first symbol of A. If only information about S is used, it is an LR(0) algorithm. (The exact definitions are given below.)

Assume that a grammar is fixed. In the sequel, we assume that for each nonterminal there exists a string of terminals derivable from it.

Let $\mathtt{K} \to \mathtt{U}$ be one of the grammar's rules (K is a nonterminal, U is a string of terminals and nonterminals). We consider a set of strings (composed of both terminals and nonterminals) called the *left context* of the rule $\mathtt{K} \to \mathtt{U}$. (Notation: LeftCont($\mathtt{K} \to \mathtt{U}$).) By definition, this set contains all the strings that may appear as a stack content immediately before the reduction of U to K in a successful LR-process.

14.1.3. Reformulate this definition in terms of rightmost derivations.

Solution. Consider all rightmost derivations of the form

$$\langle\text{axiom}\rangle \rightsquigarrow \mathtt{XKA} \to \mathtt{XUA},$$

where A is a string of terminals, X is a string of terminals and nonterminals, and $\mathtt{K} \to \mathtt{U}$ is a production rule. All strings XU that appear in those derivations form the left context of the rule $\mathtt{K} \to \mathtt{U}$. Indeed, recall that we assume that for any nonterminal there exists a string of terminals derivable from it; therefore, the rightmost derivation of the string XUA may be continued until a right derivation of some string of terminals is obtained. ∎

14.1.4. All strings from LeftCont($\mathtt{K} \to \mathtt{U}$) end with U. Prove that if we delete this suffix U, the resulting set of strings does not depend on which rule (for the nonterminal K) is chosen. This set is denoted by Left(K).

Solution. The preceding problem shows that Left(K) is the set of all strings that may appear at the left of the rightmost nonterminal K in some rightmost derivation. ∎

14.1.5. Prove that in the last sentence the words "the rightmost nonterminal" may be omitted: Left(K) is the set of all strings that may appear on the left of any occurence of K in a rightmost derivation.

Solution. The derivation may be continued and all nonterminals on the right of K may be replaced by terminals; this replacement does not change anything on the left of K. ∎

14.1.6. Let G be a grammar. Construct a new grammar G^l such that for any nonterminal K of G, the grammar G^l contains a nonterminal ⟨LeftK⟩, and all elements of Left(K) (and no others) are derivable from ⟨LeftK⟩ in G^l. The terminals of G^l are nonterminals and terminals of G.

Solution. Let P be the initial nonterminal of G. The new grammar G^l has a rule

⟨LeftP⟩ → (right-hand side is the empty string)

For any production rule of the G, say,

K → L t M N (L, M, N are nonterminals, t is a terminal)

we add the following rules to G^l:

⟨LeftL⟩ → ⟨LeftK⟩
⟨LeftM⟩ → ⟨LeftK⟩ L t
⟨LeftN⟩ → ⟨LeftK⟩ L t M

etc. The meaning of the new rules may be explained as follows. An empty string may appear on the left of P. If a string X may appear on the left of K, then X may appear on the left of L; at the same time XLt may appear on the left of M, and XLtM may appear on the left of N. By induction over the length of a rightmost derivation, we check that everything that may appear on the left of some nonterminal, appears according to these rules. ∎

14.1.7. Why is it important in the preceding problem that we consider only the rightmost derivations?

Solution. Otherwise we must take into account transformations performed on the left of K. ∎

14.1.8. A context-free grammar is given. Construct an algorithm that for any input string finds all the sets Left(K) containing the string.

Remark (for experts only). The existence of such an algorithm, even a finite automaton (an inductive extension with a finite number of values, see section 1.3), follows from the preceding problem. Indeed, the grammar constructed has a special form: The right-hand sides of rules contain only one nonterminal and it is in the leftmost position. Nevertheless, we give an explicit construction of that automaton below.

Solution. By a *situation* of a given grammar we mean one of its rules with some additional information; namely, one of the positions on the right-hand side (before the first symbol, between the first and the second symbols, . . . , after the last symbol) is marked. For example, the rule

K → L t M N

(K, L, M, N are nonterminals, t is a terminal) gives five situations

K → _LtMN   K → L_tMN   K → Lt_MN   K → LtM_N   K → LtMN_

(the position is indicated by the underscore sign).

We say that a string S is *coherent* with a situation K → U_V if S ends with U, that is, if S = TU for some T and, moreover, T belongs to Left(K). (The meaning of this definition may be explained as follows: the suffix U of the stack S is ready for the future reduction of UV into K.) Now we can give an equivalent definition of LeftCont(K → X) as the set of all strings that are coherent with the situation K → X_, and Left(K) as the set of all strings coherent with the situation K → _X (here K → X is any production rule for nonterminal K).

Here is an equivalent definition in terms of LR-processes: S is coherent with the situation K → U_V if there exists a successful LR-process such that:

- during the process, the string S appears in the stack and S ends with U;
- for some time S is not touched and the string V appears on the right of S;
- UV is reduced into K;
- the LR-process continues and eventually terminates successfully.

14.1.9. Prove the equivalence of these two definitions.

[Hint. If S = TU and T belongs to Left(K), then it is possible to have T on the stack, then add U, then V, then reduce UV to K, and finally finish the LR-process successfully. (Several times we use the assumption that for any nonterminal there exists some string of terminals derivable from it; this assumption guarantees that we may add an arbitrary string to the stack.)] ■

Our goal is to construct an algorithm that finds all K such that the input string belongs to Left(K). Consider a function that maps each string S (of terminals and nonterminals) into the set of all situations that are coherent with S. This set is called a *state corresponding to* S. We denote it by State(S). It is enough to show that the function State(S) is inductive, that is, the value State(SJ) for any terminal or nonterminal J is determined by State(S) and J. (We have seen that membership in Left(K) may be expressed in terms of that function.) Indeed, the value State(SJ) can be computed according to the following rules (1)–(3):

> (1) If the string S is coherent with the situation K → U_V, and the string V starts with the symbol J, that is, V = JW, then SJ is coherent with the situation K → UJ_W.

This rule determines completely what situations not starting with an underscore are coherent with SJ. It remains to find for which nonterminals K the string SJ belongs to Left(K). This can be done according to the following rules:

(2) If the situation L → U_V turns out to be coherent with SJ (according to (1)) and V starts with a nonterminal K, then SJ belongs to Left(K).

(3) If SJ is in Left(L) for some L, the grammar contains a production rule L → V and V starts with a nonterminal K, then SJ belongs to Left(K).

Please note that the rule (3) may be considered a version of rule (2). Indeed, if the assumptions of (3) are valid, then the situation L → _V is coherent with SJ and V starts with a nonterminal K.

The correctness of these rules becomes more or less evident upon reflection. The only thing that requires comment is why the rules (2) and (3) generate *all* terminals K such that SJ belongs to Left(K). Let us try to explain why. Consider a rightmost derivation where SJ is on the left of K. How can the nonterminal K appear in this derivation? If the production rule that created K created a suffix of the string SJ at the same time, then the membership of SJ in Left(K) will be disclosed according to the rule (2). On the other hand, if K was the first symbol in a string generated by some other nonterminal L, then (because of the rule (3)) it is enough to check that SJ belongs to Left(L). It remains to apply the same argument to L and so on.

In terms of an LR-process, the same idea may be expressed as follows. First, the nonterminal K may participate in several reductions that do not touch SJ (those reductions correspond to applications of the rule (3)). Then a reduction that touches SJ is performed (this reduction corresponds to an application of rule (2)).

It remains to determine which situations are coherent with the empty string, that is, for which nonterminals K, the empty string belongs to Left(K). This can be done according to the following rules:

(1) the initial nonterminal (the axiom) has this property;

(2) if K has this property, K → V is a production rule, and the string V starts with a nonterminal L, then L has this property as well. ■

14.1.10. Perform the above analysis on the grammar

```
E → E + T
E → T
T → T * F
T → F
F → x
F → ( E )
```

(which generates the same language as the grammar of example 3, p. 178).

String S	State(S)
empty	E → _E+T E → _T T → _T*F T → _F F → _x F → _(E)
E	E → E_+T
T	E → T_ T → T_*F
F	T → F_
x	F → x_
(	F → (_E) E → _E+T E → _T T → _T*F T → _F F → _x F → _(E)
E+	E → E+_T T → _T*F T → _F F → _x F → _(E)
T*	T → T*_F F → _x F → _(E)
(E	F → (E_) E → E_+T
(T	= T
(F	= F
(x	= x
((	= (
E+T	E → E+T_ T → T_*F
E+F	= F
E+x	= x
E+(	= (
T*F	T → T*F_
T*x	= x
T*(	= (
(E)	F → (E)_
(E+	= E+
E+T*	= T*

State(S), problem 14.1.10

Solution. The sets State(S) for different S are shown in the table, p. 203. The equals sign means that the sets of situations that are values of the function State(S) of the strings connected by the the equals sign are equal.

Here is the rule to find State(SJ), provided we know State(S) and J (here S is a string of terminals and nonterminals, and J is a terminal or nonterminal):

> Find State(S) in the right column; consider the corresponding string T in the left column; append the symbol J to the end and find the set corresponding to the string TJ. (If the string TJ is not in the table, then State(SJ) is empty.) ■

14.2 LR(0)-grammars

Recall that our goal is to find a derivation for a given string. In other words, we seek a successful LR-process on this string. In all cases where our methods are applicable, such a successful LR-process (for a given string) is unique. We find it stepwise. At any point, we find the only possible next step. To ensure that only one step is possible, we need to put some requirements on the grammar. In this section we consider the simplest case, the so-called LR(0)-*grammars*.

As we already know:

> (1) The reduction according to the rule K → U with stack S may appear in a successful LR-process if and only if S belongs to LeftCont(K → U) or, equivalently, if S is coherent with situation K → U_.

A similar statement about shift is as follows:

> (2) A shift with next symbol a and stack S may appear in a successful LR-process if and only if S is coherent with some situation of type K → U_aV.

14.2.1. Prove the above claim.

[Hint. Assume that a shift occurs and a new terminal a is added to the stack S. Consider the first reduction that includes this terminal.] ■

Assume that some grammar is fixed. Consider an arbitrary string S of terminals and nonterminals. If the set State(S) contains a situation where the underscore sign is followed by a terminal, we say that the string S *allows a shift*. If the set State(S) contains a situation where the underscore sign is the last symbol, we say that the string S *allows a reduction* (according to the corresponding rule). We say that there is a *shift/reduce* conflict for the string S if both shift and reduction are allowed. We say that there is a *reduce/reduce* conflict for S if the string S allows a reduction according to two different rules.

The grammar is called a LR(0)-*grammar* if it has no conflicts of type shift/reduce and reduce/reduce for any string S.

14.2.2. Is the grammar given above (with nonterminals E and T) a LR(0)-grammar?

Solution. No, it has shift/reduce conflicts for strings T and E+T. ■

14.2.3. Are the following grammars LR(0)-grammars?

(a)	T	→	0	(b)	T	→	0
	T	→	T1		T	→	1T
	T	→	TT2		T	→	2TT
	T	→	TTT3		T	→	3TTT

Solution. Yes, see the corresponding tables (a) and (b) (no conflicts). ■

String S	State(S)
empty string	T → _0 T → _T1 T → _TT2 T → _TTT3
0	T → 0_
T	T → T_1 T → T_T2 T → T_TT3
	T → _0 T → _T1 T → _TT2 T → _TTT3
T1	T → T1_
TT	T → TT_2 T → TT_T3
	T → T_1 T → T_T2 T → T_TT3
	T → _0 T → _T1 T → _TT2 T → _TTT3
TT2	T → TT2_
TTT	T → TTT_3 T → TT_2 T → TT_T3
	T → T_1 T → T_T2 T → T_TT3
	T → _0 T → _T1 T → _TT2 T → _TTT3
TT0	= 0
TTT3	T → TTT3_
TTT2	= TT2
TTTT	= TTT
TTT0	= 0

(14.2.3, a)

This problem shows that LR(0)-grammars may be left-recursive as well as right-recursive.

14.2.4. Assume that an LR(0)-grammar is given. Prove that each string has at most one rightmost derivation. Give an algorithm that checks whether the input string is derivable.

Solution. Assume that an arbitrary input string is given. We construct an LR-process on that string stepwise. Assume that the current stack of the LR-process is S. We have to decide whether a shift or reduce action is needed (and which rule should be used in the reduction case). The definition of LR(0)-grammar guarantees

String S	State(S)
empty string	T → _0 T → _1T T → _2TT T → _3TTT
0	T → 0_
1	T → 1_T
	T → _0 T → _1T T → _2TT T → _3TTT
2	T → 2_TT
	T → _0 T → _1T T → _2TT T → _3TTT
3	T → 3_TTT
	T → _0 T → _1T T → _2TT T → _3TTT
1T	T → 1T_
10	= 0
11	= 1
12	= 2
13	= 3
2T	T → 2T_T
	T → _0 T → _1T T → _2TT T → _3TTT
20	= 0
21	= 1
22	= 2
23	= 3
3T	T → 3T_TT
	T → _0 T → _1T T → _2TT T → _3TTT
30	= 0
31	= 1
32	= 2
33	= 3
2TT	T → 2TT_
2T0	= 0
2T1	= 1
2T2	= 2
2T3	= 3
3TT	T → 3TT_T
	T → _0 T → _1T T → _2TT T → _3TTT
3T0	= 0
3T1	= 1
3T2	= 2
3T3	= 3
3TTT	T → 3TTT_
3TT0	= 0
3TT1	= 1
3TT2	= 2
3TT3	= 3

(14.2.3, b)

that only one action is possible, and all the information needed to make the decision is contained in State(S). Therefore, we can find the (only possible) next step of the LR-process. ■

14.2.5. What happens if the input string has no derivation in the grammar?

Answer. There are two possibilities: (1) neither a shift nor a reduce action will be possible at some point; (2) all possible shifts have the next symbol different from the actual one. ■

Remarks. **1**. When implementing this algorithm, there is no need to compute the set State(S) from scratch for each value of S. These sets may be kept in a stack. (At any point we keep on the stack the sets State(T) for all prefixes T of the current value of S.)

2. In fact, the string S itself is not used at all. It is enough to keep the sets State(T) for all its prefixes T (including S itself).

The algorithm that checks whether a given string is derivable in a LR(0)-grammar uses only some of the information available. Indeed, for each state it knows in advance which action (shift or reduction — and which reduction) is the only possible one. More elaborate algorithms can make a choice using the next input symbol as well as the stack content. Looking at the set State(S), it is easy to say for which input symbols a shift is possible. (It is possible for all terminals that follow the underscore in situations coherent with S.) The more difficult problem is: How do we use the next input symbol to decide if a reduction is possible?

There are two methods: the first is simpler, the second is more powerful. The grammars for which the first method is applicable are called SLR(1)-grammars (S for Simple). The second method uses all available information; these grammars are called LR(1)-grammars. (There is also an intermediate class of grammars called LALR(1)-grammars, discussed below.)

14.3 SLR(1)-grammars

Recall that for any nonterminal K we have defined (see p. 193) the set Follow(K) of terminals that may follow K in strings derivable from the initial nonterminal; this set also includes the symbol EOI if K may appear at the end of a derivable string.

14.3.1. Prove that if at some point of the LR-process the last symbol of the stack S is K and the process can be finished successfully, then Next belongs to Follow(K).

Solution. This fact is an immediate consequence of the definition (recall the correspondence between rightmost derivations and successful LR-processes). ■

Assume that some grammar is fixed. Consider a string S of terminals and nonterminals, and a terminal x. If the set State(S) contains a situation where the underscore is followed by a terminal x, we say that the pair ⟨S, x⟩ *allows a shift*. If the set State(S) contains a situation K → U_ where x belongs to Follow(K), we say

that the pair ⟨S, x⟩ SLR(1)-*allows reduction* (according to the rule K → U). We say that for the pair ⟨S, x⟩ there is a SLR(1)-*conflict of type shift/reduce*, if both shift and reduction are allowed. We say that for the pair ⟨S, x⟩ there is a SLR(1)-*conflict of type reduce/reduce* if reductions according to different rules are allowed.

The grammar is called a SLR(1)-*grammar* if it has no SLR(1)-conflicts of type shift/reduce and reduce/reduce for all pairs ⟨S, x⟩.

14.3.2. Assume that a SLR(1)-grammar is given. Prove that each string has at most one rightmost derivation. Give an algorithm to check whether a given string is derivable in the grammar.

Solution. We repeat the argument used for LR(0)-grammars. The difference is that the choice of the next action depends on the next input symbol (Next). ■

14.3.3. Check if the grammar shown above on p. 202 (having nonterminals E, T and F) is an SLR(1)-grammar.

Solution. Yes; both conflicts that prevent it from being a LR(0)-grammar are resolved when we take the next input symbol into account. Indeed, for both T and E+T a shift is possible only when Next = *, and the symbol * belongs neither to Follow(E) = {EOI, +,)} nor to Follow(T) = {EOI, +, *,)}. Therefore, reduction is impossible when Next = *. ■

14.4 LR(1)-grammars, LALR(1)-grammars

The SLR(1) approach still does not use all available information to decide if reduction is possible. It checks (for a given rule) whether reduction is possible with a given stack content and *separately* checks whether reduction is possible with a given input symbol Next. However, these tests are not independent. It may happen that both checks give a positive answer, but nevertheless the reduction for the given S *and* the given Next is impossible. The LR(1)-approach is free of this deficiency.

The LR(1)-approach is as follows: All our definitions and statements are modified to take into account what symbol is on the right of the replaced nonterminal while using a production rule. In other words, we carefully inspect the next symbol when reduction is performed.

Let K → U be one of the production rules, and let t be a terminal or a special symbol EOI (which is assumed to be at the end of the input string). We define the set LeftCont(K → U, t) as the set of all strings that may be the stack content immediately before the reduction U to K during a successful LR-process, assuming that Next = t at the time of reduction.

All strings in LeftCont(K → U) have suffix U. If we discard this suffix, we obtain the set of all strings that appear in the rightmost derivations immediately before the nonterminal K followed by t. This set (which does not depend on the specific rule K → U, but only on the nonterminal K) is denoted by Left(K, t).

14.4.1. Write a grammar whose nonterminals generate the sets Left(K, t) for all nonterminals K of the given grammar.

Solution. Nonterminals are symbols ⟨LeftK t⟩ for any nonterminal K and any terminal t (and also for t = EOI). Its production rules are as follows: Let P be the initial nonterminal (the axiom) of the given grammar. Then our new grammar has the rule

⟨LeftP EOI⟩ → (the right-hand side is the empty string).

Each rule of the given grammar produces several rules of the new one. For example, if the given grammar has a rule

K → LuMN

(L, M, N are nonterminals, u is a terminal), then the new grammar has rules

⟨LeftL u⟩ → ⟨LeftK x⟩

for all terminals x;

⟨LeftM s⟩ → ⟨LeftK y⟩ Lu

for any s that may appear as a first character in a string derivable from N, and for any y, as well as for all pairs s = y, if the empty string is derivable from N); and

⟨LeftN s⟩ → ⟨LeftK s⟩ LuM

for any terminal s. ■

14.4.2. How should we modify the definition of a situation?

Solution. Now a situation is defined as a pair

[situation in the old sense, terminal or EOI] ■

14.4.3. How to modify the definition of a string coherent with a situation?

Solution. The string S (of terminals and nonterminals) is coherent with the situation [K → U_V, t] (here t is a terminal or EOI) if U is a suffix of S, that is, if S = TU, and, moreover, T belongs to Left(K, t). ■

14.4.4. Show how to compute inductively the set State(S) of all situations coherent with a given string S.

Answer.

(1) If a string S is coherent with a situation [K → U_V, t] and the first character in V is J, that is, V = JW, then SJ is coherent with the situation [K → UJ_W, t].

This rule determines completely which situations that do not start with underscore are coherent with SJ. It remains to find out for which nonterminals K and terminals t the string SJ belongs to Left(K, t). This is done according to the following two rules:

(2) If the situation [L → U_V, t] is coherent with SJ (according to (1)) and V starts with a nonterminal K, then SJ belongs to Left(K, s) for any terminal s that may appear as a first symbol in a string derivable from V \ K (the string V without the first symbol K), as well as for s = t, if the empty string is derivable from V \ K.

(3) If SJ is in Left(L, t) for some L and t, and L → V is a production rule, and V starts with a nonterminal K, then SJ belongs to Left(K, s) for any nonterminal s that may appear as a first symbol in a string derivable from V \ K, as well as for s = t, if the empty string is derivable from V \ K. ■

14.4.5. Give the definition of the shift/reduce and shift/shift conflicts in the LR(1)-case.

Solution. Assume that a grammar is fixed. Let S be an arbitrary string of terminals and nonterminals. If the set State(S) contains a situation where the underscore sign is followed by a terminal t, we say that the pair ⟨S, t⟩ *allows a shift*. (This definition is the same as in the SLR(1)-case; we ignore the second components of pairs in State(S).)

If State(S) contains a situation whose first component ends with the underscore sign and the second component is a terminal t, we say that the pair ⟨S, t⟩ LR(1)-*allows a reduction* (via the corresponding rule). We say that there is a LR(1)-*conflict of type shift/reduce* for a pair ⟨S, t⟩ if this pair allows both shift and reduction. We say that there is a LR(1)-*conflict of type reduce/reduce* for a pair ⟨S, t⟩ if this pair allows reductions according to different rules. ■

The grammar is called a LR(1)-*grammar*, if there are no LR(1)-conflicts of type shift/reduce and reduce/reduce for all pairs ⟨S, t⟩.

14.4.6. For any LR(1)-grammar, construct an algorithm that checks if a given string is derivable in the grammar.

Solution. As before, at each stage of the LR-process we can determine which action is the only possible one. ■

It is useful to understand how the notions of LR(0)-coherence and LR(1)-coherence are related. (It is used below, when LALR(1)-grammars are considered.)

14.4.7. Find and prove the connection between the notions of LR(0)-coherence and LR(1)-coherence.

Solution. Assume that a grammar is fixed. The string S of terminals and nonterminals is LR(0)-coherent with situation K → U_V if and only if it is LR(1)-coherent with the pair [K → U_V, t] for some terminal t (or for t = EOI). In other words, Left(K) is the union of the sets Left(K, t) for all t. (In the latter form, the statement is almost obvious.) ■

Remark. Thus the function State(S) in the LR(1)-sense is an extension of the function State(S) in the LR(0)-sense: $\mathrm{State}_{\mathrm{LR}(0)}(\mathrm{S})$ is obtained from $\mathrm{State}_{\mathrm{LR}(1)}(\mathrm{S})$ when we discard the second component of all pairs.

Now we give a definition of a LALR(1)-grammar. Assume that a context-free grammar is fixed, S is a string of terminals and nonterminals, and `t` is a terminal (or `EOI`). We say that the pair $\langle \mathrm{S}, \mathrm{t} \rangle$ LALR(1)-allows reduction (according to some production rule) if there is another string S_1 with $\mathrm{State}_{\mathrm{LR}(0)}(\mathrm{S}_0) = \mathrm{State}_{\mathrm{LR}(0)}(\mathrm{S}_1)$ such that the pair $\langle \mathrm{S}_1, \mathrm{t} \rangle$ LR(1)-allows reduction according to that rule. Thereafter, the conflicts are defined in a natural way and a grammar is called a LALR(1)-*grammar* if there are no conflicts.

14.4.8. Prove that every SLR(1)-grammar is a LALR(1)-grammar and every LALR(1)-grammar is a LR(1)-grammar.

[Hint. This is an easy consequence of the definitions.] ■

14.4.9. Find an algorithm that checks if an input string is derivable in an LALR(1)-grammar. This algorithm should keep less information in the stack than the corresponding LR(1)-algorithm.

[Hint. It is sufficient to store the sets $\mathrm{State}_{\mathrm{LR}(0)}(\mathrm{S})$ in the stack, because the LALR(1)-possibility of reduction is determined by those sets. (Therefore, the only difference with SLR(1)-algorithm is in the table of possible reductions.)] ■

14.4.10. Give an example of an LALR(1)-grammar that is not a SLR(1)-grammar. ■

14.4.11. Give an example of an LR(1)-grammar that is not a LALR(1)-grammar. ■

14.5 General remarks about parsing algorithms

Practical applications of the methods described is a delicate matter. (For example, we need to store the tables as compactly as possible.) Sometimes a given grammar is not an LL(1)-grammar but still is an LR(1)-grammar. Often the given grammar can be transformed into an equivalent LL(1)-grammar. It is not clear which of these two approaches is more practical. The following general rule may be useful. If you design the language, keep it simple and do not use the same symbols for different purposes. Then usually it is easy to write an LL(1)-grammar or a recursive-descent parser. However, if the language is already defined by an LR(1)-grammar that is not LL(1), it is better not to change the grammar, just write a LR(1)-parser. To do this, you may use tools for automatic parser generation such as `yacc` (UNIX) and `bison` (GNU).

Much useful information about the theoretical and practical aspects of parsing can be found in the well-written book of Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman on compilers [2].

Further reading

1. Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Reading, MA, Addison-Wesley, 1976.

2. Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Reading, MA, Addison-Wesley, 1986.

3. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. *Introduction to Algorithms.* Cambridge (Mass.): The MIT Press, 1990.

4. Edsger W. Dijkstra. *A discipline of programming.* Englewood Cliffs, NJ, Prentice Hall, 1976.

5. Michael R. Garey, David S. Johnson. sl Computers and Intractability: A Guide to the Theory of NP-completeness. San Francisco, Freeman, 1979.

6. David Gries. *The Science of Programming.* New York, Springer, 1981.

7. A.G. Kushnirenko, G.V. Lebedev. *Programming for mathematicians* (*Programmirovanie dlja matematikov*). Moscow, Nauka, 1988.

8. Witold Lipski. *Kombynatoryja dla programistow.* Warszawa, Wydawnictwa naukowo-techniczne, 1982.

9. Edward M. Reingold, Jurg Nievergelt, Narsingh Deo. *Combinatorial Algorithms. Theory and Practice.* Englewood Cliffs, NJ, Prentice Hall, 1977.

10. Michael Sipser. *Introduction to the Theory of Computation.* Boston, PWS Publishing Company, 1996.

11. Niklaus Wirth. *Systematic Programming: An Introduction.* Englewood Cliffs, NJ, Prentice-Hall, 1973.

12. Niklaus Wirth. *Algorithms + Data Structures = Programs.* Englewood Cliffs, NJ, Prentice-Hall, 1976.

Index

Adelson-Velsky, G.M., 165
Aho, A.V., 59, 150, 211, 212
alphabet, 134, 176
angle detector, 40
array, 15
    with minimal element, 96
automaton
    finite, 73, 133, 143
    finite nondeterministic, 148
AVL-tree, 165
axiom (of a grammar), 176

B-tree, 174
backtracking, 49
    recursive, 107
Baur, W., 19
binary search, 25
binomial coefficient, 47, 114
bipartite graph, 130
Boyer–Moore algorithm, 140
Brudno, A.L., 17

calculator
    stack, 122
Catalan numbers, 44, 48, 116
chords of a circle, 46
code
    Gray, 38
coherent, 201
comments
    nested, 74
    removal, 73
common element (in sorted arrays), 23
complement
    of a regular set, 150
compound symbols, replacement, 73
conflict
    reduce/reduce, 204, 208, 210
    shift/reduce, 204, 208, 210
connected component
    directed graph, 95, 111, 127
    undirected graph, 110
connected graph, 88
context-free grammar, 176
context-free language, 176
    polynomial decidability, 179
convex hull, 68, 92
Cormen, T., 212
cost matrix, 126
cycle, Euler (along all the edges), 88

decimal fraction
    period, 12
decimal number
    printing, 9, 12, 120
    printing, recursive, 100
    reading, 74
Deo, N., 175, 212
deque
    array implementation, 87
    pointer implementation, 91
derivation
    in a grammar, 176
    leftmost, 191
    rightmost, 198
descendant of a vertex, 101
determinization, 149
Dijkstra algorithm (shortest path), 125, 127
Dijkstra, E.W., 5, 14, 28, 212
Dimentman, A.M., 31
Diophantine equation, 5, 7
directed graph, 88
division, 3
    fast, 14
Dutch flag, 28
dynamic programming, 114, 115, 179
    shortest path, 124

edge of a graph, 110

error
  index out of bounds, 21, 26, 27, 61, 63
Euclid's algorithm, 4, 5
  binary, 6
even permutation, 27
exchange, 18
expression, 178
  regular, 146
extension, inductive, 29

factor, 178
factorial, 3
  recursive program, 98
factorization, 8
fast multiplication, 20
Fibonacci numbers, 3, 116, 165
  fast computation, 3
FIFO, 85
finite automaton, 73, 133, 143
  nondeterministic, 148
First(X), 182, 193
Floyd algorithm, 125, 150
Floyd, R.W., 13
Foll(X), 182
Follow(X), 193
Ford–Bellman algorithm, 124
function
  inductive, 29

Garey, M.R., 59, 212
Gaussian integers, 9
GCD, 4
generated string, 176
grammar
  context-free, 176
  for expressions, 178
  LALR(1), 211
  left-recursive, 194
  LL(1), 193
  LR(0), 204
  LR(1), 210
  SLR(1), 208
graph
  bipartite, 130
  connected, 88
  connected component, 95, 110, 127
  directed, 88
  edge, 110
  shortest paths, 124
  undirected, 110
  vertex, 88, 110
Gray code, 38
greatest common divisor, 4
Gries, D., 18, 23, 26, 31, 212

Hanoi, Towers of
  nonrecursive solution, 118
  recursive solution, 100
hash function, 151
  universal family, 156, 157
hashing, 151
  running time, upper bound, 155
  universal, 155
  using lists, 154
  with open addressing, 151
height, 158
Hoare sorting, 67, 111
  nonrecursive, 121
Hoare, C.A.R., 111, 121
Hopcroft, J., 59, 212
Horner's rule, 18

inductive extension, 29
inductive function, 29
integer points in a circle, 10
intersection
  of regular sets, 150
  of sorted arrays, 23
inverse permutation, 27

Johnson, D.S., 59, 212

knapsack problem, 59, 118
Knuth, D.E., 136
Knuth–Morris–Pratt algorithm, 136
Kushnirenko, A.G., 18, 19, 29, 86, 87, 212

LALR(1)-grammar, 211
Landis, E.M., 165
language
  context-free, 176
  not context-free, 178
LCM, 5
Lead($K \to V$), 193
least common multiple, 5
Lebedev, G.V., 212
left context of the rule, 199
Left(K), 199
Left(K, t), 208
LeftCont($K \to U$), 199
LeftCont($K \to U, t$), 208
Leiserson, C., 212
letter, 176
LIFO, 85
Lipski, W., 212
Lissowski, Andrzei, 119
LL(1)-grammar, 193
LL(1)-parsing, 191
LR(0)-grammar, 204
LR(1)-grammar, 210
LR-process, 198

Matijasevich, Yu.V., 13, 136
matrix multiplication, optimal order, 117
matrix product, 126
median, search, 71, 112
memoization, 118
merge
  of sorted arrays, 22
minimal element, search, 70
monotone sequences
  generation, 36, 105
Morris, J.H., 136
multiplication
  fast, 20
  of polynomials, 19

nearest sum, 23
Nievergelt, J., 175, 212
nonassociative operation, 118
nondeterministic finite automaton, 148
nonterminal, 176
NP-completeness, 59
number
  of common elements, 20
  of different elements, 16, 17, 68
  of partitions, 47

open addressing, 151
operation
  non-associative, 118
ordered tree, 159

parentheses, 46
  correct expressions, 80, 176
parsing
  general context-free language, 179
  LL(1), 191
  LR(1), 198
  recursive-descent, 181
partitions
  generation, 37, 106
  number of, 47
Pascal, 21
Pascal triangle, 47, 115
Pascal, B., 115
paths, number of, 127
pattern matching, 133, 142, 143
period of a decimal fraction, 12
permutation
  even, 27
  inverse, 27, 43
permutations
  generation, 34, 42, 104
polygon, triangulation, 46
polynomial
  derivative, 19
  multiplication, 19, 20
  value, 18
positions tree, 49
postfix notation, 122
power

computation, 1
quick computation, 2
recursive computation, 99
Pratt V.R., 136
prefix, 134
prime factors, 8
priority queue, 97
problem
knapsack, 59, 118
NP-complete, 59
product
non-associative, 46
production rule, 176
programming
dynamic, 114, 115, 179

queens problem, 49
queue, 85
array implementation, 85
made of two stacks, 86
pointer implementation, 90
priority, 97
quicksort algorithm, 67, 111
nonrecursive, 121

Rabin–Karp algorithm, 142
recursion, 98
elimination, 114
recursive procedure, 98
recursive-descent parsing, 181
reduce, 198
regular expression, 146
regular set, 147
complement, 150
intersection, 150
Reingold, E.M., 175, 212
remainder, 3
reverse Polish notation, 122
Rivest, R., 212
rotation
left, right, 166
small, big, 166

search
k-th element, 71, 164
binary, 25
breadth-first, 96, 128
depth-first, 129
majority representative, 71
of a shortest path, 124
of a substring, 133, 136, 139, 140, 142
of the k-th element, 112
of the minimal element, 70
one of substrings, 145
sequences
generation, 33
set
bit array implementation, 93
data types, 93
list implementation, 94
regular, 147
representation, 151, 154
tree representation, 158
Sethi, R., 150, 211, 212
shift, 198
simulation, event queue, 97
Sipser, M., 212
situation
for a grammar, 200
SLR(1)-grammar, 208
sorting
$n \log n$, 61
heapsort, 63, 97
Hoare (quicksort), 67, 111
lower complexity bound, 69
merge, 61, 67
number of comparisons, 69
quadratic, 60
quicksort, nonrecursive, 121
radix, 70
topological, 107, 130
spelling checker, 157
stack, 79
array implementation, 79
of postponed tasks, 119
pointer implementation, 83

two in an array, 82
stack calculator, 122
State(S), 201
Strassen, V., 19
string, 134
coherent with a situation, 201
generated by a grammar, 176
having all possible substrings of length n, 90
subsequence
common, 31
increasing, 31
test, 30
subsets
generation, 34
of given cardinality, generation, 35
substring, 134
search, 136, 139, 140, 142
subtree, 158
suffix, 134
summand, 178
symbol
initial, 176
nonterminal, 176
terminal, 176

terminal, 176
leading, 193
topological sorting, 107, 130
Towers of Hanoi
nonrecursive solution, 118
recursive solution, 100
tree
B-tree, 174
balanced, 165
binary, 63
full binary, 158
height, 103
number of leaves, 102
number of vertices, 102, 120
of positions, 49
of positions, implementation, 56
ordered, 159
pointer implementation, 101, 159
recursive processing, 102
root, 101
traversal, 49, 50, 103, 107, 120
traversal, nonrecursive, 120
vertex, 101
triangle, Pascal, 115
triangulation of a polygon, 46, 116
Turbo Pascal, 21

Ullman, J.D., 59, 150, 211, 212
undirected graph, 110

value exchange, 1
Varsonofiev, D.V., 30, 157
vertex of a graph, 88, 110

Weinzweig, M.N., 31
Wirth, N., 21, 212
word
generated by a grammar, 176

Zvonkin, A.K., 119
Zvonkin, D., 7

Alexander Shen
Algorithms and Programming
Problems and Solutions

Algorithms and Programming is primarily intended for a first year undergraduate course in programming. It is structured in a problem-solution format that requires the student to think through the programming process, thus developing an understanding of the underlying theory. Although the author assumes some moderate familiarity with programming constructs, the book is easily readable by a student taking a basic introductory course in computer science. In addition, the more advanced chapters make the book useful for a course at the graduate level in the analysis of algorithms and/or compiler construction.

Each chapter is more or less independent, containing classical and well-known problems supplemented by clear and in-depth explanations. While program examples are written in Pascal, any other procedural language (e.g., Modula, Oberon, C) may be used instead. Problems at all different levels progress in difficulty. Some problems are somewhat loosely connected to one another, and others are devoted to one specific algorithm (e.g., section on LR-parsing).

The material covered includes such topics as combinatorics, sorting, searching, queues, grammar and parsing, selected well-known algorithms, and much more. Students and teachers will find this both an excellent text for learning programming and a source of problems for a variety of courses.

ISBN: 0-8176-3847-4