

EATCS

Monographs on Theoretical Computer Science

Editors: W. Brauer G. Rozenberg A. Salomaa

---

Advisory Board: G. Ausiello S. Even  
M. Nivat Chr. Papadimitriou D. Scott

Kurt Mehlhorn

Data Structures and Algorithms 1:  
**Sorting and Searching**

Kurt Mehlhorn

Data Structures and Algorithms 1:  
**Sorting and Searching**

With 87 Figures

Springer-Verlag  
Berlin Heidelberg New York Tokyo 1984

### *Editors*

Prof. Dr. Wilfried Brauer  
FB Informatik der Universität  
Rothenbaum-Chaussee 67–69, 2000 Hamburg 13, Germany

Prof. Dr. Grzegorz Rozenberg  
Institut of Applied Mathematics and Computer Science  
University of Leiden, Wassenaarseweg 80, P.O. Box 9512  
2300 RA Leiden, The Netherlands

Prof. Dr. Arto Salomaa  
Department of Mathematics, University of Turku  
20500 Turku 50, Finland

### *Author*

Prof. Dr. Kurt Mehlhorn  
FB 10, Angewandte Mathematik und Informatik  
Universität des Saarlandes, 6600 Saarbrücken, Germany

---

With the permission of B. G. Teubner publishers, Stuttgart,  
arranged, solely authorized and revised English translation of the original  
German edition: Effiziente Algorithmen (1977)

---

ISBN 3-540-13302-X Springer-Verlag Berlin Heidelberg New York Tokyo  
ISBN 0-387-13302-X Springer-Verlag New York Heidelberg Berlin Tokyo

This work is subject to copyright. All rights are reserved, whether the whole or part of material is concerned, specifically those of translation, reprinting, re-use of illustrations, broadcasting, reproduction by photocopying machine or similar means, and storage in data banks. Under § 54 of the German Copyright Law where copies are made for other than private use a fee is payable to "Verwertungsgesellschaft Wort", Munich.

© Springer-Verlag Berlin Heidelberg 1984  
Printed in Germany

The use registered names, trademarks, etc. in the publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Offsetprinting: Beltz Offsetdruck, Hemsbach/Bergstrasse  
Binding: J. Schäffer OHG, Grünstadt  
2145/3140-543210

## Preface

The design and analysis of data structures and efficient algorithms has gained considerable importance in recent years. The concept of “algorithm” is central in computer science, and “efficiency” is central in the world of money.

I have organized the material in three volumes and nine chapters.

Vol. 1: Sorting and Searching (chapters I to III)

Vol. 2: Graph Algorithms and NP-completeness (chapters IV to VI)

Vol. 3: Multi-dimensional Searching and Computational Geometry (chapters VII and VIII)

Volumes 2 and 3 have volume 1 as a common basis but are independent from each other. Most of volumes 2 and 3 can be understood without knowing volume 1 in detail. A general knowledge of algorithmic principles as laid out in chapter 1 or in many other books on algorithms and data structures suffices for most parts of volumes 2 and 3. The specific prerequisites for volumes 2 and 3 are listed in the prefaces to these volumes. In all three volumes we present and analyse many important efficient algorithms for the fundamental computational problems in the area. Efficiency is measured by the running time on a realistic model of a computing machine which we present in chapter I. Most of the algorithms presented are very recent inventions; after all computer science is a very young field. There are hardly any theorems in this book which are older than 20 years and at least fifty percent of the material is younger than 10 years. Moreover, I have always tried to lead the reader all the way to current research.

We did not just want to present a collection of algorithms; rather we also wanted to develop the fundamental principles underlying efficient algorithms and their analysis. Therefore, the algorithms are usually developed starting from a high-level idea and are not presented as detailed programs. I have always attempted to uncover the principle underlying the solution. At various occasions several solutions to the same problem are presented and the merits of the various solutions are compared (e.g. II.1.5, III.7 and V.4). Also, the main algorithmic paradigms are collected in chapter IX, and an orthogonal view of the book is developed there. This allows the reader to see where various paradigms are used to develop algorithms and data structures. Chapter IX is included in all three volumes.

Developing an efficient algorithm also implies to ask for the optimum. Techniques for proving lower bounds are dealt with in sections II.1.6,

II.3, V.7, VII.1.1, VII.2.3; most of chapter VI on NP-completeness also deals in some sense with lower bounds.

The organization of the book is quite simple. There are nine chapters which are numbered using roman numerals. Sections and subsections of chapters are numbered using arabic numerals. Within each section, theorems and lemmas are numbered consecutively. Cross references are made by giving the identifier of the section (or subsection) and the number of the theorem. The common prefix of the identifiers of origin and destination of a cross reference may be suppressed, i.e., a cross reference to section VII.1.2 in section VII.2 can be made by either referring to section VII.1.2 or to section 1.2.

Each Chapter has an extensive list of exercises and a section on bibliographic remarks. The exercises are of varying degrees of difficulty. In many cases hints are given or a reference is provided in the section on bibliographic remarks.

Most parts of this book were used as course notes either by myself or by my colleagues N. Blum, Th. Lengauer, and A. Tsakalidis. Their comments were a big help. I also want to thank H. Alt, O. Fries, St. Hertel, B. Schmidt, and K. Simon who collaborated with me on several sections, and I want to thank the many students who helped to improve the presentation by their criticism. Discussions with many colleagues helped to shape my ideas: B. Becker, J. Berstel, B. Commentz-Walter, H. Edelsbrunner, B. Eisenbarth, Ph. Flajolet, M. Fontet, G. Gonnet, R. Güttler, G. Hotz, S. Huddleston, I. Munro, J. Nievergelt, Th. Ottmann, M. Overmars, M. Paterson, F. Preparata, A. Rozenberg, M. Stadel, R. E. Tarjan, J. van Leeuwen, D. Wood, and N. Ziviani.

The drawings and the proof reading was done by my student Hans Rohnert. He did a fantastic job. Of course, all remaining errors are my sole responsibility. Thanks to him, there should not be too many left. The typescript was prepared by Christel Korten-Michels, Martina Horn, Marianne Weis and Doris Schindler under sometimes hectic conditions. I thank them all.

Saarbrücken, April 1984

Kurt Mehlhorn

## Preface to Volume 1

Volume 1 deals with sorting and searching. In addition, there is a chapter on fundamentals. Sorting and searching are the oldest topics in the area of data structures and efficient algorithms. They are still very lively and significant progress has been made in the last 10 years. I want to mention just a few recent inventions here; they and many others are covered extensively in the book: randomized algorithms, new methods for proving lower bounds, new hashing methods, new data structures for weighted and/or dynamic data, amortized analysis of data structures, . . . . Many of these new techniques have never appeared in book form before. I have tried to combine them with the classical knowledge about sorting and searching so as to lead the reader from the basics to current research.

Chapter I (Fundamentals) covers the basics of computer algorithms. First, a realistic model of computation (deterministic or probabilistic random access stored program machine) is defined, and the concepts of running time and storage space are introduced. Then a high level programming language and its connection with the machine model is discussed. Finally, some basic data structures such as queues, stacks, linked lists, and static trees are introduced, and some of their properties are derived. The material covered in this chapter lays the ground for all other chapters; the material (except for the section on randomized algorithms) is usually covered in introductory computer science courses.

In Chapter II (Sorting) we deal with sorting, selection, and lower bounds. We start with a detailed discussion of several general sorting methods, most notably heapsort, quicksort, and mergesort. The section on quicksort also contains a treatment of recurrence equations which arise frequently in the analysis of recursive programs. In section 2 we deal with sorting by distribution (bucketsort). We apply it to sorting words and sorting reals. The last section, finally, discusses fast algorithms for the selection problem. The sections on efficient algorithms are contrasted with methods for proving lower bounds. We first discuss methods for proving lower bounds in various decision tree models and then show how to lift some of these lower bounds to (restricted) RAM models.

Chapter III (Sets) is an in depth treatment of (one-dimensional) searching problems; multi-dimensional searching is discussed in chapter VII. We cover digital search trees, hashing, weighted trees, balanced trees, and dynamic weighted trees as methods for representing subsets of an infinite (or at least very large) universe. In section 7

we compare the data structures introduced in the first six sections. The last section covers data structures for subsets of a small universe where direct access using arrays becomes feasible. In the course of chapter III we also deal with important algorithmic paradigms, e. g. dynamic programming, balancing, and amortization.

There are no special prerequisites for volume 1. However, a certain mathematical maturity and previous exposure to programming and computer science in general is required. The Vordiplom (= examination at the end of the second year within the German university system) in Computer Science certainly suffices.

Saarbrücken, April 84

Kurt Mehlhorn

## Contents Vol. 2: Graph Algorithms and NP-Completeness

### *IV. Algorithms on Graphs*

1. Graphs and their Representation in a Computer
2. Topological Sorting and the Representation Problem
3. Transitive Closure of Acyclic Digraphs
4. Systematic Exploration of a Graph
5. A Close Look at Depth First Search
6. Strongly-Connected and Biconnected Components of Directed and Undirected Graphs
7. Least Cost Paths in Networks
  - 7.1 Acyclic Networks
  - 7.2 Non-negative Networks
  - 7.3 General Networks
  - 7.4 The All Pairs Problem
8. Minimum Spanning Trees
9. Maximum Network Flow and Applications
  - 9.1 Algorithms for Maximum Network Flow
  - 9.2 (0,1)-Networks, Bipartite Matching and Graph Connectivity
  - 9.3 Weighted Network Flow and Weighted Bipartite Matching
10. Planar Graphs
11. Exercises
12. Bibliographic Notes

### *V. Path Problems in Graphs and Matrix Multiplication*

1. General Path Problems
2. Two Special Cases: Least Cost Paths and Transitive Closure
3. General Path Problems and Matrix Multiplication
4. Matrix Multiplication in a Ring
5. Boolean Matrix Multiplication and Transitive Closure
6. (Min, +)-Product of Matrices and Least Cost Paths
7. A Lower Bound on the Monotone Complexity of Matrix Multiplication
8. Exercises
9. Bibliographic Notes

### *VI. NP-Completeness*

1. Turing Machines and Random Access Machines
2. Problems, Languages and Optimization Problems
3. Reductions and NP-complete Problems
4. The Satisfiability Problem is NP-complete
5. More NP-complete Problems
6. Solving NP-complete Problems
  - 6.1 Dynamic Programming
  - 6.2 Branch and Bound
7. Approximation Algorithms
  - 7.1 Approximation Algorithms for the Traveling Salesman Problem
  - 7.2 Approximation Schemes
  - 7.3 Full Approximation Schemes
8. The Landscape of Complexity Classes
9. Exercises
10. Bibliographic Notes

### *IX. Algorithmic Paradigms*

Bibliography

Subject Index



## Contents Vol. 3: Multidimensional Searching and Computational Geometry

### *VII. Multidimensional Data Structures*

1. A Black Box Approach to Data Structures
  - 1.1 Dynamization
  - 1.2 Weighting and Weighted Dynamization
  - 1.3 Order Decomposable Problems
2. Multi-dimensional Searching Problems
  - 2.1 D-dimensional Trees and Polygon Trees
  - 2.2 Range Trees and Multidimensional Divide and Conquer
  - 2.3 Lower Bounds
    - 2.3.1 Partial Match Retrieval in Minimum Space
    - 2.3.2 The Spanning Bound
3. Exercises
4. Bibliographic Notes

### *VIII. Computational Geometry*

1. Convex Polygons
2. Convex Hulls
3. Voronoi Diagrams and Searching Planar Subdivisions
  - 3.1 Voronoi Diagrams
  - 3.2 Searching Planar Subdivisions
    - 3.2.1 Removal of Large Independent Sets
    - 3.2.2 Path Decompositions
  - 3.3 Searching Dynamic Planar Subdivisions
- 3.3 Applications
4. The Sweep Paradigm
  - 4.1 Intersection of Line Segments and Other Intersection Problems in the Plane
  - 4.2 Triangulation and its Applications
  - 4.3 Space Sweep
5. The Realm of Orthogonal Objects
  - 5.1 Plane Sweep for Iso-Oriented Objects
    - 5.1.1 The Interval Tree and its Applications
    - 5.1.2 The Priority Search Tree and its Applications
    - 5.1.3 Segment Trees
    - 5.1.4 Path Decomposition and Plane Sweep for Non-Iso-Oriented Objects
  - 5.2 Divide and Conquer on Iso-Oriented Objects
    - 5.2.1 The Line Segment Intersection Problem
    - 5.2.2 The Measure and Contour Problems
- 5.3 Intersection Problems in Higher-Dimensional Space
6. Geometric Transforms
  - 6.1 Duality
  - 6.2 Inversion
7. Exercises
8. Bibliographic Notes

### *IX. Algorithmic Paradigms*

Bibliography

Subject Index

## Contents Vol. 1: Sorting and Searching

<i>I.</i>	<i>Foundations</i>	1
1.	Machine Models: RAM and RASP	3
2.	Randomized Computation	12
3.	A High Level Programming Language	22
4.	Structured Data Types	24
4.1	Queues and Stacks	24
4.2	Lists	26
4.3	Trees	30
5.	Recursion	33
6.	Order of Growth	37
7.	Secondary Storage	37
8.	Exercises	38
9.	Bibliographic Notes	39
<i>II.</i>	<i>Sorting</i>	40
1.	General Sorting Methods	42
1.1	Sorting by Selection, a First Attempt	42
1.2	Sorting by Selection: Heapsort	45
1.3	Sorting by Partitioning: Quicksort	50
1.4	Sorting by Merging	59
1.5	Comparing Different Algorithms	66
1.6	Lower Bounds	68
2.	Sorting by Distribution	80
2.1	Sorting Words	80
2.2	Sorting Reals by Distribution	84
3.	The Lower Bound on Sorting, Revisited	85
4.	The Linear Median Algorithm	94
5.	Exercises	98
6.	Bibliographic Notes	100
<i>III.</i>	<i>Sets</i>	102
1.	Digital Search Trees	104
1.1	Tries	104
1.2	Static Tries or Compressing Sparse Tables	108

2.	Hashing . . . . .	118
2.1	Hashing with Chaining . . . . .	118
2.2	Hashing with Open Addressing . . . . .	124
2.3	Perfect Hashing . . . . .	127
2.4	Universal Hashing . . . . .	139
2.5	Extendible Hashing . . . . .	145
3.	Searching Ordered Sets . . . . .	151
3.1	Binary Search and Search Trees . . . . .	152
3.2	Interpolation Search . . . . .	155
4.	Weighted Trees . . . . .	158
4.1	Optimum Weighted Trees, Dynamic Programming, and Pattern Matching . . . . .	159
4.2	Nearly Optimal Binary Search Trees . . . . .	177
5.	Balanced Trees . . . . .	187
5.1	Weight-Balanced Trees . . . . .	189
5.2	Height-Balanced Trees . . . . .	199
5.3	Advanced Topics on (a,b)-Trees . . . . .	213
5.3.1	Mergable Priority Queues . . . . .	213
5.3.2	Amortized Rebalancing Cost and Sorting Presorted Files . . . . .	217
5.3.3	Finger Trees . . . . .	228
5.3.4	Fringe Analysis . . . . .	241
6.	Dynamic Weighted Trees . . . . .	249
6.1	Self-Organizing Data Structures and Their Amortized and Average Case Analysis . . . . .	252
6.1.1	Self-Organizing Linear Lists . . . . .	256
6.1.2	Splay Trees . . . . .	263
6.2	D-trees . . . . .	275
6.3	An Application to Multidimensional Searching . . . . .	284
7.	A Comparison of Search Structures . . . . .	286
8.	Subsets of a Small Universe . . . . .	289
8.1	The Boolean Array (Bitvector) . . . . .	289
8.2	The $O(\log \log N)$ Priority Queue . . . . .	290
8.3	The Union-Find Problem . . . . .	298
9.	Exercises . . . . .	305
10.	Bibliographic Notes . . . . .	315
<i>IX. Algorithmic Paradigms . . . . .</i>		317
Appendix . . . . .		325
Bibliography . . . . .		328
Subject Index . . . . .		335

## I. Foundations

We use computer algorithms to solve problems, e.g. to compute the maximum of a set of real numbers or to compute the product of two integers. A problem  $P$  consists of infinitely problem instances. An instance of the maximum problem is e.g. to compute the maximum of the following set of 5 numbers 2,7,3,9,8. An instance of the multiplication problem is e.g. to compute the product of 257 and 123. We associate with every problem instance  $p \in P$  a natural number  $g(p)$ , its size. Sometimes, size will be a tuple of natural numbers; e.g. we measure the size of a graph by a pair consisting of the number of nodes and the number of edges. In the maximum problem we can define size as the cardinality of the input set (5 in our example), in the multiplication problem we can define size as the sum of the lengths of the decimal representations of the factors (6 in our example). Although the definition of size is arbitrary, there is usually a natural choice.

Execution of a program on a machine requires resources, e.g. time and space. Resource requirements depend on the input. We use  $T_A(p)$  to denote the run time of algorithm  $A$  on problem instance  $p$ . We can determine  $T_A(p)$  by experiment and measure it in milliseconds.

Global information about the resource requirements of an algorithm is in general more informative than information about resource requirements on particular instances. Global information such as maximal run time on an input of size  $n$  cannot be determined by experiment. Two abstractions are generally used: worst case and average case behaviour.

Worst case behaviour is maximal run time on any input of a particular size. We use  $T_A(n)$  to denote the

$$T_A(n) = \sup\{T_A(p); p \in P \text{ and } g(p) = n\}$$

(worst case) run time of algorithm  $A$  on an input of size  $n$ . Worst case behaviour takes a pessimistic look at algorithms. For every  $n$  we single out the input with maximal run time.

Sometimes, we are given a probability distribution on the set of problem instances. We can then talk about average (expected) case behaviour;

it is defined as the expectation of the run time on problems of a particular size

$$T_A^{\text{av}}(n) = E(\{T_A(p); p \in P \text{ and } g(p) = n\})$$

In this book (chapters II and III) computing expectations always reduces to computing finite sums. Of course, average case run time is never larger than worst case run time and sometimes much smaller. However, there is always a problem with average case analysis: does the actual usage of the algorithm conform with the probability distribution on which our analysis is based?

We can now formulaize one goal of this book. Determine  $T_A(n)$  for important algorithms  $A$ . More generally, develop methods for determining  $T_A(n)$ . Unfortunately, this goal is beyond our reach for many algorithms in the moment. We have to confine ourselves to determine upper and lower bounds for  $T_A(n)$ , i.e. to asymptotic analysis. A typical claim will be:  $T(n)$  is bounded above by some quadratic function. We write  $T(n) = O(n^2)$  and mean that  $T(n) \leq cn^2$  for some  $c > 0$ ,  $n_0$  and all  $n \geq n_0$ . Or we claim that  $T(n)$  grows at least as fast as  $n \log n$ . We write  $T(n) = \Omega(n \log n)$  and mean that there are constants  $c > 0$  and  $n_0$  such that  $T(n) \geq c n \log n$  for all  $n \geq n_0$ . We come back to this notation in section I.6.

We can also compare two algorithms  $A_1$  and  $A_2$  for the same problem. We say, that  $A_1$  is faster than  $A_2$  if  $T_{A_1}(n) \leq T_{A_2}(n)$  for all  $n$  and that  $A_1$  is asymptotically faster than  $A_2$  if  $\lim_{n \rightarrow \infty} T_{A_1}(n)/T_{A_2}(n) = 0$ . Of course, if  $A_1$  is asymptotically faster than  $A_2$  then  $A_2$  may still be more efficient than  $A_1$  on instances of small size. This trivial observation is worth being exemplified.

Let us assure that we have 4 algorithms  $A, B, C, D$  for solving problem  $P$  with run times  $T_A(n) = 1000 n$ ,  $T_B(n) = 200 n \log n$ ,  $T_C(n) = 10 n^2$  and  $T_D = 2^n$  milliseconds ( $\log$  denotes  $\log$  to base two throughout this book). Then  $D$  is fastest for  $0 \leq n \leq 9$ ,  $C$  is fastest for  $10 \leq n \leq 100$  and  $A$  is fastest for  $n \geq 101$ . Algorithm  $B$  is never the most efficient.

How large a problem instance can we solve in one hour of computing time? The answer is 3600 (1500, 600, 22) for algorithm  $A(B, C, D)$ . If maximal solvable problem size is too small we can do either one of two things. Buy a larger machine or switch to a more efficient algorithm.

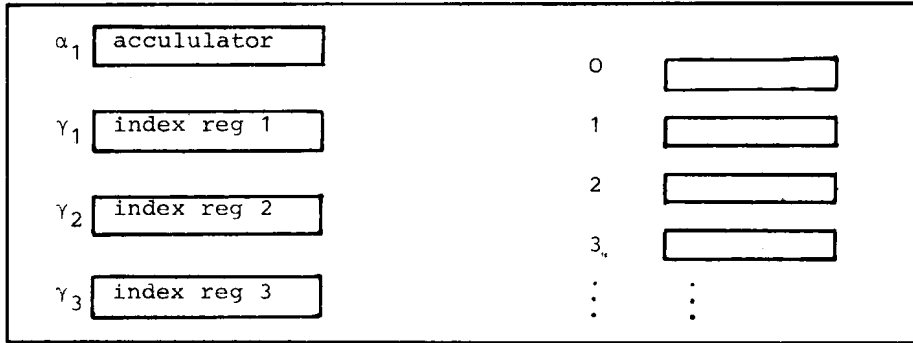
Assume first that we buy a machine which is ten times as fast as the present one, or alternatively that we are willing to spend 10 hours of computing time. Then maximal solvable problem size goes up to 36000 (13500, 1900, 25) for algorithms A(B,C,D). We infer from this example, that buying a faster machine hardly helps if we use a very inefficient algorithm (algorithm D) and that switching to a faster algorithm has a more drastic effect on maximally solvable problem size. More generally, we infer from this example that asymptotic analysis is a useful concept and that special considerations are required for small instances. (cf. sections II.1.5 and V.4)

So far, we discussed the complexity of algorithms, sometimes, we will also talk about the complexity of problems. An upper bound on the complexity of a problem is established by devising and analysing an algorithm; i.e. a problem  $P$  has complexity  $O(n^2)$  if there is an algorithm for  $P$  whose run time is bounded by a quadratic function. Lower bounds are harder to obtain. A problem  $P$  has complexity  $\Omega(n^2)$  if every algorithm for  $P$  has run time at least  $\Omega(n^2)$ . Lower bound proofs require us to argue about an entire class of algorithms and are usually very difficult. Lower bounds are available only in very rare circumstances (cf. II.1.6., II.3., III.4. and V.7.).

We will next define run time and storage space in precise terms. To do so we have to introduce a machine model. We want this machine model to abstract the most important features of existing computers, so as to make our analysis meaningful for every-day computing, and to make it simple enough, to make analysis possible.

#### I. 1. Machine Models: RAM and RASP

A random access machine (RAM) consists of 4 registers, the accumulator  $\alpha$  and index registers  $\gamma_1, \gamma_2, \gamma_3$  (the choice of three index registers is arbitrary), and an infinite set of storage locations numbered  $0, 1, 2, \dots$  (see figure next page).



The instruction set of a RAM consists of the following list of one address instructions. We use reg to denote an arbitrary element of  $\alpha$ ,  $\gamma_1, \gamma_2, \gamma_3$ ,  $i$  to denote a non-negative integer, op to denote an operand of the form  $i$ ,  $\rho(i)$  or reg, and mop to denote a modified operand of the form  $\rho(i + \gamma_j)$ . In applied position operand  $i$  evaluates to number  $i$ ,  $\rho(i)$  evaluates to the content of location  $i$ , reg evaluates to the content of reg and  $\rho(i + \gamma_j)$  evaluates to the content of location numbered  $(i + \text{content of } \gamma_j)$ . Modified operands are the only means of address calculation in RAMs. We discuss other possibilities at the end of the section. The instruction set consists of four groups:

#### Load and store instructions

$\text{reg} \leftarrow \text{op}$  , e.g.  $\gamma_1 \leftarrow \rho(2)$   
 $\alpha \leftarrow \text{mop}$  , e.g.  $\alpha \leftarrow \rho(\gamma_2 + 3)$   
 $\text{op} \leftarrow \text{reg}$  , e.g.  $\rho(3) \leftarrow \alpha$   
 $\text{mop} \leftarrow \alpha$  , e.g.  $\rho(\gamma_2 + 3) \leftarrow \alpha$

#### Jump instructions

goto  $k$  ,  $k \in \mathbb{N}_0$   
if reg  $\pi$  then goto  $k$  ,  $k \in \mathbb{N}_0$

where  $\pi \in \{=, +, <, \leq, >, \geq\}$  is a comparison operator.

#### Arithmetic instructions

$\alpha \leftarrow \alpha \pi \text{ mop}$

where  $\pi \in \{+, -, \times, \text{div}, \text{mod}\}$  is an arithmetic operator.

Index register instructions

$$Y_j \leftarrow Y_j \pm i \quad 1 \leq j \leq 3, i \in \mathbb{N}_0$$

A RAM program is a sequence of instructions numbered  $0, 1, 2, \dots$ . Integer  $k$  in jump instructions refers to this numbering. Flow of control runs through the program from top to bottom except for jump instructions.

Example: A RAM program for computing  $2^n$ : We assume that  $n$  is initially stored in location 0. The output is stored in location 1.

0: $Y_1 \leftarrow \rho(0)$	1
1: $\alpha \leftarrow 1$	1
2: <u>if</u> $Y_1 = 0$ <u>then goto</u> 6	$n+1$
3: $\alpha \leftarrow \alpha \times 2$	$n$
4: $Y_1 \leftarrow Y_1 - 1$	$n$
5: <u>goto</u> 2	$n$
6: $\rho(1) \leftarrow \alpha$	1

The right column shows the number of executions of each instruction on input  $n$ . □

In our RAMs there is only one data type: integer. It is straightforward to extend RAMs to other data types such as boolean and reals; but no additional insights are gained by doing so. Registers and locations can store arbitrary integers, an unrealistic assumption. We balance this unrealistic assumption by a careful definition of execution time. Execution time of an instruction consists of two parts: storage access time and execution time of the instruction proper. We distinguish two cost measures: unit cost and logarithmic cost. In the unit cost measure we abstract from the size of the operands and charge one time unit for each storage access and instruction execution. The unit cost measure is reasonable whenever algorithms use only numbers which fit into single locations of real computers. All algorithms in this book (except chapter VI) are of this kind for practical problem sizes and we will therefore always use unit cost measure outside chapter VI. However, the reader should be warned. Whenever, he analyses an algorithm in the unit cost measure, he should give careful thought to the size of the operands



involved. In the logarithmic cost measure we explicitly account for the size of the operands and charge according to their length  $L(\quad)$ . If binary representation is used then

$$L(n) = \begin{cases} 1 & \text{if } n = 0 \\ \lfloor \log_2 n \rfloor + 1 & \text{otherwise} \end{cases}$$

and this explains the name logarithmic cost measure. The logarithmic cost measure has to be used if the numbers involved do not fit into single storage locations anymore. It is used exclusively in chapter VI. In the following table we use  $m$  to denote the number moved in load and store instructions, and  $m_1$  and  $m_2$  to denote the numbers operated on in an arithmetic instruction. The meaning of all other quantities is immediate from the instruction format.

#### Cost for Storage Access

Operand	Unit Cost	Logarithmic Cost
$i$	0	0
reg	0	0
$\rho(i)$	1	$L(i)$
$\rho(i + \gamma_j)$	1	$L(i) + L(\gamma_j)$

#### Cost for Executing the Instruction Proper

Load and Stores	1	$1 + L(m)$
Jumps	1	$1 + L(k)$
Arithmetic	1	$1 + L(m_1) + L(m_2)$
Index	1	$1 + L(\gamma_j) + L(i)$

The cost of a conditional jump if reg  $\neq$  0 then goto  $k$  is independent of the content of reg because all comparison operators require only to check a single bit of the binary representation of reg.

Under the unit cost measure the cost of an instruction is  $1 + \#$  of storage accesses, under the logarithmic cost measure it is  $1 + \#$  of storage accesses + sum of the lengths of the numbers involved. Thus the execution time of an instruction is independent of the data in the unit