Berthold Vöcking · Helmut Alt
Martin Dietzfelbinger
Rüdiger Reischuk · Christian Scheideler
Heribert Vollmer · Dorothea Wagner (Eds.)

Algorithms Unplugged

Springer

Algorithms Unplugged

Berthold Vöcking • Helmut Alt •
Martin Dietzfelbinger • Rüdiger Reischuk •
Christian Scheideler • Heribert Vollmer •
Dorothea Wagner
Editors

Algorithms Unplugged

Springer

Editors

Prof. Dr. rer. nat. Berthold Vöcking
Lehrstuhl für Informatik 1
Algorithmen und Komplexität
RWTH Aachen University
Ahornstr. 55
52074 Aachen
Germany

Prof. Dr. rer. nat. Helmut Alt
Institut für Informatik
Freie Universität Berlin
Takustr. 9
14195 Berlin
Germany

Prof. Dr. Martin Dietzfelbinger
Institut für Theoretische Informatik
Fakultät für Informatik
und Automatisierung
Technische Universität Ilmenau
Helmholtzplatz 1
98693 Ilmenau
Germany

Prof. Dr. math. Rüdiger Reischuk
Institut für Theoretische Informatik
Universität zu Lübeck
Ratzeburger Allee 160
23538 Lübeck
Germany

Prof. Dr. rer. nat. Christian Scheideler
Institut für Informatik
Universität Paderborn
Fürstenallee 11
33102 Paderborn
Germany

Prof. Dr. rer. nat. Heribert Vollmer
Institut für Theoretische Informatik
Leibniz Universität Hannover
Appelstr. 4
30167 Hannover
Germany

Prof. Dr. rer. nat. Dorothea Wagner
Institut für Theoretische Informatik
Karlsruher Institut für Technologie (KIT)
Am Fasanengarten 5
76131 Karlsruhe
Germany

ISBN 978-3-642-15327-3 e-ISBN 978-3-642-15328-0
DOI 10.1007/978-3-642-15328-0
Springer Heidelberg Dordrecht London New York

ACM Codes: K.3, F.2

© Springer-Verlag Berlin Heidelberg 2011
This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilm or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.
The use of general descriptive names, registered names, trademarks, etc. in this 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.

Cover design: KuenkelLopka GmbH

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Many of the technological innovations and achievements of recent decades have relied on *algorithmic ideas*, facilitating new applications in science, medicine, production, logistics, traffic, communication, and, last but not least, entertainment. Efficient algorithms not only enable your personal computer to execute the newest generation of games with features unthinkable only a few years ago, but they are also the key to several recent scientific breakthroughs. For example, the sequencing of the human genome would not have been possible without the invention of new algorithmic ideas that speed up computations by several orders of magnitude.

Algorithms specify the way computers process information and how they execute tasks. They organize data and enable us to search for information efficiently. Only because of clever algorithms used by search engines can we find our way through the information jungle in the World-Wide Web. Reliable and secure communication in the Internet is provided by ingenious coding and encryption algorithms that use fast arithmetic and advanced cryptographic methods. Weather forecasting and climate change analysis rely on efficient simulation algorithms. Production and logistics planning employs smart algorithms that solve difficult optimization problems. We even rely on algorithms that perform GPS localization and routing based on efficient shortest-path computation for finding our way to the next restaurant or coffee shop.

Algorithms are not only executed on what people usually think of as computers but also on embedded microprocessors that can be found in industrial robots, cars and aircrafts, and in almost all household appliances and consumer electronics. For example, your MP3 player uses a clever compression algorithm that saves tremendous amounts of storage capacity. Modern cars and aircrafts contain not only one but several hundreds or even thousands of microprocessors. Algorithms regulate the combustion engine in cars, thereby reducing fuel consumption and air pollution. They control the braking system and the steering system in order to improve the vehicle's stability for your safety. In the near future, microprocessors might completely take over the controls, allowing for fully automated car driving in certain standardized situations. In modern aircraft, this is already put into practice for all phases of a flight from takeoff to landing.

The greatest improvements in the area of algorithms rely on beautiful ideas for tackling or solving computational problems more efficiently. The problems solved by algorithms are not restricted to arithmetic tasks in a narrow sense but often relate to exciting questions of nonmathematical flavor, such as:

- How to find an exit from inside a labyrinth or maze?
- How to partition a treasure map so that the treasure can only be found if all parts of the map are recombined?
- How to plan a tour visiting several places in the cheapest possible order?

Solving these challenging problems requires logical reasoning, geometric and combinatorial imagination, and, last but not least, creativity. Indeed, these are the main skills needed for the design and analysis of algorithms.

In this book we present some of the most beautiful algorithmic ideas in 41 articles written by different authors in colloquial and nontechnical language. Most of the articles arose out of an initiative among German-language universities to communicate the fascination of algorithms and computer science to high-school students. The book can be understood without any particular previous knowledge about algorithms and computing. We hope it is enlightening and fun to read, not only for students but also for interested adults who want to gain an introduction to the fascinating world of algorithms.

Berthold Vöcking
Helmut Alt
Martin Dietzfelbinger
Rüdiger Reischuk
Christian Scheideler
Heribert Vollmer
Dorothea Wagner

Contents

Part I

Searching and Sorting

Overview

Martin Dietzfelbinger and Christian Scheideler

Technische Universität Ilmenau, Ilmenau, Germany
Universität Paderborn, Paderborn, Germany

Every child knows that one can – at least beyond a certain number – find things much easier if one keeps order. We humans understand by keeping things in order that we separate the things that we possess into categories and assign fixed locations to these categories that we can remember. We may simply throw socks into a drawer, but for other things like DVDs it is best to sort them beyond a certain number so that we can quickly find every DVD. But what exactly do we mean by "quick," and how quickly can we sort or find things? These important issues will be dealt with in Part I of this book.

Chapter 1 of Part I starts with a quick search strategy called binary search. This search strategy assumes that the set of objects (in our case CDs) in which we will search is already sorted. Chapter 2 deals with simple sorting strategies. These are based on pairwise comparisons and flips of neighboring objects until all objects are sorted. However, these strategies only work well for a small number of objects since the sorting work quickly grows for larger numbers. In Chap. 3 two sorting algorithms are presented that work quickly even for a large number of objects. Afterwards, in Chap. 4, a parallel sorting algorithm is presented. By "parallel" we mean that many comparisons can be done concurrently so that we need much less time than with an algorithm in which the comparisons have to be done one after the other. Parallel algorithms are particularly interesting for computers with many processors or a processor with many cores that can work concurrently, or for the design of chips or machines dedicated for sorting. Chapter 5 ends the list of sorting algorithms with a method for topological sorting. A topological sorting is needed, for example, when there is a sequence of jobs that depend on each other. For example, job A must be executed before job B can start. The goal of topological sorting in this case is to come up with an order of the jobs so that the jobs can be executed one after the other without violating any dependencies between two jobs.

In Chap. 6 we get back to the search problem. This time, we consider the problem of searching in texts. More precisely, we have to determine whether a given string is contained in some text. A human being can determine this efficiently (for short search strings and a text that is not too long), but it is not that easy to design an efficient search procedure for a computer. In the chapter, a search method is presented that is very fast in practice even though there are some pathological cases in which the search time might be large.

The remainder of Part I deals with search problems in worlds that cannot be examined as a whole. How can one find the exit out of a labyrinth without ending up walking in a cycle or multiple times along the same path? Chapter 7 shows that this problem can be solved with a fundamental method called depth-first search if it is possible to set marks (such as a line with a piece of chalk) along the way. Interestingly, the depth-first search method also works if one wants to systematically explore a part of the World-Wide Web or if one wants to generate a labyrinth. In Chap. 8, we will again consider labyrinths, but this time the only item that one can use is a compass (so that there is a sense of direction). Thus, it is not possible to set marks. Still there is a very elegant solution: the Pledge algorithm. This algorithm can be used, for example, by a robot to find its way out of an arbitrary planar labyrinth caused by an arbitrary layout of obstacles. In Chap. 9, we will look at a special application of depth-first search in order to find cycles in labyrinths, street networks, or networks of social relationships. Sometimes it is very important to find cycles, for example, in order to resolve deadlocks, where people or jobs wait on each other in a cyclic fashion so that no one can advance. Surprisingly, there is a very simple and elegant way of detecting all cycles in a network.

Chapter 10 ends Part I, and it deals with search engines for the World-Wide Web. In this scenario, users issue search requests and expect the search engine to deliver a list of links to webpages that are as relevant as possible for the search requests. This is not an easy task as there may be thousands or hundreds of thousands of webpages that contain the requested phrases, so the problem is to determine those webpages that are most relevant for the users. How do search engines solve this problem? Chapter 10 explains the basic principles.

1

Binary Search

Thomas Seidl and Jost Enderle

RWTH Aachen University, Aachen, Germany

Where has the new Nelly CD gone? I guess my big sister Linda with her craze for order has placed it in the CD rack once again. I've told her a thousand times to leave my new CDs outside. Now I'll have to check again all 500 CDs in the rack one by one. It'll take ages to go through all of them!

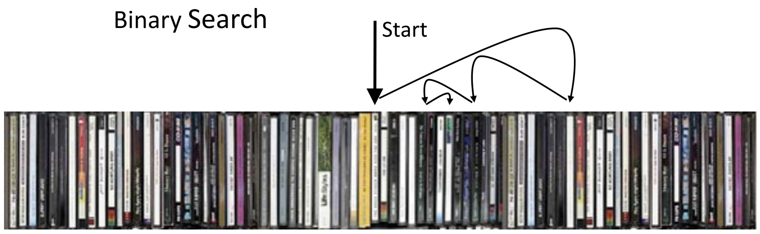
Okay, if I'm lucky, I might possibly find the CD sooner and won't have to check each cover. But in the worst case, Linda has lent the CD to her friend again: then I'll have to go through all of them and listen to the radio in the end.

Aaliyah, AC/DC, Alicia Keys ... hmmm, Linda seems to have sorted the CDs by artist. Using that, finding my Nelly CD should be easier. I'll try right in the middle. "Kelly Family"; must have been too far to the left; I have to search further to the right. "Rachmaninov"; now that's too far to the right, let's shift a bit further to the left ... "Lionel Hampton." Just a little bit to the right ... "Nancy Sinatra" ... "Nelly"!

Well, that was quick! With the sorting, jumping back and forth a few times will suffice to find the CD! Even if the CD hadn't been in the rack, this would have been noticed quickly. But when we have, say, 10,000 CDs, I'll probably have to jump back and forth a few hundred times to examine the CDs. I wonder if one could calculate that.

Start
Sequential Search
CD of interest

B. Vöcking et al. (eds.), *Algorithms Unplugged*,
DOI 10.1007/978-3-642-15328-0_1, © Springer-Verlag Berlin Heidelberg 2011



Sequential Search

Linda has been studying computer science since last year; there should be some documents of hers lying around providing useful information. Let's have a look ... "search algorithms" may be the right chapter. It describes how to search for an element of a given set (here, CDs) by some key value (here, artist). What I tried first seems to be called "sequential search" or "linear search." As already expected, half of the elements have to be scanned on average to find the searched key value. The number of search steps increases proportionally to the number of elements, i.e., doubling the elements results in double search time.

Binary Search

My second search technique seems also to have a special name, "binary search." For a given search key and a sorted list of elements, the search starts with the middle element whose key is compared with the search key. If the searched element is found in this step, the search is over. Otherwise, the same procedure is performed repeatedly for either the left or the right half of the elements, respectively, depending on whether the checked key is greater or less than the search key. The search ends when the element is found or when a bisection of the search space isn't possible anymore (i.e., we've reached the position where the element should be). My sister's documents contain the corresponding program code.

In this code, A denotes an "array," that is, a list of data with numbered elements, just like the CD positions in the rack. For example, the fifth element in such an array is denoted by A[5]. So, if our rack holds 500 CDs and we're searching for the key "Nelly," we have to call BinarySearch (rack, "Nelly", 1, 500) to find the position of the searched CD. During the execution of the program, left is assigned 251 at first, and then right is assigned 375, and so on.

The function BinarySearch returns the position of "key" in array "A" between "left" and "right"

```
function BinarySearch (A, key, left, right)
while left ≤ right do
    middle := (left + right)/2        // find the middle, round the result
    if A[middle] = key then return middle
    if A[middle] > key then right := middle − 1
    if A[middle] < key then left := middle + 1
endwhile
return not found
```

Recursive Implementation

In Linda's documents, there is also a second algorithm for binary search. But why do we need different algorithms for the same function? They say the second algorithm uses *recursion*; what's that again?

I have to look it up ...: "A recursive function is a function that is defined by itself or that calls itself." The *sum function* is given as an example, which is defined as follows:

$$\mathrm{sum}(n) = 1 + 2 + \cdots + n.$$

That means, the first n natural numbers are added; so, for $n = 4$ we get:

$$\mathrm{sum}(4) = 1 + 2 + 3 + 4 = 10.$$

If we want to calculate the result of the sum function for a certain n and we already know the result for $n - 1$, n just has to be added to this result:

$$\mathrm{sum}(n) = \mathrm{sum}(n-1) + n.$$

Such a definition is called a *recursion step*. In order to calculate the sum function for some n in this way, we still need the base case for the smallest n:

$$\mathrm{sum}(1) = 1.$$

Using these definitions, we are now able to calculate the sum function for some n:

$$\begin{aligned}\mathrm{sum}(4) &= \mathrm{sum}(3) + 4\\ &= (\mathrm{sum}(2) + 3) + 4\\ &= ((\mathrm{sum}(1) + 2) + 3) + 4\\ &= ((1 + 2) + 3) + 4\\ &= 10.\end{aligned}$$

The same holds true for a recursive definition of binary search: Instead of executing the loop repeatedly (*iterative* implementation), the function calls itself in the function body:

The function BinSearchRecursive returns the position of "key" in array "A" between "left" and "right"

```
function BinSearchRecursive (A, key, left, right)
if left > right return not found
middle := (left + right)/2       // find the middle, round the result
if A[middle] = key then return middle
if A[middle] > key then
    return BinSearchRecursive (A, key, left, middle − 1)
if A[middle] < key then
    return BinSearchRecursive (A, key, middle + 1, right)
```

As before, A is the array to be searched through, "key" is the key to be searched for, and "left" and "right" are the left and right borders of the searched region in A, respectively. If the element "Nelly" has to be found in an array "rack" containing 500 elements, we have the same function call, BinSearchRecursive (rack, "Nelly", 1, 500). However, instead of pushing the borders towards each other iteratively by a program loop, the BinSearchRecursive function will be called recursively with properly adapted borders. So we get the following sequence of calls:

BinSearchRecursive (rack, "Nelly", 1, 500)
BinSearchRecursive (rack, "Nelly", 251, 500)
BinSearchRecursive (rack, "Nelly", 251, 374)
BinSearchRecursive (rack, "Nelly", 313, 374)
BinSearchRecursive (rack, "Nelly", 344, 374)
...

Number of Search Steps

Now the question remains, how many search steps do we actually have to perform to find the right element? If we're lucky, we'll find the element with the first step; if the searched element doesn't exist, we have to keep jumping until we have reached the position where the element should be. So, we have to consider how often the list of elements can be cut in half or, conversely, how many elements can we check with a certain number of comparisons. If we presume the searched element to be contained in the list, we can check two elements with one comparison, four elements with two comparisons, and eight elements with only three comparisons. So, with k comparisons we are able to check $2 \cdot 2 \cdot \dots \cdot 2$ (k times) $= 2^k$ elements. This will result in ten comparisons for 1,024 elements, 20 comparisons for over a million elements, and 30 comparisons for over a billion elements! We will need an additional check if the searched element is not contained in the list. In order to calculate the converse, i.e., to determine the number of comparisons necessary for a certain number of elements, one has to use the inverse function of the power of 2. This function is called the "base 2 logarithm" and is denoted by $\log_2$. In general, the following holds true for logarithms:

$$\text{If } a = b^x, \text{ then } x = \log_b a. \tag{1.1}$$

For the base 2 logarithm, we have $b = 2$:

$$\begin{aligned}
2^0 &= 1, & \log_2 1 &= 0\\
2^1 &= 2, & \log_2 2 &= 1\\
2^2 &= 4, & \log_2 4 &= 2\\
2^3 &= 8, & \log_2 8 &= 3\\
&\vdots & &\vdots\\
2^{10} &= 1{,}024, & \log_2 1{,}024 &= 10\\
&\vdots & &\vdots\\
2^{13} &= 8{,}192, & \log_2 8{,}192 &= 13\\
2^{14} &= 16{,}384, & \log_2 16{,}384 &= 14\\
&\vdots & &\vdots\\
2^{20} &= 1{,}048{,}576, & \log_2 1{,}048{,}576 &= 20.
\end{aligned}$$

So, if $2^k = N$ elements can be checked with k comparisons, $\log_2 N = k$ comparisons are needed for N elements. If our rack contains 10,000 CDs, we have $\log_2 10{,}000 \approx 13.29$. As there are no "half comparisons," we get 14 comparisons! In order to further reduce the number of search steps of a binary search, one can try to guess more precisely where the searched key may be located within the currently inspected region (instead of just using the middle element). For example, if we are searching in our sorted CD rack for an artist's name whose initial is close to the beginning of the alphabet, e.g., "Eminem," it's a good idea to start searching in the front part of the rack. Accordingly, a search for "Roy Black" should start at a position in the rear part. For a further improvement of the search, one should take into account that some initials (e.g., D and S) are much more common than others (e.g., X and Y).

Guessing Games

This evening I'll put Linda to the test and let her guess a number between 1 and 1,000. If she didn't sleep during the lectures, she shouldn't need more than ten "yes/no" questions for that. (The figure below shows a possible approach for guessing a number between 1 and 16 with just four questions.)

In order to avoid asking the same boring question "Is the number greater/ less than ...?" over and over again, one can throw in something like "Is the number even/odd?". This will also exclude one half of the remaining possibilities. Another question could be "Is the number of tens/hundreds even/odd?" which would also result in halving the search space (approximately). However, when all digits have been checked, we have to return to our regular halving method (while taking into account the numbers that have already been excluded).

The procedure becomes even easier if we use the binary representation of the number. While numbers in the decimal system are represented as sums of multiples of powers of 10, e.g.,

$$\begin{aligned}107 &= \mathbf{1}\cdot 10^2 + \mathbf{0}\cdot 10^1 + \mathbf{7}\cdot 10^0\\ &= \mathbf{1}\cdot 100 + \mathbf{0}\cdot 10 + \mathbf{7}\cdot 1,\end{aligned}$$

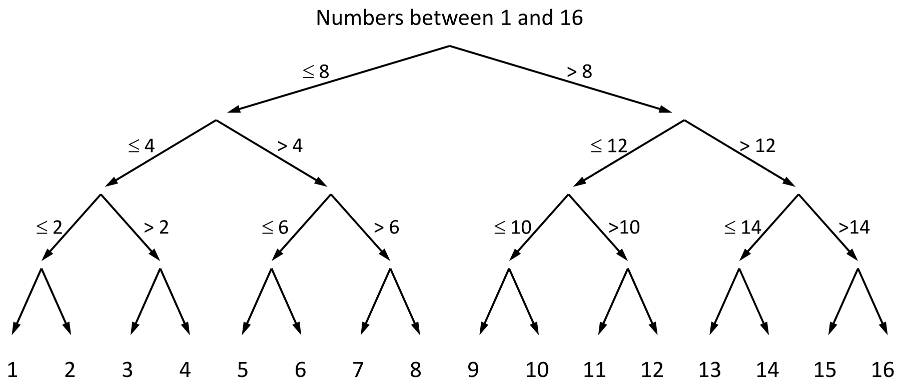
numbers in the binary system are represented as sums of multiples of powers of 2:

$$\begin{aligned}107 &= \mathbf{1}\cdot 2^6 + \mathbf{1}\cdot 2^5 + \mathbf{0}\cdot 2^4 + \mathbf{1}\cdot 2^3 + \mathbf{0}\cdot 2^2 + \mathbf{1}\cdot 2^1 + \mathbf{1}\cdot 2^0\\ &= \mathbf{1}\cdot 64 + \mathbf{1}\cdot 32 + \mathbf{0}\cdot 16 + \mathbf{1}\cdot 8 + \mathbf{0}\cdot 4 + \mathbf{1}\cdot 2 + \mathbf{1}\cdot 1.\end{aligned}$$

So the binary representation of 107 is 1101011. To guess a number using the binary representation, it is sufficient to know how many binary digits the number can have at most. The number of binary digits can easily be calculated using the base 2 logarithm. For example, if a number between 1 and 1,000 has to be guessed, one would calculate that

$$\log_2 1000 \approx 9.97 \text{ (round up!)},$$

i.e., ten digits, are required. Using that, ten questions will suffice: "Does the first binary digit equal 1?", "Does the second binary digit equal 1?", "Does the third binary digit equal 1?", and so on. After that, all digits of the binary representation are known and have to be converted into the decimal system; a pocket calculator will do this for us.



Further Reading

1. Donald Knuth: *The Art of Computer Programming*, Vol. 3: *Sorting and Searching*. 3rd edition, 1997.
   This book describes the binary search on pages 409–426.
2. Implementation of the binary search algorithm:
   `http://en.wikipedia.org/wiki/Binary_search`
3. Binary search in the Java SDK:
   `http://download.oracle.com/javase/6/docs/api/java/util/Arrays.html#binarySearch(long[],long)`
4. To perform a binary search on a set of elements, these elements have to be in sorted order. The following chapters explain how to sort the elements quickly:
   - Chap. 2 (Insertion Sort)
   - Chap. 3 (Fast Sorting Algorithms)
   - Chap. 4 (Parallel Sorting)

2 Insertion Sort

Wolfgang P. Kowalk

Carl-von-Ossietzky-Universität Oldenburg, Oldenburg, Germany

Let's sort our books in the bookcase by title so that each book can be accessed immediately if required.

How to achieve this quickly? We can use several concepts. For example, we can look at each book one after the other, and if two subsequent books are out of order we exchange them. This works since finally no two books are out of order, but it takes, on average, a very long time. Another concept looks for the book with the "smallest" title and puts it at first position; then from those books remaining the next book with smallest title is looked for, and so on, until all books are sorted. Also this works eventually; however, since a great deal of information is always ignored it takes longer than it should. Thus let's try something else.

The following idea seems to be more natural than those discussed above. The first book is sorted. Now we compare its title with the second book, and if it is out of order we exchange those two books. Now we look to find the correct position for the next book within the sequence of the first sorted books and place it there. This can be iterated until we have finally sorted all books. Since we can use information from previous steps this method seems to be most efficient.

Let us look more deeply at this algorithm. The first book alone is always sorted. We assume that all books to the left of current book i are sorted. To enclose book i in the sequence of sorted books we search for its correct position and put it there; to do this all, books on the right side of the correct place are shifted one position to the right. This is repeated with the next book at position $i+1$, etc., until all the books are sorted. This method yields the correct result very quickly, particularly if the "Binary Search" method from Chap. 1 is used to find the place of insertion.

How can we apply this intuitive method so it is useful for any number of books? To simplify the notation we will write a number instead of a book title.

In Fig. 2.1 the five books 1, 6, 7, 9, 11 on the left side are already sorted; book number 5 is not correctly positioned. To place it at the correct position

B. Vöcking et al. (eds.), *Algorithms Unplugged*,
DOI 10.1007/978-3-642-15328-0_2, © Springer-Verlag Berlin Heidelberg 2011

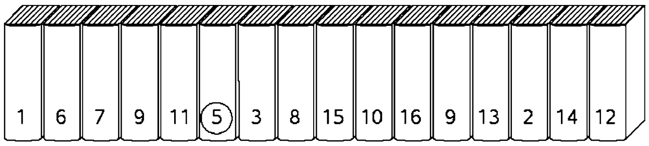


Fig. 2.1. The first five books are sorted

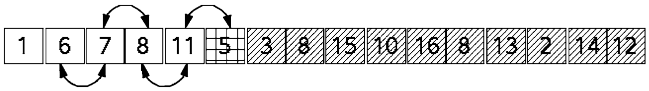


Fig. 2.2. Book "5" is situated at the correct position

we can exchange it with book number 11, then with book number 9, and so on, until it is placed at its correct position. Then we proceed with book number 3 and sort it by exchanging it with the books on the left-hand side. Obviously all books are eventually placed by this method at their correct position (see Fig. 2.2).

How can this be programmed? The following program answers this question. It uses an array of numbers A, where the cells of the array are numbered $1, 2, 3, \ldots$. Then $A[i]$ means the value at position i of array A. To sort n books requires an array of length n with cells $A[1], A[2], A[3], \ldots, A[n-1], A[n]$ to store all book titles. Then the algorithm looks like this:

Subsequent books are exchanged:

```
Given: $A$: Array with $n$ cells
**for** $i := 2$ **to** $n$ **do**
    $j := i$;      // book at position $i$ is current
                      as long as correct position not achieved
    **while** $j \geq 2$ **and** $A[j-1] > A[j]$ **do**
        $Hand := A[j]$;      // exchange current book with left neighbor
        $A[j] := A[j-1]$;
        $A[j-1] := Hand$;
        $j := j - 1$
    **endwhile**
**endfor**
```

How long does sorting take with this algorithm? Lets take the worst case where all books are sorted vice versa, i.e., the book with smallest number is at last position, that with biggest number at first, and so on. Our algorithm changes the first book with the second, the third with the first two books, the fourth with the first three books, etc., until eventually the last book is to be changed with all other $n-1$ books. The number of exchanges is

$$1 + 2 + 3 + \cdots + (n-1) = \frac{n \cdot (n-1)}{2}.$$

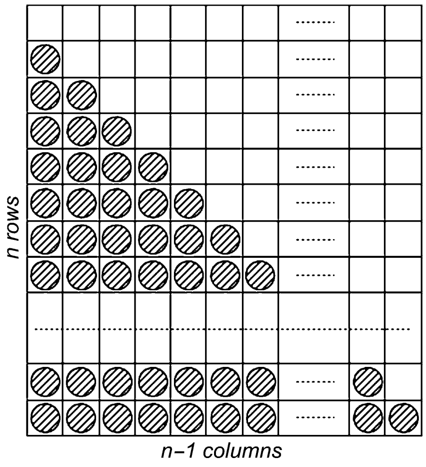


Fig. 2.3. Compute the number of exchanges

This formula is easily derived from Fig. 2.3. In the rectangle are $n\cdot(n-1)$ cells, and half of them are used for compare and exchange. This picture shows the absolute worst case. For the average case we assume that only half as many compares and exchanges are required. If the books are already almost sorted, then much less effort is required; in the best case if all books are sorted only $n-1$ comparisons have to be done.

You may have found that this algorithm is more cumbersome than necessary. Instead of exchanging two subsequent books, we shift all books to the right until the space for the book to be inserted is free.

Instead of exchanging k times two books, we shift $k+1$ times one book, which is more efficient. The algorithm look like this:

Sort books by insertion:

```
Given: $A$: array with $n$ cells;
**for** $i := 2$ **to** $n$ **do**
        // sort book at position $i$ by shifting
    $Hand := A[i]$;    // take current book
    $j := i - 1$;
        // as long as current position not found
    **while** $j \geq 1$ **and** $A[j] > Hand$ **do**
        $A[j+1] := A[j]$;    // shift book right to position $j$
        $j := j - 1$
    **endwhile**
    $A[j] := Hand$    // insert current book at correct position
**endfor**
```

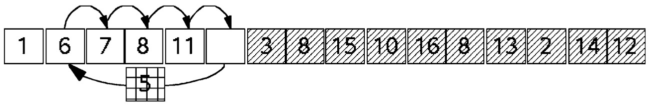


Fig. 2.4. Compute the number of exchanges

Further improvements of this sorting method, like inserting several books at once, and animations of this and other algorithms can be found at the Web site `http://einstein.informatik.uni-oldenburg.de/forschung/animAlgo/`

Considerations about computer hardware that can calculate shifting several books at the same time can be found in Chap. 4.

Even if sorting in normal computers requires a great deal of time, this algorithm is often used when the number of objects like books is not too big, or if you can assume that most books are almost sorted, since implementation of this algorithm is so simple. In the case of many objects to be sorted, other algorithms like MergeSort and QuickSort are used, which are more difficult to understand and to implement. They are discussed in Chap. 3.

To Read on

1. Insertion Sort is a standard algorithm that can be found in most textbooks about algorithms, for example, in Robert Sedgewick: *Algorithms in C++*. Pearson, 2002.
2. W.P. Kowalk: *System, Modell, Programm*. Spektrum Akademischer Verlag, 1996 (ISBN 3-8274-0062-7).

3

Fast Sorting Algorithms

Helmut Alt

Freie Universität Berlin, Berlin, Germany

The importance of sorting was described in Chap. 2. Searching a set of data efficiently, as with the binary search presented in Chap. 1, is only possible if the set is sorted. Imagine, for example, searching the telephone book of a big city if it weren't sorted alphabetically. In this example, we are dealing, as is often the case in practice, with millions of objects that have to be sorted. Therefore, it is important to find *efficient* sorting algorithms, i.e., ones with relatively short runtimes even for large data sets. In fact, runtimes can be very different for different algorithms applied to the same set of data.

In this chapter, therefore, we present two sorting algorithms which appear quite unusual at first. But if you want to sort large sets of objects, they have much faster runtimes than, e.g., the *Sorting by insertion* introduced in Chap. 2.

For simplicity we formulate the algorithms for the case of sorting sets of cards with numbers on them. Like *Sorting by insertion*, however, these algorithms work not only for numbers but also, e.g., for sorting books alphabetically by titles or, more generally, for all objects that can be compared by some kind of "size" or "value." Also, you do not necessarily need a computer to execute these algorithms. You can, for example, use these algorithms to sort a set of packages by weight, using a balance scale for each comparison of the weight of two packages. The author regularly uses Algorithm 1 for sorting the exams of his students alphabetically by name.

Therefore, the algorithms will on purpose be first described verbally instead of by a program in a standard programming language or by pseudocode.

B. Vöcking et al. (eds.), *Algorithms Unplugged*,
DOI 10.1007/978-3-642-15328-0_3, © Springer-Verlag Berlin Heidelberg 2011

3.1 The Algorithms

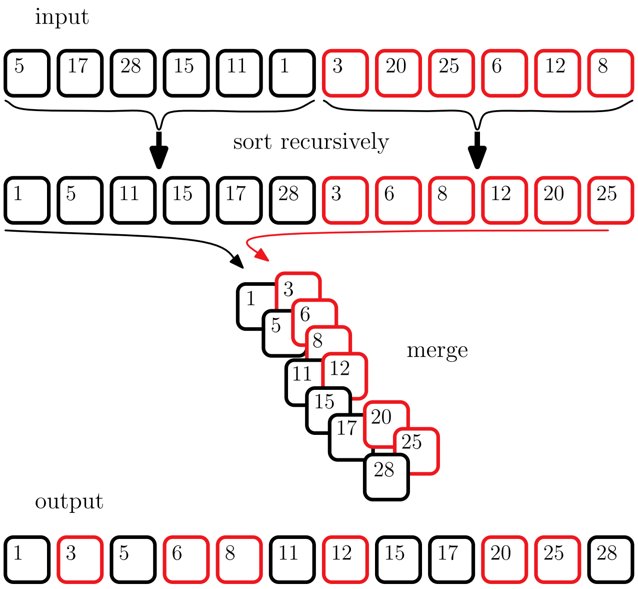
For simplicity, imagine that you receive from a master a stack of cards each of which has a number written on it. You are supposed to sort these cards in the order of ascending numbers and give them back to the master.

This is done as follows:

Algorithm 1

1. If the stack contains only one card, give it back immediately; otherwise:
2. Split the stack into two parts of equal size. Give each part to a helper and ask him to sort it *recursively*, i.e., exactly by the method described here.
3. Wait until both helpers have given back the sorted parts. Then traverse both stacks from top to bottom and merge the cards by a kind of zipper principle to a sorted full stack.
4. Return this stack to your master.

With the following example we demonstrate how this algorithm proceeds:

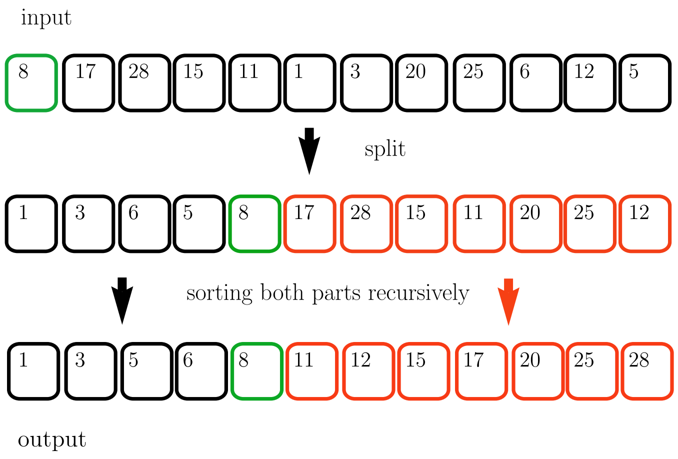


The second algorithm solves the same problem in a completely different manner:

Algorithm 2

1. If the stack consists of one card only give it back immediately; otherwise:
2. Take the first card from the stack. Go through the remaining cards and split them into the ones with a value not greater than the one of the first card (Stack 1) and the ones with a value greater than the one of the first card (Stack 2).
3. Give each of the two stacks obtained this way, if it contains cards at all, to a helper asking him to sort it *recursively*, i.e., exactly by the method described here.
4. Wait until both helpers have returned the sorted parts, then put at the bottom the sorted Stack 1, then the card drawn in the beginning, then the sorted Stack 2, and return the whole as a sorted stack.

Demonstrated with an example this looks as follows:



3.2 Detailed Explanations About These Sorting Algorithms

The first of the two algorithms is called *Mergesort*. It was already known to the famous Hungarian mathematician *John (Janos, Johann) von Neumann* (1903–1957)[1] at a time when computer science was not yet a scientific discipline by itself, and it was applied in mechanical sorting devices.

The second algorithm is called *Quicksort*. It was developed in 1962 by the famous British computer scientist *C.A.R. Hoare*.[2]

[1] Cf. http://en.wikipedia.org/wiki/John_von_Neumann

[2] Cf. http://en.wikipedia.org/wiki/C._A._R._Hoare

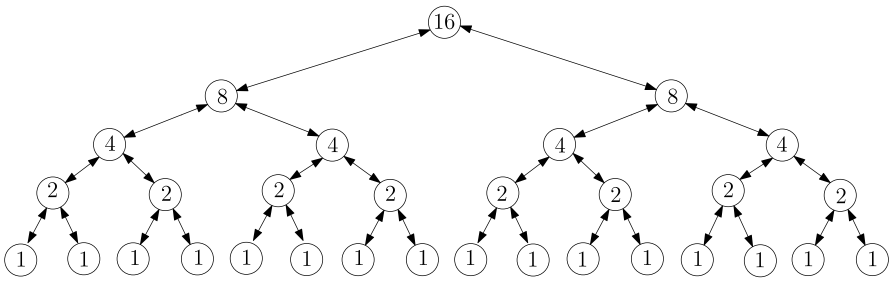


Fig. 3.1. Recursion tree for *Mergesort*

The descriptions in the previous section show that a computer is not necessarily needed for the execution of the algorithms. For a better understanding of both algorithms we recommend that you carry them out "by hand" adopting the roles of the various "helpers" yourself.

In all high-level programming languages (e.g., C, C++, Java) it is possible for a procedure to call "itself" to solve the same task in the same manner for a smaller subproblem. This concept is called *recursion* and it plays an important role in computer science. For example, if you apply *Mergesort* to a sequence of 16 numbers, then both helpers get a subsequence of length 8 each to be sorted. Each of them again calls his two helpers to sort sequences of length 4, and so on. The complete operation of this algorithm is presented in Fig. 3.1, which is called a *tree* in computer science.

The recursion stops when the subproblems become sufficiently small to be solved directly. In our algorithms this is the case for sequences of length 1, where nothing has to be done any more to have them sorted. In both descriptions of the algorithms, statement 1 takes care of this base case of the recursion.

So, our algorithms solve a large problem by decomposing it into smaller subproblems, solving those recursively, and combining the resulting partial solutions for a complete solution. Proceeding in this manner is called *divide-and-conquer* in computer science. This principle can be applied successfully not only to sorting but also to many other, quite different problems.

3.3 Experimental Comparison of the Sorting Algorithms

It is a natural question as to why algorithms that strange should be used for sorting, which seems to be a really simple problem. Therefore, we *implemented* (i.e., programmed) both algorithms, as well as *Sorting by insertion* from Chap. 2, on a computer at our institute and recorded the time that those algorithms needed for sequences of numbers of different lengths. Figure 3.2 shows the result. Obviously, *Mergesort* is much faster than *Sorting by insertion* and *Quicksort* is significantly faster than *Mergesort*.

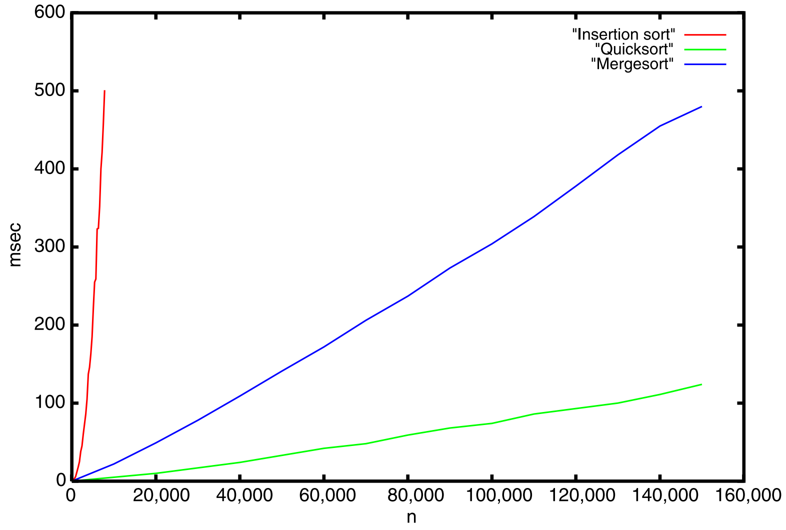


Fig. 3.2. Runtimes in milliseconds of the three algorithms determined experimentally for sorting sequences of lengths 1 to 150,000

In half a second (500 ms) of computation time, *Sorting by insertion* can sort sequences of length up to 8,000, whereas *Mergesort* manages 20 times as many numbers in the same time. *Quicksort* is four times faster than *Mergesort*.

3.4 Determining the Runtimes Theoretically

As in Chap. 2, it is possible to determine with mathematical methods how the runtimes of the algorithms depend on the number n of elements to be sorted, without having to program the algorithms and measure the time on a computer. These methods show that a simple sorting algorithm, such as *Sorting by insertion*, has runtime proportional to n^2.

Let us now carry out a similar theoretical estimate of the runtime (also called *runtime analysis*) for *Mergesort*.

First, let us think about how many comparisons are needed for step 3 of the algorithm, the *merging* of two sorted subsequences of length $n/2$ into one sorted sequence of length n. The merging procedure first compares the two lowest cards of each subsequence, and then the new complete stack is started with the smaller of the two. Then we proceed with the two remaining stacks in the same manner. In each step two cards are compared and the smaller one is put on the complete stack. Since the complete stack consists of n cards in the end, at most n comparisons were carried out (exactly, no more than $n - 1$).

In order to consider the recursive structure of the entire algorithm let us once again look at the tree in Fig. 3.1.

The master at the top has to sort 16 cards. He gives eight to each of the two helpers; they both give four to each of their two helpers; and so on. The master at the top in step 3 has to merge two times eight (in general, two times $n/2$) cards to a complete sorted sequence of length 16 (n). This takes, as we saw before, at most 16 (n) comparisons. The two helpers at the level below merge $n/2$ cards each, so they need at most $n/2$ comparisons each, so together at most n, as well. Likewise, the four helpers at third level merge $n/4$ cards each and together again need at most n comparisons; and so on.

So, it can be seen that for each level of the tree at most n comparisons are necessary. It remains to calculate the number of levels. The figure shows that for $n = 16$ there are four levels. We can see that when descending down the tree, the length of the subsequences to be sorted decreases from n at the highest level to $n/2$ at the second level, and further to $n/4$, $n/8$, and so on. So, it is cut in half from level to level until length 1 is reached at the lowest level. Therefore, the number of levels is the number of times n can be divided by 2 until 1 is reached. This number is known to be (cf. also Chap. 1) the base 2 *logarithm* of n, $\log_2(n)$. Since for each level at most n comparisons are necessary, altogether *Mergesort* needs at most $n \log_2(n)$ comparisons to sort n numbers.

For simplicity, we assumed in our analysis that the length n of the input sequence always can be divided by 2 without a remainder until 1 is reached. In other words, n is a power of 2, i.e., one of the numbers $1, 2, 4, 8, 16, \ldots$. For other values of n, *Mergesort* can be analyzed with some more effort. The idea remains the same and the result is that the number of comparisons is at most $n\lceil \log_2(n) \rceil$. Here, $\lceil \log_2(n) \rceil$ is $\log_2(n)$ rounded up to the smallest following integer.

Here, we only estimated the number of comparisons. If this number is multiplied by the time that the computer running the algorithm needs for a comparison,[3] one gets the total time needed for comparisons. This value is not yet the total runtime, since besides comparisons other operations, such as for restoring the elements to be sorted and for the organization of the recursion, are needed. Nevertheless, it can be analyzed that the total runtime is *proportional* to the number of comparisons. So, by our analysis, we know at least that the runtime for *Mergesort* is proportional to $n \log_2(n)$.

These considerations explain the superiority of *Mergesort* over *Sorting by insertion* that we observed in the previous section. For that algorithm the number of comparisons is $n(n-1)/2$, as derived in Chap. 2. Indeed, this function grows much faster than the function $n \log_2(n)$.

For *Quicksort* the situation is more complicated. It can be shown that for certain inputs, e.g., if the input sequence is already sorted, its runtime can be very large, i.e., proportional to n^2. You may get an impression why this is the case if you follow the algorithm "by hand" on such an input. This case,

[3] For a comparison of two integers a modern computer needs about one nanosecond, i.e., one billionth of a second.