

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



Algorithms Unplugged



Algorithms Unplugged

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

Algorithms Unplugged



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

e-ISBN 978-3-642-15328-0

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

<i>Martin Dietzfelbinger and Christian Scheideler</i>	3
---	---

1 Binary Search

<i>Thomas Seidl and Jost Enderle</i>	5
--	---

2 Insertion Sort

<i>Wolfgang P. Kowalk</i>	13
-------------------------------------	----

3 Fast Sorting Algorithms

<i>Helmut Alt</i>	17
-----------------------------	----

4 Parallel Sorting – The Need for Speed

<i>Rolf Wanka</i>	27
-----------------------------	----

5 Topological Sorting – How Should I Begin to Complete My To Do List?

<i>Hagen Höpfner</i>	39
--------------------------------	----

6 Searching Texts – But Fast! The Boyer–Moore–Horspool Algorithm

<i>Markus E. Nebel</i>	47
----------------------------------	----

7 Depth-First Search (Ariadne & Co.)

<i>Michael Dom, Falk Hüffner, and Rolf Niedermeier</i>	57
--	----

8 Pledge’s Algorithm

<i>Rolf Klein and Tom Kamphans</i>	69
--	----

9 Cycles in Graphs	
<i>Holger Schlingloff</i>	77
10 PageRank – What Is Really Relevant in the World-Wide Web?	
<i>Ulrik Brandes and Gabi Dorfmüller</i>	89
<hr/>	
Part II Arithmetic and Encryption	
Overview	
<i>Berthold Vöcking</i>	99
11 Multiplication of Long Integers – Faster than Long Multiplication	
<i>Arno Eigenwillig and Kurt Mehlhorn</i>	101
12 The Euclidean Algorithm	
<i>Friedrich Eisenbrand</i>	111
13 The Sieve of Eratosthenes – How Fast Can We Compute a Prime Number Table?	
<i>Rolf H. Möhring and Martin Oellrich</i>	119
14 One-Way Functions. Mind the Trap – Escape Only for the Initiated	
<i>Rüdiger Reischuk and Markus Hinkelmann</i>	131
15 The One-Time Pad Algorithm – The Simplest and Most Secure Way to Keep Secrets	
<i>Till Tantau</i>	141
16 Public-Key Cryptography	
<i>Dirk Bongartz and Walter Unger</i>	147
17 How to Share a Secret	
<i>Johannes Blömer</i>	159
18 Playing Poker by Email	
<i>Detlef Sieling</i>	169
19 Fingerprinting	
<i>Martin Dietzfelbinger</i>	181
20 Hashing	
<i>Christian Schindelhauer</i>	195
21 Codes – Protecting Data Against Errors and Loss	
<i>Michael Mitzenmacher</i>	203

Part III Planning, Coordination and Simulation

Overview

Helmut Alt and Rüdiger Reischuk 221

22 Broadcasting – How Can I Quickly Disseminate Information?

Christian Scheideler 223

23 Converting Numbers into English Words

Lothar Schmitz 231

24 Majority – Who Gets Elected Class Rep?

Thomas Erlebach 239

25 Random Numbers – How Can We Create Randomness in Computers?

Bruno Müller-Clostermann and Tim Jonischkat 249

26 Winning Strategies for a Matchstick Game

Jochen Könemann 259

27 Scheduling of Tournaments or Sports Leagues

Sigrid Knust 267

28 Eulerian Circuits

Michael Behrisch, Amin Coja-Oghlan, and Peter Liske 277

29 High-Speed Circles

Dominik Sibbing and Leif Kobbelt 285

30 Gauß–Seidel Iterative Method for the Computation of Physical Problems

Christoph Frendl and Ulrich Rüde 295

31 Dynamic Programming – Evolutionary Distance

Norbert Blum and Matthias Kretschmer 305

Part IV Optimization

Overview

Heribert Vollmer and Dorothea Wagner 315

32 Shortest Paths

Peter Sanders and Johannes Singler 317

33 Minimum Spanning Trees (Sometimes Greed Pays Off ...)	
<i>Katharina Skutella and Martin Skutella</i>	325
34 Maximum Flows – Towards the Stadium During Rush Hour	
<i>Robert Görke, Steffen Mecke, and Dorothea Wagner</i>	333
35 Marriage Broker	
<i>Volker Claus, Volker Diekert, and Holger Petersen</i>	345
36 The Smallest Enclosing Circle – A Contribution to Democracy from Switzerland?	
<i>Emo Welzl</i>	357
37 Online Algorithms – What Is It Worth to Know the Future?	
<i>Susanne Albers and Swen Schmelzer</i>	361
38 Bin Packing or “How Do I Get My Stuff into the Boxes?”	
<i>Joachim Gehweiler and Friedhelm Meyer auf der Heide</i>	367
39 The Knapsack Problem	
<i>Rene Beier and Berthold Vöcking</i>	375
40 The Travelling Salesman Problem	
<i>Stefan Näher</i>	383
41 Simulated Annealing	
<i>Peter Rossmanith</i>	393
Author Details	401

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.

Binary Search

Thomas Seidl and Jost Enderle

RWTH Aachen University, Aachen, Germany

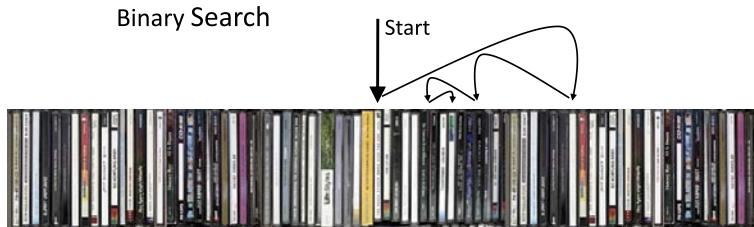
Where has the new Nelly CD gone? I guess my big sister Linda with her craze for order has placed it in the CD rack once again. I've told her a thousand times to leave my new CDs outside. Now I'll have to check again all 500 CDs in the rack one by one. It'll take ages to go through all of them!

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.





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”

```

1 function BINARYSEARCH (A, key, left, right)
2 while left ≤ right do
3     middle := (left + right)/2      // find the middle, round the result
4     if A[middle] = key then return middle
5     if A[middle] > key then right := middle - 1
6     if A[middle] < key then left := middle + 1
7 endwhile
8 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:

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

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

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

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:

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

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 :

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

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

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

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”

```

1  function BINSEARCHRECURSIVE (A, key, left, right)
2  if left > right return not found
3  middle := (left + right)/2      // find the middle, round the result
4  if A[middle] = key then return middle
5  if A[middle] > key then
6      return BINSEARCHRECURSIVE (A, key, left, middle - 1)
7  if A[middle] < key then
8      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:

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

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

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

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.,

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

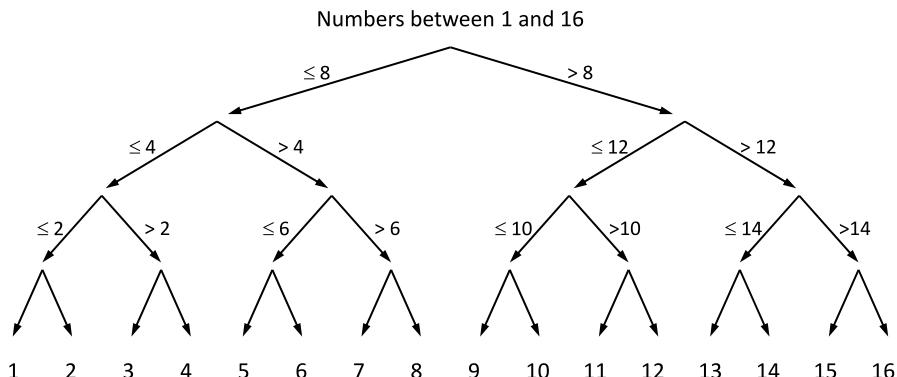
numbers in the binary system are represented as sums of multiples of powers of 2:

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

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\)](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)

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.

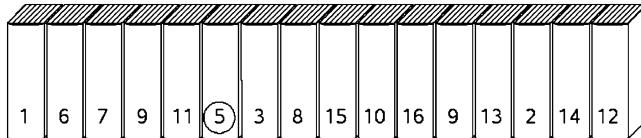
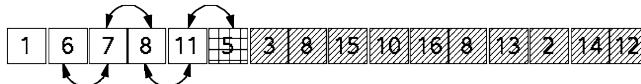
How to achieve this quickly? We can use several concepts. For example, we can look at each book one after the other, and if two subsequent books are out of order we exchange them. This works since finally no two books are out of order, but it takes, on average, a very long time. Another concept looks 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

**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 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, Then $A[i]$ means the value at position i of array A . To sort n books requires an array of length n with cells $A[1], A[2], A[3], \dots, A[n - 1], A[n]$ to store all book titles. Then the algorithm looks like this:

SUBSEQUENT BOOKS ARE EXCHANGED:

```

1 Given:  $A$ : Array with  $n$  cells
2 for  $i := 2$  to  $n$  do
3    $j := i$ ; // book at position  $i$  is current
      as long as correct position not achieved
4   while  $j \geq 2$  and  $A[j - 1] > A[j]$  do
5      $Hand := A[j]$ ; // exchange current book with left neighbor
6      $A[j] := A[j - 1]$ ;
7      $A[j - 1] := Hand$ ;
8      $j := j - 1$ 
9   endwhile
10 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

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

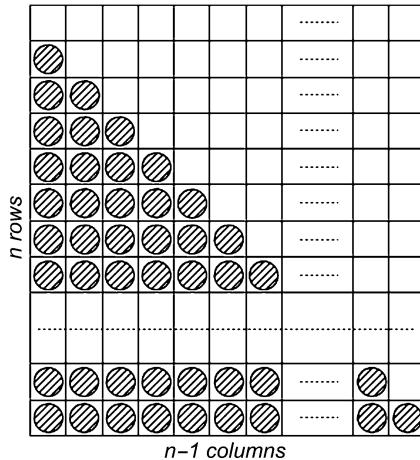


Fig. 2.3. Compute the number of exchanges

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:

```

1  Given:  $A$ : array with  $n$  cells;
2  for  $i := 2$  to  $n$  do
3      // sort book at position  $i$  by shifting
4       $Hand := A[i]$ ;    // take current book
5       $j := i - 1$ ;
6      // as long as current position not found
7      while  $j \geq 1$  and  $A[j] > Hand$  do
8           $A[j + 1] := A[j]$ ;    // shift book right to position  $j$ 
9           $j := j - 1$ 
10     endwhile
11      $A[j] := Hand$     // insert current book at correct position
12 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).

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.

3.1 The Algorithms

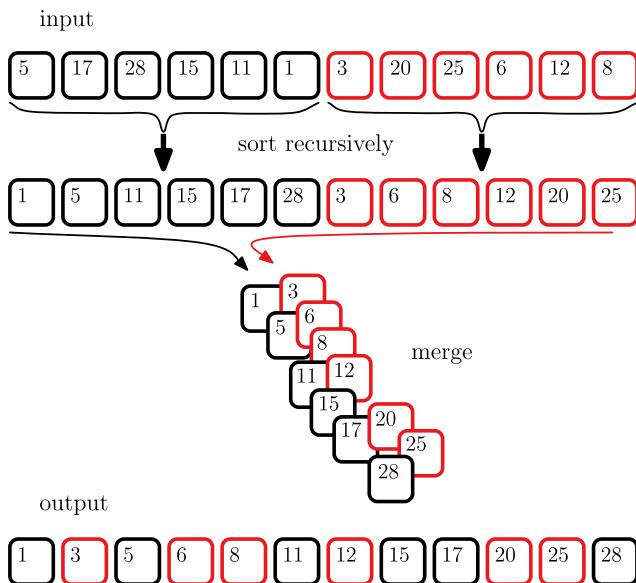
For simplicity, imagine that you receive from a master a stack of cards each of which has a number written on it. You are supposed to sort these cards in the order of ascending numbers and give them back to the master.

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:

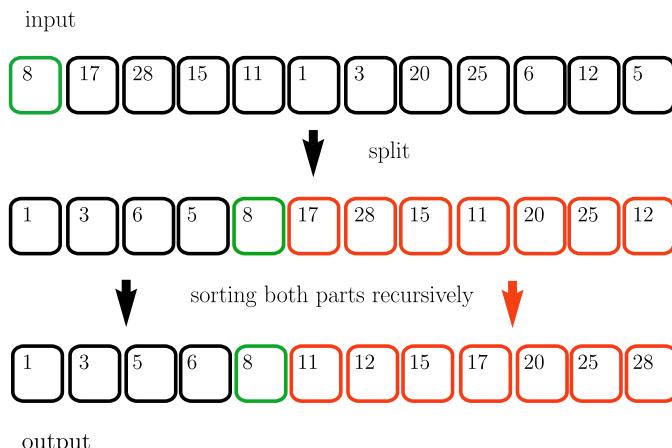


The second algorithm solves the same problem in a completely different manner:

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

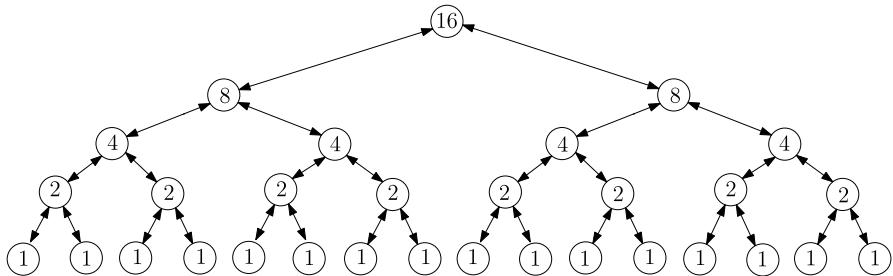


Fig. 3.1. Recursion tree for *Mergesort*

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*.

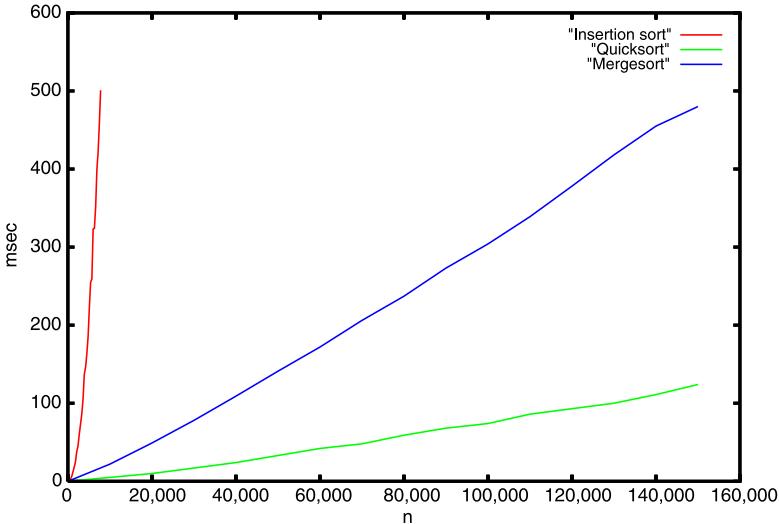


Fig. 3.2. Runtimes in milliseconds of the three algorithms determined experimentally for sorting sequences of lengths 1 to 150,000

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, 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.

however, only occurs if the element x to split the sequence, the so-called *pivot*, is the first or the last element in sorted order. If, instead, a *random* element from the sequence is chosen, then the probability that the algorithm is slow is very small. On average, the runtime is also proportional to $n \log_2(n)$. And, as our experiments show, the constant factor in front of $n \log_2(n)$ is obviously better than that in *Mergesort*. In practice, *Quicksort* is indeed the fastest sorting algorithm, as has also been demonstrated by our experiments in the previous section.

3.5 Implementation in Java

By the descriptions in Sect. 3.1 the algorithms are already well defined and well explained. Nevertheless, for readers familiar with the programming language Java who are interested in the technical details, we will, in addition, give the implementations of the algorithms. In fact, both algorithms are offered by Java and can be easily used. *Mergesort* can be found in the class “Collections” under the name “Collections.sort” and *Quicksort* can be found in the class “Arrays” under the name “Arrays.sort.” These methods can be used not only for numbers but also for arbitrary objects that are pairwise comparable.

Here, however, we will show self-written and easier to understand methods for integers. Also, these programs were used for the measurements in Sect. 3.3. One call of the method is always applied to the parts of an array A whose boundaries are given.

Let us look at *Mergesort* first. We show first the method to merge two sorted sequences into one sorted sequence:

```
public static void merge (int[] A, int al, int ar,
                        int[] B, int bl, int br,
                        int[] C)
    // merges a sorted array-Segment A[al]...A[ar] with
    // B[bl]..B[br] to a sorted segment C[0] ...
    {
        int i = al, j = bl;
        for(int k = 0; k <= ar-al+br-bl+1; k++)
            {
                if (i>ar)      // A is finished
                    {C[k]=B[j++]; continue;}
                if (j>br)      // B is finished
                    {C[k]=A[i++]; continue;}
                C[k] = (A[i]<B[j]) ? A[i++]:B[j++];
            }
    }
```

Now *Mergesort* itself can be easily written as a method in Java:

```

public static void mergeSort (int[] A, int al, int ar)
{ // sorts the array-Segment A[al] to A[ar]

    if(ar>al) {int m = (ar+al)/2;

        // recursive sorting of the halves:
        mergeSort(A,al,m);
        mergeSort(A,m+1,ar);

        // merging into array B :
        int[] B = new int[ar-al+1];
        merge(A,al,m, A,m+1,ar, B);

        // storing back into A:
        for(int i=0;i<ar-al+1;i++) A[al+i] = B[i];
    }
}

```

The program can be made even faster by saving the storing of array B to A and applying the recursive calls alternatively to A and B . For simplicity we didn't do that here.

Quicksort has an additional advantage over *Mergesort* by virtue of its not needing an auxiliary array B but only the array A , which contains the data. The splitting (step 2 of the algorithm) is done by using a “pointer variable” i . i starts at the beginning of the segment to be sorted and stops as soon as an $A[i]$ has been found which is greater than the pivot, i.e., it doesn't belong into the left half. At the same time variable j starts from the right end of the segment going left and stops at elements $A[j]$ that are smaller than the pivot. If both pointers stop, $A[i]$ and $A[j]$ are swapped and the run continues until both pointers meet.

```

public static void swap (int[] A, int i, int j)
{int t = A[i]; A[i] = A[j]; A[j]=t;}

public static void quickSort (int[] A, int al, int ar)
// sorts the segment A[al],...,A[ar]
{if(al<ar)
{
    int pivot = A[al], // 1st element as pivot
        i=al, j=ar+1;

    // splitting:
    while(true)
    {   while (A[++i] < pivot && i<ar){}
        while (A[--j] > pivot && j>al){}
}
}

```

```

        if (i<j) swap(A,i,j);
        else
            break;
    }
    swap(A,j,al);

    quickSort(A,al,j-1);
    quickSort(A,j+1,ar);
}
}

```

Further Reading and Experiments

For animations of the algorithms presented here, you can search the Internet. In particular, we recommend the following pages:

<http://math.hws.edu/TMCM/java/xSortLab/>
<http://www.cs.ubc.ca/~harrison/Java/sorting-demo.html>
<http://cg.scs.carleton.ca/~morin/misc/sortalg/>
<http://www.tcs.ifi.lmu.de/~gruberh/lehre/sorting/sort.html>

On some of those pages, other sorting algorithms and programs in a high-level programming language are also given.

Sorting by insertion is contained in most of the pages; its runtime is proportional to n^2 , just as in the case of the frequently presented algorithm *Bubblesort*. Even for small input sequences with 100 or 200 objects to be sorted, the superiority of *Mergesort* and *Quicksort* can be recognized clearly.

Parallel Sorting – The Need for Speed

Rolf Wanka

Universität Erlangen-Nürnberg, Erlangen, Germany

Since the early days of the development of “general” computing machines, there has been the idea to also build dedicated devices that are capable of solving the sorting problem (already addressed in Chaps. 2 and 3) exceptionally fast. In this chapter, we present a solution of the sorting problem that is well suited to be implemented as special purpose hardware on a microchip. It is a so-called *parallel* sorting algorithm.



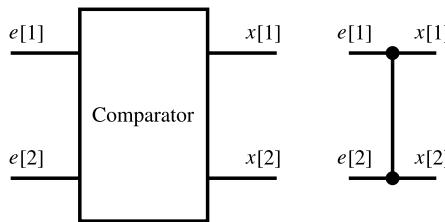
(c) Heinz Nixdorf MuseumsForum, Paderborn

When in the 1890s Herman Hollerith built his famous tabulating machine in order to evaluate the US Census, he also engineered and built an additional device used to sort the punch cards that stored the collected data. In the picture above, we see an original *Hollerith machine*. The “small” device on the right is a punch card sorting machine. Of course, the cable we see is not for data transmission, but for electrical power supply. The punch cards get sorted

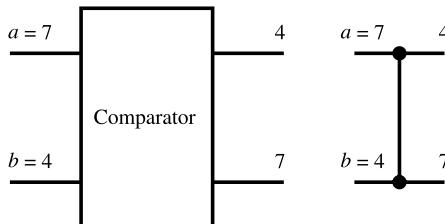
by the “sorting by insertion” method we have already encountered in Chap. 2. In the era of very large-scale integrated circuits, sorting units of course do not sort punch cards any more, but sort data stored as bits and bytes. Now we are looking for sorting algorithms that can be realized by modern microchips.

Sorting in Hardware: Comparators and Sorting Circuits

In the following, we present the construction of a hardware sorter. On n wires, it gets an arbitrarily mixed sequence of n non-negative integer numbers we call *keys*. All keys are available simultaneously. We want that the sorter consist of just one kind of module: *Comparator*. A comparator has two inputs, $e[1]$ and $e[2]$, and two outputs, $x[1]$ and $x[2]$. Two arbitrary keys, a and b , enter the comparator, and, as the output of the comparator, $x[1]$ receives the smaller key, i.e., $x[1] = \min\{a, b\}$, and $x[2]$ receives the larger key, i.e., $x[2] = \max\{a, b\}$. The following figure shows two ways to draw a comparator. In the rest of this chapter, we shall use the right, more compact picture. For our purposes, we ignore how a comparator is electronically realized.

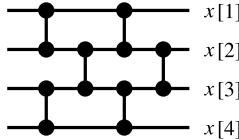


Thus, the input keys $a = 7$ and $b = 4$ will be processed as follows:



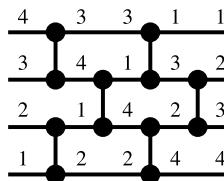
If we have only a single comparator, we may use it to implement the conditional exchange operations in the already introduced algorithms MERGESORT and QUICKSORT (see Chap. 3). However, as we only have a single comparator, all required conditional exchange operations must be executed one after another, i.e., *sequentially*.

Now we design a circuit that consists of many copies of comparators. It can sort any sequence of n keys much faster than sequential algorithms. We start with a small, but instructive example of such a circuit consisting of comparators only. Study the following figure.



The input of length 4 arrives at the left. It passes through the circuit to the right. Another term often used instead of circuit is *network*. The following simple arguments show that the circuit above consisting of six comparators can sort any sequence consisting of four keys: No matter on which wire the minimum key will enter the circuit on the left, it will always leave it on the upmost wire $x[1]$. Analogously, the maximum key will always leave the circuit on the lowermost wire $x[4]$, no matter which wire was its input wire. Finally, we see that the last comparator guarantees $x[2] \leq x[3]$. Hence, we conclude that this circuit sorts any input sequence. Therefore, it is called a *sorting circuit*.

The next figure shows how the input sequence $(4, 3, 2, 1)$ is processed by this circuit. Note that in every step an exchange is actually executed. This means that in this circuit no comparator is redundant.



We also learn from this figure that all comparators that are drawn one below the other may be executed simultaneously. So only four time units will elapse until the input becomes sorted. Rather than speak of time units, we speak of *parallel steps*.

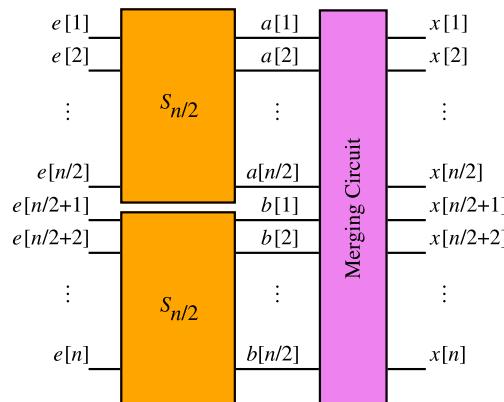
The Bitonic Sorting Circuit: Its Architecture

Can we implement the sequential (non-parallel) sorting algorithms MERGE-SORT and QUICKSORT from Chap. 3 by comparator circuits because, after all, they also apply conditional exchange operations as their basic operations?

Unfortunately, it is not possible in an immediate way. For MERGESORT and QUICKSORT we do not know in advance which index positions will be involved in a late conditional exchange operation. These indices depend on the input sequence, or, more exactly, on the results of previously executed comparisons! This is not allowed for comparator circuits. Here, we have to specify in advance, prior to any input, a fixed circuit.

In 1968, Kenneth Batcher, a computer scientist at Kent State University, designed a concrete comparator circuit that can sort any input sequence of length n in very few, namely $\frac{1}{2} \cdot \log_2 n \cdot (\log_2 n + 1)$, parallel steps. That means this circuit sorts $2^{20} = 1,048,576$ keys in just $\frac{1}{2} \cdot 20 \cdot 21 = 210$ parallel steps. The approach is *divide-and-conquer*, which we have already seen in Chap. 3.

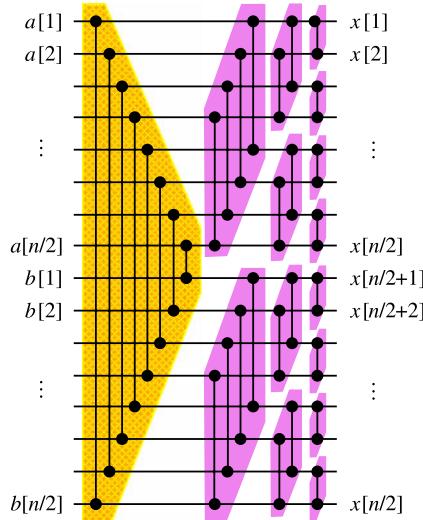
Assume we want to sort n keys. For this task, we now design a sorting circuit S_n . Because in our divide-and-conquer approach we partition the input sequence repeatedly into two equal-sized subsequences, we assume for simplicity that $n = 2^k$ for some integer k . So, we can divide n by 2 without a remainder again and again. Now, suppose we know how to sort $\frac{n}{2} = 2^{k-1}$ keys with the help of circuit $S_{\frac{n}{2}}$. Then, with the help of two copies of $S_{\frac{n}{2}}$, we at first sort the upper half-sequence and the lower half-sequence of the input. This is the divide part of the divide-and-conquer approach.



The figure above shows how our circuit has to work. It is called the *architecture* of the circuit. The interiors of the boxes consist of comparator circuits that we still have to design.

The two “half-sized” copies of $S_{\frac{n}{2}}$ generate the sequences $a[1], \dots, a[\frac{n}{2}]$ and $b[1], \dots, b[\frac{n}{2}]$, respectively. As the conquer step, as in MERGESORT in Chap. 3, we have to solve the task of *merging*. That means we have to design a *merging circuit* that receives as input the two sorted sequences $a[1], \dots, a[\frac{n}{2}]$ and $b[1], \dots, b[\frac{n}{2}]$, and outputs the overall sorted sequence $x[1], \dots, x[n]$. For this task, Kenneth Batcher invented a circuit he dubbed *Bitonic Merger*. Its architecture is presented in the following figure. The reason for the name will

be clear soon when we analyze the Bitonic Merger. This circuit will be inserted in the architecture of S_n presented above in the violet box labeled “Merging Circuit.”



The Bitonic Merger always begins with the step highlighted in yellow (left “triangle”; it is really only one single parallel step; the comparators are drawn in this “nice” way to have a clear representation). Then, the sequence of steps highlighted in violet (the “rhomboids”) are executed. In general, the architecture of the Bitonic Merger consists of the yellow triangle and a sequence of violet rhomboids, where the height of the rhomboids is halved in every successive step. It is a good idea if the reader draws the Bitonic Merger for $n = 32$.

The Bitonic Sorting Circuit: Its Correctness and Running Time

It is not obvious at all that the two sorted sequences $a = (a[1], \dots, a[\frac{n}{2}])$ and $b = (b[1], \dots, b[\frac{n}{2}])$ are really correctly merged by the Bitonic Merge circuit, i.e., that the output sequence is sorted. In order to prove this, we use a nice property of comparator circuits, the so-called 0-1 principle:

If and only if a comparator circuit sorts any sequence of length n that consists **only of 0s and 1s**,
then it sorts any sequence of length n **of arbitrary keys**.

This means that we can prove the correctness of a sorting circuit by proving that it just sorts all possible 0-1 inputs.

In what follows we present a sketch of its proof emphasizing the idea. For the proof of correctness of the Bitonic Sorter, we only need the statement of the 0-1 principle, not its proof. Therefore during first reading, the reader may skip the following short text and resume reading at (**).

The idea of the proof of the 0-1 principle is quite simple. In order to illustrate it, we show instead of the 0-1 principle the 0-1-2-3-4 principle. Here, in the above statement we replace “only of 0s and 1s” with “only of 0s, 1s, 2s, 3s, and 4s.”

Now let C_n be an arbitrary, but fixed comparator circuit with n wires. Consider an arbitrary sequence $a = (a[1], \dots, a[n])$ of the numbers 1 through n . Every number appears exactly once. Such a sequence is called a *permutation*. Let a be the input to C_n . We pick two different keys i and j , $i < j$, from a and construct the sequence $b = (b[1], \dots, b[n])$ with

$$b[k] = \begin{cases} 0 & \text{if } a[k] < i \\ 1 & \text{if } a[k] = i \\ 2 & \text{if } i < a[k] < j \\ 3 & \text{if } a[k] = j \\ 4 & \text{if } j < a[i]. \end{cases}$$

That means that in b , all keys less than i are mapped to 0, i is mapped to 1, all keys between i and j are mapped to 2, j is mapped to 3, and all keys greater than j are mapped to 4. For example, $a = (6, 1, 5, 2, 3, 4, 7)$ with $i = 3$ and $j = 5$ is transformed to $b = (4, 0, 3, 0, 1, 2, 4)$.

Now, a and b are fed into C_n . Mark in C_n the paths of i and j on their way from left to right in red and blue, respectively. Then compare the red and blue paths to the paths of 1 and 3 when b is input to the circuit. We see that 1 takes the red path, and that 3 takes the blue path! Why? If we only have a single comparator (although n is arbitrary), this is obviously true. And an arbitrary comparator circuit can be considered a successive application of single comparators. So it is true for any circuit. Hence, i from a will be output on the same wire as 1 from b , and j from a will be output on the same wire as 3 from b .

Now suppose that there is a sequence a that is not sorted by C_n . Then there are two keys i and j , $i < j$, that are output in wrong order. So, the corresponding sequence b is also not sorted. This means the other way around that, if all 0-1-2-3-4 sequences are sorted by C_n , there can be no permutation which is not sorted by C_n .

Now it is just a small step to the 0-1 principle: In the construction of b , replace the keys 0, 1, and 2 with 0, and 3 and 4 with 1. Call this sequence c . In our example, this yields $c = (1, 0, 1, 0, 0, 0, 1)$. A close look reveals that on the same wire where i from a is output, a 0 from c is output. Analogously, where j from a is output, a 1 from c is output. And it is easy to see that all arguments for b and c also hold if a is not a permutation. So we may repeat the argument: For every input sequence a that is not sorted, there is a 0-1

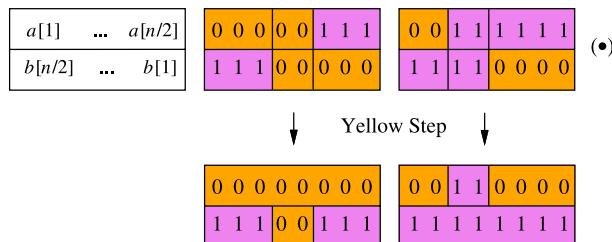
input sequence c that is also not sorted. Hence, if any 0-1 input sequence is sorted, any arbitrary input sequence is also sorted.¹

(**) So, for the time being, we consider only those input sequences that consist of only 0s and 1s. Now we can show that the Bitonic Merger transforms the sorted 0-1 sequences a and b into a sorted overall sequence. This is the moment where the term *bitonic* becomes important. Bitonic sequences arise if a monotically increasing 0-1 sequence x (this is a sequence where the numbers, considered from left to right, never get larger) and a monotically decreasing 0-1 sequence y (this is a sequence where the numbers, considered from left to right, never get smaller) are glued together in arbitrary order. That means both xy and yx are bitonic 0-1 sequences.

Sounds difficult? Not really! Consider some examples: 00111000, 11100011, 0000, 11111000, and 11111111 are bitonic sequences. I am sure that you can easily find the necessary sequences x and y , can't you? There are even many possibilities to choose such sequences!

Loosely speaking, a 0-1 sequence is bitonic if it goes up and down or if it goes down and up.

What is the outcome of the yellow step (the triangle), i.e., the first parallel step of the Bitonic Merger? Let us reverse the sequence b and place it underneath a . Then we have the following picture.



Check that the comparators of the yellow step are applied to keys listed one below the other in (\bullet) . That means that the smaller of the two keys will afterwards be in the upper row, and the larger key will be in the lower row. The colored parts of the figure show two examples. We see that when the yellow step has been applied, at least one half-sized sequence consists only of 0s or 1s. And the remaining sequence is bitonic! This bitonic sequence is the input for the violet steps (the rhomboids). Now we slit the bitonic sequence in the center and place the two sequences one above the other. The same figure as above shows up, but of only half the size, and the same happens again to the keys in the upper and lower rows! It is very instructive to try this out for

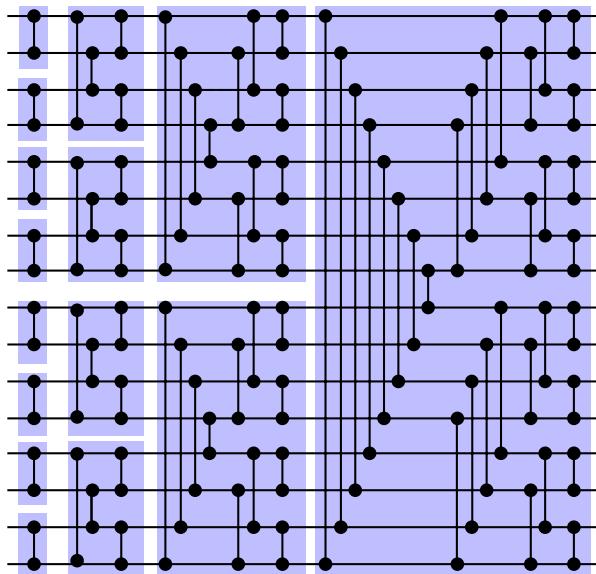
¹ A very compact, full proof of the 0-1 principle can be found in Donald Knuth's book *The Art of Computer Programming*, Vol. 3: *Sorting and Searching* (see the end of this chapter on "Further Reading") on p. 223.

some examples. In the end, after the last violet step, the whole 0-1 sequence is sorted.

Now let us consider the shape of $S_{\frac{n}{2}}$ used in the architecture of S_n . Also, $S_{\frac{n}{2}}$ ends with a Bitonic Merger this time for $\frac{n}{2}$ input wires. The inputs of this Bitonic Merger are two copies of $S_{\frac{n}{4}}$. This is repeated until the sorter that sends its output to the Bitonic Merger is responsible for only two keys. Of course, for this we use a comparator. Altogether, we have the circuit S_{16} shown in the following figure. It sorts any 0-1 sequence of length 16, and hence, by the 0-1 principle, any arbitrary sequence of 16 keys! Here, every box highlighted in blue is a Bitonic Merger.

Of course, our arguments hold for all $n = 2^k$. S_n sorts all 0-1 sequences of length n and hence sorts arbitrary length- n sequences due to the 0-1 principle. The whole circuit is called *Bitonic Sorter*.

So, here we have S_{16} .



In order to discover the relation of our construction to the divide-and-conquer approach, the reader is invited to compare this circuit to the figure presenting the architecture of S_n and to search for the two copies of S_8 .

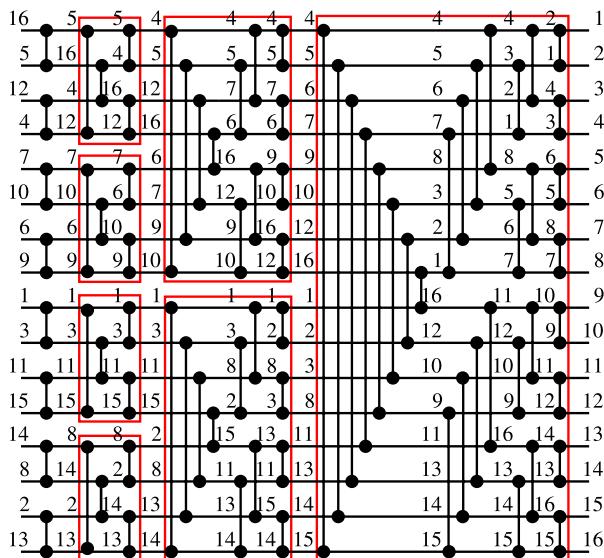
In the next figure, we see how a sequence of 16 keys is sorted by S_{16} . Consider the output sequences of the red boxes, i.e., of the Bitonic Mergers. We see: They are sorted!

Now that we have proven the correctness of the Bitonic Sorter, we conclude our analysis with the computation of the parallel running time and the number of comparators.

For the running time $t(n)$ of the Bitonic Sorter which is the number $t(n)$ of parallel steps, with $n = 2^k$, the figure above gives us

$$\begin{aligned} t(n) &= 1 + 2 + \cdots + (k - 1) + k = \sum_{i=1}^k i \\ &= \frac{1}{2} \cdot k \cdot (k + 1) = \frac{1}{2} \cdot \log_2 n \cdot (\log_2 n + 1). \end{aligned}$$

Compare this to the running time of the sequential MERGESORT from Chap. 3. There we learned that it is about $n \log_2 n$. Now, for Bitonic Sort, we may replace the factor of n in MERGESORT's running time with $\frac{1}{2} \cdot (\log_2 n + 1)$. This means for our example with $n = 2^{20}$ keys that the factor 1,048,576 is replaced with the factor 11.5. The parallel Sorter is faster than MERGESORT by a factor of almost 10,000. Every input sequence of length 2^{20} is already sorted after 210 parallel steps.



$$s(n) = \frac{n}{2} \cdot t(n) = \frac{1}{4} \cdot n \cdot \log_2 n \cdot (\log_2 n + 1).$$

For $n = 2^{20}$, this is a huge number, namely 110,100,480, but in modern VLSI technology, this is a realistic number. Note that a comparator has to be realized by an electronic circuit. It consists of more than one transistor.

Concluding Remarks

In this chapter, we presented a way to improve considerably the time for sorting with the help of a parallel hardware sorter. The price is an increase in the necessary hardware.

As we have seen, the number of parallel steps is proportional to $(\log_2 n)^2$. Is it possible to achieve a better running time? Yes! Miklós Ajtai, János Komlós, and Endre Szemerédi, three Hungarian scientists, designed a sorting circuit (see item 5 in “Further Reading” below), where the number of parallel steps is proportional to $\log_2 n$. In honor of its three inventors, the circuit is called the *AKS-sorting circuit*. Unfortunately, the actual proportionality factor, computed by Mike Paterson (see item 6 “Further Reading” below), is about 6,200. This means that the AKS-sorting circuit is better than the Bitonic sorter when $n \geq 2^{12,400}$, a number of truly astronomical magnitude!

Further Reading

1. Friedhelm Meyer auf der Heide, Rolf Wanka: *Von der Hollerith-Maschine zum Parallelrechner. Die alltägliche Aufgabe des Sortierens als Fortschrittsmotor für die Informatik*. ForschungsForum Paderborn (FFP) 3 (2000) 112–116 (in German).
<http://www.upb.de/cs/ag-madh/WWW/wanka/pubs/abstracts/FFP00ABS.html>
 This paper shows that many milestones in the development of Computer Science are closely related to the sorting problem, be it the Hollerith machine, the first computer program ever written, or the first randomized algorithm.
2. Chapter 3 (Fast Sorting Algorithms)
 Chapter 3 on fast sorting algorithms presents the merge sort approach that is also used by the Bitonic Sorter.
3. Kenneth E. Batcher: *Sorting networks and their applications*. In AFIPS Conf. Proc. 32, 307–314, 1968.
<http://www.cs.kent.edu/~batcher/>
<http://www.cs.kent.edu/~batcher/sort.ps>
 This is Kenneth Batcher’s seminal paper that introduces the Bitonic Sorter and a proof of its correctness. As in 1968 the 0-1 principle was not yet discovered, Batcher’s proof is a little bit more complicated than ours.

4. Donald E. Knuth: *The Art of Computer Programming*, Vol. 3: *Sorting and Searching*. Addison-Wesley, 2nd edition, 1998.

This is the classical text on sorting in general, and on sorting circuits in particular. Its Sect. 5.3.4 contains a wealth of information on sorting circuits.

5. Miklós Ajtai, János Komlós, Endre Szeméredi: *Sorting in $c \cdot \log n$ parallel steps*. Combinatorica, 3:1–19, 1983.

This paper describes the currently asymptotically “fastest” sorting circuit, the famous AKS-sorting circuit. Unfortunately, the constant c in the paper’s title is much too large for real-world applications of the circuit. Paterson (see item 6) presented a construction where the factor is brought down to 6,200.

6. Mike S. Paterson: *Improved sorting networks with $O(\log n)$ depth*. Algorithmica, 5:75–92, 1990.

Here, a considerably simplified construction of the AKS-sorting circuit is presented. This paper is very well written and fun to read (if you are a theoretical computer scientist). This variant of the AKS circuit “only” needs $6,200 \cdot \log_2 n$ parallel steps.

Topological Sorting – How Should I Begin to Complete My To Do List?

Hagen Höpfner

Bauhaus-Universität Weimar, Weimar, Germany

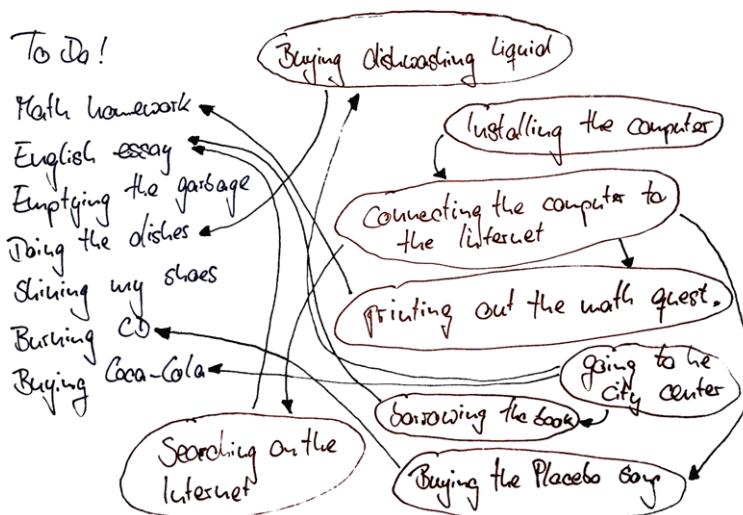
How on earth shall I handle all of this? I still have to do my homework for my mathematics class. Furthermore, I have to write an essay for my English class, but this requires that I first go to the library to borrow a book on the history of computer science. I also need to search for information on this topic in the Internet. This reminds me that, after coming back from the last LAN gaming session, I did not install and connect the computer again. By the way, I am quite sure that I did not download the math questionnaire so far. It is still waiting in my Google mail box. On top of this, we are having a party this evening which requires me to burn a music CD with my favorite artists. Needless to say this CD must contain the new single by Placebo which I first have to buy at iTunes.

This is a lot to do, but I also promised my mum to empty the garbage can, to shine my shoes and to do the dishes. Last but not least, I have to buy some Coca-Cola for the party. Luckily, the supermarket is on my way to the library, and I have to buy new dishwashing liquid anyway. But what shall I do first?

To Do!

Math homework
English essay
Emptying the garbage
Doing the dishes
Shining my shoes
Buying CD
Buying Coca-Cola

Well, it is certainly not possible to complete the tasks on my To Do list in the order they are written. The point is that I can't burn a CD without having all the songs, and in order to get all songs I have to install the computer and to connect it to the Internet first. Hence, there are some dependencies among the different tasks, and not all subtasks have been listed so far. That is why I now pick up my pen and complete my To Do list. Before doing the dishes I have to buy dishwashing liquid. Therefore, I draw an arrow from "buying dishwashing liquid" to "doing the dishes." In order to buy the dishwashing liquid I have to go to the city center. So, I draw an arrow from "going to the city center" to "buying dishwashing liquid," etc.



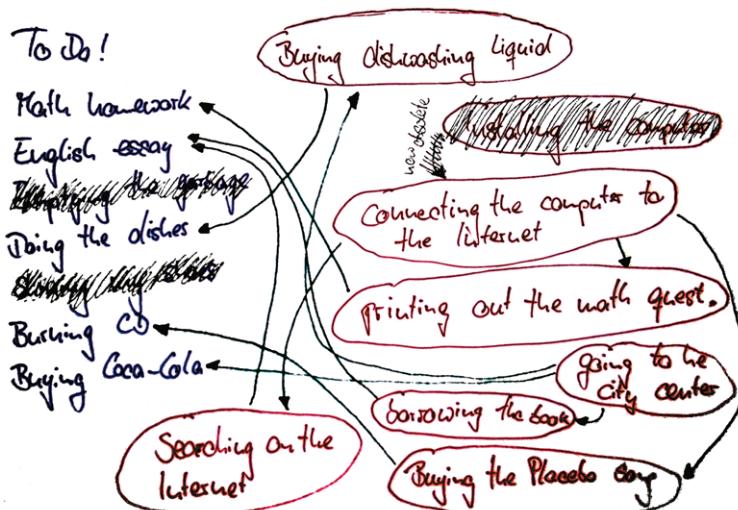
Wow, this is much worse than I thought! Where shall I start? This makes me aware that I have to do a lot. Anyway, the question remains: how shall I start off with the stuff? An arrow shows me that I have to do something before I can work on something else. Hence, I can only fulfill a task when no arrows point to it.

Very well then! I can only start with something that has no incoming arrows. Thus, I have only the following choices:

- emptying the garbage
- shining my shoes
- installing the computer
- going to the city center

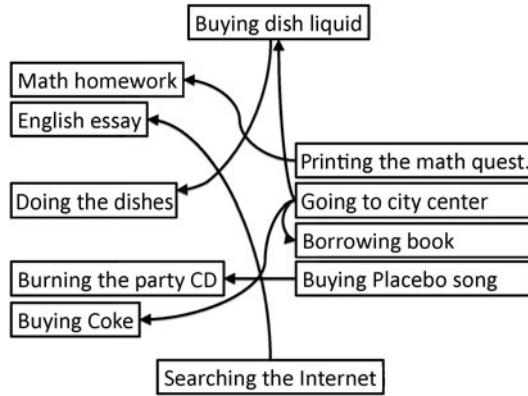
Actually it doesn't make any difference which of these four alternatives I choose. Well, I am a nice guy, and therefore first I empty the garbage.

Afterwards I'll shine my shoes before I install the computer. Following this I can update my To Do overview and remove the tasks I have done. At the same time, I can also remove the arrows that start at finished tasks (e.g. the arrow from "installing the computer" to "connecting the computer to the Internet").



Obviously, if I had used a pencil, the updated To Do overview would be clearer. Then I would have been able to erase the finished tasks and the canceled arrows. Never mind! The computer is running and I can draw an electronic task list, by simply using a graphic software. This reminds me that computer scientists like my brother call such a To Do list with sub-tasks and arrows a graph. The tasks are represented by nodes of the graph, and the dependencies are represented as directed edges between the nodes. Here "directed" means that the direction of the arrow defines the direction of reading the dependency. If it is possible to come back to the starting point while tracing (without removing the pen) such a graph, then the graph is a cyclic graph – in other words there is a cycle in the graph.

However, what shall I do next? Well, I could still go to the city center. As I have installed the computer already, I could also connect it to the Internet. In the end I removed the dependency arrow that pointed to "connecting the computer to the Internet". But, all other tasks are still blocked. Since I am working at the computer at the moment anyway, I can bring it online right now. Thus, my To Do graph changes again.

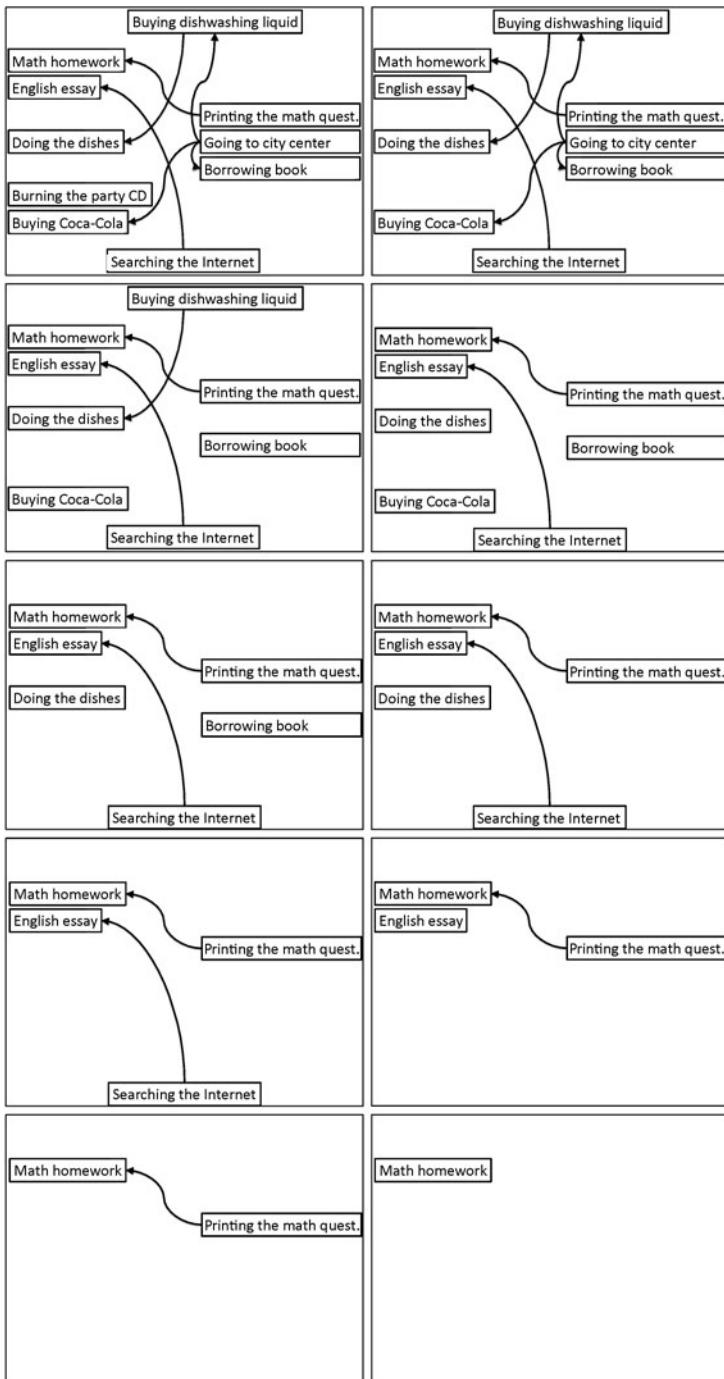


Subsequently I could still go to the city center, search online for the information that I need for my English essay, buy the Placebo song or print out my math questionnaire.

Done! In a few minutes I'll go to the party. Let me shortly summarize the order in which I finished my To Do list today:

1. emptying the garbage
2. shining my shoes
3. installing the computer
4. connecting the computer to the Internet
5. buying the Placebo song
6. burning the party CD
7. going to the city center
8. buying the dishwashing liquid
9. buying Coca-Cola
10. borrowing the book from the library
11. doing the dishes
12. searching for information on the Internet
13. writing the English essay
14. printing out the math questionnaire
15. answering the math questionnaire

After finishing a subtask, I always removed the entry and all arrows starting at this entry from my To Do graph. Hence, step by step I removed all nodes from the graph and saved the chosen sequence. You can read the results from top left to bottom right.



My big brother, who is studying computer science, told me a few minutes ago that I used topological sorting. He gave me the following algorithm description:

The TOPSORT algorithm outputs the nodes of a directed graph in a topological order. At this, the graph $G = (V, E)$ consists of the set of nodes V and a set of edges E of the form $(node1, node2)$, whereas the dependency is directed from $node1$ to $node2$ and V must contain both nodes.

```

1  function TOPSORT
2      while  $V$  is not empty do
3          cycle:=true
4          for each  $v$  in  $V$  do
5              if there is an edge  $e$  in  $E$  of the form  $(X, v)$  then
6                  //  $X$  is an arbitrary other node
7                  remove  $v$  from  $V$ 
8                  remove all edges of the form  $(v, X)$  from  $E$ 
9                  cycle:=false
10                 print  $v$     // printing out the nodes
11                 endif
12             endfor
13             if cycle=true then
14                 print I cannot resolve cyclic dependencies!
15                 break    // abort while loop
16             endif
17         endwhile
18     end
```

Moreover, the algorithm detects cyclic graphs that cannot be sorted topologically. This is done by checking whether or not each step removes one node. In the case that no node is removed before reaching an empty graph, the algorithm automatically stops.

Furthermore, the example used above illustrates a general computer problem. Computers do their jobs in a “stupid” way, step by step. TOPSORT aims at finding *one* possible topological order. Such a correct topological order would also be:

- ...
- going to the city center
- buying dishwashing liquid
- doing the dishes
- ...
- buying Coca-Cola
- ...

In this case we would have gone to the city center but would not have done all necessary shopping. The problem would be, though, that we would have to go to the city center again in order to buy Coca-Cola. However, this

information has already been removed from the graph. Hence, a little bit of organizing ability is still required to plan the daily routine.

Further Applications

Topological sorting finds an order that respects the direction of the edges. This happens independently of the situation represented by the graph and its nodes because the algorithm does not need to take this into account. The algorithm simply removes incoming and outgoing edges one by one. Therefore, it can be used in various areas of computer science. For example, it can help us to detect deadlocks that might result from parallel access to resources: If a program wants to exclusively use a resource (e.g., a file) in a computer, the resource gets locked and cannot be used by other programs. These programs must wait until the lock is released. A deadlock happens if a program that waits for a resource locks another resource that is being used by the first program. Hence, both programs wait for each other and neither of them can finish its task. It is possible to represent such a wait-for relationship in a wait-for graph. A deadlock leads to a cycle in the graph and can be detected using TOPSORT. In the end, one program participating in the deadlock must be aborted.

Additional Reading

1. From Wikipedia:
http://en.wikipedia.org/wiki/Topological_sorting

Searching Texts – But Fast! The Boyer–Moore–Horspool Algorithm

Markus E. Nebel

TU Kaiserslautern, Kaiserslautern, Germany

Within a computer’s memory many objects to be processed are represented in the form of text. A straightforward example is text generated by a word processing program. However, documents published on the Internet are usually hosted by a Web server in the form of so-called HTML documents, i.e., as text with integrated formatting instructions, links to image files, etc. In this chapter we will focus on the search of words within text. Why? Simply because of the many situations in which this problem is at hand. Imagine that we performed a Web search using Google and found a Web site with plenty pages of text. Of course, we want to know where in the text our search word can be found and we want our Web browser to perform the task of highlighting all the corresponding positions in the text. Accordingly, the browser needs a routine which finds all those occurrences as fast as possible. It should be obvious that we face the same or similar demands quite often. Therefore, we will deal with the so-called *string matching problem*, i.e., the problem of searching for all occurrences of a word w within a text t .

The Naive Algorithm

Within a computer, texts are stored symbol by symbol (letter by letter). Accordingly, it is not possible to compare a word w with a part of a text in a single step. In order to decide whether w occurs at a specific position we need to compare the text and w symbol by symbol. Assuming text t to consist of n symbols, we will denote by $t[i]$, i an integer between 1 and n , the n th symbol of t . Thus $t[1]$ denotes the first, $t[2]$ the second symbol, and so on. Finally, $t[n]$ represents the last symbol of the text. We will make use of the same notation for the symbols of w , assuming its length to be given by m . Then, when writing $w[j]$ to represent the j th symbol of word w , j must be

an integer between 1 and m . As an example consider the text **Haystack** with a needle in which we are searching for the word **needle**. In this case t and w look like the following (a column headed by number k contains the symbol $t[k]$ or $w[k]$):

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
H	a	y	s	t	a	c	k	w	i	t	h		a		n	e	e	d	l	e	

1	2	3	4	5	6
n	e	e	d	l	e

In our example, we have $n = 22$ and $m = 6$, and $t[1] = \text{H}$, $t[2] = \text{a}$, and $w[4] = \text{d}$. Please note that spaces of the text have to be considered as symbols too, and cannot be ignored. In the sequel, we will use this text as our running example.

In order to decide whether this text starts with the word $w = \text{needle}$, an algorithm must compare the beginning of t and w symbol by symbol. If all symbols match, we report success and the first occurrence of w is located at the first position of t . Obviously this is not the case for our example. In order to determine that, it is sufficient to compare $t[1]$ with $w[1]$, which yields a mismatch; $t[1] = \text{H} \neq w[1] = \text{n}$. By this mismatch our program concludes that w does not occur at the first position of t . Only if all m comparisons of the symbols of w to the corresponding symbols of t provide a match is w for sure contained in t . In our case, a single comparison is sufficient to observe the contrary, but obviously this is not the case in general. For example, consider the search of $w = \text{Hayrack}$ within t . In this case, even if w does not occur at the first position of t , the first three comparisons provide a match and we have to wait until the fourth comparison of $w[4]$ to $t[4]$ to determine a difference; s is different from r .

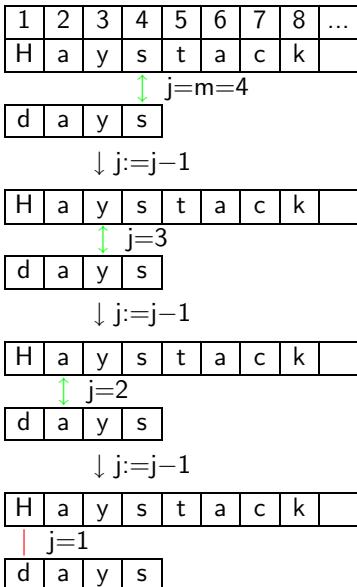
The following short program successively executes the comparisons just discussed. Contrary to our examples it starts with the last symbol of w instead of the first in order to compare w to a part of t from right to left. The reason for this will become clear later.

Comparing word w and text t at first position symbol by symbol

```

1   j := m;
2   while (j > 0) and (w[j] = t[j]) do
3       j := j - 1;
4   if (j = 0) then print("Occurrence at position 1");

```



In general,

```
while (j > 0) and (w[j] = t[j]) do j := j - 1;
```

means that j is decreased by 1 as long as it is larger than 0 and the j th symbol of the text is equal to the j th symbol of the word. Thus, in cases where the first m symbols of the text do not match w , the second condition eventually gets violated, leaving a value of j larger than 0. As a consequence, in line 4 of our program the command `if (j = 0) ...` will not report an occurrence (printing the text “Occurrence at position 1” is only a surrogate for any action to be taken in case of an occurrence of w). If, on the contrary, all m symbols of w match the first m symbols of t , then the while-loop terminates since $j = 0$ holds. In this case our program reports success.

Since we have to find all occurrences of w as a substring of t , we obviously have to search other locations of t than just the beginning. In fact, w may start at any position of t which has to be checked by our program. In this context, any position of t means that we have to expect an occurrence of w at the second, third, ... positions of t also. For the second position we must decide if $w[1] = t[2]$ and $w[2] = t[3]$ and ... and $w[m] = t[m + 1]$ hold. The third, fourth, ... positions have to be examined analogously; the $(n - m + 1)$ th position is the last to be considered where $w[m]$ and $t[n]$ are aligned. Considering position pos , we have to compare $w[1]$ and $t[pos]$, $w[2]$ and $t[pos + 1], \dots, w[m]$ and $t[pos + m - 1]$ (which our algorithm will do in reverse order). By introducing an additional variable pos we can easily extend our program to (according to our preliminary considerations) search for w at any position of t (parts of the program adopted from above are printed in blue).

The figure to the left clarifies the function of this little program when searching text t from above for $w = \text{days}$. Here, a green double arrow represents a comparison of two identical symbols; a read bar connects two symbols for which a mismatch has been observed. Beginning with $w[4]$, the text is compared symbol by symbol to w until either j becomes 0 (which is not the case in our example, and would imply an occurrence of w as part of the text) or the symbols $w[j]$ and $t[j]$ just compared do not match (which happens for $j = 1$ in our example). These two conditions are checked within the while-loop of the program.

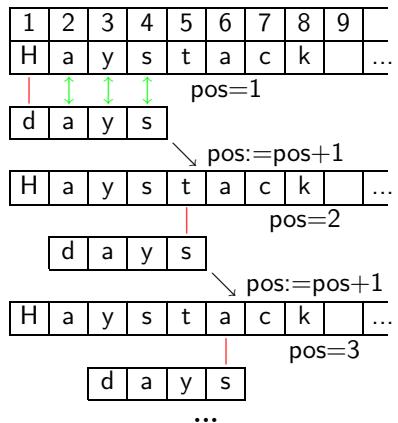
Naive String Matching Algorithm

```

1  procedure Naive
2  pos := 1;
3  while pos  $\leq n - m + 1$  do      // search all positions
4      j := m;
5      while (j > 0) and (w[j] = t[pos + j - 1]) do
6          j := j - 1;
7      if (j = 0) then print("Occurrence at position", pos);
8      pos := pos + 1;
9  wend;
10 end.

```

The outer while-loop at line 3 ensures that all positions where w might occur as a substring of t are indeed considered. The following figure clarifies the improvements of our algorithm:



For $pos = 1$ four comparisons are necessary: three matches and one mismatch. Again, those comparisons are realized by decreasing j step-by-step. Graphically speaking, by increasing pos by 1 afterwards, w is moved one position to the right. The first comparison performed there is unsuccessful; thus, pos is immediately increased (w moved one position to the right) again, and so on.

At this point we can give a first hint why comparing the word and the text from right to left is of advantage: As we will observe later, it is not always necessary to consider all positions of t (all possible values for pos). Some may be skipped without missing any occurrence of w . In such a case, comparing the word and the text from right to left allows for larger jumps without complicated calculations.

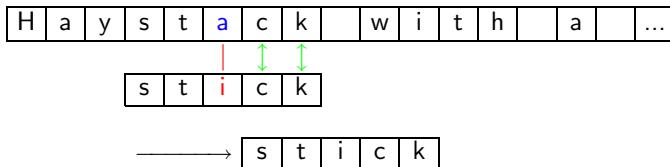
The algorithm presented before is a first solution to the string matching problem. Our program will report all occurrences of w as a substring of t in all circumstances (where else we should expect an occurrence if not at all

positions?). However, its runtime can be high since in the worst case we make about $(\text{number of symbols of } t) \times (\text{number of symbols of } w)$ comparisons to solve the problem. As an example, this situation occurs for $t = \text{aaaaaaaaaaaaaa}$ and $w = \text{baaa}$.

We can think of two cases. First, it might be impossible to find an algorithm which identifies all occurrences of w in t with less comparisons. In this case nothing is left but accepting this effort. Second, we just might not have been clever enough so far to find such an algorithm. As we will see in the subsequent section, the latter is indeed the case.

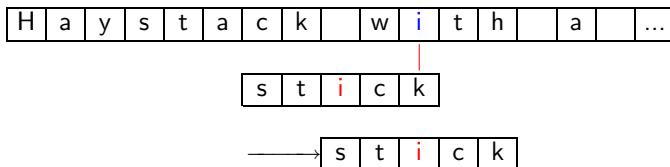
The Boyer–Moore–Horspool Algorithm

We will see in this section how to considerably speed up our naive solution to the string matching problem with some small changes only. For this purpose, we will make use of the idea presented in the following example:



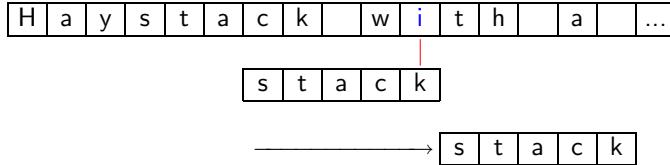
Comparing w with t symbol by symbol we observe that an **a** of the text does not match the **i** of the word. However, since symbol **a** is not present in w at all, there cannot be an occurrence of w even one or two positions to the right since in both cases symbol **a** of the text would be compared to a symbol different from **a** of the word. As a consequence, we can shift w by three positions to the right (set $pos := pos + 3$ within our program) without missing any occurrence of w . This proves that our naive algorithm performs needless comparisons.

Let's continue our example (from now on we will align the offset of w according to the rightmost symbol of t we have already seen):



If the comparison of w and t at the actual position is completed (in the situation just depicted by the mismatch of **i** and **k**), we can shift w to the right as long as we do not cover symbol **i** of the text by a symbol **i** found in w . In our example, this is symbol **i** depicted in red. Any shift of shorter distance would try to align **i** of the text to a symbol distinct from **i** and thus would yield a preassigned mismatch. Only the new position of w shown in the example deserves consideration as all shorter shifts definitely would cause a

mismatch. If the symbol of t used for realigning w (in the subsequent example symbol **i**) does not show up within w at all, we can shift w by m (i.e., 5 in our example) positions to the right without losing any occurrence:



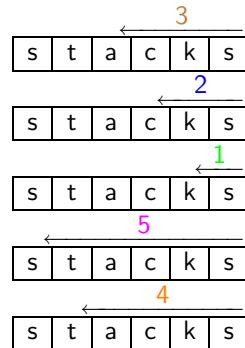
Now, notice that we can determine the number of positions we may shift w to the right independently of the actual position and from knowledge of w only. For a given w we just have to determine once at which minimal distance (number of symbols) to the right end of w each possible symbol occurs. This information is stored within a table (array) D with an entry for each possible symbol. We will use $D[a]$ to denote the entry for symbol **a** in the sequel. Accordingly, for a word w with the second to last symbol **k** we would set $D[k] = 1$ (symbol **k** is at minimal distance 1 to the right end of w). If a symbol **v** does not show up in the word at hand, we set $D[v] = m$. The following example clarifies this procedure. In order to make the presentation more convenient we decided to omit all columns of D corresponding to symbols not in use (whose entries, as already mentioned, are given by m , the length of w):

$w = \text{stacks}$

Table $D =$

a	c	k	s	t
3	2	1	5	4

Explanation:



The last **s** (the one at the right end of w) is not considered since it would imply an entry of 0, corresponding to a *shift* of word w by no position. This is due to our decision to always use the rightmost symbol of t already seen, i.e., the one currently aligned to $w[m]$, for deciding the next position to be considered.

$w = \text{needle}$ Table $D =$

e	d	l	n
3	2	1	5

Explanation:

Within word $w = \text{needle}$ the rightmost e shows up 3 positions from the right end of w . Again, we have to ignore the last symbol (an e) since it would imply a shift by zero positions. The rightmost d can be found at distance 2, the rightmost l at distance 1 and the rightmost n at distance 5 from the right end of w .

 $w = \text{with}$ Table $D =$

h	i	t	w
4	2	1	3

Explanation:

Within $w = \text{with}$ the rightmost i is located at distance 2, the rightmost t at distance 1, and the rightmost w at distance 3 from the right end of w . Since symbol h only shows up at the rightmost position of w , its entry within D is the same as in cases where h is not present at all, i.e., equal to the length of w , 4.

In general, we can describe the entries of D by the following formula:

$$D[x] = \begin{cases} m & \text{if } x \text{ is none of the first } m-1 \text{ symbols of } w, \\ m-i & \text{if } i \text{ is the rightmost position } \neq m \text{ with } w[i] = x. \end{cases}$$

According to our examples above, the first case implies that w can be shifted to the right by its entire length since the symbol at hand does not occur within w or occurs only at the rightmost position. In order to compute D using a program, we just have to execute two loops one after the other:

Computing Table D

```

1   for all symbols x do
2       D[x] := m;      // D[x] = m for any symbol x not occurring within w
3   for i := 1 to m - 1 do
4       D[w[i]] := m - i;
           // overwrite initialization for symbol observed within w

```

Note that for symbols occurring several times within w , the loop in lines 3 and 4 assigns different values to the corresponding entry of D , leaving the largest value, i.e., the rightmost occurrence, for last.

Now everything is prepared to change our naive algorithm into the so-called Boyer–Moore–Horspool algorithm. This algorithm for the string matching problem was invented by R. Horspool in 1980 as a simplification of an algorithm due to Boyer and Moore (see also the section Further Reading). All we have to do is

1. compute D once before we start the search, and
2. replace line 8 (i.e., $pos := pos + 1$) with $pos := pos + D[t[pos + m - 1]]$.

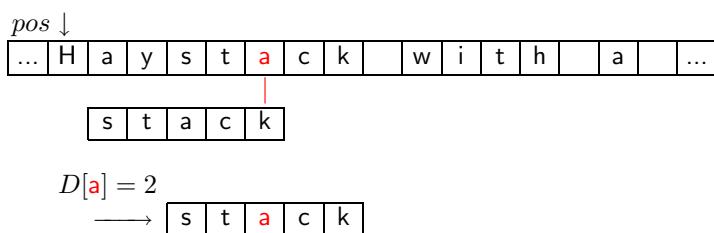
This way we obtain (the computation of D omitted):

The Boyer–Moore–Horspool Algorithm

```

1  procedure BMH
2   $pos := 1;$ 
3  while  $pos \leq n - m + 1$  do begin      // search all positions
4     $j := m;$ 
5    while ( $j > 0$ ) and ( $w[j] = t[pos + j - 1]$ ) do
6       $j := j - 1;$ 
7      if ( $j = 0$ ) then print(“Occurrence at position”,  $pos$ );
8       $pos := pos + D[t[pos + m - 1]];$ 
9  wend;
10 end.
```

If now during execution of the algorithm we have to shift w to the right (have to increase pos), then we ensure that at the new location a symbol of w matches $t[pos + m - 1]$ (according to the value of pos before increment) without missing any occurrence of w . If it is impossible to achieve this match, w entirely skips symbol $t[pos + m - 1]$:



By construction, we know that the two **a** match without comparison.

But what is our gain by this modification? First, we have to admit that for the worst case nothing has been achieved and the new algorithm performs as poorly as the naive one. There are inputs for which $D[x] = 1$ holds for all symbols x occurring within the text (in such a case, the text consists of a repetition of the second to last symbol of w). As a consequence, procedure BMH would like the naive algorithm search for w at any possible position of

the text. An example for such an input is given by text $t =$

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	a

together with $w = \text{baaaa}$. In this case every position gives rise to five comparisons since the four symbols **a** of w are always compared with symbols of the text before the fifth comparison yields a mismatch. Furthermore, this word w implies $D[\text{a}] = 1$; thus, a total number of $18 \times 5 = 90$ comparisons follows.

From a practical point of view it is rather unlikely to encounter such a text combined with such a search word. As a matter of fact our new algorithm will be much faster than the naive one for almost all inputs of practical importance. For comparison, let us return to our initial example given by the text

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
H	a	y	s	t	a	c	k		w	i	t	h		a		n	e	e	d	l	e

and the search word $w = \text{needle}$, for which we already know that D is given by:

e	d	l	n
3	2	1	5

It can easily be checked that the naive algorithm would perform 24 comparisons of text and word in order to find the single occurrence. Our improved algorithm is much faster yet. In order to determine D it must consider four different symbols. Afterwards, for the search of w only 11 comparisons are needed, of which six are necessary just to establish the single occurrence of needle. As a consequence, since our text has length 22, this shows that it is possible to search an entire text without regarding all its symbols (only half of them in our example). At first, this sounds sort of crazy. The key idea for this (in most cases) rather efficient strategy was to compare the text and the search word symbol by symbol from right to left. Only because we first regard a symbol of the text which later has to be aligned to the search word again (i.e., only because we consider $t[pos + m - 1]$ first), it becomes possible to skip certain positions without additional comparisons, since we can infer potential mismatches in advance. Thus, comparing from right to left is fundamental to our improvements – a small change of high impact.

Further Reading

1. Chapter 1 (Binary Search)

In this chapter fast search for data is discussed. Each item is assumed to be uniquely identified by a so-called *key* the same way as a number plate uniquely identifies a car. By sorting the data according to its keys it becomes possible to find items efficiently.

2. Chapter 20 (Hashing)

This chapter discusses a further idea about how to maintain a set of data in an efficient way. Again, items are assumed to be identified by a unique key; the keys then are used to compute an address (position) within a memory at which an item is stored.

3. Wikipedia article on the *Boyer–Moore string search algorithm*

http://en.wikipedia.org/wiki/Boyer_moore

This Wikipedia article deals with the Boyer–Moore Algorithm for the string matching problem. This algorithm is a variant of the one discussed in this chapter, making use of different heuristics to compute the step width applied when shifting the search word w along the text t .

4. http://en.wikipedia.org/wiki/String_matching

This article deals with the string matching problem in general and contains several links to different ideas on how to find all occurrences of a word within a text by means of an algorithm.

Depth-First Search (Ariadne & Co.)

Michael Dom, Falk Hüffner, and Rolf Niedermeier

Friedrich-Schiller-Universität Jena, Jena, Germany
Humboldt-Universität zu Berlin, Berlin, Germany
Technische Universität Berlin, Berlin, Germany

“Now this happens to those who become hasty in a maze: their very haste gets them more and more entangled.”

Lucius Annaeus Seneca (4 BC – 65 AD)

Ariadne, who according to Greek mythology was the daughter of Minos, the king of Crete, fell in love with Theseus. This Athenian hero had been entrusted with killing the Minotaur, a monster half man and half bull. The challenge was made vastly more difficult by the fact that the Minotaur was hidden in the Labyrinth. The clever Ariadne provided her hero with a ball of thread: by fixing the end of the thread at the entry of the Labyrinth and unrolling the thread while traversing the Labyrinth, Theseus could, on the one hand, avoid searching parts of the Labyrinth repeatedly, and, on the other hand, be sure to find his way back into Ariadne’s arms.



Not just the ancient Greeks had to deal with the efficient search of spaces such as labyrinths; this task also plays a central role in computer science. One method for this is *depth-first search*, which we examine more closely in the following.

Algorithmic Idea and Implementation

As already mentioned, the problem is to completely search a labyrinth. Here, a labyrinth is a system of corridors, dead ends, and junctions, and the task

is thus to visit every junction and every dead end at least once. Further, we would like to pass each corridor no more than once in each direction – after all, Theseus needs to have enough strength in the end for both the Minotaur and Ariadne.

Probably the simplest idea to solve this problem is to just walk into the labyrinth from the starting point and to tick off each junction as it is encountered. If you wind up in a dead end or a junction you have seen before, you turn around, go back to the last junction, and try again from there in another, still unexplored direction. If there is no unexplored direction, then go back to another junction and so on.

Does this method actually lead to the goal? Let us look at the search in more detail; to simplify the description, we use a piece of chalk instead of a thread. With the chalk we mark at each junction the outgoing corridors, with one tick for corridors previously traversed, and with two ticks for corridors traversed twice (that is, in two directions). Specifically, the rules for our search in the labyrinth are as follows.

- If you are in a dead end, turn around and go back to the last junction.
- If you reach a junction, tick the wall of the corridor you came from to be able to find the way back later. After this, there are several possibilities:
 1. First, you check whether you moved in a circle: If the corridor you came from just got its first tick, and there are also ticks visible on other corridors of the junction, then this is the case. You then make a second tick on the corridor you came from and turn around.
 2. Otherwise, you check whether the junction has unexplored corridors: If there are corridors without ticks, then choose an arbitrary one (say the first to the left), mark it with a tick and leave the junction through this corridor. (Incidentally, this is the case at the start of the search.)
 3. Otherwise, there is at most one corridor with only one tick, and all other corridors have two ticks. Thus, you have already explored all corridors leaving the current junction, and leave through the corridor with only one tick, giving it a second tick as a matter of form. If there is no such corridor, that is, all corridors already have two ticks, then you are back at the start and have completely searched the labyrinth.

Let us now look at the example shown in Fig. 7.1, where a path from the start A to the goal F is sought. (That is, again we must traverse the entire labyrinth, but the search can be cut short when F is found.) We assume that a dead end can be recognized as such only upon reaching it.

You start from A northwards. The first junction is C. There you leave a tick at the southbound exit (1). Of course there is no other tick here, so you choose the first unmarked corridor to the left, which is the one toward the west, and tick it (2). Then you reach a dead end at B and turn around. Back at C, the westbound corridor has now two ticks, the southbound one, but the northbound is not marked at all. Thus, you choose this way. At E, there is again an unexplored junction, and from the three possible corridors you

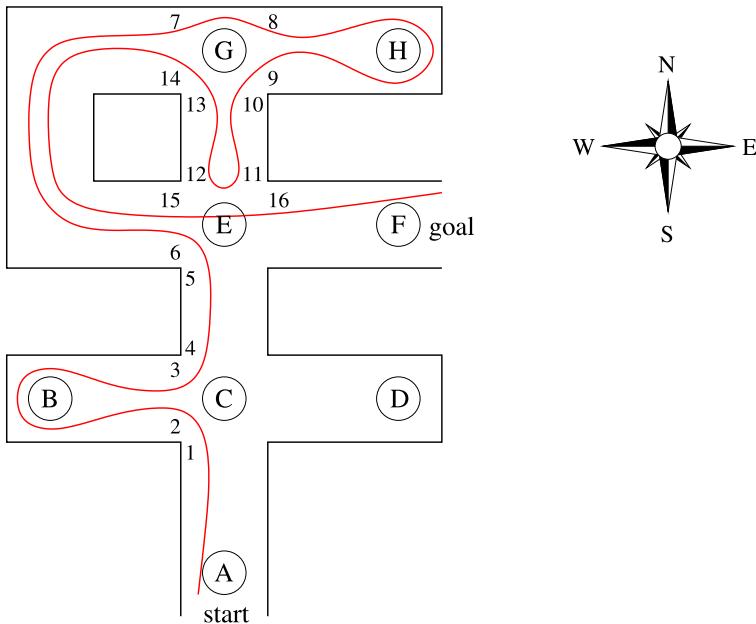


Fig. 7.1. Example for depth-first search in a labyrinth. Starting from A, a path to F is sought. Numbers mark the places where chalk ticks are left

choose the one towards the west. After two turns, you cross straight ahead over the junction at G, leaving two ticks behind (7 and 8). In H, you reach a dead end, so you again turn around. In G, there is only one option left: southwards to E. Here the rule against running in circles comes into effect for the first time: On entering E, you made a tick at the northbound exit of E (11); further, there is a tick at the southbound exit (5) and one at the westbound exit (6) – thus, you need to turn back. Over the junction G and two curves you go back, so that the northern part is now completely searched and you are back at E. Towards the east, there is no tick, and you go there. Finally, you reach the goal at F.

The principle we have learned here is called *depth-first search*, since as described we always go as deep as possible into the labyrinth and only turn around when it is not possible to proceed or a known place is encountered. Only in these cases do we go back a bit and try again from an earlier point into another direction.

The rules for depth-first search are so simple that they can be taught to a computer with only a few lines of code. For each junction, a “state” is stored, and initially the states of all junctions are set to “undiscovered.” When the DEPTHFIRSTSEARCH function is called at a junction X, it is first tested whether we moved in a circle (line 2 in the program fragment DEPTHFIRSTSEARCH I shown in Fig. 7.2). Next, it is checked whether the goal was

DEPTHFIRSTSEARCH I

```

1  function DEPTHFIRSTSEARCH( $X$ ):
2      if  $state[X] = \text{"discovered"}$  then return; endif
3      if  $X = goal$  then exit "Goal found!"; endif
4       $state[X] := \text{"discovered"};$ 
5      for each neighboring junction  $Y$  of  $X$ 
6          DEPTHFIRSTSEARCH( $Y$ );
7      end for
8  end function // End of DEPTHFIRSTSEARCH function
9  DEPTHFIRSTSEARCH(start_junction); // Main program

```

Fig. 7.2. Program code for depth-first search using recursion

reached (line 3) – if so, the program quits with the “exit” command, and the search is finished. Otherwise it goes on, and the junction X is marked as “discovered” (line 4). Now, all neighboring junctions that have not been explored yet need to be visited. To do this, the DEPTHFIRSTSEARCH function calls itself for each neighboring junction Y (lines 5–7). This is a frequent trick in programming called *recursion*, which was already described in Chap. 1. When the newly called DEPTHFIRSTSEARCH function notices that Y was already visited and we thus moved in a circle, it immediately returns (line 2) to the calling function at the junction X . Otherwise, the search continues at junction Y .

Sometimes, one wants to avoid recursion, one reason being that at each recursive call in the computer implementation additional time is spent to allocate variables etc. In this case, the depth-first search can be coded without recursion using a *stack*. A stack is a data structure that allows placing objects (in our case junctions) on top of the stack or to remove the object currently on top of the stack. For us, the stack serves to store the return path; we always put a junction X on top of the stack when leaving it, together with a number “exits” that indicates how many of the corridors leaving X have already been explored (see Fig. 7.3). For each junction, we have an array listing all neighbor junctions – thus, we can easily retrieve, e.g., the fifth neighbor junction of a junction X when needed. The variable “mode” contains the information whether we are following an unexplored corridor or whether we are coming back from an already explored junction, going through a corridor that we have already passed in the other direction.

Applications

The depth-first search method works not only for labyrinths, but it also has applications in completely different contexts, as we see in this section.

DEPTHFIRSTSEARCH II

```

1   X := start-junction;  mode := "forwards";
2   repeat
3       if mode = "forwards" then
4           // we came here through a new corridor
5           if state[X] = "discovered" then
6               mode := "backwards";
7               take the top pair (X, exits) off the stack;
8           else    // junction is unexplored so far
9               if X = goal then exit "Goal found!"; endif
10              state[X] := "discovered";
11              if X has no exits then exit "Goal not found!"; endif
12              put the pair (X, 1) on top of the stack;
13              X := the first neighboring junction of X;
14          endif
15      else    // we are coming back
16          if exits < number of neighboring junctions of X then
17              exits := exits + 1;
18              put the pair (X, exits) on top of the stack;
19              mode := "forwards";
20              X := neighboring junction number exits of X;
21          else    // there are no unexplored corridors here any more
22              if the stack is empty then
23                  exit "Goal not found!";
24              else    // we go back further
25                  take the top pair (X, exits) off the stack;
26              endif
27          endif
28      end repeat

```

Fig. 7.3. Program code for depth-first search without recursion

Example: Web Search

Here, we do not consider Theseus who is wandering about in the labyrinth, but instead observe a student called Sinon who is searching for a specific Web page.

Sinon Davis has recently been to a party given by his classmate Ariadne, and there he struck up a conversation with a cute girl. Now he would like to see her again, but unfortunately he has not asked for her name. What to do? Of course, Sinon could ask Ariadne, but first he is too shy, and second Ariadne herself does not know all the guests from her party. Finally, Sinon has a bright idea: Why not look up the girl on the Internet site “fazebook.org”? In this commonly known social network platform, almost every young person in the country has a profile, typically with a photo and with links to profiles of

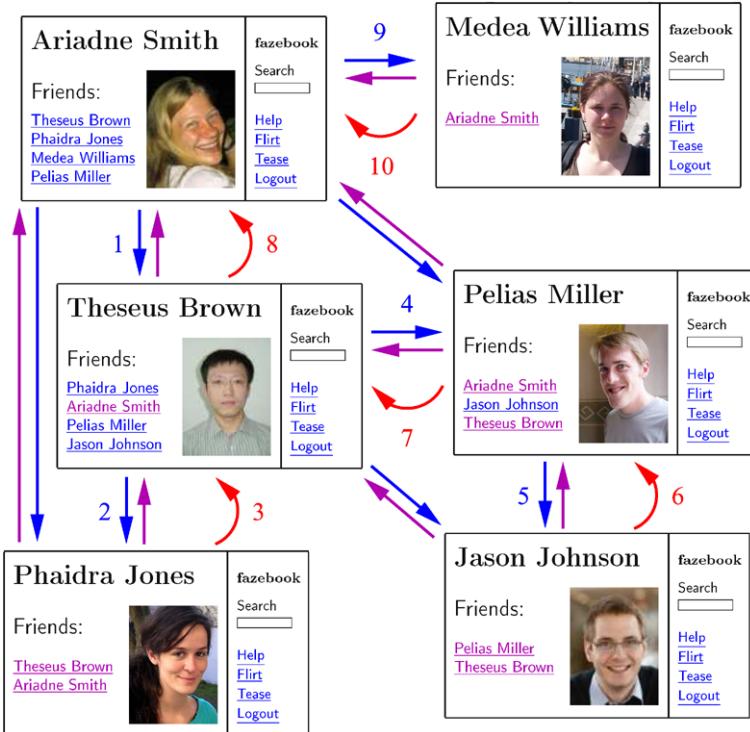


Fig. 7.4. Depth-first search in fazebook: The numbers show the order of the hops from profile to profile. A straight arrow means that a profile contains a link to another profile. A curved arrow means that the “Back” button is used at this place. Note that not all shown links are used since some of them are displayed in violet at the moment when the profile is visited

friends. Hence, Sinon could just visit Ariadne’s profile in fazebook and, starting from there, go through the profiles of all her friends, friends of friends, friends of friends of friends, and so on – until he has either found his adored (hopefully, she has a photo in her profile) or completely scanned the relevant parts of Ariadne’s surrounding. Sinon, therefore, is confronted with the following task: scan all profiles in fazebook that are reachable from Ariadne’s profile via links whose owners have been at the party. As in the previous section, the difficulty is again, on the one hand, not endlessly to move in a circle and, on the other hand, to check everything completely and systematically. This can be done efficiently with a depth-first search in the network of the profiles in fazebook.

Thus, assume that Sinon is beginning his search and is starting at Ariadne’s profile. Hence, he clicks on the first link in the list of Ariadne’s friends and arrives at the profile of Theseus (Fig. 7.4). Since Theseus is not the person Sinon is searching for, Sinon continues with clicking on the first link of his

profile. In this way, he follows the links from profile to profile, always taking care not to follow links that lead to profiles that he has already visited. Sinon, a skilled Web surfer, can recognize these links because his browser displays them in violet instead of blue (this helpful function of the browser in a sense corresponds to the chalk markings in our first example). When Sinon finally reaches a profile whose owner has not been at the party, or when all friends of the owner have already been processed (and displayed in violet), then Sinon clicks on the Back button of his browser and continues with the friends on the resulting profile. Like DEPTHFIRSTSEARCH II, the Back function of the browser uses a stack; whenever a link is clicked, the address of the page containing the link is put on the stack. Furthermore, whenever the Back button is used, the browser jumps to the address on the top of the stack and removes the address from the stack.

If Sinon's adored one has a photo on her profile, and if her profile can be reached from Ariadne's profile by following a series of links whose owners were all at the party (which was our assumption), then Sinon will definitely find her with this method! If, however, Sinon finally ends up on Ariadne's profile by clicking on the Back button, and all links on Ariadne's profile have been visited (and, hence, displayed in violet), then this means that Sinon has bad luck and will not find her – but at least he can be sure that he has not missed any of the profiles coming into question.

Example: Labyrinth Creation

Depth-first search is useful not only for Theseus, but also for the Minotaur: it can be used to create very confusing labyrinths. The method is very simple: Take a regular grid that consists of squares that are separated by “borders”, and start the depth-first search at an arbitrary square of the grid. Then, the depth-first search recursively calls itself for all neighboring squares *in random order* (for example, the random number algorithm from Chap. 25 can be used here). Whenever a square is visited for the first time, the border to the preceding square is destroyed (that is, the border to the square from which the depth-first search reached the new square). The result is a pattern like the one shown in Fig. 7.5. Since the depth-first search visits each square, it creates a path from the starting point to each square and, therefore, from each square to each other square. However, these paths are not easy to discover.

Example: Television Shows

Let us assume that the television show “Sick Sister” is going to produce two new seasons. In this show, the candidates are put into the “Sick Sister House,” where they are observed by cameras the whole day (and night). In order to provide interesting entertainment, the candidates should be at loggerheads as often as possible, which means that in each season there should be no two candidates in the house who like each other. Now assume that the candidates

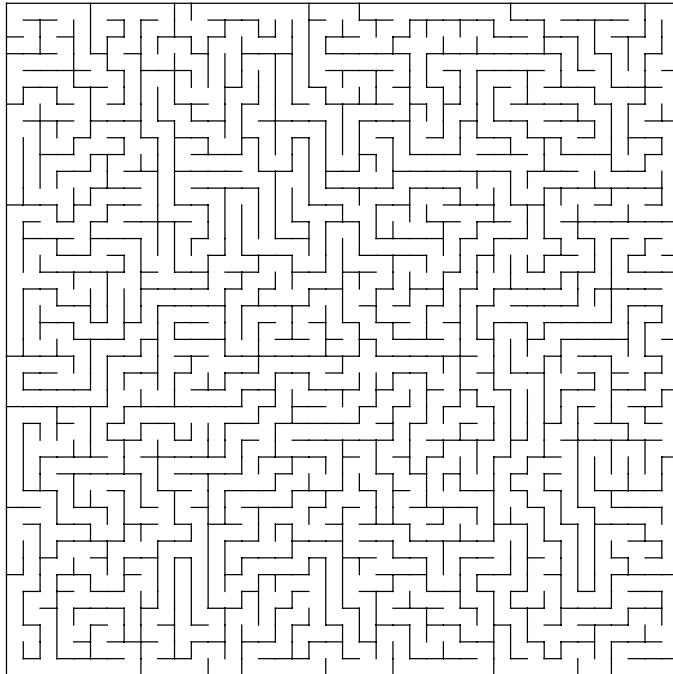


Fig. 7.5. Example for a labyrinth created by a depth-first search

for the upcoming two seasons have already been selected, but for each candidate it still has to be decided – of course, obeying the rule “no two candidates in the house that like each other” – whether he (or she) is part of the first or the second season.

To solve this task, one can create a “sympathy graph”: on a piece of paper, draw a small circle – called *vertex* – for each candidate, and connect two circles with a line – called *edge* – if the corresponding candidates like each other (see Fig. 7.6 for an example).¹ Now, the task is to color the vertices of the graph in such a way that not more than two colors are used, each vertex gets exactly one color, and no two vertices that are connected by an edge get the same color. Once the graph is colored, the color of each vertex tells in which season the corresponding candidate has to take part. The desired “two-coloring” for the graph can be found using a depth-first search: Choose an arbitrary vertex and arbitrarily assign one of the two allowed colors to it. Then start the depth-first search at this vertex. Whenever the depth-first

¹ This type of graph, consisting of vertices and edges, has nothing to do with graphs of functions as they are known in the field of mathematics called analysis. “Vertices-and-edges graphs” can be used to model a variety of circumstances and objects, for example, labyrinths: each dead end and each crossing of the labyrinth can be modeled as a vertex and each path of the labyrinth as an edge.

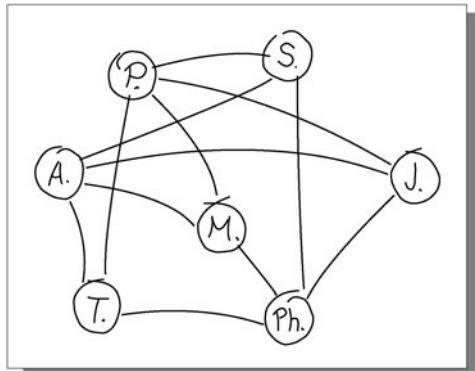


Fig. 7.6. A “sympathy graph”: If two persons like each other, then the corresponding vertices (marked with the initials of the persons) are connected by an edge

search comes from a vertex X to a vertex Y that has not been visited before, Y gets the color that X does not have. If the depth-first search comes from a vertex X to a vertex Y that has already been visited, check whether X and Y have different colors. If this is not the case (that is, two vertices of the same color have been found that are connected by an edge), then it is not possible to color the graph as desired. Otherwise, the depth-first search will yield the coloring and provide for exciting entertainment with “Sick Sister.”²

We mention in passing that graphs that can be colored with two colors as described are called *bipartite*; they are exactly those graphs that do not contain a cycle of odd length. If the candidates shall be distributed among three seasons instead of two, then the task is much more difficult and nobody knows whether it can be solved efficiently with a depth-first search. (The problem is that whenever one reaches a vertex that has not been visited before, one has the choice between two colors and does not know which of them to take.)

Example: Traffic Planning

A further application of depth-first search reads as follows. For the sake of traffic calming in his town, councilman Hermes wants to declare a number of streets to be one-way streets. Thereby, Hermes has to beware of incurring the wrath of the car drivers: He may only prune the street network in such a way

² In the special case where the sympathy graph consists of several parts – called “connected components” – that are not connected to each other, then the depth-first search has to be executed separately for each connected component. This can even provide a way to distribute the candidates among the seasons as equally as possible (concerning the number of candidates): just exchange the two colors within some of the connected components.

that no isolated “islands” are generated that cannot be entered or left – that is, it must still be possible to drive from each place to each other place. In graph theory, one would say that the street network must remain one single “strongly connected component”. Again, this problem can be solved efficiently by using depth-first search; Chap. 9 takes a closer look at this issue.

Breadth-First Search

One problem of depth-first search is that one can quickly move far away from the start. In many cases, however, it is known that the goal is not too far. For example in fazebook, it can be assumed that the wanted profile is at a distance of maybe at most three from the host. In this case, a *breadth-first search* is more appropriate: It searches the graph layer-wise from the start point, that is, first all direct neighbors (distance 1), then all vertices at distance 2, etc. For this, a data structure called “queue” is used (instead of a stack as is used for depth-first search). A queue allows us to append vertices at the end and to remove the vertex that is currently at the front; in case of breadth-first search, whenever a vertex is removed from the queue, one jumps to this vertex. Therefore, breadth-first search is not applicable for searching a labyrinth: One cannot simply note a junction on a list and “jump” to it on demand. For many other applications (such as the Web search), though, this is not a problem. The program fragment shown in Fig. 7.7 shows breadth-first search in detail.

The queue always holds the vertices that still have to be visited. Thus, initially we add the start vertex to the queue (line 2). As long as the queue has not become empty, the following is repeated: The first vertex is taken out (lines 3 and 4), and all neighbors of this vertex are added to the queue

```

BREADTHFIRSTSEARCH
1  begin // initially, the queue is empty
2      append the start vertex at the end of the queue;
3  while queue is not empty
4      take the first vertex X from the queue;
5      if state[X] ≠ “discovered” then
6          if X = goal then exit “Goal found!”; endif
7          state[X] := “discovered”;
8          for each neighboring vertex Y of X
9              append Y at the end of the queue;
10         end for
11     endif
12   end while
13 end

```

Fig. 7.7. Program code for breadth-first search

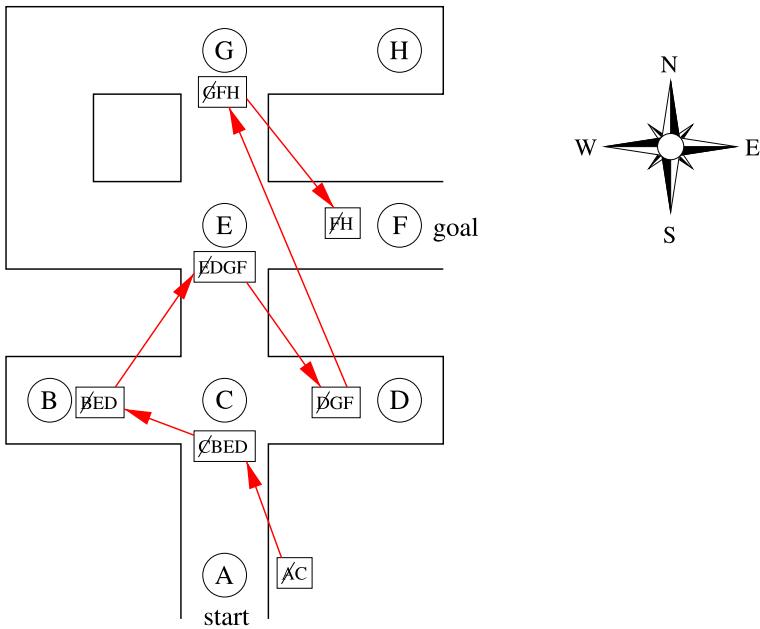


Fig. 7.8. Example for breadth-first search in the labyrinth

(lines 8 and 9). To avoid searching the neighbors of a vertex more than once, whenever a vertex has been treated in this way, it is marked as “discovered” (line 7), and discovered vertices are skipped (line 5).

As an example, we will look at the same graph that has already been used for depth-first search: the labyrinth (Fig. 7.8). Initially, the queue holds only the vertex A. This vertex is taken out, and all neighbors of A are appended; this is only C. The resulting queue is depicted next to A. The search continues with taking out the first element of the queue, which is C. Strictly speaking, all four neighbors of C would now be added to the queue, but as a small optimization we ignore A, since it is already discovered and would not have any effect when taken out of the queue. Thus, B, E, and D are added. At B, no new vertices are added, and we continue directly at E. There, G and F are appended, while C is being ignored. At D, again nothing happens, and we continue at G. Here H is appended, but at F we already find the goal.

It is easy to see that the order in which vertices are visited is very different than with depth-first search (Fig. 7.1). In breadth-first, vertices are visited in order of their distance from the start vertex: first vertex C (distance 1), then B, E, and D (distance 2), finally G and F (distance 3). This also implies that breadth-first always finds a shortest possible path to the goal, and does not consider longer paths in-between.

Incidentally, if a stack is used here instead of a queue, then the algorithm executes a depth-first search instead of a breadth-first search. In contrast to

algorithm DEPTHFIRSTSEARCH II, however, the stack is not used to store the return path, but to memorize the junctions to which corridors have been seen but that have not been visited immediately. Therefore, the stack can be significantly larger than with DEPTHFIRSTSEARCH II.

What to choose now with a concrete problem, depth-first search or breadth-first search? Depth-first search is usually slightly easier to implement, since by using recursion, it is not necessary to explicitly maintain a data structure like the queue of breadth-first search. Further, breadth-first search usually uses more memory; for difficult problems, it can even happen that available memory is not sufficient. On the other hand, breadth-first search always finds a shortest path (with respect to the number of edges) and is fast, in particular when the graph is very large but the goal is close to the start. In this case, depth-first search can easily get lost in more distant regions. Thus, which algorithm is the better one depends on the situation at hand.

Further Reading

1. Presentations on depth-first search can be found in most algorithm textbooks.
2. Chapter 9 (Cycles in Graphs)
In this chapter, another application for depth- and breadth-first search is shown.
3. Chapter 8 (Pledge's Algorithm)
In our labyrinth example, we assumed that standing at a junction, we can see all exits. But what happens when our torch extinguishes and we are in the dark? Even then it is possible to find the goal; how to do that is explained in Chap. 8.
4. Chapter 32 (Shortest Paths)
Breadth-first search finds a shortest path if the measure is the number of vertices passed. Often, though, the distances between vertices are different, and we want a path where the sum of the length of the edges is as small as possible. This problem is treated in Chap. 32.

Acknowledgement

We thank Martin Dietzfelbinger (Ilmenau) for many constructive suggestions.

“Everything on earth can be found, if only you do not let yourself be put off searching.”

Philemon of Syracuse (ca. 360 BC – 264 BC)

Pledge's Algorithm

– How to Escape from a Dark Maze

Rolf Klein and Tom Kamphans

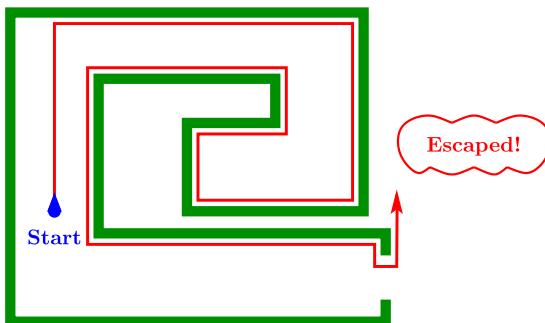
Rheinische Friedrich-Wilhelms-Universität Bonn, Bonn, Germany
Technische Universität Braunschweig, Braunschweig, Germany

“There must be some way out of here,” said the joker to the thief,
“There’s too much confusion, I can’t get no relief.”

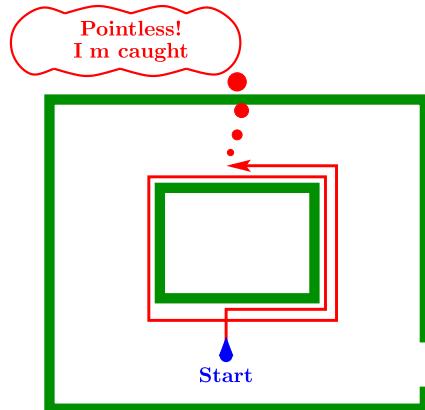
from “All Along the Watchtower” by Bob Dylan

“Oh dear, the light’s gone out! How do I get out of here? Maybe exploring the tunnels below the imperial Roman baths in Trier all alone was not such a good idea. But wait! Recently, I read something about systematically searching a maze with crossings and tunnels: depth-first search. But, unfortunately, one needs some light and chalk for markings. So this does not work in the dark. Do I really have to spend the night in here?”

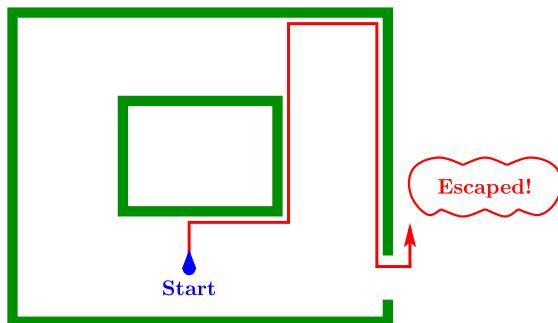
What can we do in this situation? Let’s try something. We just follow our nose, carefully moving forward until we hit a wall. Then we turn right and follow this wall, always touching the wall with our left hand, until we reach the exit. This works fine in a maze like this,



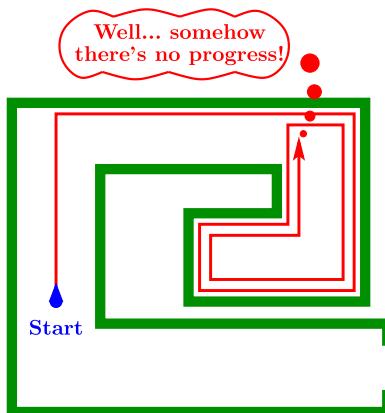
but if we hit some kind of pillar, we get into a loop!



So, our first idea does not work in general. We need to leave a pillar eventually. Next try! Follow our nose until we hit a wall; follow the wall until we can move straight ahead in the original “nose” direction. Now, the pillar is no longer a problem:



But this approach does not work in our first example.



"Now it gets scary! Whatever I try, nothing works. But there must be some way out of here – after all I did get in somehow."

Of course there is a way out of the maze. We just have to find it. So, is there an algorithm that finds a path out of *every* possible maze? Even in the dark and without any tools such as chalk or GPS?

Amazingly, such an algorithm exists!

Pledge's algorithm

```

1 Set angle counter to 0;
2 repeat
3   repeat
4     Walk straight ahead;
5   until wall hit;
6   Turn right;
7   repeat
8     Follow the obstacle's wall;
9   until angle counter = 0;
10  until exit found;
```

It is not sufficient to just watch the direction of one's nose; we need to count the turns that we make while following the walls.

For simplicity, let us assume that all corners are rectangular, as in our examples. Then there are only left and right turns of 90 degrees each. We count these turns as follows. For every left turn we add 1 to our counter, for every right turn we subtract 1 from the counter (in particular, we subtract 1 for the very first turn we make when hitting a wall).

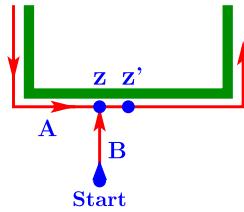
This algorithm is said to have been invented by a 12-year-old boy named John Pledge. And it works! Not only for our examples, but in every maze! Let's try to prove this.

Suppose Pledge's algorithm does not find us a way out. Then we get stuck in a loop that will be followed on and on. Why? There are but a few points where we may change our direction, the corners of obstacle walls, and from every corner the first point on an obstacle seen in the initial nose direction. If we reach one of these points twice with the same angle-counter value, the path between both visits is repeated forever, because our behavior never changes.

Otherwise, we reach every corner at most once with the same angle-counter value, in particular, with value 0. When all these visits have been made, we will never again leave the current obstacle, because whenever we can, the counter won't be 0. Thus, our path gets cyclic.

Moreover, we can show that the loop we are following forever can have no self-crossings. In a crossing, two straight segments of the path – let's call them *A* and *B* – will meet. One of them – say *A* – has to be a *free* segment; that is,

the segment does not lead along an obstacle wall, because walls of obstacles do not cross.



Let z be the crossing of A and B and let $C_A(z')$ and $C_B(z')$ be the angle-counter values in a point z' shortly behind z , having reached z via A and B , respectively. Then, we have

$$\begin{aligned} C_B(z') &= -1, \\ C_A(z') &= -1 + 4 \cdot k \quad \text{for } k \in \mathbb{Z}, \end{aligned}$$

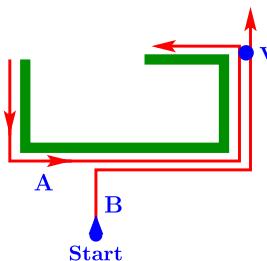
because right after z our nose points in the same direction. For $k \geq 1$, we get

$$C_A(z') = -1 + 4k > 0.$$

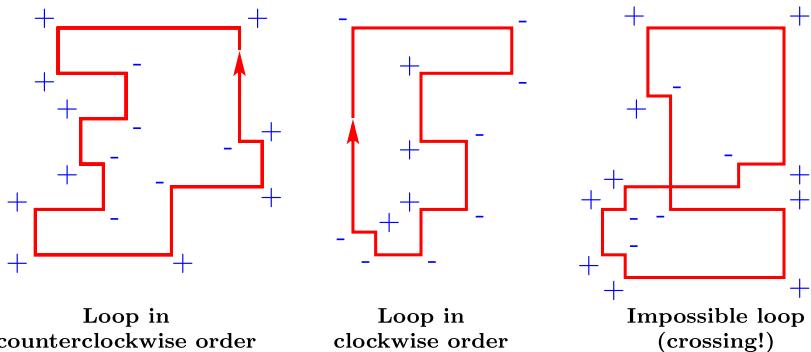
But our angle counter value can never become positive. This is because after hitting a wall the counter has value -1 . As soon as it reaches 0 , we leave the wall. After hitting the next wall, it gets negative again, and so on. Hence, we must have $k \leq 0$.

From $k = 0$ we would conclude $C_A(z') = C_B(z')$. In this case, the parts of the path along A and B behind z would never separate again. So, if we walk along segment B , after visiting z , we would never walk along A again – and vice versa. This contradicts the fact that both A and B are part of an endless loop.

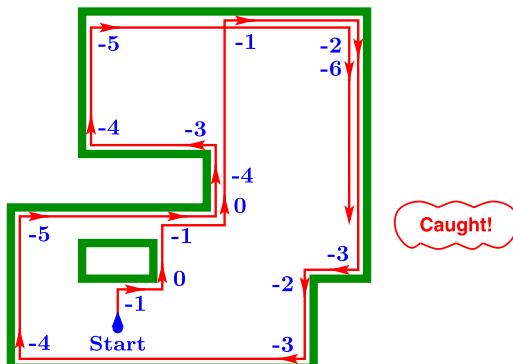
Thus, only the case $k \leq -1$ remains. Then, we have $C_A(t) < C_B(t)$ for every point t from z' to the point v where both paths split. Moreover, in v we must have $C_B(v) = 0$. So these parts of the path look as shown in the next figure. We do not have a real crossing – the parts of the path just touch each other.



So far, we have shown the following. If the Pledge algorithm doesn't get us out of a maze, we end up in an endless cycle that has no self-crossings.

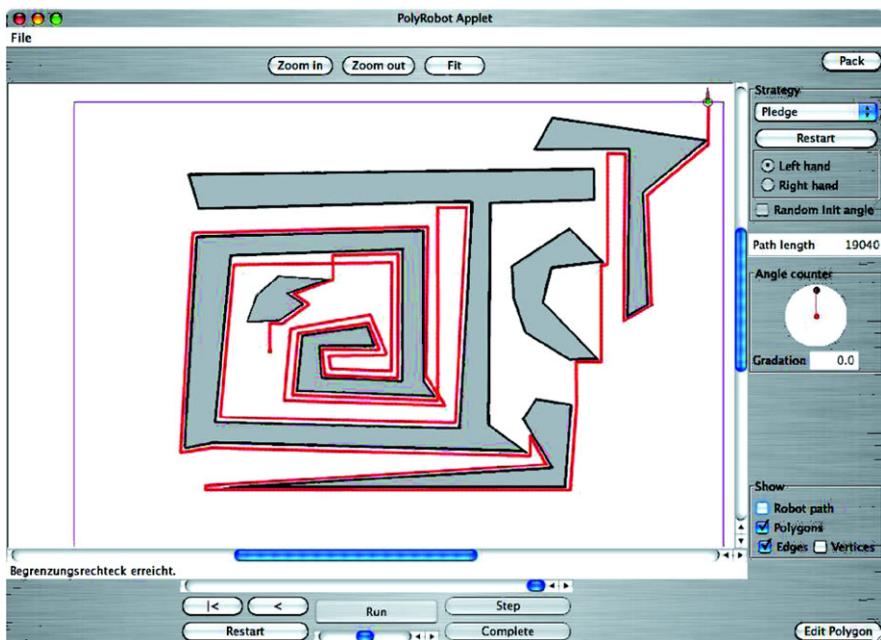


Assume that we traverse this loop in counterclockwise direction. Then, we have four more left turns than right turns, and our angle counter increases each time we complete a cycle. Eventually, its value must become positive, but we have already seen that this cannot happen. Thus, we traverse the loop in clockwise direction. In every pass, our angle counter decreases by 4. That means, we keep following a wall with our left hand without ever leaving it. But then we are inside a closed room – and there *is* no way out!



You can use the Java applet <http://www.geometrylab.de/Pledge/> to draw a maze and observe how Pledge's algorithms deals with it. By the way, it also works for mazes with non-rectangular angles. In this case, we have to exactly add the (signed) turning angles instead of just counting the number of turns.

Here is an example.



Further Reading

1. Chapter 7 (Depth-First Search)

In this chapter, you learn how to use additional tools such as threads or chalk.

2. Chapter 9 (Cycles in Graphs)

helps to prevent you running around in a circle.

The “nodes” mentioned in this chapter are rooms in our mazes with more than one exit. The “edges” are tunnels and doors that connect two rooms.

3. <http://www.geometrylab.de/Pledge/>

Here, you can try out Pledge’s algorithm with a Java applet. Moreover, you’ll find a short movie showing a small robot (a Khepera II) which uses Pledge’s algorithm to solve a maze.

4. Rolf Klein: *Algorithmische Geometrie: Grundlagen, Methoden, Anwendungen*. Springer, Heidelberg, 2nd edition, 2005 (in German).

Harold Abelson, Andrea A. diSessa: *Turtle Geometry*. MIT Press, Cambridge, 1980.

In these books you’ll find the proof that Pledge’s algorithm also solves non-rectangular mazes. Moreover, similar problems are presented; for example, how a robot can find a target point.

5. Bernd Brüggemann, Tom Kamphans, Elmar Langetepe: *Leaving an unknown maze with one-way roads*. In: *Abstracts 23rd European Workshop Comput. Geom.*, 2007, pp. 90–93.

<http://web.informatik.uni-bonn.de/I/publications/bkl-lumow-07.pdf>

Unfortunately, Pledge's algorithm does not work if there are passages that can be traversed in only one direction (e.g., one-way roads). This paper shows how you can escape anyway.

Acknowledgement

The authors thank Martin Dietzfelbinger for many valuable comments.

Cycles in Graphs

Holger Schlingloff

Humboldt-Universität zu Berlin, Berlin, Germany

This chapter is about cycles in graphs. We want to find a way to tell whether there is a cycle in a set of nodes which are connected by edges. A *cycle* is a path which leads from one node back to itself.

Scenario 1

Imagine you were on an airplane that crashed in the middle of the jungle and now you're trying to find a way back into civilization. There are some paths through the jungle that were created by natives; other than those, there is only coppice around you. The vegetation is so dense that you can't even see the sky, let alone the sun. You pack your belongings, choose one of the paths and start heading in that direction. After a little while, there is a fork in the path.

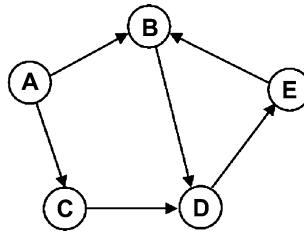
You decide to turn right. Then you encounter a junction where you keep walking straight ahead. Unfortunately, this leads you to a dead end: you have to turn around and go back to the intersection, where you now turn. At the next fork, you take the left path, then the right, and so on. Suddenly, there is a clearing in the jungle and you see your airplane, your starting point. Apparently, you've been walking in circles all the time. How can you avoid getting lost again, what would be a better way to try to return to civilization?



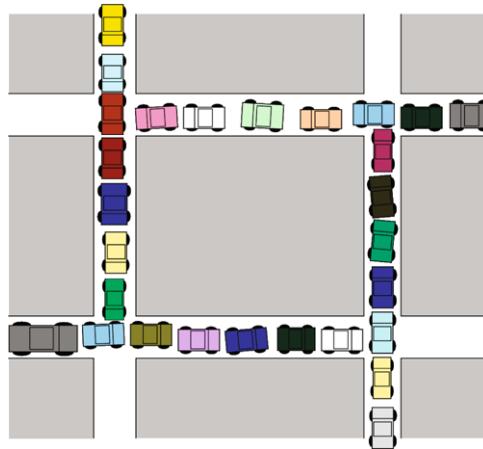
Scenario 2

Andy wants to go to the movies with Benny and Charly. Charly, however, has to babysit at home and can only leave when Dany comes and takes over. Benny may only go when he is finished with his homework. He also needs help from Dany, who has promised to come over as soon as Eddy returns her book, which he borrowed from her in school. Eddy himself is still racking his brains about the homework questions and hopes that Benny will send him an e-mail with the solutions. Why will Andy probably not get to see the movie with Benny and Charly tonight?

Both scenarios demonstrate the same problem: cycles in graphs. A *directed graph* is a structure that consists of *nodes* and *edges*, where an edge leads from one node to another. In order to visualize a graph, nodes are shown as circles and edges as arrows between nodes. For this example, we can draw a node for each person and an edge from node x to node y if person x is waiting for person y . We write A for Andy, B for Benny, C for Charly, and so on. The graph for scenario 2 then looks like this:



Obviously, there is a cycle in this graph: $B \rightarrow D \rightarrow E \rightarrow B$. That means there is a sequence of nodes connected by edges where the beginning and the end of the path are the same node. Benny is waiting for Dany, Dany is waiting for Eddy, and Eddy is waiting for Benny: if they don't do anything about it, they will wait for each other for a very long time. Cycles like the one shown here can lead to problems such as endless processes (as in the first scenario, where you can walk through the jungle endlessly) or processes getting stuck (as in the second scenario, where no one will see the movie unless someone dissolves the cycle). When this happens in a computer program, it is called an endless loop (if the process runs endlessly) or a deadlock (if the process gets stuck). In both cases, usually the program fails to show a reaction and has to be ended from the outside, e.g., by the user. This is why it is important to recognize cycles and, if possible, avoid them.



A traffic deadlock

Finding Cycles by Depth-First Search

So how can we actually find cycles in a program? Let's look at the first scenario again: you're lost in the jungle, trying to find a way out. If you want to avoid running in circles, you could use the same trick Hansel and Gretel came up with in the well-known fairytale and mark your way through the dense forest with pebbles. If you then come across a pebble that you dropped before, you know right away that you've been there already and that this way is not going to get you out. Basically, this is the same situation that was described in Chap. 7, where Theseus, trying to find his way through the labyrinth, marks his path with the thread Ariadne gave him (or, alternatively, with chalk). In order to simulate our jungle expedition with a computer program, we can use depth-first search (DFS), as in Chap. 7. First, we describe the jungle map with a graph: every fork or intersection in the path is a node in the graph, every stretch of road in between is an edge. The goal of the search is to find a node which is outside the jungle. We can write the algorithm for our depth-first search down like this (quite similarly to the algorithm in Chap. 7):

Depth-first search

```

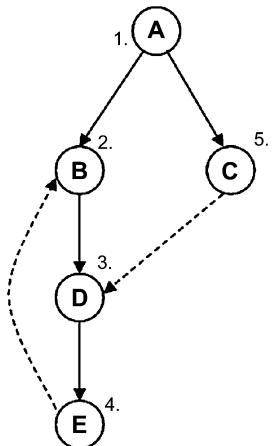
1  procedure DEPTH-FIRST-SEARCH (node  $x$ )
2  begin
3      if goal is reached then stop
4      else if  $x$  is unmarked then
5          mark  $x$ ;
6          for all nodes  $y$  succeeding  $x$  do DEPTH-FIRST-SEARCH( $y$ ) endfor
7      endif
8  end
```

Here we assume that all nodes are “unmarked” in the beginning. The depth-first search begins when the procedure $\text{DEPTH-FIRST-SEARCH}(x_1)$ is started for any node x_1 . If x_1 is followed by the nodes y_1, y_2, y_3 , etc., $\text{DEPTH-FIRST-SEARCH}(y_1)$, $\text{DEPTH-FIRST-SEARCH}(y_2)$, and so on, will be started in order. However, if for example y_2 has the successor nodes z_1, z_2 , etc., then $\text{DEPTH-FIRST-SEARCH}(z_1)$, $\text{DEPTH-FIRST-SEARCH}(z_2)$, and so on, will be completed for all z before $\text{DEPTH-FIRST-SEARCH}$ for y_3 is started. If the search leads to a node that doesn’t have any successors (a dead end), or a node that has already been marked, the search is not continued, but will return to the previous node, and so on.

For our jungle scenario this means that at an intersection you drop a pebble and then systematically test all possible paths which start from this intersection. If you reach a dead end at one path, or see a pebble lying on the ground that you know you dropped before, you return to the previous intersection and try another path from there. This makes more sense than just randomly following any path and then, if it reaches a dead end or becomes cyclic, returning all the way to the starting point to try a different path.

This picture illustrates the algorithm for our second scenario, in which Andy is trying in vain to catch a movie with his friends Benny and Charly. The nodes (which here represent the people involved) are drawn in order from top to bottom (from A to E); the numbers next to the nodes show the order in which the nodes are reached and marked by the algorithm. The search starts with node A, and therefore with the call $\text{DEPTH-FIRST-SEARCH}(A)$. Since A is not marked yet, $\text{DEPTH-FIRST-SEARCH}(B)$ and $\text{DEPTH-FIRST-SEARCH}(C)$ are started in order. $\text{DEPTH-FIRST-SEARCH}(C)$ will run after $\text{DEPTH-FIRST-SEARCH}(B)$ and $\text{DEPTH-FIRST-SEARCH}$ of all successors of B are completely finished. $\text{DEPTH-FIRST-SEARCH}(B)$ calls $\text{DEPTH-FIRST-SEARCH}(D)$, which calls $\text{DEPTH-FIRST-SEARCH}(E)$, which calls $\text{DEPTH-FIRST-SEARCH}(B)$, since B is a successor of E. Node B, however, is already marked; therefore, we go back to the previous node, E. This node does not have any other successors than B. Therefore, we return to D, then to B, and then to A. Here we see that A actually has another successor that has not been called yet, C; therefore, $\text{DEPTH-FIRST-SEARCH}(C)$ is started now. The only successor of C, node D, is already marked, so we return to C and then to A. Now all calls are finished and the algorithm is done.

When trying to find cycles in graphs, the goal is to discover whether or not a graph contains any cycles, and, if possible, have one (or some) of



them displayed. In order to achieve this, we need to adjust our algorithm a bit. As we can see in our example graph above, there are three kinds of edges:

1. forward edges, such as $A \rightarrow C$
2. sideward edges, such as $C \rightarrow D$, and
3. backward edges, such as $E \rightarrow B$.

In a directed graph, only an edge that goes backwards can cause a cycle. Backward edges differ from edges that go sideways in that they lead to nodes that have not been completely processed yet. We can include this information into the above algorithm if we extend the markings: instead of just “marked” or “unmarked,” we can have the program ‘remember’ whether the processing of a node has not started yet, is in progress, or is done. (In our jungle scenario, we could use pebbles of different color for this purpose.)

Depth-first search for cycles

```

1 procedure SEARCH-CYCLE (node  $x$ )
2 begin
3   if mark( $x$ ) = “in progress” then a cycle has been found
4   else if mark( $x$ ) = “not started yet” then
5     mark( $x$ ) := “in progress”;
6     for all nodes  $y$  succeeding  $x$  do SEARCH-CYCLE( $y$ ) endfor;
7     mark( $x$ ) := “done”
8   endif
9 end

```

For our example graph, the order of the calls would look like this:

```

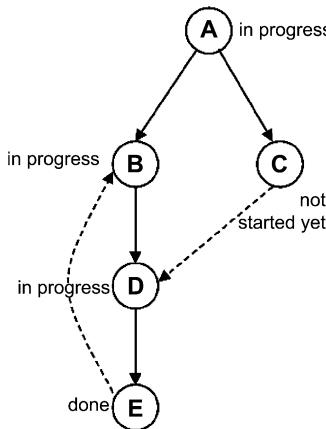
SEARCH-CYCLE(A) // A not started yet
| A in progress
| SEARCH-CYCLE(B) // B not started yet
| | B in progress
| | | SEARCH-CYCLE(D) // D not started yet
| | | | D in progress
| | | | SEARCH-CYCLE(E) // E not started yet
| | | | | E in progress
| | | | | | SEARCH-CYCLE(B) // B in progress
| | | | | | | a cycle has been found!
| | | | | | | | E done // snapshot
| | | | | | | | D done
| | | | | | | | B done

```

```

| SEARCH-CYCLE(C) // C not started yet
| | C in progress
| | SEARCH-CYCLE(D) // D done
| | C done
| A done

```



A snapshot during the execution
of procedure SEARCH-CYCLE

Strongly Connected Components

The algorithm SEARCH-CYCLE that was shown above determines whether a cycle can be reached from the starting node. However, it cannot recognize which nodes are part of the cycle. The algorithm can therefore not solve the deadlock in the second scenario: if no one knows that Benny, Dany, and Eddy are part of a cycle, they can't solve their problem.

In order to break the deadlock, we have to determine which nodes are part of it. To do so, the algorithm has to remember the order of the nodes which are “in progress”. At the snapshot shown above, the current path is $A \rightarrow B \rightarrow D \rightarrow E \rightarrow B$, which means that B, D, and E are in the cycle: when a node is encountered along the path that has already been seen (in this case, B), all nodes that follow this node (here, D and E) are part of the cycle. Whenever the algorithm is finished with a node and returns to the previous node, the finished node has to be removed from the current path (as it is obviously not part of a cycle).

As an algorithm, this idea would look as follows:

Finding cycles by depth-first search

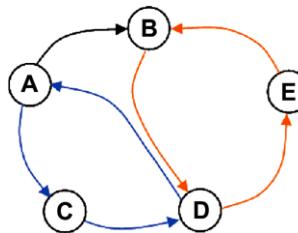
```

1  procedure FIND-CYCLE(node  $x$ )
2  begin
3      if  $\text{mark}(x) = \text{"in progress"}$  then
4          a cycle has been found;
5          all nodes on the current path starting at  $x$  are on the cycle
6      else if  $\text{mark}(x) = \text{"not started yet"}$  then
7           $\text{mark}(x) := \text{"in progress"};$ 
8          extend the current search path by appending  $x$ ;
9          for all nodes  $y$  succeeding  $x$  do FIND-CYCLE( $y$ ) endfor;
10          $\text{mark}(x) := \text{"done"};$ 
11         remove  $x$  (the last element) from the current path
12     endif
13 end

```

Here we assume that at the beginning all nodes are marked as “not started yet” and that the initial search path is empty.

What would happen in this algorithm if there were more than one cycle in the graph, for example, an additional edge from D to A (that means Dany would be waiting for Andy)?

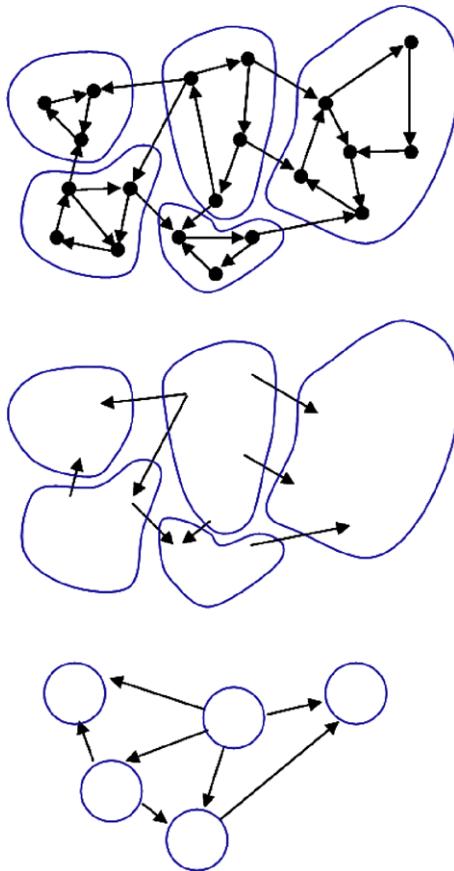


The example graph with an additional edge

In this case, firstly, we have the cycle $B \rightarrow D \rightarrow E \rightarrow B$ that already appeared in the graphs before. Additionally, we have the cycle $A \rightarrow C \rightarrow D \rightarrow A$, which has a similar structure. Furthermore, there would be other cycles, such as $E \rightarrow B \rightarrow D \rightarrow A \rightarrow B \rightarrow D \rightarrow E$. When two nodes are on the same cycle, we say that they are *connected*. For example, nodes A and E are connected if there is a cycle $A \rightarrow \dots \rightarrow E \rightarrow \dots \rightarrow A$ somewhere. All nodes that are connected with each other (that are on one common cycle) form a strongly connected component (SCC).

Within a strongly connected component, each node can be reached from every other node. Therefore, if A and E are connected, and there is a path from A to C, then there must also be a path from E to C. We can imagine a strongly connected component as a group of nodes which are “similar” with

respect to reachability. The graph that is formed by grouping all nodes of a strongly connected component is called the *quotient graph* of the original graph. A quotient graph does not contain cycles any more. The reason for this can be seen in the illustration below: if there were a cycle that connected two strongly connected components, they would ‘melt’ together into one. If in a graph all nodes are connected with each other (as in the above example graph with an additional edge), the graph consists of just one single strongly connected component, and the quotient graph has just one node.



Strongly connected components and quotient graph

Robert E. Tarjan, an American computer scientist (born 1948) who was given the Turing Award for his work on the design and analysis of algorithms and data structures in 1986, extended the algorithm FIND-CYCLE shown above. His famous algorithm for strongly connected components made it possible to find not only cycles, but also the strongly connected components that can be reached from a starting node. In order to do this, every node gets assigned two numbers: firstly, the number under which it appears in the order of depth-first

search, and secondly, the number of the first node of the strongly connected component to which the node belongs.

Strongly connected components

```

1  procedure FIND-COMPONENTS (node  $x$ )
2  begin
3      if mark( $x$ ) = "in progress" then a cycle has been found
4      else if mark( $x$ ) = "not started yet" then
5          mark( $x$ ) := "in progress";
6          depth-first-search-number( $x$ ) := counter;
7          component-number( $x$ ) := counter;
8          counter := counter +1;
9          extend the current search path by appending  $x$ ;
10         for all nodes  $y$  succeeding  $x$  do
11             if mark( $y$ ) is not "done" then
12                 FIND-COMPONENTS( $y$ );
13                 if component-number( $y$ ) < component-number( $x$ ) then
14                     component-number( $x$ ) := component-number( $y$ )
15                 endif
16             endif
17         endfor;
18         if depth-first-search-number( $x$ ) = component-number( $x$ ) then
19             strongly connected component has been found;
20             all nodes on the current path with this component-number are
21             part of the same component
22             for all these nodes  $y$  do
23                 mark( $y$ ) := "done";
24                 remove  $y$  from the current path;
25             endfor;
26         endif
27     endif
28 end
```

The counter is initialized with a fixed value (e.g., 1). An example for the marks of the nodes after running FIND-COMPONENTS can be seen in Fig. 9.1. The quotient graph in this example would have the components 1, 2, and 5; edges lead from 1 to 2, from 1 to 5, and from 5 to 2.

Searching for Cycles with Breadth-First Search

As we can see, depth-first search is very useful for finding all cycles or strongly connected components in a graph. However, if the goal is merely to find out whether a given starting node is part of a cycle, we can use a simpler algorithm: applying the so-called *breadth-first search*, we can determine the set of nodes that can be reached from our starting node. For this, we assume that we have

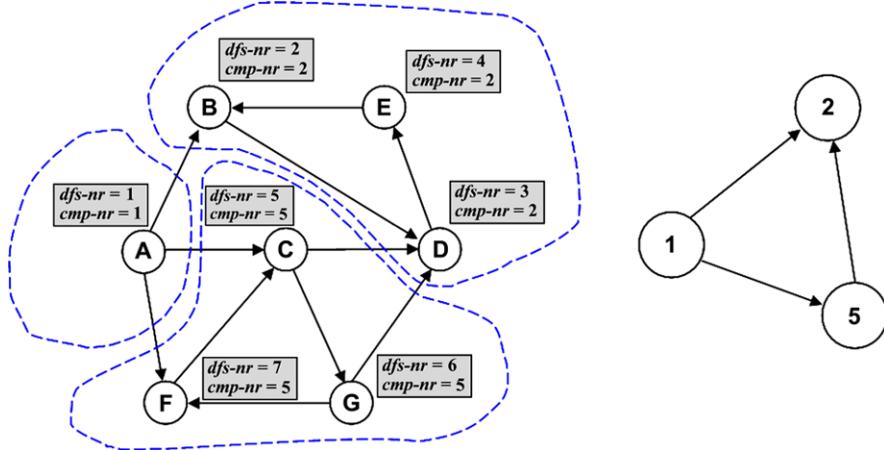


Fig. 9.1. Markings of the nodes after running FIND-COMPONENTS

an efficient method to calculate the set of nodes that are succeeding a given set of nodes (i.e., that can be reached from a node in this set by an edge). And then we start our breadth-first search: the first set of nodes includes only the starting node, the second set includes all nodes that can be reached from the starting node, the third set includes all nodes that can be reached from the nodes of the second set, and so on. At some point, one of the following will happen: either no more nodes can be reached, or we return to our starting node. If the latter is the case, we have our answer: the node is part of a cycle; there is a path that goes from the node back to itself. If the search is done (that is, no more nodes can be reached), and we have not encountered the starting node, then it is obviously not part of a cycle.

Breadth-first search

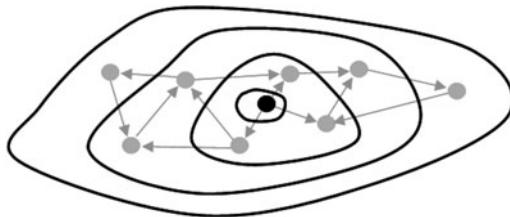
```

1  procedure BREADTH-FIRST-SEARCH (node x)
2  begin
3      reachable := {};front := {x};
4      repeat
5          front := {y | y is successor node of some z from front and
6                  y is not in reachable}
7          if x is in front then cycle from x to x exists, stop;
8      endif;
9      reachable := reachable  $\cup$  front;
10     until front = {}
11 end

```

We can imagine breadth-first search to be somewhat like the propagation of waves that are created when a stone is thrown into a calm lake. The wave

front, the outermost wave, always contains the part of the water surface that can be reached at a time.



The loop that is used in our breadth-first search algorithm is repeated at most as many times as there are nodes, usually much less often. (To be precise, the running time is defined by the longest path from one node to any other node.)

Breadth-first search can be a very fast and efficient way to determine whether some node is part of a cycle. (However, for finding *all* cycles we would have to run breadth-first search starting from all nodes, which would lead to a big increase in running time in comparison to depth-first search.) For breadth-first search, we need to set up and manage the two sets of nodes *front* and *reachable*, which may contain very many nodes compared to depth-first search. The complexity, i.e., the time and space efficiency of the search, depends on the efficiency of the set operations that are being used. With an explicit representation of nodes and edges it is possible to implement breadth-first search in such a way that the running time is proportional to the number of edges in the graph; see, e.g., the book by Sedgewick in the [References](#) below. Large sets and relations can also be represented symbolically as so-called binary decision diagrams. For such a representation, fast library functions exist, which can be used to implement breadth-first search very efficiently.

Historical Notes

The problem of finding cycles in graphs appeared early in the history of computer science. The first examples of application from the 1950s were the search for loops in circuits or data flow diagrams. Depth-first search and recursive algorithms for searching cycles in graphs have been known since the 1960s and are often used as standard examples for backtracking. Tarjan's algorithm to calculate strongly connected components was published in 1972. A very important application of algorithms used to find cycles in graphs is to recognize deadlocks in resource dependency graphs: in every multitasking operating system, cyclic waiting conditions can occur due to wrong synchronization. Popular illustrations for this are Dijkstra's dining philosophers or Lamport's bakery algorithm. Since the 1970s, more and more computer games have appeared that have the player find a way through a virtual labyrinth (a graph),

where a multitude of dangers are waiting and cycles have to be avoided (e.g., Dungeons and Dragons). In the 1990s, new efficient algorithms and data structures to recognize cycles and form quotients were developed for state space exploration in the automatic verification of models. These methods form the basis for analyzing safety-critical control software in planes, trains, and automobiles.

References

1. Robert Sedgewick: *Algorithms in {C|C++|Java}*, Part 5: *Graph Algorithms*. Addison-Wesley Professional, 3rd edition, 2003.
A classic textbook on algorithms that is continually updated and extended.
2. Thomas H. Cormen, Charles Leiserson, Ronald L. Rivest, Clifford Stein: *Introduction to Algorithms*. MIT Press, 2001.
Another widely used book on the topic.
3. Robert E. Tarjan: *Depth-first search and linear graph algorithms*. SIAM Journal on Computing 1 (2), pp. 146–160, 1972.
The original reference for the algorithm for strongly connected components.
4. Edsger W. Dijkstra: *Hierarchical ordering of sequential processes*. Acta Informatica 1 (2), pp. 115–138, 1971.
<http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF>
The original reference for the dining philosophers analogy.
5. Leslie Lamport: *A new solution of Dijkstra’s concurrent programming problem*. Comm. ACM 17 (8), pp. 453–455, Aug. 1974.
<http://research.microsoft.com/en-us/um/people/lamport/pubs/bakery.pdf>
The original reference for the bakery algorithm to synchronize the parallel access to shared resources.
6. Wikipedia articles describing the topics of this chapter:
 - Depth-first search: http://en.wikipedia.org/wiki/Depth-first_search
 - Breadth-first search: http://en.wikipedia.org/wiki/Breadth-first_search
 - Tarjan’s algorithm: http://en.wikipedia.org/wiki/Tarjan%27s_strongly_connected_components_algorithm

Acknowledgement

The author wishes to thank Laura Schlingloff for help with the pictures and translation.

PageRank – What Is Really Relevant in the World-Wide Web?

Ulrik Brandes and Gabi Dorfmüller

Universität Konstanz, Konstanz, Germany

No doubt, the most popular form of Internet usage is the *World-Wide Web* (WWW), a network of billions of files. It is made up, for the most part, of *Web pages* containing text and images that refer to each other via (*hyper*)*links*. Even if you spent your whole life, day and night, doing nothing but browsing pages, you'd see only a small fraction of the Web.¹ To find something on the Web, it is therefore necessary to know where it is, or what links there.

Practically everyone surfing the Web therefore uses search engines, i.e., special pages on which the information sought is described using a few key words (*query*) to obtain a list of pages that may be relevant to the query (*hits*). Using many computer science methods, modern search engines are capable of organizing access to billions of Web pages, and scanning them for matches with a query within fractions of a second.

Since even a query term such as *algorithm* yields millions of hits, the results themselves are too large to be read completely. Search engines therefore sort their results in such a way that the seemingly most relevant hits are shown first.

Quiz:

How do search engines manage to find Web pages that seem relevant to us out of the millions matching a typical query?

As of today, the best known search engine is run by a company called Google,² since it was the first search engine to not only sift through an enormous number of pages, but also to use a particularly clever algorithm to rank the results. Among, e.g., more than a million German-language hits matching *Algorithmus*, the site of the original project leading to this book currently ranks second only to the corresponding Wikipedia article.

¹ Assume that you are spending a second per page – how many seconds does an average life last?

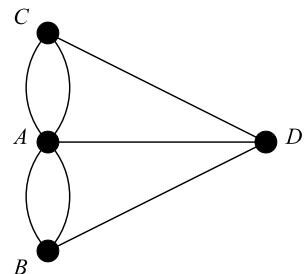
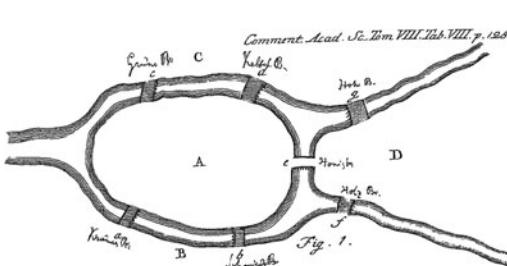
² www.google.com

Besides many straightforward criteria such as the location of query terms in the page (in headlines? near each other?) and many unknown heuristic rules, a central element of the ranking strategy is an evaluation of the Web's linking structure. This component is known as PageRank and is explained in this chapter.

Tourist Trails

Explanations of PageRank often use for motivation the idea that a page should be ranked as more relevant, the more frequently one would reach it on a random walk through the Web. We will explore this idea further, but with a completely different example.

Imagine that, in the 18th century, mathematician Leonhard Euler had not proven the inexistence of, but rather found the long-sought tour crossing the seven bridges of Königsberg (cf. Chap. 28). This tour would be famous: It would be listed in all city guides, and tourists would walk the tour in droves. Of course, there would also be vendors selling souvenirs and refreshments in places that these tourists most frequently stroll by – but where are these places?



For a tour it does not matter where it is begun. Since, however, every bridge is crossed exactly once, we can at least be sure that every part is visited half as often as there are bridges leading there: One bridge is needed to get there, and another one to leave. The most promising selling spots are where the most bridges converge. In Königsberg, this would be Kneiphof (labeled A).

Alas, there is no such tour. So let's assume tourists are wandering around with no particular goal or destination. More concretely, let them choose the next bridge to cross randomly and with equal probabilities (this is called “uniformly at random”) from all those that are feasible, including the one they just came across. How often do they arrive in a certain location?

The number b of visits at, say, node B can be described in terms of the number of immediately preceding visits at nodes that are connected to B , here A and D . If the next bridge to continue with is chosen uniformly at random from all feasible bridges, we get from A to B in two out of five cases,

and from D to B in one out of three. The unknown b can therefore be written in terms of equally unknown visiting numbers a and d :

$$b = \frac{2}{5}a + \frac{1}{3}d.$$

Corresponding equations can be given for all the unknowns:

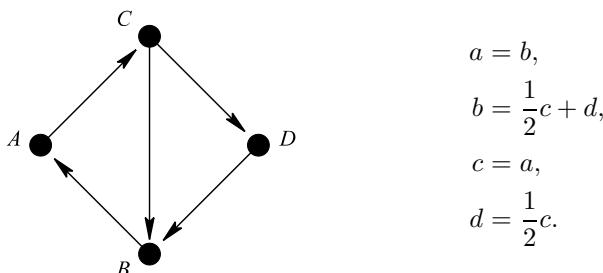
$$a = \frac{2}{3}b + \frac{2}{3}c + \frac{1}{3}d, \quad c = \frac{1}{5}a + \frac{1}{3}d, \quad d = \frac{1}{5}a + \frac{1}{3}b.$$

Interestingly, every solution to this system of equations is of the form $a = 5$ and $b = c = d = 3$ or multiples thereof; the relative sizes of these numbers are thus exactly the same as those we would have obtained had there been an Eulerian tour! Whether tourists are exploring Königsberg systematically or at random is therefore of no relevance for our vendors. Moreover, this principle applies to every other city as well, independent of its particular pattern of connections among bridges.

Trails on the Web

If we interpret hyperlinks as a recommendation to consult the destination Web page for further information, we can ask the same question that has just been considered for locations in Königsberg. Which pages does a random surfer, someone who is not searching for something in particular and is following links uniformly at random, visit most frequently? It would seem that the answer depends on the respective number of links entering a page, as it did for bridges.

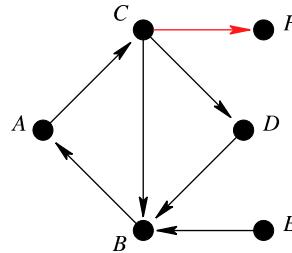
In contrast to the bridges of Königsberg, though, links on the Web are like one-way streets, because they can only be followed in one direction. Note that we are ignoring the “back” button for the moment. The following example illustrates that this simple modification complicates matters significantly.



All arguments for setting up a system of equations for the unknown relative numbers of visits are still valid, but the solutions of the equation in this particular example are multiples of $a = b = c = 2$ and $d = 1$. The correspondence between these numbers and the number of incoming and outgoing links is lost

(for otherwise, a would have to equal d and both must be different from b and c).

Real networks in general exhibit at least one other problem in addition to one-way streets, namely dead ends. In the network depicted here,



one is stuck after following the red link. While simple cases like this one are easily identified (and Google is likely to remove them), dead ends can also be less obvious. In larger networks one might be able to continue from nodes such as F , but still not be able to return to nodes A, \dots, E . Pages that eventually cannot be reached any more lead to solutions of the system of equations that are not useful for ranking purposes.

The Web-surfing behavior mimicked so far, however, is a poor match. If no, or no interesting, link is found on a page, the next page will be chosen by other means such as the back button, a bookmark, or direct entry of a new Web address.

Including such spontaneous jumps to other pages into the model yields a system of equations that is only a little more complicated. We simply assume that, for instance, on every fifth occasion the new node was not found by selecting a link, but directly by some other means. We also assume that there is no bias toward any of the six pages, i.e., in the long run, each node is equally often the destination of a jump. This way, every page can be reached at any point in time, and there are no dead ends.

$$\begin{aligned}
 a &= \frac{4}{5} \cdot b + \frac{1}{5} \cdot \frac{1}{6}, & d &= \frac{4}{5} \cdot \left(\frac{1}{3} \cdot c \right) + \frac{1}{5} \cdot \frac{1}{6}, \\
 b &= \frac{4}{5} \cdot \left(\frac{1}{3} \cdot c + 1 \cdot d + e \right) + \frac{1}{5} \cdot \frac{1}{6}, & e &= \frac{4}{5} \cdot 0 + \frac{1}{5} \cdot \frac{1}{6}, \\
 c &= \frac{4}{5} \cdot a + \frac{1}{5} \cdot \frac{1}{6}, & f &= \frac{4}{5} \cdot \left(\frac{1}{3} \cdot c \right) + \frac{1}{5} \cdot \frac{1}{6}.
 \end{aligned}$$

This system of equations is only a little more involved and just as easy to solve as the system in the previous section. Try to determine whether results from the following experiment resemble a solution!

Experiment (10 or more subjects, e.g., your class at school)

Every person starts from any of the pages in the above network and starts moving through the network by following links at random. If necessary, or out of the blue, any page can be chosen. After one minute everyone stops upon a signal and remembers the last page visited. For each page, the number of subjects that have stopped there is recorded.

A less contrived and somewhat larger example of links between entities is the present book itself. If chapters are viewed as reading locations, referrals can be interpreted as links between them.

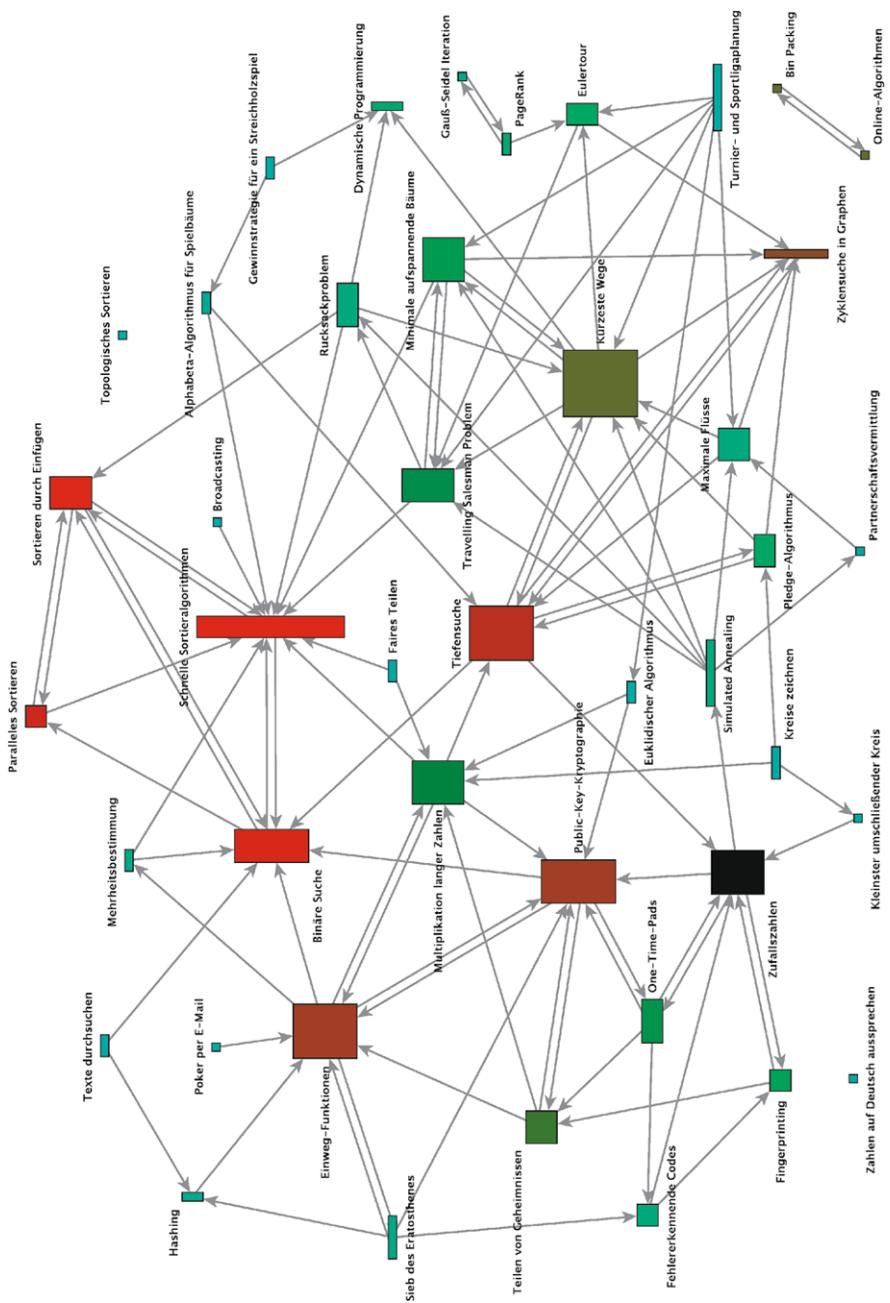
In the network diagram depicted on the next page every chapter in this book is represented by a rectangle, and their width and height is determined by the number of links to and from other chapters. A slim, tall rectangle thus represents a chapter that refers to few others, but is referred to often. Colors are computed from the equation system and therefore indicate the PageRank in the referral network: Turquoise rectangles represent chapters with lowest PageRank, orange rectangles those with highest PageRank, and appropriately mixed colors are used for values in between.

The chapters on variants of sorting are indeed the ones that a reader ends up being referred to most often when reading in random fashion, following links as if on the Web. This matches their crucial role in algorithmics.

Solutions

The model described above cannot be used directly to compare the relevance of hits returned for a query, since the network that is to be evaluated by an Internet search engine yields a system with billions of equations. Even the fastest computers cannot solve such systems using the methods we learn in school.

Fortunately, the systems resulting from our approach exhibit some special properties that can be exploited in the search for a solution. Moreover, we do not really need an exact solution, so that a very simple and efficient algorithm can instead be used to quickly approximate a solution. There is one equation for every variable, and if we know the values of all other variables, we would simply substitute them and thus determine the value of the final variable. The algorithm therefore starts with an arbitrary initial assignment (e.g., the same non-zero value for all variables) and solves for every variable of the associated equation relative to the values currently assigned for all other variables. With these newly obtained assignments the process is repeated, then repeated, and repeated, and so on.



PageRank Algorithm (almost)

- 1 Initialize relevance scores of all pages to 1
- 2 While scores are changing notably
- 3 Determine for every page P :
- 4 set new relevance score for P to

$$\frac{4}{5} \cdot \sum_{\substack{\text{for all pages } Q \\ \text{linking } Q \rightarrow P}} \frac{\text{relevance of } Q}{\text{number of links of } Q} + \frac{1}{5} \cdot \frac{1}{\text{number of pages}}$$

For our six-node example, this yields the following computation (values rounded to the 5th decimal):

	Start	1st step	2nd step	...	11th step	12th step	...	Solution
a	1.00000	0.83333	0.29467	...	0.10758	0.10740	...	0.10665
b	1.00000	0.32667	0.28222	...	0.09259	0.09241	...	0.09164
c	1.00000	0.83333	0.70000	...	0.12154	0.11940	...	0.11865
d	1.00000	0.30000	0.25556	...	0.06592	0.06574	...	0.06497
e	1.00000	0.03333	0.03333	...	0.03333	0.03333	...	0.03333
f	1.00000	0.30000	0.25556	...	0.06592	0.06574	...	0.06497

The scores are improving with every step, and when they are not changing by much anymore, this is an indication that we are close to the exact solution. A detailed explanation for this welcome behavior is given in Chap. 30.

Conclusion

At the end of this chapter the following question should be easy to answer.

Quiz: If you let many of your friends link to your personal *homepage*, will it turn out on top of Google's result list?

Answer: This is only going to work if your friends' pages are of high relevance themselves – unlikely to be the case.

Many other ways of scoring linked entities are studied in an exciting area of research called network analysis. Moreover, there are many ways in which these structural scores can be utilized in the final relevance ranking. A change in the slightest detail can have a significant effect on the results. For example, we could alter the ratio of jumps and link traversal, or introduce a bias into the selection of destinations for jumps. The specific instantiation of the algorithm and many more details of the relevance ranking are, of course, proprietary knowledge, but it seems that the decisions made at Google are working not too badly.

Further Reading

1. Chapter 30 (Gauß–Seidel Iterative Method for the Computation of Physical Problems)

More detailed explanation of how and when systems of linear equations can be solved by iteratively solving single equations.

2. Chapter 28 (Eulerian Circuits)

There are no Eulerian tours in Königsberg, but there are where Santa dwells.

3. Wikipedia article

<http://en.wikipedia.org/wiki/PageRank>

More abstract formulation, background information and more references.

Part II

Arithmetic and Encryption

Overview

Berthold Vöcking

RWTH Aachen, Aachen, Germany

Even in the very first years of school we were confronted with algorithms for basic arithmetic. We were taught how to add multidigit numbers by writing the numbers below each other and then adding digit by digit, carrying an overflow digit from right to left. Based on this algorithm for addition, we then learned an algorithm for multiplication. Two multidigit numbers are multiplied by first multiplying the multiplicand by each digit of the multiplier and then adding up all the properly shifted results. These grade-school algorithms follow simple rules and can thus be executed using a calculator or a computer which typically, however, use binary rather than decimal digits. In fact, every pocket calculator does these calculations much faster and more reliably than humans and, thus, we are no longer accustomed to performing them by hand.

This part of the book starts with algorithms for different kinds of arithmetic problems. In Chap. 11, an algorithm for fast multiplication is presented that is much more efficient than the grade school method, especially if one wants to multiply large numbers consisting of many digits. Chapter 12 deals with the Euclidean Algorithm for calculating the greatest common divisor of two given numbers in a very clever and elegant way. This algorithm was known already in the ancient world and is still used today in different variants. In ancient times, it was even known how to calculate prime numbers. Chapter 13 explains the Sieve of Eratosthenes, a very old but still practical algorithm for computing a table with all prime numbers up to a specified number.

Cryptography deals with the encryption and decryption of information. Chapters 15 and 16 present two different cryptographic approaches. Chapter 15 presents the so-called One-Time-Pad. This simple symmetric algorithm uses the same secret key for encryption and decryption. A text can be encrypted and decrypted only by someone who knows the secret key. Chapter 16 presents an asymmetric method that uses different keys for encryption and decryption. The key for encryption is announced publically so that everybody can encode a message. Only the owner of the matching secret key can decode the encrypted message. Most cryptographic schemes used today in the Internet are based on asymmetric algorithms using a secret and a public key.

Chapter 17 deals with cryptographic methods for sharing information. For example, a gang of pirates can share a treasure map in such a way that all pirates must meet in order to find the treasure, or a group of three bank clerks can share a code for a safe in such a way that it is necessary and sufficient for opening the safe if any two of the three clerks combine their codes. Chapter 18 describes an interesting application of cryptographic methods, it explains how a group of people can play poker by email, without giving an unfair advantage to any of the players.

Chapter 14 presents one-way functions, which play an important role in cryptography. A one-way function can be computed efficiently, but their inverse is very difficult to compute. Like most cryptographic algorithms they rely on findings from number theory. For example, two prime numbers of several hundred digits can be multiplied very quickly by a computer using the algorithms for multiplication mentioned above. Given only the product of these numbers, however, it is extremely difficult to factorize the product into the two prime factors. In fact, the asymmetric cryptographic algorithm described in Chap. 16 relies on this difficulty. The best known algorithms running on the fastest computers, even if we would combine the computational power of all existing computers, could not solve the factoring problem within a period of time a human could wait for.

The other three chapters of this part deal with different approaches for the compression and coding of data. Chapters 19 and 20 present so-called fingerprinting and hashing methods that condense large data sets in an extreme way so that they are represented by only a few bits. Of course, such extreme compression comes with a loss of information. However, the condensed data can, for example, be used in order to check whether two large data sets are identical by exchanging only a few bits of information, just like fingerprints are used to distinguish and identify humans. Chapter 21 discusses coding algorithms that do not condense data but, on the contrary, add a few bits in order to protect data against errors and loss. The highlight is a quite recent finding that coding algorithms can be used to increase the capacity of a network. This trick is called network coding and it is a hot research area today.

Multiplication of Long Integers – Faster than Long Multiplication

Arno Eigenwillig* and Kurt Mehlhorn

Max-Planck-Institut für Informatik, Saarbrücken, Germany

An algorithm for multiplication of integers is taught in primary school: To multiply two positive integers a and b , you multiply a by each digit of b and arrange the results as the rows of a table, aligned under the corresponding digits of b . Adding up yields the product $a \times b$. Here is an example:

$$\begin{array}{r} 5678 \cdot 4321 \\ \hline 22712 \\ 17034 \\ 11356 \\ \hline 5678 \\ \hline 24534638 \end{array}$$

The multiplication of a by a single digit is called *short multiplication*, and the whole method to compute $a \times b$ is called *long multiplication*. For integers a and b with very many digits, long multiplication does indeed take long, even if carried out by a modern computer. Calculations with very long integers are used in many applications of computers, for example, in the encryption of communication on the Internet (see Chaps. 14 and 16), or, to name another example, in the reliable solution of geometric or algebraic problems.

Fortunately, there are better ways to multiply. This is good for the applications needing it. But it is also quite remarkable in itself because long multiplication is so familiar and looks so natural that any substantial improvement comes as a surprise.

In the rest of this chapter, we will investigate:

1. How much effort does it take to do long multiplication of two numbers?
2. How can we do better?

Computer scientists do not measure the effort needed to carry out an algorithm in seconds or minutes because such information will depend on the hardware, the programming language, and the details of the implementation

* Now at Google Zurich, Switzerland.

(and next year's hardware will be faster anyway). Instead, computer scientists count the number of *basic operations* performed by an algorithm. A basic operation is something a computer or a human can do in a single step. The basic operations we need here are basic computations with the digits $0, 1, 2, \dots, 8, 9$.

1. **Multiplication of two digits:** When given two digits x and y , we know the two digits u and v that make up their product $x \times y = 10 \times u + v$. We trust our readers to remember the multiplication table!

Examples: For the digits $x = 3$ and $y = 7$ we have $x \times y = 3 \times 7 = 21 = 10 \times 2 + 1$, so the resulting digits are $u = 2$ and $v = 1$. For $x = 3$ and $y = 2$, the resulting digits are $u = 0$ and $v = 6$.

2. **Addition of three digits:** When given three digits x, y, z , we know the two digits u and v of their sum: $x + y + z = 10 \cdot u + v$. It will soon become obvious why we want to add three digits at once.

Example: For $x = 3$, $y = 5$ and $z = 4$ the result is $u = 1$ and $v = 2$ because $3 + 5 + 4 = 12 = 10 \times 1 + 2$.

How many of these basic operations are done in a long multiplication? Before we can answer this question, we need to look at two simpler algorithms, addition and short multiplication, used during long multiplication.

The Addition of Long Numbers

How much effort does it take to add two numbers a and b ? Of course, this depends on how many digits they have. Let us assume that a and b both consist of n digits. If one of them is shorter than the other, one can put zeros in front of it until it is as long as the other. To add the two numbers, we write one above the other. Going from right to left, we repeatedly do addition of digits. Its result $10 \times u + v$ gives us the result digit v for the present column as well as the digit u carried to the next column. Here is an example with numbers $a = 6917$ and $b = 4269$ with $n = 4$ digits each:

$$\begin{array}{r} 6917 \\ 4269 \\ \hline 11186 \end{array}$$

The carry digit v from the leftmost column is put in front of the result without further computation. Altogether, we have done n basic operations, namely one addition of digits per column.

Short Multiplication: A Number Times a Digit

During long multiplication, we need to multiply the number a , the left factor, with a digit y from the right factor. We now look at this *short multiplication* in more detail. To do so, we write down its intermediate results more carefully than usual: We go from right to left over the digits of a . We multiply each

digit x of a by the digit y and write down the result $10 \times u + v$ in a separate row, aligned such that v is in the same column as x . When all digits are multiplied, we add all the two-digit intermediate results. This gives us the result of the short multiplication, which is usually written as a single row.

As an example for this, we look again at the long multiplication 5678×4321 . The first of the short multiplications it needs is this one:

$$\begin{array}{r} 5678 \cdot 4 \\ \hline 32 \\ 28 \\ 24 \\ 20 \\ \hline 0010 \\ \hline 22712 \end{array}$$

How many basic operations did we use? For each of the n digits of a , we have done one multiplication of digits. In the example above: four multiplications of digits for the four digits of 5678. After that, we have added the intermediate results in $n+1$ columns. In the rightmost column, there is a single digit which we can just copy to the result without computing. In the other n columns are two digits and a carry from the column to its right, so one addition of digits suffices to add them. This means we needed to do n additions of digits. Together with the n multiplications of digits, it has taken $2 \times n$ basic operations to multiply an n -digit number a with a digit y .

The Analysis of Long Multiplication

Let us now analyze the number of basic operations used by long multiplication of two numbers a and b , each of which has n digits. In case one is shorter than the other, we can pad it with zeros at the front.

For each digit y of b we need to do one short multiplication $a \times y$. This needs $2 \times n$ basic operations, as we saw above. Because there are n digits in b , long multiplication needs n short multiplications which together account for $n \times (2 \times n) = 2 \times n^2$ basic operations.

The results of the short multiplications are aligned under the respective digits of b . To simplify the further analysis, we put zeros in the empty positions:

$$\begin{array}{r} 5678 \cdot 4321 \\ \hline 22712000 \\ 01703400 \\ 00113560 \\ 00005678 \\ \hline 24534638 \end{array}$$

The results of short multiplication are added with the method described previously. We add the first row to the second row, their sum to the third row,

and so on, until all n rows have been added. This needs $n - 1$ additions of long numbers. In our example, n is equal to 4, and we need these $n - 1 = 3$ additions: $22712000 + 1703400 = 24415400$, $24415400 + 113560 = 24528960$ and $24528960 + 5678 = 24534638$.

How many basic operations do these $n - 1$ additions need? To answer this, we have to know how many digits are required for the intermediate sums in this chain of additions. With a little bit of thinking, we can convince ourselves that the final result $a \times b$ has at most $2 \times n$ digits. While we add the parts of this result, the numbers can only get longer. Therefore, all intermediate sums have at most $2 \times n$ digits, like the final result. That means, we do $n - 1$ additions of numbers with at most $2 \times n$ digits. According to our analysis of addition, this requires at most $(n - 1) \times (2 \times n) = 2 \times n^2 - 2 \times n$ basic operations. Together with the $2 \times n^2$ basic operations used by the short multiplications, this yields a grand total of at most $4 \times n^2 - 2 \times n$ basic operations carried out in the long division of two n -digit numbers.

Let us see what this means for a concrete example. If we have to multiply really long numbers, say, with 100 000 digits each, then it takes almost 40 billion basic operations to do one long multiplication, including 10 billion multiplications of digits. In other words: Per digit in the result, this long multiplication needs, on average, 200 000 basic operations, which is clearly a bad ratio. This ratio gets much worse if the number of digits increases: For 1 million digits, long multiplication needs almost 4 trillion basic operations (of which 1 trillion are multiplications of digits). On average, it spends about 2 million basic operations for a single digit in the result.

Karatsuba's Method

Let us now do something smarter. We look at an algorithm for multiplying two n -digit numbers that needs far fewer basic operations. It is named after the Russian mathematician Anatolii Alexeevitch Karatsuba, who came up with its main idea (published 1962 with Yu. Ofman²). We first describe the method for numbers with one, two, or four digits, and then for numbers of any length.

The simplest case is, of course, the multiplication of two numbers consisting of one digit each ($n = 1$). Multiplying them needs a single basic operation, namely one multiplication of digits, which immediately gives the result.

The next case we look at is the case $n = 2$, that is, the multiplication of two numbers a and b having two digits each. We split them into halves, that is, into their digits:

² A. Karatsuba, Yu. Ofman: “Multiplication of multidigit numbers on automata” (in Russian), *Doklady Akad. Nauk SSSR* **145** (1962), pp. 293–294; English translation in *Soviet Physics Doklady* **7** (1963), pp. 595–596. Karatsuba describes his method to efficiently compute the square a^2 of a long number a . The *multiplication* of numbers a and b is reduced to squaring with the formula $a \times b = \frac{1}{4}((a + b)^2 - (a - b)^2)$.

$$a = p \times 10 + q \quad \text{and} \quad b = r \times 10 + s.$$

For example, we split the numbers $a = 78$ and $b = 21$ like this:

$$p = 7, \quad q = 8, \quad \text{and} \quad r = 2, \quad s = 1.$$

We can now rewrite the product $a \times b$ in terms of the digits:

$$\begin{aligned} a \times b &= (p \times 10 + q) \times (r \times 10 + s) \\ &= (p \times r) \times 100 + (p \times s + q \times r) \times 10 + q \times s. \end{aligned}$$

Continuing the example $a = 78$ and $b = 21$, we get

$$78 \cdot 21 = (7 \cdot 2) \cdot 100 + (7 \cdot 1 + 8 \cdot 2) \cdot 10 + 8 \cdot 1 = 1638.$$

Writing the product $a \times b$ of the two-digit numbers a and b as above shows how it can be computed using **four** multiplications of one-digit numbers, followed by additions. This is precisely what long multiplication does.

Karatsuba had an idea that enables us to multiply the two-digit numbers a and b with just **three** multiplications of one-digit numbers. These three multiplications are used to compute the following auxiliary products:

$$\begin{aligned} u &= p \times r, \\ v &= (q - p) \times (s - r), \\ w &= q \times s. \end{aligned}$$

Computing v deserves extra attention because it involves a new kind of basic operation: subtraction of digits. It is used twice in computing v . Its results $(q - p)$ and $(s - r)$ are again single digits, but possibly with a negative sign. Multiplying them to get v requires a multiplication of digits and an application of the usual rules to determine the sign (“minus times minus gives plus”, and so on).

Why does all this help to multiply a and b ? The answer comes from this formula:

$$u + w - v = p \times r + q \times s - (q - p) \times (s - r) = p \times s + q \times r.$$

Karatsuba’s trick consists in using this formula to express the product $a \times b$ in terms of the three auxiliary products u , v , and w :

$$a \times b = u \times 10^2 + (u + w - v) \times 10 + w.$$

Let us carry out Karatsuba’s trick for our example $a = 78$ and $b = 21$ from above. The three Karatsuba multiplications are

$$\begin{aligned} u &= 7 \times 2 &= 14, \\ v &= (8 - 7) \times (1 - 2) &= -1, \\ w &= 8 \times 1 &= 8. \end{aligned}$$

We obtain

$$\begin{aligned} 78 \times 21 &= 14 \times 100 + (14 + 8 - (-1)) \times 10 + 8 \\ &= 1400 + 230 + 8 \\ &= 1638. \end{aligned}$$

We have used two subtractions of digits, three multiplications of digits, and several additions and subtractions of digits to combine the results of the three multiplications.

Karatsuba's Method for 4-Digit Numbers

Having dealt with the case of $n = 2$ digits above, we now look at the case of $n = 4$ digits, that is, the multiplication of two numbers a and b with four digits each. Just like before we can split each of them into two halves p and q , or r and s , respectively. These halves are not digits anymore, but two-digit numbers:

$$a = p \times 10^2 + q \quad \text{and} \quad b = r \times 10^2 + s.$$

Again, we compute the three auxiliary products from these four halves:

$$\begin{aligned} u &= p \times r, \\ v &= (q - p) \times (s - r), \\ w &= q \times s. \end{aligned}$$

Just like before, we obtain the product $a \times b$ from the auxiliary products as

$$a \times b = u \times 10^4 + (u + w - v) \times 10^2 + w.$$

Example: We look again at the task of multiplying $a = 5678$ and $b = 4321$. We begin by splitting a and b into the halves $p = 56$ and $q = 78$ as well as $r = 43$ and $s = 21$. We compute the auxiliary products

$$\begin{aligned} u &= 56 \times 43 &= 2408, \\ v &= (78 - 56) \times (21 - 43) &= -484, \\ w &= 78 \times 21 &= 1638. \end{aligned}$$

It follows that

$$\begin{aligned} 5678 \times 4321 &= 2408 \times 10000 + (2408 + 1638 - (-484)) \times 100 + 1638 \\ &= 24080000 + 453000 + 1638 \\ &= 24534638. \end{aligned}$$

In this calculation, we had to compute three auxiliary products of two-digit numbers. In the previous section, we investigated how to do that with Karatsuba's method, using only three multiplications of digits each time. This way, we can compute the three auxiliary products using only $3 \times 3 = 9$ multiplications of digits and several additions and subtractions. Long division would have taken 16 multiplications of digits and several additions.

Karatsuba's Method for Numbers of Any Length

Recall how we built Karatsuba's method for multiplying 4-digit numbers from Karatsuba's method for 2-digit numbers. Continuing in the same way, we can build the multiplication of 8-digit numbers from three multiplications of 4-digit numbers, and the multiplication of 16-digit numbers from three multiplications of 8-digit numbers, and so on. In other words, Karatsuba's method works for any number n of digits that is a power of 2, such as $2 = 2^1$, $4 = 2 \times 2 = 2^2$, $8 = 2 \times 2 \times 2 = 2^3$, $16 = 2 \times 2 \times 2 \times 2 = 2^4$, and so on.

The general form of Karatsuba's method is this: Two numbers a and b , each consisting of $n = 2 \times 2 \times 2 \times \dots \times 2 = 2^k$ digits, are split into

$$a = p \times 10^{n/2} + q \quad \text{and} \quad b = r \times 10^{n/2} + s.$$

Then their product $a \times b$ is computed as follows, using three multiplications of numbers having $n/2 = 2^{k-1}$ digits each:

$$a \times b = p \times r \times 10^n + (p \times r + q \times s - (q - p) \times (s - r)) \times 10^{n/2} + q \times s.$$

Multiplying two numbers of 2^k digits in this way takes only three times (and not four times) as many multiplications of digits as multiplying two numbers with 2^{k-1} digits. This leads to the following table which compares how many multiplications of digits are used by Karatsuba's method, or by long multiplication, respectively, to multiply numbers with n digits.

Digits	Karatsuba	Long multiplication
$1 = 2^0$	1	1
$2 = 2^1$	3	4
$4 = 2^2$	9	16
$8 = 2^3$	27	64
$16 = 2^4$	81	256
$32 = 2^5$	243	1024
$64 = 2^6$	729	4096
$128 = 2^7$	2187	16 384
$256 = 2^8$	6561	65 536
$512 = 2^9$	19 638	262 144
$1024 = 2^{10}$	59 049	1 048 576
$1 048 576 = 2^{20}$	3 486 784 401	1 099 511 627 776
\dots	\dots	\dots
$n = 2^k$	3^k	4^k

Using logarithms, it is easy to express the entries in the table as functions of n : For $n = 2^k$, the column for long multiplication contains the value 4^k . We write \log for the logarithm with base 2. We have $k = \log(n)$ and

$$4^k = 4^{\log(n)} = (2^{\log(4)})^{\log(n)} = n^{\log(4)} = n^2.$$

For $n = 2^k$, the column for Karatsuba's method contains the value

$$3^k = 3^{\log(n)} = (2^{\log(3)})^{\log(n)} = n^{\log(3)} = n^{1.58\dots}.$$

Let us return to the question of how much effort it takes to multiply two numbers of one million digits each. Long multiplication takes almost 4 trillion basic operations, including 1 trillion multiplications of digits. To use Karatsuba's method instead, we first need to put zeros in front of both numbers, to bring their length up to the next power of two, which is $2^{20} = 1\,048\,576$. Without this padding, we could not split the numbers in halves again and again until we reach a single digit. With Karatsuba's method, we can multiply the two numbers using "only" 3.5 billion multiplications of digits, one 287th of what long multiplication needed. For comparison: One second is a 300th of the proverbial "five minutes". So we see that Karatsuba's method indeed requires much less computational effort, at least when counting only multiplications of digits, as we have done. A precise analysis also has to take the additions and subtractions of intermediate results into account. This will show that long multiplication is actually faster for numbers with only a few digits. But as numbers get longer, Karatsuba's method becomes superior because it produces less intermediate results. It depends on the properties of the particular computer and the representation of long numbers inside it when exactly Karatsuba's method is faster than long multiplication.

Summary

The recipe for success in Karatsuba's method has two ingredients.

The first is a very general one: The task "multiply two numbers of n digits each" is reduced to several tasks of the same form, but of smaller size, namely "multiply two numbers of $n/2$ digits each." We keep on subdividing until the problem has become simple: "multiply two digits." This problem-solving strategy is called *divide and conquer*, and it has appeared before in this book (for example, in Chap. 3 on fast sorting). Of course, a computer does not need a separate procedure for each size of the problem. Instead, there is a general procedure with a parameter n for the size of the problem, and this procedure invokes itself several times for the reduced size $n/2$. This is called *recursion*, and it is one of the most important and fundamental techniques in computer science. Recursion also has appeared before in this book (for example, in Chap. 7 on depth-first search).

The second ingredient in Karatsuba's method, specifically aimed at multiplication, is his trick to divide the problem in a way that results in three (instead of four) subproblems of half the size. Accumulated over the whole recursion, this seemingly minuscule difference results in significant savings and gives Karatsuba's method its big advantage over long multiplication.

Further Reading

1. A.A. Karatsuba: *The complexity of computations*. Proceedings of the Steklov Institute of Mathematics, Vol. 211, 1995, pp. 169–183, available at <http://www.ccas.ru/personal/karatsuba/divcen.pdf>
A.A. Karatsuba reports about the history of his invention and describes it in his own words.
2. A.K. Dewdney: *The (New) Turing Omnibus*. Computer Science Press, Freeman, 2nd edition, 1993; reprint (paperback) 2001.
These “66 excursions in computer science” include a visit to the multiplication algorithms of Karatsuba (for long numbers) and Strassen (a similar idea for matrices).
3. Wolfram Koepf: *Computeralgebra*. Springer, 2006.
A gentle introduction to computer algebra. Unfortunately, only available in German.
4. Joachim von zur Gathen, Jürgen Gerhard: *Modern Computer Algebra*. Cambridge University Press, 2nd edition, 2003.
This beautifully prepared textbook for advanced students of computer science and mathematics discusses Karatsuba’s method and more advanced methods (based on Fast Fourier Transformation) for the multiplication of polynomials.
5. Donald E. Knuth: *The Art of Computer Programming*, Vol. 2: *Seminumerical Algorithms*. Addison-Wesley, 3rd edition, 1998.
This heavyweight classic of theoretical computer science treats Karatsuba’s method and other, more advanced algorithms for efficient multiplication of integers, in particular the algorithm of Schönhage and Strassen, whose running time has a bound proportional to $n \times \log(n) \times \log(\log(n))$.
6. Martin Fürer: *Faster integer multiplication*. Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing, 2007, pp. 57–66.
The currently asymptotically best method for multiplying long integers. Its running time is proportional to $n \log(n) 2^{O(\log^*(n))}$.
7. Wikipedia:
http://en.wikipedia.org/wiki/Karatsuba_algorithm
http://en.wikipedia.org/wiki/Multiplication_algorithm

Acknowledgements

The authors thank H. Alt, M. Dietzfelbinger and C. Kloß for helpful remarks on an earlier version of this chapter.

The Euclidean Algorithm

Friedrich Eisenbrand

EPFL, Lausanne, Switzerland

This chapter deals with one of the oldest algorithms that appears in records from the ancient world. The algorithm is described in *The Elements*, the famous book by Euclid, which was written roughly 300 BC. Nowadays, this algorithm is a cornerstone in many areas of computer science, especially in the area of cryptography, see Chap. 16, where many fundamental routines rely on the fact that the greatest common divisor of two numbers can be efficiently computed.

Imagine that you have two bars of length a and b , respectively, where both a and b are integers. You want to cut both bars into pieces, each having the same length. Your goal is to cut the bars in such a way that the common length of the pieces is as large as possible. We could, for example, cut the bar of length a into a many pieces of length 1 and the bar of length b into b many pieces of length 1. Is a larger common length of the pieces possible?

Our algorithm computes the largest possible common length of the pieces. We describe two versions of the algorithm. The first version is slow, or *inefficient*. The second version is fast, or *efficient*.

Let d denote the largest common length of the pieces that can be possibly achieved. The bar of length a and the bar of length b are cut into a/d and b/d

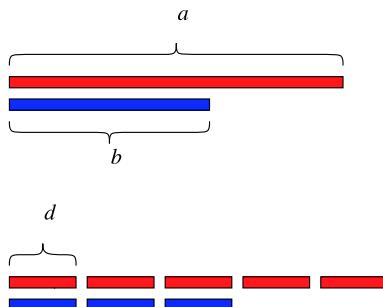


Fig. 12.1. Cutting two bars into pieces of common length d

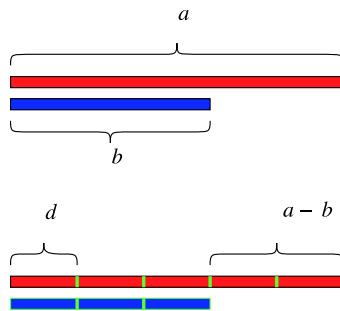


Fig. 12.2. The common length of pieces that we search for a and b is the common length of pieces for $a - b$ and b

many pieces, respectively. The picture above displays a situation where a is cut into 5 and b is cut into 3 pieces. How can we find the largest d ?

If both bars have equal length, i.e., if $a = b$, then the value of d is immediately clear. We do not have to cut the bars at all. The largest length d such that we can cut a and b into d -sized pieces is the common length of the bars itself. Let us therefore assume that the length of the two bars is different, where we assume that a is larger than b . As you lay both bars next to each other, you make an important observation, see the figure above. If we can cut both bars into pieces of length d , then we can cut off a piece of length b from the larger bar.

The resulting bar has length $a - b$ and can be cut into pieces of length d as well. Conversely, if we can cut both bars, the one of length $a - b$ and the one of length b , into pieces of length d , then we can cut also a into equal pieces of length d .

We formulate this insight separately. It is the main principle underlying our algorithm.

Principle (P)

If $a = b$, then the length d we are looking for is a .

If a is larger than b , then the common length of pieces for a and b is the common length of pieces for $a - b$ and b .

We can now formulate an algorithm that computes the largest length d of pieces into which a and b can be cut.

Largest common length of pieces

While both bars do not have equal length:

Cut off from the larger bar a piece being as long as the smaller bar and put this piece aside.

Now both bars have equal length. This common length is the length d we are looking for.

At this point we must ask ourselves whether the above algorithm ever finishes or, in computer science terminology, *terminates*. We can observe that it indeed does. Remember that the lengths of the bars in the beginning are integers a and b , respectively. The lengths of the bars remain integers as we cut off a piece from the longer bar that is as long as the shorter bar. In particular, the length of both bars is at least 1. As we cut one bar, we remove at least a piece of length 1. Thus the algorithm performs at most $a + b$ rounds.

The Greatest Common Divisor

Our analysis from above reveals that the length d that we are computing is also an integer. It is an integer which *divides* both a and b . In mathematical terminology this means that there are integers x and y such that $a = x \cdot d$ and $b = y \cdot d$, respectively. The number d is the largest number which has this property that there exist integers x and y as above. The number d is the *greatest common divisor* of a and b . The integers x and y are the number of pieces of length d into which the bars of length a and b are cut, respectively.

We can also describe our algorithm in more abstract terminology, where we no longer use bars. The inputs to our algorithm are two positive integers

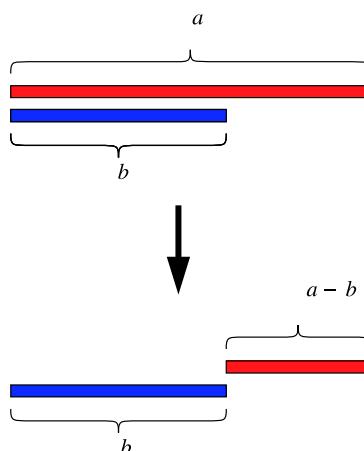


Fig. 12.3. One step of the algorithm

gers a and b . The output of our algorithm is the greatest common divisor of a and b . We call the algorithm *SlowEuclid* for a reason that is soon going to be illuminated.

SLOWEUCLID

While $a \neq b$

If a is larger than b , then replace a by $a - b$

If b is larger than a , then replace b by $b - a$

Return the common value of both numbers.

Let us consider a small example.

The input in this example is 15 and 9. In the first step, we subtract 9 from 15 and obtain the new pair of numbers $6 = 15 - 9$ and 9. In the second step, we obtain 6 and 3. In the third step, we obtain 3 and 3 and the algorithm returns the number 3.

The next example explains why we called the algorithm *SlowEuclid*. Consider the input $a = 1001$ and $b = 2$. The two numbers during the execution of the loop of the algorithm are

```

1001 and 2
999 and 2
997 and 2
995 and 2
...
(many rounds in-between)
3 and 2
1 and 2
1 and 1

```

The reason for the algorithm to take such a long time is the fact that the second number is excessively smaller than the first number of the input.

An Observation That Speeds up the Algorithm

In the example above, how often is 2 subtracted from 1001? One has $1001 = 2 \cdot 500 + 1$. Thus the number 2 is subtracted 500 times from 1001 until the value of the outcome drops below 2.

A computer can very efficiently perform a *division with remainder*. This operation computes for positive integers a and b two other integers q and r with $a = q \cdot b + r$. The integer r is at least zero and strictly smaller than b . In our example we have $a = 1001$, $b = 2$, $q = 500$ and $r = 1$.

If a and b is the input to our algorithm *SlowEuclid*, where a is larger than b , then b is repeatedly subtracted from a q times, if there is a remainder $r \geq 1$.

If b divides a exactly and $r = 0$, then b is subtracted from a $q - 1$ times and two bars of equal length are the outcome. This means that we can speed up the algorithm by immediately replacing a by the remainder r of this division. It then eventually happens that the remainder r is zero, in which case b is the greatest common divisor we are looking for and the algorithm terminates.

This is the idea of the next algorithm which we now call EUCLID.

EUCLID

```

1  if  $a < b$ : swap  $a$  and  $b$ .
2  while  $b > 0$ :
3      compute integers  $q, r$  with  $a = q \cdot b + r$ , where  $0 \leq r < b$ ;
4       $a := b$ ;  $b := r$ ;
5  return  $a$ .
```

Analysis

You probably guess that the algorithm EUCLID is much faster than SLOWEUCLID. Let us now rigorously analyze the number of iterations that the algorithm performs to substantiate this suspicion. Suppose that a is larger than b . How large then is the number r with which we replace a in the first step of the algorithm? The next picture reveals that this remainder is always at most $a/2$. This is because a is at least $b + r$ and since $b > r$ it follows that a is larger than $2 \cdot r$.

Thus in the first round, b is replaced by a number which is at most $a/2$. In the next round, a is replaced by a number which is at most $a/2$ too. Thus after two rounds, both numbers are at most $a/2$.

If we now consider $2 \cdot k$ consecutive rounds, then both numbers have a value that is at most $a/2^k$. If $k > \log_2 a$, then both numbers would have value zero. This, however, cannot happen since then the algorithm would already have finished before. Therefore the number of iterations through the loop of the algorithm is bounded from above by $2 \cdot \log_2 a$, where we again use the logarithm to the base 2.

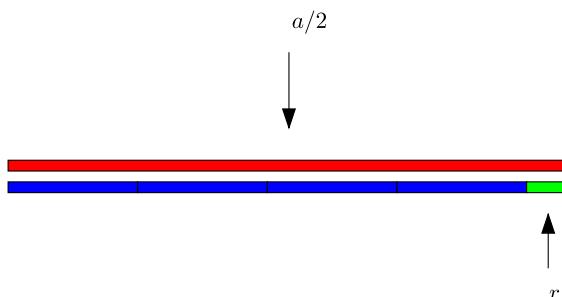


Fig. 12.4. The remainder r is small

The number of *digits* which is required in our decimal system to write down the number a is proportional to $\log_2 a$. This means that, while the algorithm SLOWEUCLID requires a number of iterations that is proportional to the *values* of a and b , the algorithm EUCLID requires only a number of iterations that is proportional to the *number of digits* that we need to write down a and b , respectively. This is an enormous difference in running time.

An Example

Finally we compute by hand the greatest common divisor of $a = 1324$ and $b = 145$.

The first division with remainder is $1324 = 9 \cdot 145 + 19$. Now a is set to 145 and b is set to 19.

The second division with remainder is $145 = 7 \cdot 19 + 12$. The third division with remainder yields $19 = 1 \cdot 12 + 7$. Then one has $12 = 7 + 5$, $7 = 5 + 2$, $5 = 2 \cdot 2 + 1$ and $2 = 2 \cdot 1 + 0$, from which we can conclude that the greatest common divisor of 1324 and 145 is 1. Two integers whose greatest common divisor is 1 are called *coprime*.

Further Reading

1. Donald E. Knuth: Arithmetic. Chapter 4 in *The Art of Computer Programming*, Vol. 2: *Seminumerical Algorithms*. Addison-Wesley, 3rd edition, 1998.

This classical textbook treats the Euclidean algorithm in Chap. 4.5.2. Among other things, the author explains that our analysis of the Euclidean algorithm is the best possible. More precisely it is shown that there exists a sequence F_0, F_1, F_2, \dots of natural numbers, for which the number of digits which are necessary to represent F_i is proportional to i while the Euclidean algorithm requires on input F_i and F_{i-1} exactly i iterations.

2. Joachim von zur Gathen, Jürgen Gerhard: *Modern Computer Algebra*. Cambridge University Press, 2nd edition, 2003.

This very nice textbook for advanced students of Computer Science and Mathematics discusses in Chap. 6 the Euclidean algorithm. The book also discusses the number of *elementary operations* (see Chap. 11) which are required by the Euclidean algorithm and some of its variants. Here, and also in the book of Knuth, the authors describe algorithms to compute the greatest common divisor of two integers that require a number of elementary operations which is proportional to $M(n) \log n$. The number n is then the total number of digits of the input and $M(n)$ denotes the number of elementary operations that are necessary to compute the product of two integers having at most n digits each.

3. In Wikipedia:

http://en.wikipedia.org/wiki/Euclidean_algorithm

Acknowledgement

The author is grateful to M. Dietzfelbinger for many helpful comments and suggestions.

The Sieve of Eratosthenes – How Fast Can We Compute a Prime Number Table?

Rolf H. Möhring and Martin Oellrich

Technische Universität Berlin, Berlin, Germany

Beuth Hochschule für Technik Berlin, Berlin, Germany

A **prime number**, or just **prime**, is a natural number that is not divisible without remainder by any other natural number but 1 and itself. Primes are scattered irregularly among the set of natural numbers. This fact has fascinated and occupied mathematicians throughout the centuries.

A **prime number table up to n** is a list of all primes between 1 and n . It begins as follows:

2 3 5 7 11 13 17 19 23 29 31 37 41 ...

Over time, many problems involving primes were found. Not all of them have been solved. Here are two examples.

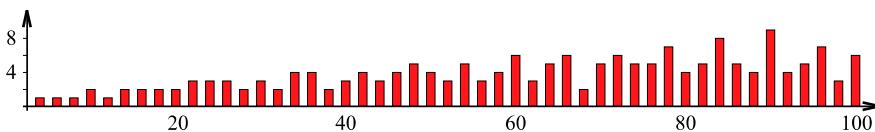
Christian Goldbach (1694–1764) formulated an interesting observation in 1742:

Every even number greater than or equal to 4 can be written as the sum of two primes.

For instance, we find:

$$4 = 2 + 2, \quad 6 = 3 + 3, \quad 8 = 3 + 5, \quad 10 = 3 + 7 = 5 + 5, \quad \text{etc.}$$

This proposition demands that there be at least one such representation. In fact, there are several for most numbers. The following diagram, based on a prime table, shows the number of different prime sums. On the x -axis, we see the partitioned (even) numbers.

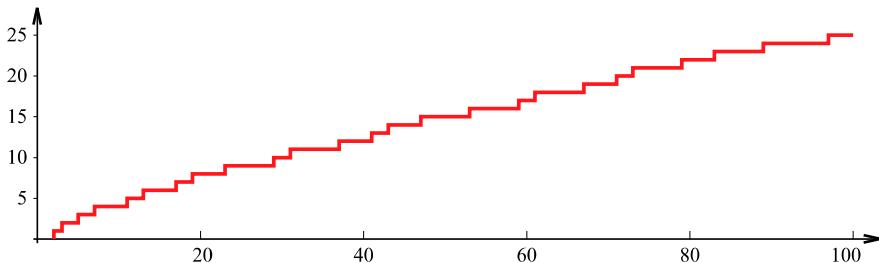


The slight trend upward in the columns continues with increasing n . No even number was found for which the proposition fails. Nevertheless, no proof is known that it holds *for all* of them.

Carl Friedrich Gauß (1777–1855) examined the distribution of the primes by counting them. He considered the function

$$\pi(n) := \text{number of primes between } 1 \text{ and } n.$$

A diagram of this function looks like this:



$\pi(n)$ is called a *step function*, for obvious reasons. Gauß constructed a “continuous” curve clinging as close as possible to $\pi(n)$, no matter how large n grows. In order to picture his plan and to check his results later, he needed a prime table. (This problem has since been solved. Deeper coverage exceeds the scope of this book.)

Today, primes are not only a challenge for mathematicians, but are of very practical value. For instance, 100-digit primes play a central role in electronic cryptography.

From the Idea to a Method

As far as we know today, an ancient Greek introduced the first algorithm for the computation of primes: **Eratosthenes of Kyrene** (276–194 BC). He was a high-ranking scholar in Alexandria and a director of its famous library, containing the complete knowledge of ancient mankind. He and others studied the essential astronomical, geographical, and mathematical questions of their time: What is the perimeter of the Earth? Where does the Nile come from? How can one construct a cube containing twice the volume of another given one? We will follow his steps below from a simple basic idea to a practical method. Even in his time, it could be executed well on papyrus or sand. We will also investigate how fast it is in computing a fairly large prime table. As a measure for “large,” we let $n = 10^9$, i.e., *one billion*.

A Simple Idea

According to the definition of prime numbers, for any m *not* a prime there are two natural numbers i, k with the property that

$$2 \leq i, k \leq m \quad \text{and} \quad i \cdot k = m.$$

We use this fact and formulate a very simple prime table algorithm:

- write down all numbers between 2 and n into a list,
- form all products $i \cdot k$, where i and k are numbers between 2 and n , and
- cross out all reoccurring results from the list.

We immediately see how this prescription does what we want: all numbers remaining in the list never occurred as a product. Consequently, they cannot be written as a product and are thus prime.

How Fast Is the Computation?

In order to analyze the algorithm, we write the individual steps of the basic idea more formally and enumerate the lines:

PRIME NUMBER TABLE (basic version)

```

1  procedure PRIME NUMBER TABLE
2  begin
3      write down all numbers between 2 and  $n$  into a list
4      for  $i := 2$  to  $n$  do
5          for  $k := 2$  to  $n$  do
6              remove the number  $i \cdot k$  from the list
7          endfor
8      endfor
9  end

```

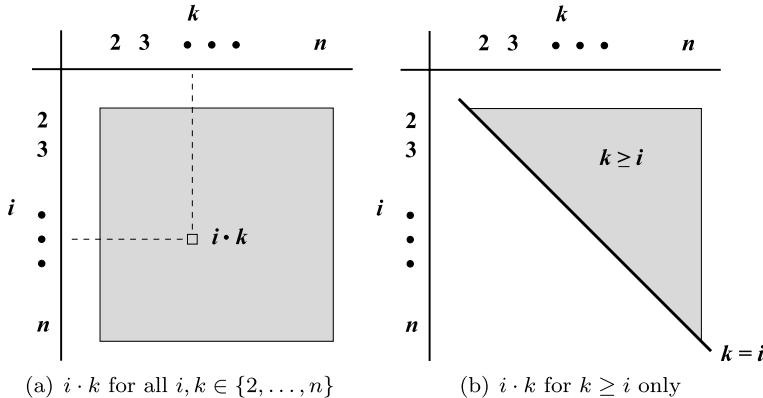
If the number $i \cdot k$ in step 6 is not present in the list, nothing happens.

This algorithm can be programmed in a straightforward fashion on a computer, and we can measure its time consumption. On a Linux PC (3.2 GHz), we get the following running times:

n	10^3	10^4	10^5	10^6
Time	0.00 s	0.20 s	19.4 s	1943.4 s

We clearly see how increasing n by a factor of 10 leads to a longer computation time by a factor of approx. 100. This was to be expected, since i as well as k run over a range about 10 times as large. The algorithm forms almost 100 times as many products $i \cdot k$.

From this, we can calculate the time needed for $n = 10^9$: we must multiply the time for $n = 10^6$ by a factor of $(10^9/10^6)^2 = 10^6$, resulting in $1943 \cdot 10^6$ seconds = *61 years and 7 months*. Clearly, this is of no practical use.

**Fig. 13.1.** Computing products $i \cdot k$ in a certain range

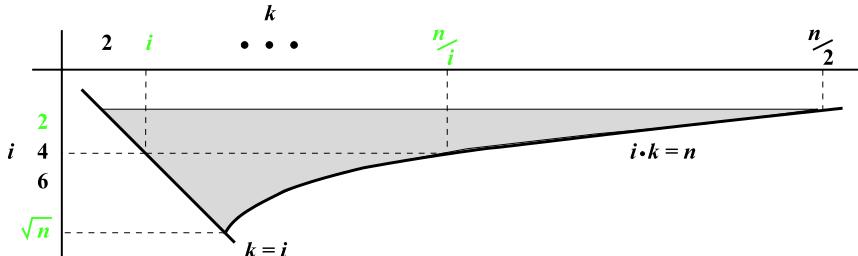
How Does the Algorithm Spend Its Time?

The algorithm generates all products in a certain range (Fig. 13.1(a)).

However, every individual result of $i \cdot k$ is needed only once. After being removed from the list, the algorithm would never have to generate it again. Where does it do surplus work? This happens, for instance, when i and k attain exchanged values, say, $i = 3$, $k = 5$ and later $i = 5$, $k = 3$. In both cases, the results of the product are identical, as is assured by the commutative rule of multiplication: $i \cdot k = k \cdot i$. For this reason, we restrict $k \geq i$ and avoid these duplications (Fig. 13.1(b)).

This idea instantly saves half the work! Yet, even 30 years and 10 months are still too long to wait for our table. Where can we save more? In those cases when in step 6 never anything happens: for $i \cdot k > n$. The list contains numbers up to n , so there is nothing to remove beyond that.

So we need to execute the k -loop (line 5) only for those values satisfying $i \cdot k \leq n$. This condition immediately delivers the applicable k -range: $k \leq n/i$. As a side effect, we can also limit the i -range. From the two restrictions $i \leq k \leq n/i$, we conclude $i^2 \leq n$, and ultimately, $i \leq \sqrt{n}$. For larger i , there are no k -values to enumerate. The number domain generated now looks as follows:



The algorithm has now attained the following form:

PRIME NUMBER TABLE (BETTER)

```

1  procedure PRIME NUMBER TABLE
2  begin
3      write down all numbers between 2 and  $n$  into a list
4      for  $i := 2$  to  $\lfloor \sqrt{n} \rfloor$  do
5          for  $k := i$  to  $\lfloor n/i \rfloor$  do
6              remove the number  $i \cdot k$  from the list
7      endfor
8  endfor
9  end

```

(The notation $\lfloor \cdot \rfloor$ means floor rounding, as i and k can attain integral values only.)

How fast have we become? The new running times:

n	10^4	10^5	10^6	10^7	10^8	10^9
Time	0.00 s	0.01 s	0.01 s	2.3 s	32.7 s	452.9 s

The effects are considerable, with our target of 10^9 within close sight: it is just seven and a half minutes away. We let the computer run and use this time to make some more improvements!

Do We Need Every i Value?

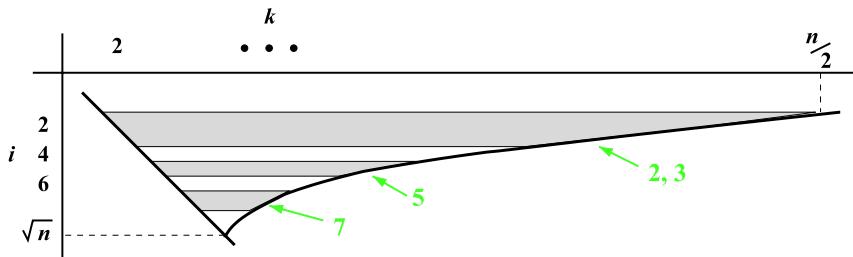
Let us consider what exactly happens within the i -loop (line 4): i remains fixed and k traverses its own loop (line 5). Doing so, the product $i \cdot k$ attains the values

$$i^2, \quad i(i+1), \quad i(i+2), \quad \dots$$

So when the k -loop is finished, no proper multiples of i are left in the list. The same applies to multiples of numbers less than i . They were removed earlier in the same way.

What happens if i is not a prime? Example $i = 4$: the product $i \cdot k$ attains the values $16, 20, 24, \dots$. These numbers are all multiples of 2, since 4 itself has this property. In principle, there is nothing to do in the case $i = 4$. The same is true for all other even numbers $i > 4$.

Example $i = 9$: the product attains only multiple values of 9. Yet, those have already been enumerated as multiples of 3 and are thus redundant. This reasoning applies to all non-primes, since they possess a smaller prime divisor that was an i -value before them. Consequently, we need to execute the k -loop exclusively for prime i -values. See the following figure:



Whether i is a prime or not could be looked up in the list itself – if it were complete. We can trust it to contain primes only no earlier than termination. Or is it?

The answer is: yes and no. In general *yes*; otherwise we could abbreviate the algorithm. Consider, for instance, $n = 100$: the non-prime 91 must eventually be removed from the list. It is generated close to termination, when $i = 7, k = 13$.

Yet in our special case, *no*. We do not attempt to recognize arbitrary numbers as prime, but just the specific value i . Also, not at an arbitrary moment, but exactly at the beginning of the k -loop for the i -value in question. In this restricted case, the list returns the correct primeness information on $i!$ Why?

We observed above that for every fixed i all removed values satisfy $i \cdot k \geq i^2$. Put differently, the range $2, \dots, i^2 - 1$ remains unaltered. Upon growing i -values, this range expands and comprises all previous ranges. In the following figure, these ranges are marked blue. The first “wrong” number in every table row is printed in red.

	$i^2 - 1 = 3$	$= 8$	$= 15$	$= 24$
$i = 2$	2 3 4 5 6 7 8	9 10 11 12 13 14 15	16 17 18 19 20 21 22 23 24	25 26 27
$i = 3$	2 3 5 7	9 11 13 15	17 19 21 23	25 27
$i = 4$	2 3 5 7	11 13	17 19	23
$i = 5$	2 3 5 7	11 13	17 19	23
• • •				25

Now all of these ranges do not change until termination. Therefore, they must be correct *before* execution of the k -loop for the i -value in question. We can say the table is completed in quadratic steps. The essential i -values – the ones whose primeness we need – are printed in green. It is obvious how they always lie within a blue range. So in order to decide whether i is a prime number or not, we may simply look it up in the current list.

In our algorithm, we can thus enhance the i -loop as follows:

PRIME NUMBER TABLE (Eratosthenes)

```

1  procedure PRIME NUMBER TABLE
2  begin
3      write down all numbers between 2 and  $n$  into a list
4      for  $i := 2$  to  $\lfloor \sqrt{n} \rfloor$  do
5          if  $i$  is present in the list then
6              for  $k := i$  to  $\lfloor n/i \rfloor$  do
7                  remove the number  $i \cdot k$  from the list
8              endfor
9          endif
10         endfor
11     end
```

It was this version of the method that the clever Greek presented. It is called the **Sieve of Eratosthenes**: *sieve* because it does not construct the desired objects, the primes, but filters out all non-primes.

Our time measurements on his algorithm read as follows:

n	10^6	10^7	10^8	10^9
Time	0.02 s	0.43 s	5.4 s	66.5 s

Even with $n = 10^9$, it needs roughly one minute!

Can We Get Even Faster?

With an argument similar to that for the prime i -values above, we can further restrict the k -values needed: we take only those found in the list! If k was removed as a non-prime, it possesses a prime divisor $p < k$. In the i -loop where $i = p$, all proper multiples of p were removed. In particular, k and its multiples were among them. Nothing remains to do.

Again enhancing the algorithm, it appears to be natural to do it as follows:

```

6      for  $k := i$  to  $\lfloor n/i \rfloor$  do
7          if  $k$  is present in the list then
8              remove the number  $i \cdot k$  from the list
9          endif
```

But caution! This formulation is misleading. Running the algorithm like that, we get the following list:

2 3 5 7 8 11 12 13 17 19 20 23 27 28 29 31 32 37 ...

What is wrong? Let us look at the first steps more closely. After initializing the list with all numbers up to n (line 3), it reads:

2 3 4 5 6 7 8 9 10 11 ...

First, we set $i = 2$ and then $k = 2$. The number 2 stands in the list, so we remove $i \cdot k = 4$:

2 3 – 5 6 7 8 9 10 11 ...

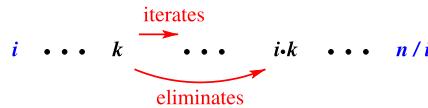
Next, we set $k = 3$. The number 3 also stands in the list and we remove $i \cdot k = 6$:

2 3 – 5 – 7 8 9 10 11 ...

Now something happens: $k = 4$ is no longer present in the list, since we removed it. According to the new **if**-condition, we must skip the k -loop and continue with $k = 5$:

2 3 – 5 – 7 8 9 – 11 ...

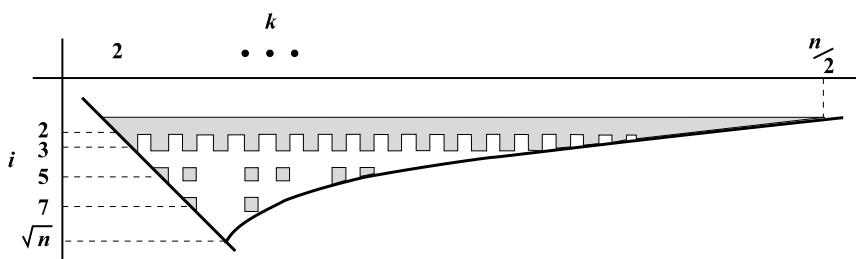
This way, $2 \cdot 4 = 8$ is erroneously never removed from the list. The problem is that k eliminates only numbers $i \cdot k > k$ and is subsequently incremented. Eventually, k attains the value of a former product $i \cdot k$ and the method unfavorably effects on itself:



The solution is to let k traverse its loop range *backwards*, thus avoiding the unwanted feedback:



According to this reasoning, only the following products $i \cdot k$ are formed:



Summarizing, we achieve the following version of the algorithm:

PRIME NUMBER TABLE (final version)

```

1  procedure PRIME NUMBER TABLE
2  begin
3      write down all numbers between 2 and  $n$  into a list
4      for  $i := 2$  to  $\lfloor \sqrt{n} \rfloor$  do
5          if  $i$  is present in the list then
6              for  $k := \lfloor n/i \rfloor$  to  $i$  step -1 do
7                  if  $k$  is present in the list then
8                      remove the number  $i \cdot k$  from the list
9                  endif
10             endfor
11         endif
12     endfor
13 end
```

Its running times:

n	10^6	10^7	10^8	10^9
Time	0.01 s	0.15 s	1.6 s	17.6 s

This result is very acceptable by today's standards. Starting out with a naive basic version, we have accelerated the method for $n = 10^9$ with a few closer looks by a **factor of 254.5 million!**

What Can We Learn from This Example?

1. Simple computation methods are not always *efficient*.
2. In order to accelerate them, we need to *understand well* how they work.
3. Often *several different improvements* are possible.
4. *Mathematical ideas* can lead very far!

Further Considerations

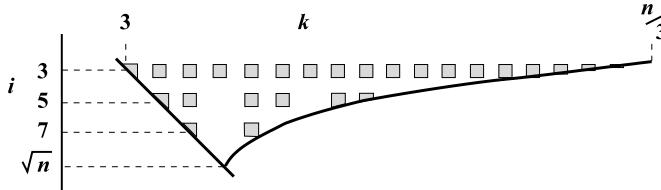
A time of 17.6 seconds is a good result for spending a few thoughts on the algorithm. Yet after all, how good is that? Have we reached the best possible?

Let us establish what the algorithm generally has to achieve. It must generate all non-primes up to n at least once and remove them from the list. Below $n = 10^9$ there are exactly 949,152,466 of them. Counting the number of products $i \cdot k$ computed, we obtain the following values for our variants above:

	Basic version	Better	Eratosthenes	Final version
Num. products	10^{18}	$9.44 \cdot 10^9$	$2.55 \cdot 10^9$	$9.49 \cdot 10^8$
Relation to nonprimes	$1.1 \cdot 10^9$	9.9	2.7	1.0

In fact, the final version performs just as much work as necessary. In this respect, it is optimal!

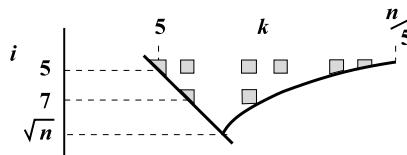
Interestingly, this does not mean we could no longer improve the algorithm. The comparison with the number of non-primes is no absolute measure, since we can still diminish them. The following trick works: the list is not initialized with *all* numbers from 2 on, but just with 2 itself and *all odd* numbers ≥ 3 . We know that the even numbers ≥ 4 are never prime, so why generate them in the first place? The procedure has considerably less work to do on a list containing odd prime candidates only. The k -loop for $i = 2$, the longest one of all, is completely omitted. Only the following products are being generated now:



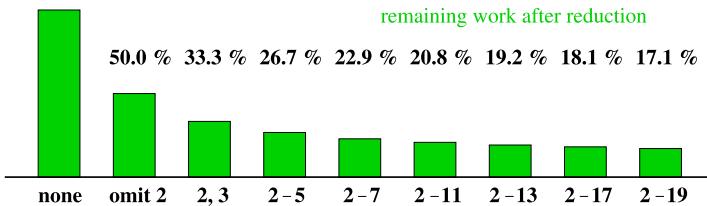
We can play more on this theme: we also omit initializing the list with all proper multiples of 3. At the start, it contains the numbers

2 3 5 7 11 13 17 19 23 25 29 31 35 37 41 43 47 49 ...

and the actual sieve work begins with $i = 5$:



This way, we can achieve ever-decreasing initial lists for the same range up to n by omitting the proper multiples of 5, 7, 11, 13, etc.:



We find the same characteristic in the running times and the memory consumptions, as the lists themselves are diminished:

Reduction	None	Omit							
		2	2, 3	2-5	2-7	2-11	2-13	2-17	2-19
Run time [s]	17.6	33.0	22.6	17.8	14.7	13.3	12.6	24.0	25.9
Memory [MByte]	119.2	59.6	39.7	31.8	27.3	24.8	22.9	21.6	20.4

The list is represented here by a bit array. At position i in the array, a 1 marks the primeness of i while a 0 indicates that i is a nonprime.

If we choose a linked list as the memory representation instead, we would incur two disadvantages. First, in order to look up the primeness property of some number, we would have to search for the appropriate entry first. Second, the numbers run up to 9 digits and we would need to store all 50,847,534 prime numbers below one billion in 1551.7 MByte memory.

A note on the almost double computation time after omitting the even initial numbers. Since the list is condensed, we need a transformation between the list indices ($1, 2, 3, 4, \dots$) and the numbers addressed (in this example: $2, 3, 5, 7, 9, \dots$). This uses up some time. Yet altogether, we achieve an excellent 12.6 s with a list reduced by the multiples of 2 through 13. Beyond that, the preparation of the transformation data dominates over the actual list computation, rendering further reduction useless.

Further Reading

1. http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

This Wikipedia article offers a concentrated introduction to the topic.

2. On the homepage of one author, we provide the C code with which we measured the running times in the tables:

<http://prof.beuth-hochschule.de/oellrich/aktivitaeten-mit-schuelern.html>

3. Chapters 14 (One-Way Functions) and 16 (Public-Key Cryptography)
In Chaps. 14 and 16, the primary topic is not prime numbers. But the treated problems essentially reduce to finding divisors of very large natural

numbers fast. Since primes possess no proper divisors, none can be found fast and thus the factorization problem cannot be broken down. Large primes are hard to recognize and therefore qualify as building bricks for encryption methods. However, primes with 100 or more digits cannot be practically handled by means of prime tables. For this purpose, other methods generate numbers of this magnitude directly.

4. Chapter 20 (Hashing)

In Chap. 20, the main topic is also not prime numbers. Yet for most purposes, hash tables of prime length are advantageous. Hash functions of the form *key value modulo table length* generally have good distribution properties when the keys are uniformly distributed. A certain lookup method called *double hashing* can in this way assert to find any free entry. For the purpose of selecting a suitable prime for a hash table, a prime table is very useful.

5. A *twin prime* is a pair of prime numbers with a difference of exactly two. The first such pairs are: (3, 5), (5, 7), (11, 13), (17, 19), (29, 31), (41, 43), (59, 61), (71, 73), The very first one (3, 5) is an exception for two reasons. First, the number 5 is the only one to occur in two pairs. Second, in all other twins, the number between the primes is divisible by 6. This must be the case, since among three contiguous numbers exactly one must be divisible by 3 and at least one by 2. The two primes (except with (3, 5)) are neither divisible by 2 nor by 3, therefore the middle number contains both factors. Twin primes have been found in arbitrary large ranges of natural numbers. However, there is still no proof whether finitely or infinitely many of them exist. In a prime table up to 10^9 , we can find 3,424,506 such twins.

One-Way Functions. Mind the Trap – Escape Only for the Initiated

Rüdiger Reischuk and Markus Hinkelmann

Universität zu Lübeck, Lübeck, Germany

The contributions so far have shown how a specific algorithmic problem can be solved fast – slightly more generally, computer scientists use the term *efficiently*. If we cannot find a fast algorithm one might be tempted to feel unhappy. Here we show that problems that are not efficiently solvable can still be very valuable. Thus, we like to phrase the message of this article as

bad news can turn out to be quite useful.

The Mirror Image of Multiplication: Factorization

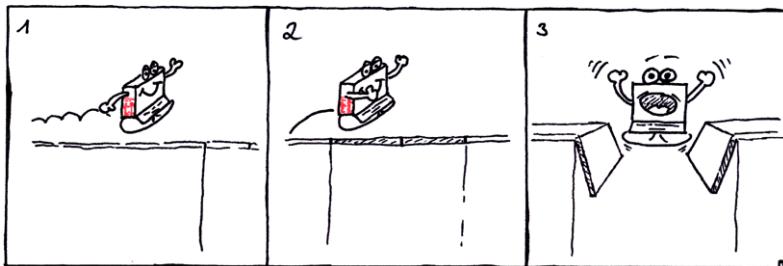
In Chap. 11 it is shown that the product of numbers can be computed very fast, even when the numbers are quite big and therefore require many decimal or binary digits to write them down. A simple algorithm for multiplying two numbers is taught at elementary school. This way, a human being can compute the product of two larger numbers on a piece of paper in a few minutes, even if it might be boring and require concentration to avoid errors. For computers it is one of the easiest tasks to multiply numbers with hundreds or thousands of digits in a few milliseconds.

Let us consider the inverse problem: to split a product back into its factors. Remember that certain numbers, called primes, cannot be divided further into a product of smaller numbers, and that the first prime numbers are $2, 3, 5, 7, 11, 13, 17, 19, 23, \dots$. Mathematicians have shown that every natural number uniquely can be expressed as a product of primes, for example,

$$20,518,260 = 2 \cdot 2 \cdot 3 \cdot 5 \cdot 7 \cdot 7 \cdot 997.$$

Naturally, the question comes up to design an algorithm that finds this sequence of divisors, the *factorization problem*. Can we do this as efficiently as multiplication, even for large numbers?

The last digit of a decimal number N tells us whether it can be divided by 2 or 5. Divisibility by 3 can be decided by adding the digits of N . For numbers

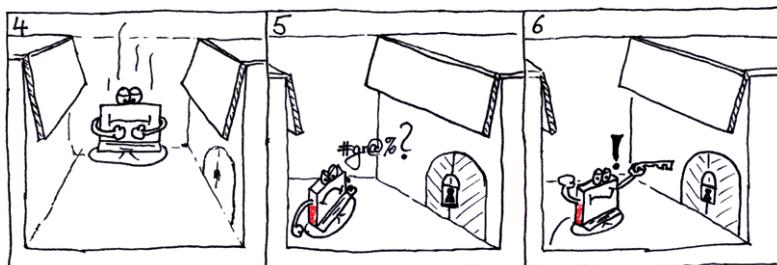
**Fig. 14.1.** A one-way function

that have only large prime divisors, however, finding the divisors does not seem to be that easy. The ancient Greeks knew a method: test N with all primes p that are at most as large as the square root of N . However, for a 100-digit number this requires about $8.5 \cdot 10^{48}$ division tests, which is incredibly many and would last longer than the expected lifetime of our universe.

A simpler variant of the factorization problem is the **prime number problem**, where one has to find out whether a given number is already prime and thus has no smaller divisors. Efficient algorithms are known for the prime number problem, so this will not be the topic here – see Chap. 13.

In the 1970s three computer scientists, Ron Rivest, Adi Shamir and Leonard Adleman, invented the **RSA cryptosystem**. It encrypts texts to be sent over an insecure communication medium like the Internet such that a third person is not able to find out the contents of the message. Nowadays, the RSA cryptosystem is used worldwide as a basic tool for secure Web transactions. It selects a number N that is the product of two large primes p and q , but keeps these two factors secret. Breaking this encryption scheme is closely related to finding out the factors p, q of N .

In order to demonstrate how secure their system is, in 1977 the inventors encrypted a message with a 129-digit decimal number N . They then made N and the encrypted message public, challenging everybody to decrypt the message, where

**Fig. 14.2.** How can we get back?

$N =$

114381.625757.888867.669235.779976.146612.010218.295721.242362.562561.842935.

706935.235733.897830.597123.563958.705058.989075.147599.290026.879543.541

It took 17 years until in 1994 very complex algorithms had been developed that were more clever than simply testing all possible prime divisors. After eight months of intensive computing on a worldwide cluster of hundreds of computers, finally the two factors

$N = 3490.529510.847650.949147.849619.903898.133417.764638.493387.843990.820577$

$\times 32769.132993.266709.549961.998190.834461.413177.642967.992942.539798.288533$

were found. In total, about 160 trillion computer instructions were executed, that is $1.6 \cdot 10^{17}$. Are you interested in knowing the encrypted message? “*The magic words are squeamish ossifrage.*”

One-Way Functions

Today, 16 years after this event:

- *The bad news:* even with the best algorithms known and the fastest supercomputers available now or in the near future, and using many thousands of such machines together, for a decimal number with several hundred digits that has only large prime factors one cannot find these factors in acceptable time (for example, within 100 years).
- *The good news:* as long as no new factoring algorithms are found that are dramatically faster, data encrypted with the help of the RSA-scheme – for example, passwords during online banking – can be considered absolutely secure.

Let us summarize so far:

1. The multiplication of numbers, in particular prime numbers, can be done quite efficiently.

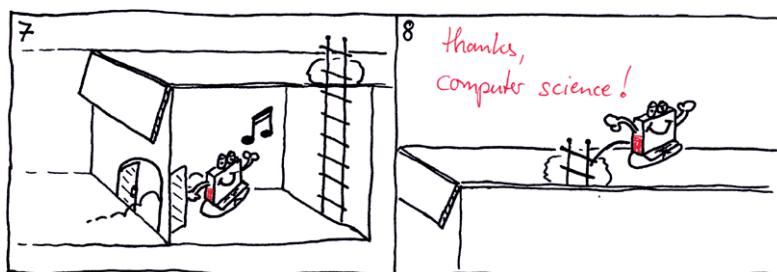


Fig. 14.3. Return through a secret door

2. Its inverse operation, splitting a product back into its prime divisors, cannot be done efficiently – at least according to the current state of knowledge in informatics and mathematics.

How can we make use of such a situation in general? In mathematics, the term function denotes an operation that transforms mathematical objects into other objects. In our case, the objects are numbers or sequences of numbers. Factorization can be considered as the inverse function to multiplication.

An operation that can be computed easily, but for which the inverse is difficult is called a **one-way function**. Such functions are important for the encryption of messages. Instead of sending the original message M one could send the result of applying the one-way function to M . The encryption can be done fast, whereas the decryption, that is, applying the inverse function, is very difficult according to the properties of one-way functions. An attacker thus has no chance to deduce M from its encryption.

Consider the following example. *Alice* wants to send to *Bob* the message “top secret”. Let us assume that they use only capital letters and instead of a space between two words write the letter “X”. The letters are then numbered with 01 up to 26 and each letter is replaced by its number. Then

TOPXSECRET

turns into

$$T = 201516240503180520.$$

In general, T will not be prime; however, by adding a few digits at the right end one can always make this number prime – in our case, for example, using the digits 13 we get the prime number

$$p = 20151624050318052013.$$

Then *Alice* chooses a second slightly larger prime number, for example,

$$q = 567891624050318052137$$

and computes their product

$$N = p \cdot q = 11443938509186566743788165964622411801781.$$

This number N can be sent by *Alice* to *Bob* without any worry – nobody will be able to decipher the message p , resp. T since this would mean that the factorization of N has been found. To be honest, these numbers are too small to guarantee high security – in reality one should choose prime numbers with at least 150 digits, thus *Alice* simply adds more digits to T to generate a p that is long enough.

But, wait, even *Bob* cannot decrypt the message. Grrrrr ... encryption with one-way function is not totally simple. We need an additional trick.

A **one-way function f with secret information**, in technical terms a **trapdoor function**, has the following additional property:

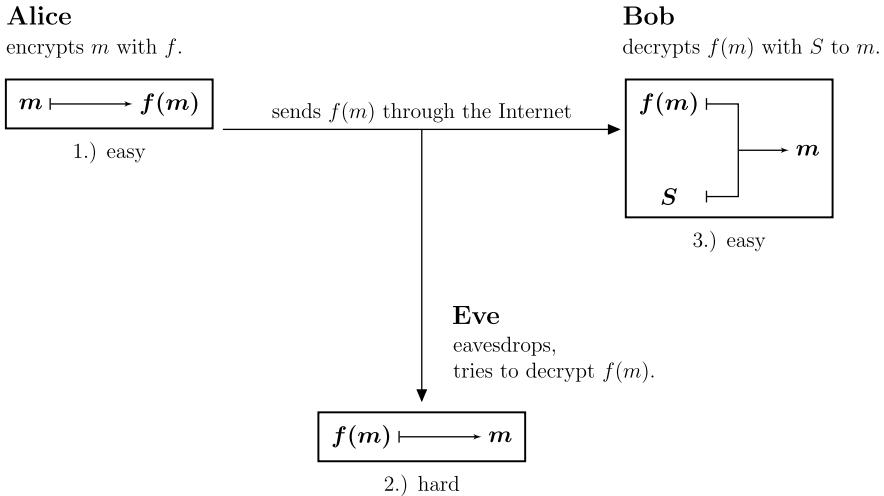


Fig. 14.4. Using a trapdoor function in cryptology

3. The inverse function of f can be computed efficiently if one possesses a secret key S .

This requirement may sound strange at first thought; however, we can illustrate it with a simple example from our daily life.

A Practical Problem: Searching a Telephone Book

Let's take a step back into history when telephone numbers were not available online in electronic databases, but had to be looked up in printed telephone books. If *Alice* wants to call her old friend *Bob* after a long time, but does not have his current number, she only has to search for his name in the telephone book and then find the number behind *Bob's* name. This is simple and fast – why?

Remember Chap. 1 on binary search. The telephone book for the German city Lübeck, for example, has about $n = 250,000$ entries, which means that with at most $18 \approx \log_2 n$ many comparisons of names one can find *Bob* since the names are listed in alphabetical order. If they were not ordered one would have to read the telephone book from the beginning and compare every name until *Bob* is found. On average, this will require about half the book size since *Bob* could be anywhere. For a midsize city like Lübeck this would be completely impractical (instead of transmitting her invitation for a dinner in the evening to *Bob* over the phone, *Alice* would still be searching for his number the next morning). Although searching in telephone books rarely is done by a perfect binary search – even by computer scientists – everybody knows from experience: searching for telephone numbers is easy because of the alphabetical ordering.

Adams	3 67 890	Lincoln	2 35 520
Baker	6 00 712	Mann	6 54 167
Brown	1 42 361	Newman	7 23 104
Cook	4 77 288	Phillipps	1 52 731
Cox	2 76 201	Robinson	8 87 236
Cruz	3 51 682	Simpson	7 36 917
Davis	7 19 763	Smith	9 67 171
Derrick	7 28 987	Spencer	5 24 605
Dobb	2 35 680	Stevenson	4 86 993
Edwards	7 56 194	Thompson	3 69 237
Emmett	5 37 165	Turing	7 48 828
Fairburn	3 10 673	Wainwright	4 82 729
Grant	2 28 469	Wilkens	2 78 831
Hawkins	1 23 456	Zuse	6 57 827
Knight	3 59 572		
Lawrence	3 88 636		

Fig. 14.5. Normal telephone book

What about searching the other way, for a given number find the owner of this number?

Alice finds on the display of her telephone that somebody with the number 123456 has tried to call her while she was away. She cannot remember this number, thus would not like to call back right away, but first find out whom to expect at the other end.

Having available only a printed telephone book she has no other choice than scanning the book entry by entry until she finds the number. Thus the mapping *name* → *number* is a one-way function, at least for human beings only having access to printed telephone books.

Computers, however, can sort a sequence of data pairs with clever algorithms very fast (as explained in previous articles) such that nowadays information providers also offer telephone books ordered by numbers. With the help of such an *inverted telephone book* one can now search for names given the numbers very fast, too, again exploiting binary search. Thus, in our modern times one could also solve this problem efficiently. But for a moment let us stick to the situation where only standard telephone books are available.

A nice telephone company could solve *Alice's* problem by assigning numbers to customers according to their alphabetical ordering. For example, *Alice* gets a number m_A that is smaller than *Bob's* number m_B since her name precedes *Bob* in alphabetical order. In such a system we can find names in a few seconds if we perform binary search with respect to the entries of numbers.

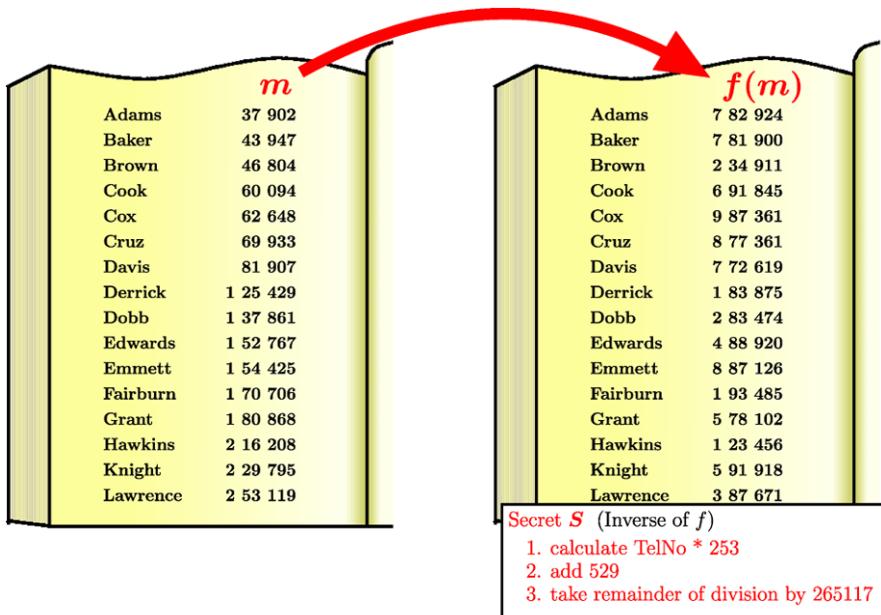


Fig. 14.6. Telephone book ordered by numbers and book with numbers transformed

A telephone book for an ordered number system thus has lost its one-way property. *Bob*, who is concerned about protection personal information, does not like such a system. He demands to keep the “practical” anonymity of a caller even if his number is displayed, and thus requests to stick to a chaotic distribution of telephone numbers. Therefore, he asks the international society of cryptologists (ISCRY) for help. The ISCRY solves the problem by assigning new numbers to all people. The new numbers are obtained by a secret transformation h of their numbers in the ordered system such that the originally ordered sequence of numbers

$$m_A \quad m_{\text{Andreas}} \quad m_{\text{Axel}} \quad \dots \quad \text{of } Alice, Andreas, Axel, \dots$$

after transformation to

$$h(m_A) \quad h(m_{\text{Andreas}}) \quad h(m_{\text{Axel}}) \quad \dots \quad \text{looks completely chaotic.}$$

ISCRY, however, knowing the transformation, can recompute the original number m from $h(m)$ and perform a binary search on these recomputed numbers.

The mapping $\text{name} \rightarrow \text{number}$ in the transformed telephone book is therefore a one-way function with a secret key, the transformation h . Everybody can easily find the number for each name. The inverse search, however, can only be done efficiently by those who know the secret transformation h .

Table 14.1. Numbers are estimated and rounded

Subject	Computation time / Numbers
Using the fastest algorithms known today and all computers on earth for:	
Factoring a 256-digit decimal number	More than 2 months
Factoring a 512-digit decimal number	More than 10 million years
Factoring a 1024-digit decimal number	More than 10^{18} years
Lifetime of our universe	$\approx 10^{11}$ years
Cycles of a 5-GHz processor in 1 year	$\approx 1.6 \cdot 10^{17}$
Number of electrons in our universe	$\approx 8 \cdot 10^{77}$
Number of 100-digit primes	$\approx 1.8 \cdot 10^{97}$

Security and Googles

Can one make sure for a specific problem that there is no efficient algorithm to solve it? Intuitively, it seems more difficult to prove the nonexistence of an object than its existence since in the latter case one only has to present it.

This is surely true when searching for fast algorithms since there are potentially infinitely many (compare the discussion in Chap. 24).

To remove any last doubts concerning the security of modern cryptosystems, one has to prove that there do not exist efficient algorithms for the inverse of an encryption function. More than 30 years' research in informatics has tried to develop methods for analytical proofs that certain problems cannot be solved efficiently. Although some milestones have been met, there still seems to be a long way to go to reach this goal.

Finally, we would like to give the reader some feelings for big numbers. Table 14.1 contains estimations for the time to factor a number that is composed of two big primes. For comparison, some physical constants are added.

Further Reading

1. Chapter 16 (Public-Key Cryptography).
Chapter 16 deals with public-key cryptosystems for which one-way functions are an essential precondition.
2. More details about the RSA system can be found at:
<http://www.rsa.com/rsalabs/node.asp?id=2214>,
 for more information to break RSA see:
<http://www.wired.com/wired/archive/4.03/crackers.html>
3. If you like to read more about cryptography and its history we recommend the following books:
 - B. Schneier, *Applied Cryptography*. Wiley, 1996. <http://www.schneier.com/book-applied.html>

- A. Menezes, P. Oorschot, S. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 2001. <http://www.cacr.math.uwaterloo.ca/hac/>
 - D. Kahn, *The Codebreakers – The Story of Secret Writing*. Scribner, 2nd edition, 1996.
 - F. Bauer, *Decrypted Secrets. Methods and Maxims of Cryptology*. Springer, 3rd edition, 2002.
4. Interesting facts about prime numbers can be found at C. Caldwell, The Prime Pages: <http://primes.utm.edu/>

The One-Time Pad Algorithm – The Simplest and Most Secure Way to Keep Secrets

Till Tantau

Universität zu Lübeck, Lübeck, Germany

School started just a few weeks ago and Max is already dreading the upcoming exam in computer science, which will be about something with the slightly menacing name “encryption algorithms.” Max’s dread is not unfounded since he has almost no clue as to what “encryption algorithms” might be about, which in turn is due to the fact that all of Max’s attention has lately been focused on Lisa, his new girlfriend. Lisa, by comparison, is not only fascinated by Max, but also by cryptology. Since Max simply *has* to pass the exam, Lisa devises a rather straightforward plan: “Fortunately, the exam will consist only of multiple choice questions, so all I need to do is to pass to you slips of paper on which I will write the answers to the questions. The teacher never notices such things.” Max raises a slight objection: “But Peter will be sitting between us.” Lisa answers: “Ah, don’t worry about Peter, I can sweet-talk him into passing on anything I give him. So, on the paper, I will put a 1 for each answer box that you should check and I will put a 0 for the other answer boxes. For instance, if there are five answer boxes and you should check the first and the third, I will send:



Lisa’s plan works remarkably well: Lisa and Max (and also Peter by the way) get excellent grades. However, the teacher becomes somewhat suspicious since Max is not renowned for his great knowledge concerning the subject. She decides that during the next exam Max’s former girlfriend, rather than Peter, will sit between Lisa and Max. The teacher’s idea seems to go in the right direction since Max starts brooding: “I’d rather fail the exam than have *her* copy the answers!” (Max obviously is not all that fond of his former girlfriend.)

Lisa gives the matter some thought and then informs Max: “Ok, it seems like we will have to encrypt the solutions using a one-time pad.”

“One-time pad?” Max asks, looking slightly clueless.

“That’s an encryption method for securely encrypting messages a single time.”

“Encryption method?” Max asks, looking even more clueless.

“You really don’t pay attention in class . . .” Lisa begins. Max interrupts her by interjecting “But I love you!”, which Lisa ignores and continues: “Encryption means that you and I agree on a *key* beforehand. I can use this key to lock the answers to the exercises. You can then use the key to unlock the answers once more. Your former girlfriend won’t be able to find out the correct solutions without the key.”

“Eh?” is Max’s only answer, being totally lost at this point.

Encrypting Messages

“Pass me five coins from your wallet,” Lisa asks. Max obliges, although he seems to mumble something along the lines of “But I will get them back . . .” Lisa places the coins on the table and continues: “When a coin face displays a number, you must check the corresponding box, otherwise you don’t. For instance, if the first and third box should be checked, we can represent this using coins as follows:”



“Very well. Whatever. I don’t really see where you are going with this,” Max answers, slightly bored. “It does not matter whether you write 10100 on a piece of paper or use five coins, my ex-girlfriend is not *that* stupid. She will catch on that this means: Check the first and third boxes.”

“Ah, but now encryption comes into play. Pass me that deck of notes, thanks. On some of them I’m going to write ‘flip’ and on some others I’m going to write ‘do not flip’. Now you pick five of them randomly and place them below the coins.”

Max does as Lisa asks:



“Great,” Lisa says encouragingly. “Those notes will be our *key*, which we fix before the exam and have to learn by heart.”

“Now I do see where you are going with this.” Max replies, not being stupid after all. “Instead of the original coins you pass the coins after they have been flipped or not flipped according to our key:

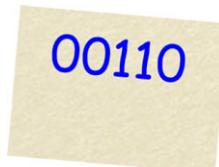


My ex-girlfriend can look at this as long as she wants, it won’t help her one bit – for instance, the first coin can both mean ‘check’ or ‘do not check’ the first box. The chances are exactly fifty-fifty.

However, come to think of it, why should she pass notes at all? Why shouldn’t she just swallow them to spite me?”

“She also wants you to pass the exam – so that she can pester you next year in class.”

“Arghh,” Max cringes, then gives a big sigh. “Well, let’s go over the plan once more: In the exam you solve the first question. Suppose you want to tell me that I should check the first and third box, which corresponds to 10100. However, since our key says that the first and fourth ‘coin’ should be flipped, you send the following instead:



“I have memorized the key and can ‘flip the coins’ once more, which yields 10100. Thus, I check the first and third box.”

“I knew my boyfriend was reasonably smart.” Max looks suspiciously at Lisa, but she continues: “You will have to agree that this one-time pad method is pretty simple, but still perfectly safe. By the way, we are not the only ones who use this method: When heads of state want to send each other messages and they really want to make sure that these messages are kept secret, they also use a one-time pad.”

Max bursts out laughing. “You really believe that when the American president wishes to send a message to the German chancellor, he is going to start flipping coins and write zeros and ones on a postcard?!”

“Obviously not,” Lisa replies grumpily, “a computer will do that for them. For this, one needs to write down the algorithm, which would be a rather good exercise for you.”

The Algorithm

“Let’s see,” Max starts, still smirking. “First of all, I think that, indeed, there really is only one algorithm because both ‘encoding’ or ‘locking’ our messages and ‘decoding’ or ‘unlocking’ them work in the same way: In both cases we start with an array of zeros and ones and a key, and in both cases we end up with an array of zeros and ones once more. All the algorithm needs to do is to replace zeros by ones and vice versa, whenever the key says ‘flip’.”

The algorithm ONE TIME PAD encrypts and decrypts the array A with n entries using the *key*.

```

1  procedure ONETIMEPAD ( $A$ , key)
2  begin
3      for  $i := 1$  to  $n$  do
4          if key[ $i$ ] = “flip” then
5              if  $A[i] = 0$  then
6                   $A[i] := 1$ 
7              else
8                   $A[i] := 0$ 
9          endfor
10     end
```

“Exactly” Lisa replies. “That’s one way to do this. However, the key normally is not an array of strings like ‘turn’ or ‘do not turn’, but rather also an array of zeros and ones. A one in the key array means ‘turn’, while a zero means ‘do not turn’. The algorithm can then be rewritten rather succinctly as follows:”

Short version of ONE TIME PAD.

```

1  procedure ONETIMEPAD ( $A$ , key)
2  begin
3      for  $i := 1$  to  $n$  do
4           $A[i] := A[i] \text{ xor } \text{key}[i]$ 
5      endfor
6  end
```

“Ah, just a moment,” Max interrupts. “I think I remember, rather vaguely I might add, that ‘xor’ stands for ‘eXclusive OR’. But I can’t really recall what that is supposed to mean.”

“The ‘exclusive or’ tests whether exactly one of two numbers is a 1. If this is the case, the answer is 1, otherwise 0. So, in order for the exclusive or to be 1, either $A[i]$ must be 1 or *key*[i] must be 1, but not both. These two conditions exclude each other, hence the name. Have a look at this table, which shows what’s going on.”

Table showing the values of $A[i]$ xor $key[i]$.

	$key[i] = 0$	$key[i] = 1$
$A[i] = 0$	0	1
$A[i] = 1$	1	0

“I see. This ‘xor’ is exactly what we need since it is going to ‘flip’ the value of $A[i]$ when $key[i] = 1$, and it’s going to leave $A[i]$ as it is when $key[i] = 0$.”

Breaking the Code

Max is rather pleased with the algorithm. “I like this method. All I need to do is to memorize a key consisting of five little notes and we can then use it for the whole exam. That’s much easier than actually studying cryptology.”

“Unfortunately, my sweet Max, things aren’t that simple. Suppose we are going to use the same key for all of the twenty questions of the exam, each having five check boxes. Suppose your former girlfriend succeeds in solving just one question by herself and gets my encoded message. For instance, I find out that the last three boxes must be checked for this question. Using coins, this would be represented like this:



“This will also be known to her if she can answer the question by herself. Now she gets a note from me with the encoded messages 01010 or, written in coins:



“I think you can see the problem?”

“Indeed, she can deduce the key! Obviously, the first coin has not been flipped, while the second has been, and so on. She will know that our key must be:



"Once she has got the key, she can solve all the other questions! That's a disaster!"

"That's why the method is called *one-time* pad method. *The key can only be used once.* If you do not follow this rule, it is pretty easy to find out the key that has been used. One of the older methods for encrypting wireless communication used the same key over and over again – and so the keys could be deduced pretty quickly. So, if you were using a laptop in a coffeehouse to write emails, students sitting at the other tables could easily read all the emails you send me." At this point, Max's facial color undergoes a rapid succession of changes. First, his face is completely drained of all color, only to turn bright red seconds later. Lisa gives him a sweet smile: "Those highly paid people who came up with the method obviously did not pay attention in school. We will have to memorize a new key for each question."

"But that's horrible, I'll have to memorize the correct sequence of 100 times 'flip' or 'do not flip'. It would be much easier to just study the material on cryptology instead!"

"Hmm. Perhaps that's the best solution anyway. Cryptology is not that difficult, after all."

Further Reading

1. Chapter 16 (Public-Key Cryptography)

The method presented in this chapter allows one to reuse keys, unlike the one-time pad. Even more impressive is the fact that the keys can even be made public!

2. Chapter 17 (How to Share a Secret)

Some secrets, like the keys in cryptology or the position of a pirate's chest on an island, are too important to just store them in one place. This chapter describes a method for splitting up a secret into parts so that you really need to get together all parts in order to retrieve the secret.

3. Chapter 25 (Random Numbers)

Lisa and Max created their key by randomly drawing notes from a deck. However, how does a computer generate random keys? This is a surprisingly difficult question since, normally, there is nothing random about the way a computer works.

4. <http://en.wikipedia.org/wiki/Cryptography>

This star-reviewed Wikipedia article is a nice introduction to cryptology in general.

5. <http://en.wikipedia.org/wiki/One-Time-Pad>

This Wikipedia article describes the one-time pad algorithm and some variants in more detail. It includes some background on the history of the algorithm.

Public-Key Cryptography

Dirk Bongartz and Walter Unger

Gymnasium St. Wolfhelm, Schwalmthal, Germany
RWTH Aachen University, Aachen, Germany

Who has never thought about sending a secret message? Even Julius Caesar did so. He allegedly moved each single character in his message three positions to the right within the alphabet. Using this procedure, A becomes a D, B becomes an E, and so on. Finally, W is replaced by Z, X by A, Y by B, and Z by C.

If one knows this procedure it is certainly quite easy to decrypt an intercepted “secret message.” For instance, what is behind the following message?



Caesar's cipher:

ZH ORYH FUBSWRJUDSKB

Which message has been encrypted here?

To make this idea a little bit more general, one obviously can choose a number k ($k < 26$) and move each character of the message to be encrypted (usually called *plaintext*) k positions to the right. In this way we obtain a secret message, the so-called *ciphertext*. The *encryption method* is moving the characters to the right within the alphabet, starting again from the beginning when the end is reached. In this context k is called the (*secret*) *key*. To decrypt the message, one simply has to move each character in the ciphertext k positions to the left.

Once anybody knows the key in such a system, he is able to encrypt as well as to decrypt all messages. For this reason these procedures are called *symmetric cryptosystems*. Also, the *one-time pad* method, presented in Chap. 15, belongs to this category.

Summing up, this means that everybody who could encrypt a message could also decrypt it and other messages encrypted by the same method.

On the other hand, there is the major drawback that the regular receiver of the ciphertext must know not only the used method but also the used key. But how do we exchange this key if communication partners are really far away from each other?

In the next section we show that having the same key for encryption and decryption is not essential. It is even advantageous if keys are not equivalent. Methods based on two kinds of keys, where one is public and the other one is private, are called *asymmetric cryptosystems*.

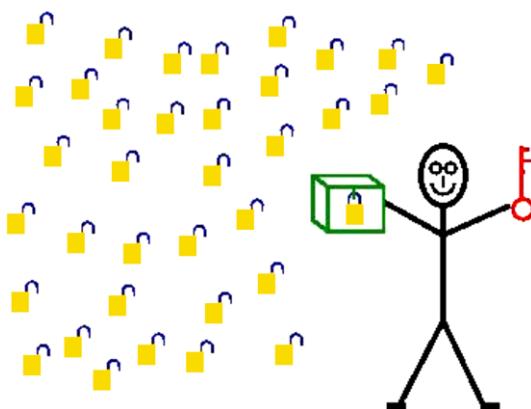
Public Keys

At first sight, this title might look senseless and the question might come up how this could work out. For any symmetric system a publicly known key would reveal all secrets. The problem for the symmetric systems is that it is possible to compute the decryption key from the encryption key.

However, looking at it again we see that we may use a public key for encryption, and keep a (different) key for decryption secret. We only have to make sure that the decryption key cannot be computed from the public encryption key.

Such a system would be attractive because anyone could send a secret message to you and only you would be able to decrypt the message.

In fact, it is not so complicated to construct such a system. Just assume you have several thousand padlocks which may be opened by a single key. (Note that this is an unusual requirement, as normally only single padlocks with several keys are on offer.) Now, you distribute your padlocks to all your friends. You may even place a pile of padlocks at the school office or library. The key to open all these padlocks stays in your possession.



Now, anyone who wants to send a message to you may take a box and place the message in it. Because you need no key to lock one of the padlocks,

everyone is able to lock away a message in a box. Furthermore, only you are able to read the messages because you hold the only key opening the boxes. Thus, your friends may even ask the biggest blabbermouth of the school to forward the box to you.

This example shows that such a system is possible. However, there is one drawback. This example requires a big effort. We have to purchase these padlocks and we have to distribute them. This would be quite expensive and labor-intensive.

It would be nice to have a system which provides such a public-key system without padlocks. This is possible in principle, using the one-way functions from Chap. 14. Loosely speaking, it is easy to compute the one-way function in the forward, standard direction, but hard to do the reverse: given a function value, recover the corresponding argument.

In our case we need some system in which encryption is easy (so that everyone is able to send us a message) but decryption is hard. However, you as the legal receiver should be able to decrypt easily. You should keep some backdoor for this. One possibility would be to use the inverse telephone book from Chap. 14. However, here we describe a different idea.

A Limited Algebra

Real public-key cryptosystems which employ computers to encrypt and decrypt messages use advanced algebra. We do not describe this kind of arithmetic here. Instead, we illustrate the principles by using a limited algebra.

Within this limited algebra we only use addition, subtraction, and multiplication of integers. We explicitly assume that no one is able to do division in this limited algebra. Just think yourself back at the time when you had just learned multiplication, but not division. With this limited algebra we now describe how Bob sends a message to Alice. For our example, this message is just one number.

The procedure has three parts. First, we describe how to generate the public and the private key; then we explain how to encrypt and decrypt messages.

Construction of the Keys

First we need the keys, the private and the public one. The message is going to be sent from Bob to Alice. Thus, Alice needs the private key to decrypt the message. To get this private key, Alice thinks up two numbers and multiplies them. The first of the two numbers becomes the *private key*. The other one and the product are the *public key*. Thus, the private key is just one number and the public key consists of two numbers, the *public factor* and the *public product*.

The public and the private key

p	private key
11	public factor
143	public product

For you it is easy to compute the private key because you are able to do a division. The private key is $p = 143/11 = 13$. In our limited mathematics, however, this private key stays secret.

At the bulletin board of the school Alice posts the following note for everybody to read. However, nobody is able to do divisions. Thus the private key of Alice stays secret.



Encryption

Bob, who wants to send a message to Alice, also reads this note. Assume the message is to contain the date of the next party at Bob's home, which is December, 5th. The message is the number 5 because it is already known that the party takes place in December.

Bob encrypts the message as follows. He knows the messages 5 and the public key of Alice, that is, the numbers 11 and 143.

Bob thinks up a fourth number; this number is called *sending secret*. Using this sending secret he computes the encrypted data. This encrypted data consists of two numbers, the *encrypted message* and the *decryption help*.

Bob computes the product of the *sending secret* and the *public product* of Alice. The *encrypted message* is computed by adding the *message* to this product. If we assume that Bob chose 3 as the *sending secret*; then, the *encrypted message* is $5 + 3 \cdot 143 = 434$. This number, 434, is published, but the number 3 has to be kept secret. Otherwise everyone could reconstruct the message by computing $434 - 3 \cdot 143 = 5$.

Since we revealed the *sending secret*, we have Bob choose another one, called s , and he doesn't tell what the value is. Bob computes the *secret message*:

Computing the secret message

$$5 + s \cdot 143 = 1292$$

The 1292 becomes public, while s is kept secret.

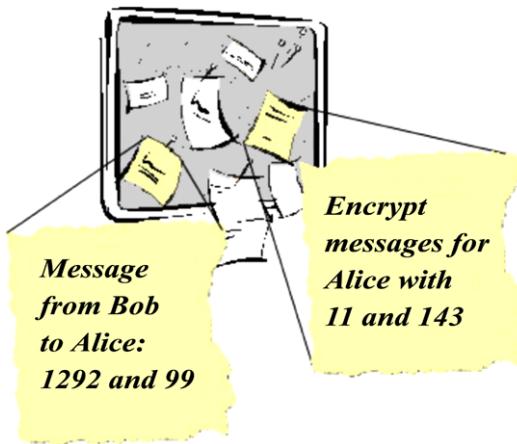
As long as only Bob knows the *sending help*, no one will be able to decrypt this message. But how can Alice find out what the message is? In order that only Alice be able to decrypt the 1292, Bob provides as *decryption help* the product of the *sending help* and the *public factor* of Alice.

Bob computes $11 \cdot s = 99$. This number is made public as well.

Computing the decryption help

$$11 \cdot s = 99$$

Bob places the following note at the bulletin board of the school.



Thus, everyone knows the following numbers because everyone can read the two notes at the bulletin board.

Numbers known publicly after encryption and posting

11	<i>public factor from Alice</i>	
143	<i>public product from Alice</i>	$11 \cdot p$
1292	<i>encrypted message</i>	$5 + s \cdot 143$
99	<i>decryption help</i>	$s \cdot 11$

Even if everyone knows the computation steps of Bob, without division it is impossible to compute the *sending secret* or the *private key*. And without the *sending secret* it is impossible to find out what Bob's message was.

Decryption

Now Alice decrypts the message from Bob. Like the others, Alice can not do divisions. However, Alice knows her *private key*. Let us take a closer look at the encrypted message.

The encrypted message

$$\begin{array}{r|l}
 1292 & \text{message} + \text{sending secret} \cdot 143 \\
 = & \text{message} + \text{sending secret} \cdot \text{public product} \\
 = & \text{message} + \\
 & \quad \text{sending secret} \cdot \text{public factor} \cdot \text{private key} \\
 = & \text{message} + \text{decryption help} \cdot \text{private key}
 \end{array}$$

Alice can obtain the message by the following calculation.

$$\begin{array}{r}
 99 \ 13 \\
 - 99 \\
 \hline
 297 \\
 - 1292 \\
 \hline
 1287 \\
 - 1287 \\
 \hline
 5
 \end{array}$$

*Party at
December 5th*

Decryption

$$1292 - 99 \cdot (\text{private key } p) = 5$$

As you can see, Alice did not need division for this calculation.

The Eavesdropper

In espionage movies we see how spies eavesdrop on calls and other conversations. Often, secure connections are used to prevent this. How can we make sure that these connections are really secure?

For our little message from Bob to Alice we did not need any secure connection. All communication was done publicly by using the bulletin board of

the school. An eavesdropper who cannot do division cannot decrypt the message. It is kept private as long as Alice and Bob do not each reveal a secret number. Furthermore, it was not necessary for Alice and Bob to meet before. This is the main advantage of this system compared to the one-time pad from Chap. 15.

Without Limited Mathematics

Our system is secure as long as no one is able to do divisions. However, all of us have learned at school how to do divisions.

If you have read the previous chapters of this book carefully, you may have noticed that by using one of the previous methods a division could be achieved very fast (see Chap. 1). To compute the fraction a/b one may simply guess a candidate c between 0 and a for given natural numbers a and b . Then, one checks whether $b \cdot c$ really equals a . If the result is bigger than a , a smaller value for c must be the true one. If the result is smaller than a , it has to be the other way around. Applying this idea repeatedly, always choosing the median of the remaining interval (i.e., performing a binary search), we get the result very fast.

This method uses the fact that the fraction a/b becomes smaller when b gets larger. Otherwise, binary search would not be applicable.

A binary search will fail, however, if our calculus is based on residues. The mathematical background for modular arithmetic is explained below. Since a division is still efficiently computable for residues, we also have to replace the division by something else.

ElGamal's Method

There are (many) more operations in mathematics than just the basic arithmetic operations. We cleverly choose new operations that are easy to perform, and use them to replace addition, subtraction, and multiplication in our primitive system from above. The operations are the following.

- Instead of addition we use *modular multiplication*. (Modular multiplication is a multiplication on residues. See also the algorithms from Chap. 17 where modular multiplication is used for sharing a secret.)
- Instead of subtraction we use *modular division*.
- Instead of multiplication we use *modular exponentiation*.
- Instead of division we use the *modular logarithm*.

The resulting method is known as the ElGamal cryptosystem. Till now, no one knows an algorithm to compute the modular logarithm (also known as the discrete logarithm) for large numbers (numbers with more than 1000 digits) efficiently. All known algorithms, however, need several centuries even when running on the fastest computers. Even if at some time in the future the message is decoded without the private key, it does not help because the party will be over by that time.

Modular Multiplication and Modular Exponentiation

Before we describe how modular exponentiation can be performed efficiently, let us take a quick look at *modular multiplication* first.

In our modular arithmetic, all calculations will use a prime number p , that is, a number having exactly two distinct divisors, p and 1. As you know, the first prime numbers are 2, 3, 5, 7, 11, 13, 17, 19,

The prime number p is called the *modulus* for our calculations. The result of the modular multiplication $(a \cdot b) \bmod p$ is defined as the remainder of the product $a \cdot b$ when divided by p . For instance, $(5 \cdot 8) \bmod 17 = 6$ since dividing $5 \cdot 8 = 40$ by 17 gives the quotient 2 (which we ignore) and the remainder 6.

Now we take a closer look at *modular exponentiation*. In a modular exponentiation a number a is multiplied precisely b times by itself. Each time the result of the multiplication is the remainder of the division by p . This result is written as $(a^b) \bmod p$.

The result of $(3^9) \bmod 17$ is 14, as shown in the following calculation.

Computing $(3^9) \bmod 17$

$$\begin{aligned} 3^9 \bmod 17 &= (3 \cdot 3 \cdot 3) \bmod 17 \\ &= ((3^8 \bmod 17) \cdot 3) \bmod 17 \\ 3^8 \bmod 17 &= ((3^4 \bmod 17) \cdot (3^4 \bmod 17)) \bmod 17 \\ 3^4 \bmod 17 &= ((3^2 \bmod 17) \cdot (3^2 \bmod 17)) \bmod 17 \\ 3^2 \bmod 17 &= (3 \cdot 3) \bmod 17 \\ &= 9 \\ 3^4 \bmod 17 &= (9 \cdot 9) \bmod 17 \\ &= 13 \\ 3^8 \bmod 17 &= (13 \cdot 13 \cdot 3) \bmod 17 \\ &= 14 \end{aligned}$$

This example also shows that $3^9 \bmod 17$ can be computed with fewer than eight multiplications by cleverly using intermediate results in multiplications. The following procedure uses this idea.

Computing $(a^b) \bmod p$

- 1 If $b = 0$, then the result is 1.
- 2 If $b = 1$, then the result is a .
- 3 If b is odd, the result is $((a^{b-1} \bmod p) \cdot a) \bmod p$.
- 4 The only remaining case is that b is even and we compute:
- 5 $h = (a^{b/2}) \bmod p$.
- 6 The result is $(h \cdot h) \bmod p$.

In this description, for computing $(a^b) \bmod p$ we used other values of the form $(c^d) \bmod p$. The values c and d are always smaller than the values a

and b . Therefore, this method works out and eventually terminates. From this we get the following recursive algorithm:

Recursive algorithm for computing $(a^b) \bmod p$

```

1  ExpMod( $a, b, p$ )
2    If  $b = 0$  then return 1.
3    If  $b = 1$  then return  $a$ .
4    If  $b$  is odd then
5      begin
6         $h = \text{ExpMod}(a, b - 1, p)$ 
7        return  $(h \cdot a) \bmod p$ 
8      end
9     $h = \text{ExpMod}(a, b/2, p)$ 
10   return  $(h \cdot h) \bmod p$ 

```

In the following table we present the values $2^b \bmod 59$ for all even b . This table shows how irregular the results are compared to a normal multiplication due to the application of the modulo operation in intermediate steps.

Values for $2^b \bmod 59$

$b 2^b \bmod 59$	$b 2^b \bmod 59$	$b 2^b \bmod 59$
0 1	20 28	40 17
2 4	22 53	42 9
4 16	24 35	44 36
6 5	26 22	46 26
8 20	28 29	48 45
10 21	30 57	50 3
12 25	32 51	52 12
14 41	34 27	54 48
16 46	36 49	56 15
18 7	38 19	

The *modular logarithm problem* is to do the inverse operation, which means to obtain for a given number x (such as $x = 42$) a number b such that $2^b = x$. Of course, for the modulus 59 this number b can be found by trying out all possible exponents. But for large numbers this becomes very wasteful. However, even for large numbers modular exponentiation is still possible, as the numbers in the above algorithm quickly get smaller.

If the exponent b has ten digits, then at least a million possible solutions have to be considered to find the modular logarithm. On the other hand, the computation of the modular exponentiation with an exponent with ten digits requires at most 65 modular multiplication. This shows the big difference between the complexity of a modular logarithm and modular multiplication. Nowadays, the method of ElGamal is used on numbers with more than the thousand digits. Thus, the “complexity gap” becomes much larger: Trying out

all possibilities is absolutely impossible, and in fact no alternative algorithm is known that solves the discrete logarithm problem for moduli p as large as this in reasonable time.

Description of ElGamal's Cryptosystem

Now we are able to describe ElGamal's cryptosystem. It is a simple transformation of the system which was presented above using limited mathematics.

In the above method the associative law $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ was used. Within ElGamal's cryptosystem a similar law for the exponential calculation should apply. The notation g^x indicates that g is to be multiplied with itself x times. Without going into details, we just state the law which applies here: $(g^{a \cdot b})^c = g^{(a \cdot b) \cdot c} = g^{a \cdot (b \cdot c)} = (g^a)^{b \cdot c}$.

Furthermore, it is important that all numbers between 1 and $p - 1$ can appear as values $g^x \bmod p$, that is, for each number i between 1 and $p - 1$ there has to be a number j with $i = g^j \bmod p$. In such a case g is called a *generator* (modulo p), as g generates all elements of $\{1, \dots, p - 1\}$. The following table shows that 4 is not a *generator* modulo 7, but 3 and 5 are.

Generators for modulo 7: 3 and 5

i	$3^i \bmod 7$	i	$4^i \bmod 7$	i	$5^i \bmod 7$
0	1	0	1	0	1
1	3	1	4	1	5
2	2	2	2	2	4
3	6	3	1	3	6
4	4	4	4	4	2
5	5	5	2	5	3
6	1	6	1	6	1

Alice – in the meantime she has moved on to high school – first finds a prime number p and a generator g for modulo p . For example, let 59 be the prime number and 2 be the generator. In the next step, Alice chooses her *private key* x . With this private key she computes the first part of her *public key* by $y = (g^x) \bmod p$, in our case, $42 = (2^x) \bmod 59$. She announces publicly the numbers $p = 59$, $g = 2$, and $y = 42$ on the bulletin board of her school. Note that in the above table the number 42 does not appear. Thus the private key of Alice is odd. Can you find her private key?

Numbers for Alice's ElGamal system

59	prime number of Alice
2	generator of Alice
x	private key of Alice
42	public key of Alice

Bob wants to send the date of the new party to Alice. This year the party is ten days later, so the secret number is 15. To send this secret, Bob reads Alice's three numbers from the bulletin board. Bob chooses 9 as the *sending secret*. He computes: $a = 2^9 \bmod p$, obtaining $a = 40$. To get the *encrypted message* he further computes $b = (15 \cdot 42^9) \bmod 59$, which yields $b = 38$. Then he announces these two numbers on the bulletin board of the school.

Bob's ElGamal values

40	<i>decryption help for Alice</i>
38	<i>encrypted message for Alice</i>

Alice starts the decryption. At first she computes $h = 40^x \bmod 59$. The result is 34. By “division modulo p ,” Alice can find out that $33 \cdot 34 \bmod 59 = 1$; we say that 34 is the *inverse* of 34 modulo 59.

Let us have a closer look at such inverses. When dealing with addition, the inverse of a number x is $-x$ since $x + (-x) = 0$. The inverse of x for a multiplication is $\frac{1}{x}$ because $x \cdot \frac{1}{x} = 1$. For modular multiplication the inverse of a number x is a number y for which $(x \cdot y) \bmod p = 1$.

Given a prime number p and an x between 1 and $p - 1$, it is possible to calculate the inverse of x modulo p (it is unique) by fast exponentiation: Just calculate $y = x^{p-2} \bmod p$. Then, clearly $x \cdot y \bmod p = x^{p-1} \bmod p$, and it is known (“Fermat’s Little Theorem” from Number Theory) that this number is always equal to 1.

The message is decrypted by computing $38 \cdot 33 \bmod 59$. Alice thus obtains 15 – the original message.

In our description we avoided showing off the private key of Alice. You are asked to detect this number. Maybe you see that this will be very hard if the numbers become large.

Further Methods

There are many public-key cryptosystems that are built in a similar way. To make them more secure, more complicated operations are used, for instance, “elliptic curves” or even “hyperelliptic curves.”

Security

Of course, the question arises about how secure such a system is. One important parameter is the size of the used numbers. The numbers have to be large; so large that a program which uses trial and error takes a very long time. It should take such a long time that the revelation of the secret is dispensable. In our example it suffices to keep the secret safe till the party is over.

On the other hand, it could happen that suddenly one finds an algorithm computing the modular logarithm efficiently. So far there exists no proof that

such an efficient algorithm does not exist. Mathematicians and computer scientists have been trying to find such a method for many years. Despite all their efforts no such efficient algorithm has been found yet. Scientists therefore assume that such an algorithm does not exist.

If one day an efficient solution for the modular logarithm is found, then the decryption by ElGamal will become insecure. We hope that this will not happen. Note, however, that in this case we still could use the above trick with other mathematical operations.

Further Reading

1. ElGamal's cryptosystem is also described in Wikipedia:
http://en.wikipedia.org/wiki/ElGamal_cryptosystem
2. Chapter 14 (One-Way Functions)
In that chapter another public-key cryptosystem is mentioned, the so-called RSA cryptosystem.
3. There are several books introducing encryption systems and discussing their history. For instance, you may have a look at the following:
Simon Singh: *The Code Book: Science of Secrecy from Ancient Egypt to Quantum Cryptography*. Anchor Books, 2000.

How to Share a Secret

Johannes Blömer

Universität Paderborn, Paderborn, Germany

The following story line appears again and again in movies and novels like *Cutthroat Island*, *The Good, the Bad and the Ugly* and *Treasure Island*. Part of a treasure map is found, but this part of the map is not sufficient to locate the treasure. To find the treasure the whole map is needed. Therefore, the finder of the fragment tries to find the other parts of the treasure map as well. Of course, the owners of these other parts are also extremely interested in finding the parts of the map they are missing, and so the adventure begins.

This story line is just one example of the problem we present in this chapter: how to share a secret. We want to investigate how to partition a treasure map or any other kind of information in such a way that, without knowing all pieces, the treasure cannot be found or the information cannot be reconstructed completely. We will see that there are much better methods than just cutting the map into pieces. Actually, if we think about it, it is then not very convincing that one needs all parts of a treasure map to find the hidden treasure.

The general problem is easy to state. We want to partition a secret, let us call it S , into a certain number of pieces or shares. The individual pieces are then given to different persons. Furthermore, we want to achieve the following goals:

1. If all persons collaborate and combine their shares, then they can reconstruct the secret S completely.
2. However, if only a subset of the persons that received some piece of the secret collaborate, then the persons in this subset should not be able to reconstruct the secret S completely. Moreover, the persons in this subset should only be able to gain little information about the secret S .

Of course, sharing a secret is not just a common story line in movies and novels; it has many more serious and realistic applications. Imagine that an important document of a government or an enterprise is stored in a safe. There is consensus that the document will be published only if all members of a special committee agree to the publication. To realize this, the safe is

secured with several padlocks, one for each member of the committee. Every member of the committee has the key to one of the padlocks. To open the safe and publish the document every member has to unlock her padlock, thereby agreeing to the publication of the document.

With secret sharing we can also guarantee that the document is published only if all members of the committee approve the publication. To do so, we secure the safe, not with padlocks that require a key, but instead with a single combination lock, whose secret combination consists of, say, decimal digits. The secret combination is divided into several pieces, one piece for each member of the committee, and every member of the committee gets her own piece of the secret combination, that is, her own partial secret. If all members of the committee agree to the publication of the document, they combine their partial secrets to retrieve the secret combination, open the safe, and publish the document. The partial secrets of the committee members are like keys for different padlocks that secure the safe. This example demonstrates how we can use secret sharing to replace physical keys by secret information.

In addition to sharing the secret combination of a safe, there are many other applications of secret sharing. In fact, secret sharing is one of the most important techniques in cryptology, the science of encrypting messages, or, more generally, the science of securing information against unauthorized access and modification. If we combine methods to share a secret with *public-key cryptography* (see Chap. 16), then we can replace keys as well as safes and locks by secret information and algorithms. Using such a combination of methods we can encrypt data in such a way that, like in our example above, documents can be recovered or decrypted only if all committee members contribute their shares of the secret. Here the partial secrets are parts of a public key in a public-key encryption scheme.

A Simple Method to Share a Secret

So far we have not described methods to share a secret. How can we replace locks and keys by partial secrets, each of which is known to a single committee member? To discuss the first idea, we return to our document locked in a safe that is secured by a combination lock with a 50-digit secret combination. Let us assume that the secret combination is

$$S = 65497 \ 62526 \ 79759 \ 79230 \ 86739 \ 20671 \ 67416 \ 07104 \ 96409 \ 84628.$$

Let us also assume that our committee has ten members. Therefore, we want to partition our secret S into ten partial secrets such that only all ten committee members together are able to reconstruct the secret S . What about giving each committee member 5 of the 50 digits of our secret combination S (Fig. 17.1).

You can see immediately that this is not such a great idea. If 9 out of 10 committee members decide that they want to publish the document, they already know 45 of the 50 digits of the secret combination necessary to open

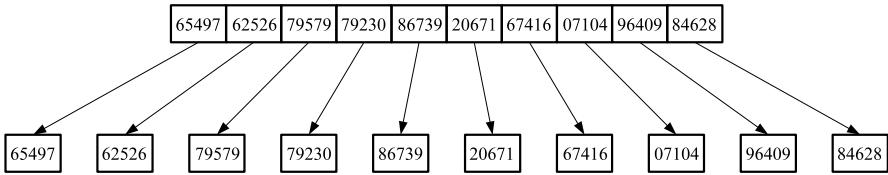


Fig. 17.1. A simple example of how to share a secret

the safe. However, remember that we want that even nine committee members together can learn little or nothing about the secret S . But with our simple idea, once 9 out of 10 committee members collaborate, each of them all at once knows 45 digits of S instead of the 5 digits each of them individually knew before they collaborated. We can put this differently. Before the collaboration each committee member had to try 10^{45} possible combinations for the 45 digits they did not know. By collaborating, the nine committee members who exchange their parts of the secret combination reduce the number of possible combinations they have to try to determine the secret S to $10^5 = 100000$ combinations. To appreciate the information gain the 9 collaborating committee members achieve, let us assume that a single person can check in one second whether a given 50-digit number is the secret combination for the safe. To try all 10^5 possible combinations for the 5 digits that are still unknown, the 9 collaborating committee members require roughly 3 hours. This follows from the fact that a single hour has 3600 seconds; hence in an hour 3600 possible combinations for the safe can be tested. We see that within a relatively short time the nine collaborating committee members can determine the secret combination of the safe, open it and publish the document even without the consent of the tenth committee member. Now let us consider how much time it takes a single committee member, who knows only 5 of the 50 digits of the secret combination, to try out all 10^{45} possible combinations for the digits she does not know. Assuming again that it takes a second to test a single combination, a simple calculation shows that a single member will need roughly 10^{35} years to determine the secret combination. Physicists tell us that the universe has not existed for this long, and that in all likelihood it will cease to exist well before a single committee member will be able to determine the complete secret combination.

So, let us try a different technique that brings us much closer to our ultimate goal of sharing secrets. In this method we distribute our secret S among the ten committee members by choosing ten random numbers larger than zero that add up to S . These numbers are the partial secrets that are given to the ten committee members. Let us look at a small example. In this example the secret S is a natural number, and the partial secrets are numbers between 1 and 50. To simplify the example, we want to share or distribute the secret among four people rather than the ten committee members from our previous example. Assume the secret S is 129. Then the partial secrets may be chosen

as 17, 47, 31 and 34, since $17 + 47 + 31 + 34 = 129$. It is clear that if all four people owning a partial secret collaborate then they can reconstruct the secret S ; they simply have to add up their partial secrets. But again we have a problem: All participants know that the partial secrets lie between 1 and 50. Hence, even before receiving their shares the participants know that the secret S lies between 4 and 200. Now assume that the first three participants join forces to gain information about S . To do so, they simply add up their partial secrets and obtain $17 + 47 + 31 = 95$. Now they know that S will be a number between $95 + 1 = 96$ and $95 + 50 = 145$, since the fourth participant also received a partial secret between 1 and 50. The number of possible values for S has dropped from almost 200 to 50, and the three colluding participants have gained a lot of information about S . But a simple trick helps us modify the method in such a way that a proper subset of participants will not gain any information about the secret S even if they exchange all their partial secrets. The trick is to use *division with remainder*.

The new method works as follows. We assume that the secret S that we want to share among a certain number of participants is a natural number between 0 and some really large number N . In our example with the document stored in a safe secured with a 50-digit secret combination, the number N will be 10^{50} . To work with concrete numbers, let us assume again that we want to share a secret value among ten participants. But once you understand the method you realize that with this method we can share a secret among an arbitrary number of participants. To share the secret S we proceed in two steps.

1. First we choose nine random numbers between 0 and $N - 1$. Let us call these numbers t_1, t_2, \dots, t_9 . These numbers are the partial secrets for the first nine participants.
2. To determine the tenth partial secret t_{10} , first we compute $t_1 + \dots + t_9$ and divide this sum by N . However, we perform division with remainder and are only interested in the remainder R . Next we look at the difference $S - R$. If $S - R$ is positive, then t_{10} is $S - R$. If $S - R$ is negative, then t_{10} is $S - R + N$. With this recipe we get that the secret S is the remainder if we divide $t_1 + t_2 + \dots + t_{10}$ by N .

To illustrate this method let us look at a simple example, where all numbers are small enough to compute them by hand. We choose $N = 53$, and we want to share the secret $S = 23$ among four people.

1. We choose the first three partial secrets. Let these be 17, 47 and 31.
2. To determine the fourth partial secret, first we compute the sum of the first three secrets, $17 + 47 + 31 = 95$. We divide 95 by 53 and obtain remainder $R = 42$. Since $S - R = 23 - 42$, which is negative, the fourth partial secret is $23 - 42 + 53 = 34$.

$$(17 + 47 + 31 + 34) : 53 = 2 \text{ remainder } 23$$

partial secrets: 

secret: 

Fig. 17.2. Example for secret sharing by division with remainder

You can also follow this example in Fig. 17.2.

Does this method really satisfy our requirements? Let us look at our simple example. If the four owners of partial secrets collaborate they can compute the sum of their partial secrets. The secret S is then simply the remainder if they divide the sum by $N = 53$. In our case, the sum is 129, and the remainder after division by 53 is 23, the secret. You can easily verify that the approach works not only in our simple example but in general.

What happens if not all participants collaborate? Can a proper subset of the participants determine the secret S or gain significant information about S ? At first glance it might seem that we treat the last participant differently from the remaining participants, since her partial secret depends on the other partial secrets. However, if we take a more careful look we realize that this impression is misleading. Let us go back once more to our simple example, and let us look at the first partial secret, 17. The sum of the partial secrets excluding 17 is $47 + 31 + 34 = 112$. Then $x = 17$ is the unique number x such that 23 is the remainder of dividing $112 + x$ by 53. We see that the first partial secret 17 depends on the remaining partial secrets in exactly the same way that the last partial secret 34 depends on the remaining partial secrets.

We still have not answered the question of what happens if a proper subgroup of participants tries to determine the secret from its set of partial secrets. Phrased differently, the question becomes: Do we really need all partial secrets to determine the secret? Again, we first look at our example. Assume that the last three participants with partial secrets 47, 31 and 34 try to determine the secret, or more modestly try to gain some information about the secret. Of course, we assume that they know the number $N = 53$, but they do not know the partial secret of the first participant. They also know the method we use, so they know that the secret is the remainder we get by dividing the sum of the partial secrets by 53. Now they can compute the sum of their partial secrets, which is 112. Dividing this number by 53 gives the remainder 6. Had the first partial secret been 0 instead of 17, then the overall secret would have been 6 instead of 23. A first partial secret of 1 would have led to the secret 7. And so on, until the possible first partial secrets 51 and 52, which would have led to secrets 4 and 5, respectively. More precisely, for every number s between 0 and 52 there is a number t such that the sum of 121 and t has remainder s when divided by 53. This means that with first partial secret t the overall secret would have been s instead of $S = 23$. This, in turn, implies that if the last three participants knew their own partial secrets

they cannot exclude a single value for the overall secret. Summarizing, we can say that the last three participants together do not learn anything about the secret from their partial secrets. You can easily check that this is not only true for our simple example, but that it is correct in general.

However, you have to ensure that your numbers are not too small. In our example with $N = 53$ it is certainly not very difficult to try all possible values for the missing partial secret. After all, there are just 53 possible values. Even simpler, since the secret itself can take on just 53 values, you can try all possible values for the secret without knowing or taking into account any partial secrets. In applications of secret sharing therefore one chooses much larger values for N . For example, you may take $N = 10^{50}$. With this choice, there are 10^{50} possible values for a partial secret of the secret itself. As we have seen before, in this case it is unrealistic to simply try out all possible values for a partial secret.

General Secret Sharing

So far we have considered only the situation where all recipients of partial secrets must collaborate to recover the secret. Now we want to look at a more general situation in which any sufficiently large number of recipients of partial secrets together can reconstruct the secret.

Let us begin with the simple case where any two out of three recipients of partial secrets should be able to recover the secret. However, a single recipient of a partial secret should not be able to reconstruct the secret or gain a lot of information about the secret. In this case, representing the secret as the sum of partial secrets, the idea that was so successful in our scheme above, will not get us very far. We need a new idea, and a little geometry will help us. Let our secret be a point P in the plane. We can imagine that the two coordinates of point P constitute the secret combination of a safe. We also choose three lines in the plane that meet in point P . The three lines are the partial secrets. We have illustrated this idea in Fig. 17.3. Now if any two out of the three recipients of partial secrets collaborate they can simply compute the intersection point of the lines that constitute their partial secrets to compute the overall secret. You can see this in the three pictures in Fig. 17.4.

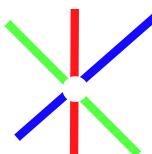


Fig. 17.3. Three lines that intersect in a point. The intersection point is the secret

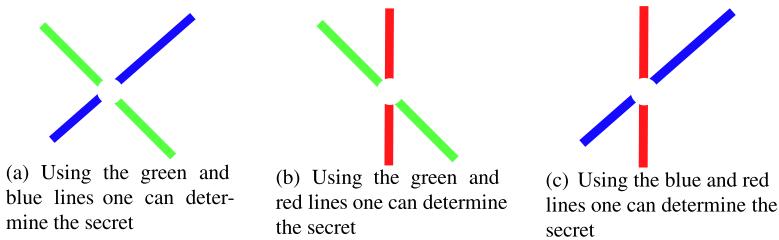


Fig. 17.4. Secret sharing in the plane

What does a single participant learn about the secret from her partial secret? Clearly, she learns something. Before receiving her partial secret she only knew that the secret is some point in the plane. After seeing her partial secret she knows the secret lies on the line that is her partial secret. So, she definitely has learned something, but she still does not know the secret.

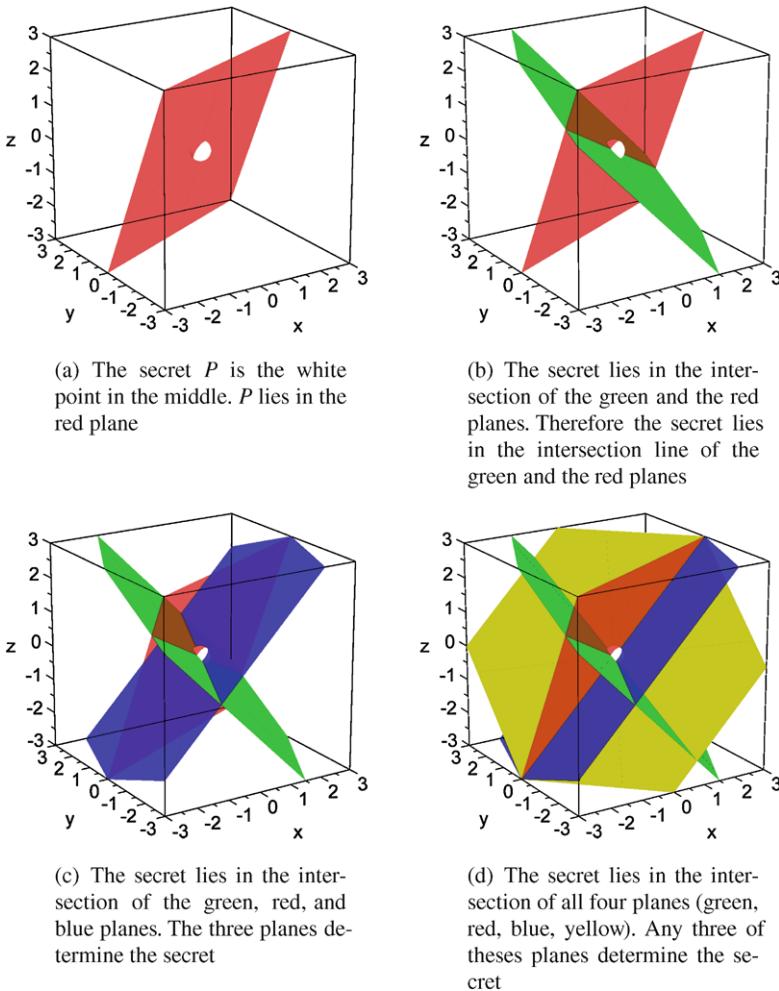
We can easily generalize this method such that any two out of m recipients of partial secrets can reconstruct the secret from their partial secrets. To do so, the secret is again a point P in the plane. But instead of three lines that intersect in P we choose m lines that intersect in P . Every one of these lines is a partial secret.

What about the case that any three recipients of partial secrets must be able to compute the secret? In this case we leave the plane and enter (three-dimensional) space. Again, our secret is a point P ; this time, however, it is a point in the three-dimensional space. As partial secrets we choose planes in the space such that any three of these planes intersect in point P . In Fig. 17.5 you can see this for four partial secrets.

Any three of our recipients of partial secrets can reconstruct the secret by computing the intersection of the three planes they received as partial secrets. You can see this nicely in Fig. 17.5(c). However, if fewer than three participants collaborate they also learn something about the secret. For example, if two recipients of partial secrets work together they can compute the line in which their two planes intersect. You can observe this by looking at the red and green planes in Fig. 17.5(b). If the recipients of the red and green planes combine their partial secrets, they can deduce that the secret must lie on the intersection line of the red and green planes. However, they still have no clue which point on this line is the secret.

Secret Sharing, Information Theory and Cryptography

Of course, we can further generalize our problem and ask whether it is possible to share a secret S among m people such that any t (or more) of these people are able to reconstruct the secret, whereas fewer than t people are not able to reconstruct the secret, or more stringently, do not gain a lot of information

**Fig. 17.5.** Secret sharing in space

about the secret S . As it turns out, for any combination of m and t this is possible and is called *t-out-of-m secret sharing*. One way to realize such general secret sharing schemes is by generalizing the geometric constructions that we discussed above. To obtain a t -out-of- m secret sharing scheme one has to go to t -dimensional spaces.

There are even constructions for t -out-of- m secret sharing in which fewer than t people learn absolutely nothing about the secret. These schemes do not rely on geometry; instead they use so-called *polynomials*. This construction was invented by Adi Shamir, a famous cryptologist, and therefore it is called *Shamir's secret sharing scheme*.

Who computes and distributes the partial secrets? So far we have completely ignored this question. Obviously, this is an important question. Whoever computes and distributes the partial secrets must know the secret. If one applies secret sharing schemes, you usually have to assume that a trustworthy person exists who computes and distributes the partial secrets. One can think of this person as a completely trustworthy and incorruptible referee.

What exactly do we mean, if we say that someone has gained information? What is information? Somehow we all know what information is. But if we want to deal with information in a precise mathematical sense, as we want in secret sharing, then we have to be more precise. Once you understand the secret sharing methods that we presented above, you will be able to define concepts like information and information gain in a precise mathematical manner. For example, partial shares reveal no information about the overall secret if knowing the partial shares does not reduce the number of possible values for the secret. In 1948 the famous mathematician Claude Shannon used these ideas to found *information theory* as a branch of mathematics.

We can go even further. Information that we can gain in principle but only by spending an unreasonable amount of time like 10^{35} years is useless. Secret sharing provides a good example. No matter how we share the 50 digit secret combination of a safe among 10 members of a committee, in principle it is possible to determine the secret combination by trying all possible 10^{50} combinations. But in practice this is not a viable option. The number 10^{50} is so enormous that even with the help of a computer, we cannot try all 10^{50} possible secret combinations. Therefore, we can say that a secret cannot be revealed if determining the secret simply takes too much time to be practically feasible. These considerations lead us way beyond information theory. They lead to questions about how many resources are needed to compute something or to gain some information. In this book you can learn, for many interesting problems, like multiplying large integers (Chap. 11), how much time is required to solve them. On the other hand, in the chapter about public-key cryptography (Chap. 16) or about one-way functions (Chap. 14) you also see that sometimes it is very useful if a problem cannot be solved efficiently or if it is impossible or very difficult to gain certain information.

Further Reading

1. Chapter 16 (Public-Key Cryptography)

In many applications the secret keys of public-key encryption schemes as described in this chapter are not given to a single person. Instead, using secret sharing techniques, secret keys are distributed among a group of people. In this way you can avoid situations in which the secret key of a large cooperation is owned by a single person.

2. [**http://en.wikipedia.org/wiki/Secret_sharing**](http://en.wikipedia.org/wiki/Secret_sharing)

In this article you will find (sometimes longer) descriptions of the techniques that we discussed.

3. Wade Trapp and Lawrence Washington: *Introduction to Cryptography*. Pearson Education International, 2nd edition, 2006.

If you want to learn more about secret sharing this book is a good starting point. Of course, the explanations in this book are much more detailed and technical than we were able to give in this short chapter.

Playing Poker by Email

Detlef Sieling

Technische Universität Dortmund, Dortmund, Germany

In this chapter we explore how to play card games like poker without a meeting of the players. Instead, the cards are distributed using email or snail mail. Different from commercial online poker systems, the cards are shuffled and distributed by the players. There is no extra dealer who the players have to trust. Obviously, there are some difficulties: The player shuffling and distributing the cards has to do this without gaining any knowledge of the cards he is distributing, while he has to send emails with information about those cards. Furthermore, this player must not distribute any card more than once, while he must not know which cards have already been distributed. Finally, if any player cheats, the other players should be able to detect this.

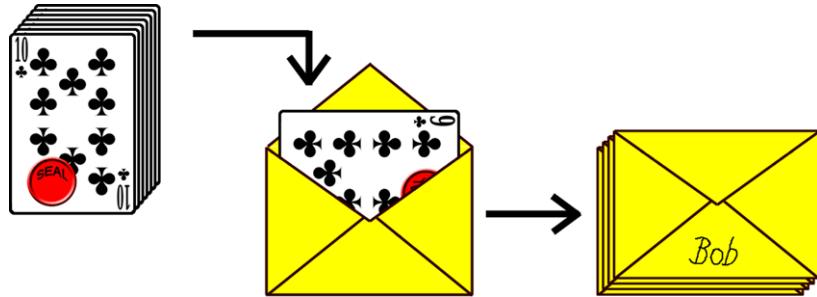
Dealing Cards by Snail Mail

In order to collect ideas on how to play card games by email, we first try to do this by snail mail. We first look at the case of two players, who are called Alice and Bob. They are at different places and are thus not able to observe each other. Then each player can easily cheat by using a second deck of identical cards. From those cards, he can select a good hand, e.g., a royal flush. In order to make this impossible, each card of the deck in use will have a seal indicating that this card is really from the deck in use and not from some other deck.

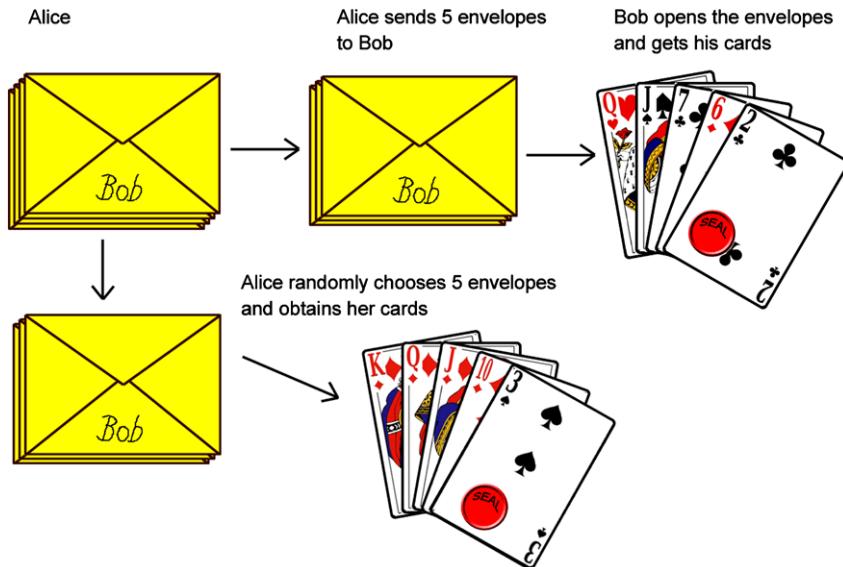
How to Shuffle and Distribute the Cards

For our poker game we use a card deck with the 52 cards club-ace, club-two, ..., diamond-king. After shuffling the cards each player has to get five cards. How is it possible for one player to shuffle and distribute the cards without gaining any information on the cards given to the other player? We do this with envelopes. Bob puts each of the 52 cards into a separate yellow envelope.

Later on, it will be crucial to prove who has put a card into a particular envelope. Thus Bob signs each of the envelopes.



Then Bob shuffles the envelopes and sends them to Alice. Alice cannot distinguish the envelopes according to their contents. Thus she is not able to select good cards for herself and poor cards for Bob. So Alice can only shuffle the envelopes again and select five ones for herself and five for Bob. This is exactly a random selection of cards for herself and for Bob. Alice returns the five envelopes to Bob. Then both players can open the envelopes and obtain their cards.



How can the players cheat? For example Alice can open other envelopes in order to get a larger number of cards from which she could select the best ones. Obviously, Alice cannot be prevented from doing this. On the other hand, if Alice does not cheat, she retains 42 closed envelopes with Bob's signature,

and she can present those envelopes if the two players meet later on. Since the envelopes are signed by Bob, Alice is not able to put a card back into the envelope or into a new one. Bob could also try to cheat when he puts the cards into the envelopes. For example, he could retain a card for himself and leave an envelope empty. If he gets the empty envelope during the game, he gets the card he retained and thus this is not at his advantage. Otherwise, the empty envelope and Bob's cheating will be detected at the latest when Alice and Bob check the envelopes after the game.

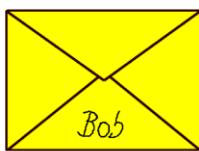
How to Bid

The next step is the bidding. What is said during the bidding can also be written into letters. Thus we do not need new ideas to do this by mail.

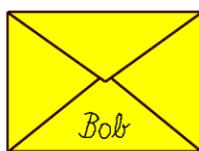
How to Replace Cards

After the bidding each player may replace one or more of his cards with randomly chosen ones from the card deck. First Alice may replace n cards (where n is between 1 and 5). Here we have the following problem: In the first step Alice has to drop n cards, and only afterwards may she get the new cards. If she takes her new cards by selecting envelopes as described above, it is not possible to prevent her from opening the new envelopes and selecting the cards to be dropped afterwards. Thus Bob also needs to be involved into replacing the cards of Alice.

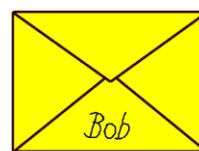
On the other hand, Alice cannot return the remaining 42 envelopes to Bob because then she would lose the proof that she has not cheated so far. Furthermore, Bob could have marked the yellow envelopes and could thus recognize the envelopes with good or bad cards. He could do this, for example, by signing the envelopes in a slightly different way. This can be done even less obviously than shown in the following picture where the lower-case "b" in Bob is written differently according to the contents of the envelope.



Envelope with
an ace

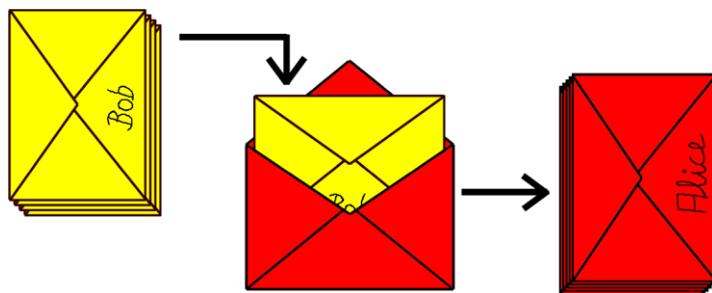


Envelope with
a king

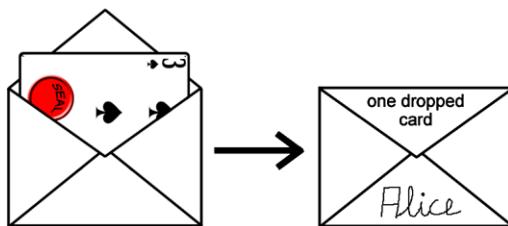


Envelope with
a two

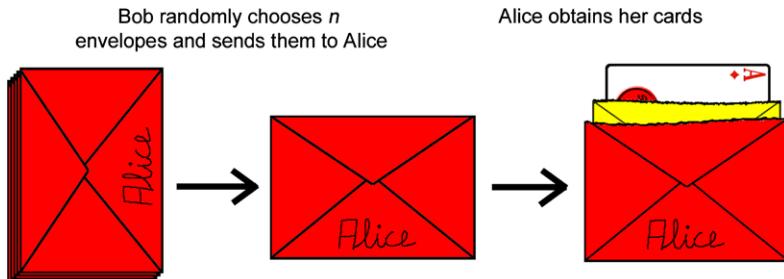
However, the trick with envelopes works again. Alice puts the remaining 42 yellow envelopes into slightly larger red envelopes, signs those envelopes, shuffles them and sends the pile to Bob.



Furthermore, she puts the n cards she would like to replace into a separate envelope and sends it also to Bob. Bob must not open the envelope with the dropped cards because he should not gain any knowledge on those cards. Later on, both players together can verify that the envelope is still closed and afterwards that it really contains n cards.



Since Bob cannot distinguish the red envelopes, he can only shuffle them again and select n envelopes to return to Alice. In this way Alice gets the replaced cards.



If Bob would like to replace cards, this can be done similarly, where Bob has to put the red envelopes into another layer of envelopes.

The Showdown

The game ends with presenting the cards to the other player. Each player writes a letter with the information on his cards. Each player retains his cards

in order to prove later on that he really has the cards he claimed. In this way the winner is determined.

How to Verify That No One Has Cheated

For both players there are many possibilities to cheat by deviating from the scheme described. For example, each player could open further envelopes in order to obtain a larger choice for his cards or information on the hand of the opponent. However, this is easily detected if the players meet after the game. The players have to retain their cards and the closed envelopes until this meeting. Then they can present the cards to their opponents and can open the remaining envelopes together in order to verify that they did not cheat.

Discussion

There are several obvious disadvantages in this poker scheme:

- Putting the cards into envelopes can only be done manually and it is expensive. The envelopes cannot be reused.
- The check whether one of the players has cheated cannot be done before the next meeting of the players. For each other game before this meeting they need another card deck with a different seal.
- Snail mail is too slow to make the game interesting and it is more expensive than email.
- If a letter is lost, the game cannot be completed. So, if a player recognizes that he is likely to lose, he could destroy a letter because it is not possible to find out who intercepted the letter.

Now the question arises whether there is some kind of “electronic” envelopes with similar or even better properties than paper envelopes. In particular, they could be produced using a computer and sent by email. This saves the work of putting each card manually into an envelope. Furthermore, emails can be stored by the sender and can be resent if they are lost.

Dealing Cards by Email

Electronic Envelopes

How can we realize envelopes with emails? The first idea is to use codes for the cards. Bob creates and shuffles these codes and sends them to Alice. Since Alice does not know of the correspondence between the codes and the cards, she can select cards without knowing the cards. Before we describe how to use the codes for shuffling and distributing the cards, we focus on the special case that only Bob has to obtain cards. We sometimes assume that the cards

have fixed numbers and that both players know this numbering scheme. We assume that 0 corresponds to the ace of clubs, 1 to the two of clubs, 2 to the three of clubs, ..., 12 to the king of clubs, 13 to the ace of spades, and so on until 51 to the king of diamonds.

How to Shuffle the Cards and Distribute Them to Bob

At the beginning Bob randomly creates codes for the cards, i.e., a table like the following one:

Card	Code	Card	Code
0 (Ace of clubs)	→ 1	5 (Six of clubs)	→ 0
1 (Two of clubs)	→ 42	6 (Seven of clubs)	→ 43
2 (Three of clubs)	→ 22	:	
3 (Four of clubs)	→ 25		
4 (Five of clubs)	→ 51	51 (King of diamonds)	→ 13

The left column of the table is a list of all cards. The right column has been created randomly such that each number between 0 and 51 occurs exactly once. Up to now Alice does not know this table.

In order to select five cards for Bob, Alice randomly chooses five codes between 0 and 51 and send them to Bob. Then Bob uses the table in order to find out his cards. If, for example, Alice has selected the codes 0, 1, 13, 42 and 51, according to the above table Bob gets the six of clubs, ace of clubs, king of diamonds, two of clubs, and five of clubs. Since Alice does not know the table, she cannot influence the choice of the cards for Bob. Furthermore, Alice knows the codes already used such that she can avoid selecting some card for a second time.

The method described is similar to that using the real envelopes. Instead of putting the ace of clubs into a yellow envelope, Bob assigns a number to it; in the example this is the number 1. If envelopes are used, Alice cannot look into them. Here, Alice does not know the meaning of the code 1. In both cases Alice cannot influence the cards selected for Bob.

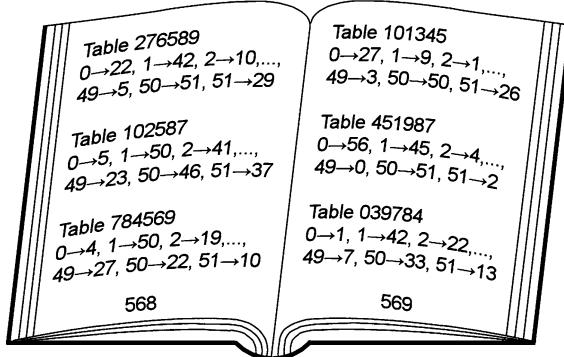
However, at the end of the game, Bob could claim that he has created a different table and in this way he could select better cards for himself. Thus Bob has to fix a table in such a way that he cannot change the table later on. We are going to show how this can be done using one-way functions.

One-Way Functions

One-way functions are presented in Chap. 14. We recall: A one-way function f is a function that can easily be computed, but for which the inverse function f^{-1} is hard to compute. An example in Chap. 14 was a telephone directory. Computing the one-way function f corresponds to finding the telephone number for a given name, which is obviously easy. Computing the inverse function

f^{-1} corresponds to finding the name for a given telephone number, which is obviously difficult.

How can we use one-way functions in order to prevent Bob from changing the table with the codes of the cards? Similarly to the telephone directory, we assume that Alice and Bob have copies of a book with a large number of tables with codings. Furthermore, for each such table there is a unique number in the book.



Then the distribution of cards to Bob can be done in the following way: In the first step Bob randomly selects a table with codings from the book (e.g., the table in the bottom of page 569). Bob sends the number of this table to Alice. In this example this is the number 039784. If Alice would like to find the table used by Bob, she essentially has to scan the whole book. If this takes too much time for her, she has no possibility to find the table with codes used by Bob. Then her only possibility is to select five random number from 0 to 51 and to send them to Bob. Then Bob can obtain his cards according to the table. The card with the number 0 (ace of clubs) has the code 1, the card with the number 1 (two of clubs) has the code 42 and so on. At the end of the game, Bob can send Alice the position of the table in the book. Then Alice knows the table and can verify whether this table has really the number given by Bob at the beginning of the game. Furthermore, she can verify whether Bob really got the cards he claimed.

How to Replace Cards

Now replacing the cards does not require new ideas. If Bob would like to replace the two of clubs in the previous example, he just says to Alice that he would like to replace the card with the code 42. Since Alice does not know that this code corresponds to the two of clubs, she does not know either which card is dropped by Bob. Afterwards, Alice can select a new code in order to send a new card to Bob. Since she knows which codes have already been used, she can avoid distributing a card for a second time. We see that this is even possible without Alice knowing the set of cards she has already distributed.

A Mathematical Description

Now we describe the scheme from above a bit more mathematically. The book with the coding tables corresponds to a one-way function f . This function maps the positions of the coding tables in the book to numbers. In the scheme from above, Bob randomly selects a coding table with the position x and sends $f(x)$ to Alice. Since f is a one-way function, it is hard for Alice to obtain x from $f(x)$. This corresponds to searching the whole book. At the end of the game, she obtains x from Bob. Then Alice can easily compute $f(x)$ and verify that this is the number she got from Bob at the beginning of the game. Thus Bob cannot cheat by the claim that he used some different coding table.

On the other hand, it is easy to store and search a whole book with a computer. Instead of a book we should use one-way functions in order to obtain a number from the coding table, and use this number to commit to this coding table. However, here we would like to skip the details of such one-way functions.

Each coding table can also be considered as a function. Above we already mentioned the numbering scheme for the cards. Then the coding table is a function b that maps the numbers of the cards (from 0 to 51) to their codes (also a number between 0 and 51). The coding table also describes the inverse function b^{-1} which maps each code to the number of the corresponding card. In order to compute $b^{-1}(z)$ we search for z in the right column of the table and obtain the result in the left column. For tables with only 52 entries this is easy to do. Since in the right column each number occurs exactly once, we conclude that for all x we have $b^{-1}(b(x)) = x$.

Distribution of Cards to Both Players

In order to distribute cards to both players, Alice and Bob use separate coding tables which they create independently and which the opponent does not know. We call the function described by Alice's coding table a and the function of Bob's coding table b . Another requirement on a and b is that for all x between 0 and 51 it holds that $a(b(x)) = b(a(x))$. Mathematicians call such functions commuting. For commuting functions a and b we obtain the same result if we apply a on x and afterwards b on the result or vice versa.

An example of commuting functions are the following ones:

$$a(x) = \begin{cases} x + 25, & \text{if } x + 25 < 52, \\ x + 25 - 52, & \text{if } x + 25 \geq 52, \end{cases}$$

and

$$b(x) = \begin{cases} x + 37, & \text{if } x + 37 < 52, \\ x + 37 - 52, & \text{if } x + 37 \geq 52. \end{cases}$$

In this example the coding tables are not given completely. Instead, we give formulas how to compute the entry in the right column from the entry in the left column. It is easy to verify that $a(b(x))$ and $b(a(x))$ coincide: For the computation of $a(b(x))$ we first have to add 37 to x and afterwards 25, where we have to subtract 52, if any of the results is larger than 51. For the computation of $b(a(x))$ we just have to reverse the order of the additions. Instead of 37 and 25 we could also use arbitrary different numbers.

This easy example of commuting functions is obviously not suitable for our application of coding tables. If for example Bob obtains $a(x)$ for the number x , he can easily compute the number 25 which Alice uses for the addition. In this way Bob obtains the whole function a and can break all the codes of Alice. More details on commuting functions suitable for this application are given in the articles mentioned in the section Further Reading.

Commitment to the Selected Coding Tables

Again we use a and b to denote the coding tables selected by Alice and Bob. As above, we use a one-way function f . Alice computes $f(a)$ and sends it to Bob. Similarly, Bob computes $f(b)$ and sends it to Alice. From $f(a)$ and $f(b)$, Bob and Alice, resp., cannot obtain information on the coding table of the opponent because f is a one-way function. On the other hand, the players have committed to the selected coding tables a and b . After the end of the game, Alice sends her table a to Bob. Then Bob can compute $f(a)$ and verify that a is really the table selected by Alice at the beginning of the game. Similarly Alice obtains b from Bob and can verify that b is the table selected by Bob at the beginning.

Putting Cards into Envelopes

If Alice wants to put the card x into an envelope, she only has to compute $a(x)$. In order to remove the card from the envelope $a(x)$, she applies the inverse function a^{-1} to $a(x)$ because $a^{-1}(a(x)) = x$. Similarly, Bob can put cards into envelopes using the function b . Since we used the numbers 0 to 51 for the cards and the codes (i.e. the envelopes) as well, Alice can also put envelopes $b(x)$ created by Bob into her envelopes by computing $a(b(x))$.

In our poker protocol for snail mail, in a first step Bob puts the cards into envelopes and afterwards Alice puts those envelopes into another layer of envelopes. Bob's task corresponds to computing $b(0), \dots, b(51)$. It is easy to see that these are exactly the numbers between 0 and 51. Afterwards Alice had to compute $a(b(0)), \dots, a(b(51))$. The result is again the set of numbers between 0 and 51. Thus Alice and Bob know without any computation that the set of codes $a(b(0)), \dots, a(b(51))$ is the set of the numbers between 0 and 51. Since Alice does not know b , she is not able to obtain a card from any code. Similarly, Bob cannot do this because he does not know a .

Distributing Cards to Alice

Bob selects from the list of unused codes (which consists at the beginning of the number 0 to 51) an arbitrary code for each card that Alice has to get and removes this code from the list. Since he does not know a , he has no influence on the selected card. On the selected code $a(b(x))$ he applies b^{-1} and obtains $b^{-1}(a(b(x))) = b^{-1}(b(a(x))) = a(x)$ because a and b commute and $b^{-1}(b(z)) = z$. Then Bob sends $a(x)$ and $a(b(x))$ to Alice. Alice applies a^{-1} to $a(x)$ and obtains the corresponding card x . Furthermore, she can remove $a(b(x))$ from the list of unused codes. Thus the same card is not distributed again.

Distributing Cards to Bob

Alice selects from the list of unused codes one element for each card Bob has to obtain. Since she does not know b , she cannot influence the selected cards. On each code $a(b(x))$ chosen by Alice she applies a^{-1} and obtains $a^{-1}(a(b(x))) = b(x)$ because of $a^{-1}(a(z)) = z$. Then she sends $b(x)$ and $a(b(x))$ to Bob. From $b(x)$ Bob computes x , i.e., the number of the selected card. Furthermore, he can remove $a(b(x))$ from the list of unused codes.

Dropping Cards

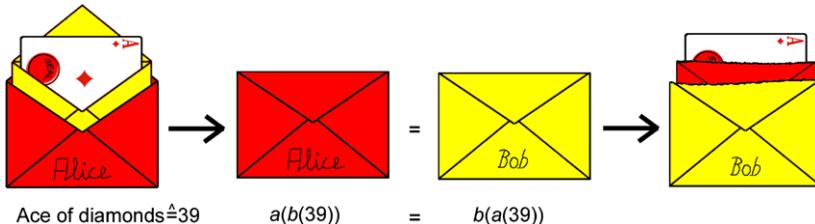
In order to drop the card x without sending any information on x to the opponent, the player sends a message with the code $a(b(x))$. We already know that the opponent cannot obtain any information from $a(b(x))$.

Properties of the Electronic Envelopes

We compare paper envelopes with their electronic counterparts: First we look at the selection of the cards for Alice. Putting the cards into yellow envelopes by Bob corresponds to applying the function b . Putting those envelopes into red envelopes by Alice corresponds to applying the function a . After selecting the cards for Alice, in the electronic version of envelopes Bob can remove the interior yellow envelopes by applying b^{-1} without opening the outer red envelopes and without getting any knowledge of the contents of the yellow envelopes. Afterwards, Alice can open the red envelopes by applying a^{-1} . Thus the electronic envelopes have some properties that paper envelopes cannot have:

- Alice cannot open the yellow envelopes, and Bob cannot open the red ones. In particular, after the end of the game it is no longer necessary to verify that the players did not open the unused envelopes because they just cannot do this.

- It is possible to copy envelopes together with their contents and without knowing the contents or obtaining any information about it.
- A red envelope with a yellow envelope and a particular card inside is identical to a yellow envelope with a red envelope and the same card inside. Thus it is possible to remove the interior yellow envelope without destroying the outer red envelope.



How to Check Whether the Opponent Has Cheated

At the end of the game, the players send their coding tables a and b to their opponent. Then they can compute $f(a)$ and $f(b)$, resp., and verify that a and b are really the coding tables the other player has committed to. Using the coding tables the players can verify all computations. During the game the player have to cooperate to open a particular envelope because it is necessary to evaluate a^{-1} and b^{-1} to open an envelope. Thus a meeting of the players in order to examine the unused envelopes is no longer necessary.

Poker with More than Two Players

Up to now we only considered the situation of two players. What happens for three or more players? Of course, we could try to generalize the above ideas to a larger number of players. However, there is a fundamental problem. Our starting-point was that the players cannot observe each other. How can a third player prevent Alice and Bob from exchanging information about their cards using their mobile phones? Commercial online poker systems have the possibility to arrange groups of players that do not know of each other. Another possibility would be the analysis of the behavior of the other players, e.g., whether the player with the worse cards always refrains from bidding. Nevertheless, it is hard to prove that the other players are cheating. So if we would like to play with some larger number of players, we should meet, which also might be more appealing.

Further Reading

1. Adi Shamir, Ronald L. Rivest and Leonard M. Adleman: *Mental Poker*. In: *The Mathematical Gardner*, pp. 37–43. Edited by David A. Klarner,

Wadsworth International, 1981. Available at: <http://people.csail.mit.edu/~rivest/ShamirRivestAdleman-MentalPoker.pdf>

In this article a protocol for poker was presented for the first time. The protocol presented above is a modification of that one.

2. Bruce Schneier: *Applied Cryptography*. Wiley, 1996.

In this book the protocol of Shamir, Rivest and Adleman for poker as well as several other protocols for other tasks are described.

The playing cards in the figures are designed by David Bellot. They are available at <http://svg-cards.sourceforge.net/> according to the LGPL (<http://www.gnu.org/copyleft/lesser.html>).

Fingerprinting

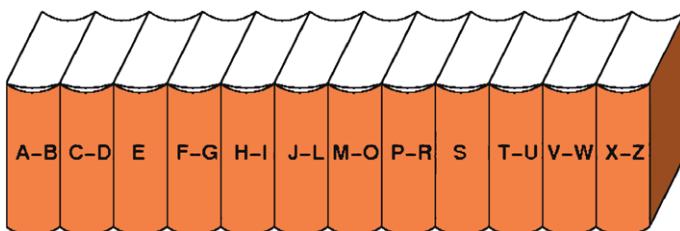
Martin Dietzfelbinger

Technische Universität Ilmenau, Ilmenau, Germany

How to Compare Long Texts over the Telephone

Alice and Bob are very good friends. Alice lives in Adelaide in Australia, and Bob lives in Barnsley in (Great) Britain. They love to talk to each other on their cellphones, but this can cost quite some money when they talk for long. The two of them share many interests, and they are interested in lots of things. So it comes as no surprise that they both buy an encyclopedia. They tell each other about their acquisition on the phone. What a surprise – they both bought the same encyclopedia! On the phone, Alice asks Bob a simple question: Do both copies contain exactly the same text? Are they really identical, word by word, comma by comma?

Even if Alice and Bob figure out that both have bought the 37th edition of the encyclopedia, say, this does not necessarily mean that the copy that was printed in Australia contains exactly the same text as the one that was printed in England. Maybe some misprints were corrected? How can they find out whether both copies are really identical? Alice could read her copy to Bob, on the phone. Bob would read along and compare each word and each punctuation mark. This would do the job, but it would cost a hell of a lot in telephone charges (at least if the two use their cellphones), because Alice and Bob have chosen quite a voluminous encyclopedia: 12 volumes, about 1,500 pages per volume, about 2,800 characters per page; all in



all about 18,000 pages and about 50 million characters. If Alice needs only five minutes to read one page, and they work on it without any breaks, the two of them and the phone tariff unit counter will be busy for more than 60 days.

In a computer, we may represent a character, including commas, spaces, and line and page breaks, by a binary code, e.g., with eight bits (one byte) per character. For the whole encyclopedia this will amount to 50 million bytes, or about 50 Megabytes. Let us assume that Alice and Bob have individually managed to enter the text of their respective copies into a computer. As soon as the encyclopedia is stored electronically, it is actually no big deal to send this amount of data from Australia to England by e-mail. For the sake of argument, let us assume however that the connection is either very expensive or very error-prone, so that the transmission of such a large amount of data via e-mail is impossible or undesirable.

Is there a way for Alice and Bob to find out whether the two texts are identical without comparing them character by character? They would prefer to carry out any conversation needed by cellphone, and hence keep the amount of information that has to be exchanged very small.

If you sometimes send large files via e-mail or if you ever found it necessary to make room on your brimful hard disk, you know that there are methods that do something called “data compression.” By such methods, one “squeezes together” data such as texts or images, so that they take up less space and transmission times are shorter. Alice and Bob could use such methods. But even if they manage to achieve a compression to, say, a fifth of the original length of their data, it would still take too long to carry out the communication. So, data compression does not solve the problem.

Here is a very simple observation: Alice should start by counting the characters in her encyclopedia. Let us denote the result by n . Alice tells Bob what n is, which means reading out eight decimal digits, which is a matter of seconds. Bob also has counted the characters in his copy, with a result of n' . If n and n' are different, Bob can announce that the texts are different. (We assume that Alice and Bob haven't miscounted.) From now on we may assume the texts of Alice and Bob have exactly the same length.

Texts as Sequences of Numbers and Modular Arithmetic

Our long-term goal is to find a trick that makes it possible to solve the text comparison problem with very short messages. For this, we want to translate texts into numbers and then calculate with these numbers. A little basic work is needed for this. We already noted that in a computer each character is represented by a sequence of eight bits, that is, a byte. A standard way of doing this is given by the ASCII code. In this code, the characters A, B, C, ... look like this: 01000001, 01000010, 01000011, etc. These bit patterns can also

be regarded as binary representations of numbers. In this way, we reach the following way of coding characters by numbers:

A	B	C	...	Z	a	b	c	...	z
65	66	67	...	90	97	98	99	...	122

The punctuation marks are assigned numbers as well: for example, the exclamation mark (“!”) has number 33 and the space (“ ”) has number 32. In this way, every character is represented by a number between 0 and 255. The text

Alice and Bob have a chat.

including spaces and the period translates into

65 108 105 99 101 32 97 110 100 32 66 111 98 32
104 97 118 101 32 97 32 99 104 97 116 46,

which we write “mathematically” as a sequence

(65, 108, 105, 99, 101, 32, 97, 110, 100, 32, 66, 111, 98, 32,
104, 97, 118, 101, 32, 97, 32, 99, 104, 97, 116, 46).

Now we can imagine that Alice has translated her whole encyclopedia into one long sequence

$$T_A = (a_1, a_2, \dots, a_{n-1}, a_n)$$

of numbers between 0 and 255, and that Bob has done the same with his copy:

$$T_B = (b_1, b_2, \dots, b_{n-1}, b_n).$$

Here n is about 50 million, and you see that it is impossible to write down sequences as long as that in this little book. As a manageable example we take two sequences of length $n = 8$:

Texts (“Adelaide” and “Barnsley”) as sequences of numbers

$$\begin{aligned} T_{\text{Ad}} &= (a_1, a_2, \dots, a_8) = (65, 100, 101, 108, 97, 105, 100, 101) \\ T_{\text{Ba}} &= (b_1, b_2, \dots, b_8) = (66, 97, 114, 110, 115, 108, 101, 121) \end{aligned}$$

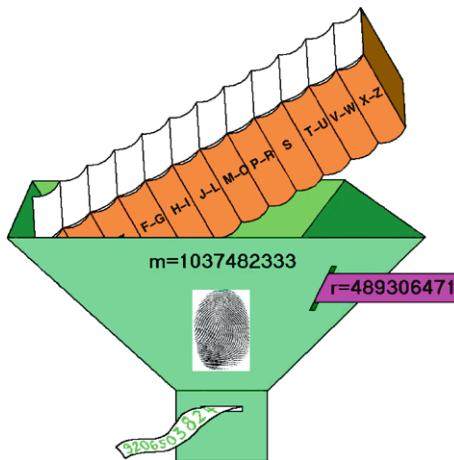
Now we want to calculate with these number sequences. For this we need a method that has been mentioned already in Chap. 17 and will be explained in more detail in Chap. 25: “modular arithmetic.” Taking an arbitrary integer a modulo an integer $m > 1$ simply means that one calculates the remainder when a is divided by m ; in other words, one counts how many steps one has to walk to the left on the number line, starting at a , until one hits a multiple of m . We write $a \bmod m$ for this number. For example, if $m = 7$, then $16 \bmod 7 = 2$ and $-4 \bmod 7 = 3$. In the following table you find some more values of $a \bmod 7$. You will spot the pattern at once: if you walk along the number line to the right, the values of the remainders $a \bmod m$ run around $\{0, 1, \dots, m - 1\}$ in repeating circles.

a	...	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9	10	...
$a \bmod 7$...	3	4	5	6	0	1	2	3	4	5	6	0	1	2	3	...

“Modular arithmetic” then means that numbers are added and multiplied “modulo m ,” which is done as follows. One adds and multiplies as usual and then calculates the remainder of the result when divided by m . For example, $3 \cdot (-6) \bmod 7 = (-18) \bmod 7 = 3$. To make longer calculations easier, one may replace any intermediate result by its remainder modulo m . For example, to calculate $(6 \cdot 5 + 5 \cdot 4) \bmod 7$, one gets $6 \cdot 5 \bmod 7 = 30 \bmod 7 = 2$ in a first step and $5 \cdot 4 \bmod 7 = 20 \bmod 7 = 6$ in a second, and obtains the final result as $(2 + 6) \bmod 7 = 8 \bmod 7 = 1$.

Fingerprints

Now we apply modular arithmetic to our texts T_{Ad} and T_{Ba} . We fix some number m . Later we will see that m should be a prime number larger than 255 and larger than n , maybe about as large as $2n$ or $10n$. In order to keep the numbers in the example calculation small and simple, we choose $m = 17$.



For $r = 0, 1, 2, \dots, m - 1$ and a text $T = (a_1, a_2, \dots, a_{n-1}, a_n)$ we look at the following number:

$$\text{FP}_m(T, r) = (a_1 \cdot r^n + a_2 \cdot r^{n-1} + \dots + a_{n-1} \cdot r^2 + a_n \cdot r) \bmod m.$$

Example: For T_{Ad} and $r = 3$ we obtain:

$$\begin{aligned} \text{FP}_m(T_{\text{Ad}}, 3) &= (65 \cdot 3^8 + 100 \cdot 3^7 + 101 \cdot 3^6 + 108 \cdot 3^5 + 97 \cdot 3^4 + 105 \cdot 3^3 \\ &\quad + 100 \cdot 3^2 + 101 \cdot 3) \bmod 17. \end{aligned}$$

One should notice right from the start that the length n of the text may be large, but the number of digits of m and hence the number of digits of $\text{FP}_m(T, r)$ is really small. We call the number $\text{FP}_m(T, r)$ (you should imagine it is written in decimal or in binary) a “*fingerprint*” of the text $T = (a_1, a_2, \dots, a_n)$, calculated with respect to r .

The name “*fingerprint*” tries to convey the idea that the number $\text{FP}_m(T, r)$ stores in little space some information about T that makes it possible to distinguish T from other texts, in a way similar to that in which a little fingerprint is enough to distinguish one human being from another. Of course, the length n of T can also be counted as a (very rudimentary) “*fingerprint*.”

Calculating $\text{FP}_m(T, r)$ at first glance looks like quite an adventure, in particular for the long texts in which we are really interested, because the high powers of r will be extremely large numbers. A simple trick, namely, factoring out in a clever way, helps us to eliminate this problem:

$$\text{FP}_m(T, r) = (((\cdots (((a_1 \cdot r) + a_2) \cdot r) + \cdots) \cdot r + a_{n-1}) \cdot r + a_n) \mod m.$$

If this expression is evaluated in the usual manner, starting from the inside, working outwards, and taking remainders modulo m after each step, the intermediate results stay small. For example, we have

$$\begin{aligned} \text{FP}_m(T_{\text{Ad}}, 3) &= (((((((((65 \cdot 3) + 100) \cdot 3 + 101) \cdot 3 + 108) \cdot 3 + 97) \cdot 3 \\ &\quad + 105) \cdot 3 + 100) \cdot 3 + 101) \cdot 3) \mod 17, \end{aligned}$$

and with $r = 3$ the single steps are the following:

	Values	Intermediate result	
a_1	65	$(65 \cdot 3) \mod 17 = (14 \cdot 3) \mod 17 = 8$	
a_2	100	$((8 + 100) \cdot 3) \mod 17 = (6 \cdot 3) \mod 17 = 1$	
a_3	101	$((1 + 101) \cdot 3) \mod 17 = (0 \cdot 3) \mod 17 = 0$	
a_4	108	$((0 + 108) \cdot 3) \mod 17 = (6 \cdot 3) \mod 17 = 1$	
a_5	97	$((1 + 97) \cdot 3) \mod 17 = (13 \cdot 3) \mod 17 = 5$	
a_6	105	$((5 + 105) \cdot 3) \mod 17 = (8 \cdot 3) \mod 17 = 7$	
a_7	100	$((7 + 100) \cdot 3) \mod 17 = (5 \cdot 3) \mod 17 = 15$	
a_8	101	$((15 + 101) \cdot 3) \mod 17 = (14 \cdot 3) \mod 17 = 8$	

Thus, $\text{FP}_{17}(T_{\text{Ad}}, 3) = 8$.

Now we can formulate the algorithm for calculating a fingerprint $\text{FP}_m(T, r)$:

Algorithm FP calculates $\text{FP}_m(T, r)$

```

1   procedure FP( $m, T, r$ )
2   begin
3       fp := ( $a_1 \cdot r$ ) mod  $m$ ;
4       for  $i$  from 2 to  $n$  do
5           fp := ((fp +  $a_i$ ) ·  $r$ ) mod  $m$ ;
6       endfor
7   return fp
8   end

```

Clearly, the resulting fingerprint is a remainder modulo m and hence a number between 0 and $m - 1$, and the intermediate results are always smaller than m^2 .

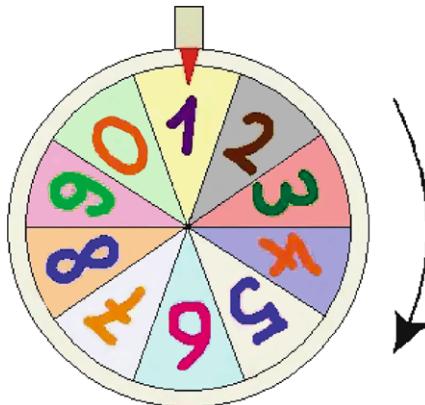
Let us use this procedure to calculate all $m = 17$ fingerprints for T_{Ad} and for T_{Ba} :

r	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$\text{FP}_m(T_{\text{Ad}}, r)$	0	12	7	8	11	14	15	5	11	1	2	12	13	13	6	6	0
$\text{FP}_m(T_{\text{Ba}}, r)$	0	16	9	2	2	6	14	3	11	12	2	10	11	15	2	10	11

(Alice could calculate the entries in the first row, Bob the entries in the second row.) We compare the values for T_{Ad} and for T_{Ba} . For $r = 0$ we see a 0 in both rows – no surprise, since in Algorithm FP the last operation is a multiplication by r . A fingerprint with $r = 0$ does not contain any information, and we need not consider $r = 0$ at all. Apart from that, it is hard to spot a pattern in the sequences of numbers in the two rows. Let us compare numbers that appear in the same column. For $r = 3$ (the example from above) we have $\text{FP}_{17}(T_{\text{Ad}}, r) = 8$ and $\text{FP}_{17}(T_{\text{Ba}}, r) = 2$. Note that knowing these two fingerprints would allow Bob to decide on the spot that the texts cannot be the same. For $r = 8$ and $r = 10$, on the other hand, the results are the same (11 and 2, respectively) – these values of r do not help.

Fingerprints with Random Numbers

Here is the central idea that gets the approach off the ground. Why should Alice and Bob calculate *all* values in the table? (Thinking of larger values of n and m , they will not be able to do this anyway, as there is not enough time.) However, Alice chooses a number r between 1 and $m - 1$ *at random*. For example, in order to determine the decimal digits of r , she could repeatedly turn a “wheel of fortune” whose circumference is subdivided into ten segments of equal length:



(In every programming language there is an operation for generating “random numbers.” Chapter 25 deals with the question of what mechanisms are behind these “random number generators.”) Alice calls Bob and tells him which number she has chosen. For this, she has to tell him only very few decimal digits. Then they may hang up. Alice calculates the number $\text{FP}_m(T_A, r)$ – or, rather, lets her computer calculate this number. Simultaneously, Bob calculates $\text{FP}_m(T_B, r)$. This may take a while, but no communication is necessary, and hence there are no phone charges. As soon as both are finished, Alice calls Bob again and tells him “her” fingerprint $\text{FP}_m(T_A, r)$. Now there are several possibilities.

Case 1: The texts T_A and T_B are equal. Then Alice and Bob will have obtained the same result, no matter which r Alice had chosen.

Case 2: The texts T_A and T_B are different (in the example, “Adelaide” and “Barnsley” for $m = 17$).

- If Alice picked a number r with $\text{FP}_m(T_A, r) = \text{FP}_m(T_B, r)$ (in the example, $r = 8$ or $r = 10$), then Bob will obtain the same fingerprint as Alice, and it will look for both of them as if the texts could be equal.
- If Alice picked a number r with $\text{FP}_m(T_A, r) \neq \text{FP}_m(T_B, r)$ (in the example, one of the other 14 numbers), then Bob will obtain a fingerprint that is different from Alice’s and will be able to announce that with certainty the texts are different.

In our simple example the chances that Alice and Bob find the difference is 14 : 16 or 87.5 percent. What are the chances in the general case for noticing the difference? In order to be able to say something about this, we must rummage a little more deeply in the toolbox provided by number theory, a part of mathematics that comes in handy also when one wants to encrypt a text (see Chap. 16). One can prove that the following is always true.

Fingerprinting Theorem

If T_A and T_B are different texts (sequences of numbers) of length n , and if m is a prime number larger than all numbers in T_A and T_B , then at most n out of the m pairs

$$\text{FP}_m(T_A, r), \text{FP}_m(T_B, r), \quad r = 0, 1, \dots, m - 1,$$

can consist of two equal numbers.

This mathematical fact has been known for centuries. Prime numbers have lots of wonderful properties – this fact belongs to the simpler ones. How one proves the Fingerprinting Theorem is not really important for Alice and Bob and for our considerations here, since the proof does not figure in the algorithm at all. We postpone a sketch of the proof to the last section of this chapter, and first look at the way in which it helps Alice and Bob solve their problem.

For our example with $n = 8$ the theorem means the following: No matter what T_A and T_B look like, if they are different, then in our table there will never be more than seven values $r \neq 0$ that make Alice and Bob calculate the same fingerprint. This means that their chance of noticing the difference is always $9 : 16$, or more than 50 percent. But wait a second! This is not quite true. In the example we chose for m a very small number, which is not larger than 255 (the largest possible number that may appear in a text), as required in the Fingerprinting Theorem. So, the conclusion that the odds of noticing the difference between T_A and T_B are larger than 50 percent is only true for pairs T_A and T_B that satisfy $a_i \bmod 17 \neq b_i \bmod 17$ for at least one character position i . This problem disappears when m is chosen to be larger than 255.

Now let us try our technique on the original problem with texts of length about 50 million characters. In order to obtain something useful, Alice and Bob must choose their prime number m somewhat larger than 50 million – let us say they choose $m = 1,037,482,333$. (Such prime numbers, and much larger ones, can be found in tables on the Internet.) For numbers n and m of that size we definitely do not want to write down the table of all $\text{FP}_m(T, r)$ values. But by the Fingerprinting Theorem we *know* that if we did, then among the m columns there would never be more than n many in which the numbers $\text{FP}_m(T_A, r)$ and $\text{FP}_m(T_B, r)$ are the same. (One of those is the column for $r = 0$.)

Now, if Alice chooses r at random from the numbers between 1 and $m - 1$ and Alice and Bob calculate and tell each other numbers and fingerprints as just described, then the probability that Alice happens to choose one of the “bad” values for r and they fail to notice that T_A and T_B are different is at

most

$$\frac{n-1}{m-1} \approx \frac{50000000}{1000000000} = 0.05,$$

or 5 percent. The chance that they discover that the texts are different is at least 95 percent!

What about communication cost? Although Alice and Bob have to calculate a lot (or have their computers compute a lot), they need to exchange only very little information: Alice has to tell Bob her character count n (eight decimal digits) and the prime number m (ten digits), and she has to tell him the two numbers r and $\text{FP}_m(T_A, r)$ (20 digits).

Alice and Bob can achieve an error probability of less than 5 percent by communicating fewer than 40 decimal digits!

This means that what seemed impossible at the beginning – that one could compare extremely long texts by some phone calls that do not last longer than a minute – is indeed feasible.

Maybe Alice and Bob are not satisfied with an error probability of 5 percent, and insist that it must be much smaller. In this case there are improvements to the algorithm that are not really much more expensive in terms of communicated digits. Alice chooses two numbers r_1 and r_2 at random and tells Bob these two numbers and the corresponding fingerprints $\text{FP}_m(T_A, r_1)$ and $\text{FP}_m(T_A, r_2)$. Bob declares the two texts to be equal (with a little risk of being wrong) if he obtains the same two fingerprints for T_B . The chance that Bob declares the texts to be equal while in fact they are not is no larger than

$$\frac{(n-1)^2}{(m-1)^2} < \left(\frac{n}{m}\right)^2 \approx 0.05^2 = 0.0025,$$

the chance that the difference is noted is at least 99.75 percent. If Alice even sends three number pairs (this means the total communication amounts to fewer than 80 digits), the error probability drops to $(n^3/m^3) \approx 0.000125$ or 0.0125 percent; the chance of detecting a difference increases to 99.9875 percent.

The Protocol

We summarize Alice and Bob's method to check whether their texts are identical. As this method involves both computation and communication, one does not call this an “algorithm” but, rather, a “protocol” (in the sense of a set of rules that say who has to do what and when).

Protocol Text Comparison by Fingerprinting

Alice has the sequence $T_A = (a_1, \dots, a_n)$ of numbers between 0 and $d - 1$. Bob has the sequence $T_B = (b_1, \dots, b_{n'})$ of numbers between 0 and $d - 1$.

1. Alice tells Bob what n is. If $n \neq n'$, Bob says “different,” and STOP.
2. Alice and Bob agree on a number k of repetitions.
3. Alice finds some prime number m a little larger than d and $10n$.
She chooses k numbers r_1, \dots, r_k between 1 and $m - 1$ at random, and tells Bob m and r_1, \dots, r_k .
4. Alice calculates $\text{FP}_m(T_A, r_1), \dots, \text{FP}_m(T_A, r_k)$.
(For this, she modifies algorithm FP in such a way that the text T_A is only run through once to calculate all k results.)
5. Bob calculates $\text{FP}_m(T_B, r_1), \dots, \text{FP}_m(T_B, r_k)$ in the same way.
6. Alice tells her k results to Bob.
7. Bob compares with his k values.
If there are differences, he says “different,” and STOP.
If all values are equal, he says “can’t see a difference,” and STOP.

One can say the following about the result of the protocol.

- If Alice and Bob have the same text, then the sequences of k fingerprints they calculate are the same. Hence the result always is “can’t see a difference.”
- If Alice and Bob have different texts (of the same length), then by the Fingerprinting Theorem, among the $m - 1$ numbers Alice chooses from, there are at most $n - 1$ many values for r that make $\text{FP}_m(T_A, r)$ and $\text{FP}_m(T_B, r)$ coincide. For r randomly chosen, the probability that $\text{FP}_m(T_A, r)$ and $\text{FP}_m(T_B, r)$ are equal is at most $(n - 1)/(m - 1)$. The probability that Alice chooses such “bad” r ’s in all k trials, making Bob say “can’t see a difference” erroneously, is no larger than

$$\frac{(n - 1)^k}{(m - 1)^k} = \left(\frac{n - 1}{m - 1}\right)^k < \left(\frac{n}{m}\right)^k.$$

Since we have assumed that $m \geq 10n$, the bound is smaller than $1/10^k$, and by choosing k large enough Alice and Bob can adjust the error probability bound to as tiny a value as they wish.

If $m \approx 10n$, and n has exactly l decimal digits, and Alice and Bob want the error probability to be at most 10^{-k} , it is sufficient that Alice communicate $(l + 1) \cdot (2 + 2k)$ digits. It is astonishing that this figure changes only very slowly when the length of the text is increased: When comparing texts that are ten times as long, i.e., n increases by a factor of ten, the number of digits that have to be communicated increases only by $2k$.

Summary

- If one wants to have absolute certainty when comparing two texts, one may use “lossless data compression” techniques, but normally it will not be possible to save more than a factor of 5 or so over the full length of the text.
- If it is acceptable that one erroneously comes to the conclusion that two texts are the same with a (very) small probability, fingerprinting techniques can be employed. This will dramatically reduce the length of the messages to be transmitted.
- For texts of length n , a prime number $m > n$ is used. When sending k fingerprints, the error probability is no larger than $(\frac{n}{m})^k$. In this case, $2k + k$ numbers not larger than m must be transmitted.
- Using randomness in algorithms and communication protocols can lead to significant savings in resources such as storage space or transmission time if it is acceptable that with some small probability a wrong result occurs. More often than not, by simple means (such as repeating the algorithm/protocol) the error probability can be made so small that errors can be practically eliminated.
- Algorithms and protocols that use randomness to make some decisions or choices are called “randomized.” In Chap. 25 it is discussed how “randomness gets into the computer,” i.e., how one gets the computer to produce “random” numbers.
- Sometimes very abstract mathematical facts that at first glance look nice but do not seem to have a practical value can be utilized in order to save computing cost, storage space, or communication cost.

Remarks on the Fingerprinting Theorem

For the mathematically interested reader, we now give an informal argument that makes it plausible for the Fingerprinting Theorem to be true.

For this, we consider “*polynomials*,” more precisely “*rational polynomials*.” These are expressions

$$f(x) = c_n x^n + c_{n-1} x^{n-1} + \cdots + c_1 x + c_0$$

with a “*variable*” x , where the “*coefficients*” $c_n, c_{n-1}, \dots, c_1, c_0$ are rational numbers, or fractions p/q , with p, q integers and $q > 0$. For example, the following expressions are rational polynomials:

$$2x^2 + \frac{3}{2}, \quad \frac{3}{4}x - \frac{1}{10}, \quad x^5 + 4x^4 - 3x^2 - \frac{15}{29}x + \frac{1}{3}, \quad \frac{7}{8}, \quad 0.$$

In the second to last example $(\frac{7}{8})$ we have $n = 0$ and $c_0 = \frac{7}{8}$; in the last example (0) there are no nonzero terms at all. When writing polynomials, terms $c_i x^i$ with $c_i = 0$ are usually omitted.

Polynomials may be added and subtracted by applying the usual rules:

$$(2x^2 + \frac{3}{2}) + (-3x^2 + \frac{3}{4}x - 1) = (2 - 3)x^2 + \frac{3}{4}x + (\frac{3}{2} - 1) = -x^2 + \frac{3}{4}x + \frac{1}{2},$$

$$(2x^2 + \frac{3}{2}) - (-3x^2 + \frac{3}{4}x - 1) = (2 + 3)x^2 - \frac{3}{4}x + (\frac{3}{2} + 1) = 5x^2 - \frac{3}{4}x + \frac{5}{2}.$$

If one subtracts a polynomial $f(x)$ from itself, the result is the “zero polynomial”: $f(x) - f(x) = 0$. Of course, one may also multiply polynomials. For this, one expands the product by the distributive law and then collects coefficients that belong to the same power of x . Here is an example:

$$(2x^2 + \frac{3}{2}) \cdot (\frac{3}{4}x^3 - x) = \frac{3}{2}x^5 + \frac{9}{8}x^3 - 2x^3 - \frac{3}{2}x = \frac{3}{2}x^5 - \frac{7}{8}x^3 - \frac{3}{2}x.$$

Another important operation with polynomials is “substitution”: If $f(x)$ is a polynomial and r is a rational number, we write $f(r)$ for the result that is obtained by substituting r for x in $f(x)$ and evaluating. That means that if $f(x) = x^3 - \frac{1}{2}x^2 + 2x - 1$, then $f(0) = -1$ and $f(\frac{1}{2}) = 0$ and $f(1) = \frac{3}{2}$. A rational number r is called a *root* of a polynomial $f(x)$ if $f(r) = 0$. For example, $r = \frac{1}{2}$ is a root of $f(x) = x^3 - \frac{1}{2}x^2 + 2x - 1$.

Of course, the zero polynomial 0 has infinitely many roots, namely, all rational numbers. The polynomial $f(x) = 10$ has no roots at all, and the polynomial $2x + 5$ has exactly one root, namely $r = -\frac{5}{2}$. The polynomial $x^2 - 1$ has two roots, namely 1 and -1 , and the polynomial $x^2 + 1$ has no (rational) roots. One can prove the following:

Theorem (Number of Roots of Rational Polynomials)

If $f(x) = c_n x^n + c_{n-1} x^{n-1} + \cdots + c_1 x + c_0$ with $n \geq 0$ and $c_n \neq 0$ is a polynomial, then f has at most n distinct roots.

(Roughly, the reason for this is the following: If r_1, \dots, r_k are k distinct roots of $f(x)$, one can write $f(x)$ as a product $(x - r_1) \cdots (x - r_k) \cdot g(x)$, for some polynomial $g(x) \neq 0$. Since the highest power of x in $f(x)$ is x^n , the number k cannot be larger than n .)

From the theorem about the roots we can conclude the following: If

$$g(x) = c_n x^n + c_{n-1} x^{n-1} + \cdots + c_1 x + c_0 \quad \text{and}$$

$$h(x) = d_n x^n + d_{n-1} x^{n-1} + \cdots + d_1 x + d_0$$

are different polynomials, then there are at most n different numbers r that satisfy $g(r) = h(r)$. Why is that so? Consider the polynomial

$$\begin{aligned} f(x) &= g(x) - h(x) \\ &= (c_n - d_n)x^n + (c_{n-1} - d_{n-1})x^{n-1} + \cdots + (c_1 - d_1)x + (c_0 - d_0). \end{aligned}$$

It could happen that $c_n = d_n$, and $c_{n-1} = d_{n-1}$, and so on, so that many coefficients in $f(x)$ are 0. But since $g(x)$ and $h(x)$ are different, $f(x)$ cannot be the zero polynomial, and we can write $f(x) = e_k x^k + \cdots + e_1 x + e_0$, with $0 \leq k \leq n$ and $e_k \neq 0$. By the theorem on the number of roots, we conclude

that $f(x)$ does not have more than $k \leq n$ roots. But for each r we have that if $g(r) = h(r)$, then $f(r) = g(r) - h(r) = 0$, and hence r is a root of $f(x)$. Hence, there cannot be more than n numbers r with $g(r) = h(r)$.

Hey – this is almost the formulation of the Fingerprinting Theorem! The only difference is that in the Fingerprinting Theorem we are talking about calculations modulo some prime number m , and here we consider calculations with rational numbers. However, for the theorem about the number of roots of a polynomial to be true we do not really need the rational numbers, but only a domain of numbers in which we can add, subtract, multiply, and divide (by any element that is not zero). One can show that arithmetic modulo m has this property if and only if m is a prime number. This means that the theorem about the number of roots holds in modular arithmetic modulo a prime m as well.

Further Reading

1. J. Hromkovič: *Design and Analysis of Randomized Algorithms*. Springer, Berlin, 2005.

This book describes many randomized algorithms, protocols, and methods as well as the fundamentals of the design and the study of such methods.

2. For readers who want to know all the details, a complete proof of the Fingerprinting Theorem from first principles can be found on the page <http://eiche.theoinf.tu-ilmenau.de/fingerprint/>
3. A complete description of the ASCII code, and much more information about it, can be found at <http://en.wikipedia.org/wiki/ASCII>
4. A list of prime numbers with size about 2^k , $1 \leq k \leq 400$: <http://primes.utm.edu/lists/2small/0bit.html>

Acknowledgement

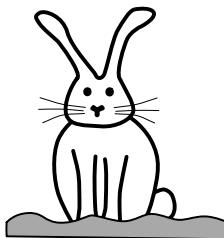
Many thanks to J.D. for help with the illustrations.

Hashing

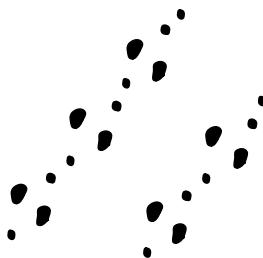
Christian Schindelhauer

Albert-Ludwigs-Universität Freiburg, Freiburg, Germany

Some Germans have problems distinguishing between *hashing* and *Häschen* (bunny).



Bunnies are hard to find. These shy animals hide very well. A careful observer may spot the following tracks on a snow-covered field:



Here, two bunnies hopped next to each other in the snow. A track can tell a lot about an animal: The size and weight, whether it travels in a group and much more. Sometimes a track comes with something like this:



Such droppings are the excrement of a bunny or hare. The droppings tell us what the animal has eaten, whether it is healthy and many other things. Nowadays each individual animal can be identified from it using complicated lab tests. This method may serve also for other animals (and in Madrid canine environmental polluters are identified this way).

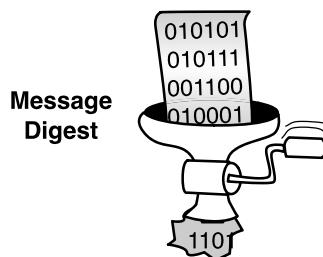
Message Digest

What is the connection to algorithms? Well, the droppings come from the food eaten by the bunny.



The food is *hashed*, mixed, digested, dehydrated and deposited; the result is a small heap. This product can be mapped back to the food. A lot of information is lost: the (digestive) path, yet the food can still be identified.

Something similar can be done with text documents, music files and video files. For computer engineers these are just series of bits, being either zero or one. Such a sequence is mixed, merged and compressed into a bit sequence of fixed length. We say the bit sequence is *hashed*. This may look like this.



A well-known algorithm for hashing is MD-5 (Message Digest 5). It was developed in 1991 by Ronald Rivest, and an accurate description can be found in Wikipedia (<http://en.wikipedia.org/wiki/MD5>). Other known algorithms are Secure Hash Algorithms 1, 224, 256, 284, and 512 (SHA-1, SHA-224, SHA-256, SHA-384 and SHA-512 – <http://en.wikipedia.org/wiki/SHA-1>). They serve the same purpose as the message digest algorithm.

Secure Hashing

What purpose do these hash algorithms serve? First, they map files of different lengths onto bit sequences of the same length. Second, the result helps to identify the original files, but does not necessarily allow us to reconstruct the original. The mathematical description of the identifying feature follows.

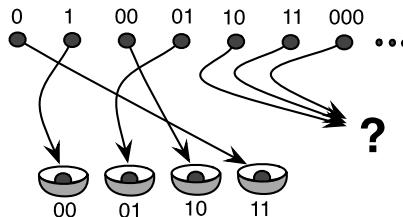
By $\{0,1\}^*$ we describe the set of all bit sequences, which is $\{0, 1, 00, 01, 10, 11, 0000, 0001, \dots\}$, as well the empty bit sequence, which is no bit at all. By $\{0,1\}^k$ we describe the set of all bits of length exactly k bits: $\{000, 001, 010, 011, 100, 101, 110, 111\}$ in the case $k = 3$. A hash function is a mapping:

$$f : \{0,1\}^* \rightarrow \{0,1\}^k.$$

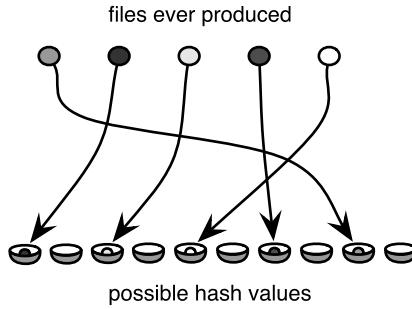
What does it mean that the result of a hash function identifies the input? It means that $f(x) = f(y)$ if and only if $x = y$. The opposite statement is that there are two different values x and y such that $f(x) = f(y)$.

Such an equation is called a collision. We claim that a hash functions exists where all infinitely many bit sequences never collide. Mathematically this is complete rubbish. Why?

Imagine as many bins as we have bit sequences of length k . So we have 2^k bins: two possibilities for each bit. We claim that in every bin $f(x)$ only one original bit sequence is deposited. However, there are many more than 2^k sequences, namely infinitely many. So, at least one of the bins experiences infinitely many collisions.



But there is a loophole! We choose k so large that we have enough bins for every file that ever may be created. So that for each ever-to-be-created file on each ever-to-be-built computer on this planet one hash value is reserved. This trick is used in Chap. 14 where one-way-functions are presented. For example, choose $k = 512$. Then there are $2^{512} > 10^{154}$ bins. If we succeed in filling these bins such that nobody (human or machine) can construct a collision, we are done.



To this day it is not clear whether such a feat is possible, although we have candidates for hash functions like SHA-512 (512 bits for each hash value). But this is of little value. For example, MD-5 is seen as collision free for all existing files until a method is found to produce arbitrarily many collisions.

Practically collision-free hash functions are so useful that computer engineers choose to work with these candidate functions. For example, they can be used to prove the correctness of file transmissions. For this, the hash value can prove that no single bit was lost or altered by the transmission, nor did somebody exchange the whole message.

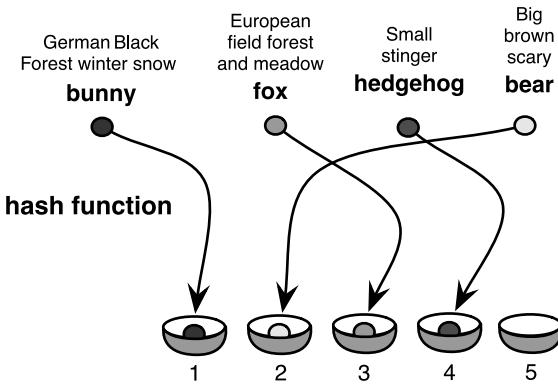
Hashing for Dictionaries

Hash functions can be used not only for storing data hash functions. Consider a storage S for storing m data items. This storage is organized as a table or array. So, the storage positions $S[1], \dots, S[m]$ can be directly accessed to retrieve a data item. We want to save the data items into S , which is identified by a bit sequence (or a text).

For example,

Name	Total number
German Black Forest winter snow bunny	12
European field forest and meadow fox	2
Small stinger hedgehog	4
Big brown scary bear	1

The data is quite compact, while the search index is very long. If we had a hash function which maps to an interval $\{1, 2, 3, \dots, m\}$, it would look like this.



Now we could store at the storage position $S[f(\dots \text{bunny} \dots)]$ the value 12 in this hash table, at the position $S[f(\dots \text{fox} \dots)]$ the value and so on. This operation is named PUT.

PUT

```

1   procedure PUT (string x, int z)
2   begin
3       S[f(x)] := z
4   end

```

GET returns the value where the value 0 denotes that no data has been stored.

GET

```

1   procedure GET (string x)
2   begin
3       return S[f(x)]
4   end

```

Unfortunately, these functions work correctly only if the hash function is without any collisions. So, for example, bunny and hedgehog are not mapped to the same storage position. This can be guaranteed only for small m , if the keys are known in advance (like here with bunny, fox, hedgehog and bear) and a so-called *perfect hash function* has been chosen.

If one does not know the keys in advance, empty positions must be found. For this, several methods are at hand. The easiest method is the so-called *linear probing*. Then, in addition to the data, also the key must be stored.

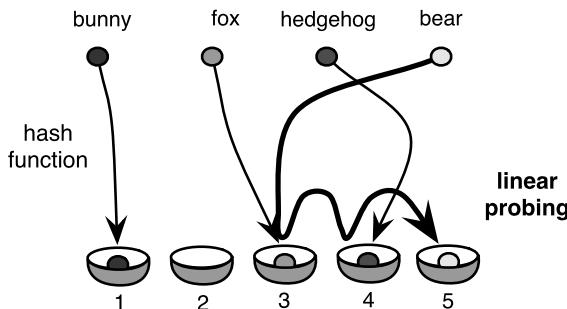
Storing a Data Item z with Key x

First we compute the hash value $f(x)$ of x . If the $S[f(x)]$ is already occupied, the right neighbor is tested ($f(x) + 1$), and its right neighbor, until an empty

storage position has been found. If during the search one ends at the right end of the storage array, the search continues at the first position. If an empty position is eventually found, the key x and the data item z are stored. If all storage positions have been unsuccessfully investigated, then the storage is full and the operation returns an error message.

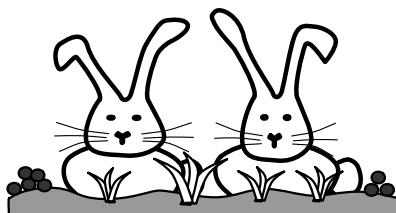
Searching a Data Item Corresponding to Key x

Again, we compute the hash value $f(x)$ of the key x . If the storage position holds another key we start the search to the right until the fitting k is found or an empty storage position has been found. On this search we jump from the right end of the storage array to the first position. The search is unsuccessful if an empty storage position has been found or all storage items have been investigated. In all other cases we find the key x and can return the data item.



With this collision resolution we can always store m data items no matter how good or bad the hash functions we have chosen. If you want to try this yourself, start with natural numbers as keys and choose as the hash function the modulo- m function, which is the remainder after division by m . At the beginning, storing and retrieving the data will be surprisingly fast. But when the table is nearly full, then this algorithm becomes slower and slower.

This is caused by the way we handle collisions. Linear probing always tests neighboring positions. Better solutions are known, like quadratic probing or double hashing, which some Germans mix up with *Doppel-Häschen* ...



External Links and References

1. Wikipedia: MD5, SHA-1, Hash-Table.
2. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. *Introduction to Algorithms*. 1184 pages, MIT Press, 2001, ISBN 0-262-53196-8.

Codes – Protecting Data Against Errors and Loss

Michael Mitzenmacher

Harvard University, Cambridge, USA

21.1 Introduction

Suppose that, after meeting a new friend at a party, you want to get his or her cell phone number – ten digits. (If you’re single, feel free to put this story in the context of meeting a special someone in a cafe or other locale.) You don’t have your phones with you, so you resort to having your friend write their telephone number on a nearby scrap of paper. Sadly, your new friend’s handwriting is rather messy, and the ink has a tendency to smudge, so you are worried you will not be able to read one (or more) of the digits later. Let us assume that if you cannot clearly read a number you will simply treat it as unknown, or erased. You might read

617-555-0?23,

where we use the question mark to mean you were not sure what that number was. You’d prefer to avoid calling multiple numbers hoping to find the right one, so you consider what you can do to cope with the problem in advance.

Your friend could write the cell phone number down for you twice, or even three times, and then you would be much more likely to be able to determine it later. By just repeating the cell number twice, you would be guaranteed to know it if any single digit was erased; in fact, you would know it as long as, for every digit, both copies of that digit were not erased. Unless you expect a great number of messy smudges, though, repeating the number seems like overkill. So here is the challenge – can you write down an eleventh digit that will allow you to correct for *any single* missing digit? You might first want to consider a slightly easier problem: can you write down an extra number between 1 and 100 that will allow you to correct for any single missing digit?

In this puzzle, we are trying to come up with a *code*. Generally, a code is used to protect data during transmission from specific types of errors. In this example, a number that is so messy or smudged that you cannot tell what it is would be called an *erasure* in coding terminology, so our resulting code would be called an *erasure code*. There are many different types of errors that

can be introduced besides erasures. I might write down (or you might read) a digit incorrectly, turning a 7 into a 4. I might transpose two digits, writing 37 when I meant 73. I might forget to write a number, so you only have nine digits instead of the ten you would expect. There are also codes for these and other more complicated types of errors.

Codes protect data by adding redundancy to it. Perhaps the most basic type of coding is just simple repetition: write everything down two or three or more times. Repetition can be effective, but it is often very expensive. Usually, each piece of data being transmitted costs something – time, space, or actual money – so repeating everything means paying at least twice as much. Because of this, codes are generally designed to provide the most bang for the buck, solving as many or as many different kinds of errors as possible with the least additional redundancy.

This brings us back to the puzzle. If I wanted to make sure you could correct for the erasure of any single digit, I could provide you with the sum of the digits in my phone number. If one digit then went missing, you could subtract the other numbers to find it. For example, if I wrote

617-555-0123 35,

and you read

617-555-0?23 35,

you could compute

$$35 - (6 + 1 + 7 + 5 + 5 + 5 + 0 + 2 + 3) = 1$$

to find the missing number.

We can reduce the amount of information passed even further because, in this case, you really do not even need the tens digit of the sum. Instead of writing 35, I could just write down 5. This is because no matter what digit is missing, the sum you will get from all of the remaining digits will be between 26 and 35. You will therefore be able to conclude that the sum must have been 35, and not 25 or smaller (too low) or 45 or larger (too big). Just one extra digit is enough to allow you to handle any single erasure.

It is interesting to consider how helpful this extra information would be in the face of other types of errors. If you misread exactly one of the digits in the phone number, you would see that there was a problem somewhere. For example, if you thought what I wrote was

617-855-0123 5,

you would see the phone number and the ones digit of the sum did not match, since the sum of the digits in the phone number is 38. In this case, the extra information allows you to *detect* the error, but it does not allow you to *correct* the error, as there are many ways a single digit could have been changed to end up with this sequence. For example, instead of

617-555-0123,

my phone number might originally have been

617-852-0123,

which also differs from what you received in just one digit, and also has all the digits sum to 35, matching the extra eleventh digit 5 that was sent. Since without additional information you cannot tell exactly what my original number was, you can only detect but not correct the error. In many situations, detecting errors can be just as or almost as valuable as correcting errors, and generally detection is less expensive than correction, so sometimes people use codes to detect rather than correct errors.

If there were two changed digits, you might detect that there is an error. Or you might not! If you read

617-556-0723 5,

the extra sum digit does not match the sum, so you know there is an error. But if you read

617-555-8323 5,

you would think everything seemed fine. The two errors match up in just the right way to make the extra digit match. In a similar manner, providing the sum does not help if the error is a transposition. If instead of

617-555-0123 5,

you thought I wrote

617-555-1023 5,

that would seem perfectly fine, since transpositions do not change the sum.

21.1.1 Where Are Codes Used?

You are probably using codes all the time, without even knowing it. For example, every time you take your credit card out of your wallet and use plastic to pay a bill, you are using a code. The last digit of your credit card number is derived from all of the previous digits, in a manner very similar to the scheme we described to handle erasures and errors in the telephone number puzzle. This extra digit prevents people from just making up a credit card number off the top of their heads; since one must get the last digit right, at most only one in ten numbers will be valid. The extra digit also prevents mistakes when transcribing a credit card number. Of course, transposition is a common error, and as we've seen, using the sum of the digits cannot handle transposition errors. A more difficult calculation is made for a credit card, using a standard called the *Luhn formula*, which detects all single-digit errors (like the sum) and most transpositions. Additional information of this sort, which is used to detect errors instead of correct them, is typically called a *checksum*.

Error-correcting codes are also used to protect data on compact discs (CDs) and digital video discs (DVDs, which are also sometimes called digital versatile discs, since they can hold more than video). CDs use a method of error-correction known as a cross-interleaved Reed–Solomon code (CIRC). We'll say a bit more about Reed–Solomon codes later on. This version of Reed–Solomon code is especially designed to handle bursts of errors, which might arise from a small scratch on the CD, so that the original data can be reconstructed. The code can also correct for other errors, such as small manufacturing errors in the disc. Roughly one fourth of the data stored on a CD is actually redundancy due to this code, so there is a price to pay for this protection. DVDs use a somewhat improved version known as a Reed–Solomon product code, which uses about half as much redundancy as the original CIRC approach. More generally, error-correcting codes are commonly used in various storage devices, such as computer hard drives. Coding technology has proven to be key to fulfilling our desire for easy storage of and access to audio and video data.

Of course, codes for error correction and error detection also come into play in almost all communication technologies. Your cell phone, for example, uses codes. In fact, your cell phone uses multiple complex codes for different purposes at different points. Your iPod uses codes. Computer modems, fax machines, and high-definition television use error-correction techniques. Moving from the everyday to the more esoteric, codes are also commonly used in deep-space satellites to protect communication. When pictures are sent back to NASA from the far reaches of our solar system, they come protected by codes.

In short, coding technology provides a lynchpin for all manner of communication technology. If you want to protect your data from errors, or just detect errors when they occur, you apply some sort of code. The price for this protection is paid with redundancy and computation. One of the key things people who work on codes try to determine is how cheap we can make this protection while still making it as strong as possible.

21.2 Reed–Solomon Codes

The invention of Reed–Solomon codes revolutionized the theory and practice of coding, and they are still in widespread use today. Reed–Solomon codes were invented around 1960 by two scientists, Irving Reed and Gustave Solomon. The codes can be used to protect against both erasures and errors. Also importantly, there are efficient algorithms for both *encoding*, or generating the information to send, and *decoding*, or reconstructing the original message from the information received. Fast algorithms for decoding Reed–Solomon codes were developed soon after the invention of the codes themselves by Berlekamp and Welch. Most of the coding circuits that have ever been built implement Reed–Solomon codes, and use some variation of Berlekamp–Welch decoding.

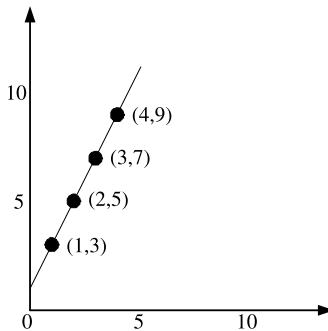


Fig. 21.1. An example of Reed–Solomon codes. Given the two numbers 3 and 5, we construct the line between the two points $(1, 3)$ and $(2, 5)$ to determine additional numbers to send. Receiving any two points, such as the points $(3, 7)$ and $(4, 9)$, will allow us to reconstruct the line and determine the original message

The basic idea behind Reed–Solomon codes can be explained with a simple example. I would like to send you just two numbers, for example a 3 followed by a 5. Suppose that I want to protect against erasures. We will think of these numbers as not just being numbers, but being points on a line: the first number becomes the point $(1, 3)$, to denote that the first number is a 3. The second number becomes the point $(2, 5)$, to denote that the second number is a 5. The line between these points can be pictured graphically, as in Fig. 21.1, or can be thought of arithmetically: the second coordinate is obtained by doubling the first, then adding 1. *My goal will be to make sure that you obtain enough information to reconstruct the line.* Once you can reconstruct the line, you can determine the message, simply by finding the first two points $(1, 3)$ and $(2, 5)$. You then know I meant to send a 3 and a 5. The key idea is that instead of thinking about the data itself, we think about a line that encodes the data, and focus on that.

To cope with erasures, I can just send you other points on the line! That is, I can send you extra information by finding the next points on the line, $(3, 7)$ and $(4, 9)$. I would send the second coordinates, in order:

$$3, 5, 7, 9.$$

I could send more values – the next would be 11, for $(5, 11)$ – as many as I want. The more values I send, the more erasures you can tolerate, but the cost is more redundancy.

Now, as long as *any* two values make it to you, you can find my original message. How does this work? Suppose you receive just the 7 and 9, so what you receive looks like

$$?, ?, 7, 9,$$

where the “?” again means the number was erased. You know that those last two numbers correspond to the points $(3, 7)$ and $(4, 9)$, since the 7 is in the

3rd spot on your list and the 9 is in the 4th. Given these two points, you can yourself draw the line between them, because *two points determine a single line!* The line is exactly the same line as in Fig. 21.1. Now that you know the line, you can determine the message.

The key fact we are using here is that two points determine a line, so once you have any two points, you are done. There is nothing particularly special about the number two here. If I wanted to send you three numbers, I would have to use a parabola, instead of a line, since any three points determine a parabola. If I wanted to send you 100 numbers, I could build a curve determined by the 100 points corresponding to these numbers. Such curves are written as polynomials: to handle 100 points, I would use curves with points (x, y) satisfying an equation looking like $y = a_{99}x^{99} + a_{98}x^{98} + \dots + a_1x + a_0$, where the a_i are appropriately chosen numbers so that all the points satisfy the equation. To send k numbers, I need k coefficients, or a polynomial of degree $k - 1$, to represent the numbers.

Reed–Solomon codes have an amazing property, with respect to erasures: if I am trying to send you 100 numbers, we can design a code so that you get the message as soon as you receive any 100 points I send. And there is nothing special about 100; if I am trying to send you k numbers, all you need to receive is k points, and any k points will do. In this setting, Reed–Solomon codes are optimal, in the sense that if I want to send you 100 numbers, you really have to receive some 100 numbers from me to have a good chance to get the message. What is surprising is that it does not matter which 100 you get!

An important detail you might be wondering about is what happens if one of the numbers I am supposed to send ends up not being an integer. Things would become a lot more complicated in practice if I had to send something like 16.124875. Similar problems might arise if the numbers I could send could become arbitrarily long; this too would be impractical. To avoid this, all work can be done in *modular* or *clock* arithmetic. In modular arithmetic, we always take the remainder after dividing by some fixed number. If I work “modulo 17”, instead of sending the number 47, I would send the remainder after dividing 47 by 17. This remainder would be 13, since $47 = 2 \times 17 + 13$. Modular arithmetic is also called clock arithmetic, because it works like counting on a clock. In Fig. 21.2 we see a clock corresponding to counting modulo 5. After we get to 4, when we add 1, we go back to 0 again (since the remainder when dividing 5 by 5 is 0). We have already seen an example of modular arithmetic in our original puzzle. Instead of sending the entire sum, I could get away with just sending the ones digit, which corresponds to working “modulo 10”. It turns out that all the arithmetic for Reed–Solomon codes can be performed modulo a big prime number, and with this, all the numbers that need to be sent will be suitably small integers. (Big primes are nice mathematically for various reasons. In particular, each nonzero number has a *multiplicative inverse*, which is a corresponding number whose product with the original number is one. For example, 6 and 2 are inverses modulo 11, since

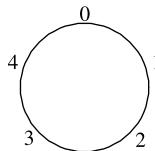


Fig. 21.2. Modular arithmetic: counting “modulo 5” is like counting on a clock that goes back to zero after four. Equivalently, you just consider the remainder when dividing by 5, so 7 is the same as 2 modulo 5

$6 \times 2 = 12 = 11 + 1$, so 6×2 is equivalent to 1 modulo 11. In practice, things are slightly more complicated; one often does not work modulo some prime, but uses a number system with similar properties, including the property that each nonzero number has a multiplicative inverse.)

What about dealing with errors, instead of erasures? As long as the number of errors is small enough, all is well. For example, suppose again I sent you the numbers

$$3, 5, 7, 9,$$

but you received the numbers

$$3, 4, 7, 9,$$

so that there is one error. If we plot the corresponding points, $(1, 3)$, $(2, 4)$, $(3, 7)$, and $(4, 9)$, you can see that there is only one line that manages to pass through three of the four points, namely the original line. Once you have the original line, you can correct the point $(2, 4)$ to the true point $(2, 5)$, and recover the message. If there were too many errors, it is possible that we would obtain no line at all passing through three points, in which case we would detect that there were too many errors. For example, see Fig. 21.3. Another possibility is that if there were too many errors, you might come up with the wrong line, and you would decode incorrectly. See Fig. 21.4 for an example of this case. Again, there is nothing special about wanting to send two numbers. If I wanted to send you three numbers, and cope with one error, I would send you five points on a parabola. If there was just one error, there would be just one parabola passing through four of the five points. The idea extends to larger messages, and the Berlekamp–Welch algorithm decodes efficiently even in the face of such errors.

In general, if I am trying to send you a message with k numbers, in order for you to cope with e errors, I need to send you $k + 2e$ points using a Reed–Solomon code. That is, each error to be handled requires sending *two* additional numbers. Therefore, to send you two numbers and handle one error, I needed to send $2 + 2 \cdot 1 = 4$ symbols.

Because Reed–Solomon codes have proven incredibly useful and powerful, for many years, it proved hard to move beyond them, even though there were reasons to do so. On the theoretical side, Reed–Solomon codes were not

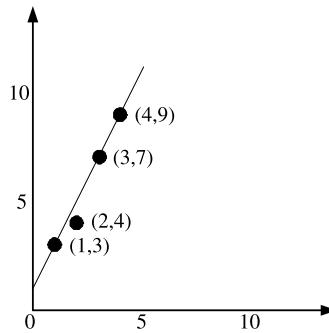


Fig. 21.3. An example of decoding Reed–Solomon codes when there are errors. Given the four points, one of which is in error, there is just one line that goes through three of the four points, namely the line corresponding to the original message. Determining this line allows us to correctly reconstruct the message

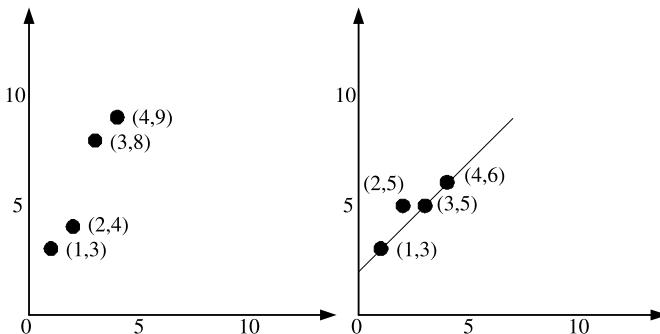


Fig. 21.4. Examples of decoding Reed–Solomon codes when there are too many errors. Given the four points, when there are two or more errors, there may be no line that goes through three or more of the points, as in the example on the left, in which case one can detect there is an error but not correct it. Alternatively, when there are two errors, three of the points may lie on an incorrect line. On the right-hand side, one could find a line going through three of the points, and incorrectly conclude that the point (2,5) should have been (2,4), giving an incorrect decoding

optimal for many types of errors. Theoreticians always like to have the best answer, or at least something very close to it. On the practical side, although Reed–Solomon codes are very fast for small messages, they are not very efficient when used for larger messages. This is in part because of the overhead of using modular arithmetic, which can be nontrivial, and because the decoding schemes just take longer when used for messages with lots of numbers and lots of erasures or errors. Specifically, the decoding time is roughly proportional to the product of the length of the message and the number of errors, or ke . This means that if I want to double the message length and handle twice as many errors, so that the overall percentage of errors remains the same, the

time to decode increases by roughly a factor of four. If I want the message length to increase by a factor of one hundred, and handle the same percentage of errors, the time to decode increases by roughly a factor of 10,000, which is substantial! The problems of dealing with larger messages were not too important until computers and networks became so powerful that sending huge messages of megabytes or even gigabytes became ordinary. These problems could be dealt with using tricks like breaking up a large message into several small messages, but such approaches were never entirely satisfactory, and people began looking for other ways to code to protect against erasures and errors.

21.3 New Coding Techniques: Low-Density Parity-Check Codes

Over the past fifteen years, a new style of coding has come into play. The theory of these codes has become quite solidly grounded, and the number of systems using these codes has been growing rapidly. Although there are many variations on this style, they are grouped together under the name Low-Density Parity-Check codes, or LDPC codes for short. LDPC codes are especially good for situations where you want to encode large quantities of data, such as movies or large software programs.

Like Reed–Solomon codes, LDPC codes are based on equations. With Reed–Solomon codes, there was one equation based on *all* of the data in the message. Because of this, Reed–Solomon codes are referred to as *dense*. LDPC codes generally work differently, using lots of small equations based on small parts of the data in the message. Hence they are called *low-density*.

The equations used in LDPC codes are generally based on the exclusive-or (XOR) operation, which works on individual bits. If the two bits are the same, the result is 0, otherwise it is 1. So writing XOR as \oplus , we have

$$0 \oplus 0 = 0; \quad 1 \oplus 1 = 0; \quad 1 \oplus 0 = 1; \quad 0 \oplus 1 = 1.$$

Another way of thinking about the XOR operation is that it is like counting modulo 2, which corresponds to a clock with just two numbers, 0 and 1. This makes it easy to see that, for example,

$$1 \oplus 0 \oplus 0 \oplus 1 = 0.$$

We can extend XOR operations to bigger strings of bits of the same length, by simply XORing the corresponding bits together. For example, we would have

$$10101010 \oplus 01101001 = 11000011,$$

where the first bit of the answer is obtained from taking the XOR of the first bits of the two strings on the left, and so on.

The XOR operation also has the pleasant property that for any string S , $S \oplus S$ consists only of zeroes. This makes it easy to solve equations using XOR, like the equations you may have seen in algebra. For example, if we have

$$X \oplus 10101010 = 11010001,$$

we can “add” 10101010 to both sides to get

$$X \oplus (10101010 \oplus 10101010) = 11010001 \oplus 10101010.$$

This simplifies, since $10101010 \oplus 10101010 = 00000000$, to

$$X = 01111011.$$

We will now look at how LDPC codes work in the setting where there are erasures. LDPC codes are based on the following idea. I will split my message up into smaller blocks. For example, when you send data over the Internet, you break it up into equal-sized packets. Each packet of data could be a block. Once the message is broken into blocks, I repeatedly send you the XOR of a small number of random blocks. One way to think of this is that I am sending you a bunch of equations. For example, if you had blocks of eight bits, labeled $X_1, X_2, X_3, X_4, \dots$, taking on the values

$$X_1 = 01100110, \quad X_2 = 01111011, \quad X_3 = 10101010, \quad X_4 = 01010111, \quad \dots,$$

I might send you $X_1 \oplus X_3 \oplus X_4 = 10011011$. Or I could just send you $X_3 = 10101010$. I could send you $X_5 \oplus X_{11} \oplus X_{33} \oplus X_{74} \oplus X_{99} \oplus X_{111} = 10111111$. Notice that whenever I send you a block, I have to choose *how many* message blocks to XOR together, and then *which* message blocks to XOR together.

This may seem a little strange, but it turns out to work out very nicely. In fact, things work out remarkably nicely when I do things *randomly*, in the following way. Each time I want to send out an encoded block, I will randomly pick how many message blocks to XOR together according to a distribution: perhaps 1/2 of the time I send you just 1 message block, 1/6 of the time I send you the XOR of 2 message blocks, 1/12 of the time I send you the XOR of 3 message blocks, and so on. Once I decide how many blocks to send you, I choose which blocks to XOR uniformly, so each block is equally likely to be chosen.

With this approach, some of the information I send could be redundant, and therefore useless. For example, I might send you $X_3 = 10101010$ twice, and certainly you only need that information once. This sort of useless information does not arise with Reed–Solomon codes, but in return LDPC codes have an advantage in speed. It turns out that if I choose just the right random way to pick how many message blocks to XOR together, then there will be very little extra useless information, and you will almost surely be able to decode after almost just the right number of blocks. For example, suppose I had a message of 10,000 blocks. On average, you might need about 10,250 blocks to decode with a well-designed code, and you would almost surely be done after 10,500 blocks.

LDPC codes can be decoded in an interesting and quite speedy way. The basic idea works like this: suppose I receive an equation $X_3 = 10101010$, and another equation $X_2 \oplus X_3 = 11010001$. Since I know the value of X_3 , I can substitute the value of X_3 into the second equation so that it becomes $X_2 \oplus 10101010 = 11010001$. Now this equation has just one variable, so we can solve it, to obtain $X_2 = 01111011$. We can then substitute this derived value for X_2 into any other equations with the variable X_2 , and look for further equations with just one variable left that can be solved. If all works out, we can just keep substituting values for variables, and solving simple equations with just one variable, until everything is recovered. At some point, we get a chain reaction, where solving for one variable lets us solve for more variables, which lets us solve for more variables, until we end up solving everything!

Just as Reed–Solomon codes can also be used to deal with errors instead of simply erasures, variations of LDPC codes can also be used to deal with errors. One of the amazing things about LDPC codes is that for many basic settings one can prove that they perform almost optimally. That is, the theory tells us that if we send data over a channel with certain types of errors, there is a limit to how much useful information we can expect to get. For erasures, as we saw with Reed–Solomon codes, this limit is trivial and can be achieved: to obtain a message of 100 numbers, you need to receive at least 100 numbers, and Reed–Solomon codes allow you to decode once you receive any 100 numbers. LDPC codes, in many situations with other types of errors, brush right up against the theoretical limits of what is possible! LDPC codes are so close to optimal we can hope to do very little better in this respect.

21.4 Network Codes

One thought that might be going through your mind is that if the recent success of low-density parity-check codes means we can reach the theoretical limits in practice, is coding research essentially over? This is not at all a strange idea. It comes up as a point of discussion at scientific conferences, and certainly it is possible for scientific subfields to grow so mature that there seem to be few good problems left.

I am happy to report that coding theory is far from dead. There are many areas still wide open, and many fundamental questions left to resolve. Here I will describe one amazing area that is quite young and is currently the focus of tremendous energy.

For the next decade, a key area of research will revolve around what is known as *network coding*. Network coding has the potential to transform the underlying communication infrastructure, but it is too early to tell if the idea will prove itself in the real world, or simply be a mathematical curiosity.

To understand the importance of network coding, it is important to understand the way networks have behaved over the last few decades. Computer networks, and in particular the Internet, have developed using an architecture based on routers. If I am sending a file from one side of the country to the other, the data is broken into small chunks, called packets, and the packets are passed through a series of specialized machines, called routers, that try to move the packets to the end destination. While a router could, conceivably, take the data in the packets and examine, change, or reorganize them in various ways, the Internet was designed so that all a router had to do is look at where the packet is going, and pass it on to the next hop on its route. This step is called *forwarding* the packet. By making the job of the routers incredibly simple, companies have been able to make them incredibly fast, increasing the speed of the network and making it possible to download Web pages, movies, or academic papers at amazing speeds.

The implication of this design for coding was clear: if you wanted to use a code to protect your data, the encoding should be done at the machine sending the data, called the *source*, before the packets were put on the network. Decoding should be done at the destination, after the packets are taken off the network. The routers would be uninvolved, so they could concentrate on *their* job, passing on the data.

Currently, this network design principle is being reexamined. Scientists and the people who run networks have been considering whether routers can do more than simply route. For example, it would be nice if routers could find and remove harmful traffic, such as viruses and worms, as soon as possible. Or perhaps routers could monitor traffic and record statistics that could be used for billing customers. The reason for this change in mindset has to do with how the Internet has developed and with changes in technology. As issues like hacking attacks and e-commerce have moved to the forefront, there has been a push to think about what more the network can do. And as routers have grown in speed and sophistication, it becomes natural to question whether in the future they should be designed to do more, instead of just forwarding packets even faster.

Once you lose the mindset that all the router should do is forward packets, all sorts of questions arise. In the context of coding, we might ask if the routers can usefully be involved with coding, or if coding can somehow help the routers. These ideas represent a fundamental shift from the previous paradigm where encoding is done only at the source and decoding only at the receiver. Once people started asking these questions, interesting and innovative results started to arise, leading to the new field of network coding.

What sort of coding could you do on a network? There is a nice prototypical example, given pictorially in Fig. 21.5. To explain the example, it helps to start by thinking about the connections as pipes, carrying commodities like water and oil. Suppose that I control the supplies of oil and water, located at S , and I want to ship water and oil along the connections shown, which are my pipes. I can send one gallon per minute of either on any pipe, but I can't

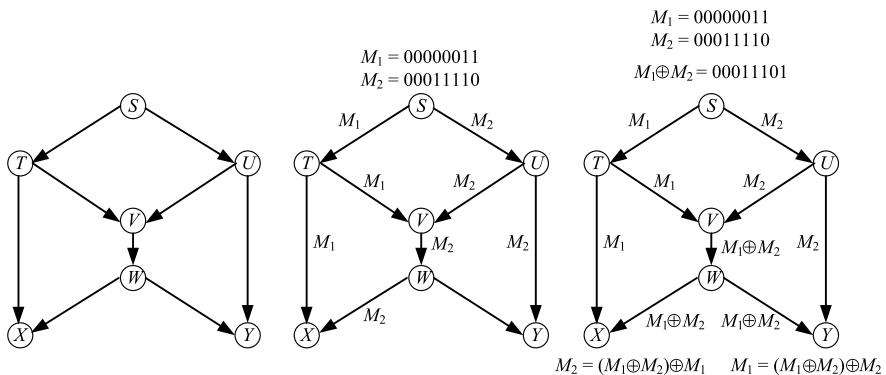


Fig. 21.5. Network coding in action. The left-hand side is the original network. The middle shows what happens if you just use forwarding; there is a bottleneck from V to W , slowing things down. I can get both blocks to X or to Y , but not to both simultaneously. The right-hand side demonstrates network coding. I avoid the bottleneck by sending the XOR of M_1 and M_2 to both X and Y by way of W

send both water and oil on the same pipe at the same time. After all, oil and water do not mix! Can I get two gallons of water and oil per minute flowing at each of the final destinations X and Y ?

Obviously, the answer is no. The problem is that I can only get two gallons of stuff per minute out of S in total, so there is no way I can get two gallons of both oil and water to each of X and Y .

Now suppose that instead of sending oil and water I was sending data – packets – corresponding to movies that people want to download. The network would look the same as my original network in Fig. 21.5, but now all my pipes are really fiber-optic cables, and I can carry 10 Megabits per second along each cable. Can I get both movies to both final destinations each at a rate of 10 Megabits per second? Now the oil and water limitation does not necessarily hold, since I can forward information on multiple links, making copies.

If I just use forwarding, however, it does not seem possible to get both movies to both X and Y , as shown in the middle of Fig. 21.5. To simplify things, suppose I just have two message blocks of eight bits, $M_1 = 00000011$ and $M_2 = 00011110$, that need to get to each destination X and Y , without any interference anywhere along the path. Again, because I can copy data at intermediate points, the fact that there are just two pipes out of S is not immediately a problem. The block M_2 I send from S to U can be forwarded to both V and Y , and from V it can be forwarded on to X , as shown in Fig. 21.5. But even though there are two pipes into X and two pipes into Y , it does not seem like I can make effective use of all of them, because of the bottleneck from V to W . I can send $M_1 = 00000011$ to X by going from S to T to X , and similarly I can send $M_2 = 00011110$ to Y by going from S to U to Y . But if I then try to send M_1 to Y by way of V , and M_2 to X by way

of V , the two blocks will meet at V , and one will have to wait for the other to be sent, slowing down the network.

One way to fix this would just be to speed up the link between V and W , so that both messages could be sent. If I could send two blocks on that link at a time, instead of just one, the problem would disappear. Equivalently, I could build a second link between V and W .

Is there any way around the problem without adding a link between V and W or speeding up the link that is already there? Amazingly, the answer is *yes!* The reason is that, unlike in the case of shipping physical commodities like oil and water, data can be mixed as well as copied! To see how this would work, consider the right side of Fig. 21.5. I start sending my data as before, but now, when M_1 and M_2 get to V , I take the XOR of the two blocks and transport that value as a block 00011101 to X and Y through W . This mixes the data together, but when everything gets to the destinations, the data can be unmixed just by using XOR again. At X , we take the block $M_1 \oplus M_2 = 00011101$ and XOR it with $M_1 = 00000011$ to get

$$M_2 = 00011101 \oplus 00000011 = 00011110,$$

and at Y we take the block $M_1 \oplus M_2 = 00011101$ and XOR it with $M_2 = 00011110$ to get

$$M_1 = 00011101 \oplus 00011110 = 00000011.$$

An XOR here, an XOR there, and the bottleneck disappears! The magic is that the value $M_1 \oplus M_2$ is useful at both X and Y .

Network coding is naturally much harder than this simple example suggests. In a large network, determining the best ways to mix things together can require some work! In many cases, we do not yet know what gains are possible using network coding. But there is a great deal of excitement as new challenges arise from our new view of how the network might work. Our understanding of network coding is really just beginning, and its practical impact, to this point, has been negligible. But there is a great deal of potential. Perhaps someday most of the data traversing networks will be making use of some form of network coding, and what seems novel today will become commonplace.

21.5 Places to Start Looking for More Information

1. There are several relevant pages on Wikipedia, including:
 - http://en.wikipedia.org/wiki/Luhn_algorithm
 - <http://en.wikipedia.org/wiki/Checksum>
 - http://en.wikipedia.org/wiki/Reed-Solomon_error_correction
 - http://en.wikipedia.org/wiki/Cross-interleaved_Reed-Solomon_coding
 - http://en.wikipedia.org/wiki/Low-density_parity-check_code

- http://en.wikipedia.org/wiki/Network_coding
2. The book *Information Theory, Inference, and Learning Algorithms*, by David MacKay, Cambridge University Press, 2003. A version is also available online at <http://www.inference.phy.cam.ac.uk/mackay/itila/book.html>
 3. The book *Modern Coding Theory*, by Tom Richardson and Rüdiger Urbanke, Cambridge University Press, 2008.

Acknowledgement

The author is supported by NSF grant CCF-0915922.

Part III

Planning, Coordination and Simulation

Overview

Helmut Alt and Rüdiger Reischuk

Freie Universität Berlin, Berlin, Germany
Universität zu Lübeck, Lübeck, Germany

Strategic thinking and planning are commonly regarded as typically human capabilities. Ever since computer programs demonstrated that they can beat chess grand masters, however, one can see that some of these skills can be successfully managed by machines. On the other hand, some games can be won with very simple strategies, one must have the right knowledge. In a chapter in this part of the book we see this demonstrated impressively by the match game Nim.

In many games, it is important that we don't allow the enemy to anticipate our moves. A simple strategy – referred to in computer science jargon as deterministic – can, however, be predicted. This can be avoided if you include random decisions – without this many games such as rock–paper–scissors would be quite boring. Many algorithms can be improved or be speeded up in this way – these are called probabilistic or randomized algorithms. Now we need to ask ourselves how a computer could toss a coin, given that we would expect only full precision? The chapter here on random numbers provides an answer.

A strategic and algorithmic approach makes sense even with everyday problems, and not just during games. For example, if we wish to disseminate a message to a broad group of people through phone calls or to many computers via an electronic network, then we need a good plan in order to achieve this objective quickly and reliably. We see this in the chapter on broadcasting. In a further chapter we see a clever approach to determining the winner of an election.

Some tasks require careful long-term planning. An example is the game schedule for the Bundesliga, the German soccer league, which requires us to consider various constraints.

Two chapters in this section deal with simulations, i.e., simulating natural processes using computers. First we consider a problem from physics. We see how to calculate the heat distribution in a metal rod or plate using so-called Gauss–Seidel iteration. In the other chapter we consider a theme from biology. We see how one can determine how closely two organisms are related to

each other from their genetic information (DNA); and we see from mutations, minimal changes in the genetic heritage, how far apart they are from each other or from a common ancestor.

The famous mathematician Leonhard Euler posed the Königsberg Bridges Problem: Can you cross all seven bridges exactly once on a walk and then return to the starting point? This playful question – by the way, the answer is “No”! – has important applications, such as in route planning, covered in the chapter on Eulerian circuits. In vehicle navigation we are now accustomed to a friendly voice that offers directions or tells us the distance to travel before the next turn. For a long time natural speech was an unsolved problem for computers. In this part of the book we see that even pronouncing long numbers involves considerable computational effort.

Finally, we consider a problem in computer graphics. Draw a circle as round as possible on a screen, realized using a grid of individual pixels. Strictly speaking we cannot draw a slanted or curved line, as we could with paper and pencil. However, a detailed analysis of the problem leads to surprisingly easy and fast solution algorithms.

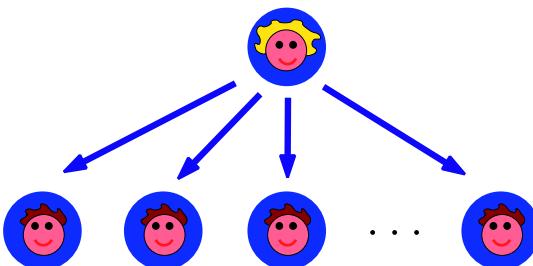
Broadcasting – How Can I Quickly Disseminate Information?

Christian Scheideler

Universität Paderborn, Paderborn, Germany

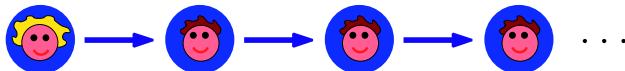
In the Middle Ages, there was no mass media like TV or radio. As most people were not able to write or read, information was mostly disseminated on a mouth-to-mouth basis, and since the travel speed of humans was quite restricted at those times, the spreading of information was mostly bounded by the speed of horses (though other means like pigeons and smoke or light signals were also used occasionally). Nowadays, the telephone and other media like the Internet allow anyone to spread information very quickly around the world. Let us consider a specific example here.

Steffi has just been given the task to organize a party for her class, and this at a time when the school holidays have just started! Now, she has to try to reach all fellow students by phone or email. Unfortunately, Steffi does not know their email addresses, but she has a list of all 121 students with their phone numbers which was recently given to every student (and that hopefully no one has thrown away yet!). Now, Steffi could try to do 120 phone calls, which would consume a lot of time. Hence, she thinks about an alternative approach to reach all students as quickly and cheaply as possible.



Strategy 1: Call everybody directly

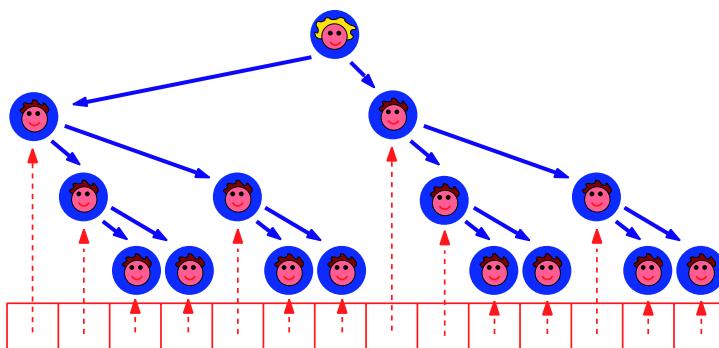
The first strategy that comes to her mind is the silent post game: she just calls the first person on the list and asks him or her to call the next one on the list, who will then call the next one, and so on, until everybody on the list has been reached.



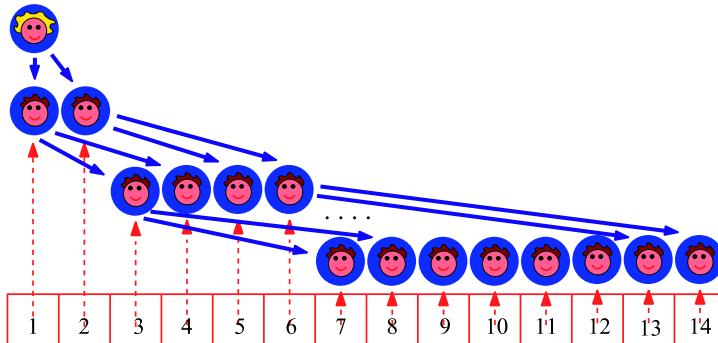
Strategy 2: Silent post

The advantage of this strategy is that every student only has to make one call. However, since the calls have to be performed one after the other, a very long time can go by until all students have been reached. In fact, if just 10% of the students do not reach the next one on the list within the same day they were called, it takes at least 12 days until everybody has been informed. Even worse: if someone does not bother to call the next one on the list, the whole system will break down! Thus, Steffi thinks about an alternative approach.

Since she is interested in computer science, she recalls a sorting method that has also been presented in Chap. 3. There, a master uses two helpers to cut a sorting problem into two smaller sorting problems, who themselves use two helpers each to cut their sorting problems into even smaller problems, and so on, until just one element is left. Something similar to that should also work for the distribution of calls! For example, Steffi could divide the phone list into two halves and call the first person on each of the two halves. Each of them will then be asked to cut their list into two further halves and call the first person on these halves. This is continued until everybody has been called, i.e., we reach a level in which people are called who just have to take care of an empty list. In this way, the students can be reached much quicker.



Strategy 3: Partitioning the phone list into sublists



Strategy 4: Everybody at list position i calls positions $2i + 1$ and $2i + 2$

Indeed, Steffi determines that just seven rounds of calls are sufficient to reach all 120 fellow students. This is much better than 120 rounds of calls! However, her strategy sounds very technical, so it's questionable whether the other students can be made to adhere to the rules without errors. Thus, she thinks about an alternative strategy.

Suppose that she calls the first two people on the list, Andi and Berthold, and asks Andi to call the students at positions 3 and 4 while Berthold is asked to call the students at positions 5 and 6. In general, the rule would be that everybody at position i in the list will call the students at positions $2i + 1$ and $2i + 2$ (if they exist). Then the information spreads at the same speed as in the previous strategy, but the calling rule sounds now much more natural and easy to understand.

Nevertheless, Steffi is not quite happy with her calling strategy. What if one of her fellow students does not count right and calls a wrong pair of students on the list? Moreover, there can still be a couple of students who just forget or do not bother about calling their pair on the list. In this case, some students would not be informed, who would then be mad at Steffi!

Therefore, Steffi thinks about a more robust strategy. One possibility would be that everyone at list position i would call the four students at positions $2i + 1$ to $2i + 4$. In this case, all students (except for the first four on the list who will directly be called by Steffi) will be called by exactly two students in the ideal case. Thus, as long as for each such pair at most one of the students is unreliable (by not being reachable or forgetting to make the call), all of the reliable students will still be informed. Intuitively, this can be argued as follows: If one can select a caller for each student who works reliably, then everybody who is reliable has a reliable call chain from himself or herself back to Steffi (see also Fig. 22.1).

Steffi quickly realizes that this strategy can be made even more robust, so that she can be really sure to reach everybody who is reachable: If every student at position i calls the students at positions $2i + 1$ to $2i + 2r$ for some

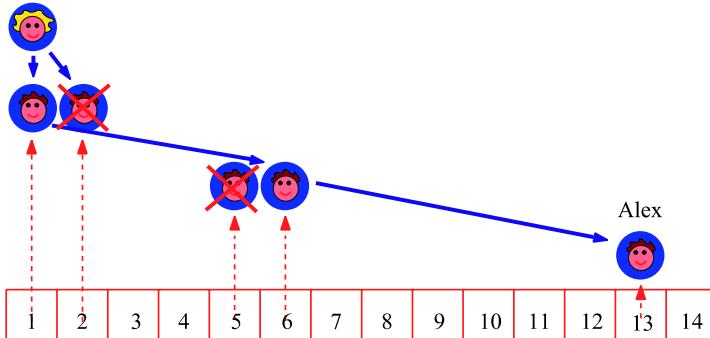


Fig. 22.1. Chain of reliable students for Alex, if the positions $2i + 1$ to $2i + 4$ are called

fixed r , then every student (except for the first $2r$ ones who are directly called by Steffi) will be called by exactly r many students in the ideal case. Hence, as long as at most $r - 1$ of these are not calling, all reliable students will still be reached.

Now, we have come to a point where it would be helpful to conduct some experiments. For a given number x (e.g., 10) of unreliable students, who are assumed to be randomly distributed over the list, we want to determine the minimum value of r for which the probability that all reliable students are reached is still above, say, 90%. In order to determine this r , one can use the algorithm presented below. This algorithm does not emulate the dissemination of information (that runs concurrently in reality) but just determines whether under the given communication rule all reliable students can be reached. For this it suffices to run the for-loop in line 6 till $N/2$ since students with larger list positions will not call any other student. The algorithm is based on an array A that is defined as follows:

- A : array $[1..N]$ of integers; $A[i]$ counts, for a reliable student at position i , the number of calls that student would get from other reliable students. N is the total number of students.

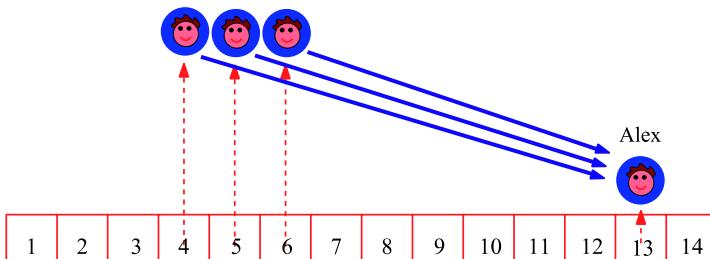


Fig. 22.2. The r students that will call Alex in the ideal case for $r = 3$

- For every reliable student, $A[i]$ is initially set to 0.
- For all unreliable students at position i , $A[i]$ will initially be set to $-r$ (so that even after r calls there will not be a positive value in $A[i]$).

Algorithm for r -fold information dissemination

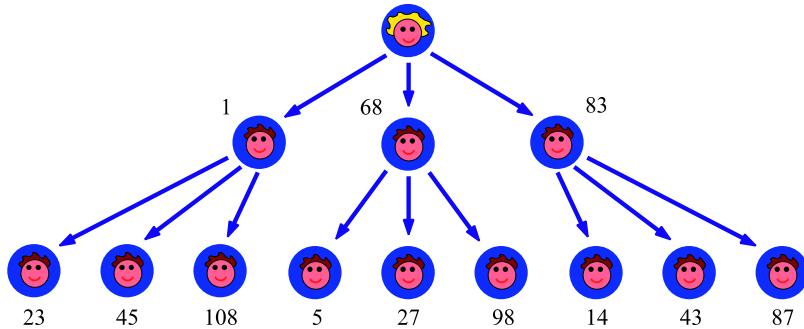
```

1  procedure BROADCAST ( $r$ )
2  begin
3    for  $j := 1$  to  $2 * r$  do      // Steffi calls students 1 to  $2r$ 
4       $A[j] := A[j] + 1$ 
5    endfor
6    for  $i := 1$  to  $N/2$  do      // Student  $i$  calls  $2i + 1$  to  $2i + 2r$ 
7      if  $A[i] > 0$  then      // if call has been received
8        for  $j := 2 * i + 1$  to  $2 * i + 2 * r$  do
9          if  $j \leq N$  then  $A[j] := A[j] + 1$ 
10         endif
11       endfor
12     endif
13   endfor
14   // Did it work?
15   for  $i := 1$  to  $N$  do
16     if  $A[i] = 0$  then output "not everybody reached", stop
17     endif
18   endfor
19   output "everybody reached"
20 end
```

After all this thinking, Steffi has more and more fun in inventing new rules. Next, she considers the more challenging case that every student has a different list of all the other students, so all of her prior strategies are not applicable any more. Is there still a fast and robust strategy to reach all of the reliable students if, say, an arbitrary quarter of the students is unreliable?

After some thinking, Steffi has the idea that she, like everybody else who is called the first time, just randomly picks r students on the list and calls them (see Strategy 5).

If Steffi starts with this strategy, then she will certainly inform r students who have not already been called (if all of them are reachable). In the ideal case, all of them are reachable and reliable, so each of them will call r other students. Hence, at best, r^2 students will then be informed. In reality, however, it can happen that a student is called more than once. Since every student will become active only once (otherwise, the calls would never terminate!), this harms the dissemination of Steffi's information. Also, it can happen that unreliable students are called, which will further lower the dissemination of the information. Nevertheless, one can verify through experiments that Steffi's information will reach all reliable students with high probability if r is sufficiently large (but still reasonably small). In order to determine this r , one can use the algorithm below. It is based on two arrays A and C that are defined as follows:



Strategy 5: Every student, including Steffi, calls r random students for $r = 3$

Algorithm for random r -fold information dissemination

```

1  procedure RANDOMBROADCAST ( $r$ )
2  begin
3      for  $j := 1$  to  $r$  do      // calls from Steffi
4          if  $A[C[0][j]] = 0$  then  $A[C[0][j]] := 1$ 
5          endif
6      endfor
7       $continue := 1$       // indicator for newly called students
8      while  $continue = 1$  do
9           $continue := 0$ 
10         for  $i := 1$  to  $N$  do      // search for newly called students
11             if  $A[i] = 1$  then
12                  $continue := 1$ ;  $A[i] := 2$ 
13                 for  $j := 1$  to  $r$  do
14                     if  $A[C[i][j]] = 0$  then  $A[C[i][j]] := 1$ 
15                     endif
16                 endfor
17             endif
18         endfor
19     endwhile
20     // Did it work?
21     for  $i := 1$  to  $N$  do
22         if  $A[i] = 0$  then output "not everybody reached", stop
23         endif
24     endfor
25     output "everybody reached"
26 end

```

- A : array $[1..N]$ of integers; initially $A[i] = -1$ if student i is unreliable and otherwise $A[i] = 0$. N is the number of students.

- If a reliable student i is called for the first time, then $A[i]$ is set to 1, and once he or she has finished all calls, $A[i]$ is set to 2.
- C : array $[0..N][1..r]$ of integers; $C[i][j]$ gives the number of the j th student who is called by student i (Steffi counts here as student 0). C is chosen at random.

Of course, one can think of many other strategies to disseminate information in a group of people, and everybody is encouraged to do so. Which strategy would you have chosen if you were Steffi?

References

1. [http://en.wikipedia.org/wiki/Broadcasting_\(computing\)](http://en.wikipedia.org/wiki/Broadcasting_(computing))

This Wikipedia article gives an introduction to broadcasting and to standard strategies used in this area.

2. C. Diot, W. Dabbous, and J. Crowcroft: *Multipoint Communication: A Survey of Protocols, Functions and Mechanisms*. IEEE Journal on Selected Areas in Communications 15(3), pp. 277–290, 1997.

K. Obraczka: *Multicast Transport Protocols: A Survey and Taxonomy*. IEEE Communications Magazine 36(1), pp. 94–102, 1998.

M. Hosseini, D.T. Ahmed, S. Shirmohammadi, and N.D. Georganas: *A Survey of Application-Layer Multicast Protocols*. IEEE Communications Surveys & Tutorials 9(3), pp. 58–74, 2007.

These articles are recommended for an introduction to the scientific literature on broadcasting.

3. R. Karp, S. Shenker, C. Schindelhauer, and B. Vöcking: *Randomized Rumor Spreading*. In: *IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 565–574, 2000.

This article contains advanced broadcasting methods that are more effective but also more complex than the strategies presented here. It is recommended to everyone interested in learning about the newest results in this field and who is not afraid of mathematical formulas.

Converting Numbers into English Words

Lothar Schmitz

Universität der Bundeswehr München, Munich, Germany

The problem we are considering here is how to convert numbers into English words. This is what we naturally do when talking to somebody or when filling out a cheque. An amount of, say, \$ 31,264 would be pronounced as “thirty-one thousand two hundred and sixty-four dollars.” In contrast, telephone numbers are often articulated digit by digit. So tel. 31264 would be pronounced as “telephone three-one-two-six-four.”

To indicate a distance of 1,723 miles to New York, the software of a modern GPS route guidance system would not use telephone style

one-seven-two-three miles to New York

Rather, we expect its friendly voice to say (like we do)

one thousand seven hundred and twenty-three miles to New York

A route guidance system for outer space would have to be able to pronounce very large numbers. For example, 12,345,678,987,654,321 would have to be spoken as

twelve quadrillion

three hundred and forty-five trillion

six hundred and seventy-eight billion

nine hundred and eighty-seven million

six hundred and fifty-four thousand

three hundred and twenty-one

In principle, we all can do that (given the names of very large cardinals). But how would a computer program solve this problem? Here, the details will probably turn out to be tricky! We start by precisely specifying the problem to be solved.

Problem: Given a natural number x satisfying $1 \leq x < 10^{27}$ generate the English wording for x . For simplicity, we assume that numbers of this size can be represented in the programming language we are using and that comparison

operators as well as basic arithmetic operations like addition, subtraction, multiplication and integer division are available for these numbers.

Stepwise Development of an Algorithm

The simplest method would be to store for each number its associated English wording and to retrieve the wordings on demand. Because of its enormous storage requirements this approach is not viable. Note that memorizing such an amount of data would overstrain our human brains as well. Instead, we systematically generate number wordings each time we need them. Obviously, this allows us to keep far less data in our brains.

Now let us try to become aware of and explicitly describe the method we are unconsciously applying ourselves when generating number wordings! How does that method work?

Usually, large numbers are separated by commas into groups of three digits, starting with the three last digits – those with the lowest weight. This indicates how our mental processing works: We first split large numerals into groups of three digits each. For the last example this gives:

ones:	321
thousands:	654
millions:	987
billions:	678
trillions:	345
quadrillions:	12

We observe that each group denotes an integer between 0 and 999. The left-most group may have fewer than three digits (but at least one digit).

Now the groups are processed from left to right. For each group first the wording of its associated number (an integer between 0 and 999) is generated. To this an indication of the group's weight is attached: “quadrillions,” “trillions,” “billions,” etc. A weight indication for ones is simply left out.

When generating the wording of a number between 0 and 999 we process the digits from left to right, i.e., in decreasing weight order: hundreds, tens, and ones. These weights independently may occur or not. Hundreds are joined to the following digits (if any) with an “and.” Likewise, tens and following ones are joined with a hyphen. There are a number of special cases like “twelve” and “seventeen”, where tens and ones are merged into one word. More subtleties will turn up later in the program development.

Splitting Numbers into Three-Digit Groups ...

In order to split a number into three-digit groups, we repeatedly divide the number by 1000 and thus each time obtain the next three-digit group. STEP 1

below additionally computes the number i of three-digit groups contained in the given *number*.

STEP 1: Splits number into three-digit groups. The three-digit groups are stored in array *group*: the group with weight 1 in *group*[0], the group with weight 1,000 in *group*[1], the group with weight 1,000,000 in *group*[2], and so on. Variable i records the number of three-digit groups found so far.

```

1   i := 0      // initially no group found
2   while number ≥ 0 do
3       group[i] := number mod 1000    // remainder and ...
4       number := number/1000        // quotient when dividing by 1000
5       i := i + 1      // one more group found
6   endwhile
```

... and Generating the English Words

In STEP 2 below, the more complex parts are deferred to the auxiliary functions GENERATEGROUP and GENERATEWEIGHT, which compute the English wording of a three-digit group and its weight, respectively.

STEP 2: Generating the English words. Indices in array *group* range from 0 to $i - 1$, for the variable i computed in STEP 1. The index of the leftmost group (i.e. the one to be translated first) is $i - 1$. Variable *text* records the English number text generated so far. The & operator joins (“concatenates”) two pieces of text into one piece.

```

1   text := ""      // initially text is empty
2   i := i - 1      // index of the leftmost group
3   while i ≥ 0 do
4       text := text & GENERATEGROUP(group[i])    // generate words ...
5       text := text & GENERATEWEIGHT(i)
            // ... for group and its weight
6       i := i - 1      // on to the next group
7   endwhile
```

After executing both STEP 1 and STEP 2 in order, variable *text* contains the English wording of the given number, as required!

Function generateGroup

If a three-digit group denotes the value 0, it does not contribute to the English wording of the number. This is demonstrated by an example: The number 1,000,111 is spoken “one million one hundred and eleven.” The middle group 000 denoting the value 0 indeed does not show in the generated text.

Three-digit groups denoting values greater than 0 are split into three digits: digit h with weight 100, digit t with weight 10, and o with weight 1.

Here, we cannot proceed without knowing the English names of the Arabic digits and a few other names that are used to systematically build up the

English numbers. Since numbers less than 20 are not built as regularly as numbers beyond 19, we simply store the full names of numbers less than 20 in an array *lessThan20* of strings. Likewise, we store the names of multiples of 10 (between 20 and 90) in an array *times10* of strings. This results in:

Declaration of arrays with small numbers (between 1 and 19) and multiples of 10. For i between 1 and 19 we find the name of i in *lessThan20*[i]. For j between 2 and 9 the element *times10*[j] contains the name of $j \cdot 10$.

```

1  lessThan20: array [0..19] of string :=
2    [ "", "one", "two", "three", "four", ..., "eighteen", "nineteen" ]
3  times10: array [2..9] of string :=
4    [ "twenty", "thirty", "forty", ..., "eighty", "ninety" ]
```

We are now prepared to define the core function GENERATEGROUP, which generates the English wording of a number between 0 and 999. If (part of) a number has value 0, its generated text will be empty. The text will be built up in the string variable *words*, which initially is empty. First, digit h is translated, then the rest r of the number. If $r < 20$, the translation is taken from array *lessThan20*[r]. Otherwise, r is split into the two digits, t and o , which are translated into words using the arrays *times10* and *lessThan20*, respectively. Non-empty hundreds and rests are joined with the word “and”. Likewise, non-empty translations of t and o are joined with a hyphen. Keep in mind, that practically all number components may be missing. Note how spaces separating words are inserted into the text.

Translate a *number* between 0 and 999 into English text.

```

1  function GENERATEGROUP(number)
2    h := number/100
3    r := number mod 100
4    t := r/10
5    o := r mod 10
6    words := ""
7  begin
8    if h > 0 then words := words & lessThan20[h] & "hundred " endif
9    if h > 0 and r > 0 then words := words & "and " endif
10   if r < 20 then
11     // for r = 0 this works because of lessThan20[0] = ""
12     words := words & lessThan20[r]
13   else
14     if t > 0 then words := words & times10[t]
15     if t > 0 and o > 0 then words := words & "-"
16     if o > 0 then words := words & lessThan20[o]
17   endif
18   return words
19 end
```

Function generateWeight

The English names of the weights, like the names of Arabic digits, must simply be known. For this purpose we introduce an array *weight* of strings. Recall that a weight of 1 is “not spoken.”

Declaration of array *weight*.

```

1   weight : array [1..8] of string :=
2       [ "", "thousand", "million", "billion", "trillion",
3         "quadrillion", "quintillion", "sextillion", "septillion" ]
```

Compared to other languages, English allows the text of numbers to be generated very systematically, almost without grammatical irregularities. One exception is that if the last part is less than one hundred it is joined to the preceding text with an “and” even if there are no preceding hundreds. For example, 4,000,001 is pronounced as “four million *and* one” and 5,004,003 as “five million four thousand *and* three.”

Function GENERATEWEIGHT below takes all this into account and also inserts blanks to properly separate the wordings of weights and three-digit groups from each other. If you want to change any of these features, function GENERATEWEIGHT is the place for modifications.

Generate the weight wording for the *i*th group. Ensure that all words are separated by blanks.

```

1   function GENERATEWEIGHT(i)
2       words := ""
3   begin
4       if group[i] > 0 then
5           words := " " & weight[i] & " "
6       endif
7       if i = 1 and group[0] < 100 and group[0] > 0 then
8           words := words & "and"
9       endif
10      return words
11  end
```

Lessons Learned

We now understand how to program the speech output for a GPS route guidance system. Since we have tried to model our algorithm using our own, intuitive approach we now also better understand the way humans generate English number texts.

It is surprising how little we have to know “by heart” in order to be able to generate a multitude of English numbers. All primitive names “known to” the algorithm are stored in the arrays *lessThan20*, *times10*, and *weight*. A total of 36 short strings suffices for generating almost 10^{27} English number names – more than we can use in our whole life (100 years have only about 3,153,600,000 seconds)!

And it goes on like this: In order to increase the range of numbers that can be generated by a factor of 1,000 you need to extend the *weight* array by only one more string entry. You can continue this way as long as you find correct weight names (many are listed on the web site <http://home.hetnet.nl/~vanadovv/BignumbyN.html>).

For other languages, say French or German, that also use a positional number system, number names can be generated “in principle” in the same way. As a consequence, we probably can adapt our algorithm to these languages rather easily. Actually, the original version of our algorithm was written for German numbers. More about that below!

Adaptability to changing requirements is an important property of modern software – for our algorithm obvious modifications would be to extend the number range or to produce numbers in other natural languages as indicated above. Often, this is accomplished by storing data that will probably be changed in some data structure. For our algorithm we have achieved adaptability by storing all name primitives in the three arrays *lessThan20*, *times10*, and *weight*.

What to Read and Try out for Yourself

1. The Wiktionary page <http://en.wiktionary.org/wiki/Appendix:Numbers>

Here, you can learn interesting facts about number representation systems, e.g., what names there are for very large numbers. Among other things, we learn that millions and billions are followed by trillions, quadrillions and quintillions, and that a “googol” corresponds to the number 10^{100} , that is, a 1 followed by 100 zeroes!

2. Richard Bird. *Introduction to Functional Programming Using Haskell*. Prentice-Hall, 1998.

In Sect. 5.1 of this textbook, a Haskell program is developed in detail that allows you to generate English numbers smaller than one million.

3. Write a “real program.”

If you try to code the algorithm in your favorite programming language, in order to obtain an executable program, you will probably find very soon that the finite representations typically used for natural numbers (`int`, `integer`, `long` or something similar) are not suitable for storing the big numbers we are discussing here. Actually, it is much better to store decimal

constants like 12345678987654321 as strings (i.e., “12345678987654321”) in the first place. In the algorithm, arithmetic operations like *number mod* 1000 and *number/1000* are only used to access and delete the last three-digit group, respectively.

4. Processing even larger numbers.

As already indicated this algorithm can easily be adapted to handle larger numbers. You only have to extend the array *weight* accordingly. If you want your program to be able to handle novemdecillions (10^{60}), you have to add a dozen strings. Other simple extensions are to include zero and negative numbers.

5. Speaking other languages.

It takes more effort to adapt the program to another natural language: Obviously, all the primitive names (i.e., the contents of the three arrays *lessThan20*, *times10*, and *weight*) have to be replaced. The trickier part is to adapt the functions GENERATEGROUP and GENERATEWEIGHT to the grammatical peculiarities of the new target language. For example, in German numbers the order of the last two digits of every three-digit group is inverted, like in “one-and-twenty.” Also, the different grammatical genders of words and different singular and plural forms require different versions of the *weight* array to be used. If you buy the German original of this book (*Taschenbuch der Algorithmen*, Springer, 2008), you can find out all the details.

Majority – Who Gets Elected Class Rep?

Thomas Erlebach

University of Leicester, Leicester, UK

Adam stares at the pile of papers in front of him. His class has just finished voting in the election of a class rep. All pupils have written the name of their preferred candidate on a piece of paper, and Adam has volunteered to count the votes and determine the result of the election. Prior to the election the class agreed that a candidate should become class rep only if more than half the class voted for him or her. If none of the candidates wins the absolute majority of the votes, the election will have to be repeated. Adam's task is now to find out whether any candidate has received more than half of all the votes.

How should Adam approach this task? He doesn't think about it much and decides to use the most straightforward method, namely putting down the names of all candidates that receive votes on a piece of paper, and keeping a tally of how many votes each of them has received. He picks up each of the ballot papers in turn to see which name has been written on it. If the name is not yet on his sheet, he writes down the name and puts one tally mark next to it. If the name is already on his sheet, he simply adds an extra tally mark next to that name. When Adam is done with all the ballot papers, his sheet looks as follows:

Shayna	
Liam	{ }
James	{
Kate	{ }
Kevin	{ }
Hannah	- { }
Laura	{

Now Adam looks for the name that has the most tally marks. If the number of those tally marks is larger than half the class size, that candidate is the winner of the election. Otherwise, none of the candidates has won the absolute majority, and the election needs to be repeated. Adam sees that

Hannah has received the most votes, namely 14. Adam's class has 27 pupils. As 14 is more than 13.5 (half the class size), Hannah has won the absolute majority and is therefore elected class rep. Hannah has already been class rep in the previous year, and she will surely continue to do well in that role. All the pupils and the teacher congratulate Hannah on winning the election.

Later, Adam starts to think about the counting of the votes once more. He wonders how good the method that he had used was. For each ballot paper he had to look through the list of names on his sheet and add a tally mark or even a new name to the list. Doing this for 27 votes was fine, but in a larger election that would surely become quite tedious. Just imagine an election with hundreds or thousands of votes! The list of names could then also get very long, and consequently it would take quite long to check for the name on a ballot paper whether it is already in the list or not. Furthermore, Adam's method of counting the votes has produced more information than was required: He didn't just determine the winner of the election, but he also counted the votes for each of the other candidates. The latter information wasn't really necessary for solving his task. Maybe it would have been possible to avoid determining unnecessary information and solve the task with less effort?

Let us remark here that it is also often important for data protection reasons to avoid unnecessary collection and processing of data and to generate only the information that is required to solve the task at hand; this significantly reduces the danger of misuse of personal data. A more detailed discussion of data protection issues is beyond the scope of this chapter, however.

Adam has recently become interested in algorithms and knows by now that there are often methods that are much faster than the most obvious method for solving a task. Therefore, he decides to investigate this further and find out whether there is a faster method for checking absolute majority in an election. Together with Laura, who is also interested in algorithms, Adam searches through various books on algorithms to see if they can find out something about the majority problem.

Majority Algorithm

Laura and Adam indeed find information about the majority problem, i.e., the problem of determining from among N given elements the majority element (the element that occurs more than $N/2$ times, if it exists). They come across the description of the following algorithm.

Majority Algorithm

- 1 Use a stack of elements that is initially empty.
- 2 Phase 1: Process one by one each of the N given elements and execute for each element X the following:
 - 3 If the stack is empty, put X on top of the stack.
 - 4 Otherwise compare X with the top element of the stack. If X and that element are identical, put X on top of the stack; if the two elements differ, remove the top element from the stack.
- 5 If the stack is empty, report that there is no majority element.
- 6 Phase 2: Otherwise take the top element Y of the stack and count how often Y occurs among all N given elements. If Y occurs more than $N/2$ times, output Y as answer. If Y does not occur more than $N/2$ times, report that there is no majority element.

Laura and Adam are surprised that this should work. That algorithm would indeed significantly simplify the process of determining the winner in a large election: When processing a ballot paper, the name on the ballot paper would only have to be compared with a single other name, namely the name of the ballot paper on top of the stack. In the end one would have to go through all the ballot papers a second time to count how often the name determined in Phase 1 occurs, but that wouldn't be too time-consuming either. The algorithm appears very interesting, but Laura and Adam still doubt whether it really works correctly. It seems that at the end of Phase 1 the top of the stack might contain an arbitrary element that happened to occur a couple of times among the last elements that were processed in Phase 1. A completely different element could then be the majority element. Laura and Adam are still doubtful and decide to first try out how the algorithm processes a few sample inputs. For example, they consider an input consisting of the following $N = 7$ elements (for the sake of simplicity, we use capital letters to denote elements): B, B, A, A, C, A, A. The algorithm executes Phase 1 as shown in the following table. Each line of the table represents one step of the algorithm, and the current elements in the stack are specified in each line by listing them in the order from the bottom to the top of the stack:

Stack	Considered element	Action
empty	B	put B on the stack
B	B	put B on the stack
B, B	A	remove B from the stack
B	A	remove B from the stack
empty	C	put C on the stack
C	A	remove C from the stack
empty	A	put A on the stack
A		

Indeed, in the end there is an A at the top of the stack. In Phase 2 the algorithm counts how often A occurs among the given elements. Since A occurs four times among the $N = 7$ elements B, B, A, A, C, A, A, the algorithm correctly recognizes and outputs A as majority element. Thus, the algorithm has worked correctly for this example. One also notices that in each step the elements on the stack are all identical; actually this has to be the case because the algorithm puts an element on the stack only if the stack is empty or the element is identical to the element at the top of the stack.

How does the algorithm behave if the input does not contain a majority element, for example, if the input is A, B, C, C? Here, C occurs twice among the $N = 4$ elements, but a majority element would have to occur strictly more than $N/2$ times. Phase 1 of the algorithm would proceed as follows:

Stack	Considered element	Action
empty	A	put A on the stack
A	B	remove A from the stack
empty	C	put C on the stack
C	C	put C on the stack
C, C		

This time we have a C at the top of the stack when Phase 1 ends. This shows that Phase 2 of the algorithm is really necessary. In Phase 2 the algorithm counts how often C occurs in the input. Since C occurs only twice, the algorithm reaches the correct conclusion that the input does not contain a majority element.

How does the algorithm deal with an input of the form A, A, A, B, B, in which an element different from the majority element occurs several times near the end of the input? Even this does not fool the algorithm:

Stack	Considered element	Action
empty	A	put A on the stack
A	A	put A on the stack
A, A	A	put A on the stack
A, A, A	B	remove A from the stack
A, A	B	remove A from the stack
A		

At the end there is an A at the top of the stack, so the algorithm finds the correct majority element once again. Somehow the algorithm seems to always produce the correct output, although Laura and Adam still don't understand completely why it actually works. They take another look at the algorithms book. There they find a proof for the correctness of the algorithm. At first the proof seems quite complicated and difficult to comprehend, but Laura and Adam go through it several times and help each other by explaining the parts of the proof that they so far understand. Finally, they are convinced that the algorithm always works correctly.

Correctness of the Majority Algorithm

The proof of correctness of the majority algorithm can be summarized as follows: If the algorithm outputs an element X , then that element must indeed be the majority element, because the algorithm has verified in Phase 2 that X occurs more than $N/2$ times. Thus, the only case in which the algorithm could possibly go wrong is when the input contains a majority element X but the algorithm wrongly reports that the input does not contain a majority element. This can only happen if at the end of Phase 1 the stack is empty or an element different from X is at the top of the stack. The following considerations show that this is impossible.

Consider an arbitrary input with N elements, among which more than $N/2$ elements are identical to X . Assume that at the end of Phase 1 there is no X at the top of the stack. As at any time all elements of the stack are identical, this means that at the end of Phase 1 there is not a single X in the stack. Therefore, there are only the following two possibilities for what could have happened to each X in the input:

1. When the X was processed, the stack contained one or several elements different from X . The algorithm did not put X on the stack, but removed an element Y from the stack.
2. When the X was processed, the stack was empty or contained elements identical to X . Therefore, the X was put on the stack. Since there is no X in the stack at the end of Phase 1, there must have been an element Z that came after the X and caused the X to be removed from the stack.

Based on these considerations, each element X can be assigned to an element different from X : In the first case X is assigned to the respective element Y , and in the second case to the respective element Z . In addition, one can see that no two elements X are assigned to the same element different from X . From this we can deduce that there were at least as many elements different from X as there were elements identical to X . Since there are only N elements in total, this contradicts the assumption that the input contains more than $N/2$ elements identical to X . Therefore, it is impossible that X is the majority element but there is no X at the top of the stack at the end of Phase 1. Consequently, there must be an X at the top of the stack when Phase 1 ends, and the algorithm correctly recognizes X as majority element.

How Many Comparisons Are Necessary?

It is also interesting to ask how many comparisons between elements a majority algorithm needs to make in order to solve the problem. Here, the term ‘comparison’ refers to the operation of checking whether two elements are identical or not. The majority algorithm described above performs at most

$N - 1$ comparisons in Phase 1: The first element is put on the stack without a comparison, and when each of the remaining elements is processed, that element can be compared only to the one element that is at the top of the stack at that time. Phase 2 can also be carried out with at most $N - 1$ comparisons. Altogether the algorithm makes at most $2N - 2$ comparisons if the input consists of N elements.

The natural question now is, Can we do better, i.e., are fewer than $2N - 2$ comparisons sufficient to solve the problem? The answer is yes, because the following variation of the majority algorithm never needs more than $\lceil 3N/2 \rceil - 2$ comparisons. (The notation $\lceil . \rceil$ means that the value gets rounded up to the next integer; for example, for $N = 5$ we have $\lceil 3N/2 \rceil = \lceil 15/2 \rceil = \lceil 7.5 \rceil = 8$, and for $N = 6$ we have $\lceil 3N/2 \rceil = \lceil 18/2 \rceil = \lceil 9 \rceil = 9$.)

Refined Majority Algorithm

- 1 Use two stacks of elements that are initially empty.
- 2 Phase 1: Process one by one each of the N given elements and execute for each element X the following:
 - 3 If Stack 2 is not empty and X is identical to the element at the top of Stack 2, then put X on Stack 1.
 - 4 Otherwise put X on Stack 2 and, if Stack 1 is not empty, remove the top element of Stack 1 and put it on Stack 2.
- 5 Assume that at the end of Phase 1, the top element of Stack 2 is a Y .
- 6 Phase 2: While Stack 2 is not empty, repeat the following operations:
 - 7 Compare the element at the top of Stack 2 with Y .
 - 8 If the top element of Stack 2 is identical to Y , remove two elements from the top of Stack 2. (If Stack 2 contains only a single element, remove it from Stack 2 and put it on Stack 1.)
 - 9 Otherwise (i.e., if the top element of Stack 2 is different from Y) remove the top element from Stack 1 and the top element from Stack 2. (If Stack 1 was already empty and we can't remove an element from Stack 1, terminate and report that there is no majority element.)
- 10 If Stack 1 is not empty, output Y as majority element. Otherwise report that there is no majority element.

We see that even the description of the individual steps of the refined majority algorithm is rather complicated. It is not only difficult to become convinced that the algorithm is correct, it is also a challenge to work through the steps of the algorithm correctly for an example input. To better understand the refined majority algorithm, let us first consider the input

B, B, A, A, C, D, A, A, A

and see how the algorithm executes Phase 1 (in each line, the contents of the two stacks are listed in the order from bottom to top):

Stack 1	Stack 2	Considered element	Action
empty	empty	B	put B on Stack 2
empty	B	B	put B on Stack 1
B	B	A	put A on Stack 2, remove one B from Stack 1 and put it on Stack 2
empty	B,A,B	A	put A on Stack 2
empty	B,A,B,A	C	put C on Stack 2
empty	B,A,B,A,C	D	put D on Stack 2
empty	B,A,B,A,C,D	A	put A on Stack 2
empty	B,A,B,A,C,D,A	A	put A on Stack 1
A	B,A,B,A,C,D,A	A	put A on Stack 1
A,A	B,A,B,A,C,D,A		

Indeed, the majority element A is at the top of Stack 2 when Phase 1 ends. Phase 2 now executes as follows:

Stack 1	Stack 2	Action
A,A	B,A,B,A,C,D,A	The element at the top of Stack 2 is identical to A (this must be the case, because A was chosen to be the element that was at the top of Stack 2 when Phase 1 ended), so remove the two elements (D,A) from Stack 2
A,A	B,A,B,A,C	The element C at the top of Stack 2 is different from A, so remove one element from Stack 2 and one element from Stack 1
A	B,A,B,A	The element at the top of Stack 2 is identical to A, so remove the two elements (B,A) from Stack 2
A	B,A	The element at the top of Stack 2 is identical to A, so remove the two elements (B,A) from Stack 2
A	empty	

Now Stack 2 is empty and Phase 2 ends. Since Stack 1 is not empty, the algorithm correctly outputs A as the majority element. We also see that the algorithm has made only four comparisons in Phase 2. In the first of the four comparisons the result of the comparison was already clear in advance (as explained by the remark in brackets), so the number of comparisons that actually need to be performed in Phase 2 is only three.

Let us briefly sketch the analysis that shows that the refined majority algorithm is correct. Essentially, the role of the stack in the first majority

algorithm is now performed by Stack 1 together with the top element of Stack 2. Furthermore, the elements in Stack 1 are identical at any time (and are also identical to the top element of Stack 2). Besides, it can never happen that two identical elements are placed in Stack 2 directly next to each other (i.e., one directly on top of the other). This implies that the element Y that is at the top of Stack 2 at the end of Phase 1 is the only possible candidate for a majority element. Each iteration of Phase 2 then removes two elements, one of which is identical to Y and one of which is different from Y . It follows that Y is the majority element if and only if Stack 1 is not empty at the end of Phase 2 (and thus still contains at least one Y). This establishes the correctness of the refined majority algorithm. Regarding the number of comparisons, we find that the algorithm makes at most $N - 1$ comparisons in Phase 1 and at most $\lceil N/2 \rceil - 1$ comparisons in Phase 2 (the latter holds because in each iteration of Phase 2 two elements are removed, but only one comparison is made; this can also be seen in the example above). This means that the algorithm needs at most $\lceil 3N/2 \rceil - 2$ comparisons to identify the majority element (or to detect that there is none).

Can we do even better than that? This time the answer is negative. One can prove that every algorithm requires at least $\lceil 3N/2 \rceil - 2$ comparisons on some input with N elements. Consequently, it is impossible to beat the refined majority algorithm with respect to the number of comparisons that it needs in the worst case.

Applications and Extensions

The majority problem occurs not only in the context of elections, but also in various other applications. For example, consider safety-critical computations where a correct result is extremely important. Such computations can be carried out by N different processors in parallel, and the majority element of the N solutions computed by the different processors is then taken as the result. This method ensures that as long as less than half of the processors are faulty and produce a wrong solution, one still obtains the correct result.

Among N given elements, an element is the majority element if it occurs more than $N/2$ times. As a generalization of this concept, one can consider *frequent* elements, i.e., elements that occur more than N/K times, where K is a fixed number. For example, for $K = 10$ these are elements that occur with a frequency larger than 10%. Identifying frequent elements is a very relevant problem in the context of monitoring Internet traffic, because one wants to know which applications or users produce the largest amount of traffic. As data packets need to be processed extremely quickly, one needs an algorithm that can handle each new packet in the shortest possible time. The majority algorithm can be generalized to solve such problems.

What Can We Learn from the Solutions to the Majority Problem?

- The most straightforward solution to a problem is not always the fastest one.
- Often there are clever algorithms that can solve the same task with much less effort.
- It is not always easy to see whether an algorithm produces the correct result for every possible input.
- Sometimes one can prove that for a problem it is impossible to find an algorithm that is better than the one we already have.

Further Reading

1. Chapter 1 (Binary Search)

Binary search is a method that allows us to search for a value in a sorted array very quickly. For an election one could store the names of all candidates in a sorted array, together with a counter for each candidate that is initially zero. When processing a vote, one could then find the candidate in the array very quickly and increase the corresponding counter by 1.

2. Chapter 3 (Fast Sorting Algorithms)

Fast algorithms for sorting could be used to quickly rearrange a given list of candidates into sorted order.

3. Further information on the majority algorithm and its extensions can be found in original articles that were published in journals and conference proceedings. Both the refined majority algorithm and the proof that every majority algorithm needs at least $\lceil 3N/2 \rceil - 2$ comparisons on some inputs were presented in the following article:

M.J. Fischer, S.L. Salzberg: *Finding a majority among N votes*. Journal of Algorithms 3(4):375–379, 1982.

The extensions of the majority algorithm to the problem of identifying frequent elements in a data stream are discussed in the following article:

J. Misra, D. Gries: *Finding repeated elements*. Science of Computer Programming 2:143–152, 1982.

Further refinements of the algorithm for identifying frequent elements for the purpose of analyzing packet streams on the Internet are discussed in this article:

E.D. Demaine, A. López-Ortiz, I. Munro: *Frequency estimation of Internet packet streams with limited space*. Proceedings of the 10th Annual Symposium on Algorithms (ESA 2002), LNCS 2461, Springer, 2002, pp. 348–360.

Random Numbers – How Can We Create Randomness in Computers?

Bruno Müller-Clostermann and Tim Jonischkat

Universität Duisburg-Essen, Essen, Germany

Algorithms are clever procedures to efficiently solve a variety of problems. In the preceding chapters we learned numerous examples for “normal” algorithms, like binary search, insertion sort, depth-first search in graphs and finding shortest paths. As a consequence one might assume that algorithms – despite all their cleverness and efficiency – are stubborn and purely replicative procedures yielding in any case perfect and unique solutions. Seemingly algorithms have nothing to do with randomness (or chance). But wait! When applying *QUICKSORT* the pivot-element is proposed to be selected randomly. The One-Time-Pad procedure uses keys that have been randomly chosen. In Fingerprinting numbers are randomly selected.

Tactical and strategic PC games likewise apply algorithms where randomness is highly desirable or even indispensable. Often the computer operates as an opponent, steering its actions by algorithms that imitate meaningful and intelligent behaviors. This is well known from interactive games like *Sims*, *SimCity*, the *Settlement Game*, and *World of Warcraft*. Under identical situations the computer is not expected to show identical behaviors, to the contrary a range of various effects and actions is pleasing. As a consequence, diversity and stimulation increase.

Generating random numbers or random events by throwing a die (getting numbers 1, 2, . . . , 6) or a coin (getting heads or tails) is obviously not possible in a programmed algorithm. On the other hand, can random behavior be programmed? Is it possible to create randomness by algorithms? The answer is as follows: Randomness is imitated by algorithms, which generate numbers that are apparently random. Hence such numbers are often called pseudorandom numbers (although the term random numbers is quite common). Here we consider well-known and approved procedures to construct random number generators. There are many fields of application for random numbers. Here we introduce two examples: A computer game and the so called Monte Carlo simulation for the determination of surface areas.



Fig. 25.1. Rock, paper, scissors (Picture credits: Tim Jonischkat)



Fig. 25.2. Coin: heads or tails; die: 1, 2, . . . , 6; roulette wheel: 0, 1, . . . , 36 (Picture credits: Lukasz Wolejko-Wolejszo, Toni Lozano)

A Tactical Game: “Rock, Paper, Scissors”

As a simple example for a programmed game we can consider an algorithm for the popular game “Rock, Paper, Scissors”. The game works like this: As a player you have to choose one of three options: rock, paper or scissors (Fig. 25.1). Afterwards the algorithm is executed and also yields one of the three options. The evaluation of this round follows these rules: Rock beats scissors, scissors beats paper, paper beats rock. The winner gets one point and the next round follows.

How shall the algorithm proceed? A permanent alternation between “scissors” and “rock” would be a possible (and boring) tactic that is soon deciphered by a human opponent! Obviously the result of the algorithm should be *unforeseeable*, like throwing dice, drawing lottery numbers or spinning a roulette wheel.

A mechanical coupling of an algorithm with dice or a roulette wheel (Fig. 25.2) would be rather troublesome, in any case such a construction would be neither efficient nor clever. Hence an algorithmic procedure is needed that chooses seemingly at random among the three possibilities rock, paper and scissors. In other words, we need a *random number generator*.

Means for the Generation of Random Numbers: Modular Arithmetic

Before we have a closer look at the computation of random numbers, we need the concept of modular arithmetic. The modulo-function (also called mod-function) determines the remainder that is left over under division of

two natural numbers. For example consider the division of 27 by 12; since $27 = 2 \cdot 12 + 3$, we obtain the remainder 3. In the case of two arbitrary natural numbers x and m , we may divide x through m and find as result $x = a \cdot m + r$, where a is called quotient and r is called the remainder. The remainder r is a natural number from the interval $\{0, 1, \dots, m - 1\}$.

Examples for Modular Arithmetic

- $9 \bmod 8 = 1$
- $16 \bmod 8 = 0$
- $(9 + 6) \bmod 12 = 15 \bmod 12 = 3$
- $(6 \cdot 2 + 15) \bmod 12 = 27 \bmod 12 = 3$
- $1143 \bmod 1000 = 143$

In the case of division by 1000 the remainder r is given by the last 3 digits.

Illustration of Modular Arithmetic

Modular arithmetic can be considered like a walk along a circle that carries the numbers $0, 1, 2, \dots, m - 1$. To find out $x \bmod m$ (the perimeter the remainder r when dividing x through m) we start at position 0 and make x steps in clockwise direction. The number of completed revolutions along the circle equals the quotient a ; the position we finally arrive at is the remainder r .

As an example consider the hour hand of a clock in the interval between January 1st, 0 a.m. and January 2nd, 7 p.m. Obviously a period of 43 hours has passed, yielding 3 complete revolutions of the hour hand (each revolution accounting for 12 hours) and another 7 steps that move the hour hand to 7 hours. Since a clock displays hours modulo 12, this number 7 is exactly the result of $r = 43 \bmod 12$. In modular arithmetic addition corresponds to movement in clockwise direction. We consider two examples (cf. Fig. 25.3).

If we start at position $x = 9$ and add the number 6, we have to take 6 steps and arrive at position 3; e.g., it holds that $(9 + 6) \bmod 12 = 3$ (cf. Fig. 25.3, left).

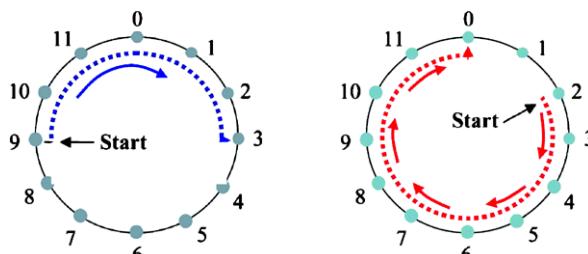


Fig. 25.3. Two examples of modular arithmetic: $9 \bmod 6 = 3$ and $(6 \cdot 2) \bmod 12 = 0$

Multiplying a number x with a factor a can be resolved into a series of additions; if we start at position $x = 2$ and multiply by $a = 6$, we have to add 5 times the value 2, i.e., we have to take 5 two-steps in clockwise direction, leading us to position 0; e.g., it holds that $(6 \cdot 2) \bmod 12 = 0$ (cf. Fig. 25.3, right).

In computer science modular arithmetic may be seen as the “natural” arithmetic because a computer’s storage is always limited (finite); moreover, storage cells are also finite and can only store numbers up to a certain size.

An Algorithm for the Generation of Pseudorandom Numbers

The algorithm as given below employs modular arithmetic and provides random numbers from the interval $\{0, 1, \dots, m - 1\}$. The basic principle is quite simple: Take a starting value x_0 , the factor a , the constant c , and the modulus m ; compute the remainder r as $r = (a \cdot x_0 + c) \bmod m$. We set this value to be the first random number: $x_1 = r$. Now we use x_1 to compute the second random number $x_2 = (a \cdot x_1 + c) \bmod m$; proceeding with x_2 in the same way, we obtain a sequence of random numbers x_1, x_2, x_3, \dots .

This procedure can be written down more precisely as follows:

$$\begin{aligned}x_1 &:= (a \cdot x_0 + c) \bmod m, \\x_2 &:= (a \cdot x_1 + c) \bmod m, \\x_3 &:= (a \cdot x_2 + c) \bmod m, \\x_4 &:= (a \cdot x_3 + c) \bmod m, \\&\text{etc.}\end{aligned}$$

In general notation we achieve an iterative procedure as follows:

$$x_{i+1} := (a \cdot x_i + c) \bmod m, \quad i = 0, 1, 2, \dots$$

To achieve a concrete random number generator, numerical values for factor a , constant c , and modulus m must be supplied; furthermore we need a starting value x_0 , which is sometimes called the seed of the generator.

Setting $a = 5$, $c = 1$, $m = 16$ and $x_0 = 1$ results in these calculations

$$\begin{aligned}x_1 &:= (5 \cdot 1 + 1) \bmod 16 = 6, \\x_2 &:= (5 \cdot 6 + 1) \bmod 16 = 15, \\x_3 &:= (5 \cdot 15 + 1) \bmod 16 = 12, \\x_4 &:= (5 \cdot 12 + 1) \bmod 16 = 13, \\x_5 &:= (5 \cdot 13 + 1) \bmod 16 = 2, \\x_6 &:= (5 \cdot 2 + 1) \bmod 16 = 11,\end{aligned}$$

$$\begin{aligned}x_7 &:= (5 \cdot 11 + 1) \bmod 16 = 8, \\x_8 &:= (5 \cdot 8 + 1) \bmod 16 = 9, \\&\text{etc.}\end{aligned}$$

Obviously this sequence is completely determined by the prescribed calculations. That is, for a given starting value x_0 we will always get the same, strictly defined and reproducible elements; such a behavior is called deterministic. By choosing another starting value x_0 , the entry point into the sequence can be newly selected for another run of the algorithm.

Periodic Behavior

If we continue the above calculation we see that after 16 steps the sequence returns to its start value 1, and that each of the 16 possible numbers from the interval $\{0, 1, \dots, 15\}$ has occurred exactly once. Computing the next 16 values $x_{16}, x_{17}, \dots, x_{31}$, the sequence will repeat itself, and we notice a reproducing behavior, here with a period of length 16. When the modulus m is set to a very large number and furthermore we choose the factor a , and the constant c precisely, larger periods are achieved. In the ideal case we manage to reach a full period of length m . Sometimes programming languages provide a built-in random number generator or provide it through a function library; the programming language Java offers a full-period generator with parameter values $a = 252149003917$, $c = 11$ and $m = 2^{48}$.

Simulation of True Random Number Generators

Pseudorandom numbers from the interval $\{0, 1, \dots, m - 1\}$ are basic for many applications. Examples are the simulation of throwing a coin yielding heads or tails, throwing a die resulting in one of the values 1, 2, 3, 4, 5 and 6, or spinning a roulette wheel providing the 37 possibilities 0, 1, ..., 36.

Let us assume that for each example the random generating devices coin, die and roulette wheel are fair and do produce the corresponding outcomes with probabilities $\frac{1}{2}$, $\frac{1}{6}$ and $\frac{1}{37}$. To simulate dice throwing we need a procedure to transform a random number $x \in \{0, 1, \dots, m - 1\}$ in a number $z \in \{0, 1\}$, where 0 stands for “heads” and 1 for “tails”. An easy procedure could be a mapping of “small” numbers to the value 0, and of “large” numbers to 1, or mathematically expressed: $z := 0$ if $x < \frac{m}{2}$, $z := 1$ if $x \geq \frac{m}{2}$. In the case of the roulette wheel we could use as transformation procedure $z := x \bmod 37$, and for the die $z := x \bmod 6 + 1$.

The Algorithm for Rock, Paper, Scissors

Now, after all, we return to our tactical game. We need an algorithm that decides randomly (or at least apparently random) between the three prospects:

rock, paper and scissors. To this end we use a random number generator that generates for each round a random number x ; by the calculation $z := x \bmod 3$ we obtain a new number that can just provide one of the values 0, 1 and 2. Dependent on the actual value of z the algorithm chooses rock (0), paper (1) or scissors (2). This algorithm has been implemented as a small program, the rock-paper-scissors-applet.

A selection of four different (pseudo) random number generators is available; the first one is given just as the extreme case of a fixed sequence of numbers 0, 1 and 2.

1. Deterministic: The fixed sequence of numbers 2, 0, 1, 1, 0, 0, 0, 2, 1, 0, 2
2. RNG-016: $a = 5$, $c = 1$, $m = 16$ and start value $x_0 = 1$ (period is of length 16)
3. RNG-100: $a = 81$, $c = 1$, $m = 100$ and start value $x_0 = 10$ (period is of length 100)
4. Java Generator: The generator of programming language Java: “java.util.Random”

The algorithm NEXTRANDOMNUMBER uses the input value x from the interval $\{0, 1, \dots, m - 1\}$ and the constant parameters a , c and m . With **return** the computed value from the interval $\{0, 1, \dots, m - 1\}$ is passed back. This value is the next random number following the input number x .

```

1  procedure NEXTRANDOMNUMBER ( $x$ )
2  begin
3      return  $(a \cdot x + c) \bmod m$ 
4  end

```



Fig. 25.4. Scissors cut paper (left), rock is wrapped by paper (right)

Table 25.1. State of game after three rounds: human 2, machine 1

Round	Human	Machine
1	Rock	0 1
2	Scissors	1 1
3	Rock	2 1

This procedure `RANDOMNUMBEREXAMPLE(n)` is to illustrate the usage of the procedure `NEXTRANDOMNUMBER(x)`. Firstly parameters a , c and m are initialized. The start value is set to value 1. Each invocation of procedure `NEXTRANDOMNUMBER(x)` changes the value of x to a new value that is returned and printed out. Additionally the value $x \bmod 3$ is also printed out.

```

1  procedure RANDOMNUMBEREXAMPLE ( $n$ )
2  begin
3       $a := 5; c := 1; m := 16;$ 
4       $x := 1;$ 
5      for  $i := 1$  to  $n$  do
6           $x := \text{NEXTRANDOMNUMBER}(x);$ 
7           $\text{print}(x);$ 
8           $\text{print}(x \bmod 3)$ 
9      endfor
10 end
```

The generated random numbers are as follows:

x_i : 6, 15, 12, 13, 2, 11, 8, 9, 14, 7, ...

x_i modulo 3: 0, 0, 0, 1, 2, 2, 2, 1, 2, 1, ...

The pictures show two possible outcomes of a round “human” versus “machine” (Fig. 25.4) and the state of the game after several rounds (Table 25.1).

Figure 25.5 illustrates the working principle of the selected random number generator (here RNG-016). The current calculation is highlighted, and furthermore the future values are already visible! Hence, one can forecast the future decision of the machine (a small cheat).

<input type="radio"/> deterministic	<input checked="" type="radio"/> RNG-016	<input type="radio"/> RNG-100	<input type="radio"/> Java generator
	Randomizer: RNG-016		$a=5, c=1, m=16, x_0=1$
	$x_0 = 1$		
	$x_1 = (a * x_0 + c) \bmod m = 6$		
	$x_2 = (a * x_1 + c) \bmod m = 15$	$x_2 \bmod 3 = 0 \rightarrow \text{Rock}$	
	$x_3 = (a * x_2 + c) \bmod m = 12$	0 equals "rock"	
	$x_4 = (a * x_3 + c) \bmod m = 13$	1 equals "paper"	
	$x_5 = (a * x_4 + c) \bmod m = 2$	2 equals "scissors"	

Fig. 25.5. Algorithm of the machine

Monte Carlo Simulation: Determination of Areas Using “Random Rain”

An important field of application for random numbers is the so called Monte Carlo simulation, named after the gambling casino in Monte Carlo. Monte Carlo simulation can be used – for example – to determine the areas of irregular shaped geometric figures by means of “random rain”. The term “random rain” describes the situation where many two-dimensional random points (x, y) are hitting a flat surface like rain drops. A random point on a flat surface is given as an (x, y) -pair of random coordinates, where x -value and y -value are random values from the real interval $[0, 1]$. To this end random values $x \in \{0, 1, \dots, m - 1\}$ are transformed to real values between 0.0 and 1.0; the transformation rule is plainly given as $x := x/(m - 1)$.

If an arbitrary area is placed into a unit square, i.e., all edges have length 1.0, and random points are thrown into the square, a fraction will hit the area whereas the others will miss it. Consider Fig. 25.6 for an example.

An estimate for the area is obtained by counting the hits and calculation of the term $A = (\text{hit count})/(\text{drop count})$. To achieve a good estimate the number of drops should be rather large, say many millions or even some billions of random points must be thrown. Of course, carrying out the algorithm by hand is out of discussion; as a substitute a programmed algorithm running on a computer has no problem to drop some million random points into a square.

As an example for the practical application of the Monte Carlo technique we consider a technique for the determination of the few first digits of the famous mathematical constant $\pi = 3.14159\dots$. This can be done by throwing random points (x, y) into the unit circle. We consider a unit square (a square with side length 1.0 and area 1.0 as well), with an inscribed quadrant of the unit circle. Since the circle has radius $r = 1$ its area is $A = r^2 \cdot \pi = \pi$ and the area of the quadrant is $\frac{\pi}{4}$.

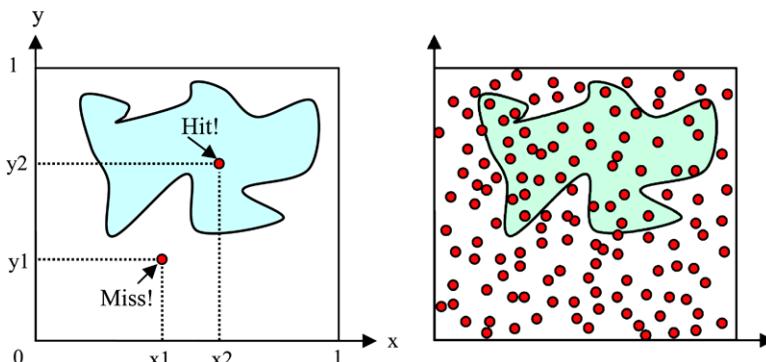


Fig. 25.6. Two random points (x_1, y_1) and (x_2, y_2) , a hit and a miss (left); random rain: Count the number of drops hitting the area (right)

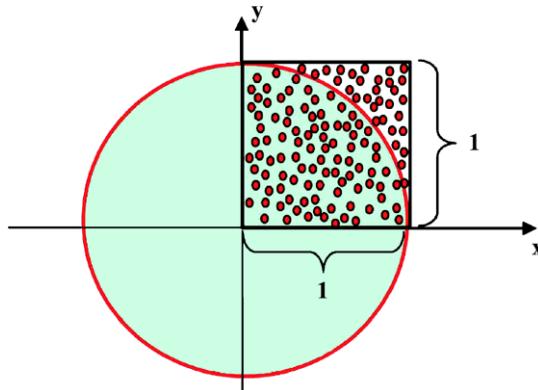


Fig. 25.7. How many points hit the quadrant?

If T is the number of hits in the quadrant and N is the number of hits in the unit square, we can approximately calculate $\pi \approx 4 \cdot \frac{T}{N}$. The figure shows 130 random points where 102 points fall into the quadrant, i.e., we calculate $\pi \approx 4 \cdot \frac{102}{130} \approx 3.1384$. This result is still not very precise, although with 100,000 or some millions or even one billion points the result should become significantly better. This procedure is summarized more precisely in the algorithm RANDOMRAIN.

The algorithm RANDOMRAIN delivers an estimate for the mathematical constant π . We assume a unit square (side length = 1) with area 1.0.

```

1  procedure RANDOMRAIN ( $n$ )
2  begin
3       $a := 1103515245$ ;  $c := 12345$ ;  $m := 4294967296$ ;      // parameters
4       $z := 1$ ;  $hits := 0$ ;      // start values
5      for  $i := 1$  to  $n$  do
6           $z := (a \cdot z + c) \bmod m$ ;
7           $x := z/(m - 1)$ ;
8           $z := (a \cdot z + c) \bmod m$ ;
9           $y := z/(m - 1)$ ;
10         if INCIRCLE( $x,y$ ) then
11              $hits := hits + 1$ 
12         endif
13     endfor
14     return  $4 \cdot hits/n$ 
15 end
```

Comment: The function INCIRCLE() tests whether the point (x, y) is located inside the circle area. Therefore it uses the equation $x^2 + y^2 = r^2$; the point (x, y) is counted as a hit in the unit circle if $x^2 + y^2 \leq 1$.

There are an abundant number of applications of Monte Carlo simulations in engineering and the natural sciences. In computer science the field of randomized (or probabilistic) algorithms developed out of Monte Carlo simulation.

Further Reading

1. A starting point for further reading is Wikipedia:
http://en.wikipedia.org/wiki/Pseudorandom_number_generator
2. Modular arithmetics as used for pseudorandom number generation is essential for other fields within computer science, e.g., for techniques like Public-Key Cryptography (Chap. 16), Fingerprinting (Chap. 19), the One-Time Pad (Chap. 15) and Simulated Annealing (Chap. 41).
3. Other fields of application for random numbers are stochastic simulation programs that are used to investigate the performance of complex systems like computer networks, mobile systems and Web services.

Winning Strategies for a Matchstick Game

Jochen Könemann

University of Waterloo, Waterloo, Canada

I am currently feeling mildly exasperated, and am also somewhat confused. Last night, I found my brother in a suspiciously cheerful mood – which is in general not a good sign! Grinning broadly, he tells me of a “very interesting” new IQ test which he would love me to try. Knowing my brother, my first instinct is to be *very* careful as I remember countless previous occasions just like this that ended up being a set-up. At the same time, however, I am also too curious to let him walk away, and I say, “Sounds interesting. Tell me more about it!”

“Great,” he says and his grin widens. He rummages in his pants pocket, only to produce a small box of matchsticks seconds later. He opens the box, and out fall 18 matchsticks which now lie in between the two of us on the table. “Let’s play a game!” he says, and continues explaining the surprisingly simple rules. He and I would move in turns. The first player decides to remove 1, 2 or 3 matchsticks from the table. Now it is the other player’s turn, and he must take either 1, 2 or 3 of the available matchsticks, and remove them from the table. The game continues like this, and the player who takes the last matchstick loses the game.



“All clear?” asks my brother. He always has to show off and make me feel like the little brother! “Of course!” I say, “After all, the game isn’t all that complicated”. The suspicious grin on his face broadening yet again he says, “Good! Let’s get going then, and since it doesn’t *really* matter, let me start, as I am the oldest of the two of us.” He promptly removes one of the matchsticks from the table, and, accordingly, 17 sticks remain.

Great, so I can’t lose in this turn; no matter whether I remove 1, 2 or 3 matchsticks, there will always be at least 14 remaining sticks, and hence my brother will have to play again. I decide to pick 2 sticks, and it is once again my brother’s turn with 15 remaining matchsticks on the table. He still smirks (I wonder *why*) and grabs two more sticks, which leaves a total of

13 matchsticks on the table. The table below summarizes the current game situation. The left column shows the number of remaining sticks, and the right column has the subsequent move.

Number of remaining matchsticks	Move
 (13)	I take 3 sticks.
 (10)	My brother takes 1 stick.
 (9)	I take 2 sticks.
 (7)	My brother takes 2 sticks.
 (5)	I take 1 stick.
 (4)	My brother takes 3 sticks.

After my brother's last move, there is one last matchstick left on the table. The rules of the game now force me to remove this stick from the table, and I have then lost! "Well, that was clearly just luck!" I say, and ask for a revenge game. Unfortunately, my brother wins this game and the following two as well, and I am starting to get a little frustrated. My brother seems to have an edge in this game. But how does he do that?

Learning from Small Examples

In order to better understand the game, I consider a few small examples. I know that I lose the game if it is my turn and there is only a single matchstick left on the table. But what should I do if there are exactly two sticks on the table and it is my turn? Well, in this case, the solution is rather straightforward: I should pick a single stick, which leaves my brother with one remaining stick, and he has therefore lost the game. This says that I have a *winning strategy* whenever two matchsticks remain, and it is my turn in the

game. Given the above argument, it is not difficult to see that I also have a winning strategy if it is my turn and there are 3 or 4 matchsticks on the table. In the former case, I remove 2 sticks, and in the latter case, I pick 3. In both cases, my brother is faced with a single stick lying in front of him, and he will therefore lose the game.

Let us summarize the insights we gained so far. We now know that a player has a winning strategy if it is his turn, and if there are 2, 3 or 4 matchsticks remaining on the table. In either one of these cases, the player can move strategically, and force the other player to lose the game.

The table below has a column for each of the game situations that we have analyzed so far. The lower entry in column i (which we denote by GS_i) indicates whether the current player has a winning strategy, given that i matchsticks remain.

i	1	2	3	4
GS_i	No	Yes	Yes	Yes

So far so good. But what happens if there are 5 matchsticks on the table? The answer to this question appears to be slightly less obvious in comparison to the previous cases. It is, however, a game situation that interests me quite a bit as it was the second to last configuration I faced in the first game against my brother. Could I have forced him to lose out of this configuration? Let's see! The rules of the game force me to remove at least 1 and at most 3 matchsticks from the table. No matter what I do, my brother will have 2, 3 or 4 matchsticks left on the table. But we already know that he has a winning strategy in each of these situations! Therefore, if my brother plays smartly, he will force me to lose no matter how I move. As long as the other player moves strategically, a player cannot win starting from a game configuration with 5 matchsticks, and thus $\text{GS}_5 = \text{No}$.

If there are 6 matchsticks remaining on the table, I can take either 1, 2 or 3 sticks which leaves my brother with 3, 4 or 5 sticks. Looking at the table, I know that my brother has a winning strategy, given that there are 3 or 4 sticks remaining for his move. However, if I leave him 5 sticks, then he can't win given that I play cleverly. Therefore, I do have a winning strategy, given that there are 6 sticks remaining: I just have to take one stick off the table. Therefore, we have $\text{GS}_6 = \text{Yes}$.

An Algorithm to Compute a Winning Strategy

It is now easy to extend the example calculations above. For example, assume that we have already computed GS_i for $1 \leq i \leq 14$, and that we know whether a player has a winning strategy, given that there are i matchsticks remaining, whenever i is at most 14. If there are 15 matchsticks remaining, then the current player needs to pick 1, 2 or 3 sticks off the table, and would leave 12,

13 or 14 sticks for the next player. If the other player has a winning strategy in each of these situations (i.e., $GS_{12} = GS_{13} = GS_{14} = \text{Yes}$), and if he moves cleverly, then the current player will lose the game; hence, we let $GS_{15} = \text{No}$ in this case. If, however, if $GS_j = \text{No}$ for at least one $j \in \{12, 13, 14\}$, then the current player can force the other player to lose by picking $15 - j$ matchsticks, and hence $GS_{15} = \text{Yes}$.

We obtain the following algorithm to compute GS_1 to GS_x :

WINNINGSTRATEGY(x)

```

1   GS1 = No, GS2 = Yes, GS3 = Yes
2   i := 3
3   while i < x do
4       i := i + 1
5       if GSi-3 = GSi-2 = GSi-1 = Yes then
6           GSi = No
7       else
8           GSi = Yes
9       endif
10  endwhile
```

We can now use this algorithm to compute GS_1 to GS_{18} . We summarize the results of this run in the following table. For brevity, we use “N” for “No” and “Y” for “Yes”.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
GS _i	N	Y	Y	Y	N	Y	Y	Y	N	Y	Y	Y	N	Y	Y	Y	N	Y

Wonderful! Now I know that my brother has a winning strategy whenever he starts the game, and 18 matchsticks remain. But how exactly does he figure out what number of matchsticks he needs to remove in each move?

Let us recall the first game between me and my brother. In this game, we start with 18 matchsticks, and my brother has the first move. Looking at the table, we see that $GS_{18} = Y$, and he therefore has a winning strategy. He can therefore win against me, as long as he makes the right moves. In his first move, he will have to take 1, 2 or 3 matchsticks off the table. What should he do? Another look at the table tells us that I have a winning strategy if my brother leaves me 15 matchsticks. My brother will naturally want to avoid this, and probably won't remove three matchsticks. Similarly, he won't take 2 matchsticks which would leave me with 16 sticks, and the table tells us that I can win the game from this situation. The sole remaining option for my brother is to remove 1 matchstick, and that will leave me 17 matchsticks for my move. According to the table ($GS_{17} = N$), I will lose the game from here if my brother makes no mistake.

“What was it that I did, after my brother removed 1 matchstick in his first move?” Right! I picked up 2 of the remaining 17 matchsticks, leaving 15 for my brother. The table tells us that my brother still has a winning strategy,

starting from a configuration with 15 sticks. Furthermore, removing 2 is the only option for him to maintain his advantage; i.e., $GS_{12} = GS_{14} = Y$, and $GS_{13} = N$. “So, after all this ‘game’ was a set-up and not an IQ-test! My brother knew exactly what to do in order to make me lose!”

The Running Time of the Algorithm

How many elementary operations (comparisons and assignments) does the algorithm need in order to compute GS_x ? The inner **while**-loop is executed $x - 3$ times. Every iteration in turn accesses three previously computed terms GS_{i-3} , GS_{i-2} and GS_{i-1} , in order to compute GS_i . Therefore, three comparisons and two assignments suffice for steps 4–9 of the algorithm, and the total number of steps needed is

$$(x - 3) \cdot (3 + 2) + 4 = 5x - 11.$$

The running time is therefore proportional to the given number of matchsticks. This is probably not a problem if we play the game in practice with the matchsticks in front of us on the table. Naturally, we could use a computer to execute our algorithm, and that would enable us to compute GS_x for much larger numbers x . For example, we could use a computer to determine who wins starting from $x = 9876543210$ matchsticks. Even though this number can be described with 10 digits, the algorithm would need roughly $5 \cdot 9876543210$ steps to find an answer; this is already quite a large number!

It would be a lot nicer if the number of steps in our algorithm were proportional to the *number* of digits needed to write x , instead of being proportional to the number itself. The number of digits needed to write x depends on the *representation* used. Computers typically store numbers in so-called *binary* representation, in which x corresponds to a string

$$a_k a_{k-1} \dots a_0$$

of 0, 1-bits such that

$$x = \sum_{i=0}^k 2^i a_i.$$

The length of the input for our algorithm is then the length $k + 1$ of the representation of x , and not the value of x . How large is x with respect to its representation length k ? We clearly have

$$2^{\lceil \log_2 x \rceil + 1} \geq 2^{\log_2 x + 1} = 2 \cdot x, \quad (26.1)$$

and we therefore conclude that $a_i = 0$ for all $i > \lceil \log_2 x \rceil$. In other words, the binary representation of x has at most $\lceil \log_2 x \rceil + 1$ bits. The largest number x that can be represented using $k + 1$ bits is

$$\sum_{i=0}^k 2^i = 2^0 + 2^1 + \cdots + 2^k.$$

This is a *geometric series*, and one can show that its value is exactly $2^{k+1} - 1$ (we provide a reference to an article with further information below). Substituting $k = \lfloor \log_2 x \rfloor - 1$ we obtain

$$\sum_{i=0}^{\lfloor \log_2 x \rfloor - 1} 2^i = 2^{\lfloor \log_2 x \rfloor} - 1 \leq 2^{\log_2 x} - 1 = x - 1, \quad (26.2)$$

and the binary representation of x therefore needs at least $\lfloor \log_2 x \rfloor + 1$ bits.

The two bounds in (26.1) and (26.2) together imply that the length of the binary representation of x is in the interval $[\lfloor \log_2 x \rfloor + 1, \lceil \log_2 x \rceil + 1]$. The number of operations needed by our algorithm is therefore

$$5x - 11 \geq 5 \cdot 2^k - 11,$$

and this is exponential in the *length* k of the input x . Generally speaking, we call an algorithm efficient if its running time is bounded by a polynomial of the length of its input. Algorithms whose running time is bounded by a polynomial of actual values in the input (like the algorithm in our example) are called *pseudo-polynomial time* algorithms.

By the above reasoning, the algorithm WINNINGSTRATEGY from our example is therefore not a truly efficient algorithm; it is a pseudo-polynomial time algorithm. It seems unlikely that my brother used it to compute his winning strategy in the game against me. Indeed, my brother tells me that it is easy to prove that we have $GS_x = Y$ whenever the remainder of x divided by 4 is 1. Thus, once I have reached a configuration in which I have a winning strategy, I can determine my move easily: whenever my opponent picks y matchsticks, I will need to pick $4 - y$ sticks so that the sum of the sticks taken by him and by me is always exactly 4.

It is easy to check that this formula is correct for the game played between me and my brother, and that it explains why my brother did not need a table or a computer to figure out his moves in his play against me.

Extensions and Background

The matchstick game we discussed in this chapter has numerous extensions. One popular variant is called *Nim* and works as follows: the game starts with several rows of matchsticks. Much like our game, Nim is once again a two-player game. The two players move in turns. The moving player first picks one of the remaining rows of matchsticks, and then takes at least one and arbitrarily many of the remaining sticks in this row. Just like before, the player who picks the last matchstick loses. The analysis of this game is very

similar to the analysis presented before for the simpler (single-row) version. In particular, there is also a *formula* describing the winning strategy for this game.

The game Nim is quite old, and is generally believed to have its origins in China (it is quite similar to the Chinese game *Tsyanshidzi*). The game appeared first in Europe during the 16th century. Its name Nim was coined by Charles Bouton, who published a complete analysis of Nim in 1901.

The algorithm WINNINGSTRATEGY presented here is a relatively simple example of a method which is commonly known as *dynamic programming*. This powerful method was developed during the 1940s by the American Richard Bellman, and is mostly used for the solution of more general optimization problems whose instances can be decomposed into similar but smaller subproblems. For example, in our matchstick game, we showed that the question of whether a player has a winning strategy facing x remaining matchsticks can be reduced to the same question with $x - 1$, $x - 2$ and $x - 3$ remaining sticks.

A far more complex example for dynamic programming is given in Chap. 31, where the method is used to compute the *mutation distance* of two genetic strings.

Further Reading

1. <http://en.wikipedia.org/wiki/Nim>
An article discussing the game Nim.
2. http://en.wikipedia.org/wiki/Dynamic_programming
Dynamic programming on Wikipedia.
3. http://en.wikipedia.org/wiki/Geometric_series
An article discussing geometric series.
4. D.P. Bertsekas: *Dynamic Programming and Optimal Control*, Vols. 1 and 2. Athena Scientific, 3rd edition, 2005.
This comprehensive textbook provides in-depth coverage of dynamic programming.

Scheduling of Tournaments or Sports Leagues

Sigrid Knust

Universität Osnabrück, Osnabrück, Germany

Let us consider a small table tennis club that wants to organize a tournament for six players. Each player has to play against each other player exactly once, and every player should play at most one match per evening. Since each of the six players has to play against five other players, in total $\frac{6 \cdot 5}{2} = 15$ matches have to be scheduled (we must divide by 2 since the match i against j is counted for player i as well as for player j). If every player plays exactly one match each evening, we have three matches per evening, i.e., for all matches in total $\frac{15}{3} = 5$ evenings are needed.

The players start their competition, and every evening each player searches for another player against whom he has not played before. After three evenings the following matches have been performed:

Evening 1	Evening 2	Evening 3
1-2	1-3	1-4
3-5	2-6	2-5
4-6	4-5	3-6

It is easy to see that the remaining 6 matches, 1-5, 1-6, 5-6, 2-3, 2-4, 3-4, cannot be completed within two additional evenings (when the match 1-5 takes place, the two matches 1-6 and 5-6 cannot be scheduled in parallel since each player may only play one match per evening). Thus, the tournament can only be finished during three additional evenings:

Evening 4	Evening 5	Evening 6
1-5	1-6	5-6
2-3	2-4	3-4

Hence, the club needs one more evening for the whole tournament than originally intended. Furthermore, during the last two evenings two players do not play.

Somebody who is interested in sports may notice that similar scheduling problems occur for several sports leagues. For example, in the German soccer league 18 teams play a tournament with 2 half-series, where in each half-series every team plays against every other team exactly once. Usually, in every round (weekend) within $2 \cdot 17 = 34$ weeks every team plays exactly one match. Thus, we may ask the question whether, for any number of teams, a schedule always exists in which all teams play in every round.

Let us now consider this scheduling problem from a more general point of view: Given are an even number n of teams (or players) and $n - 1$ rounds (days for matches). The objective is to determine a schedule such that each team plays against each other team exactly once, and each team plays exactly one match per round. Thus, for each even number n , we ask whether such a schedule exists and how we can construct a corresponding schedule. In the following we show that for each even n a solution exists (i.e., for $n = 18$ as well as for $n = 6$, but also for $n = 100$ or $n = 1024$). Furthermore, we describe an algorithm which constructs such a schedule for any even number n .

Generation of Schedules

In order to describe an algorithm for the generation of schedules we first model our problem using so-called graphs, which are often used in computer science (see also Chaps. 28, 32, 33, 34, and 40). A graph consists of a set of vertices and a set of edges, where an edge connects two vertices. For example, with a graph we may model a network of roads where the roads correspond to the edges and the crossings correspond to the vertices.

For our sports scheduling problem we introduce a vertex for each team; the matches correspond to the edges. For example, for $n = 6$ teams we obtain the graph in Fig. 27.1.

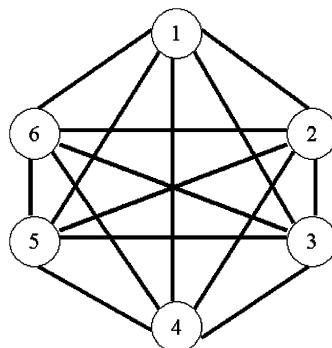


Fig. 27.1. Graph for six teams

Such a graph is also called complete since each vertex is connected with every other vertex by an edge (recall that each team plays against every other team). In order to obtain a schedule we color the edges with the colors $1, 2, \dots, n - 1$, where every color represents a round. Such a coloring is called feasible if it corresponds to a feasible schedule. This is the case if all edges that are connected to the same vertex are colored with different colors (otherwise not every team plays once per round).

For our graph with $n = 6$ vertices, the edge coloring in Fig. 27.2 with $n - 1 = 5$ colors is feasible. A corresponding schedule is shown beside the graph, where the color “red” represents the first round, the color “blue” the second round, etc.

It remains to show how a feasible edge coloring of the complete graph for any even number n can be constructed. In our example let us consider the graph which is obtained after eliminating vertex 6 and all 5 edges that are connected to this vertex. We get a pentagon where the 5 edges on the boundary (1-2, 2-3, 3-4, 4-5, 5-1) are colored differently. If we draw this pentagon as a regular pentagon as in Fig. 27.3 (i.e., all angles at the 5 corners are equal), we see that each edge in the inner part of the pentagon is colored with the same color as the corresponding parallel edge on the boundary. If in an edge coloring always only parallel edges (which are not connected to the same vertex) are colored with the same color, obviously the condition is fulfilled that edges which are connected to the same vertex are colored differently. In our example we also see that for each of the five vertices four colors are used and always one other color is missing (which can be used for the edges to vertex 6).

These considerations can be used to construct a feasible edge coloring of any complete graph having an even number n of vertices. The origin of this procedure is not precisely known, but it is reported that the English pastor and mathematician Thomas P. Kirkman (1806–1895) did similar considerations. Please see the algorithm “Edge Coloring” which iteratively colors sets

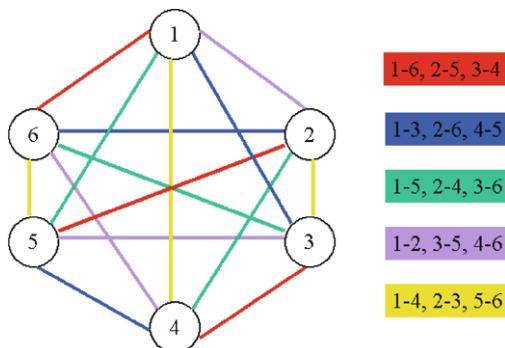
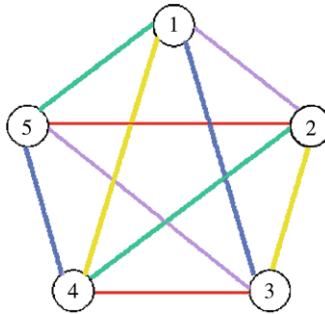


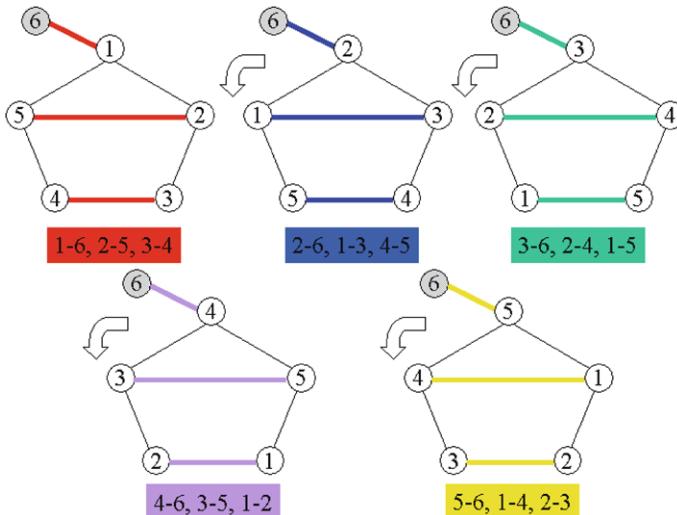
Fig. 27.2. Edge coloring of the graph with six teams

**Fig. 27.3.** Resulting pentagon**Algorithm Edge Coloring (geometrical version):**

1. Build a regular polygon from the vertices $1, 2, \dots, n-1$ and place the vertex n to the top left side beside the polygon.
2. Connect vertex n with the “top” of the polygon.
3. Connect each of the remaining vertices with the opposite vertex on the same height in the polygon.
4. Color the resulting $\frac{n}{2}$ edges with the first color (in the example we color the edges 6-1, 5-2, 4-3).
5. Rotate the vertices $1, \dots, n-1$ of the polygon cyclically anti-clockwise by one position (i.e., vertex 2 is moved to the position of vertex 1, vertex 3 replaces the old vertex 2, ..., vertex $n-1$ replaces the old vertex $n-2$ and vertex 1 replaces the old vertex $n-1$). Vertex n stays at the same place beside the polygon, and the edges added in Steps 2 and 3 remain at their positions in the polygon.
6. Color the resulting $\frac{n}{2}$ edges with the second color (in the example we color the edges 6-2, 1-3, 5-4).
7. Repeat Steps 5 and 6 for the remaining colors $3, \dots, n-1$.

of parallel edges in $n - 1$ iterations (see also Fig. 27.4). It can be shown that this algorithm generates a feasible edge coloring for any even number n .

In order to implement the algorithm presented above using a computer, it would be relatively costly to store polygons and rotate them cyclically in each iteration. Using the division of integers with remainder (modulo operation, see Chap. 12), the algorithm can simply be written as follows:

**Fig. 27.4.** Iterations of the procedure

The algorithm COLOREDGES colors all edges of the complete graph with an even number n of vertices with $n - 1$ colors.

```

1  procedure COLOREDGES ( $n$ )
2  begin
3      for all colors  $i := 1$  to  $n - 1$  do
4          color edge  $[i, n]$  with color  $i$ 
5          for  $k := 1$  to  $\frac{n}{2} - 1$ 
6              color all edges  $[(i + k) \text{ mod}(n - 1), (i - k) \text{ mod}(n - 1)]$ 
7                  with color  $i$ 
8          endfor
9      endfor
10 end
```

In this algorithm $a \text{ mod } b$ denotes the remainder after dividing the integer a by the integer b . For example,

- $14 \text{ mod } 4 = 2$, since $14 = 3 \cdot 4 + 2$,
- $9 \text{ mod } 3 = 0$, since $9 = 3 \cdot 3 + 0$ and
- $-1 \text{ mod } 5 = 4$, since $-1 = (-1) \cdot 5 + 4$.

If we divide by the number $n - 1$ in Step 6, we get remainders from the set $0, 1, \dots, n - 2$. Since our vertices are numbered by $1, \dots, n - 1$ (and not by $0, 1, \dots, n - 2$), we identify the remainder 0 with $n - 1$.

In our example in Step 6 we get for $i = 1$ the edges

- $[(1 + 1) \text{ mod } 5, (1 - 1) \text{ mod } 5] = [2, 5]$ for $k = 1$, and
- $[(1 + 2) \text{ mod } 5, (1 - 2) \text{ mod } 5] = [3, 4]$ for $k = 2$.

The calculation of the values for $i = 2, 3, 4, 5$ may be done by the reader.

Schedules with Home–Away Assignments

Let us again consider a soccer league. In contrast to the table-tennis tournament, the matches are not performed at the same location, but in the stadiums of the teams. If in the first half series the match between teams i and j is scheduled in the stadium of team i , then in the second half series team j is the home team. Thus, in a schedule for such a league, in addition to the pairings (who plays whom) per round, for each pairing a home team has to be determined.

For various reasons (e.g., fairness, attractiveness for the spectators) for each team home and away matches should alternate as much as possible. If a team has two consecutive home or away matches, this is also called a break (the alternating sequence of H- and A-matches is broken). When breaks are undesirable, a schedule without any breaks would be the best. But, if we have a closer look at the schedules of different leagues in various sports disciplines we see that in every season breaks occur. We may ask the question whether breaks are unavoidable or whether better schedules (without any breaks) exist.

It is relatively easy to see that with respect to our constraints no schedule without any breaks exists. If no team has a break, each team must have a home–away sequence of the form HAHA...H or AHAH...A. On the other hand, two teams with the same home–away sequence (e.g., HAHA...H) can never play against each other (since always both teams play either home or away). For this reason at most two teams can have a home–away sequence without any break. Thus, the remaining $n - 2$ teams must have at least one break, i.e., each schedule for a half series contains at least $n - 2$ breaks.

It can be shown that schedules with exactly $n - 2$ breaks exist which can be generated with an extension of the algorithm described above. In this extension additionally the home team has to be determined in Steps 4 and 6 of algorithm COLOREDGES as follows:

- Any match $[i, n]$ is a home match for team i if i is even; otherwise it is a home match for team n .
- Any match $[(i + k) \bmod(n - 1), (i - k) \bmod(n - 1)]$ is a home match for team $(i + k) \bmod(n - 1)$ if k is odd; otherwise it is a home match for team $(i - k) \bmod(n - 1)$.

In our graph model, home and away matches can easily be integrated if the edges are oriented. If the directed arc $i \rightarrow j$ means that the match between teams i and j takes place at team j , then for our example with $n = 6$ teams, using the extended algorithm we obtain the schedule from Fig. 27.5 with $n - 2 = 4$ breaks (teams 1 and 6 have no break, teams 2, 3, 4, 5 each have one break).

The geometrical construction procedure based on the polygon can also be extended by giving each edge an orientation. While the edges in the polygon are always oriented in the same direction, the orientation of the edge to the outer vertex n alternates in each iteration (see Fig. 27.6).

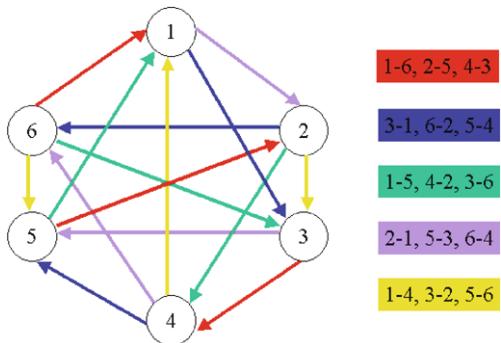
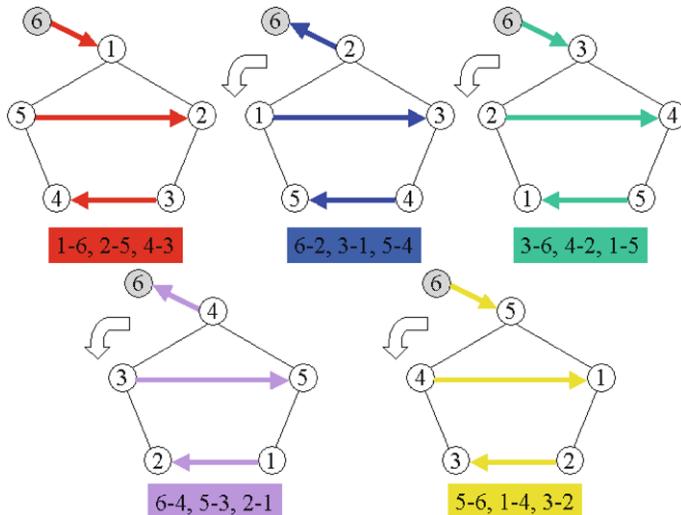
Fig. 27.5. Schedule with $n - 2$ breaks

Fig. 27.6. Iterations of the extended procedure

Summarizing, we can state that for any even number n of teams a schedule with $n - 1$ rounds and $n - 2$ breaks exists which can be constructed by the described algorithm.

Finally, let us come back to the soccer league. Since there we have two half series, in each half series at least $n - 2$ breaks occur. In the professional German soccer league the matches of the second half series are scheduled in the same sequence as in the first half series (with exchanged home rights). For such a system it can be shown that at least $3n - 6$ breaks occur. A schedule having $3n - 6$ breaks ($n - 2$ in both half series and $n - 2$ at the border between first and second half series) can easily be constructed with the above method. Using a slight modification, it can additionally be ensured that no team has two consecutive breaks.

Usually, scheduling of a sports league in practice is more difficult since additional constraints have to be respected. For example, if a league contains teams which share the same stadium, two such teams cannot play at home simultaneously. Thus, when one of these teams plays at home, the other must play away. Furthermore, due to limited train or police capacities, not too many matches should be played in a region at the same day. Additionally, it may happen that in some rounds a stadium is unavailable since another event (e.g., a concert or a fair) is already taking place. Then the corresponding team has to play away in such a round. Finally, the media and spectators expect a varied, eventful, and exciting season (e.g., top matches should be scheduled at the end of a season) and attractive matches should be distributed evenly over the season.

For now most sports leagues schedules are still constructed manually. A scheduler generates a generic schedule for the corresponding league size with the above algorithm where at first the numbers $1, \dots, n$ are used as placeholders for the teams. Afterwards, in a second step each specific team is assigned to a number (e.g., 1 = Werder Bremen, 2 = Hamburger SV, 3 = Bayern München, etc.). In this step as many additional constraints are met as possible (e.g., that two teams sharing the same stadium never play at home simultaneously).

Let us now consider how many different team assignments are possible for a league with $n = 18$ teams. For the first number we can choose among 18 teams, and for the second number we have 17 possible teams (since the first team is already chosen), then 16 possibilities for the third number, etc. Thus, in total we get $18! = 18 \cdot 17 \cdot 16 \cdots 2 \cdot 1 \approx 6.4 \cdot 10^{15}$ possibilities. If we assume that a computer can generate 10^9 solutions per second, we need 74 days in order to check all possibilities. This enormously huge number shows that a human scheduler even with the help of a computer can only check a very small number of possible schedules.

A further disadvantage of the described method is that only one specific schedule is used as the foundation for the assignment. There are several other schedules which cannot be generated with the above method. Thus, with this procedure it cannot be guaranteed that good schedules (i.e., schedules satisfying the given constraints as much as possible) are found. For this reason, current research in the area of sports scheduling deals with the development of new algorithms that try to calculate good schedules in a reasonable amount of time.

Further Reading

1. Chapter 28 (Eulerian Circuits), Chap. 32 (Shortest Paths), Chap. 33 (Minimum Spanning Trees), Chap. 34 (Maximum Flows) and Chap. 40 (The Travelling Salesman Problem).

Further models and algorithms based on graphs can be found in these chapters.

2. J.M. Aldous and R.J. Wilson: *Graphs and Applications*. Springer, 2003.
An introduction to graphs and their applications.
3. Eric W. Weisstein: *Kirkman's Schoolgirl Problem*. From MathWorld – A Wolfram Web Resource:
<http://mathworld.wolfram.com/KirkmansSchoolgirlProblem.html>
An additional combinatorial problem where solutions can be represented by graph colorings.
4. S. Knust: *Construction Methods for Sports League Schedules*
<http://www.informatik.uos.de/knust/sportssched/webapp/index.html>
A website where different construction methods for sports league schedules are shown (e.g., the construction methods discussed above).

Eulerian Circuits

Michael Behrisch, Amin Coja-Oghlan, and Peter Liske

Humboldt-Universität zu Berlin, Berlin, Germany
 University of Warwick, Coventry, UK
 Humboldt-Universität zu Berlin, Berlin, Germany

Teasing your mates with riddles can be quite an amusing pastime – provided that you know the answer already! The “House of Santa Claus” provides a nice little teaser:

This figure consists of five *nodes* (the blue dots) and eight *edges* (the lines that connect the nodes). *Can you draw the House of Santa Claus in one sweep, without lifting the pen and without drawing any edge twice?*

Of course, it won’t be long until all your friends know how to solve this one (as there are actually 44 different ways of drawing the House of Santa Claus). But fortunately there are plenty of other figures that can be drawn in one sweep as well, provided that you know how. In some other cases you may end up trying for quite a while, just to realize that drawing the figure in one go seems all but impossible.

In this chapter we present an *algorithm* that will *always* produce a way to draw a given figure in one go if this is possible. To devise such an algorithm, let us first try to deal with figures that can be drawn in one sweep such that

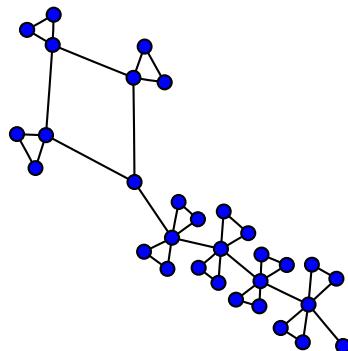
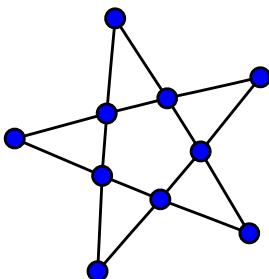
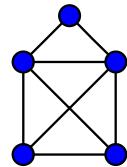


Fig. 28.1. Try the star and the dragon!

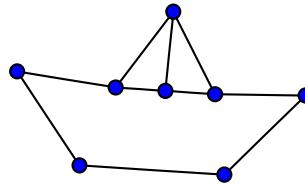


Fig. 28.2. The sailing boat provides a counterexample

the pen ends up at the same node where it started. The route that the pen traverses is then called an *Eulerian circuit*. This is because the mathematician *Leonhard Euler* was the first to find out what figures can be sketched in this way. (Actually Leonhard Euler was mostly interested in one particular figure, namely the roadmap of his home town of Königsberg.) Following Euler, we first deal with the question of whether an Eulerian circuit exists. Later on we sort out how to find an Eulerian circuit quickly if there is one.

In what follows we are going to figure out why, for example, the star has an Eulerian circuit, and why neither the House of Santa Claus nor the ship admit one. Bizarrely, the House of Santa Claus can be drawn in one go if we allow that the pen ends up at a different node than where it started, whereas even this is impossible in the case of the ship. To find out why, we need to take a closer look at the *nodes*, i.e., the places where the pen can change direction. (Recall that the nodes are just the blue dots in our figures.)

When Does an Eulerian Circuit Exist?

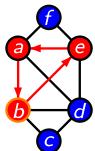
The *degree* of a node is the number of edges that pass through that node. For instance, the degree of the mast top of the ship is three. If we draw a figure in one sweep so that in the end the pen returns to the starting point, then the pen leaves every vertex exactly as many times as it enters that vertex. In effect, *the degree of each node is even*.

Observe that the ship has four nodes of odd degree (namely, all the nodes that belong to the sail). This shows that it is impossible to draw the ship in one go such that the starting point and the endpoint coincide. The same is true of the House of Santa Claus, because it has two nodes of degree three. However, there is a little twist (to be revealed later) that enables us to draw the figure in one sweep, but with different starting point and endpoint. By contrast, all nodes of the star have even degree. *But does any figure with this property feature an Eulerian circuit? And, if so, how do we actually find one?*

Finding Eulerian Circuits

Suppose that all nodes have even degrees. In the absence of a better idea, we could just *start somewhere and go ahead drawing*. That is, we start at an arbitrary node and follow an arbitrary edge to another node. Once we get there, we pick another edge that we haven't used before arbitrarily, and so on.

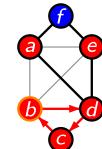
Each time we enter or leave a node, we use up two of its edges (because we are not allowed to use an edge more than once). Hence, if the node had an even degree initially, the number of available edges at that node will remain even. As a consequence, we won't get stuck in a dead end. In other words, our "just go ahead" strategy will eventually lead us back to the node where we started.



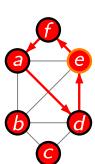
In the left figure a *circuit* (i.e., a route through several edges that leads back to the origin) consisting of three edges has emerged. The circuit passes through the vertices b , e , and a .

Yet following the above strategy ("start somewhere and keep on going until you get back to the origin") does not necessarily yield a circuit that covers *all* edges. Namely, we could have taken a "shortcut" at some node, thereby skipping a part of the figure. If this happens, we will need to *extend the circuit that we have constructed so far*. Before we do this, we remove the edges that we have visited already (because we are not allowed to pass through the same edges again anyway).

For instance, in the above figure upon returning to the origin b , we can repeat the same procedure to construct another circuit. In the above example the second circuit is a triangle through the nodes b , d , and c (right figure).



Linking the two circuits that we have obtained so far, we obtain a longer circuit (b, e, a, b, d, c, b) . Alas, even this circuit does not comprise all the edges. Hence, we are in for another extension. Of course, since all the edges that pass through the start vertex b are already used up, we need to pick another vertex to construct the next circuit.



Observe that removing all edges that our current circuit (b, e, a, b, d, c, b) passes through leaves us with a figure in which all nodes have even degrees. Hence, we can easily *find yet another circuit* as follows: Pick a node on the "old" circuit that has an edge that we haven't visited yet. Declare this node the new starting vertex and proceed to find another circuit by following the "just draw ahead" strategy. In the above example we get the quadrangle with the corners a , d , e , and f as shown in the left figure.

Thus, in addition to our "old" circuit we have got a new one that starts and ends at some node of the old circuit. Now, the plan is to hook the new

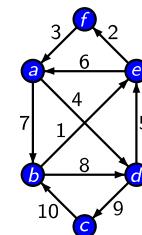
circuit into the old one. To achieve this, we first follow the old circuit until we reach the node where the new circuit starts. In the above example this is node e . We then proceed through the new circuit until we get back to its starting point (i.e., e). Finally, having completed the new circuit, we continue following the old circuit until the end. Hence, the complete route is $(b, e, f, a, d, e, a, b, d, c, b)$.

In summary, we have created a “big” circuit by combining two “small” ones. In our example the big circuit comprises the entire figure, as desired. But what do we do in other cases, where even the big circuit does not pass through all the edges?

Well, it’s easy: Why not just repeat the entire procedure? That is, we search for a node on the big circuit that has an edge that we have not passed through yet. Since the original figure was just one connected object, such a vertex exists so long as we have not passed through all the edges. After removing all edges of the big circuit from the figure, we start at the chosen node and construct a new circuit just as before. Then, we link the two circuits to obtain a bigger one.

We keep doing this until all the edges of the figure are used up. Once all edges are finished, we have an Eulerian circuit.

In the above example we first combined the two circuits (b, e, a, b) and (b, d, c, b) to obtain the circuit (b, e, a, b, d, c, b) . Then, starting anew from node e , we obtained (e, f, a, d, e) . Linking this to the previously obtained circuit (b, e, a, b, d, c, b) , we finally constructed the Eulerian circuit $(b, e, a, d, e, f, a, b, d, c, b)$, which is depicted in the right figure. The numbers indicate the order in which the tour traverses the edges.



The Algorithm

The algorithm below works similarly to the example above except for the linking step, which is performed in place. Hence, after finding a subcircuit (e.g. (b, e, a, b, d, c, b)), the next circuit will be inserted right away. For instance, assume the current circuit is (b, e, a, b, d, c, b) and the node chosen in line 4 is $u = a$. The edge selected could be (a, d) , which would extend the circuit preliminary to (b, e, a, d, b, d, c, b) . Obviously, this is not a valid circuit yet but the algorithm will continue until it reaches a again.

The algorithm EULERIANCIRCUIT calculates for a figure with even degree nodes the way to draw it in one sweep, and prints the order in which the edges have to be followed

```

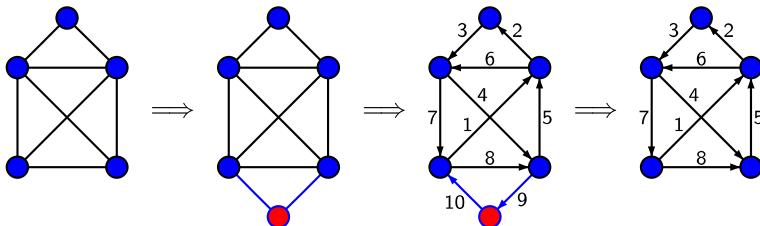
1  function EULERIANCIRCUIT(Figure F)
2  begin
3      Circuit := (s), for an arbitrary (start) node s in F
4      while there is a node u with an outgoing edge in the Circuit
5          v := u
6          repeat
7              take an edge v – w, starting in v
8              insert the other end node w into the Circuit after v
9              v := w
10             remove the edge from F
11         until v = u      // the circuit is closed
12     endwhile
13     return Circuit
14 end
```

The House of Santa Claus

Up to now, we have only cared about figures featuring an Eulerian circuit, which could thus be drawn in a single sweep with matching start and end point. We already know that this works only if all nodes of the figure have even degree. But this is not true for the House of Santa Claus: Both bottom nodes have degree 3. Nevertheless, it is possible to draw the figure in one sweep, if we do not insist on starting point and endpoint being identical.

How can we adapt the algorithm so that it works for the House of Santa Claus as well?

The simple trick is to insert a new node and connect it to both nodes of degree 3. The resulting figure has only nodes of even degree, and thus the algorithm will work and produce an Eulerian Circuit.



In the end we simply omit our “artificial” node from the Eulerian Circuit: We start drawing at the left “neighbor node” and finish at the right neighbor, thus getting a solution for the House of Santa Claus! This works particularly well in our example because the edges added are the last ones in the circuit.

If this is not the case we have to “shift” the circuit (this means, rotate the order of the edges such that 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 becomes 4, 5, 6, 7, 8, 9, 10, 1, 2, 3, for example), to move those edges to the end.

If we have exactly two nodes of odd degree in our figure, this trick will always work. Figures with more than two nodes of odd degree cannot be drawn in one sweep at all, even if you allow the start and end point to differ.

Of Postmen and Garbage Collectors

In addition to the fact that you can impress your mates with the algorithm described (especially by deciding at “first sight”, i.e., by testing the degrees of the nodes, whether the figure can be drawn in one sweep), there are also some serious applications. Assume the edges are streets and the nodes junctions. Then a path visiting each street exactly once is an Eulerian Circuit. Thus, if you have Eulerian Circuits in your road network they provide fast and fuel-saving routes for postal delivery and garbage collection. Our algorithm can calculate these routes, but we would, of course, implement it on a computer to process road networks with hundreds of streets.

Unfortunately, in reality there are often more than two junctions with an odd number of streets. That is why the calculation of the garbage collection routes has to take into account that some streets have to be used multiple times. When calculating the shortest route in this scenario the length of the streets comes into play as well. This leads to the so-called *Chinese Postman Problem*.

Thus, dealing with Eulerian Circuits does not only allow for a round trip through edges and nodes but also leads from one pleasant carrier (Santa Claus) to another (the postman).

Further Reading

1. http://en.wikipedia.org/wiki/Seven_Bridges_of_Koenigsberg
This Wikipedia article explains everything about the starting point of this problem as a “touristical” question addressed by Euler more than 270 years ago.
2. http://en.wikipedia.org/wiki/Leonhard_Euler
Life and work of the eponym not only of the Eulerian Circuit but also Eulerian Number and many other important mathematical achievements.
3. In contrast to an Eulerian Circuit, which visits all the edges of a given figure, Chap. 9 is about testing whether a figure contains a given vertex. The algorithm for this problem is quite similar to the one for finding Eulerian Circuits.

4. Chapter 40 (the Travelling Salesman Problem)

In this chapter we saw an algorithm for finding an Eulerian Circuit, i.e., a circuit that traverses every *edge* of a given figure exactly once. Surprisingly, the problem of finding a cycle that merely visits every *vertex* precisely once seems much more difficult. If in fact the goal is to find a *shortest* such cycle, we end up with a notoriously difficult problem known as the *Travelling Salesman Problem*.

5. Think beyond: We have seen that the algorithm works for figures with an even number of outgoing lines at each point, we know that we can easily repair two “odd nodes” and that it does not work with four. But what about one or three “odd nodes”?

High-Speed Circles

Dominik Sibbing and Leif Kobbelt

RWTH Aachen University, Aachen, Germany

If you look very closely at the screen of a computer, you will realize that a picture contains thousands of small colored dots, the so-called pixels (= picture elements). Generating an image with a computer means determining the color each pixel glows with. This is similar to painting a picture by coloring the squares of quad paper, as in Fig. 29.1. On current monitors the number of pixels is 10,000 times more than the number of squares on regular quad paper. Since in videos and computer games the images typically have to be redrawn at 30 frames per second, the use of very efficient algorithms becomes necessary. To draw a geometrically complicated scene, common methods in computer graphics break the scene down to a large number of simple elements like triangles, lines, and circles, which approximate this complex scene. As mentioned, these elements have to be drawn efficiently, i.e., by using as few and simple operations as possible. In this chapter we want to explain how to design such an efficient algorithm for drawing circles.

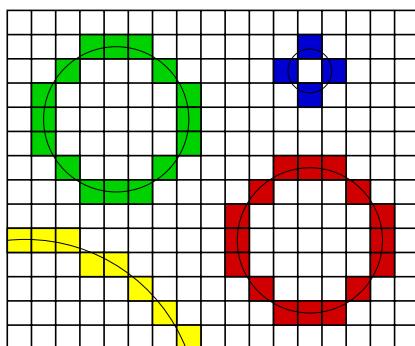


Fig. 29.1. Drawing with circles

Drawing Circles: Keep It Simple!

If you want to draw a circle, the first thing which might come to mind is to use dividers. The principle of such a device is to keep the distance of the pen from the center of the circle constant while turning the pen 360° . To mimic this method we have to calculate the position of the pen at every point in time. Knowing the radius R of the circle and the angle α between the x -axis and the line connecting the pen and circle center, we can determine the position of the pen by using the sine and cosine functions (Fig. 29.2):

$$(x, y) = (R \cdot \cos(\alpha), R \cdot \sin(\alpha)).$$

Assume we have a command “*plot* (x, y)” which activates the pixel with coordinates (x, y) . Together with the formula from above, we can activate a set of pixels which belong to the circle. Here is a first try at activating N pixels lying on a circle:

Naive Algorithm for drawing circles

```

1   for i := 0 to N - 1 do
2       x = R · cos(360 · i/N)
3       y = R · sin(360 · i/N)
4       plot(x, y)
5   endfor

```

As you might see, the expression within the brackets of the sine and cosine function goes from 0° to 360° , so the whole circle gets covered.

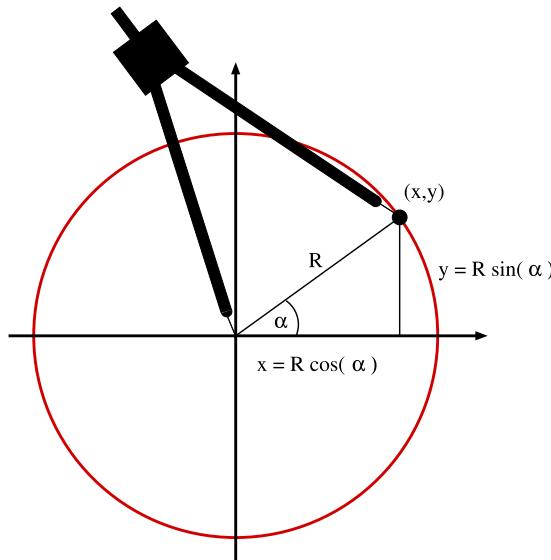


Fig. 29.2. Parametrization of a circle

But how do we choose the number N of pixels to be activated? On the one hand we want to calculate as few points as possible, because every calculation takes time. On the other hand, if we do not calculate a sufficient number of points, our circle might contain gaps. Assume the width of a pixel is equal to 1. By using the formula for the circumference,

$$U = 2\pi R \leq R,$$

one can estimate that $7R$ pixels are sufficient to draw a circle without any gaps.

Although this algorithm works well, many multiplications and summations and the evaluation of the complicated sine and cosine functions require high computation times, and so the drawing of thousands of circles will cause long waiting periods.

Circles are symmetric. Would it be possible to save some time and improve our algorithm?

If we look at a circle, we realize its symmetry. One can utilize this symmetry by, e.g., drawing only the upper half of the circle and get the lower part by simply reflecting each point w.r.t. the x -axis. Mathematically this reflection is very simple, since we only need to change the sign of the y -coordinate. If you look at Fig. 29.3 you see other lines of symmetry which are as simple as the x -axis. The reflection w.r.t. the y -axis just changes the sign of the x -coordinate. Taking both axes into account, it is only necessary to compute the points of one quarter of the circle. It is even possible to compute just one-eighth of the circle if we also consider both diagonals of the coordinate system. For both lines the reflection is also very simple, since we just need to exchange the x -

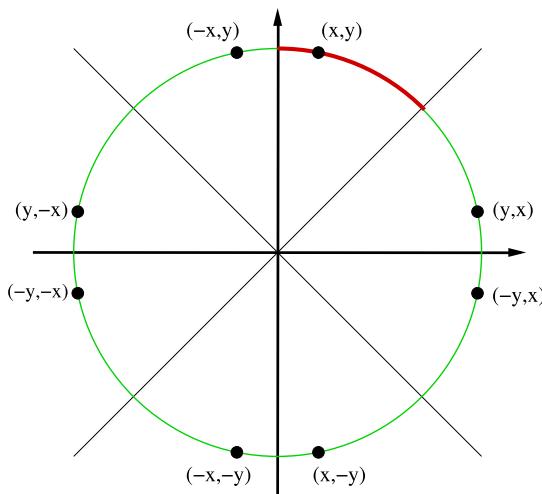


Fig. 29.3. Symmetry of a circle

and y -coordinates. So, by calculating a point (x, y) lying on the arc between 12 o'clock and 1:30, we directly derive the other seven symmetric points

(y, x) , $(-y, x)$, $(x, -y)$, $(-x, -y)$, $(-y, -x)$, $(y, -x)$, and $(-x, y)$

with nearly no additional computational costs (see Fig. 29.3).

Using this knowledge one can produce an algorithm that is nearly seven times faster than the “naive algorithm” from above:

Improved naive algorithm for drawing a circle

```

1    $N = 7R$ 
2   for  $i := 0$  to  $N/8$  do
3        $x = R \cdot \cos(360 \cdot i/N)$ 
4        $y = R \cdot \sin(360 \cdot i/N)$ 
5        $\text{plot}(x, y); \text{plot}(-y, x)$ 
6        $\text{plot}(-x, y); \text{plot}(y, -x)$ 
7        $\text{plot}(x, -y); \text{plot}(-y, -x)$ 
8        $\text{plot}(-x, -y); \text{plot}(y, x)$ 
9   endfor

```

Bresenham's Algorithm for Circles

In this section we look at the calculation of the points (x, y) themselves, which still needs too much time to evaluate. Especially the sine and cosine functions are quite complicated to compute. Using Pythagoras's theorem one can immediately calculate the y -coordinate dependent on the x -coordinate (see Fig. 29.2):

$$x^2 + y^2 = R^2, \quad \text{also } y = \sqrt{R^2 - x^2}.$$

The first advantage of an algorithm using Pythagoras's theorem is that it does not need to precompute the number N of pixels to be activated, since it just enumerates the x -coordinates. The second advantage is that it does not evaluate the sine and cosine functions.

Unfortunately the evaluation of the single square root is computationally as costly as the evaluation of a sine function. In addition to this, the result of a square root and a sine and a cosine function always comes with a very small error. So it would be best if these functions would completely vanish in our algorithm and if we could only use simple summations and multiplications.

The high speed algorithm is called “Bresenham's algorithm for circles.” It was invented by Jack E. Bresenham (1962) and was originally used to draw lines. An adaption of this algorithm to draw circles will be described in the following chapter.

The Idea. We start by drawing the topmost pixel with coordinates $(0, R)$ in whole numbers. In each step of the algorithm the x -coordinate is increased by 1. For the arc between 12 o'clock and 1:30, the slope of the curve is between

0 and -1 . So, the y -coordinate will not change if we go east and it will be decreased by one if we go southeast. The decision in which direction to go about is made so that the center of the new activated pixel (the blue dot of Fig. 29.4) lies close to the red circle. The algorithm stops when we have drawn one eighth of the circle, since we can complete the remaining parts by reflecting the points w.r.t. the lines of symmetry.

To decide whether the center of the pixel lying in the east or southeast direction is closer to the circle, we look at the green dots (Fig. 29.4). With respect to the current position these dots are always shifted by one unit of length to the right and a half unit of length downwards. Since we start at $(0, R)$, the first green dot to be tested lies at $(1, R - \frac{1}{2})$.

With regard to the green points, the decision to go east or southeast is made as follows:

Case 1: If the green dot is inside the circle, we go east:

$$(x, y) \leftarrow (x + 1, y).$$

Case 2: If the green dot is outside the circle, we go southeast:

$$(x, y) \leftarrow (x + 1, y - 1).$$

Decision: inside or outside the circle? To distinguish both cases we need a way to decide whether a point is inside or outside the circle.

A point (x, y) is inside the circle if the distance between the center of the circle and this point is smaller than the radius R of the circle, i.e., it is inside if the value of the function

$$F(x, y) = x^2 + y^2 - R^2$$

is smaller than 0. The good thing is that we do not have to compute a square root any more!

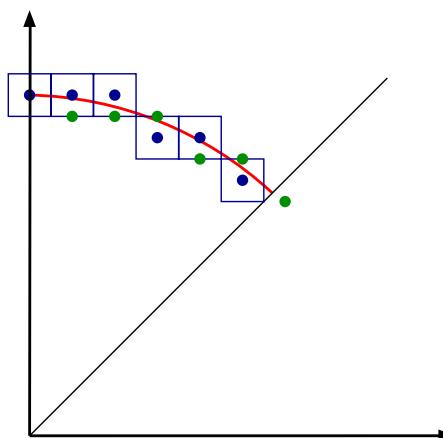


Fig. 29.4. Bresenham's Algorithm for circles

The value for the first green dot can easily been found:

$$F\left(1, R - \frac{1}{2}\right) = 1 + R^2 - R + \frac{1}{4} - R^2 = \frac{5}{4} - R.$$

If we decided to go from a blue point (x, y) in the east direction, the value of F , evaluated at the green point $(x_g, y_g) = (x + 1, y - \frac{1}{2})$, will change in the following way:

$$\begin{aligned} F(x_g + 1, y_g) &= (x_g + 1)^2 + y_g^2 - R^2 \\ &= x_g^2 + 2x_g + 1 + y_g^2 - R^2 \\ &= F(x_g, y_g) + 2x_g + 1. \end{aligned}$$

Going in southeast direction, we have to add 1 to the x -coordinate and subtract 1 from the y -coordinate of the green point, so the value of F changes as follows:

$$\begin{aligned} F(x_g + 1, y_g - 1) &= (x_g + 1)^2 + (y_g - 1)^2 - R^2 \\ &= F(x_g, y_g) + 2x_g - 2y_g + 2. \end{aligned}$$

After these steps we activate the pixel and start the process for the next pixel. As described, we do not recompute F in every step, but we just update it by adding small values, depending on the direction we went in. This is called an “incremental” computation of F , which can be done much faster than recomputing the whole expression.

Having this in mind we can write down a first version of Bresenham’s Algorithm for circles.

Bresenham’s Algorithm for circles

```

1    $(x, y) = (0, R)$ 
2    $F = \frac{5}{4} - R$ 
3    $\text{plot}(0, R); \text{plot}(R, 0)$ 
4    $\text{plot}(0, -R); \text{plot}(-R, 0)$ 
5   while  $(x < y)$  do
6     if  $(F < 0)$  then
7        $F = F + 2 \cdot (x + 1) + 1$ 
8        $x = x + 1$ 
9     else
10     $F = F + 2 \cdot (x + 1) - 2 \cdot (y - \frac{1}{2}) + 2$ 
11     $x = x + 1$ 
12     $y = y - 1$ 
13  endif
14   $\text{plot}(x, y); \text{plot}(y, x)$ 
15   $\text{plot}(-x, y); \text{plot}(y, -x)$ 
16   $\text{plot}(x, -y); \text{plot}(-y, x)$ 
17   $\text{plot}(-x, -y); \text{plot}(-y, -x)$ 
18 endwhile
```

But we can do faster. For both increments $(2x_g + 1)$ and $(2x_g - 2y_g + 2)$ we need multiplications. Our algorithm would be faster if we could limit all calculations to simple additions.

To achieve this we need two more variables, which we call d_E and d_{SE} , so we can track the change of the increments. The idea is that we increase the function F by d_E if we go in the east direction, and by d_{SE} if we go in the southeast direction, which would only require one summation. In addition to this we also need to change d_E and d_{SE} depending on the direction we went in. For both variables we need the initial values, which can be found by plugging in the initial values of x_g and y_g :

$$d_E\left(1, R - \frac{1}{2}\right) = 2 \cdot 1 + 1 = 3,$$

$$d_{SE}\left(1, R - \frac{1}{2}\right) = 2 \cdot 1 - 2 \cdot R + 1 + 3 = 5 - 2 \cdot R.$$

Now we have to think about how d_E and d_{SE} will change if we decide upon one direction. This can be done in a way very similar to changing the function F itself. Assuming we go in the east direction, both values change in the following way:

$$d_E(x_g + 1, y_g) = 2 \cdot (x_g + 1) + 1 = d_E(x_g, y_g) + 2,$$

$$d_{SE}(x_g + 1, y_g) = 2 \cdot (x_g + 1) - 2 \cdot y_g + 2 = d_{SE}(x_g, y_g) + 2.$$

Going in the southeast direction will affect the values for d_E and d_{SE} as follows:

$$d_E(x_g + 1, y_g - 1) = 2 \cdot (x_g + 1) + 1 = d_E(x_g, y_g) + 2,$$

$$d_{SE}(x_g + 1, y_g - 1) = 2 \cdot (x_g + 1) - 2 \cdot (y_g - 1) + 2 = d_{SE}(x_g, y_g) + 4.$$

The fraction is gratuitous. All the increments are whole numbers, but since the initial value for F contains a fraction, we carry around this fraction during the entire process. Since whole numbers can be represented with more accuracy, it would be nice to just deal with integers.

To figure out if we can omit the fraction in some way, we consider what $F < 0$ means. Starting with

$$F = \frac{5}{4} - R$$

and assuming K to be a whole number, we see that F equals $K + 1/4$ in every step of the computation, since F is only increased by a whole number. So, in every step F is one of these numbers

$$F \in \left\{ \dots, -\frac{3}{4}, \frac{1}{4}, \frac{5}{4}, \dots \right\}.$$

This means, if F drops below zero, then also $F - 1/4$ drops below zero. So, instead of starting with $F = 5/4 - R$, we are also allowed to start with $F = 1 - R$, without affecting the correctness of the algorithm.

The final version of Bresenham's Algorithm for drawing circles uses only simple summations over whole numbers:

Improved Bresenham's Algorithm for drawing circles

```

1    $(x, y) = (0, R)$ 
2    $F = 1 - R$ 
3    $d_E = 3$ 
4    $d_{SE} = 5 - 2 \cdot R$ 
5    $\text{plot}(0, R); \text{plot}(-R, 0)$ 
6    $\text{plot}(0, -R); \text{plot}(-R, 0)$ 
7   while  $(x < y)$  do
8       if  $(F < 0)$  then
9            $F = F + d_E$ 
10       $x = x + 1$ 
11       $d_E = d_E + 2$ 
12       $d_{SE} = d_{SE} + 2$ 
13  else
14       $F = F + d_{SE}$ 
15       $x = x + 1$ 
16       $y = y - 1$ 
17       $d_E = d_E + 2$ 
18       $d_{SE} = d_{SE} + 4$ 
19  endif
20   $\text{plot}(x, y); \text{plot}(y, x)$ 
21   $\text{plot}(-x, y); \text{plot}(y, -x)$ 
22   $\text{plot}(x, -y); \text{plot}(-y, x)$ 
23   $\text{plot}(-x, -y); \text{plot}(-y, -x)$ 
24 endwhile
```

A Racing Duel

Drawing circles with this algorithm works very well: By letting all presented algorithms compete against each other, one can figure out that the last algorithm is able to draw circles 14 times faster than the one that was presented first! In addition to that, it just needs summations on whole numbers, which is useful if we deal with specialized processors. You should try it at home!

There are similar algorithms for other geometric primitives, such as lines and triangles. These algorithms are integrated as an essential part of current graphics cards, so displaying detailed pictures at high frame rates becomes possible for applications such as computer games.

The presented process of finding an algorithmic solution for a problem is typical in computer science. First of all it is necessary to find a precise mathematical description of the problem. This makes the implementation of a simple algorithm for the specific task possible; it might still have some

problems (such as gaps in the circle), but it helps us understand the problem better. After that one can try to find a solution, which can be calculated much faster. Therefore, we can take additional knowledge into account (symmetry of a circle) or reformulate some equations to simplify some calculations (incremental computation). In the end we can consider architectural properties of our computer to further accelerate the computations (e.g., summations can be calculated faster than multiplications). This process might not only lead to the optimal solution of our specific problem, it might also give solutions for similar problems. With some minor changes, the high speed algorithm for drawing circles can also be used to draw ellipses, parabolas, hyperbolas, or similar curves used in computer graphics.

Further Reading

1. Chapter 8 (Pledge's Algorithm) and Chap. 36 (The Smallest Enclosing Circle)
Results of the presented methods can be illustrated in a graphical sense. This requires fast rendering algorithms like Bresenham's Algorithm for lines and circles, which are capable of drawing geometric primitives.
2. Chapter 11 (Multiplication of Long Integers)
Bresenham's algorithm for drawing circles does not need any multiplications. If, however, multiplications become necessary, it should be possible to calculate them in a fast way. How this works is described in this chapter.
3. Alan Watt: *3D Computer Graphics*. Addison-Wesley, 3rd edition, 1999
The book describes basic algorithms from the field of computer graphics. It contains many examples.
4. Mason Woo, Jackie Neider, Tom Davis, Dave Shreiner: *The OpenGL Programming Guide*. Addison-Wesley Professional, 5th edition, 2005.
The book explains techniques far beyond the drawing of simple primitives and gives an introduction to OpenGL programming. Since it comes with many examples it motivates the reader to try out different techniques.

Gauß–Seidel Iterative Method for the Computation of Physical Problems

Christoph Freundl and Ulrich Rüde

Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, Germany

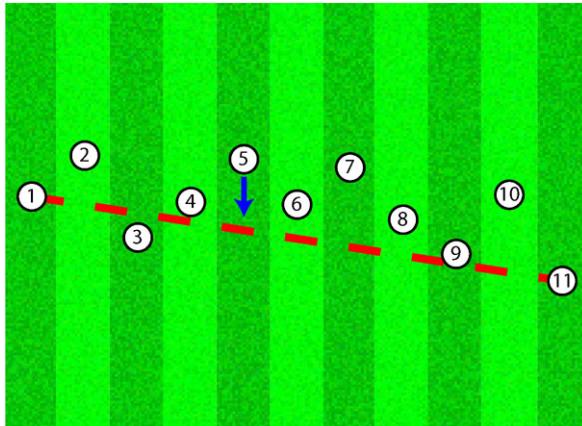
Warmup: Soccer

The following algorithm deals with the simulation of physical effects. Computers can also be utilized to simulate processes in physics, chemistry, and anywhere in nature. This is becoming more and more important because it helps to understand how nature works. For example, our weather forecast is based on a simulation that attempts to represent the natural weather as accurately as possible on a computer. New car models and planes are also simulated, long before they are built for the first time. Many scientists actually completely depend on simulations. Astronomers who want to understand what happens if two black holes collide have to use computer simulations, since experiments with real black holes are impossible. Computer games are also often similar to simulations, only for games it is not necessarily a goal that the computations coincide with the real world.

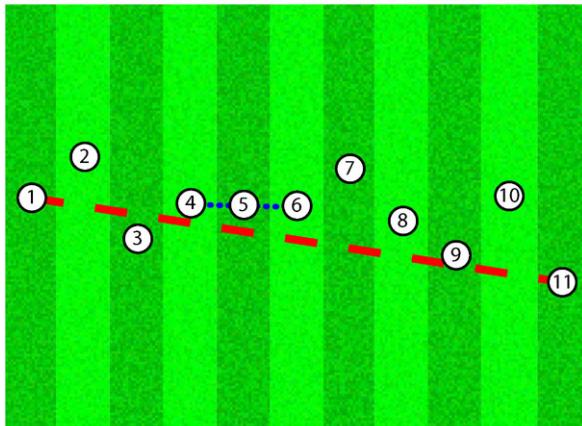
Here, we are going to simulate an important problem with our algorithm, namely that of heat distribution, as it would also be important for weather forecasts. However, we will restrict ourselves to solid bodies like a two-dimensional plate because it makes the simulation easier to explain if we do not have to take the air flow into account. But let us first start with a “warmup exercise” and have a look at soccer games.

After having barely missed the final round of the last World Cup, the national team finally reaches the final game four years later. The players are very nervous, so nervous that the coach is convinced that they will not even be able to form a line, arranged by shirt numbers, for the national anthem.

As the coach is not allowed to enter the field where he could easily put every player into his place in the line, he desperately comes up with the following method: he impresses upon the two players with the numbers 1 and 11, which are the left- and rightmost players in the line, to take care that there is enough space between them for all other players. After that, they shall not leave their places anymore.



(a) It is player 5's turn



(b) Player 5 is at its new position

Fig. 30.1. Situation during the line-up: player 5 is being called by the coach. He moves to the exact middle between his neighbors, the players 4 and 6. The red dashed line is given by the two players at the boundaries, in the end all players should stand on it ideally

The other players with the numbers from 2 to 10 receive the following instruction: if they are called they shall move to the exact middle between their right and left neighbors (see Fig. 30.1). The coach calls all players consecutively in the order of their shirt numbers. After the player with the highest number has moved, the coach starts again from the beginning.

You can try this out yourself quickly, for example with game pawns. The website for this algorithm contains a program that visualizes this approach. You will see: after several cycles of this method the players have indeed moved

such that they form at least an approximate line. It is not completely perfect but good enough such that nobody notices it.

In order to get the players exactly onto the line, the algorithm would have to run for an infinitely long time. This is why you will never get the completely right result in practice. But that does not matter because after sufficiently many steps the algorithm's outcome can get arbitrarily close to the correct solution. You encounter algorithms like this often when it comes to so-called numerical problems, i.e., when real decimal numbers have to be computed, such as those needed by physicists or engineers.

If the players line up in the order of their back numbers, they will be called from left to right over and over again but we can also do it differently. A popular variant is the so-called *red-black ordering* of the players. There the half of the players with the lower numbers occupy every other place in the line first, then the players with the high numbers fill up the remaining places. But even if the players are arranged completely at random, the method still works, only it must be obvious to every player who both his neighbors are.

This method is the algorithm of Gauß and Seidel, which we are going to apply for physical problems in the following.

Temperature Calculation in a Rod (1D)

Now, let us really consider the calculation of a temperature distribution. Is it not somewhat amazing that we can use the principle of lining up in a row? For example, if you look at the temperature distribution in a thin rod you will notice that the temperature at every point along the rod is the average of the temperatures in the neighborhood of that point. If you fix the temperature at both ends of the rod, the temperature between both ends runs “in a row”, i.e., linearly from one end to the other.

In order to compute this linear distribution of values, you need not use a computer, just as a soccer coach needed no complicated algorithm in order to place players in a row. But we can now see that the problems are related, and maybe they can be solved in the same way. Observe that the position of a player corresponds to a temperature value, otherwise the computation can proceed in the same way as in our soccer problem.

Next, we make the task a bit more interesting: how does the temperature distribution look if we heat the rod at some point in the middle? Then the temperature at that point is of course no longer the average of the neighboring temperatures, the additional heating has to be taken into account.

Now there is no longer an obvious possibility to determine the resulting temperature distribution so we ponder how to utilize the computer for solving this problem. A first difficulty is posed by the fact that there are infinitely many points along the rod, however a computer can only treat finitely many objects in finite time. Therefore, we choose a finite number of points along the rod (see Fig. 30.2) at which we want to compute the temperature. This

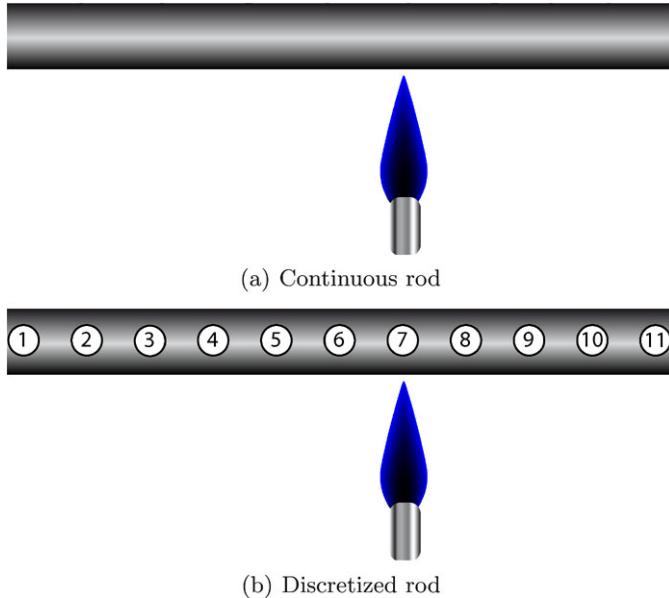


Fig. 30.2. Discretization of a continuous rod with 11 points

method is also called *discretization* as we are mapping a continuous problem to a discrete problem.

If the points are distributed evenly along the rod, and if u_i denotes the temperature value and f_i the heating at point i , then the temperature at a particular point is updated by the formula

$$u_i := \frac{1}{2}(u_{i-1} + u_{i+1}) + f_i.$$

The algorithm GAUSSSEIDEL1D

```

1  procedure GAUSSSEIDEL1D ( $n, u, f$ )
2  begin
3      for  $i := 2$  to  $n - 1$  do
4           $u[i] := \frac{1}{2}(u[i - 1] + u[i + 1]) + f[i]$ 
5      endfor
6  end
```

Like in the example of the soccer players only the points in the interior of the rod are continuously recalculated, we assume the temperatures at both boundary points of the rod are given. Depending on the physical experiment it could also be the case that the temperatures at the rod's boundaries are

not fixed. We also have not considered that in practice the rod would lose some heat to its neighborhood. We could take that into account by using correspondingly more complicated formulas and computations, but that would lead us too far for now. Instead, we will investigate another difficulty which arrives if we do not want to compute the temperature distribution in a one-dimensional rod, but in a two-dimensional plate.

Temperature Computation on a Plate (2D)

We can generalize the formulation of the task by leaving the one-dimensional rod behind and will now consider a two-dimensional cooking plate, which might again be heated at some places.

The discretization works similarly to the above case, only now we get a two-dimensional grid of points. We want to compute the temperature at these points. Averaging the neighboring temperatures at particular points means now to take not only the right and left neighbors into account but also the upper and lower neighbors (see Fig. 30.3).

Because of this representation of the dependencies of a point with respect to its neighboring points one identifies the *stencil* which has to be applied to the point and its surrounding in order to recompute its value. In this case we have a five-point stencil (because it involves five neighboring points), and the

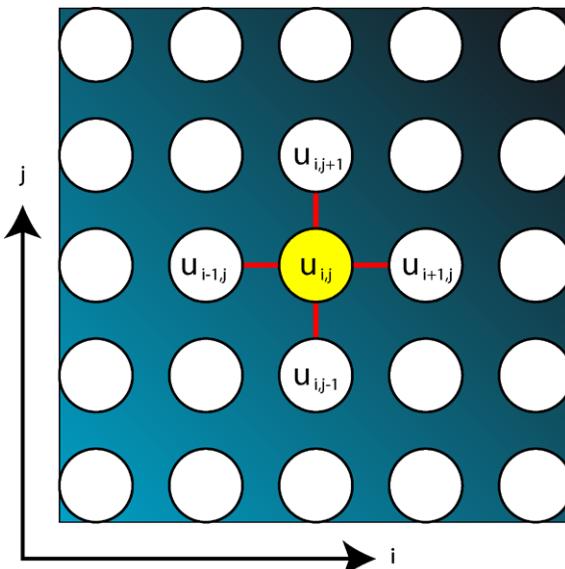


Fig. 30.3. Schematic view of a discretized two-dimensional plate and the five-point stencil

new value of a point in the interior of the plate is computed by

$$u_{i,j} := \frac{1}{4}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1}) + f_{i,j}.$$

The algorithm GAUSSSEIDEL2D

```

1  procedure GAUSSSEIDEL2D ( $n, u, f$ )
2  begin
3      for  $i := 2$  to  $n - 1$  do
4          for  $j := 2$  to  $n - 1$  do
5               $u[i,j] := \frac{1}{4}(u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i,j+1])$ 
6                   $+ f[i,j]$ 
7          endfor
8      endfor
9  end
```

Let us consider a quadratic plate whose left lower corner is located at the point $(0, 0)$ and whose right upper corner is located at the point $(1, 1)$. The temperature values at the boundary points of the plate are fixed, the temperature distribution at the right boundary is a curve described by the function $\sin(\pi y)$, and the temperature at all other boundaries is zero. We plot the temperature as a value in the third dimension depending on the position on the plate. In this three-dimensional picture we see therefore a landscape whose height corresponds to the temperature at that point. The first picture (Fig. 30.4(a)) shows a temperature distribution on a grid consisting of 33×33 points, where the temperature is zero at all points except on the right boundary. There we have the mentioned temperature curve.

This is not the correct temperature distribution which has to be computed first by our Gauß–Seidel method. It has to arrive at a smooth temperature distribution which is no longer linear as in the one-dimensional case even if there are no additional heat sources, which is assumed in this case.

If we execute algorithm GAUSSSEIDEL2D once, we call this one executed *iteration*. This already implies that we have to perform the algorithms multiple times to get a good solution. If we trace several Gauß–Seidel iterations (Fig. 30.4(b) to Fig. 30.4(f)) we see how the preset temperature at the boundary spreads into the interior of the plate until the temperature distribution over the whole plate finally looks nicely smooth. The amount of computation is not to be underestimated as every iteration of the Gauß–Seidel method has to compute the average of four numbers at $31 \times 31 = 961$ points. For a thousand iterations the computer has to perform already nearly 5 million arithmetic operations.

Even though the computed solution looks fairly good after 100 iterations (Fig. 30.4(e)), we must not stop the method at this time as we can see in Fig. 30.5. The first picture (Fig. 30.5(a)) shows the exact solution for this problem and the computed solution after 100 Gauß–Seidel iterations together such that we can see that the computed result is not quite right yet. (You

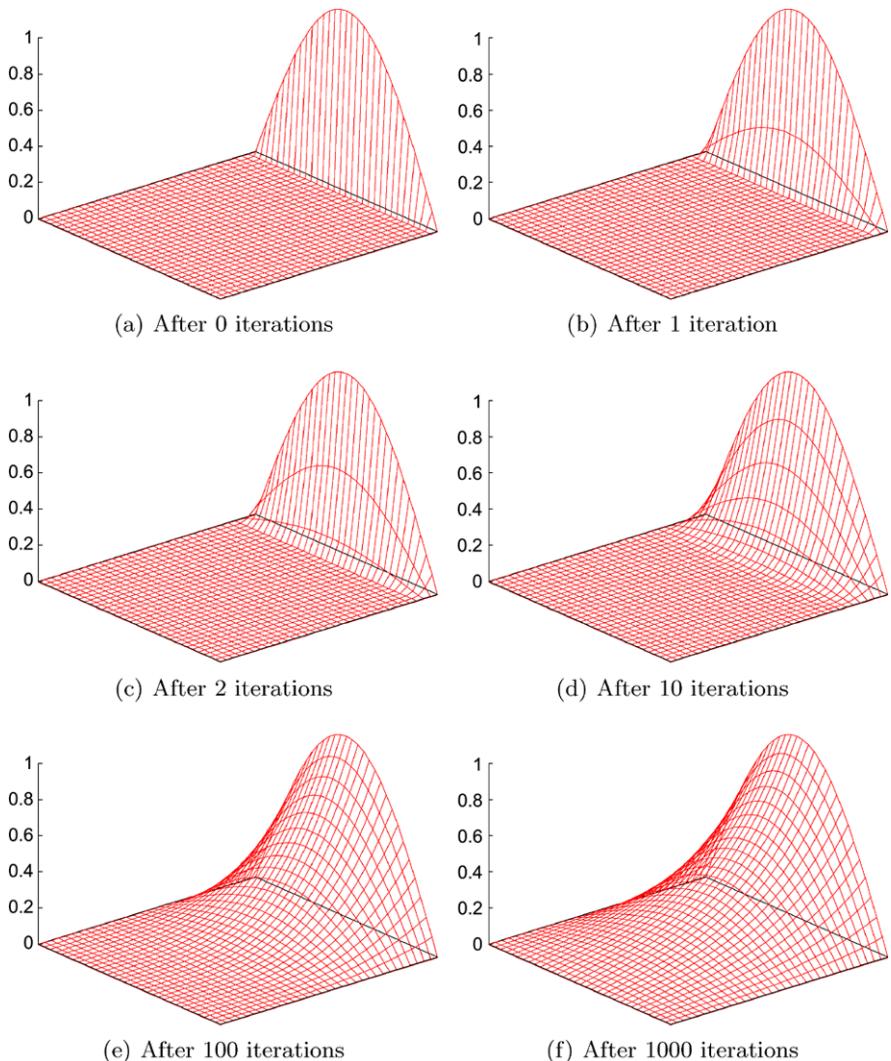


Fig. 30.4. Progress of Gauß–Seidel iterations

can determine the exact solution in this special case using a mathematical formula, it is $u(x, y) = \frac{1}{\sinh \pi} \sinh(\pi x) \cdot \sin(\pi y)$ which is the only function that fulfills the given boundary conditions and for which the sum of the derivatives by x and y equals zero.)

Only after about 1000 iterations (Fig. 30.4(f) and Fig. 30.5(b)) does the overlay show no difference between the exact and the computer solutions anymore.

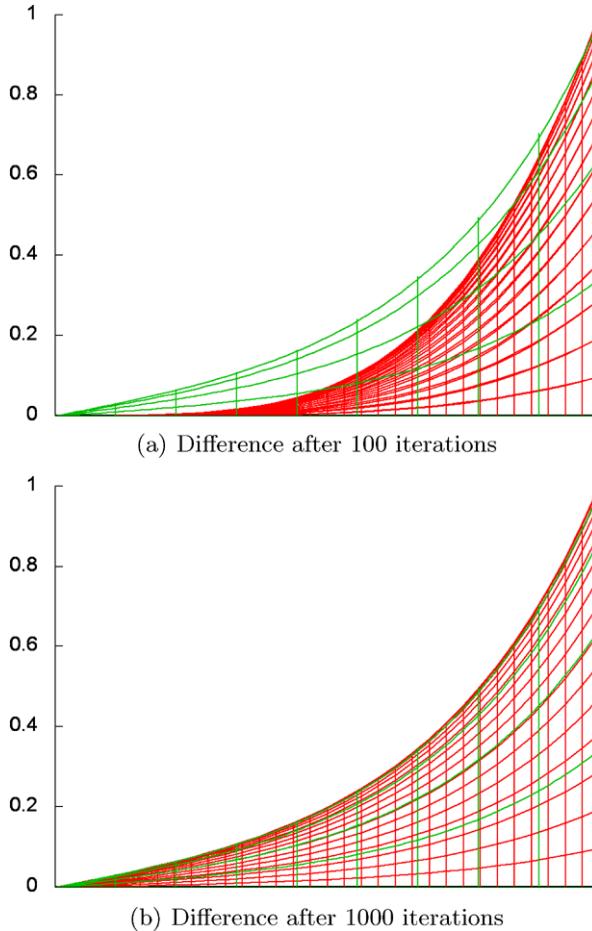


Fig. 30.5. Differences between approximated (red) and exact (green) solution

Even if it might appear that the temperature distribution shows the temporal advance of the heating during the iterations, this is not the case here. What we compute here is the state when the system is at an equilibrium with the given heating. Other, slightly more complicated methods would allow us to compute the correct temporal changes of heating or cooling.

Of course it is an interesting question how many iterations of the Gauß–Seidel method are actually needed in order to get a good approximation to the physical solution. By experience (or better by a mathematical analysis of the method) we can tell that for a grid consisting of N points we have to perform circa $N \times N$ iterations. On the other hand, as the temperature on the plate can be represented more exactly the finer the grid points are located, we quickly reach a computational effort that exceeds even the power of modern PCs. This

is especially true if we want to simulate not only two-dimensional (like the plate) but three-dimensional objects and if additional physical phenomena – like in weather forecasts – make the computations more complicated. Then we need not only supercomputers which are a lot more expensive (costing millions of Euros) but also significantly improved algorithms that can compute the same result in a shorter time.

For those who are interested here are two ideas for how the method could be accelerated: in the pictures it can be seen that the exact solution is approximated from one side, namely from below. In other words, every Gauß–Seidel iteration brings us closer to the exact solution but it does not go far enough. We can utilize this observation by increasing the computed change in every single iteration, i.e., by multiplying with a number larger than one (but smaller than two). The resulting method is the so-called SOR method (from “successive over-relaxation”). The second idea is even more complex and is based on several grids with different point intervals working together in a skillful way. This so-called multigrid method needs only a very small number of iterations before it computes good solutions and is regarded as the fastest known method for these types of problems.

Finally we want to mention that the Gauß–Seidel method was invented by the most famous of all mathematicians, Carl Friedrich Gauß, in 1823, and was further developed by one of his colleagues, Philipp Ludwig Seidel. At the time of Gauß and Seidel you had to perform the calculations manually of course. Gauß wrote in a letter: “The method can be performed half blindfold or one can think of other things when performing it.” If we program the method nowadays on a computer, we can also think of other things while the computers do the computational work.

Further Reading

1. Chapter 10 (PageRank)

The article about the page-rank algorithm shows how to solve a system of equations step by step: this is exactly the Gauß–Seidel method, only for a different equation system.

2. The Gauß–Seidel method as Algorithm of the Week:

<http://www-i1.informatik.rwth-aachen.de/~algorithmus/algo39.php>

In the online version of this article (German only) are Java applets which illustrate the execution of the method.

3. http://en.wikipedia.org/wiki/Gauss%20%93Seidel_method

This Wikipedia article contains an exact mathematical description of the Gauß–Seidel method for arbitrary equation systems.

4. “Why Multigrid Methods Are so Efficient” by Irad Yavneh:

<http://doi.ieee.org/10.1109/MCSE.2006.125>

This article, appearing in the journal *Computing in Science & Engineering*, explains the idea of the multigrid methods mentioned above, but it also requires more mathematics for understanding.

5. TOP500 – the list of the 500 fastest computers in the world:

<http://www.top500.org/>

This list is presented twice a year. It comprises those computers which can be used for computing solutions to really big problems.

6. http://en.wikipedia.org/wiki/Carl_Friedrich_Gauss

http://en.wikipedia.org/wiki/Philipp_Ludwig_von_Seidel

These articles describe in short the lives of the German mathematicians Carl Friedrich Gauß and Philipp Ludwig von Seidel to whom the presented method can be traced back.

Dynamic Programming – Evolutionary Distance

Norbert Blum and Matthias Kretschmer

Rheinische Friedrich-Wilhelms-Universität Bonn, Bonn, Germany

150 years ago parts of the skeleton of the so-called Neanderthal man were found near Düsseldorf (Germany) for the first time. Since then we've wanted to know how *Homo sapiens*, our ancestor, and the Neanderthal man are related. Today we know that *Homo neanderthalensis* is not an ancestor of *Homo sapiens*, and vice versa. Differences in the genotype of *Homo sapiens* and *Homo neanderthalensis* proved that. New technologies allow us to extract genotypes from bones that are more than 30,000 years old. The genotypes are extracted in the form of DNA sequences. These DNA sequences are like construction plans of animals and humans. In the course of time, DNA sequences change through mutations. Given the DNA sequences of different species, one can calculate their similarity with the help of a computer. We measure the similarity of two sequences by specifying a distance between them. A small distance of two DNA sequences indicates a high similarity of both sequences. How do we calculate the distance of two DNA sequences? We will show how to develop an algorithm for solving this problem. To do this, we need a mathematical model of DNA sequences, mutations and the distance between two DNA sequences.

Mathematical Modeling

A DNA sequence consists of bases. A sequence of three bases encodes an amino acid. There exist four different bases which are represented by the letters *A*, *G*, *C* and *T*. Hence, a DNA sequence may be represented by a string over the alphabet $\Sigma = \{A, G, C, T\}$. For example,

CAGCGGAAGGTCACGGCCGGGCCTAGCGCCTCAGGGGTG

is a part of the DNA sequence of the chicken.

In nature DNA sequences change through mutations. A *mutation* can be considered as a mapping from a DNA sequence *x* to a DNA sequence *y*. We assume that all mutations are modeled using the following three types of basic mutations:

1. deletion of a character,
2. insertion of a character, and
3. substitution of a character with another character.

For example, let $x = AGCT$ be a DNA sequence. Then the mutation substitute G by C would mutate x to the sequence $y = ACCT$. We use the notation $a \rightarrow b$ to represent the substitution of a by b . \rightarrow represents the deletion of character a and $\rightarrow b$ is the insertion of character b . The position where the mutation has to be performed is explicitly given by the algorithm.

To measure the distance of two DNA sequences, we give every basic mutation a specific cost. The mutation s has cost $c(s)$. The cost of a mutation corresponds to the probability of that mutation. The more likely a mutation is the lower the cost. We use the following costs for the three basic mutations:

- deletion: 2
- insertion: 2
- substitution: 3

To compare two DNA sequences x and y , we require in most cases more than one basic mutation to transform x to y . For example, if we want to compare $x = AG$ and $y = T$, then one basic mutation would not be sufficient. But we could perform the transformation using the sequence of mutations $S = A \rightarrow, G \rightarrow T$ (delete A and substitute G by T). The cost $c(S)$ of a sequence of mutations $S = s_1, \dots, s_t$ is the sum of the costs of its basic mutations, i.e.,

$$c(S) := c(s_1) + \dots + c(s_t).$$

The distance of two DNA sequences is defined by the cost of a specific sequence of mutations which transform one to the other. The problem is that there might be many different sequences of mutations that transform one DNA sequence to another. For example, we could use the following mutation sequences to transform the DNA sequence $x = AG$ to $y = T$:

- $S_1 = A \rightarrow, G \rightarrow T; c(S_1) = c(A \rightarrow) + c(G \rightarrow T) = 2 + 3 = 5$
- $S_2 = A \rightarrow T, G \rightarrow; c(S_2) = c(A \rightarrow T) + c(G \rightarrow) = 3 + 2 = 5$
- $S_3 = A \rightarrow, G \rightarrow, \rightarrow T; c(S_3) = c(A \rightarrow) + c(G \rightarrow) + c(\rightarrow T) = 2 + 2 + 2 = 6$
- $S_4 = A \rightarrow C, G \rightarrow, C \rightarrow, \rightarrow T;$
 $c(S_4) = c(A \rightarrow C) + c(G \rightarrow) + c(C \rightarrow) + c(\rightarrow T) = 3 + 2 + 2 + 2 = 9$

There exist many other sequences that transform x to y , but none of them have lower cost than the sequences S_1 and S_2 . To define the distance of two DNA sequences, we use the sequence of mutations with the lowest cost. Given a cost function c , the distance $d_c(x, y)$ of the DNA sequences x and y is defined by

$$d_c(x, y) := \min\{c(S) \mid S \text{ transforms } x \text{ to } y\}.$$

For example, the sequences S_1 and S_2 are the sequences of mutations with the lowest cost that transform $x = AG$ to $y = T$. Hence, the evolutionary distance $d_c(x, y)$ of x and y is five.

Calculation of the Evolutionary Distance

How to calculate the evolutionary distance $d_c(x, y)$? We only allow the basic mutations deletion, insertion and substitution to transform x to y . The definition of the basic mutations and of their costs are in such a way that multiple mutations at the same position can be replaced by a single mutation with lower cost. For example, the deletion of the character A and the insertion of the character B can be replaced by the substitution of A by B which has lower cost (cost 3 for the substitution and $2 + 2 = 4$ for the deletion and insertion). Operations at different positions do not depend upon each other. Hence, we can perform the mutations in any order. Assume that the last mutation will always be performed at the last position of both DNA sequences. Let $x = a_1a_2 \dots a_m$ and $y = b_1b_2 \dots b_n$ two DNA sequences; i.e., x consists of m and y of n characters. By the definition of our three mutations, we have the following three possibilities for the last mutation:

1. *Deletion*: Transform $a_1a_2 \dots a_{m-1}$ to $b_1b_2 \dots b_n$ and then delete a_m .
2. *Insertion*: Transform $a_1a_2 \dots a_m$ to $b_1b_2 \dots b_{n-1}$ and then insert b_n .
3. *Substitution*: Transform $a_1a_2 \dots a_{m-1}$ to $b_1b_2 \dots b_{n-1}$ and then substitute a_m by b_n .

For the calculation of the evolutionary distance, we only need to consider the last mutation with the lowest cost.

We develop our algorithm by using the scheme above. Let $x[i]$ be the sequence consisting of the first i characters of x . This sequence is called the *prefix of length i* of x . The prefix of length 0 of x is the empty sequence $x[0]$. The sequence of length m of x is x itself (x consists of m characters). Analogously, $y[j]$ denotes the prefix of length j of y . Now we can reformulate the three possible last mutations:

1. Transform $x[m - 1]$ to y and then delete a_m .
2. Transform x to $y[n - 1]$ and then insert b_n .
3. Transform $x[m - 1]$ to $y[n - 1]$ and then substitute a_m by b_n .

It remains to solve the following problem: How to transform $x[m - 1]$ to y , x to $y[n - 1]$ and $x[m - 1]$ to $y[n - 1]$? For these three transformations, we can just apply the same scheme. To calculate the evolutionary distance $d_c(x[i], y[j])$ of the prefix of length i of x and the prefix of length j of y , we use the following scheme:

$$d_c(x[i], y[j]) := \min \begin{cases} d_c(x[i - 1], y[j]) + c(a_i \rightarrow) & \text{(deletion),} \\ d_c(x[i], y[j - 1]) + c(\rightarrow b_j) & \text{(insertion),} \\ d_c(x[i - 1], y[j - 1]) + c(a_i \rightarrow b_j) & \text{(substitution).} \end{cases}$$

Hence, we require the distances $d_c(x[i-1], y[j])$, $d_c(x[i], y[j-1])$ and $d_c(x[i-1], y[j-1])$ to calculate the distance $d_c(x[i], y[j])$.

If one of the prefixes has length zero then not all of the three mutations can be performed. We cannot delete a character from or substitute a character into the empty string. To transform $x[i]$ to the empty string $y[0]$ with minimum cost, we will never insert or substitute a character. Each character that is inserted or substituted has to be deleted, thus we can omit the insert and substitute mutations and get a sequence of mutations with lower cost. The lowest cost transformation from $x[i]$ to $y[0]$ is thus the sequence of basic mutations which consists only of deletions. Similarly, in the case of the transformation from $x[0]$ to $y[j]$, the lowest cost transformation is to insert the characters of the string $y[j]$. Hence, for $i = 0$ and $j > 0$ we perform only insertions and for $i > 0$ and $j = 0$ we perform only deletions. The cost of the sequences of mutations in the case of $i = 0$ and $j > 0$ is $2 \cdot j$ and in the case of $i > 0$ and $j = 0$ is $2 \cdot i$. If both i and j are zero then we have to transform the empty string to the empty string. Of course, we do not need any mutation for this transformation. Hence, the cost $d_c(x[0], y[0])$ is zero.

For example, consider the two DNA sequences $x = AGT$ and $y = CAT$. Assume that we know the distances $d_c(x[1], y[2]) = d_c(A, CA)$, $d_c(x[2], y[1]) = d_c(AG, C)$ and $d_c(x[1], y[1]) = d_c(A, C)$. Then we can use the scheme above to get the evolutionary distance $d_c(x[2], y[2]) = d_c(AG, CA)$ by calculating the minimum of

- $d_c(x[1], y[2]) + c(a_2 \rightarrow) = d_c(A, CA) + c(G \rightarrow) = d_c(A, CA) + 2$,
- $d_c(x[2], y[1]) + c(\rightarrow b_2) = d_c(AG, C) + c(\rightarrow A) = d_c(AG, C) + 2$, and
- $d_c(x[1], y[1]) + c(a_2 \rightarrow b_2) = d_c(A, C) + c(G \rightarrow A) = d_c(A, C) + 3$.

The minimum of these three cases is the evolutionary distance $d_c(AG, CA)$.

The Algorithm

How to get an algorithm from this scheme? The distances of most prefixes of x and y are required multiple times. For example, the distance $d_c(x[i-1], y[j-1])$ is required to calculate $d_c(x[i-1], y[j])$, $d_c(x[i], y[j-1])$ and $d_c(x[i], y[j])$. To save time, we only want to calculate $d_c(x[i-1], y[j-1])$ once. Hence, we have to keep this value to use it multiple times without recalculation. To do this, we use a table in which we store all previous calculated evolutionary distances of prefixes of x and y . We store in the cell (i, j) (row i and column j) of the table the value of the evolutionary distance $d_c(x[i], y[j])$. The advantage of this method is that we only need to calculate the distances of prefixes once and can use a simple table lookup operation to get the value again. We require the evolutionary distance for all $0 \leq i \leq m$ and $0 \leq j \leq n$ to calculate the distance of the DNA sequences x and y . Thus the table consists of $m+1$ rows and $n+1$ columns.

		$y = TCAAT$					
		0	1	2	3	4	5
		0	—	—	—	—	—
$x = ATGAACG$		0	2	4	6	8	10
T		2	3	5	4	6	8
G		4	2	4	6	7	6
A		6	4	5	7	9	8
A		8	6	7	5	7	9
A		10	8	9	7	5	7
C		12	10	8	9	7	8
G		14	12	10	11	9	10

Fig. 31.1. Table for the calculation of the evolutionary distance of $x = ATGAACG$ and $y = TCAAT$

For example, let $x = ATGAACG$ and $y = TCAAT$. The corresponding table for the calculation of the evolutionary distance is given in Fig. 31.1.

We start by calculating the distance of $d_c(x[0], y[0])$ and storing the value in cell $(0, 0)$. As mentioned above, we know that this distance is always zero. We already know the values of the entries in Column 0 and in Row 0. In Column 0 we only perform deletions and in Row 0 only insertions. Hence, we store the values $2, 4, 6, \dots$ in this column and this row.

We use the developed scheme to calculate the values of the other cells. For example, consider the cell $(2, 1)$. The cell represents the evolutionary distance of $x[2] = AT$ and $y[1] = T$. The deletion of A is intuitively the only mutation of minimum cost to transform $x[2]$ to $y[1]$. The algorithm has to calculate the same distance. It chooses one of the following possible mutations:

1. $d_c(x[1], y[1]) + c(a_2 \rightarrow) = d_c(A, T) + c(a_2 \rightarrow) = 3 + 2 = 5$ (deletion of T),
2. $d_c(x[2], y[0]) + c(\rightarrow b_1) = d_c(AT, y[0]) + c(\rightarrow b_1) = 4 + 2 = 6$ (insertion of T) and
3. $d_c(x[1], y[0]) + c(a_2 \rightarrow b_1) = d_c(A, y[0]) + c(a_2 \rightarrow b_1) = 2 + 0 = 2$ (substitution of T by T).

The substitution of T by T is no mutation. We use this to keep the notation simple. In reality we do not substitute T by T . Hence, the cost for this operation is zero. The deletion of A is not explicitly given in this step of the algorithm. This mutation is performed during the transformation of $x[1] = A$

to $y[0]$. Since the algorithm will choose the third possible mutation in the last step, it will generate the solution which we have intuitively generated.

The lines in the table that form a path from cell $(0, 0)$ to a cell (i, j) , represent the possible transformations from $x[i]$ to $y[j]$. In the case of $(2, 1)$, we used the path over $(1, 0)$ by first deleting A and then substituting T by T . The table shows that there might be multiple paths to a cell (i, j) . Hence, there might exist multiple different optimal sequences of mutations to transform x to y . The lines can be calculated by the algorithm, as these just represent the case that lead to a minimum distance in a single step. The red colored lines constitute the path of minimum cost for the transformation of x to y . Hence, these represent the sequences of mutations that transform x to y with minimum cost and which can be used to get the evolutionary distance.

We do not know which cells we require for the calculation of a path of minimum cost to transform x to y . Thus, we have to calculate the values of all cells. To calculate the value $d_c(x[i], y[j])$ of cell (i, j) , we need the values of the cells $(i - 1, j)$, $(i, j - 1)$ and $(i - 1, j - 1)$. To make sure that we have already calculated these values, we generate the table row by row or column by column. If we do it row by row, we need to go through the rows from the left to the right. Similarly, if we generate the table column by column, we need to go through the columns from the top to the bottom. This ensures that all required values are stored in the table, when we calculate the distance $d_c(x[i], y[j])$. At the end, the evolutionary distance $d_c(x[m], y[n]) = d_c(x, y)$ is stored in the cell (m, n) .

Conclusion

Starting with the smallest subproblem, the calculation of $d_c(x[0], y[0])$, we have solved larger and larger subproblems. In each step we have increased the lengths of the prefixes of x and y and calculated their evolutionary distances. For the calculation of the distance $d_c(x[i], y[j])$ we have used the distances $d_c(x[i - 1], y[j])$, $d_c(x[i], y[j - 1])$ and $d_c(x[i - 1], y[j - 1])$. Hence, we have used the optimal solution of these smaller subproblems to solve the larger subproblem.

Given a problem, the calculation of a solution of minimum cost is called an *optimization problem*. The calculation of the evolutionary distance of two DNA sequences is such an optimization problem. We have used a specific technique to create an algorithm for our optimization problem. This generic technique may be applied to other but not all optimization problems. Implicitly, we have used the following property of our optimization problem:

- Every subsolution of an optimal solution which is a solution for a subproblem is an optimal solution for that subproblem.

Many optimization problems have this property. The technique we have used for solving the problem of finding the evolutionary distance may be applied

to any problem with this property. This technique is called *dynamic programming*. In the case of dynamic programming, we split the problem into subproblems. We solve the smallest subproblems directly. In our case, this is the calculation of $d_c(x[0], y[0])$. The solution for this subproblem is zero. From the optimal solutions of small subproblems we calculate the optimal solutions of larger subproblems. We repeat this until we have calculated the optimal solution of the original problem. Dynamic programming is an important generic technique that is often used for the development of algorithms.

The algorithm for calculating the minimum evolutionary distance of two DNA sequences can be used for other purposes than calculating the similarity of two species. For example, one can use it to measure the similarity of two words. This can be useful for spellchecking software. The correct spelling of a word has most probably a very small distance to the incorrectly spelled word given by the user. So the software may show the user all words within a given maximum distance as possible correct spellings of the given word.

References

The references below provide a generic introduction to dynamic programming. They present the theoretical background and also examples of its application.

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein: *Introduction to Algorithms*. MIT Press, 2nd edition, 2001. Chapter 3.
2. Jon Kleinberg, Éva Tardos: *Algorithm Design*. Addison-Wesley, 2005. Chapter 6.

Part IV

Optimization

Overview

Heribert Vollmer and Dorothea Wagner

Universität Hannover, Hannover, Germany
Karlsruher Institut für Technologie, Karlsruhe, Germany

How can we find the shortest way to go from one city to another? How can we find the order in which to visit different cities such that the resulting round tour is the shortest among all possibilities? In the final part of this book we look at such tasks where, given a generally very large set of “possible solutions” to an algorithmic goal, we have to determine in a certain sense the “optimal solution.” In computer science, such problems are known as optimization problems.

We will see that for many optimization problems very tricky algorithms are known that produce an optimal solution very quickly (“efficiently”). The above-mentioned shortest-path problem is one of them. In Chap. 32 an algorithm for this and related problems will be presented. Also the subsequent six chapters of the final part of this book explain efficient procedures to find solutions for different optimization problems. In Chap. 33 some islands have to be connected via a system of bridges in such a way that it is possible to drive from each island to each other, but the number and size of the bridges has to be as small as possible. Chapter 34 explains how car traffic can be distributed among the different streets of a city with different numbers of lanes in such a way that we have as few traffic jams as possible. This is an example of a so-called network flow problem – these problems are of immense importance in computer science today. The task of a dating service to arrange meetings among marriage-minded ladies and gentlemen is solved optimally (at least from a theoretical point of view) in Chap. 35. In Chap. 36 we must choose the location for a new suburban fire brigade headquarters.

Finally we have to find solutions for optimization problems where the problem specification is not completely known from the beginning. For these online problems the parameters become known only little by little. In Chap. 37 we have to decide if, for a skiing holiday, it is better to buy or rent the skis, but we do not know yet if we will reuse the skis later. In Chap. 38 we want to move, and we want to use as few boxes as possible to pack our stuff, but we are not fully decided yet which things we want to move and which we want to throw away.

For many other important optimization problems no efficient solution algorithm is known to this day. The only way to find the optimal solution is to compare all possible solutions. The time requirement for this simple procedure is of course dependent on the number of solutions and hence in general it is very large. For example, in Chap. 39 a knapsack has to be packed optimally for a hike, but there are many different ways to use its capacity. Also the above problem to determine the shortest round-trip through a number of cities is one of the hard problems for which we do not know how to find an optimal solution. But in Chap. 40 we will see how a so-called approximation algorithm finds a tour that is maybe not the shortest one but one whose length usually is quite close to the optimum; in the worst case it is twice as long. The final chapter of this book, Chap. 41, introduces simulated annealing, an algorithmic method that produces approximate solutions for a number of optimization problems with certain mathematical properties. The name of this magical method is due to an analogy with an industrial technique that involves the heating and controlled cooling (“annealing”) of a material to improve its stability.

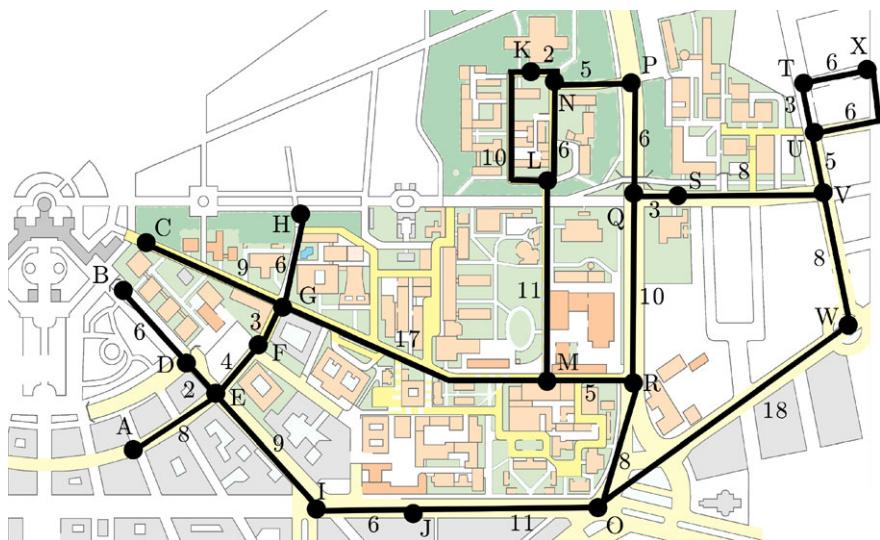
Shortest Paths

Peter Sanders and Johannes Singler

Karlsruher Institut für Technologie, Karlsruhe, Germany

I have just moved to Karlsruhe and into my first flat. Such a big city is rather complicated. I already have a city map, but how can I find the fastest way to get from A to B? I like cycling but I am notoriously impatient, so I really need the shortest path to the university, to my girlfriend, and so on.

Systematic planning could look like this: I pin the city map on a table and put thin yarn threads along the streets, knotting them at crossroads and junctions. I also knot all possible start points, end points and dead ends.



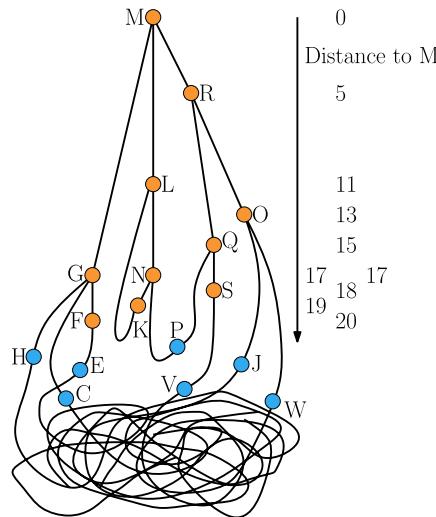
Copyright Karlsruher Institut für Technologie, Institut für Photogrammetrie und Fernerkundung

Here comes the trick: I pick the starting knot and slowly lift it up. One after another the knots leave the table surface. I have labeled the nodes so that

I always know where each knot comes from. At last, all knots hang vertically below the starting knot.

The rest is really easy: To find the shortest path, I only have to find the end knot and trace the straight threads back to the start. The distance between both points can then be found with a measuring tape. The path found this way must indeed be the shortest, because if there was a shorter one, it would have kept the start and end closer together.

Suppose, for example, I need the shortest path from the cafeteria (M) to the computing center (F). I pick knot M, lifting all other knots off the table. The figure below shows the situation at the moment when knot F hangs in the air for the first time. To make the figure clearer, the knots are pulled apart horizontally a bit. The orange knots are hanging, the numbers on the right indicate the distance to M using the thread length from the first figure.

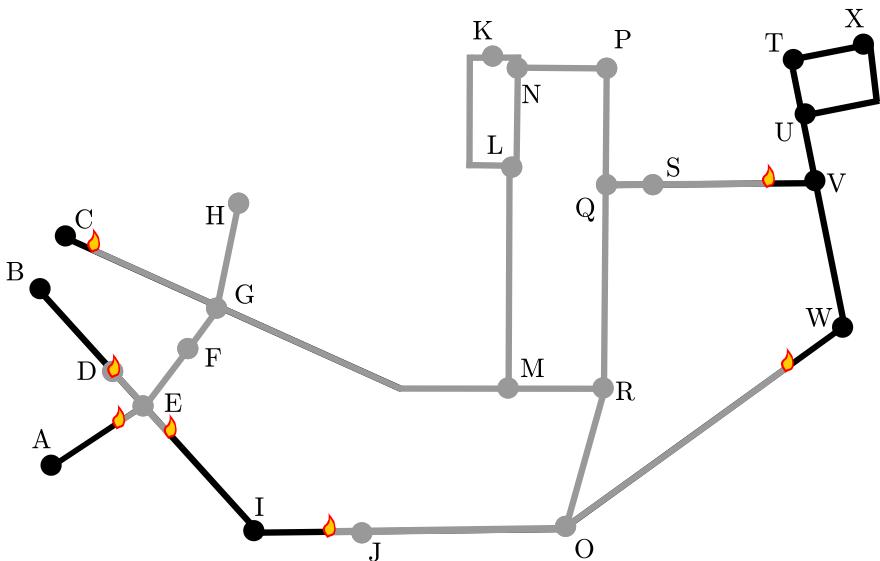


It is obvious that the shortest path from M to F leads via G. Between L and K, the thread is already sagging with no chance of hanging any straighter. This means there is no shortest path from M using this connection.

I have tried this method successfully for the campus and its surroundings. But I failed miserably with my first trial run for the whole city of Karlsruhe, which produced nothing but a heap of tangled threads. It took me half the night to disentangle them and lay them out on the city map again.

My younger brother drops by the next day. “No problem,” he says, “I will solve the problem with superior technology!” He turns to his chemical kit and soaks the threads in a mysterious liquid. Good grief! He ignites the web at the starting point. Seconds later, the room disappears in a cloud of smoke. This pyromaniac turned the threads into fuses. He explains proudly: “All

threads are burning at the same speed. So the time before a knot catches fire is proportional to the distance from the starting point. Besides, the direction from which a knot catches fire contains the same information as the straight threads of my hanging web approach.” Great! Unfortunately, he forgot to record the inferno, so we have only ashes left. Even with a video tape I would have to start over with every new starting point. Below, there is a snapshot of the threads after the flames from starting point M have burned part of the way (gray).



I throw my brother out and start to think. I have to get over my fear of abstraction and make the problem clear to my stupid computer. This does have certain advantages: threads that do not exist cannot get tangled up or burn. My professor told me that back in 1959 a certain Mr. Dijkstra developed an algorithm that solves the shortest-path problem in a way that is quite similar to the thread method. Neatly enough, Dijkstra’s algorithm can be described in thread terminology.

Dijkstra’s Algorithm

Mainly, it is about simulating the thread algorithm. For every knot, a computer implementation must know the threads starting from it and their respective lengths. It also administrates a table d which estimates the distance from the starting point. The distance $d[v]$ is the length of the shortest connec-

tion from the starting knot to v using only *hanging* knots. As long as there are no “hanging connections,” $d[v]$ is infinite. Therefore, in the beginning, $d[\text{starting knot}] = 0$, and $d[v] = \text{infinite}$ for all other knots.

The pseudocode given here describes the calculation of all knots’ distances to the starting knot:

Dijkstra’s Algorithm in Thread Terminology

```

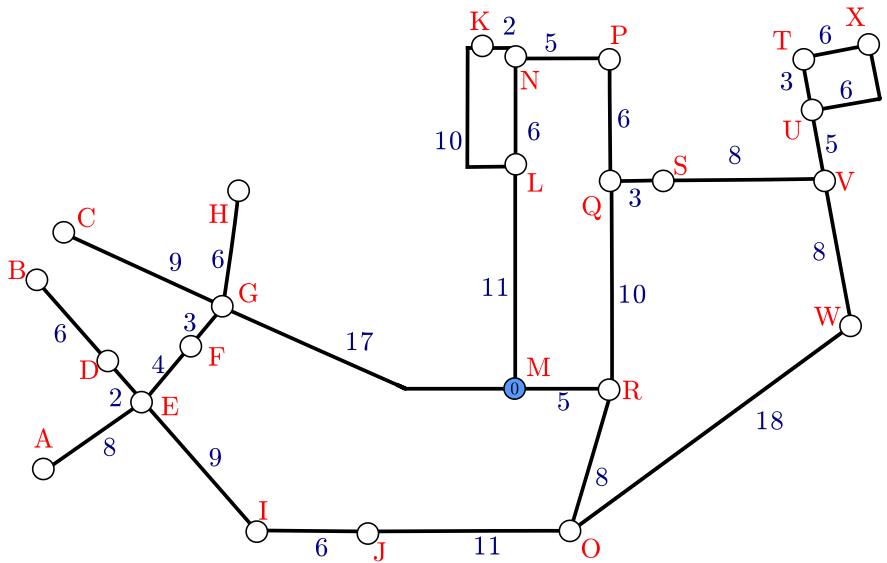
1  all knots are waiting, all  $d[v]$  are infinite, only  $d[\text{starting knot}] = 0$ 
2  while there are waiting knots do
3       $v :=$  the waiting knot with the smallest  $d[v]$ 
4      Turn  $v$  hanging
5      for all threads from  $v$  to a neighbor  $u$  of the length  $\ell$  do
6          if  $d[v] + \ell < d[u]$ , then  $d[u] := d[v] + \ell$ 
              // found shorter path to  $u$ , leads via  $v$ 
```

How is this algorithm linked to the process of slowly lifting up the starting knot? Each iteration of the while-loop corresponds to the transition of knot v from waiting to hanging. The next knot lifted in turn is the waiting knot v of the smallest value $d[v]$. This value is the height to which we have to lift the starting knot to make v hanging. Since other threads lifted to this height later on cannot decrease it, $d[v]$ is the definitive distance from the starting knot to v .

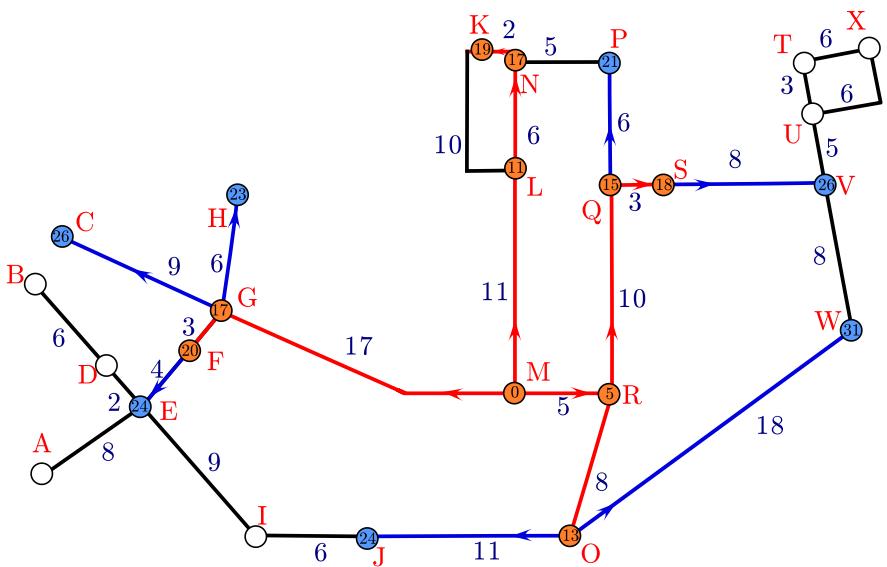
The plus side of Dijkstra’s algorithm is that the $d[u]$ values of the other knots can easily be adapted when v becomes hanging: only the threads starting from v have to be considered, which is done by the inner for-loop. A thread between v and a neighboring knot u of the length ℓ builds a connection of the length $d[u] := d[v] + \ell$ from the starting knot to u . If this value is smaller than the last value of $d[u]$, the latter is diminished accordingly. In the end, all knots reachable from the starting knot are hanging and the $d[u]$ values give the lengths of the shortest paths.

In the following, you see some steps in the execution of the algorithm. Hanging knots are orange, blue ones are waiting and the remaining, as of yet unreached knots are white. Inside the circles the current $d[u]$ is given. After the algorithm has finished you can go backwards from the target knot and along the red threads to find the shortest path.

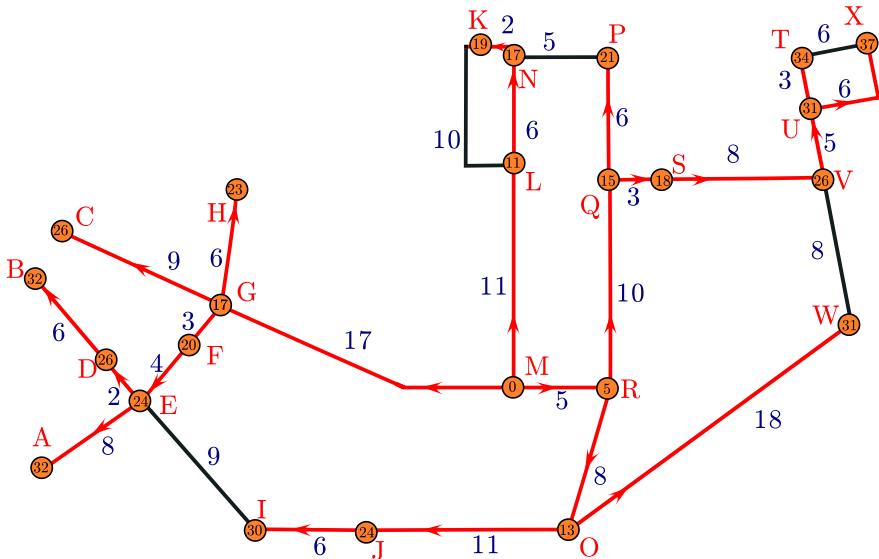
The algorithm starts with the following configuration, when all knots are still lying on the table.



The next figure shows the algorithm's state after ten steps, i.e., the same situation as in the hanging thread web in this chapter's first figure.



The end state looks like this: All knots are hanging in the air, and the shortest paths from M lead along the red threads.



With my computer implementation, I can now calculate distances between knots to my heart's content, without having to clean up ashes or disentangle threads. My anger is subsiding slowly, and maybe my brother will not be permanently banned from my flat after all. However, for real route planning, I must expand Dijkstra's algorithm to make the shortest paths themselves available. Whenever $d[u]$ is set to a new value, the program remembers which knot was responsible for this: knot v that has just been lifted. In the end, the route is reconstructed by going backwards, following the predecessor information from the target to the start knot. The pointers contain the same information as the red threads in the figures.

FAQs and Further Reading

Where can I find a more detailed description of Dijkstra's algorithm? There are many good algorithm textbooks explaining everything there is to know, e.g., K. Mehlhorn, P. Sanders. *Algorithms and Data Structures – The Basic Toolbox*. Springer, 2008.

Who was Dijkstra? Edsger W. Dijkstra (http://de.wikipedia.org/wiki/Edsger_Wybe_Dijkstra) was born in 1930 and died in 2002. Not only did he invent the algorithm described above, he also made substantial contributions in the field of systematic programming and to the modelling of parallel processes. In 1972 he was presented with the Turing Award, the most prestigious

award for computer scientists. His famous article “*A note on two problems in connexion with graphs*” was published in 1959 in the journal *Numerische Mathematik*. The “other” problem he mentions is the calculation of minimum spanning trees. If you leave out the “ $d[v] +$ ” in line 6 of our pseudocode you get the Jarník–Prim algorithm from Chap. 33.

What are threads, etc., in “technicalese”? *Knots* are called *nodes* in computer science; instead of threads we have *edges*. Networks of nodes and edges form *graphs*.

Have I not already come across something like this in this book? In computer science, the search for paths and the related problem of finding circles are very important.

- *Depth search* systematically lists certain paths, which is the basis of many algorithms. See for example Chap. 7 (Depth-First Search) and Chap. 9 (Cycles in Graphs).
- The *Eulerian Circles* in Chap. 28 use each edge exactly once.
- The Travelling Salesman Problem described in Chap. 40 is concerned with a round trip between cities that has to be as short as possible. Determining the travelling time between the cities, however, is a shortest path problem.

How to implement the pseudocode efficiently? We need a data structure that supports the following operations: insert nodes, delete nodes with the shortest distance, and change distance. Since this combination of operations is needed quite often, there is a name for it – *priority queue*. Fast priority queues need time at most *logarithmic* in the number of nodes for any of the operations.

Can we do this even faster? Do we really have to look at the whole Western European road network in order to find the shortest path from Karlsruhe to Barcelona? Common sense says otherwise. Current commercial route planners only look at highways when “far apart” from starting points and destinations, but cannot guarantee not to overlook short cuts. In recent years, however, faster procedures have been developed that guarantee optimal solutions, see, for example, the work of our group <http://algo2.itii.kit.edu/routeplanning.php>.

Is this a route planner for road networks only? The problem is much more common than it seems. For instance, Dijkstra’s algorithm does not have to know about a node’s geographical position, and is not limited to (spatial) distances, but can also use travelling times as thread lengths. This even works for one-way streets or differing travelling times for the trips from A to B and back, since our algorithm only considers the thread length from start to end. The network can also model many other things, e.g., public transport including departure times, or communication channels in the internet. Even problems that, at first sight, look unrelated to paths, can often be rephrased appropriately. For example, the distance between two strings of characters (genome sequences) in Chap. 31 can be interpreted as a distance in a graph.

Nodes are pairs of letters of the two inputs that are matched. Edges encode the operations *delete*, *insert*, *replace* and *transfer*.

Is it possible to have streets of negative length? This can be quite useful, e.g., it is possible to factor in that my favorite ice cream parlor is in a certain street, so I do not mind detours. However, a round trip of negative length is not allowed, otherwise you could go in circles for as long as you like (eating ice cream) while the path is continuously becoming shorter – the concept of *the* shortest path would not make sense anymore. But even if there are no negative circles, Dijkstra's algorithm could fail. The problem is that via a knot already hanging, a thread of negative length could provide improved routes for other nodes. Dijkstra's algorithm does not handle this case. A better-suited alternative is Bellman and Ford's algorithm, which takes more care to cover all cases but is much slower.

Minimum Spanning Trees (Sometimes Greed Pays Off ...)

Katharina Skutella and Martin Skutella

Technische Universität Berlin, Berlin, Germany

Once upon a time in a remote island kingdom there lived the Algo clan. The clansmen lived scattered all over the seven islands of the kingdom shown below.

The seven islands and the mainland were connected by several ferries allowing visits and excursions to the mainland. The ferry connections are plotted in dashed lines on the map (Fig. 33.1). The numbers indicate the length of the ferry connection in meters.

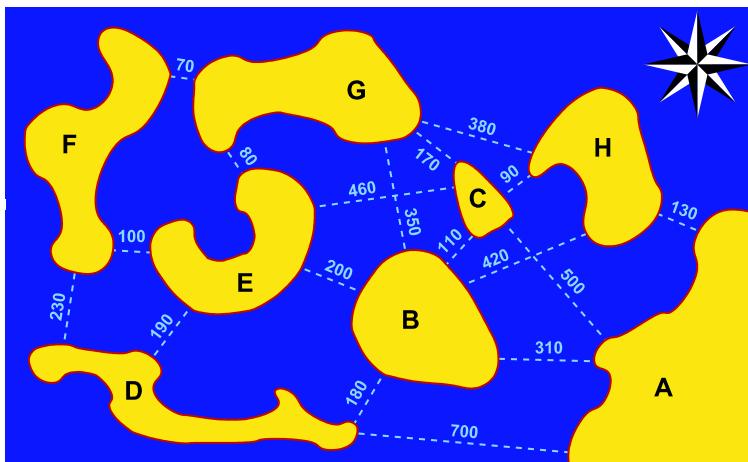


Fig. 33.1. The island kingdom of the Algos comprised the seven islands B, C, ..., H. The dashed lines depict ferry connections. The numbers indicate the lengths of the ferry connections in meters. For example, one ferry cruised between the mainland A and the island D, covering a distance of 700 m one-way

The Bridge Project of the Algos

Once in a while, during stormy weather, ferries capsized. Therefore, the Algos decided to replace certain ferry connections by bridges.

In the first year, one of the seven islands was going to be connected to the mainland by a new bridge. The Algos built a bridge of length 130 m between A and H, since all connections from A to other islands are longer.

During the second year, they wanted to connect another island to the mainland. Therefore, the construction of a bridge between A or H and another island was taken into consideration. The Algos built the shortest possible bridge of length 90 m between H and C.

In the third year, the construction of another bridge (starting from A, C, or H) should connect a third island to the mainland. This time, the shortest possible option was the bridge of length 110 m connecting B and C.

The status quo of the ongoing building project is shown in Fig. 33.2. As you can see, the Algos were not yet ready with their project.

In the following years, the bridge of length 170 m between C and G, then the bridge of length 70 m connecting F and G, next the bridge of length 80 m from G to E, and finally the bridge of length 180 m connecting B and D were built.

Finally, after seven years, all islands were connected to each other and to the mainland by bridges. The bridge project was therefore completed.

The final bridge system of the Algos is shown in Fig. 33.3.

The Algos were delighted. The time and effort for building the bridges had been enormous, but they were convinced that they had avoided the con-

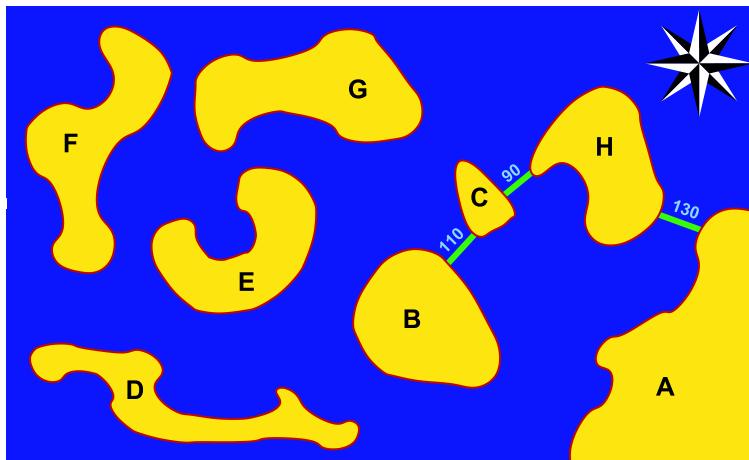


Fig. 33.2. The status quo of the building project after three years. The islands B, C, and H are already connected to the mainland

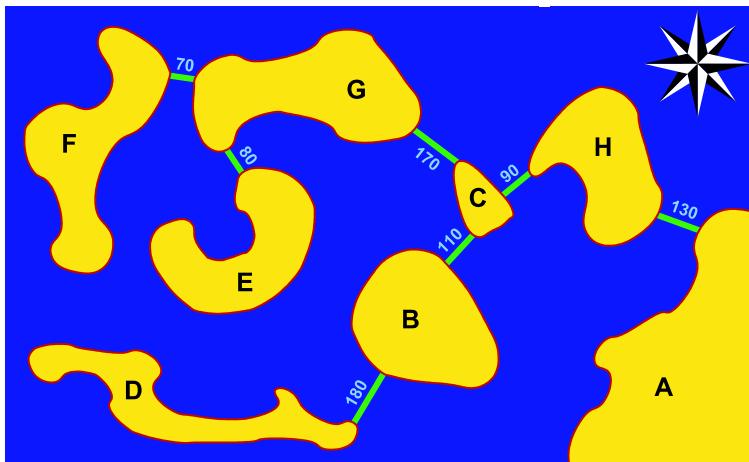


Fig. 33.3. The final bridge system of the Algos

struction of excessively long bridges as far as possible. The total length of all bridges added up, as you can easily check, to exactly 830 m.

Building Bridges After the Hurricane

Shortly after the completion of the last bridge, a terrible hurricane swept over the kingdom and completely destroyed the precious bridges. After recovering from the shock, the Algos decided to construct a new bridge system. Again, the bridges were supposed to connect all islands to each other and to the mainland.

Due to the hurricane, there was a lack of building material. It was agreed to first build a shortest possible bridge. Therefore, in the first year after the hurricane, the bridge of length 70 m connecting F and G was built.

Also in the second year building material was scarce, so that the next longer bridge of length 80 m from E to G was built.

According to this strategy, in the third year the bridge of length 90 m between C and H was built. After the construction of these three bridges the three islands E, F, and G and the two islands C and H were connected to each other (see Fig. 33.4).

In the fourth year, the shortest connection that had not been realized yet was the one of length 100 m between E and F. Because both islands were already connected to each other via G, they instead built the bridge of length 110 m connecting B and C.

During the fifth year, the bridge of length 130 m from A to H was added, then the bridge of length 170 m connecting C and G, and finally, in the seventh

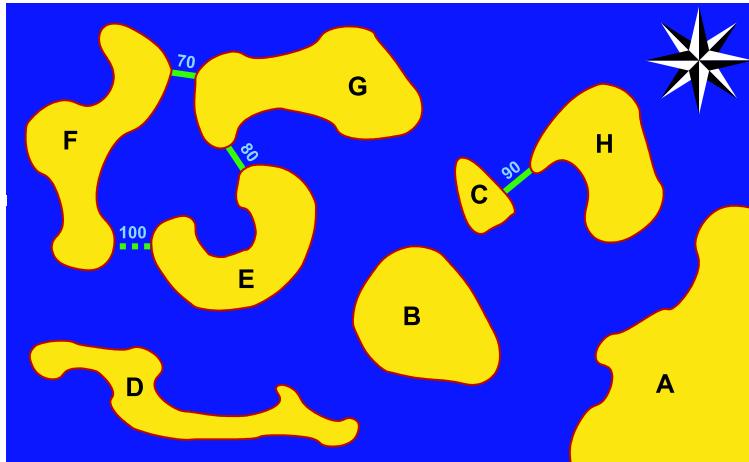


Fig. 33.4. The status quo of the second bridge project three years after the hurricane. The bridge system consists of three bridges of lengths 70 m, 80 m, and 90 m. The shortest connection that has not been realized yet is the one between E and F (marked by a dashed line on the map). In the fourth year, however, the Algos decided against building this bridge, because these two islands were already connected to each other via G

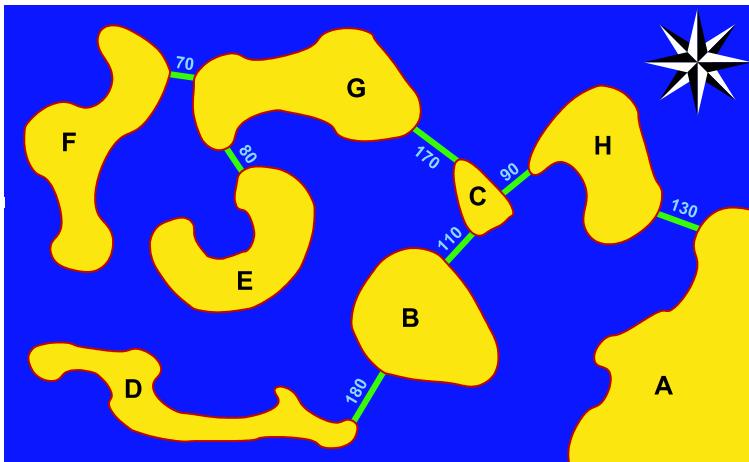


Fig. 33.5. The second bridge system of the Algos

year, the bridge of length 180 m from B to D. The new bridge system of the Algos is depicted in Fig. 33.5.

With great astonishment the Algos realized that despite their new strategy in building bridges, they had ended up with the same bridge system of total length 830 m (compare Fig. 33.3). This confirmed the Algos' belief that they

had found the optimal bridge system for their islands. And unless no second hurricane has made trouble ever since, the Algos stroll happily and proudly over their bridges till today.

The Algorithms of Prim and Kruskal

Now you probably wonder, whether the Algos were rightly proud of their bridge system. Perhaps, there is still a better, thus shorter bridge system? Using trial and error you can find out that any other bridge system that connects the seven islands and the mainland to each other is longer than 830 m.

A bridge system of minimum total length that connects several places (here mainland A and islands B to H) to each other is called “minimum spanning tree.” The problem of finding a minimum spanning tree has many different practical applications aside from building bridges. For example, it arises when planning the sewage system of a new housing estate. The goal is to connect all estates to the sewage system at the lowest possible price. Other applications are the design of computer chips and the planning of traffic or communication networks (telephone, TV, internet, etc.).

Both strategies of the Algos exemplify well-known algorithms for solving the problem. The first strategy is known as “Prim’s Algorithm.” This algorithm connects places to the mainland one after another. In each step the shortest possible bridge is being built.

Prim’s Algorithm

- 1 Choose a special place (mainland) and call it reachable.
- 2 Initially all other places are not reachable.
- 3 Repeat the following steps until all places are reachable:
Build the shortest bridge between two places, of which one is reachable and one is not reachable, and call the one that had not been reachable yet reachable.

The second strategy described above (after the hurricane) is known as “Kruskal’s Algorithm.” This algorithm builds in each step the shortest bridge connecting two places that have not been connected before. It differs from Prim’s Algorithm in the way that it not only takes bridges establishing a connection to the mainland into consideration, but allows for arbitrary bridges connecting two places that have not been connected yet.

Kruskal’s Algorithm

- Repeat the following step until all places are connected to each other through bridges:
- Build the shortest bridge that connects two places that are currently not yet reachable from each other.

The algorithms of Prim and Kruskal both compute a minimum spanning tree and they have something in common. Recall the strategy of the Algos. In both strategies, the Algos planned fairly nearsightedly from year to year. Every year they chose the best (i.e., shortest) bridge coming into question at that time. In doing so they did not take into consideration the effects of their decisions for the future of their construction project. They acted “greedily.” Algorithms with this property are also called “greedy,” because, at every step, they make the best possible choice under the current circumstances. As you can see, sometimes greediness pays off.

But such a greedy approach is not always successful. Imagine, for example, you were asked to connect only island D to the mainland A by a bridge system of minimum total length. The greedily designed bridge system of the Algos connects D to A via the islands H, C, and B. The total length of bridges along this path is 510 m. You can certainly find a shorter connection from mainland A to island D! If you want to learn more about how to find the shortest connection from mainland A to island D, go ahead and check Chap. 32.

In fact, the two algorithms described above have another interesting property. They always compute a solution, in which the length of the longest bridge is as small as possible. You can check this using the example of the island kingdom.

Further Reading

1. Chapter 32 (Shortest Paths)

Not all Algos were happy with their bridges. For example, chief “Limping Leg” from island D, who regularly consulted the medicine man on the mainland, complained that the way from D to A via B, C, and H was obviously too long (510 bridge meters). One should have better built a bridge from A to B, which would have reduced the distance to 490 bridge meters. Find out in Chap. 32 how to find the shortest connection from the mainland to all bridges!

2. Chapter 40 (Travelling Salesman Problem)

Also milkman “Whining Whey,” who had to deliver a bag of coconuts to each island day-to-day, kept complaining. In his opinion, one should have built the bridges in such a manner that they yield a shortest round trip starting from the mainland and passing by all islands. How to please the milkman is the topic of Chap. 40.

3. Chapter 3 (Fast Sorting Algorithms)

To apply Kruskal’s Algorithm, it is advisable to first sort the possible connections according to their lengths and then process them in ascending order. You can find out how to sort fast in Chap. 3.

4. Chapter 9 (Cycles in Graphs)

In the fourth year after the hurricane, the Algos did not build the shortest bridge connecting E to F, but instead the bridge from B to C. Because

the bridge from E to F would have connected two islands, which were already connected by several other bridges. In other words: building a bridge from E to F would have closed a cycle from E to G via F and back to E. More on cycles and how to detect them is discussed in Chap. 9.

5. T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein: *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

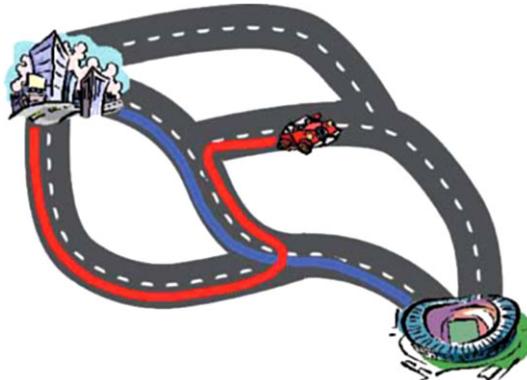
How to implement Prim's and Kruskal's Algorithms on a computer, such that they find the solution as fast as possible, and many more things, can be found in this textbook, which is frequently used in courses for first-year computer science students.

Maximum Flows – Towards the Stadium During Rush Hour

Robert Görke, Steffen Mecke, and Dorothea Wagner

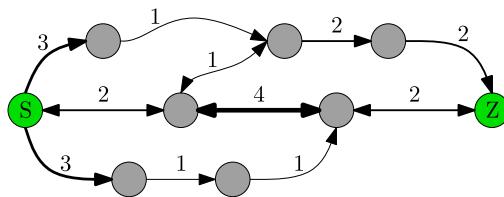
Karlsruher Institut für Technologie (KIT), Karlsruhe, Germany

“What the heck is this? We’ll never get to the soccer stadium this way!” Jogi sat in the car, next to his mother, and was beginning to get nervous. “It’s not my fault that everybody’s using this street to the stadium. Now there is a traffic jam,” she said. “Then simply turn around, and let’s take Ford Street over there. Nobody uses that route.” Jogi’s mother did not really buy this, but for the sake of peace she drove back, and, indeed, on Ford Street there was less traffic. Up to the next junction at least. There, Ford Street led into busy and broad Station Street, and there was the traffic jam again. “These idiots don’t know what they’re doing. Turn left, mum!” – “But the stadium is straight ahead,” she replied. “That’s true,” Jogi returned, “but we can take Karl Street over there. While that is a detour, that street will definitely be free.” Jogi’s mum remained skeptical, but she gave it a try. And it actually appeared that Jogi was right. Nobody was taking that detour. Jogi even made a futile attempt to encourage other cars coming towards them to turn around, but nobody followed them. “Those fools! In a second they’ll be stuck in the traffic jam although they could easily get through here!”



Jogi arrived at the stadium in time, but the match turned out to be a boring waste of time, which made Jogi think about the traffic situation earlier. “Letting drivers choose their routes by themselves easily leads to congestion. Traffic signs should be put up at each junction in order to route the cars in such a way that traffic keeps flowing and as many cars as possible can reach their goal soon. But how can we find the best solution for this routing problem?” He did not arrive at a satisfactory answer immediately. However, a few days later he spoke to his older sister about the incident. She studied computer science but nonetheless did not have a solution at hand.

“Let’s simplify the problem as much as possible first: Let’s assume that all drivers start from the same point …” She marked the point on a piece of paper and labeled it S for “start.” “…and want to go to another point.” She marked that one with a Z for Zuse Stadium. “In between, there are streets which meet at the junctions.” She drew several more points and lines between the start and Zuse Stadium.

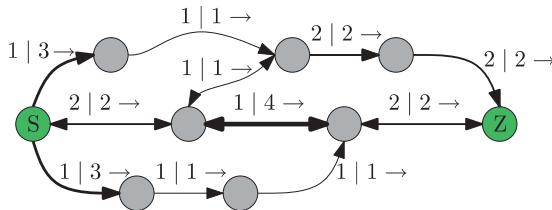


“But the streets can differ in capacity. Let’s put the number of lanes right next to each street and thicken the lines in the drawing accordingly. Hmm, we learned about shortest paths in a lecture recently, but that doesn’t help much here. Of course we can start by finding the shortest path from S to Z .” – “This one here?” Jogi marked it in the drawing. “But now we can continue looking for more routes. We just have to record how many cars are already using each street.”

Let us leave Jogi and his sister alone for now and continue their approach: We are given a road network with road capacities, and we would like to know how to route the cars in order to let traffic flow optimally. To keep things simple, we assume that all cars start at S and aim for Z . Usually, car drivers try to take the shortest path to their goal. But when too many cars take a route at the same time, the traffic gets stuck. In our case “at the same time” means “in the hour before the soccer match starts.” Each street can only handle a certain number of cars passing through it (the *capacity* of the street). This number does not so much depend on the street’s length but rather on its width (the number of lanes). For example, a street with 1 lane can be used by 1000 cars per hour. However, we still write 1 (the number of lanes) instead of 1000, since we want to avoid dealing with such big numbers.

Other critical points in road networks are junctions. In case more cars arrive at a junction than can continue, we have a congestion. By contrast, if the number of cars that can leave a junction does not fall short of the

number of arriving cars, all is well. We now ask ourselves how many cars we can simultaneously push through the network from S to Z . (“Simultaneously” means “per hour” here.) A solution to the problem in our example would be the following “traffic flow”:



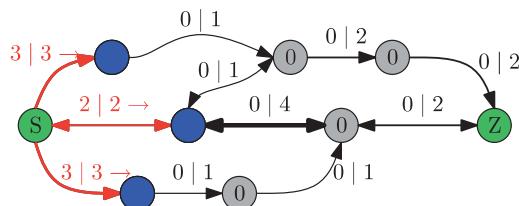
Each street is now additionally labeled by the number of cars using it and an arrow denoting their driving direction. Thus, $1 | 3 \rightarrow$ means that one of three available lanes is in use by cars going to the right. The first number must never exceed the number of lanes (i.e., something like $3 | 2$ is not allowed). This rule is so important that we give it a name. We call it the *capacity rule*. The requirement that the numbers of cars leaving and arriving at a junction are equal (otherwise cars will get stuck) is called *flow conservation rule* (as it ensures a steady flow at junctions). Only S and Z are an exception to this rule.

Among all traffic flows fulfilling these two rules, we now seek one which allows as many cars as possible to start off simultaneously or – which is the same – to arrive at the goal. Computer scientists call the result a *maximum flow*.

This kind of problem does not only occur in the field of traffic routing. For instance, you could also think about how to evacuate buildings as fast as possible or how to route data through a computer network. Can you think of other examples?

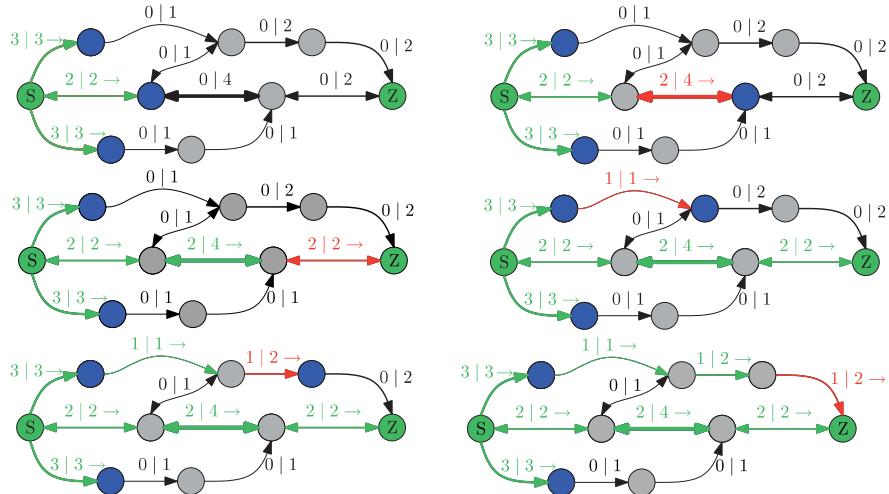
The Algorithm

So how do we find a maximum flow? Let's just give it a try: to start with, there are no cars at all in our road network. We start by just sending out (red in the picture) as many cars as possible from S without violating the capacity rule.



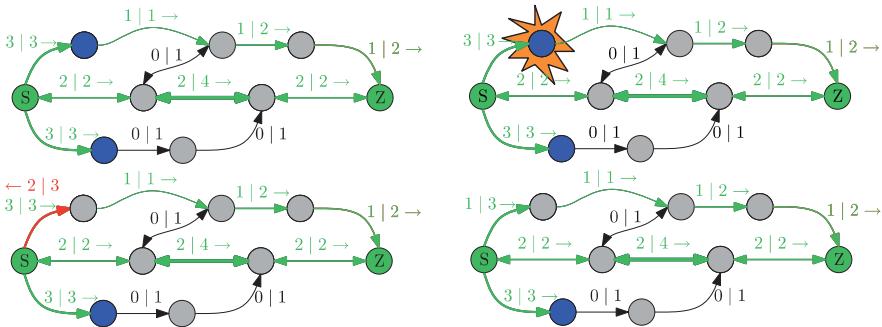
On each street leaving S , there are now as many cars as there are free lanes (green). Of course, we do not use real cars for this purpose but, e.g., toy cars instead. Or simply pen and paper. As soon as we have determined the best solution for our problem, we can start routing real traffic on real streets with real cars.

Now, obviously, the junctions these streets end at have an *excess flow* (blue), that we have to send somewhere



such that the flow conservation rule is not violated. In order to get rid of this congestion, we just push cars along some street leaving this junction (again red). However, we always have to comply with the capacity rule and must not push forth more cars than there are lanes. Inevitably, this causes a new congestion at the next junction (blue). But when the cars finally arrive at Z , we need not push them any further (but simply put them in the parking lot).

We have seen that we cannot always push forth as many cars as we would like to. The crucial point is: We always push forth as many cars as possible but never more cars than there are lanes and, of course, at most as many as are stuck at the junction. And, by any means, we have to obey one-way streets. In case we are stuck altogether – because more cars arrive at a junction than can possibly depart by all the streets leaving it – we have to be allowed to “push back” cars (like in the junction at the upper left in the following picture).



By doing so we reduce the number of cars that use a street. In fact it is not allowed to go backwards on one-way streets, but remember that we are only simulating here. Needless to say, we only push back as many cars as necessary in order to get rid of the excess flow at the junction (after that the junction is grey). And naturally we cannot push back more cars than have arrived in the first place.

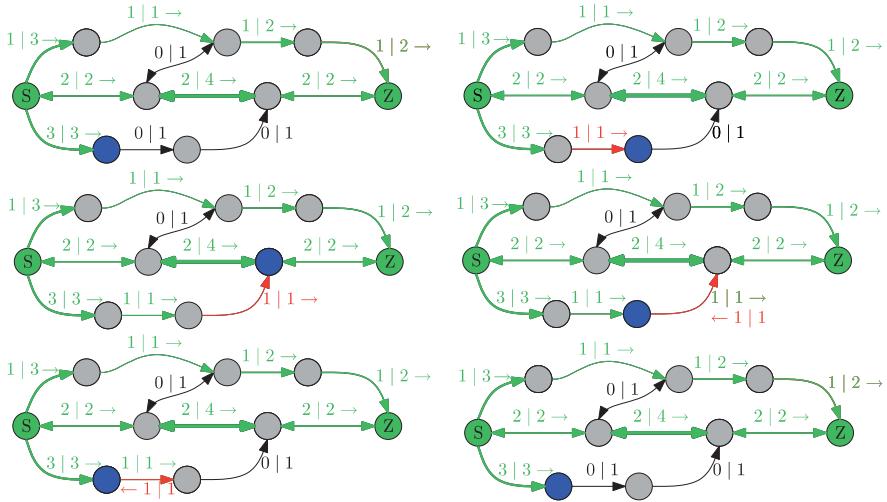
To sum up: In every step we select a junction with excess flow (i.e., a junction where more cars arrive than depart) and push forth as big a portion of those excess cars as possible. This results in the following procedure:

The procedure **PUSH** pushes cars from a junction with excess flow (either forward or backward).

```

1 procedure PUSH ( $C$ )
2 precondition  $C$  is a junction with excess flow
3 begin
4   choose one of the following:
5     Select among the streets leaving  $C$  one with free lanes and push
        forth as many cars as possible on it.
6   or
7     Select among the streets leading to  $C$  one with cars driving on it
        and push back as many of the excess cars as possible.
8   end of choice
9 end
```

Unfortunately, this procedure does not yet lead to success. The reason for this is that it can easily happen that two junctions (or more than two) simply push their excess flow back and forth forever and we never come to an end, as happens with the three junctions in the following six pictures. Computer scientists say: “The algorithm doesn’t *terminate*.”



Thus, we need a good idea for making our search for the best flow more goal-directed. Let's introduce the following additional rule: Each junction is assigned a *height*. At the start all junctions have height 0. Later on we gradually raise the junctions in the following fashion: We stipulate that cars may only be pushed *downwards*. Hence, in order to push forth an excess flow from a junction we first need to raise it to, say, height 1. Then we are allowed to push (forth or back) excess cars only on streets leading to lower junctions. In the beginning we raise S to height 1 and push, like before, as many cars as possible away from S . Afterwards, we raise the next junction with excess flow to height 1, push forth, then raise the next junction, and so on. We never have to raise the goal Z because once the cars have arrived there, they have reached their final destination.

Usually, many cars arrive at Z in this fashion. Nevertheless we might end up with junctions having excess flow but no neighboring junctions at height 0 to get rid of their excess flow. Then we are allowed to raise them even higher. How high? Well, at least by 1. Chances are that this does not yield a new possibility for pushing downwards. In that case we may continue raising the junction. But only far enough to arrive at a height that allows pushing forth (or backwards). Naturally, we raise the junction only in our model. Once we have found our final solution, there is no need to call the construction workers with their diggers to raise real junctions.

The procedure RAISE raises a junction C if it has excess flow but can push it neither forward nor backward.

```

1 procedure RAISE ( $C$ )
2 precondition Pushing from  $C$  not possible.
3 begin
4     Raise  $C$  until there is an opportunity to push flow to a lower junction.
5 end
```

We amend the procedure PUSH with the prerequisite that excess flow must only be pushed downwards.

This PUSH procedure has – on top of the previous version – the restriction that flow may only be pushed downwards.

```

1 procedure PUSH ( $C$ )
2 begin
3     choose one of the following:
4         Select among the streets leaving  $C$  one that leads to, say,  $N$  and
            that is not full yet. If  $C$  is higher than  $N$ , push excess cars along
            this street; but neither more than fit on the street nor more than
            there is excess.
5     or
6         Select among the streets leading from, say,  $N$  to  $C$  one with cars
            driving on it. If  $C$  is higher than  $N$ , then push back as many of
            the excess cars as possible; but never more than  $C$  has excess.
7     end of choice
        [sometimes neither is possible]
8 end
```

We can now repeat these two procedures (PUSH and RAISE) until there is no junction with excess flow left. We can stop raising S as soon as it reaches height n , where n denotes the number of junctions in our network. After that we merely deal with the excess flow at the remaining junctions, and then we are done. The reason why stopping then is okay will be explained in the section “Why does it work?” below. Note that we could also simply raise S to height n directly at the beginning. In our example S has height 9.

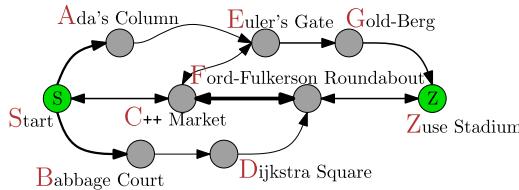
The actual algorithm thus looks like this:

The algorithm MAXIMUM FLOW finds a maximum flow from the start S to the target Z , by repeatedly calling the procedures RAISE and PUSH.

```

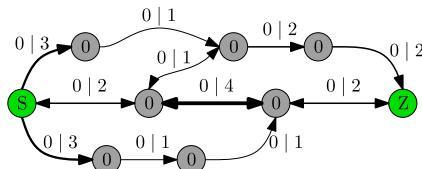
1  procedure MAXIMUM FLOW ( $G, S, Z$ )
2  begin
3      Raise  $S$  to height  $n$  ( $n$  is the number of junctions).
4      For each street leaving  $S$ , push as many cars as possible away from  $S$ .
5      Leave all other junctions (except  $S$ ) at height 0.
6      while there is a junction  $C$  with excess flow do
7          if pushing from  $C$  possible
8              Apply procedure PUSH (at junction  $C$ ).
9          else
10             Apply procedure RAISE (at junction  $C$ ).
11         endwhile
12     end
```

In order to describe a complete run of our algorithm, we first have to give names to the junctions:

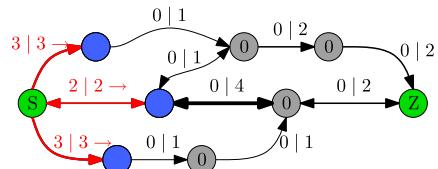


The following pictures show the progress of the algorithm. We annotated each junction with its current height.

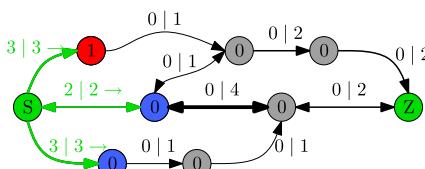
Initial state:



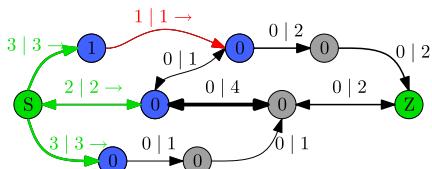
0. Push all cars away from the start:



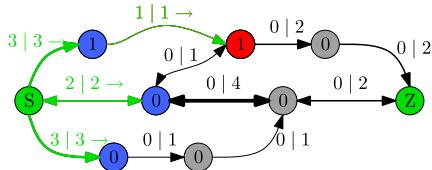
1. RAISE (Ada's Column) to 1:



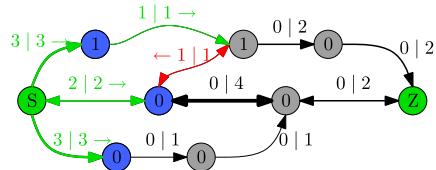
2. PUSH (Ada's Column):
one car to Euler's Gate:



3. RAISE (Euler's Gate) to 1

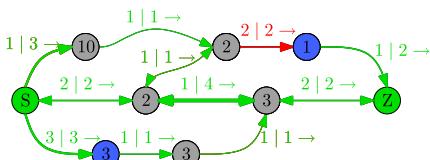


4. PUSH (E): 1 to C

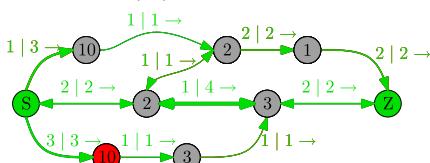


We list some intermediate steps in short only:

5. RAISE (C) to 1
7. RAISE (F) to 1
9. RAISE (F) to 2
11. RAISE (C) to 2
13. PUSH (E): 1 to G
15. PUSH (G): 1 to Z
17. PUSH (B): 1 to D
19. PUSH (D): 1 to B
21. PUSH (B): 1 to D
23. PUSH (D): 1 to F
25. PUSH (A): 2 to S
27. PUSH (F): 1 to C
29. RAISE (E) to 2
30. PUSH (E): 1 to G

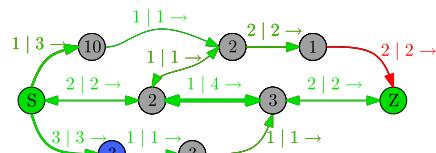


32. RAISE (B) to 10

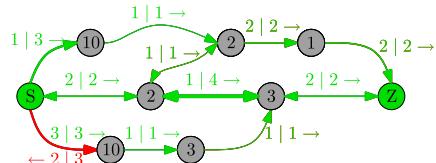


6. PUSH (C): 3 to F
8. PUSH (F): 2 to Z
10. PUSH (F): 1 to C
12. PUSH (C): 1 to E
14. RAISE (G) to 1
16. RAISE (B) to 1
18. RAISE (D) to 2
20. RAISE (B) to 3
22. RAISE (D) to 3
24. RAISE (A) to 10
26. RAISE (F) to 3
28. PUSH (C): 1 to E

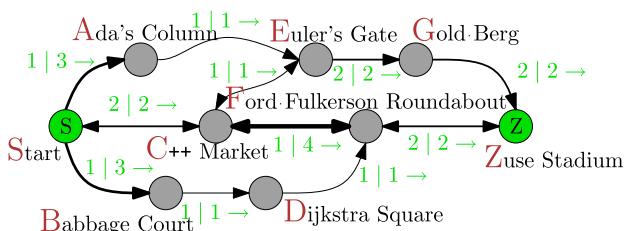
31. PUSH (G): 1 to Z



33. PUSH (B): 2 to S



This is the solution to our problem:



We could now use this solution to route traffic in the real world, avoiding congestions.

Some Open Questions

- The algorithm does not specify which junction to choose each time step 6 is called. Any choice works as long as we obey the rules for pushing (“always downwards”) and raising (“only if no more pushing is possible and then only as high as necessary to push again”)! Our example needs 33 steps: 15 times RAISE and 18 times PUSH (excluding the initial raising of and pushing from the start). Try coming up with a different sequence of operations for our example. Can you find one that reaches a solution more quickly?
- Have you realized that excess flow may only be pushed back to S , after a junction has been raised higher than n ?

Why Does It Work?

As if by magic, the algorithm always works correctly. If you are in the mood, try playing around with different road networks. But if you are interested in why the algorithm works, continue reading:

First of all, we should convince ourselves that the algorithm outputs a valid traffic flow. To this end we must merely realize that:

- We have never pushed more cars along a road than fit on it (capacity rule), and
- no junction ends up with excess flow (flow conservation rule).

Thus the traffic can flow unhindered.

Having tried a few examples, you will notice that there is a crucial difference between junctions which are higher than n , and those that are not. From high junctions traffic is always pushed back towards S . These are the junctions having no chance any more to pass their excess flow on towards Z . But what happens before this?

In the beginning excess flow is only pushed from junctions at height 1 to junctions at height 0. In a way these are the simple cases. Only after all simple options are exhausted, are junctions gradually raised higher and higher. An important observation is that excess flow is always only pushed down one level, i.e., from junction C at height h to a neighboring junction D at height $h - 1$. It can never happen that this neighboring junction D has, e.g., height $h - 2$. After all, we would then not have been allowed to raise C that high in the first place. Cars that are supposed to arrive at Z must gradually descend from h to $h - 1$, then from $h - 1$ to $h - 2$ and so on, until they arrive at Z , which always remains at height 0. Therefore at least h different stages are visited.

This ensures that excess flow cannot be pushed back and forth perpetually between two junctions. This is due to the fact that there can be at most $n - 1$ different stages (not n because we never pass through S). Moreover, this guarantees that really all options to push forth excess flow are exhausted, before pushing back flow to S . If by no means excess flow can be routed to Z , backward routing toward S works by the same principle. Ultimately all excess flow ends up at either Z or S , and we have finally found the best traffic flow.

Epilogue

Some time later Jogi's sister learned how to meticulously prove that this algorithm finds the best traffic flow. In fact, this is quite involved. It turned out, that the order in which junctions are selected for PUSH and RAISE operations does not affect the correctness of the algorithm. She also learned that Andrew Goldberg and Robert Tarjan found this algorithm in 1988. Jogi often still gets stuck in traffic jams on his way to the stadium.

Solution

There is actually a sequence of operations that requires only 19 steps to find the best traffic flow:

- | | |
|----------------------------|---------------------------|
| 1. RAISE (A) to 1 | 2. PUSH (A): 1 to E |
| 3. RAISE (C) to 1 | 4. PUSH (C): 1 to E |
| 5. PUSH (C): 1 to F | 6. RAISE (B) to 1 |
| 7. PUSH (B): 1 to D | 8. RAISE (D) to 1 |
| 9. PUSH (D): 1 to F | 10. RAISE (F) to 1 |
| 11. PUSH (F): 2 to Z | 12. RAISE (E) to 1 |
| 13. PUSH (E): 2 to G | 14. RAISE (G) to 1 |
| 15. PUSH (G): 2 to Z | 16. RAISE (A) to 10 |
| 17. PUSH (A): 2 to S | 18. RAISE (B) to 10 |
| 19. PUSH (B): 2 to S | |

Further Reading

1. Chapter 7 (Depth-First Search)

Many flow algorithms are based on a so-called depth-first search or breadth-first search in the graph. This is also true for the algorithm of Ford–Fulkerson, for instance. In this chapter you can read up on how a depth-first search in a graph works, and how it can be used.

2. Chapter 9 (Cycles in Graphs)

In rare cases, the algorithm of Goldberg and Tarjan sends a few units of flow around in a circle somewhere in the graph that does not contribute to the actual flow from the start to the target. By means of a cycle search, these circular paths can be removed after the maximum flow has been determined. This chapter explains how this can be done.

3. Chapter 32 (Shortest Paths)

A problem closely related to maximum flows is the search for a shortest path. Here the goal is not to route a whole flow of cars from the start to the target but to find the quickest route for a single car. You can read up on how these shortest paths can be found in this chapter.

4. The 3D-animation *Flow Commander* at

<http://i11www.iti.uni-karlsruhe.de/adw/jaws/GTVisualizer3D.jnlp> (requires Java Web Start).

Why not look at height as an actual third dimension? Fly through the graph and behold PUSH and RAISE operations taking place in 3D! If Java is installed on your computer (which is most likely the case), you can install and launch *Flow Commander* at this URL.

5. One of the quickest algorithms for maximum flows: Lester R. Ford, Jr. and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.6. The original publication of the presented algorithm: Andrew V. Goldberg and Robert E. Tarjan: *A new approach to the maximum-flow problem*. Journal of the ACM 35:921–940, 1988.

<http://dx.doi.org/10.1145/48014.61051>

7. The English Wikipedia article on the algorithm of Goldberg and Tarjan:
http://en.wikipedia.org/wiki/Push-relabel_algorithm

8. The Wikipedia article on network flow:

http://en.wikipedia.org/wiki/Flow_network

Among other things this article explains how flows are interrelated to so-called *cuts*.

Marriage Broker

Volker Claus, Volker Diekert, and Holger Petersen

Universität Stuttgart, Stuttgart, Germany

35.1 Problem

A marriage broker attempts to create the greatest possible number of couples from a given set of men and women. A necessary condition is that the two persons who form a couple are mutually friendly. We also assume the classic marriage: Each pair consists of exactly one man and one woman, and each person appears in the set of couples not more than once.

Put somewhat more abstractly, there is a set \mathcal{H} of men, a set \mathcal{D} of women and a set \mathcal{L} of “friendly couples” of the form HD . An entry of the form HD means that the man H and the woman D are mutually friendly. The goal is to form the largest number of couples from \mathcal{L} , with no person a member of more than one couple. Figure 35.1 shows a set $\mathcal{H} = \{A, B, C, D, E\}$ of 5 men and a set $\mathcal{D} = \{P, Q, R, S, T\}$ of 5 women. If a man and a woman are mutually friendly, we connect them with a line (a so-called edge). As an example, we look at a set \mathcal{L} with the sympathy-relations $AP, AR, BP, BQ, BS, CQ, CR, CT, DR, DS, ES$ and ET . This is graphically shown in Fig. 35.2.

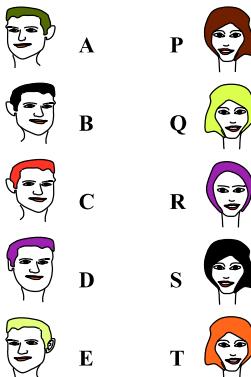


Fig. 35.1. Five women and five men

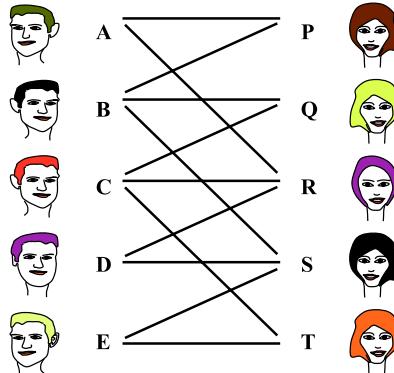


Fig. 35.2. The sympathy-relations between men and women

Now the broker could choose the following couples: BP , CR , ES . However then the broker has to stop, because he cannot put together another suitable pair. But the broker may also have formed couples AP , BQ , CR , DS and ET , in which case all persons would have a partner. Now the problem is: How does one find a largest possible subset M of \mathcal{L} , in which every person from \mathcal{H} and \mathcal{D} appears not more than once? Such a subset M is called a *maximum matching* for \mathcal{H} , \mathcal{D} and \mathcal{L} .

First we want to introduce the idea and after that the algorithm which helps to construct such a maximum matching. It is unimportant whether the algorithm is executed by an employee of the broker (using index cards and other tools) or by request of the affected persons between themselves.

35.2 The Basic Principle of the Procedure

Initially, one can put people together into couples if they are friendly but not already assigned to another couple. Once this is no longer possible, there can be a partnerless person (here we take a partnerless man H) who asks all the women who are friendly to him whether they can leave their partner.

These women are assigned to exactly one man, and therefore they hand over the question to their partner. In the next step all these men ask all the women who are friendly to them whether these women can leave their partner. Now these women ask their assigned men whether they could find a new partner, etc. A wave of inquiries runs from the partnerless man H to women who are friendly to him, and from these to their respective assigned men and even spreads further to the set of all persons until one of the following conditions is met:

- (i) At one point a man asks a previously partnerless woman. In this case the whole wave of requests stops immediately. From this new partnership one

follows the chain of requests backwards toward the man H and exchanges the previous partnerships along exactly this way.

- (ii) One finds out that the wave of requests reaches only women who already have partners. In this case one deletes the man H from the set of men.

Subsequently one can adopt this picture of a wave of requests, spreading from H toward all persons, alternately going through a friendly relationship and an already taken assignment. Once a new partnership is discovered, this wave breaks abruptly, and the couples are rearranged on the path of requests from this new partnership to H .

Hint: This wave can be processed simultaneously or in any sequence, but the most vivid picture is the parallel uniformly spreading wave (in Computer Science we talk about breadth-first search regarding \mathcal{L}). Here no person is asked who has already been questioned once, so that the wave only expands to persons who have not yet been considered.

35.3 The Construction of a Maximum Matching

One starts with any pair from the set \mathcal{L} . Let this pair be HD and put $M = \{HD\}$. (If the set \mathcal{L} is empty, there are, of course, no couples to form and there is nothing to do.) Note that in the set M below each person appears not more than once.

Now we can assume that a set of couples $M = \{H_1D_1, H_2D_2, \dots, H_rD_r\}$ has been constructed. If all men occur in M one is finished, because bigamy is illegal. The interesting situation is when a man H is without a female partner, thus he does not occur in M . How can we assign to him a female partner D ?

(a) Obviously one has to ask every person D about person H who has sympathies for D (i.e., HD is included in the set \mathcal{L}), whether she still has no partner. If there exists such a person D without a partner, then one just adds HD to the set M and applies the algorithm again to another partnerless person. We have performed this by starting from the pair BP by adding pairs CR and ES in Fig. 35.3. The set $M = \{BP, CR, ES\}$ is given by the red edges. Case (a) is not true any more and we cannot enlarge M in this way, although there are still partnerless persons left.

(b) What has to be done if there is no partnerless person D matching the partnerless person H in \mathcal{L} ? Then there is a partner H' assigned to each person D , who is found friendly by H . Now one tries to take away woman D from man H' in favor of man H , which means one tries to replace $H'D$ by HD , consequently H' becomes partnerless.

Now one tries to take away the partner H'' from one of the women D' (but not the woman D), who is friendly to H' , making H'' partnerless and requiring another partner, etc. Only such women $D', D'', D''',$ etc., are being considered who have not yet been included. Based on H a sequence of alternating friendship relations HD and already allocated $H'D$ relations from the

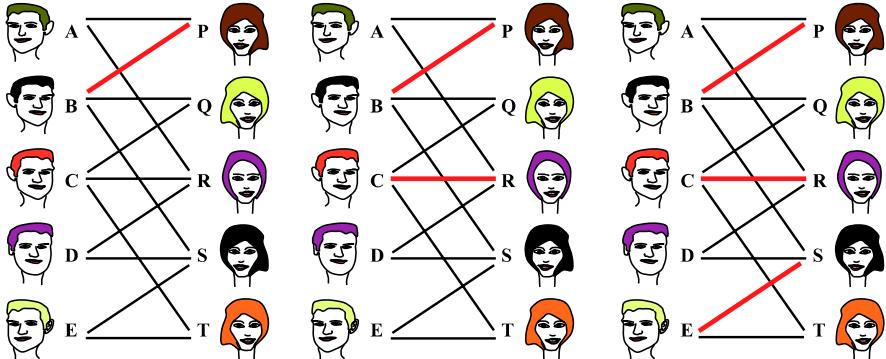


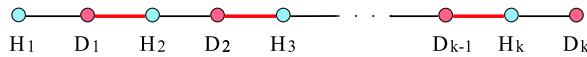
Fig. 35.3. Assembly of the set M only with criterion (a) of the algorithm

set M is passed through. At one point either case (a) occurs, or the iteration stops when all persons reachable in this way have been tried unsuccessfully.

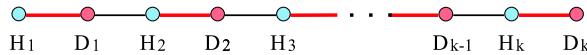
If case (a) occurs, then one has obtained a new set of couples by actually making the replacements “ $H'D$ was replaced by HD ,” which leads to the case (a), while all other couples in M remain unchanged. Expressed more precisely: If case (a) occurs the first time, then there is exactly one sequence (starting with $H = H_1$)

$$H_1, D_1, H_2, D_2, \dots, H_k, D_k$$

with H_1 being partnerless, $H_1D_1 \in \mathcal{L}$, $H_2D_1 \in M$, $H_2D_2 \in \mathcal{L}, \dots, H_kD_{k-1} \in M$, $H_kD_k \in \mathcal{L}$ and D_k is partnerless. The situation with the partnerless woman D_k



allows the replacement of the red edges in M by the black edges (“re-coloring” of red and black edges on this path):



where M grows by one pair and all persons who already had a partner before now have a partner once again. With this increased set M one starts the algorithm again.

In our example, this approach leads to success (see Figs. 35.4–35.8): Previously, the couples BP , CR and ES were selected. More couples are no longer possible. So now couples have to be regrouped. For this we assume in Fig. 35.4 the man A , who still has no partner. We assign to him one of his potential partners P or R , which now makes man B (former partner of P) or C (former partner of R) partnerless; for each of these again an attempt is made to find new female partners (Fig. 35.5), etc. Each person is not treated more than

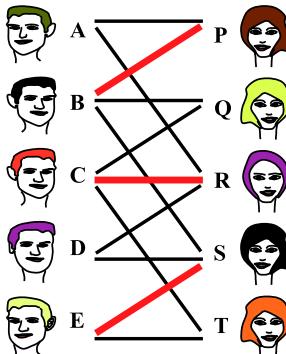


Fig. 35.4. A is partnerless. A could be paired with P or with R . We start with P . Her partner is B . Experimentally we now replace BP by AP , indicated by dashed edges

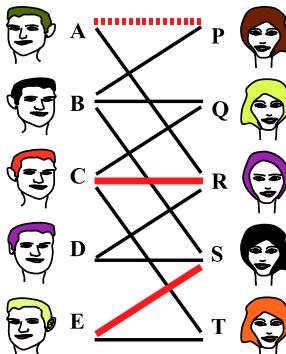


Fig. 35.5. Now B is partnerless. The case (a) occurs already, because the couple BQ with the partnerless woman lies in \mathcal{L}

once, i.e., if during the process a person is met who has already been investigated, he or she is not considered again. If case (a) occurs, one gets a new set M , as shown in Fig. 35.8.

(c) If this method is not successful by finding case (a), a mathematical theorem states that the person H may be excluded from the further procedure.

In principle, this situation is quite easy to understand. Because what happens if the set M does not get enlarged? Then there is a set of d women, who have been totally questioned starting from H . Each of these d women also has a partner after the rearrangement, so there are d men with a partner. Only the last of the men who lost his partner, now has none. So there are $d+1$ men who occurred in this row. And here comes the crucial observation: The

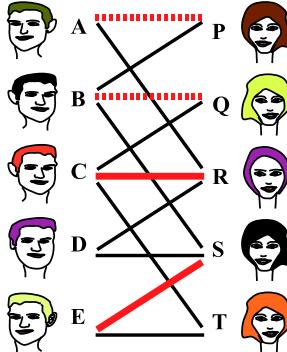


Fig. 35.6. The “experimental” repairing will be definitively adopted, and we get the assignment $\{AP, BQ, CR, ES\}$, where BP has been replaced by AP and BQ

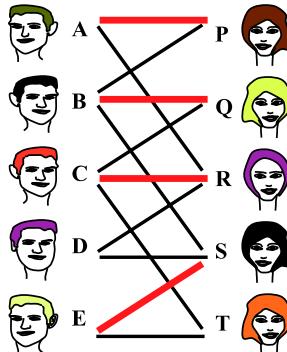


Fig. 35.7. Now we start the process again with the partnerless person D (note: D is the name of the man in this example). Experimentally we match DR and therefore remove CR . The newly partnerless person C may be matched to a pair with the person T

totality of the possible female partners of these $d + 1$ men is exactly the set of these d women. Consequently, one man has to remain partnerless. Thus we can remove him from the beginning. This argumentation is explained again in Sect. 35.6.

We maintain: If M is a maximum assignment, there is also a maximum assignment M' , which has as many elements as M , but does not contain H . Therefore one can abandon H . Therefore we delete H and start the algorithm again with another partnerless person. (By the way, instead of H we could have also deleted one of the last men from an experimental rearrangement. This is rarely done in practice. First the important persons from the broker's point of view receive a partner. The approach described here guarantees that once-arranged persons are not partnerless in the end.)

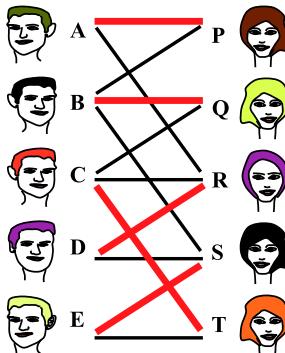


Fig. 35.8. Finally we have obtained the set $M = \{AP, BQ, CT, DR, ES\}$. Now, every person has a partner and therefore the algorithm ends. (The experimental rearranging takes up as many steps as there are edges left, because already tested persons will not be tested again)

(d) The procedure ends when every person not deleted has a partner, that means it is included in the set M . Now this set M is the maximum assignment.

So how is the gradual increase of the number of couples in the assignment set M (see Figs. 35.4—35.8)? The red edges belong to the couples of the set M . The black edges show the remaining friendly relations ($\mathcal{L} - M$). Now the set M is reordered every time. We look at a path of alternating black and red edges, starting and ending with a black edge and which has a partnerless person at the beginning and at the end. If this happens, we can replace in M the red edges by the black edges of exactly this path. In this way the number of couples is increased by 1.

Hint: Such paths of alternating black and red edges, whose ending nodes do not have outgoing red edges, are called *augmenting paths*.

35.4 The Algorithm

The following MARRIAGE BROKER algorithm provides a maximum set M of couples, if the sets \mathcal{H} , \mathcal{D} , and \mathcal{L} are given. It is clear that one just has to look at one of the two sets \mathcal{H} and \mathcal{D} in order to choose the next partnerless candidate. We restrict ourselves to the set \mathcal{H} (see line 2 of the algorithm); in practice one takes the smaller of the two sets \mathcal{H} and \mathcal{D} .

Given: The sets \mathcal{H} and \mathcal{D} , and the set \mathcal{L} as the set consisting of only one couple HD for some H from \mathcal{H} and D from \mathcal{D} .

The algorithm MARRIAGE BROKER calculates a maximum set of couples M

```

1  choose a couple  $HD$  from  $\mathcal{L}$ ;  $M := \{HD\}$ ;
2  while there is still a partnerless person  $H$  in  $\mathcal{H}$  do
3      follow all paths starting from  $H$ , which consist alternately
4          of one edge from  $\mathcal{L}$ , which is not in  $M$ ,
5          and one edge from  $M$  and
6          contains no person more than once;
7      if one finds a partnerless person
8          (which necessarily is in  $\mathcal{D}$ )
9          then replace all edges in  $M$  on this path in  $M$ 
10         by edges which are not in  $M$  on this path;
11      else (in this case there is no such path)
12          remove  $H$  from  $\mathcal{H}$ 
13      end if
14  end while;
15  return  $M$ ;
```

Inside the while-loop one has to realize a systematic search in the part “follow all paths starting from H, \dots .” This part is implemented “recursively.” We have already written in Sect. 35.3, part (b): “Now one tries to take away woman D from man $H' \dots$. This is exactly the recursion that has to do the same procedure with H' instead of H , if H' hasn’t been treated already. One should maintain in a boolean array the components which are set to *false* at the beginning of the while loop (before row 3 in the program) and in which one records if a person has been considered in this iteration or not. How the overall procedure is implemented can be found in books. (At the end of this chapter is a list of references.)

35.5 The Marriage Theorem

That the procedure works correctly is based on the *Marriage Theorem* of the English mathematician Philip Hall in 1935. It follows from this theorem that there exists an assignment of all men to appropriate women if and only if the following marriage condition holds: For each subset of men, there is an at least equally large subset of potential female partners.

The condition means that if, e.g., we look at 17 men from the set \mathcal{H} , then there are at least 17 potential female partners available for them. Instead of 17 we may also use any other number. Initially just the right relation of the numbers is important. The assignment is insignificant at this time.

The criterion of the Marriage Theorem is not directly suitable for finding a solution. First, an inspection for each 50 men and women would need more than one million years, even if one billion combinations of men per second could be examined (because you have to test $2^{50} = 1,125,899,906,842,624$ subsets). Second, the theorem only provides information whether a solution

exists, but not what it looks like. For this we need the algorithm described above.

35.6 Where Is the Marriage Theorem Needed by the Algorithm

The Marriage Theorem is needed by the algorithm at the point where one has to find at least another man whose partner gets newly assigned and who has the chance to find a different woman for himself. We explain this with the help of an example:

Consider Fig. 35.9. Let $M = \{H_2D_1, H_4D_2, H_3D_3\}$. H is partnerless. We start with H .

$\{H\}$ is a subset T_1 . To proceed, the subset $\{D_1, D_2\} = T_2$ (= all with H connected persons) belonging to T_1 has to be at least as large as $|T_1| = 1$ (Marriage Theorem!). See Fig. 35.10.

Now we replace H_2D_1 on a trial basis by HD_1 , making H_2 partnerless. We could also replace H_4D_2 by HD_1 , making H_4 partnerless. Overall, we have to find partners for $T_3 = \{H, H_2, H_4\}$. The set of women belonging to T_3 is $T_4 = \{D_1, D_2, D_3, D_6\}$, see Fig. 35.11. Again $|T_3| \leq |T_4|$ holds, so following the Marriage Theorem there has to exist an assignment here. We detect this by further examining the men belonging to women from T_4 , etc.

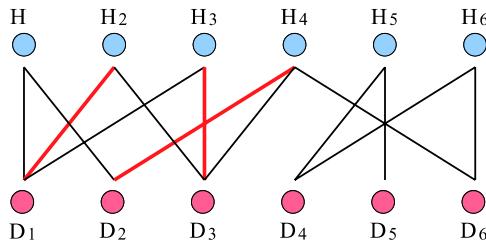


Fig. 35.9. $M = \{H_2D_1, H_4D_2, H_3D_3\}$, H is partnerless

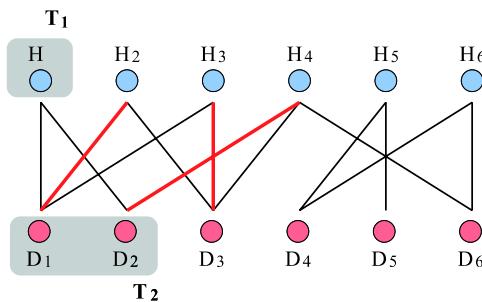


Fig. 35.10. $|\{H\}| \leq |\{D_1, D_2\}|$

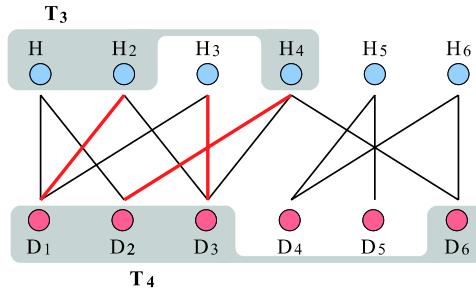


Fig. 35.11. $|\{H, H_2, H_4\}| \leq |\{D_1, D_2, D_3, D_6\}|$

If a solution exists, the Marriage Theorem ensures that the algorithm can always proceed at any point with a person who has not yet been considered.

35.7 Time Analysis

The running-time of the procedure can be estimated easily from above. We assume that there are n men and n women as well as m edges (= number of elements in the set \mathcal{L}). The while-loop in the algorithm is finished after at most m steps, because each person is taken into account not more than once and therefore each edge has to be examined only once. Because after every loop either one man is eliminated or another couple is added, the whole process ends after the while-loop has been executed $(n - 1)$ times. For the duration estimation we receive $n \cdot m$ as upper bound for the number of steps taken by the execution of the algorithm.

Are there faster ways to calculate a maximum set of couples? One piece of evidence of wasted time in our algorithm is that at the beginning of a pass of the loop no information is available, although earlier passes had collected information about the course of augmenting paths. Making clever use of this, one can achieve a running-time proportional to $m \cdot \sqrt{n}$, saving the factor \sqrt{n} . This accelerated procedure was developed in 1971 by the American researchers John E. Hopcroft and Richard M. Karp. For these and many other achievements these two researchers received the Turing Award in 1986 and 1985, respectively. [This is a kind of Nobel Prize in Computer Science.]

Further Reading

1. In Chap. 34 of this book the problem of a maximum flow is examined. A solution of this problem can be used to determine a maximum set of couples. Furthermore, the procedure that was presented here can be generalized to arbitrary graphs. (Think for yourself or refer to one of the following books.)

2. Thomas Ottmann, Peter Widmayer: *Algorithmen und Datenstrukturen*. Spektrum Akademischer Verlag, 2002. A standard textbook written in German. In the chapter *Zuordnungsprobleme* maximum assignments are found with the help of the calculation of maximum flows.
3. Reinhard Diestel: *Graph Theory*, 3rd edition. Springer, 2006. Chapter 1 deals in great detail with pairings in general and bipartite graphs. There you can also find a proof of the Marriage Theorem.
4. Dexter C. Kozen: *The Design and Analysis of Algorithms*. Springer, 1992. This book contains 40 chapters on topics of the theory of algorithms, each of which corresponds to a lecture. In Chaps. 19 and 20 the more efficient method of Hopcroft and Karp with the corresponding correctness proofs is presented.
5. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2001. Section 26.3: Maximum bipartite matching, pp. 664–669. This is a very detailed and very successful book on the theory of algorithms.
6. Several articles about the *Matching Problem* and the *Marriage Theorem* can be found in the free encyclopedia wikipedia, www.wikipedia.org.

Acknowledgements

We thank Botond Draskoczy and Sascha Riexinger for technical support.

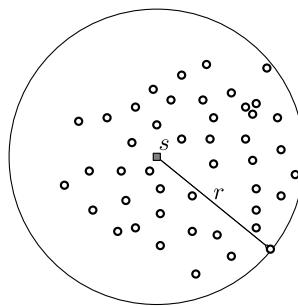
The Smallest Enclosing Circle – A Contribution to Democracy from Switzerland?

Emo Welzl

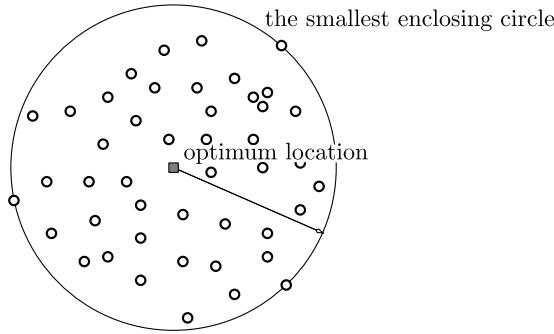
ETH Zürich, Zürich, Switzerland

The fire service needs a new station and it should be located optimally with respect to the houses served by it. The quality of the location is measured by the maximum of the distances to the relevant houses, and this greatest distance should obviously be as short as possible. We idealize the houses and the new location as points in the plane and model the real distances by distances between the points. So the input for our problem is a set P of points in the plane.

Let us choose a point s as a potential location. The distance from s to the farthest point in P will be denoted by r . Then a circle of radius r centered at s encloses all the points of P .



It is clear that the best location is the center of a circle enclosing all the points of P that has the smallest radius. (From there, the firemen can reach even the farthest house as fast as possible.) Such a circle exists and is unique – we take this as a fact.



There are many houses and people are thinking how to decide on the best location.

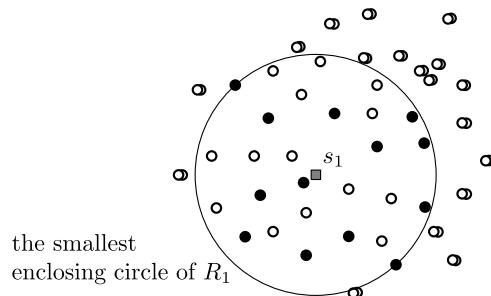
Somebody got the idea to invite representatives of a small random sample R_1 of, say, 13 houses and to let them determine the best location for themselves – without any regard for the other houses. A method can be found in the literature, which indeed solves the problem quickly for 13 houses, but unfortunately turns out to be too slow for all the houses.

So they come up with a proposal for a place s_1 and a radius r_1 , so that the circle with the center s_1 and radius r_1 encloses all the chosen houses – it is the smallest enclosing circle of R_1 .

The first location is found.

Even though the set R_1 is chosen at random, people strongly oppose this choice of location, especially the occupants of the houses standing outside the determined circle.

To accede to the protests, a second sortition is called, but, as before, nobody knows how to find the smallest enclosing circle for such a large set – it was already hard enough for 13. The proposal to invite all who stay outside the first circle appears unrealistic, too. It comes down to the following compromise: all houses standing outside the ominous circle have two entries in the sortition of 13 new representatives.



The 13 representatives, a set R_2 , are chosen, and they meet and agree on their most preferred location s_2 and the corresponding radius r_2 of the smallest circle enclosing R_2 .

Less surprisingly, even this encounters resistance. Again, there are many houses standing outside the circle determined as a solution for R_2 .

Now, we should take into account that the municipality has not yet secured the sources for financing of the new fire-service building. Therefore they like the decision-making process – they are actually glad when this procedure does not lead quickly, if at all, to a conclusion satisfactory for everyone.

So the result of the second round is rejected as well. For the next round, the number of entries is doubled for every house standing outside the second circle. If there is a house out of the circle for both the solutions determined so far, then it gets in return four tickets in the polling urn!

Round three proceeds as before.

Et cetera, et cetera.

It becomes a routine. The circle-finding emerges as a popular entertainment, not least because the municipality provides food and drinks. It is no longer disappointing to be out of the announced circle, as the proposal would not be realized anyway and the chance of participating in the next meeting grows. The polling urn swells up, but the municipal secretary has soon arranged for an electronic sortition (encouraged by Chap. 25 on random numbers).

But then, after 13 representatives have met again and proposed a solution, which is the best for themselves, and themselves only, something unexpected happens. No house is outside the calculated circle. The information spreads quickly and a speedily obtained expertise confirms what all have guessed: This must be the smallest circle enclosing *all points*. A circle enclosing all points cannot be smaller than a circle enclosing only 13 points after all.

The optimum location is found!

Have we been lucky that the chosen representatives were so successful, or should we have been expecting it? The latter: We have learned a randomized (i.e., based on randomness) algorithm developed by Kenneth Clarkson. It calculates the smallest enclosing circle for n points. It can be shown that the procedure computes the circle with probability 1 and the expected number of rounds is in fact only logarithmic in n . It is necessary not to set the sizes of the randomly chosen subsets too small (in our story 13) – though 13 is enough, independent of how big n is. In the same way, it is also possible to calculate the smallest enclosing ball of points in 3-dimensional space or even in higher dimensions (only the random sample must be somewhat bigger depending on the dimension).

Why It Works

For those keen enough we should reveal, omitting many details, why it really works. We need therefore to understand the structure of the problem somewhat better. The smallest circle enclosing a point-set P – it's high time to give it a name: $K(P)$ – is determined by at most three points of P . To put it more precisely, there is a set B of at most three points of P for which we have $K(B) = K(P)$. Next we need to observe: If there is a subset R of P such that the circles $K(R)$ and $K(P)$ are not yet equal, then there must be a point of B lying outside $K(R)$. For our procedure it means that at least one point of B doubles its number of entries in the sortition poll in each round. Consequently, after k rounds, at least one of the points has at least $2^{k/3} \approx 1.26^k$ entries in the poll. It grows quite nicely.

On the other hand, we can show that, thanks to the random choice, there are not too many new entries in the poll on average. It is because exactly the entries of the houses outside the current circle are doubled. This number is on average $3z/13$, when there are z entries in the poll (we will owe you the proof of this statement). It means that for the next round, we expect about $(1 + 3/13)z \approx 1.23z$ entries in the poll. (To understand it correctly: The “3” comes from the size of B and the “13” is the size of the sample as we have set it.)

On the one hand, the number of entries in a round increases approximately by a factor of 1.23, i.e., there are some $n \cdot 1.23^k$ entries after k rounds (n is the number of points). On the other hand, there is a point that has at least 1.26^k entries in the poll after k rounds. Regardless of how big n is, since $1.26 > 1.23$, sooner or later that single point will have more than all points together. There is only one way of resolving this paradox: The procedure must come to an end before this happens.

It may sound confusing. But this is what randomized algorithms are like: simple in themselves – but at the same time very intriguing that they really work.

Further Reading

1. Kenneth L. Clarkson: *A Las Vegas Algorithm for linear and integer programming when the dimension is small*. Journal of the ACM 42(2): 488–499, 1995.

This is the original work developing and analyzing the procedure we have described here.

2. Chapter 25 (Random Numbers)

Here you can learn how to generate the random numbers necessary for our procedure.

Online Algorithms – What Is It Worth to Know the Future?

Susanne Albers and Swen Schmelzer

Humboldt-Universität zu Berlin, Berlin, Germany
 Albert-Ludwigs-Universität Freiburg, Freiburg, Germany

The Ski Rental Problem

This year, after a long time, I would like to go skiing again. Unfortunately, I grew out of my old pair of skis. So I am faced with the question of whether to rent or buy skis. If I rent skis, I pay a fee of \$10 per day. On the seven days of my vacation I would pay a total of \$70. Buying a new pair of skis costs me more, namely \$140. Maybe I will continue going skiing in the future. Then buying skis could be a good option. However, if I lose interest after my first trip, it is better to pay the lower renting cost. Figure 37.1 depicts the total cost of a single purchase and the daily rental if skis are used on exactly x days.

Without knowing how often I go skiing, I cannot avoid paying a bit more. It is always easy to be clever in hindsight. However, how can I avoid saying later, “I could have gone skiing for less than half the cost”? Problems of this kind are called *online problems* in computer science. In an online problem one

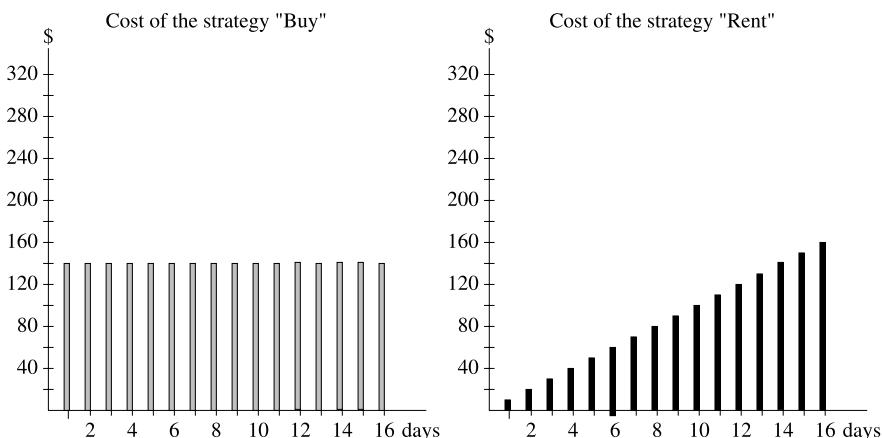


Fig. 37.1. Costs of the strategies “Buy” and “Rent”

has to make decisions without knowing the future. In our concrete case, we do not know how often we will go skiing but have to decide whether to rent or buy equipment. If we knew the future, the decision would be simple. If we go skiing for less than 14 days, renting skis incurs the smaller cost. For periods of exactly 14 days, both strategies are equally good. For longer periods of more than 14 days, buying skis yields the smaller cost. If the entire future is known in advance, we have a so-called *offline problem*. In this case we can easily determine the optimum offline cost. This cost is given by a function f , where x denotes the number of days for which we need skis.

$$f(x) = \begin{cases} 10x, & \text{if } x < 14, \\ 140, & \text{if } x \geq 14. \end{cases}$$

In contrast, in an online problem, at any time, we have to make decisions without knowing the future. If we rent skis on the first day of our trip, then on the second day we are faced with the same renting/buying decision again. The same situation occurs on the third day if we continue renting skis on the second day. If using an online strategy our total cost is always at most twice as high as the optimum offline cost (knowing the entire future), the corresponding strategy is called *2-competitive*. That is, the strategy can compete with the optimum. In general, an online strategy or an *online algorithm* is called c -competitive if we never have to pay more than c times the optimum offline cost. Here c is also referred to as the *competitive ratio*. For our ski rental problem we wish to construct a 2-competitive online strategy.

Online strategy for the ski rental problem: First, in the beginning, we rent skis. When, at some point, the next rental would result in a total renting cost equal to the buying cost, we buy a new pair of skis. In our example we would rent skis on the first 13 days and then buy on the 14th day.

In Fig. 37.2 the left chart depicts in grey the optimum offline cost (function f). In the right chart, the cost of our online algorithm is shown in black. Using the two charts we can immediately convince ourselves that the online strategy never pays more than twice the optimum offline cost. If we go skiing for less than 14 days, we pay the same amount as the optimum cost. On longer periods, we never pay more than twice the optimum cost.

We are satisfied with the solution to our concrete ski rental problem. However, what happens if the renting and buying costs take other values? Do we have to change strategy? In order to see that our strategy is 2-competitive for *arbitrary values* of renting and buying costs, we scale the costs such that a rental incurs \$1 and a purchase incurs n . Our online strategy buys a new pair of skis on the n th day when the total rental cost is exactly $n - 1$. Figure 37.3 demonstrates that, for periods of $x < n$ days, the cost of our online strategy is equal to that of the optimum offline cost. For $x \geq n$, the incurred cost is no more than twice the optimum cost. Hence we are 2-competitive.

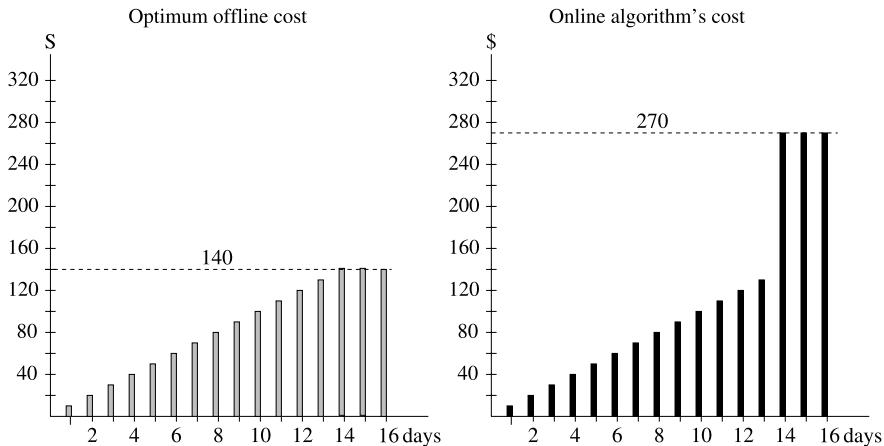


Fig. 37.2. Illustration of the competitive ratio of 2 – concrete example

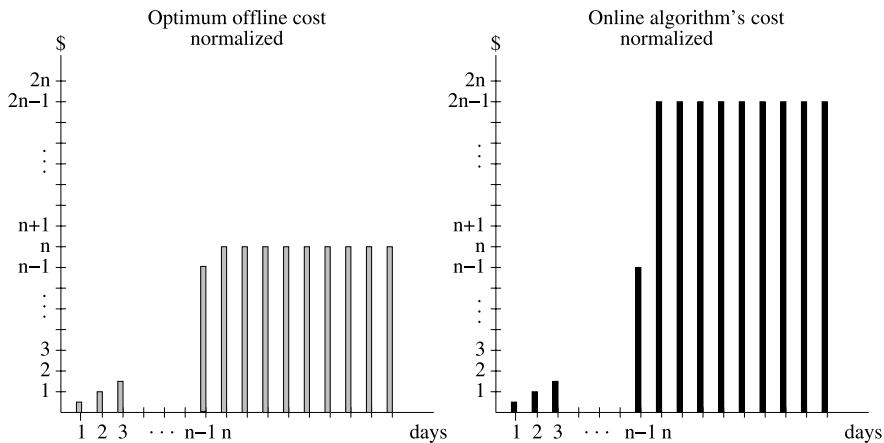


Fig. 37.3. Illustration of the competitive ratio of 2 – general case

This is an appealing result but, actually, we would like to be closer to the optimum. Unfortunately, this is impossible. If an online algorithm buys according to a different strategy, that is, at a different time, there is always at least one case where the incurred total cost is at least twice the optimum offline cost. Let us consider our concrete example. If an online strategy buys earlier, say on the 11th day, the incurred cost is equal to \$240 instead of the required \$110. If the strategy buys later, for instance, on the 17th day, the resulting cost is \$300 instead of the \$140. In both cases the incurred cost is more than twice the optimum offline cost. In general, no online algorithm can be better than 2-competitive.

The Paging Problem

In computer science an important online problem is *paging*, which permanently arises in a computer executing tasks. In the paging problem, at any time, one has to decide which memory pages should reside in the main memory of the computer and which ones reside on hard disk only; see Fig. 37.4.

The processor of a computer has very fast access to the main memory. However, this storage space has a relatively small capacity only. Much more space is available on hard disk. However, data accesses to disk take much more time. The approximate relative order of the access times is $1 : 10^6$. If an access to main memory took one second, an access to hard disk would require about 11.5 days. Therefore, computers heavily depend on algorithms loading memory pages into and evicting them from main memory so that a processor rarely has to access the hard disk. In the following example (see Fig. 37.4), memory pages A, B, C, D, E , and G currently reside in the main memory. The processor (CPU) generates a sequence of memory requests to pages D, B, A, C, D, E , and G , and is lucky to find all these pages in the main memory.

On the next request the processor is unlucky, though. The referenced page F is not available in main memory and resides on hard disk only! This event is called a *page fault*. Now the missing page must be loaded from disk into main memory. Unfortunately, the main memory is full and we have to evict a page to make room for the incoming page and satisfy the memory request. However, which page should be evicted from main memory? Here we are faced with an online problem: On a page fault a paging algorithm has to decide which page

Memory requests: D B A C D E G F

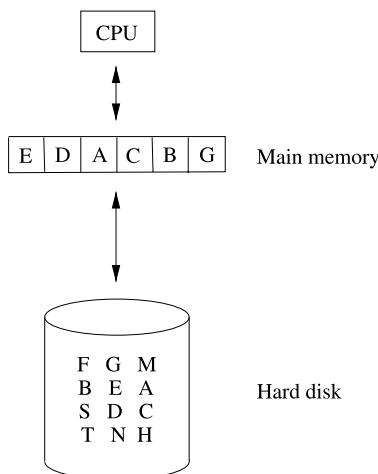


Fig. 37.4. The memory hierarchy in the paging problem

to evict without knowledge of any future requests. If the algorithm knew which pages will not be referenced for a long time, it could discard those. However, this information is not available as a processor issues requests to pages in an online fashion. The following algorithm works very well in practice.

Online strategy Least-Recently-Used (LRU): On a page fault, if the main memory is full, evict the page that has not been requested for the longest time, considering the string of past memory requests. Then load the missing page.

In our example LRU would evict page B because it has not been requested for the longest time. The intuition of LRU is that pages whose last request is long ago are not relevant at the moment and hopefully will not be needed in the near future. One can show that LRU is k -competitive, where k is the number of pages that can simultaneously reside in main memory. This is a high competitive ratio, as k takes large values in practice. On the other hand, the result implies that LRU has a provably good performance. This does not necessarily hold for other online paging algorithms.

Beside LRU, there are other well-known strategies for the paging problem:

Online strategy First-In First-Out (FIFO): On a page fault, if the main memory is full, evict the page that was loaded into main memory first. Then load the missing page.

Online strategy Most-Recently-Used (MRU): On a page fault, if the main memory is full, evict the page that was requested last. Then load the missing page.

One can prove that FIFO is also k -competitive. However, in practice, LRU is superior to FIFO. For MRU, one can easily construct a request sequence such that MRU, given a full main memory, incurs a page fault on each request. Given a full main memory, consider two additional pages A and B , not residing in main memory that are requested in turn. On the first request to A , this page is loaded into main memory after some other page has been evicted. On the following request to B , page A is evicted because it was requested last. The next request to A incurs another page fault as A was just evicted from main memory. MRU evicts page B , which was referenced last, so that the following request to B incurs yet another fault. This process goes on as long as A and B are requested. MRU incurs a page fault on each of these requests. On the other hand, an optimal offline algorithm loads A and B on their first requests into main memory by evicting two other pages. Both A and B remain in main memory while the alternating requests to them are processed. Hence an optimal offline algorithm incurs two page faults only, and MRU exhibits a very bad performance on this request sequence. In practice, cyclic request sequences are quite common as programs typically contain loops accessing a few different memory pages. Consequently, MRU should not be used in practice.

Further Reading

Online problems arise in many areas of computer science. Examples are data structuring, processor scheduling, and robotics, to name just a few.

1. Chapter 38 (Bin Packing)

A chapter studying the bin packing problem, which also represents a classical online problem.

2. <http://en.wikipedia.org/wiki/Paging>

An easily accessible article on the paging problem.

3. D.D. Sleator and R.E. Tarjan: *Amortized efficiency of list update and paging rules*. Communications of the ACM, 28:202–208, 1985.

A seminal research paper in the area of competitive analysis. The article addresses the paging problem as well as online algorithms for a basic data structures problem.

4. S. Irani and A.R. Karlin: *Online computation*. In: *Approximation Algorithms for NP-hard Problems*. PWS Publishing Company, pp. 521–564, 1995.

A survey article on online algorithms studying, among other problems, also ski rental and paging.

5. A. Borodin and Ran El-Yaniv: *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

A comprehensive textbook on online algorithms.

Bin Packing or “How Do I Get My Stuff into the Boxes?”

Joachim Gehweiler and Friedhelm Meyer auf der Heide

Universität Paderborn, Paderborn, Germany

I'm going to finish high school this summer, and I'm looking forward to beginning with my studies – in computer science, of course! Since there is no university in my small city, I'll have to move soon. For this, I'll have to place all the thousands of things from my closets and shelves into boxes. In order to make moving as inexpensive as possible, I'll try to place all these things in such a way that I'll need as few boxes as possible.



If I just take objects one after another out of my shelves and put them into one box after another, I'll waste a lot of space in the boxes: The objects have various shapes and sizes and would therefore leave many holes in the packing.

I'd surely find the optimal solution for this problem, i.e., the fewest possible boxes required, if I try out all possibilities to place the objects into the boxes. But with this many objects, this strategy would take ages to complete and furthermore create chaos in my flat.

That's why I'd like to place the objects into the boxes in the same arbitrary order I get them into my hands when taking them out of the closets and shelves. Thus, the crucial question now is: “How many more boxes am I going to use this way compared to the optimal solution?” In order to find out, I'll now analyze the problem.

The Online Problem “Moving Inexpensively”

Since I'll take the objects out of the closets and shelves consecutively, my problem is an online problem (see introduction to online algorithms, Chap. 37):

- The relevant data (the sizes of the particular objects) becomes available over time. We denote the list of these sizes, in order of their appearance, by G .
- There is no information on future data (the sizes of not-yet-considered objects).
- The number of items to process (objects to pack) is not known in advance.
- The current request has to be processed immediately (objects can't be put aside temporarily).

In reality, there are some estimates on the sizes: Since I've been living in this flat for years, I thus have a certain overview of what is in the closets and on the shelves. But let's abstract away from this and consider the problem in an idealized way. Experts generally refer to this problem as the *Bin Packing* problem.

My packing strategy can now be formalized as an online algorithm: The input consists of a sequence $G = (G_1, G_2, \dots)$ of sizes G_i of the objects to pack. The boxes are denoted by K_j , and the output is the number n of boxes used.

Algorithm NEXTFIT

```

1   Set  $n := 1$ .
2   For each  $G_i$  in  $G$  do:
3       If  $G_i$  doesn't fit into box  $K_n$ ,
4           close box  $K_n$  and
5           set  $n := n + 1$ .
6   Place  $G_i$  in box  $K_n$ .

```

With a little more effort, I could also proceed as follows: Before placing an object, I could check all boxes used so far for a sufficient space to fit it and only close the boxes when finished. This approach is an advantage when I have to open a new box for a rather big object and then have a series of very small objects to pack, which still fit into previous boxes. This second strategy formalized as an online algorithm looks as follows:

Algorithm FIRSTFIT

```

1 Set  $n := 1$ .
2 For each  $G_i$  in  $G$  do:
3   For  $j := 1, \dots, n$  do:
4     If  $G_i$  fits into box  $K_j$ ,
5       place  $G_i$  in box  $K_j$  and
6       continue with the next object (go to step 3).
7   Set  $n := n + 1$  and
8   place  $G_i$  in box  $K_n$ .

```

Analysis of the Algorithms

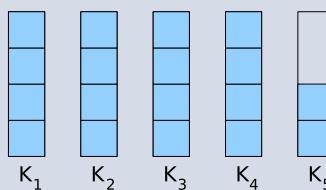
To simplify the analysis of how well or poorly my strategies perform, I make the following assumption: An object fits into a box if and only if the volume of the object to place does not exceed the volume left over in the box, i.e., I neglect any space in the boxes which is not usable due to “clipping.” Furthermore, I choose the unit of volume so that the capacity of the boxes is exactly 1 (and, thus, the sizes of the objects to pack are less than or equal to 1).

Let's take a look at some examples in order to get a feeling for the quality of the results of my online algorithms. If all objects have the same size – like in Example 1 – both NEXTFIT and FIRSTFIT find the optimal solution:

Example 1

$$G = \left(\frac{1}{4}, \frac{1}{4} \right)$$

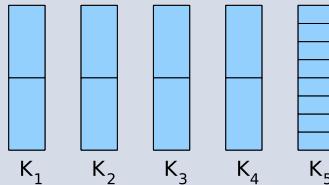
NEXTFIT = FIRSTFIT: $n = 5$



For the input in Example 2 both NEXTFIT and FIRSTFIT still find the optimal solution:

Example 2

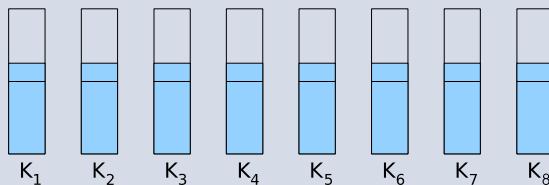
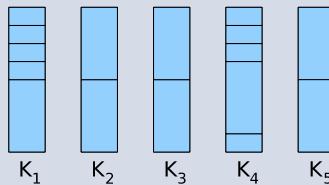
$$G = \left(\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8}, \frac{1}{8} \right)$$

NEXTFIT = FIRSTFIT: $n = 5$ 

But Example 3 illustrates that the order of the objects can influence the result – algorithm NEXTFIT performs rather poorly here:

Example 3

$$G = \left(\frac{1}{2}, \frac{1}{8}, \frac{1}{2}, \frac{1}{8} \right)$$

NEXTFIT: $n = 8$ FIRSTFIT: $n = 5$ 

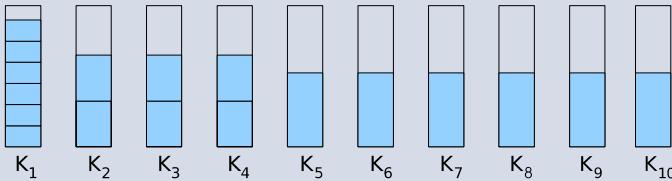
If one chooses a much smaller number than $\frac{1}{8}$ for the sizes of the small objects in Example 3, the unused space in each box grows to almost half of the size of the box when using NEXTFIT, i.e., in a worst-case scenario NEXTFIT would need almost double the amount of boxes compared to an optimal packing.

Although strategy FIRSTFIT has still performed optimally in Example 3, there are also bad inputs for FIRSTFIT, as shown in Example 4:

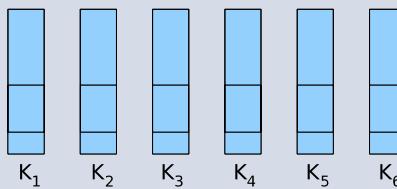
Example 4

$$G = (0.15, 0.15, 0.15, 0.15, 0.15, 0.15, \\ 0.34, 0.34, 0.34, 0.34, 0.34, 0.34, \\ 0.51, 0.51, 0.51, 0.51, 0.51)$$

FIRSTFIT: $n = 10$



Optimal: $n = 6$



Here we obtain a ratio of $10 : 6$ for FIRSTFIT compared to the optimal packing, i.e., FIRSTFIT needs 1.67 times as many boxes as an optimal algorithm would require.

Now we are curious whether or not the negative examples 3 and 4 are already the worst possible inputs for NEXTFIT and FIRSTFIT:

In Chap. 37 we learned that we call an online strategy α -competitive if it does not produce – for any input – more than α times the cost compared to an optimal solution, i.e., compared to the cost we would have produced if we knew the future. In order to determine the competitive factor of NEXTFIT, we denote the number of objects by k and the number of boxes used by NEXTFIT by n . Furthermore, we write $v(G_i)$ for the volume of object G_i and $v(K_j)$ for the volume used in box K_j . When now considering two successively filled boxes K_j and K_{j+1} , $1 \leq j < n$, it holds:

$$v(K_j) + v(K_{j+1}) > 1.$$

If this condition was not satisfied, all objects in K_{j+1} would still have fit into K_j and would thus not have been put into K_{j+1} by NEXTFIT. Now, when adding the volumes used in K_1 and K_2 , in K_3 and K_4 , in K_5 and K_6 , and so on, we notice that all these sums are always greater than 1. Thus, we get for

the sum over all boxes:

$$\sum_{j=1}^n v(K_j) > \left\lfloor \frac{n}{2} \right\rfloor.$$

Using the floor function $\lfloor \cdot \rfloor$ we round down the fraction on the right-hand side of the inequality in the case that n is odd. Hence, we obtain a lower bound for the number of required boxes, i.e., a value that cannot be undercut even with an optimal packing. Now we observe that the total volume of all objects is equal to the volume totally occupied in all boxes:

$$\sum_{i=1}^k v(G_i) = \sum_{j=1}^n v(K_j).$$

Thus, even the best possible packing requires at least

$$\left\lceil \sum_{i=1}^k v(G_i) \right\rceil \geq \left\lceil \frac{n}{2} \right\rceil$$

boxes, due to the total volume of the objects. The volume of the objects has to be round up here, because the number of the boxes has to be an integer. Hence, we obtain for the ratio of the number of boxes required by NEXTFIT to the number of boxes required for an optimal packing:

$$\frac{\text{solution NEXTFIT}}{\text{optimal solution}} = \frac{n}{\lceil \frac{n}{2} \rceil} \leq 2.$$

Thus the online algorithm NEXTFIT is 2-competitive (see introduction to online algorithms, Chap. 37). As this proof can be transferred to FIRSTFIT, it is 2-competitive as well. It is possible to show that FIRSTFIT is even 1.7-competitive, using a much more complicated proof (which we won't exercise here, see Further Reading).

How Well Can Online Algorithms for Bin Packing Perform?

We now know lower bounds for the approximation quality of NEXTFIT and FIRSTFIT, and we know that there are input sequences for which these bounds are almost reached, i.e., for which the results cost almost twice (for NEXTFIT) and 1.7 times, respectively, (for FIRSTFIT) as much as an optimal solution. On the one hand, this is a good result as we know that the wasted space in the boxes never exceeds a certain factor. But on the other hand, this result still is somehow unsatisfactory because you sorely feel it in your budget if moving costs 2000 Euro (or 1700 Euro, respectively) instead of 1000 Euro.

To finally judge how well or how poorly a strategy really performs, we also have to take into account how well any online algorithm for bin packing

can perform at all. Since the input sequence is not known in advance, it somehow seems impossible to design an online algorithm which always outputs an optimal result. We're now going to prove this fact.

Suppose we have an input sequence which contains $2 \cdot x$ objects of size $\frac{1}{2} - \varepsilon$, where x is a positive integer and ε is an arbitrarily small, positive number. The optimal packing for this input sequence obviously is: x boxes which contain 2 objects each. Let's now consider an arbitrary online algorithm and denote it by BINPAC. BINPAC will spread the $2 \cdot x$ objects over the boxes so that – depending on the strategy – each box either contains one or two objects. We denote the number of boxes containing one object by b_1 , and those containing two objects by b_2 . By $b = b_1 + b_2$ we denote the total number of boxes used by BINPAC. Then we can find this correlation:

$$b_1 + 2 \cdot b_2 = 2 \cdot x \quad \Rightarrow \quad b_1 = 2 \cdot x - 2 \cdot b_2.$$

By inserting this into $b = b_1 + b_2$ we get:

$$b = (2 \cdot x - 2 \cdot b_2) + b_2 = 2 \cdot x - b_2. \quad (38.1)$$

We'll get back to this intermediate result later. Now let's check what happens if our input sequence consists of $4 \cdot x$ objects, in which the first $2 \cdot x$ objects again have size $\frac{1}{2} - \varepsilon$ and the remaining $2 \cdot x$ size $\frac{1}{2} + \varepsilon$. Since online algorithms generally cannot look into the future, BINPAC will behave on the first $2 \cdot x$ smaller objects the same way as in the previous example, where no further objects were following. Thus, when placing the remaining objects of size $\frac{1}{2} + \varepsilon$, we can first fill up the b_1 many boxes with one object, and then we'll have to open a new box for each of the remaining $2 \cdot x - b_1$ objects. Hence, BINPAC at least needs

$$b + (2 \cdot x - b_1) = (b_1 + b_2) + (2 \cdot x - b_1) = 2 \cdot x + b_2 \quad (38.2)$$

boxes for this input sequence in total. But the optimal solution would have been to put one of the smaller and one of the bigger objects in each box, leading to a total of only $2 \cdot x$ required boxes.

Now we are ready to prove that no online algorithm is better than $\frac{4}{3}$ -competitive. We argue by first assuming that there would be a better online algorithm and then showing that this assumption leads to a contradiction.

Suppose BINPAC would be better than $\frac{4}{3}$ -competitive. Then the number of boxes used for the first input sequence would have to be strictly less than $\frac{4}{3}$ times the number of boxes required for an optimal solution. Formally:

$$b < \frac{4}{3} \cdot x.$$

Applying this to (38.1), we get:

$$2 \cdot x - b_2 < \frac{4}{3} \cdot x \quad \Rightarrow \quad b_2 > \frac{2}{3} \cdot x. \quad (38.3)$$

Analogously, for the second input sequence, we perceive that the number of the boxes used (see (38.2)) is strictly less than $\frac{4}{3}$ times the number of boxes required for an optimal solution ($2 \cdot x$). Formally:

$$2 \cdot x + b_2 < \frac{4}{3}(2 \cdot x) \Rightarrow b_2 < \frac{2}{3} \cdot x. \quad (38.4)$$

Now we end up in a contradiction because, as of (38.3) and (38.4), b_2 would have to be both strictly less and strictly greater than $\frac{2}{3} \cdot x$ at the same time, which is impossible. Thus, our assumption must have been wrong, and we have proven:

Theorem 1

There is no α -competitive online algorithm for the bin packing problem with $\alpha < \frac{4}{3}$.

Now, knowing that even the best possible online strategy for bin packing cannot be better than $\frac{4}{3}$ -competitive, the 1.7-competitive strategy FIRSTFIT appears in a much more positive light. Okay, people, let's get packing!

A further application for bin packing is, for example, to assign files to CDs when backing up a huge amount of data. In this case, the above-described strategies can even be applied directly, i.e., we don't have to make simplifying assumptions as there is no "clipping" problem when dealing with (one-dimensional) data streams.

Further Reading

1. <http://www-cg-hci-f.informatik.uni-oldenburg.de/~da/iva/baer/start/bin1.html>
This Java applet interactively illustrates how the FIRSTFIT algorithm works.
2. D. Johnson: *Fast algorithms for bin packing*. Journal of Computer and System Sciences 8 (1974), pp. 272–314.
This is the first publication on online bin packing.
3. S. Seiden: *On the online bin packing problem*. In: *Proceedings of the 28th International Colloquium on Automata, Languages and Programming* (July 2001), Springer, pp. 237–249.
The best algorithm for online bin packing known so far (HARMONIC++), which is 1.58889-competitive, is introduced in this article.
4. A. van Vliet: *An improved lower bound for online bin packing algorithms*. Information Processing Letters 43, 5 (October 1992), pp. 277–284.
In this article the value for α in Theorem 1 is improved from $\frac{4}{3}$ to 1.5401.

The Knapsack Problem

Rene Beier and Berthold Vöcking

Max-Planck-Institut für Informatik, Saarbrücken, Germany
RWTH Aachen, Aachen, Germany

In two months, the next rocket will leave Earth heading to the space station. The space agency, being a little short of money, offers to carry out scientific experiments for other research institutions at the space station. For each experiment, some equipment needs to be lifted up to the station. However, there is a limit on the weight the rocket can carry. Apart from the obligatory food rations, the rocket is able to carry up to 645 kg of scientific equipment. The space agency receives several offers from different research institutions stating how much they are willing to pay for the execution of an experiment and specifying the weight of the necessary equipment. The space agency wants to figure out which of these experiments should be chosen in order to maximize the profit.

This scenario exemplifies a classic optimization problem, the so-called knapsack problem: Suppose we have a knapsack that has a certain capacity described by a weight threshold T . We are given a set of n items, each bearing a weight and a profit. The task is to choose a subset of the items that should go into the knapsack. Only subsets of total weight at most T can be chosen. The objective is to make the profit as large as possible. That is, one seeks for a subset such that the sum of the profits of the items in the subset is as large as possible under the constraint that the sum of the weights of these items is at most T .

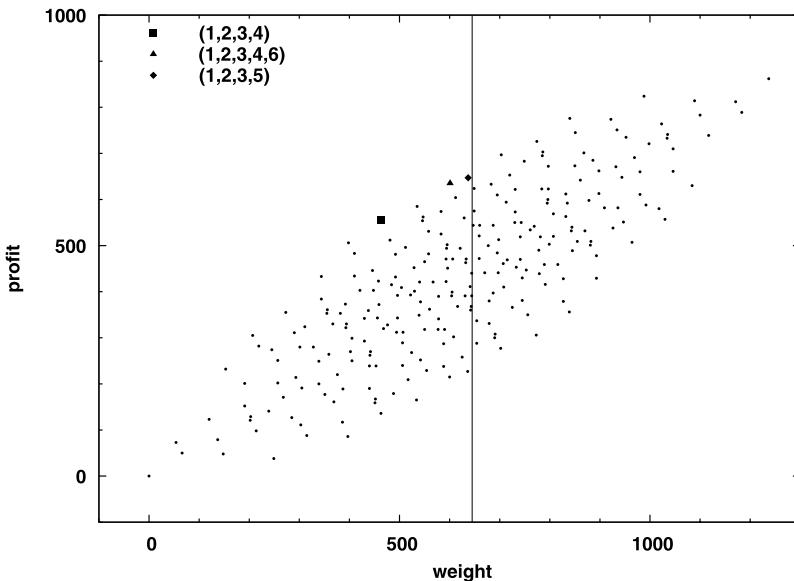
In the introductory example the knapsack corresponds to the rocket. The weight threshold is $T = 645$. The items correspond to the experiments. To make the example more concrete we assume that the space agency can choose from $n = 8$ experiments listed with the following weights and profits.

Item	1	2	3	4	5	6	7	8
Weight in kg	153	54	191	66	239	137	148	249
Profit in 1000 Euro	232	73	201	50	141	79	48	38
Profit density	1.52	1.35	1.05	0.76	0.59	0.58	0.32	0.15

How can we find a feasible set of items that maximizes the profit? Intuitively we could choose those items first that achieve the largest profit per unit weight. This ratio between profit and weight is called profit density. We have calculated the profit density for all items. The table lists the items from left to right in such a way that this ratio is decreasing.

Our first algorithm follows the idea above and sorts all items by decreasing profit density, e.g., by using the algorithms from Chap. 2 or Chap. 3. Starting with the empty knapsack we add items one by one in this order as long as the capacity of the knapsack is not exceeded. In our example, we would pack items 1, 2, 3, and 4 as their cumulative weight is 464, which is still below the threshold. Adding item 5 would result in a total weight of $464 + 239 = 703$, which would overload the knapsack. The four packed items yield a profit of 556. Is this the maximal profit that we can achieve? Not quite. By adding item 6 we obtain a knapsack packing with a total weight of 601 that is still feasible and has a total profit of 635. Is this now the most profitable subset? Unfortunately not.

Of course, in order to guarantee optimality we could test all possible combinations of packing the knapsack. To better illustrate let us draw all possible packings in a weight–profit diagram, irrespective of the weight threshold. For instance, we draw the point $(601, 647)$ for the packing with items $\{1, 2, 3, 4, 6\}$.



Each point represents a subset of items. How many points do we have to draw? We must decide whether to include each item or not; hence there are two possibilities per item. As the choice for each item is independent from the choice made for the other items, there are 2^n possibilities for n items. Thus, in our example, we would need to test $2^8 = 256$ different packings.

The packings with weights greater than the threshold T are not feasible. These subsets correspond to points in the diagram that lie to the right of the vertical line. Points to the left of or exactly on the vertical line correspond to feasible packings. Among those feasible packings we choose the one with maximum profit. In our example, it is the point with coordinates 637 and 647, corresponding to the packing containing items 1, 2, 3, and 5. This is the *optimal solution*.

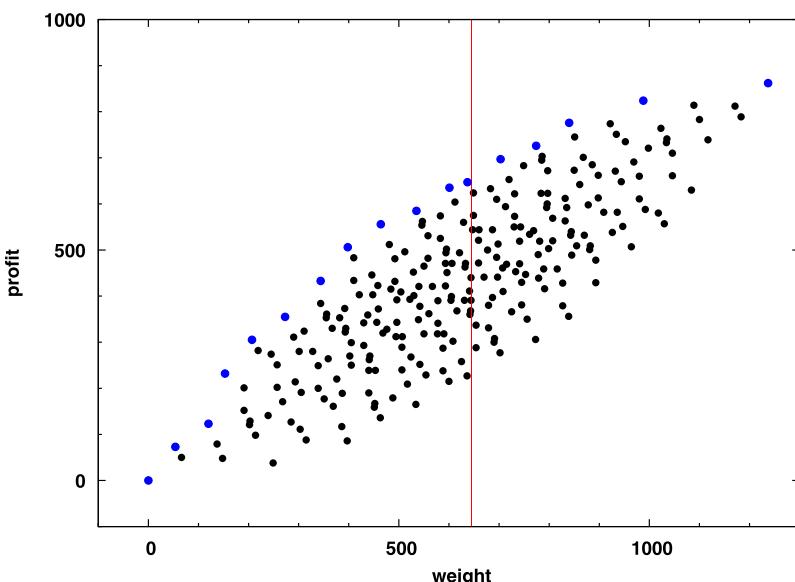
This approach of finding the best solution is practical only for a small number of items as the computational effort increases rapidly when increasing the number of items. Each additional item doubles the number of packings that need to be tested. When the space agency, for example, can choose among 60 experiments the number of possible packings grows to

$$2^{60} = 1,152,921,504,606,846,976.$$

Assuming that a modern computer can test 1,000,000,000 packings per second, it would still take more than 36 years to finish the computation. Of course, we do not want to delay the start of the rocket for such a long time.

Pareto-Optimal Solutions

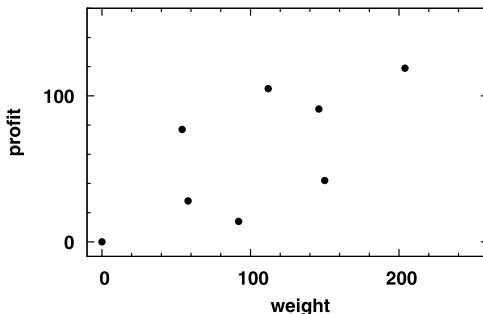
How can we find the optimal solution faster? The key to a more efficient algorithm is the following observation: A packing cannot be optimal if there exists another packing with lower weight and higher profit. Let's go back to our example.



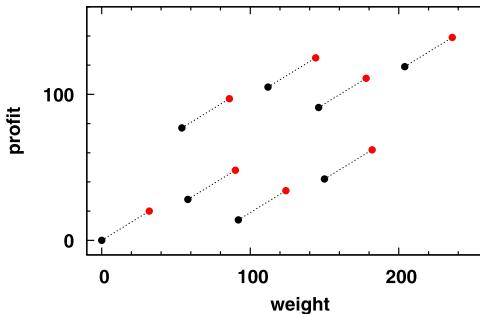
None of the black points can be the optimal solution since for each of them we can find at least one other point that is better, i.e., that has both lower weight and higher profit. We say that this better point *dominates* the black point. The blue points in the diagram are exactly those points that are not dominated by any other point. These points are called *Pareto-optimal*. Hence, a packing is Pareto-optimal if there exists no other packing of less weight that yields a higher profit. In our example, only 17 of the 256 packings are Pareto-optimal. The optimal solution must be among those 17 packings. Observe that the property of being Pareto-optimal is independent of the choice of the weight threshold. In particular, for any given threshold we can find the optimal solution among those 17 packings.

So far we have ignored in the discussion the fact that two packings can have the same weight or the same profit. In order to correctly handle those cases we define that packing A dominates packing B if A is at least as good as B in weight and in profit and additionally if A is strictly better than B in at least one of the two criteria.

But how can we compute a list of all Pareto-optimal packings efficiently, namely without testing all 2^n packings? Consider the following small example starting with three items. We plotted all $2^3 = 8$ different packings in a weight-profit diagram.



Let us assume that we know the set of Pareto-optimal packings for these three items, and we consider an additional fourth item. For each of the eight packings we can generate a new packing by adding the fourth item. This way, we obtain eight additional packings. Each black point generates a new red point with a horizontal and vertical shift corresponding to the weight and profit of the fourth item. Hence, the set of red points is just a shifted copy of the set of black points.



What can we say about the Pareto-optimality of these 16 points? We exploit that we already know the set of Pareto-optimal packings for three items. A black point dominated by some other black point is by definition not Pareto-optimal. The same holds for each red point dominated by some other red point. In other words, a black point that is not Pareto-optimal with respect to the black point set (3 items) cannot become Pareto-optimal with respect to all 16 points (4 items). The same applies to red points that are not Pareto-optimal with respect to the red-point set. As a consequence, only points from the following two sets can potentially be Pareto-optimal.

- A: black points that are Pareto-optimal with respect to the black point set,
and
- B: red points that are Pareto-optimal with respect to the red point set.

Observe that the points in B are just shifted copies of the points in A . Now consider a point p from A . Suppose p is dominated by a red point q . If q does not belong to B then there must be a red point q' in B that dominates q and thus dominates p as well. Hence, in order to check the Pareto-optimality of a point from A (with respect to both black and red points), one only needs to check if this point is not dominated by a point from B . Analogously, in order to check the Pareto-optimality of a point from B , one only needs to check if it is not dominated by a point from A .

Now we have a procedure for computing the set of Pareto-optimal packings when adding one additional item: First construct all red points that are generated from Pareto-optimal black points. Then delete all red points dominated by a black point. Finally, delete all black points dominated by a red point.

The Nemhauser–Ullmann Algorithm

The following algorithm was invented by Nemhauser and Ullmann in 1969. It uses the arguments above in an iterative fashion, adding one item after the other. That is, it starts with the empty set of items and adds items one by one until it finally obtains the set of Pareto-optimal packings for all n items.

An efficient implementation maintains the set of Pareto-optimal points in a list that is sorted according to the weight of the points. The initial list L_0 contains only the point $(0, 0)$, which corresponds to the empty packing. Now we iteratively compute lists L_1, L_2, \dots, L_n , where L_i denotes the list of Pareto-optimal points with respect to items 1 to i .

Using L_{i-1} and the i th item we compute L_i as follows. First we generate L'_{i-1} (the red point set), which is a shifted copy of L_{i-1} (the black point set). Each point in L_{i-1} has to be copied and shifted right and up by the weight and profit of the i th item, respectively. Now we merge the two lists, L_{i-1} and L'_{i-1} , filtering out points that are dominated. As both lists are sorted according to the weight of the points (and therefore also according to the profit – can you tell why?), this task can be achieved by scanning only once through both of these lists. Thus, the time needed to merge these lists is linear in the sum of the length of the two lists.

The algorithm MERGE merges two sorted lists of points L and L' .

```

1  procedure MERGE ( $L, L'$ )
2  BEGIN
3      PMAX = -1;  $E = \{\}$ 
4      REPEAT
5          Scan  $L$  for a point  $(w, p)$  with profit  $p >$  PMAX
6          Scan  $L'$  for a point  $(w', p')$  with profit  $p' >$  PMAX
7          If no point has been found in line 5 (finished scanning  $L$ )
8              Insert remaining points from  $L'$  into  $E$ ; RETURN( $E$ )
9          If no point has been found in line 6 (Finished scanning  $L'$ )
10             Insert remaining points from  $L$  into  $E$ ; RETURN( $E$ )
11         IF  $(w < w') \text{ OR } (w = w' \text{ AND } p > p')$ 
12             THEN insert  $(w, p)$  into  $E$  and set PMAX :=  $p$ 
13         ELSE insert  $(w', p')$  into  $E$  and set PMAX :=  $p'$ 
14     END
```

The resulting list L_n contains all Pareto-optimal points with respect to the n items. From this list, we choose the point with maximal profit whose weight is at most T . The packing belonging to this point is the optimal solution.

Is this algorithm better than simply testing all possible packings? Not in every case, as it is possible that all 2^n packings are Pareto-optimal. This can happen, for instance, when the profit density of all items has the same value c , i.e., $p_i = c * w_i$ for some constant c . However, this is not what one typically experiences. Usually, the number of Pareto-optimal solutions is much smaller than 2^n . In our introductory example we generated weights and profits of the eight items at random. Only 17 out of the 256 packings are Pareto-optimal. Mathematical and empirical analyses show that typically only a very small fraction of the packings are Pareto-optimal. For this reason, the described algorithm can handle instances of the knapsack problems with thousands of items in a reasonable amount of time.

Further Reading

1. H. Kellerer, U. Pferschy and D. Pisinger: *Knapsack Problems*. Springer, 2004.

This nice textbook is completely devoted to algorithms for the knapsack problem. It discusses various practical variations of this problem and presents several algorithmic approaches to tackle these problems. It gives a comprehensive overview of the state of the art in this field.

2. S. Martello and P. Toth: *Knapsack Problems: Algorithms and Implementations*. Wiley, Chichester, 1990.

This is an older textbook about the knapsack problem; nevertheless it gives some interesting insights into different approaches for solving the problem and its variations.

3. Wikipedia elaborates on the knapsack problem, too. In particular, it presents an algorithm using the dynamic programming paradigm. (An introduction to this paradigm applied to a different problem can be found in Chap. 31 of this book.) The same algorithm can be found in several introductory textbooks about algorithms as well. In many instances, however, it is much slower than the algorithm that we have presented here:
http://en.wikipedia.org/wiki/Knapsack_problem

4. The knapsack problem is closely related to bi- or multicriteria optimization problems which optimize two or more criteria simultaneously. For example, one is given n objects, each of which comes with a profit and a weight, like in the knapsack problem, but there is no threshold on the weight. Instead one assumes that there is a decision-maker that, on the one hand, seeks for a subset of items giving a large profit. On the other hand, the decision-maker wants to select a subset of low weight. Of course, these are conflicting goals and the question is how to resolve them. Such problems arise in many practical applications. For example, consider a navigation system for traveling by car. (Such a system uses algorithms similar to the shortest-path algorithm explained in Chap. 32.) The objective might be to find a short route between a starting point and a destination in a traffic network. The shortest route, however, is not always the quickest one, as it might directly lead through the city rather than taking a relatively short detour along the highway on which one can travel much faster. Here every route could be attached two values, distance and time. The interesting solutions are the Pareto-optimal ones with respect to these two criteria among which the driver should make his choice. An entry point into the field of Pareto optimization can be found in Wikipedia:

<http://en.wikipedia.org/wiki/Pareto-efficiency>

The Travelling Salesman Problem

Stefan Näher

Universität Trier, Trier, Germany

Introduction

The Travelling Salesman Problem (TSP) is one of the most famous and most-studied problems in combinatorial optimization. It is defined as follows: A *travelling salesman* has to visit n cities in a *round trip* (often called *tour*). He starts in one of the n cities, visits all remaining cities one by one, and finally returns to his starting point.



The actual optimization problem is to find a tour of minimal total length. For this purpose the distances between all pairs of cities are given in a table or matrix. Besides the exact geometric distance values other values may be used, such as travel time or the cost of the required amount of fuel. The goal is to plan the tour in such a way that the total distance, travel time, or the total cost is minimized, respectively.

TSP belongs to a class of very difficult problems: the so-called NP-hard problems (see also, for example, Chap. 39). No efficient algorithms are known for this class of problems. In fact, it is assumed that every algorithm to solve these problems exactly needs exponentially many steps. However, there exists no mathematical proof of this assumption. Efficient algorithms have been found for special variants of TSP and for the computation of approximative solutions, which allow tours that may be more expensive than the optimal tour.

The TSP appears in many practical optimization problems as a subproblem, e.g., optimization problems in traffic scheduling or logistics. On the other hand, many NP-hard problems can be reduced to the TSP problem.

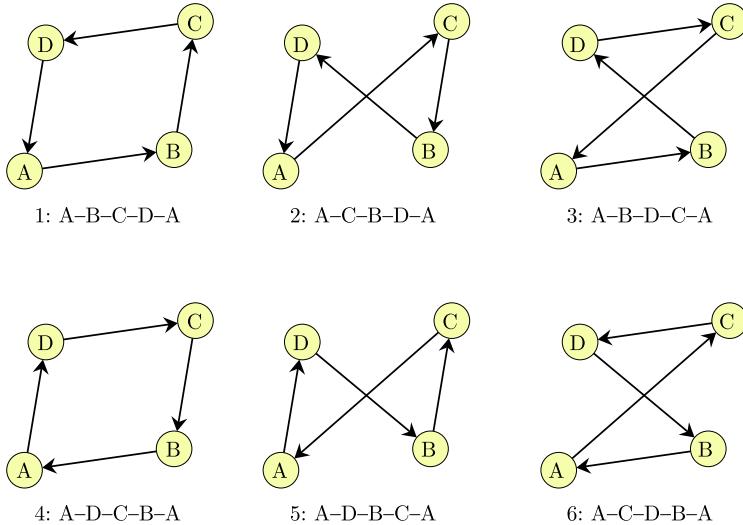
We will begin with a very simple strategy for solving TSP exactly, the so-called *brute-force* method. This method is a good example for the disastrous slowness of algorithms with exponential running time.

The Brute-Force Method

The brute-force method is the simplest algorithm to solve TSP exactly. It looks at all possible tours one by one, computes the total length of each tour, and computes the optimal tour by finding the minimum of these length values using comparisons. Unfortunately, the total number of all possible tours grows extremely fast with increasing numbers of cities. It is easy to see that there are $(n - 1)! = 1 \cdot 2 \cdot 3 \cdot 4 \cdots \cdots (n - 1)$ different ways to visit n cities by a tour. Each tour has to start in one city (e.g., the first one), then it has to visit all $n - 1$ remaining cities in an arbitrary order, and finally return to the first city. However, there are exactly $(n - 1)!$ possible orderings or permutations of the remaining $n - 1$ cities.

Figure 40.1 shows the $6 = (4 - 1)!$ possible tours for four cities A, B, C, and D. Note that the lower three tours are just reversals of the upper three tours. So it is only necessary to compute the total length of half of the tours. This leads to the observation that the algorithm has to look at $\frac{1}{2} \cdot (n - 1)!$ tours. After all, there exist variants of TSP without this symmetry property. Then, in fact, all tours have to be considered.

Table 40.1 shows how extremely quickly the number of tours grows with increasing n . The last column gives an estimate of the running time of a program solving TSP using the brute-force method. Here we assume that the handling of one single tour takes about one millisecond. In the last row of the table one can see that such a program would run for about 20 years to solve a TSP with only 16 cities. The table also shows that running times not exceeding one hour are only possible for problems with at most ten cities. Chapter 39 of this book shows a similar problem with an extremely fast-growing number of possible situations.

**Fig. 40.1.** All possible tours for four cities

Cities	Possible tours	Running time
3	1	1 msec
4	3	3 msec
5	12	12 msec
6	60	60 msec
7	360	360 msec
8	2,520	2.5 sec
9	20,160	20 sec
10	181,440	3 min
11	1,814,400	0.5 hours
12	19,958,400	5.5 hours
13	239,500,800	2.8 days
14	3,113,510,400	36 days
15	43,589,145,600	1.3 years
16	653,837,184,000	20 years

Table 40.1. Number of all possible tours and running time of brute-force method

Dynamic Programming

Recursion is a strategy that is used very frequently in computer science and in particular in algorithm design. The idea is to reduce the solution of a problem to smaller problems of the same kind (see also Chap. 3). *Dynamic Programming* is a special variant of this technique that maintains the results of recursively defined subproblems in a table.

Let us assume that the n cities of TSP are numbered from 1 to n such that we can write the set of all cities as the set $S = \{1, 2, \dots, n\}$. Furthermore we assume that the distances between all pairs of cities are given in a table $DIST$ such that $DIST[i, j]$ is equal to the distance from city i to city j . Since a tour can start and end in an arbitrary city, we can assume that every tour begins in city 1, then moves on to some city $i \in \{2, \dots, n\}$, visits all remaining cities $\{2, \dots, i-1, i+1, \dots, n\}$, and finally returns to city 1.

Let i be an arbitrary city and A some set of cities, then define $L(i, A)$ to be the length of a shortest path

- that begins in city i ,
- visits every city in set A exactly once,
- and ends in city 1.

The following observations lead to an algorithm for the computation of all $L(i, A)$ values:

1. $L(i, \emptyset) = DIST[i, 1]$ for each city $i \in S$.

It is obvious that the shortest path from i to 1 that visits no other city must be the direct connection from i to 1.

2. For each city $i \in S$ and subset $A \subseteq S \setminus \{1, i\}$ we have

$$L(i, A) = \min\{DIST[i, j] + L(j, A \setminus \{j\}) \mid j \in A\}.$$

If set A is not empty then the optimal path can be defined recursively as follows: For every $j \in A$ consider the path that beginning in i first visits j . For the shortest of these paths the rest, leading from j to 1, must be optimal too. According to our definition this path has length $L(j, A \setminus \{j\})$.

3. $L(1, \{2, \dots, n\})$ is the length of an optimal TSP tour. This follows immediately from the fact that any tour begins in 1, visits all remaining cities, and finally returns to 1.

Then the table of all $L(i, A)$ values can be computed for larger and larger subsets A by the following algorithm.

Algorithm TSP WITH DYNAMIC PROGRAMMING

```

1  for  $i := 1$  to  $n$  do  $L(i, \emptyset) := DIST[i, 1]$  endfor;
2  for  $k := 1$  to  $n - 1$  do
3      forall  $A \subseteq S \setminus \{1\}$  with  $|A| = k$  do
4          for  $i := 1$  to  $n$  do
5              if  $i \notin A$  then
6                   $L(i, A) = \min\{DIST[i, j] + L(j, A \setminus \{j\}) \mid j \in A\}.$ 
7              endif
8          endfor
9      endfor
10 endfor
11 return  $L(1, \{2, \dots, n\});$ 
```

n cities	Size of the table ($n^2 \cdot 2^n$)	Running time
3	72	72 msec
4	256	0.4 sec
5	800	0.8 sec
6	2304	2.3 sec
7	6272	6.3 sec
8	16,384	16 sec
9	41,472	41 sec
10	102,400	102 sec
11	247,808	4.1 minutes
12	589,824	9.8 minutes
13	1,384,448	23 minutes
14	3,211,264	54 minutes
15	7,372,800	2 hours
16	16,777,216	4.7 hours

Table 40.2. Table size and running time of dynamic programming

The algorithm fills a table of n rows ($i = 1, \dots, n$) and at most 2^n columns (for all subsets of size $k \leq n - 1$). Thus the size of this table is at most $n \cdot 2^n$. For every entry a linear search for the minimum of n numbers is performed (lines 4 to 7). Then line 6 of the algorithm is executed at most $n \cdot n \cdot 2^n = n^2 \cdot 2^n$ times.

Table 40.2 shows in column 2 the values of this number for TSP problems with at most 16 cities. The last column gives the corresponding running times. Here it is assumed that the computation of a single table entry takes one millisecond. It is easy to see that the total running time is still growing faster than exponentially. However, we achieve a dramatic improvement compared to the brute-force method described in the previous section. The time for solving a TSP for 15 cities is reduced from 20 years to 5 hours.

Approximative Solutions

The dynamic programming approach has dramatically improved the running time compared to the brute-force method. However it still requires time exponential in the input size n and is completely useless for larger values of n . A second problem is the huge memory consumption for storing the table. This section presents an algorithm that does not always find the optimal TSP tour but it does find a pretty good one. More precisely, it computes a tour with a length not exceeding two times the optimal tour length. Such an algorithm is also called a *heuristic*.

To this purpose we model TSP as a problem on a graph G whose nodes represent the cities, and an edge between two nodes i and j represents the direct shortest connection from i to j . We assume that every edge is labeled

with the corresponding distance. Since for every pair (i, j) of cities there exists such a direct connection, the constructed graph $G = (V, E)$ is a so-called *complete graph* that contains all pairs of nodes as edges, i.e., $E = V \times V$. In this model a tour is represented by a cycle in G that visits every node exactly once. Such a cycle is called *Hamiltonian*.

There is a simple relation between possible tours (i.e., Hamiltonian cycles) and special subgraphs of G , the so-called spanning trees. A *spanning tree* is an acyclic subgraph connecting all nodes of G . A *minimal spanning tree* is a spanning tree with minimal total cost (summing up the cost of all edges).

Removing an arbitrary edge from a tour yields a so-called *Hamiltonian path*. Since any Hamiltonian path fulfills the conditions of a spanning tree (an acyclic subgraph connecting all nodes), its total cost must be at least as large as the cost of a minimum spanning tree. In other words, the total cost of a minimum spanning tree cannot exceed the total cost of an optimal tour.

Efficient algorithms for minimum spanning trees are presented in Chap. 33 of this book. The most expensive step of these algorithms is to sort all edges of the graph according to their cost. Starting with a minimal spanning tree it is easy to construct a TSP tour. The complete algorithm consists of the following steps: The corresponding figures have been produced by a program that can be downloaded from <http://www-i1.informatik.rwth-aachen.de/~algorithmus/Algorithmen/algo40/tsp.exe>.

MST Algorithm

1. Construct the complete graph $G = (V, E)$, where V represents the set of all cities and $E = V \times V$ (see Fig. 40.2).
2. Compute a minimal spanning tree T of G such that the cost of every edge (i, j) is equal to the distance between city i and city j ($DIST[i, j]$). Figure 40.3 shows the result of this step.
3. In the next step, tree T is transformed into a first tour by simply walking around T along its edges (see Fig. 40.4). The total length of this tour is

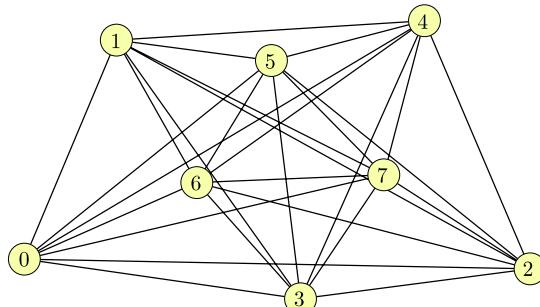


Fig. 40.2. The complete graph of all direct connections

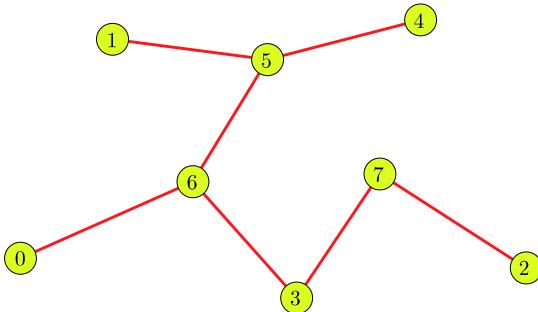


Fig. 40.3. The minimal spanning tree

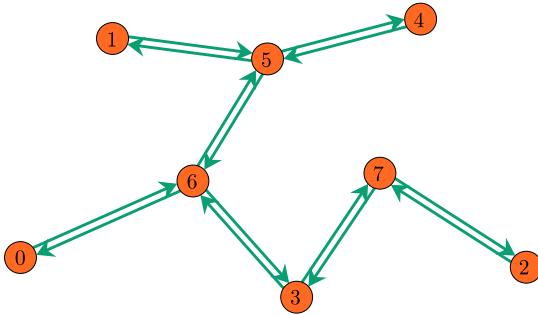
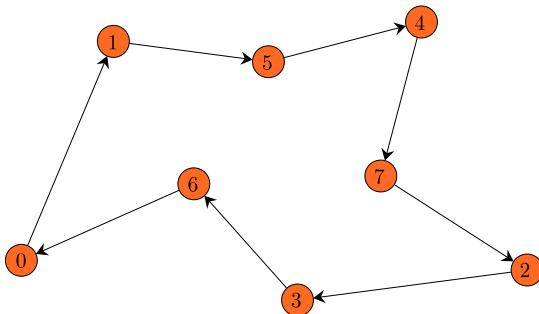


Fig. 40.4. Tour around the minimum spanning tree

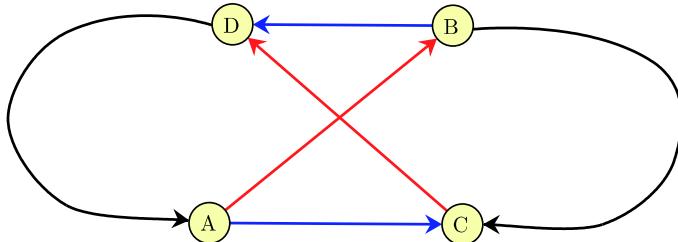
apparently twice as large as the total cost of T , because every edge is used twice. Following the considerations made above we conclude that the total length of this tour is *at most twice* as large as the length of an optimal TSP tour.

4. Obviously, the tour constructed in the previous step is not optimal. Actually it is not a correct tour at all, because it visits every node twice. However, it is not difficult to turn it into a correct tour with smaller total length. Consider three successive nodes a, b, c , and check whether the two edges $a \rightarrow b \rightarrow c$ can be replaced by the shortcut $a \rightarrow c$ without isolating node b . The result of this step is shown in Fig. 40.5.

Table 40.3 shows the running time of the MST algorithm for randomly chosen cities. The program used to create the table computes a minimum spanning tree of a graph with 1000 edges in about one millisecond. As shown in the table, the algorithm finds an approximate solution for a TSP problem with 1000 cities in about one minute. The computed tour can be improved by further heuristics. As an example, we explain the 2-OPT method (see Fig. 40.6). Consider two arbitrary edges of the computed tour $A \rightarrow B$ and $C \rightarrow D$. Removing these edges splits the tour into two pieces from B to C

**Fig. 40.5.** The minimum spanning tour

Cities	Running time
100	0.01 sec
200	0.08 sec
300	0.36 sec
400	1.30 sec
500	3.62 sec
600	8.27 sec
700	16.07 sec
800	29.35 sec
900	50.22 sec
1000	85.38 sec

Table 40.3. Running time of the MST-heuristics**Fig. 40.6.** One step of the 2-OPT heuristics

and from D to A . Now check if these two parts can be combined to a shorter tour by adding edges $A \rightarrow C$ and $D \rightarrow B$.

Some Final Remarks

- There are many more heuristics for improving the quality of approximative solutions. In many practical cases they are even able to find the optimal tour.

- Algorithms for the exact solution of TSP have been considerably improved over recent years. They can solve problems with several thousand cities in a few hours.
- In the worst case solving TSP remains difficult and requires an exponential number of computation steps.

Further Reading

1. Wikipedia: Travelling Salesman Problem
http://en.wikipedia.org/wiki/Travelling_Salesman_Problem.
2. An introduction by Martin Grötschel and Manfred Padberg (in German)
<http://elib.zib.de/pub/UserHome/Groetschel/Spektrum/index2.html>
3. The Travelling Salesman Problem Home Page
<http://www.tsp.gatech.edu>
4. The Knapsack Problem (Chap. 39)
An optimization problem with a similarly fast growing number of possibilities.
5. Fast Sorting Algorithms (Chap. 3)
An introduction to recursive algorithms.
6. A demo program that provided the figures in this chapter
<http://www-i1.informatik.rwth-aachen.de/~algorithmus/Algorithmen/algo40/tsp.exe>

Simulated Annealing

Peter Rossmanith

RWTH Aachen, Aachen, Germany

Let us look at a simple combinatorial puzzle game: we can place several quadratic tiles on a wooden board at 12×8 different positions. At the beginning the tiles might be placed as indicated in Fig. 41.1. As you can see, each tile has four sides that may be colored blue, yellow, green, and orange.

We can exchange the positions of any two tiles, but we are not allowed to rotate them. The goal of the game is to obtain as many neighboring sides of tiles of the same color as possible: You score one point for each such neighboring pair. In that way it is not possible to score more than 172 points: Each

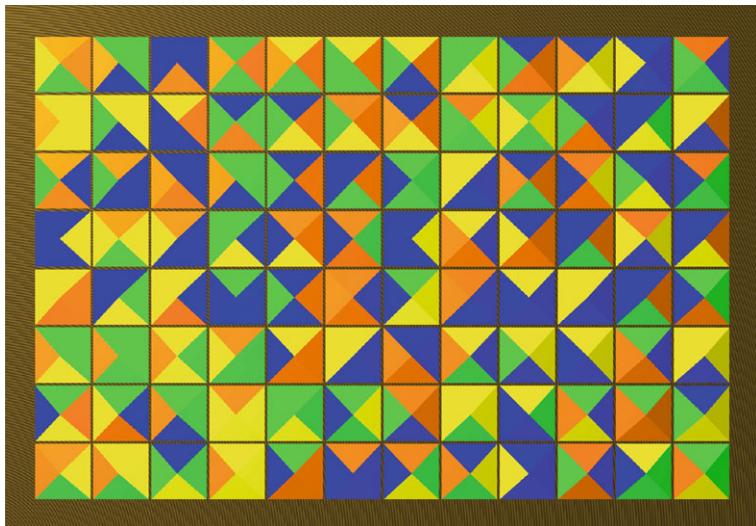


Fig. 41.1. Background of the domino game: The pieces are arranged at random in the box. Even in this situation we see 36 points

row contains 11 tile pairs and each column contains 7. There are 8 rows and 12 columns making altogether $11 \cdot 8 + 7 \cdot 12 = 172$.

Throughout this chapter we will continue to consider this tiling game as a typical example that is similar to many problems in combinatorial optimization. Some of the optimization problems mentioned in this book can be solved quite efficiently (Chaps. 32, 33, 34, and 35), while we can exactly solve others only for relatively small input sizes (Chaps. 39 and 40). Here we are interested more in the latter kind, in particular problems that can neither be solved by trying out all possibilities nor with the help of backtracking. One method that works well in many of those cases is *simulated annealing*, which is the topic of this chapter.

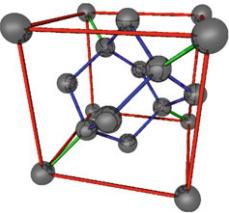
What Is Simulated Annealing?

Simulated Annealing means simulating a process that involves first heating and then slowly cooling some material. There are several technological processes based on a similar principle. For example, cooling red hot iron in metallworking quickly or slowly leads to materials with quite different properties. So why do that? Let us think about what happens to the particles (atoms) of the metal:

The atoms are bound in a rigid crystal structure. When we start to heat the metal they start to break free from their bonds and move about. If we let the metal cool again, the particles will find new bonds. Interestingly, often their new distribution will be more regular than before. Doing it the right way leads to metal that is softer, more flexible, and contains fewer irregularities.

To better understand the effects of slowly cooling, we can think of the particles as small balls. If you just throw them into a vessel, they will lie around higgledy-piggledy. What can we do to put them into order? Shake them! At first their disorder increases and the balls fly about, but soon they get into order all by themselves. If we, however, stop shaking too suddenly, the balls will not be packed very closely.

This is also an important principle in the manufacturing of silicon semiconductors, of which computers' processors and memory chips consist. In that case we require very pure silicon crystals that do not contain any defects. Usually silicon is polycrystalline: similarly to grains of sand, many small monocrystals are placed close to each other. Each monocrystal itself consists of very many small elementary cubes that are flawlessly stacked next to each other. On the left side you can



see such a silicon elementary cube. On the outside eight silicon atoms form a cube in whose interior ten more silicon atoms reside. In semiconductor production we need relatively large monocrystals. To this end a monocrystalline "pillar" is slowly drawn out of a bath of molten silicon. Consequently, with

the help of a saw the monocrystal is cut into wafers, which finally are turned into electronic semiconductors.

The monocrystalline state of silicon has the smallest energy possible: the bonds between individual atoms are strongest. From this perspective, the difference from our tiling game is not that big anymore. Here, too, we are concerned with elementary particles whose relative position can be changed. The bond between two neighboring tiles is stronger if sides with the same color face each other. Again, we are looking for a tessellation with smallest possible energy. If our game were a heated crystal, the tiles would wildly jump about. The lower the temperature gets, the harder it becomes to tear a tile from its position, and the more stable bonds they have with their neighbors, the stronger they stick to their position.

It seems to be very hard to modify the board game in order to make it behave just like that, although it might be possible: Perhaps we could enforce stronger bonds between sides of same color by cleverly located magnets and mount the whole board on a vibrating table. On the other hand, it is much easier to simulate the whole process using a computer. Shaking the board corresponds to swapping the positions of two tiles in the simulation. Doing so, it is quite easy to compute how much the number of points scored increases or decreases. At high temperatures we tolerate swaps that decrease the number of points, while at low temperatures we get stricter and will tend to allow only “good” changes to the board. The following algorithm can easily be implemented in all usual programming languages:



Tiling Game

Repeat quite often:

1. Decrease the temperature a little
2. Swap two randomly chosen tiles
3. If the number of points has decreased:
 - Randomly decide whether to keep this swap
 - The probability for keeping the swap decreases with the temperature
 - Undo the swap in case of a negative decision

This algorithm works surprisingly well for our tiling game. During the execution of the algorithm the point score both increases and decreases. In the beginning its fluctuation is quite big, but the longer we wait – and the lower the temperature gets – the smaller it becomes. Finally, we cannot notice

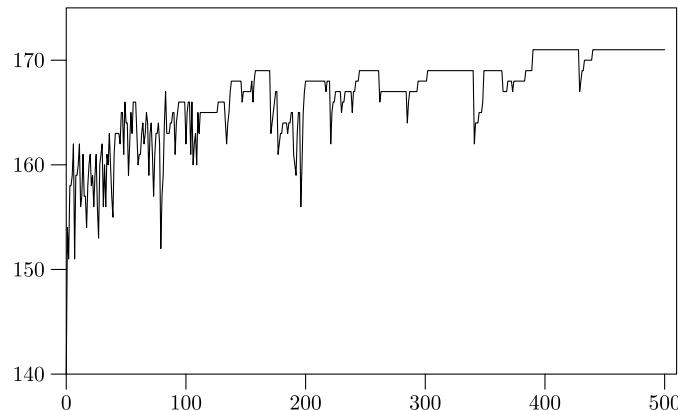


Fig. 41.2. How the point score changes by repeatedly swapping tiles: The y -axis indicates the number of points and the x -axis shows the number of steps in units of 10,000. Altogether 500,000 steps are executed. In the end the score amounts to 171 points

any change in the point score anymore. Figure 41.2 shows how the point score changes in the course of time, showing only scores higher than 140. This area is reached very quickly, while further improvements become more infrequent toward the end.

At this point, the question why we do not always take back swaps that decrease the point score arises – in that way we would never voluntarily give up earned points? Very often such a strategy works quite well and, therefore, bears its own name: *method of steepest descent* (in our metaphor of reaching the highest mountain, *method of steepest ascent* would make more sense, but some optimization problems happen to be maximization problems, while others are minimization problems, and the name was chosen based on the latter ones). This strategy only climbs upwards, never downwards. If we want to climb the highest mountain, we choose a direction that leads upwards, until no such direction exists anymore, which is only the case on a summit. Can we be sure that it is the highest summit? Not necessarily!

One who wants to climb the highest mountain, must be also willing to descend along the way.

An application of the method of steepest descent to our tiling problem leads to a solution with 167 points, from which point on it is impossible to further increase the score by swapping tiles. This score is quite good. Simulated annealing, however, leads to the better solution with 171 points, shown in Fig. 41.3. Only one possible point is missing, whose location is not at all easy to spot with just a quick glance on the board. (Hint: Tilt the picture by 45° and look along the visually emerging diagonal lines.)

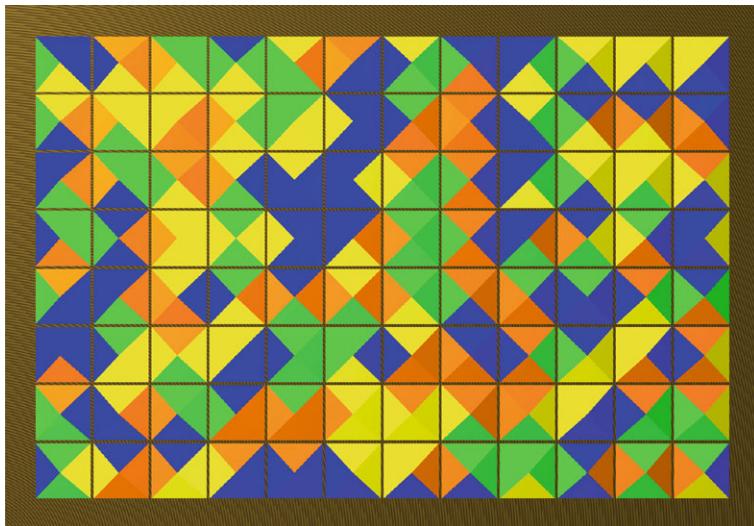


Fig. 41.3. Positions of the tiles after application of simulated annealing: The score is 171 points, and a closer look reveals that we got almost all possible points

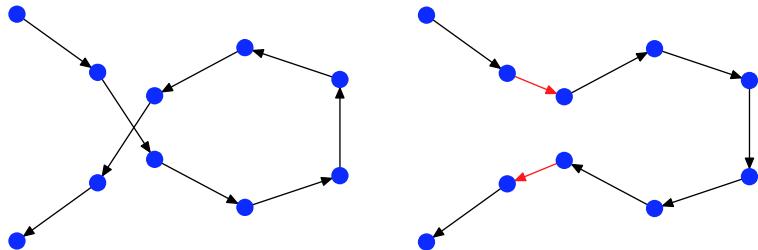
Scoring 171 of theoretically possible 172 points is an excellent result achieved by simulated annealing. We leave the natural question of whether a perfect solution with 172 points exists as an open question to the interested reader.

The Travelling Salesperson Problem

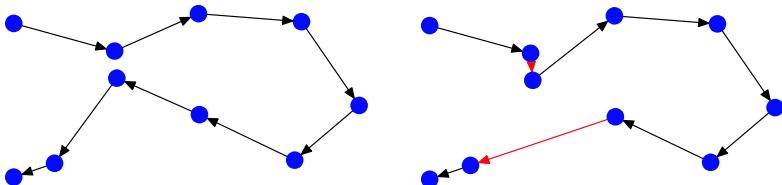
Let us have another look at a typical and famous problem in computer science, the *travelling salesperson problem*, which is also a topic in Chap. 40 in this book: A salesperson likes to visit a number of cities, spending as little time as possible for the trip. So the goal is to fix the sequence in which he or she visits the cities in such a way that the total mileage traveled becomes as small as possible. Surprisingly, this turns out to be a very hard problem.

Maybe we can find a nearly optimal solution by employing *simulated annealing*. To find some solution to start with is easy: Just put the cities in some arbitrary order! So we start with a random tour and, of course, it is highly improbable that this tour is short.

How can we improve this tour by incorporating only small adjustments? Well, one possibility is to choose randomly a segment of the tour and traverse it in the opposite direction:



Another possibility is to visit some city at a different point in time, while preserving the order in which we visit the remaining cities:



After adjustments of either kind, the new tour might be better or worse than the old one. If the new one is better, we keep it, otherwise we return to the old tour.

A big tour connecting 200 cities is depicted in Fig. 41.4. Of course, it was found by using the two kinds of adjustments mentioned above and simulated annealing.

Further Applications

We can employ simulated annealing if the following conditions are met:

1. The quality of a solution can be expressed as a number.
2. An initial solution can be easily computed.
3. There are simple adjustment rules that locally change a solution.
4. Every solution can be turned into every other solution by application of these adjustment rules.

As these conditions are by no means very restrictive, it turns out that a surprisingly large number of problems can be successfully solved by simulated annealing.

Let us finally note that computer scientists were not the first to use crystal growing by slow cooling as a model for solving technical problems. If you take this book into your hand, for example, you might notice the accurate

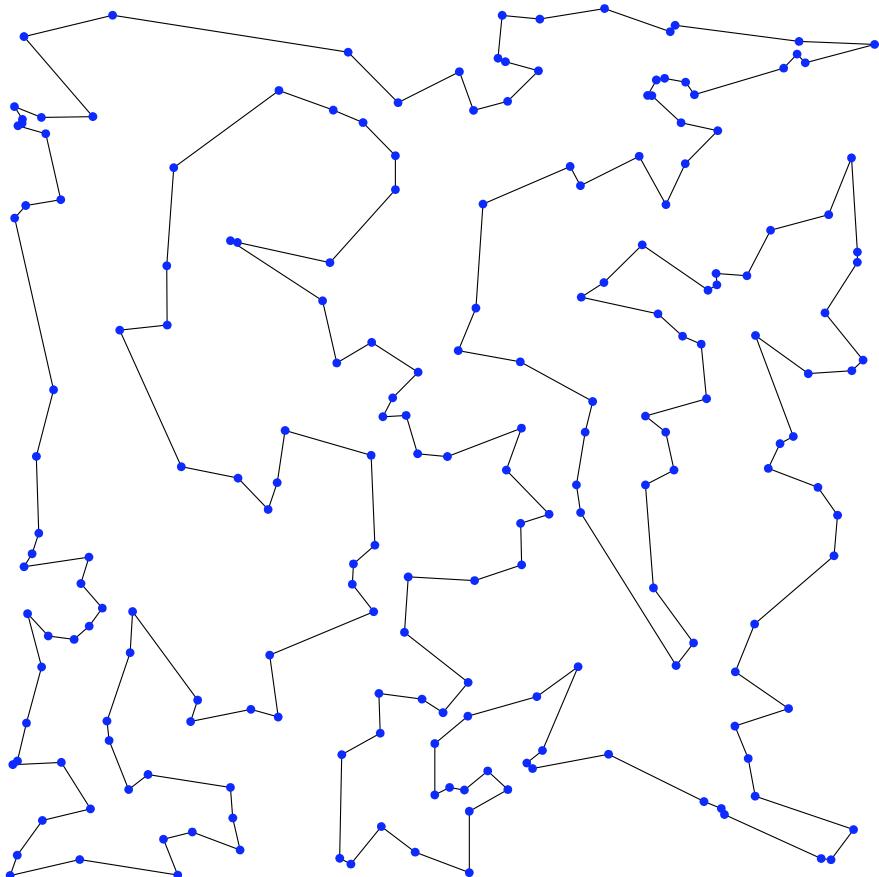


Fig. 41.4. A tour that connects 200 cities

fitting of the pages' edges thanks to a precise cutting machine. If you, however, flip quickly through the pages you might also notice that the edges of the *text* are perfectly aligned. This alignment is due to the perfect adjustment of the pages before binding them into a book. Technically, this is a quite complex task: If you have a pile of loose pages (or cards from a deck), it is quite hard to align their edges perfectly. Even brute force is not very helpful.

The technical solution to this problem is a machine called a *paper jogger*. It perfectly aligns the pages by using strong vibrations. Figure 41.5 shows a small paper jogger that can be operated manually and is used to align exercise sheets coming freshly out of a printing machine. The control dial is used to turn the vibration from strong to gentle: simulated annealing!



Fig. 41.5. The exercise sheets come out of the printing machine. The edges are not aligned. The paper jogger shakes them, and in the end we get a perfectly aligned stack of paper

Further Reading

Simulated annealing was presented in this chapter in a simplified way. Although you can achieve good results by using the method as presented, there are many details whose observance leads to much better results. The entry on simulated annealing in Wikipedia is a good starting point to learn more (http://en.wikipedia.org/wiki/Simulated_annealing). Textbooks on simulated annealing quite often are too specialized for the casual reader. To learn more you can consult a book that contains, besides simulated annealing, other interesting methods for the solution of hard problems:

Juraj Hromkovič. *Algorithmics for Hard Problems (Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics)*. Springer, Heidelberg, 2nd edition, 2002.

You can also find many applets on the Internet for the interactive demonstration of simulated annealing. Just look for them using a search engine.

Author Details

Susanne Albers, Institut für Informatik, Humboldt-Universität zu Berlin, Germany [Chap. 37, Online Algorithms]

Helmut Alt, Institut für Informatik, Freie Universität Berlin, Germany [Chap. 3, Fast Sorting Algorithms; Overview of Part III, Planning, Coordination and Simulation]

Michael Behrisch, Institut für Informatik, Humboldt-Universität zu Berlin, Germany [Chap. 28, Eulerian Circuits]

Rene Beier, Max-Planck-Institut für Informatik, Saarbrücken, Germany [Chap. 39, The Knapsack Problem]

Johannes Blömer, Institut für Informatik, Universität Paderborn, Germany [Chap. 17, How to Share a Secret]

Norbert Blum, Institut für Informatik V, Rheinische Friedrich-Wilhelms-Universität Bonn, Germany [Chap. 31, Dynamic Programming – Evolutionary Distance]

Dirk Bongartz, Gymnasium St. Wolfhelm, Schwalmtal, Germany [Chap. 16, Public-Key Cryptography]

Ulrik Brandes, Fachbereich Informatik & Informationswissenschaft, Universität Konstanz, Germany [Chap. 10, PageRank – What Is Really Relevant in the World-Wide Web]

Volker Claus, Institut für Formale Methoden der Informatik, Universität Stuttgart, Germany [Chap. 35, Marriage Broker]

Amin Coja-Oghlan, Mathematics and Computer Science, University of Warwick, UK [Chap. 28, Eulerian Circuits]

Volker Diekert, Institut für Formale Methoden der Informatik, Universität Stuttgart, Germany [Chap. 35, Marriage Broker]

Martin Dietzfelbinger, Institut für Theoretische Informatik, Fakultät für Informatik und Automatisierung, Technische Universität Ilmenau, Germany [Chap. 19, Fingerprinting; Overview of Part I (Searching and Sorting)]

Michael Dom, Lehrstuhl Theoretische Informatik I, Friedrich-Schiller-Universität Jena, Germany [Chap. 7, Depth-First Search (Ariadne & Co.)]

Gabi Dorfmüller, Fachbereich Informatik & Informationswissenschaft, Universität Konstanz, Germany [Chap. 10, PageRank – What Is Really Relevant in the World-Wide Web]

Arno Eigenwillig, Max-Planck-Institut für Informatik, Saarbrücken, Germany; now at Google, Zürich, Switzerland [Chap. 11, Multiplication of Long Integers – Faster than Long Multiplication]

Friedrich Eisenbrand, Chair of Discrete Optimization, EPFL, Lausanne, Switzerland [Chap. 12, The Euclidean Algorithm]

Jost Enderle, Lehrstuhl für Informatik 9 (Datenmanagement und -exploration), RWTH Aachen University, Germany [Chap. 1, Binary Search]

Thomas Erlebach, Dept. of Computer Science, University of Leicester, UK [Chap. 24, Majority – Who Gets Elected Class Rep.?]

Christoph Freundl, Lehrstuhl für Informatik 10 (Systemsimulation), Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany [Chap. 30, Gauß-Seidel Iterative Method for the Computation of Physical Problems]

Joachim Gehweiler, Heinz Nixdorf Institut und Institut für Informatik, Universität Paderborn, Germany [Chap. 38, Bin Packing or “How Do I Get My Stuff into the Boxes?”]

Robert Görke, Institut für Theoretische Informatik, Karlsruher Institut für Technologie (KIT), Germany [Chap. 34, Maximum Flows – Towards the Stadium During Rush Hour]

Markus Hinkelmann, Institut für Theoretische Informatik, Universität zu Lübeck, Germany [Chap. 14, One-Way Functions. Mind the Trap – Escape Only for the Initiated]

Hagen Höpfner, Mobile Medien, Fakultät Medien, Bauhaus-Universität Weimar, Germany [Chap. 5, Topological Sorting – How Should I Begin to Complete My To Do List?]

Falk Hüffner, Institut für Informatik, Humboldt-Universität zu Berlin, Germany [Chap. 7, Depth-First Search (Ariadne & Co.)]

Tim Jonischkat, Institut für Informatik und Wirtschaftsinformatik, Universität Duisburg-Essen, Germany [Chap. 25, Random Numbers – How Can We Create Randomness in Computers?]

Tom Kamphans, Institut für Betriebssysteme und Rechnerverbund (IBR), Technische Universität Braunschweig, Germany [Chap. 8, Pledge’s Algorithm]

Rolf Klein, Lehrstuhl für Informatik I, Rheinische Friedrich-Wilhelms-Universität Bonn, Germany [Chap. 8, Pledge’s Algorithm]

Sigrid Knust, Institut für Informatik, Universität Osnabrück, Germany [Chap. 27, Scheduling of Tournaments or Sports Leagues]

Leif Kobbelt, Lehrstuhl für Informatik 8 (Computergraphik und Multimedia), RWTH Aachen University, Germany [Chap. 29, High-Speed Circles]

Jochen Könemann, Dept. of Combinatorics and Optimization, University of Waterloo, Canada [Chap. 26, Winning Strategies for a Matchstick Game]

Wolfgang P. Kowalk, Dept. für Informatik (Rechnernetze), Carl-von-Ossietzky-Universität Oldenburg, Germany [Chap. 2, Insertion Sort]

Matthias Kretschmer, Institut für Informatik V, Rheinische Friedrich-Wilhelms-Universität Bonn, Germany [Chap. 31, Dynamic Programming – Evolutionary Distance]

Peter Liske, Institut für Informatik, Humboldt-Universität zu Berlin, Germany [Chap. 28, Eulerian Circuits]

Steffen Mecke, Institut für Theoretische Informatik, Karlsruher Institut für Technologie (KIT), Germany [Chap. 34, Maximum Flows – Towards the Stadium During Rush Hour]

Kurt Mehlhorn, Algorithmen und Komplexität, Max-Planck-Institut für Informatik, Saarbrücken, Germany [Chap. 11, Multiplication of Long Integers – Faster than Long Multiplication]

Friedhelm Meyer auf der Heide, Heinz Nixdorf Institut und Institut für Informatik, Universität Paderborn, Germany [Chap. 38, Bin Packing or “How Do I Get My Stuff into the Boxes?”]

Michael Mitzenmacher, School of Engineering and Applied Sciences, Harvard University, USA [Chap. 21, Codes – Protecting Data Against Errors and Loss]

Rolf Möhring, Institut für Mathematik, Technische Universität Berlin, Germany [Chap. 13, The Sieve of Eratosthenes – How Fast Can We Compute a Prime Number Table?]

Bruno Müller-Clostermann, Lehrstuhl für Praktische Informatik (Systemmodellierung), Institut für Informatik und Wirtschaftsinformatik, Universität Duisburg-Essen, Germany [Chap. 25, Random Numbers – How Can We Create Randomness in Computers?]

Stefan Näher, Fachbereich IV – Informatik, Universität Trier, Germany [Chap. 40, The Travelling Salesman Problem]

Markus E. Nebel, Fachbereich Informatik, AG Algorithmen und Komplexität, TU Kaiserslautern, Germany [Chap. 6, Searching Texts – But Fast! The Boyer-Moore-Horspool Algorithm]

Rolf Niedermeier, Institut für Softwaretechnik und Theoretische Informatik, Technische Universität Berlin, Germany [Chap. 7, Depth-First Search (Ariadne & Co.)]

Martin Oellrich, Fachbereich II – Mathematik, Physik, Chemie, Beuth Hochschule für Technik Berlin, Germany [Chap. 13, The Sieve of Eratosthenes – How Fast Can We Compute a Prime Number Table?]

Holger Petersen, Institut für Formale Methoden der Informatik, Universität Stuttgart, Germany [Chap. 35, Marriage Broker]

Rüdiger Reischuk, Institut für Theoretische Informatik, Universität zu Lübeck, Germany [Chap. 14, One-Way Functions. Mind the Trap – Escape Only for the Initiated; Overview of Part III, Planning, Coordination and Simulation]

Peter Rossmanith, Lehr- und Forschungsgebiet Theoretische Informatik, RWTH Aachen University, Germany [Chap. 41, Simulated Annealing]

Ulrich Rüde, Lehrstuhl für Informatik 10 (Systemsimulation), Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany [Chap. 30, Gauß-Seidel Iterative Method for the Computation of Physical Problems]

Peter Sanders, Institut für Theoretische Informatik, Karlsruher Institut für Technologie, Germany [Chap. 32, Shortest Paths]

Christian Scheideler, Institut für Informatik, Universität Paderborn, Germany [Chap. 22, Broadcasting – How Can I Quickly Disseminate Information?; Overview of Part I, Searching and Sorting]

Christian Schindelhauer, Rechnernetze und Telematik, Institut für Informatik, Albert-Ludwigs-Universität Freiburg, Germany [Chap. 20, Hashing]

Holger Schlingloff, Institut für Informatik, Humboldt-Universität zu Berlin, Germany [Chap. 9, Cycles in Graphs]

Swen Schmelzer, Lehrstuhl für Informations- und Kodierungstheorie, Albert-Ludwigs-Universität Freiburg, Germany [Chap. 37, Online Algorithms]

Lothar Schmitz, Institut für Softwaretechnologie, Fakultät für Informatik, UniBw München, Germany [Chap. 23, Converting Numbers into English Words]

Thomas Seidl, Lehrstuhl für Informatik 9 (Datenmanagement und -exploration), RWTH Aachen University, Germany [Chap. 1, Binary Search]

Dominik Sibbing, Lehrstuhl für Informatik 8 (Computergraphik und Multimedia), RWTH Aachen University, Germany [Chap. 29, High-Speed Circles]

Detlef Sieling, Lehrstuhl Informatik 2 (Effiziente Algorithmen und Komplexitätstheorie), Technische Universität Dortmund, Germany [Chap. 18, Playing Poker by Email]

Johannes Singler, Institut für Theoretische Informatik, Karlsruher Institut für Technologie, Germany [Chap. 32, Shortest Paths]

Katharina Skutella, Institut für Mathematik, Technische Universität Berlin, Germany [Chap. 33, Minimum Spanning Trees]

Martin Skutella, Institut für Mathematik, Technische Universität Berlin, Germany [Chap. 33, Minimum Spanning Trees]

Till Tantau, Institut für Theoretische Informatik, Universität zu Lübeck, Germany [Chap. 15, The One-Time Pad Algorithm – The Simplest and Most Secure Way to Keep Secrets]

Walter Unger, Lehrstuhl für Informatik 1, Algorithmen und Komplexität, RWTH Aachen University, Germany [Chap. 16, Public-Key Cryptography]

Berthold Vöcking, Lehrstuhl für Informatik 1, Algorithmen und Komplexität, RWTH Aachen University, Germany [Chap. 39, The Knapsack Problem; Overview of Part II, Arithmetic and Encryption]

Heribert Vollmer, Institut für Theoretische Informatik, Leibniz Universität Hannover, Germany [Overview of Part IV, Optimization]

Dorothea Wagner, Institut für Theoretische Informatik, Karlsruher Institut für Technologie (KIT), Germany [Chap. 34, Maximum Flows – Towards the Stadium During Rush Hour; Overview of Part IV, Optimization]

Rolf Wanka, Lehrstuhl für Informatik 12 (Hardware-Software-Co-Design), Universität Erlangen-Nürnberg, Germany [Chap. 4, Parallel Sorting – The Need for Speed]

Emo Welzl, Institut für Theoretische Informatik, ETH Zürich, Switzerland [Chap. 36, The Smallest Enclosing Circle – A Contribution to Democracy from Switzerland?]