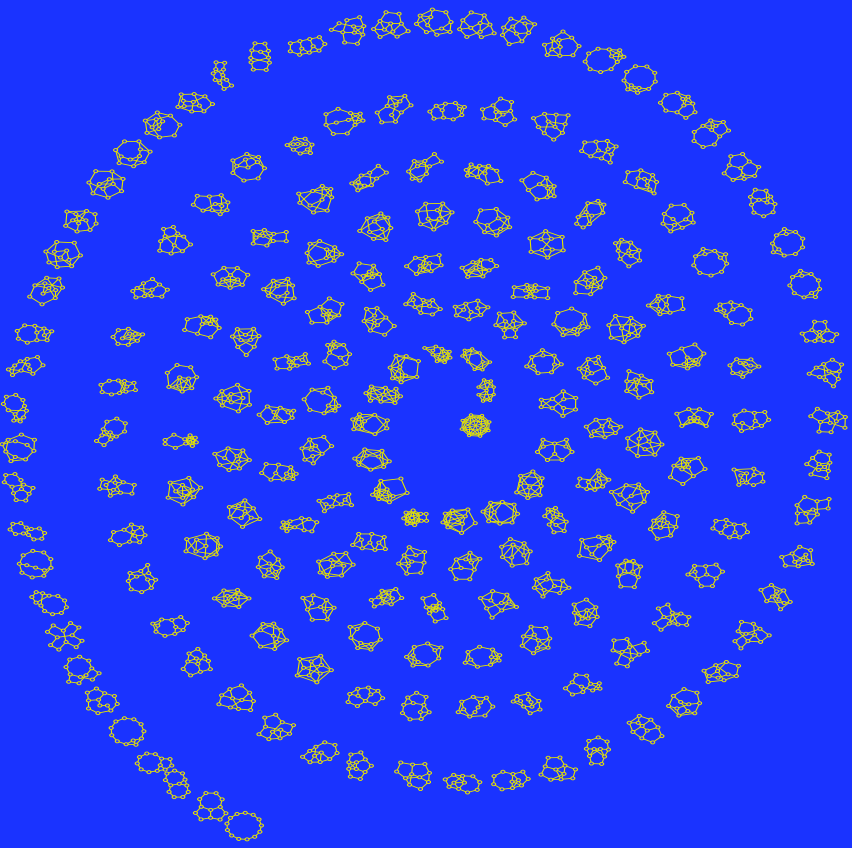
Introduction to Algorithms and Data Structures

Michael J. Dinneen Georgy Gimel'farb Mark C. Wilson



(Fourth edition)

Contents

List of Figures

List of Tables

Introduction to the Fourth Edition

The fourth edition follows the third edition, while incorporates fixes for the errata discovered in the third edition.

The textbook's coverpage displays the complete list of 88 connected forbidden minors for graphs with vertex cover at most 6, computed by M. J. Dinneen and L. Xiong in 2001.

Michael J. Dinneen
Georgy Gimel'farb
Mark C. Wilson

Department of Computer Science
University of Auckland
Auckland, New Zealand

{mjd, georgy, mcw}@cs.auckland.ac.nz

February 2016
(Minor revisions December 2017)

Introduction to the Third Edition

The focus for this third edition has been to make an electronic version that students can read on the tablets and laptops that they bring to lectures. The main changes from the second edition are:

- Incorporated fixes for the errata discovered in the second edition.
- Made e-book friendly by adding cross referencing links to document elements and formatted pages to fit on most small screen tablets (i.e., width of margins minimized).
- Limited material for University of Auckland's CompSci 220 currently taught topics. Removed the two chapters on formal languages (Part III) and truncated the book title.
-

Michael J. Dinneen
Georgy Gimel'farb
Mark C. Wilson

Department of Computer Science
University of Auckland
Auckland, New Zealand

{mjd, georgy, mcw}@cs.auckland.ac.nz

March 2013

Introduction to the Second Edition

Writing a second edition is a thankless task, as is well known to authors. Much of the time is spent on small improvements that are not obvious to readers. We have taken considerable efforts to correct a large number of errors found in the first edition, and to improve explanation and presentation throughout the book, while retaining the philosophy behind the original. As far as material goes, the main changes are:

- more exercises and solutions to many of them;
- a new section on maximum matching (Section 5.9);
- a new section on string searching (Part III);
- a Java graph library updated to Java 1.6 and freely available for download.

The web site `http://www.cs.auckland.ac.nz/textbookCS220/` for the book provides additional material including source code. Readers finding errors are encouraged to contact us after viewing the errata page at this web site.

In addition to the acknowledgments in the first edition, we thank Sonny Datt for help with updating the Java graph library, Andrew Hay for help with exercise solutions and Cris Calude for comments. Rob Randtoul (`PlasmaDesign.co.uk`) kindly allowed us to use his cube artwork for the book's cover. Finally, we thank

MJD all students who have struggled to learn from the first edition and have given us feedback, either positive or negative;

GLG my wife Natasha and all the family for their permanent help and support;

MCW my wife Golbon and sons Yusef and Yahya, for their sacrifices during the writing of this book, and the joy they bring to my life even in the toughest times.

31 October 2008

Introduction to the First Edition

This book is an expanded, and, we hope, improved version of the coursebook for the course COMPSCI 220 which we have taught several times in recent years at the University of Auckland.

We have taken the step of producing this book because there is no single text available that covers the syllabus of the above course at the level required. Indeed, we are not aware of any other book that covers all the topics presented here. Our aim has been to produce a book that is straightforward, concise, and inexpensive, and suitable for self-study (although a teacher will definitely add value, particularly where the exercises are concerned). It is an introduction to some key areas at the theoretical end of computer science, which nevertheless have many practical applications and are an essential part of any computer science student's education.

The material in the book is all rather standard. The novelty is in the combination of topics and some of the presentation. Part I deals with the basics of algorithm analysis, tools that predict the performance of programs without wasting time implementing them. Part II covers many of the standard fast graph algorithms that have applications in many different areas of computer science and science in general. Part III introduces the theory of formal languages, shifting the focus from what can be computed quickly to what families of strings can be recognized easily by a particular type of machine.

The book is designed to be read cover-to-cover. In particular Part I should come first. However, one can read Part III before Part II with little chance of confusion.

To make best use of the book, one must do the exercises. They vary in difficulty from routine to tricky. No solutions are provided. This policy may be changed in a later edition.

The prerequisites for this book are similar to those of the above course, namely two semesters of programming in a structured language such as Java (currently used at Auckland). The book contains several appendices which may fill in any gaps in

the reader's background.

A limited bibliography is given. There are so many texts covering some of the topics here that to list all of them is pointless. Since we are not claiming novelty of material, references to research literature are mostly unnecessary and we have omitted them. More advanced books (some listed in our bibliography) can provide more references as a student's knowledge increases.

A few explanatory notes to the reader about this textbook are in order.

We describe algorithms using a pseudocode similar to, but not exactly like, many structured languages such as Java or C++. Loops and control structures are indented in fairly traditional fashion. We do not formally define our pseudocode of comment style (this might make an interesting exercise for a reader who has mastered Part III).

We make considerable use of the idea of ADT (abstract data type). An ***abstract data type*** is a mathematically specified collection of objects together with operations that can be performed on them, subject to certain rules. An ADT is completely independent of any computer programming implementation and is a mathematical structure similar to those studied in pure mathematics. Examples in this book include digraphs and graphs, along with queues, priority queues, stacks, and lists. A ***data structure*** is simply a higher level entity composed of the elementary memory addresses related in some way. Examples include arrays, arrays of arrays (matrices), linked lists, doubly linked lists, etc.

The difference between a data structure and an abstract data type is exemplified by the difference between a standard linear array and what we call a list. An array is a basic data structure common to most programming languages, consisting of contiguous memory addresses. To find an element in an array, or insert an element, or delete an element, we directly use the address of the element. There are no secrets in an array. By contrast, a list is an ADT. A list is specified by a set S of elements from some universal set U, together with operations `insert`, `delete`, `size`, `isEmpty` and so on (the exact definition depends on who is doing the defining). We denote the result of the operation as S.`isEmpty`(), for example. The operations must satisfy certain rules, for example: S.`isEmpty`() returns a boolean value TRUE or FALSE; S.`insert`(x,r) requires that x belong to U and r be an integer between 0 and S.`size`(), and returns a list; for any admissible x and r we have S.`isEmpty`(S.`insert`(x,r)) = FALSE, etc. We are not interested in how the operations are to be carried out, only in what they do. Readers familiar with languages that facilitate object-based and object-oriented programming will recognize ADTs as, essentially, what are called classes in Java or C++.

A list can be implemented using an array (to be more efficient, we would also have an extra integer variable recording the array size). The `insert` operation, for example, can be achieved by accessing the correct memory address of the r-th element of the array, allocating more space at the end of the array, shifting along some elements by one, and assigning the element to be inserted to the address vacated by the shifting. We would also update the size variable by 1. These details are unimportant

in many programming applications. However they are somewhat important when discussing complexity as we do in Part I. While ADTs allow us to concentrate on algorithms without worrying about details of programming implementation, we cannot ignore data structures forever, simply because some implementations of ADT operations are more efficient than others.

In summary, we use ADTs to sweep programming details under the carpet as long as we can, but we must face them eventually.

A book of this type, written by three authors with different writing styles under some time pressure, will inevitably contain mistakes. We have been helped to minimize the number of errors by the student participants in the COMPSCI 220 coursebook error-finding competition, and our colleagues Joshua Arulanandham and Andre Nies, to whom we are very grateful.

Our presentation has benefitted from the input of our colleagues who have taught COMPSCI 220 in the recent and past years, with special acknowledgement due to John Hamer and the late Michael Lennon.

10 February 2004

Part I

Introduction to Algorithm Analysis

Chapter 1

What is Algorithm Analysis?

Algorithmic problems are of crucial importance in modern life. Loosely speaking, they are precisely formulated problems that can be solved in a step-by-step, mechanical manner.

Definition 1.1 (informal)**.** An ***algorithm*** is a list of unambiguous rules that specify successive steps to solve a problem. A ***computer program*** is a clearly specified sequence of computer instructions implementing the algorithm.

For example, a sufficiently detailed recipe for making a cake could be thought of as an algorithm. Problems of this sort are not normally considered as part of computer science. In this book, we deal with algorithms for problems of a more abstract nature. Important examples, all discussed in this book, and all with a huge number of practical applications, include: sorting a database, finding an entry in a database, finding a pattern in a text document, finding the shortest path through a network, scheduling tasks as efficiently as possible, finding the median of a statistical sample.

Strangely enough, it is very difficult to give simple precise mathematical definitions of algorithms and programs. The existing very deep general definitions are too complex for our purposes. We trust that the reader of this book will obtain a good idea of what we mean by algorithm from the examples in this and later chapters.

We often wish to compare different algorithms for the same problem, in order to select the one best suited to our requirements. The main features of interest are: whether the algorithm is ***correct*** (does it solve the problem for all legal inputs), and how *efficient* it is (how much time, memory storage, or other resources it uses).

The same algorithm can be implemented by very different programs written in different programming languages, by programmers of different levels of skill, and

then run on different computer platforms under different operating systems. In searching for the best algorithm, general features of algorithms must be isolated from peculiarities of particular platforms and programs.

To analyse computer algorithms in practice, it is usually sufficient to first specify elementary operations of a "typical" computer and then represent each algorithm as a sequence of those operations.

Most modern computers and languages build complex programs from ordinary arithmetic and logical operations such as standard unary and binary arithmetic operations (negation, addition, subtraction, multiplication, division, modulo operation, or assignment), Boolean operations, binary comparisons ("equals", "less than", or "greater than"), branching operations, and so on. It is quite natural to use these basic computer instructions as algorithmic operations, which we will call ***elementary operations***.

It is not always clear what should count as an elementary operation. For example, addition of two 64-bit integers should definitely count as elementary, since it can be done in a fixed time for a given implementation. But for some applications, such as cryptography, we must deal with much larger integers, which must be represented in another way. Addition of "big" integers takes a time roughly proportional to the size of the integers, so it is not reasonable to consider it as elementary. From now on we shall ignore such problems. For most of the examples in this introductory book they do not arise. However, they must be considered in some situations, and this should be borne in mind.

Definition 1.2 (informal)**.** The ***running time*** (or computing time) of an algorithm is the number of its elementary operations.

The actual execution time of a program implementing an algorithm is roughly proportional to its running time, and the scaling factor depends only on the particular implementation (computer, programming language, operating system, and so on).

The memory space required for running an algorithm depends on how many individual variables (input, intermediate, and output data) are involved simultaneously at each computing step. Time and space requirements are almost always independent of the programming language or style and characterise the algorithm itself. From here on, we will measure effectiveness of algorithms and programs mostly in terms of their time requirements. Any real computer has limits on the size and the number of data items it can handle.

1.1 Efficiency of algorithms: first examples

If the same problem can be solved by different algorithms, then all other things being equal, the most efficient algorithm uses least computational resources. The theory of algorithms in modern computer science clarifies basic algorithmic notions

```
algorithm linearSum
    Input: array a[0..n − 1]
begin
    s ← 0
    for i ← 0 step i ← i + 1 until n − 1 do
        s ← s + a[i]
    end for
    return s
end
```

Figure 1.1: Linear-time algorithm to sum an array.

such as provability, correctness, complexity, randomness, or computability. It studies whether there exist any algorithms for solving certain problems, and if so, how fast can they be. In this book, we take only a few small steps into this domain.

To search for the most efficient algorithm, one should mathematically prove correctness of and determine time/space resources for each algorithm as explicit functions of the size of input data to process. For simplicity of presentation in this book, we sometimes skip the first step (proof of correctness), although it is very important. The focus of this chapter is to introduce methods for estimating the resource requirements of algorithms.

Example 1.3 (Sum of elements of an array)**.** Let a denote an array of integers where the sum $s = \sum_{i=0}^{n-1} a[i]$ is required. To get the sum s, we have to repeat n times the same elementary operations (fetching from memory and adding a number). Thus, running time $T(n)$ is proportional to, or *linear* in n: $T(n) = cn$. Such algorithms are also called ***linear algorithms***. The unknown factor c depends on a particular computer, programming language, compiler, operating system, etc. But the relative change in running time is just the same as the change in the data size: $T(10) = 10T(1)$, or $T(1000) = 1000T(1)$, or $T(1000) = 10T(100)$. The linear algorithm in Figure 1.1 implements a simple loop.

Example 1.4 (Sums of subarrays)**.** The problem is to compute, for each subarray $a[j..j+m-1]$ of size m in an array a of size n, the partial sum of its elements $s[j] = \sum_{k=0}^{m-1} a[j+k]$; $j = 0, \ldots, n-m$. The total number of these subarrays is $n-m+1$. At first glance, we need to compute $n-m+1$ sums, each of m items, so that the running time is proportional to $m(n-m+1)$. If m is fixed, the time depends still linearly on n.

But if m is growing with n as a fraction of n, such as $m = \frac{n}{2}$, then $T(n) = c\frac{n}{2}\left(\frac{n}{2}+1\right) = 0.25cn^2 + 0.5cn$. The relative weight of the linear part, $0.5cn$, decreases quickly with respect to the quadratic one as n increases. For example, if $T(n) = 0.25n^2 + 0.5n$, we see in the last column of Table 1.1 the rapid decrease of the ratio of the two terms.

Table 1.1: Relative growth of linear and quadratic terms in an expression.

n	$T(n)$	$0.25n^2$	$0.5n$	
			value	% of quadratic term
10	30	25	5	20.0
50	650	625	25	4.0
100	2550	2500	50	2.0
500	62750	62500	250	0.4
1000	250500	250000	500	0.2
5000	6252500	6250000	2500	0.04

Thus, for large n only the quadratic term becomes important and the running time is roughly proportional to n^2, or is quadratic in n. Such algorithms are sometimes called ***quadratic algorithms*** in terms of relative changes of running time with respect to changes of the data size: if $T(n) \approx cn^2$ then $T(10) \approx 100T(1)$, or $T(100) \approx 10000T(1)$, or $T(100) \approx 100T(10)$.

```
algorithm slowSums
    Input: array a[0..2m − 1]
begin
    array s[0..m]
    for i ← 0 to m do
        s[i] ← 0
        for j ← 0 to m − 1 do
            s[i] ← s[i] + a[i + j]
        end for
    end for
    return s
end
```

Figure 1.2: Quadratic time algorithm to compute sums of an array.

The "brute-force" quadratic algorithm has two nested loops (see Figure 1.2). Let us analyse it to find out whether it can be simplified. It is easily seen that repeated computations in the innermost loop are unnecessary. Two successive sums $s[i]$ and $s[i-1]$ differ only by two elements: $s[i] = s[i-1] + a[i+m-1] - a[i-1]$. Thus we need not repeatedly add m items together after getting the very first sum $s[0]$. Each next sum is formed from the current one by using only two elementary operations (addition and subtraction). Thus $T(n) = c(m + 2(n-m)) = c(2n-m)$. In the first paren-

theses, the first term m relates to computing the first sum $s[0]$, and the second term $2(n-m)$ reflects that $n-m$ other sums are computed with only two operations per sum. Therefore, the running time for this better organized computation is always linear in n for each value m, either fixed or growing with n. The time for computing all the sums of the contiguous subsequences is less than twice that taken for the single sum of all n items in Example 1.3

The linear algorithm in Figure 1.3 excludes the innermost loop of the quadratic algorithm. Now two simple loops, doing m and $2(n-m)$ elementary operations, respectively, replace the previous nested loop performing $m(n-m+1)$ operations.

```
algorithm fastSums
    Input: array a[0..2m − 1]
begin
    array s[0..m]
    s[0] ← 0
    for j ← 0 to m − 1 do
        s[0] ← s[0] + a[j]
    end for
    for i ← 1 to m do
        s[i] ← s[i − 1] + a[i + m − 1] − a[i − 1]
    end for
    return s;
end
```

Figure 1.3: Linear-time algorithm to compute sums of an array.

Such an outcome is typical for algorithm analysis. In many cases, a careful analysis of the problem allows us to replace a straightforward "brute-force" solution with much more effective one. But there are no "standard" ways to reach this goal. To exclude unnecessary computation, we have to perform a thorough investigation of the problem and find hidden relationships between the input data and desired outputs. In so doing, we should exploit all the tools we have learnt. This book presents many examples where analysis tools are indeed useful, but knowing how to analyse and solve each particular problem is still close to an art. The more examples and tools are mastered, the more the art is learnt.

Exercises

Exercise 1.1.1. A quadratic algorithm with processing time $T(n) = cn^2$ uses 500 elementary operations for processing 10 data items. How many will it use for processing 1000 data items?

Exercise 1.1.2. Algorithms **A** and **B** use exactly $T_A(n) = c_A n \lg n$ and $T_B(n) = c_B n^2$ elementary operations, respectively, for a problem of size n. Find the fastest algorithm for processing $n = 2^{20}$ data items if **A** and **B** spend 10 and 1 operations, respectively, to process $2^{10} \equiv 1024$ items.

1.2 Running time for loops and other computations

The above examples show that running time depends considerably on how deeply the loops are nested and how the loop control variables are changing. Suppose the control variables change linearly in n, that is, increase or decrease by constant steps. If the number of elementary operations in the innermost loop is constant, the nested loops result in polynomial running time $T(n) = cn^k$ where k is the highest level of nesting and c is some constant. The first three values of k have special names: ***linear time***, ***quadratic time***, and ***cubic time*** for $k = 1$ (a single loop), $k = 2$ (two nested loops), and $k = 3$ (three nested loops), respectively.

When loop control variables change non-linearly, the running time also varies non-linearly with n.

Example 1.5. An exponential change $i = 1, k, k^2, \ldots, k^{m-1}$ of the control variable in the range $1 \le i \le n$ results in ***logarithmic time*** for a simple loop. The loop executes m iterations such that $k^{m-1} \le n < k^m$. Thus, $m - 1 \le \log_k n < m$, and $T(n) = c\lceil \log_k n \rceil$.

Additional conditions for executing inner loops only for special values of the outer variables also decrease running time.

Example 1.6. Let us roughly estimate the running time of the following nested loops:

```
m ← 2
for j ← 1 to n do
    if j = m then
        m ← 2m
        for i ← 1 to n do
            ... | constant number of elementary operations |
        end for
    end if
end for
```

The inner loop is executed k times for $j = 2, 4, \ldots, 2^k$ where $k < \lg n \le k + 1$. The total time for the elementary operations is proportional to kn, that is, $T(n) = n\lfloor \lg n \rfloor$.

Conditional and switch operations like **if** {*condition*} **then** {*constant running time* T_1} **else** {*constant running time* T_2} involve relative frequencies of the groups of computations. The running time T satisfies $T = f_{\text{true}} T_1 + (1 - f_{\text{true}}) T_2 < \max\{T_1, T_2\}$ where f_{true} is the relative frequency of the true condition value in the if-statement.

The running time of a function or method call is $T = \sum_{i=1}^{k} T_i$ where T_i is the running time of statement i of the function and k is the number of statements.

Exercises

Exercise 1.2.1. Is the running time quadratic or linear for the nested loops below?

```
m ← 1
for j ← 1 step j ← j + 1 until n do
    if j = m then m ← m · (n − 1)
        for i ← 0 step i ← i + 1 until n − 1 do
            ... | constant number of elementary operations
        end for
    end if
end for
```

Exercise 1.2.2. What is the running time for the following code fragment as a function of n?

```
for i ← 1 step i ← 2 ∗ i while i < n do
    for j ← 1 step j ← 2 ∗ j while j < n do
        if j = 2 ∗ i
            for k = 0 step k ← k + 1 while k < n do
                ... | constant number of elementary operations
            end for
        else
            for k ← 1 step k ← 3 ∗ k while k < n do
                ... | constant number of elementary operations
            end for
        end if
    end for
end for
```

1.3 "Big-Oh", "Big-Theta", and "Big-Omega" tools

Two simple concepts separate properties of an algorithm itself from properties of a particular computer, operating system, programming language, and compiler used for its implementation. The concepts, briefly outlined earlier, are as follows:

- The ***input data size***, or the number n of individual data items in a single data instance to be processed when solving a given problem. Obviously, how to measure the data size depends on the problem: n means the number of items to sort (in sorting applications), number of nodes (vertices) or arcs (edges) in

graph algorithms, number of picture elements (pixels) in image processing, length of a character string in text processing, and so on.

- The number of elementary operations taken by a particular algorithm, or its running time. We assume it is a function $f(n)$ of the input data size n. The function depends on the elementary operations chosen to build the algorithm.

The running time of a program which implements the algorithm is $cf(n)$ where c is a constant factor depending on a computer, language, operating system, and compiler. Even if we don't know the value of the factor c, we are able to answer the important question: *if the input size increases from $n = n_1$ to $n = n_2$, how does the relative running time of the program change, all other things being equal?* The answer is obvious: the running time increases by a factor of $\frac{T(n_2)}{T(n_1)} = \frac{cf(n_2)}{cf(n_1)} = \frac{f(n_2)}{f(n_1)}$.

As we have already seen, the approximate running time for large input sizes gives enough information to distinguish between a good and a bad algorithm. Also, the constant c above can rarely be determined. We need some mathematical notation to avoid having to say "*of the order of* ..." or "*roughly proportional to* ...", and to make this intuition precise.

The standard mathematical tools "*Big Oh*" (O), "*Big Theta*" (Θ), and "*Big Omega*" (Ω) do precisely this.

Note. Actually, the above letter O is a capital "omicron" (all letters in this notation are Greek letters). However, since the Greek omicron and the English "O" are indistinguishable in most fonts, we read $O()$ as "Big Oh" rather than "Big Omicron".

The algorithms are analysed under the following assumption: *if the running time of an algorithm as a function of n differs only by a constant factor from the running time for another algorithm, then the two algorithms have essentially the same time complexity.* Functions that measure running time, $T(n)$, have nonnegative values because time is nonnegative, $T(n) \geq 0$. The integer argument n (data size) is also nonnegative.

Definition 1.7 (Big Oh)**.** Let $f(n)$ and $g(n)$ be nonnegative-valued functions defined on nonnegative integers n. Then $g(n)$ is $O(f(n))$ (read "$g(n)$ is Big Oh of $f(n)$") iff there exists a positive real constant c and a positive integer n_0 such that $g(n) \leq cf(n)$ for all $n > n_0$.

Note. We use the notation "***iff***" as an abbreviation of "if and only if".

In other words, if $g(n)$ is $O(f(n))$ then an algorithm with running time $g(n)$ runs for large n at least as fast, to within a constant factor, as an algorithm with running time $f(n)$. Usually the term "***asymptotically***" is used in this context to describe behaviour of functions for sufficiently large values of n. This term means that $g(n)$ for large n may approach closer and closer to $c \cdot f(n)$. Thus, $O(f(n))$ specifies an ***asymptotic upper bound***.

Note. Sometimes the "Big Oh" property is denoted $g(n) = O(f(n))$, but we should not assume that the function $g(n)$ is equal to something called "Big Oh" of $f(n)$. This notation really means $g(n) \in O(f(n))$, that is, $g(n)$ is a member of the set $O(f(n))$ of functions which are increasing, in essence, with the same or lesser rate as n tends to infinity ($n \to \infty$). In terms of graphs of these functions, $g(n)$ is $O(f(n))$ iff there exists a constant c such that the graph of $g(n)$ is always below or at the graph of $cf(n)$ after a certain point, n_0.

Example 1.8. Function $g(n) = 100\log_{10} n$ in Figure 1.4 is $O(n)$ because the graph $g(n)$ is always below the graph of $f(n) = n$ if $n > 238$ or of $f(n) = 0.3n$ if $n > 1000$, etc.

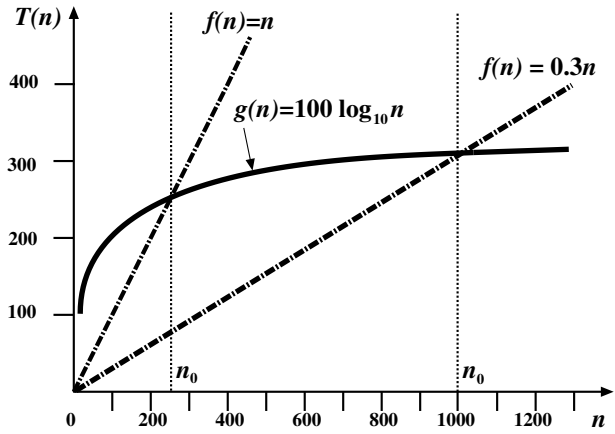


Figure 1.4: "Big Oh" property: $g(n)$ is $O(n)$.

Definition 1.9 (Big Omega)**.** The function $g(n)$ is $\Omega(f(n))$ iff there exists a positive real constant c and a positive integer n_0 such that $g(n) \geq cf(n)$ for all $n > n_0$.

"Big Omega" is complementary to "Big Oh" and generalises the concept of "lower bound" ($\geq$) in the same way as "Big Oh" generalises the concept of "upper bound" ($\leq$): if $g(n)$ is $O(f(n))$ then $f(n)$ is $\Omega(g(n))$, and vice versa.

Definition 1.10 (Big Theta)**.** The function $g(n)$ is $\Theta(f(n))$ iff there exist two positive real constants c_1 and c_2 and a positive integer n_0 such that $c_1 f(n) \leq g(n) \leq c_2 f(n)$ for all $n > n_0$.

Whenever two functions, $f(n)$ and $g(n)$, are actually of the same order, $g(n)$ is $\Theta(f(n))$, they are each "Big Oh" of the other: $f(n)$ is $O(g(n))$ and $g(n)$ is $O(f(n))$. In other words, $f(n)$ is both an asymptotic upper and lower bound for $g(n)$. The "Big

Theta" property means $f(n)$ and $g(n)$ have asymptotically tight bounds and are in some sense equivalent for our purposes.

In line with the above definitions, $g(n)$ is $O(f(n))$ iff $g(n)$ grows *at most* as fast as $f(n)$ to within a constant factor, $g(n)$ is $\Omega(f(n))$ iff $g(n)$ grows *at least* as fast as $f(n)$ to within a constant factor, and $g(n)$ is $\Theta(f(n))$ iff $g(n)$ and $f(n)$ grow *at the same rate* to within a constant factor.

"Big Oh", "Big Theta", and "Big Omega" notation formally capture two crucial ideas in comparing algorithms: the exact function, g, is not very important because it can be multiplied by any arbitrary positive constant, c, and the relative behaviour of two functions is compared only asymptotically, for large n, but not near the origin where it may make no sense. Of course, if the constants involved are very large, the asymptotic behaviour loses practical interest. In most cases, however, the constants remain fairly small.

In analysing running time, "Big Oh" $g(n) \in O(f(n))$, "Big Omega" $g(n) \in \Omega(f(n))$, and "Big Theta" $g(n) \in \Theta(f(n))$ definitions are mostly used with $g(n)$ equal to "exact" running time on inputs of size n and $f(n)$ equal to a rough approximation to running time (like $\log n$, n, n^2, and so on).

To prove that some function $g(n)$ is $O(f(n))$, $\Omega(f(n))$, or $\Theta(f(n))$ using the definitions we need to find the constants c, n_0 or c_1, c_2, n_0 specified in Definitions 1.7, 1.9, 1.10. Sometimes the proof is given only by a chain of inequalities, starting with $f(n)$. In other cases it may involve more intricate techniques, such as mathematical induction. Usually the manipulations are quite simple. To prove that $g(n)$ is *not* $O(f(n))$, $\Omega(f(n))$, or $\Theta(f(n))$ we have to show the desired constants do not exist, that is, their assumed existence leads to a contradiction.

Example 1.11. To prove that linear function $g(n) = an + b$; $a > 0$, is $O(n)$, we form the following chain of inequalities: $g(n) \le an + |b| \le (a + |b|)n$ for all $n \ge 1$. Thus, Definition 1.7 with $c = a + |b|$ and $n_0 = 1$ shows that $an + b$ is $O(n)$.

"Big Oh" hides constant factors so that both $10^{-10}n$ and $10^{10}n$ are $O(n)$. It is pointless to write something like $O(2n)$ or $O(an + b)$ because this still means $O(n)$. Also, only the dominant terms as $n \to \infty$ need be shown as the argument of "Big Oh", "Big Omega", or "Big Theta".

Example 1.12. The polynomial $P_5(n) = a_5n^5 + a_4n^4 + a_3n^3 + a_2n^2 + a_1n + a_0$; $a_5 > 0$, is $O(n^5)$ because $P_5(n) \le (a_5 + |a_4| + |a_3| + |a_2| + |a_1| + |a_0|)n^5$ for all $n \ge 1$.

Example 1.13. The exponential function $g(n) = 2^{n+k}$, where k is a constant, is $O(2^n)$ because $2^{n+k} = 2^k 2^n$ for all n. Generally, m^{n+k} is $O(l^n)$; $l \ge m > 1$, because $m^{n+k} \le l^{n+k} = l^k l^n$ for any constant k.

Example 1.14. For each $m > 1$, the logarithmic function $g(n) = \log_m(n)$ has the same rate of increase as $\lg(n)$ because $\log_m(n) = \log_m(2) \lg(n)$ for all $n > 0$. Therefore we may

omit the logarithm base when using the "Big-Oh" and "Big Theta" notation: $\log_m n$ is $\Theta(\log n)$.

Rules for asymptotic notation

Using the definition to prove asymptotic relationships between functions is hard work. As in calculus, where we soon learn to use various rules (product rule, chain rule, ...) rather than the definition of derivative, we can use some simple rules to deduce new relationships from old ones.

We present rules for "Big Oh"—similar relationships hold for "Big Omega" and "Big Theta".

We will consider the features both informally and formally using the following notation. Let x and y be functions of a nonnegative integer n. Then $z = x + y$ and $z = xy$ denote the sum of the functions, $z(n) = x(n) + y(n)$, and the product function: $z(n) = x(n)y(n)$, respectively, for every value of n. The product function $(xy)(n)$ returns the product of the values of the functions at n and has nothing in common with the composition $x(y(n))$ of the two functions.

Basic arithmetic relationships for "Big Oh" follow from and can be easily proven with its definition.

Lemma 1.15 (Scaling)**.** For all constants $c > 0$, cf is $O(f)$. In particular, f is $O(f)$.

Proof. The relationship $cf(n) \leq cf(n)$ obviously holds for all $n \geq 0$. □

Constant factors are ignored, and only the powers and functions are taken into account. It is this ignoring of constant factors that motivates such a notation.

Lemma 1.16 (Transitivity)**.** If h is $O(g)$ and g is $O(f)$, then h is $O(f)$.

Proof. See Exercise 1.3.6. □

Informally, if h grows at most as quickly as g, which grows at most as quickly as f, then h grows at most as quickly as f.

Lemma 1.17 (Rule of sums)**.** If g_1 is $O(f_1)$ and g_2 is $O(f_2)$, then $g_1 + g_2$ is $O(\max\{f_1, f_2\})$.

Proof. See Exercise 1.3.6. □

If g is $O(f)$ and h is $O(f)$, then is $g + h$ is $O(f)$. In particular, if g is $O(f)$, then $g + f$ is $O(f)$. Informally, the growth rate of a sum is the growth rate of its fastest-growing term.

Lemma 1.18 (Rule of products)**.** If g_1 is $O(f_1)$ and g_2 is $O(f_2)$, then $g_1 g_2$ is $O(f_1 f_2)$.

Proof. See Exercise 1.3.6. □

In particular, if g is $O(f)$, then gh is $O(fh)$. Informally, the product of upper bounds of functions gives an upper bound for the product of the functions.

Using calculus we can obtain a nice time-saving rule.

Lemma 1.19 (Limit Rule)**.** Suppose $\lim_{n\to\infty} f(n)/g(n)$ exists (may be ∞), and denote the limit by L. Then

- if $L = 0$, then f is $O(g)$ and f is not $\Omega(g)$;
- if $0 < L < \infty$ then f is $\Theta(g)$;
- if $L = \infty$ then f is $\Omega(g)$ and f is not $O(g)$.

Proof. If $L = 0$ then from the definition of limit, in particular there is some n_0 such that $f(n)/g(n) \leq 1$ for all $n \geq n_0$. Thus $f(n) \leq g(n)$ for all such n, and $f(n)$ is $O(g(n))$ by definition. On the other hand, for each $c > 0$, it is not the case that $f(n) \geq cg(n)$ for all n past some threshold value n_1, so that $f(n)$ is not $\Omega(g(n))$. The other two parts are proved in the analogous way. □

To compute the limit if it exists, the standard *L'Hôpital's rule* of calculus is useful (see Section D.5).

More specific relations follow directly from the basic ones.

Example 1.20. Higher powers of n grow more quickly than lower powers: n^k is $O(n^l)$ if $0 \leq k \leq l$. This follows directly from the limit rule since $n^k/n^l = n^{k-l}$ has limit 1 if $k = l$ and 0 if $k < l$.

Example 1.21. The growth rate of a polynomial is given by the growth rate of its leading term (ignoring the leading coefficient by the scaling feature): if $P_k(n)$ is a polynomial of exact degree k then $P_k(n)$ is $\Theta(n^k)$. This follows easily from the limit rule as in the preceding example.

Example 1.22. Exponential functions grow more quickly than powers: n^k is $O(b^n)$, for all $b > 1$, $n > 1$, and $k \geq 0$. The restrictions on b, n, and k merely ensure that both functions are increasing. This result can be proved by induction or by using the limit-L'Hôpital approach above.

Example 1.23. Logarithmic functions grow more slowly than powers: $\log_b n$ is $O(n^k)$ for all $b > 1$, $k > 0$. This is the inverse of the preceding feature. Thus, as a result, $\log n$ is $O(n)$ and $n \log n$ is $O(n^2)$.

Exercises

Exercise 1.3.1. Prove that $10n^3 - 5n + 15$ is not $O(n^2)$.

Exercise 1.3.2. Prove that $10n^3 - 5n + 15$ is $\Theta(n^3)$.

Exercise 1.3.3. Prove that $10n^3 - 5n + 15$ is not $\Omega(n^4)$.

Exercise 1.3.4. Prove that $f(n)$ is $\Theta(g(n))$ if and only if both $f(n)$ is $O(g(n)$ and $f(n)$ is $\Omega(g(n))$.

Exercise 1.3.5. Using the definition, show that each function $f(n)$ in Table 1.3 stands in "Big-Oh" relation to the preceding one, that is, n is $O(n \log n)$, $n \log n$ is $O(n^{1.5})$, and so forth.

Exercise 1.3.6. Prove Lemmas 1.16–1.18.

Exercise 1.3.7. Decide on how to reformulate the Rule of Sums (Lemma 1.17) for "Big Omega" and "Big Theta" notation.

Exercise 1.3.8. Reformulate and prove Lemmas 1.15–1.18 for "Big Omega" notation.

1.4 Time complexity of algorithms

Definition 1.24 (Informal)**.** A function $f(n)$ such that the running time $T(n)$ of a given algorithm is $\Theta(f(n))$ measures the ***time complexity*** of the algorithm.

An algorithm is called ***polynomial time*** if its running time $T(n)$ is $O(n^k)$ where k is some fixed positive integer. A computational problem is considered ***intractable*** iff no deterministic algorithm with polynomial time complexity exists for it. But many problems are classed as intractable only because a polynomial solution is unknown, and it is a very challenging task to find such a solution for one of them.

Table 1.2: Relative growth of running time $T(n)$ when the input size increases from $n = 8$ to $n = 1024$ provided that $T(8) = 1$.

Time complexity		**Input size** n				**Time** $T(n)$
Function	*Notation*	*8*	*32*	*128*	*1024*	
Constant	1	1	1	1	1	1
Logarithmic	$\lg n$	1	1.67	2.67	3.33	$\lg n/3$
Log-squared	$\lg^2 n$	1	2.78	5.44	11.1	$\lg^2 n/9$
Linear	n	1	4	16	128	$n/8$
"$n \log n$"	$n \lg n$	1	6.67	37.3	427	$n \lg n/24$
Quadratic	n^2	1	16	256	16384	$n^2/64$
Cubic	n^3	1	64	4096	2097152	$n^3/512$
Exponential	2^n	1	2^{24}	2^{120}	2^{1016}	2^{n-8}

Table 1.2 shows how the running time $T(n)$ of algorithms having different time complexity, $f(n)$, grows relatively with the increasing input size n. Time complexity functions are listed in order such that g is $O(f)$ if g is above f: for example, the linear function n is $O(n \log n)$ and $O(n^2)$, etc. The asymptotic growth rate does not depend on the base of the logarithm, but the exact numbers in the table do — we use $\log_2 = \lg$ for simplicity.

Table 1.3: The largest data sizes n that can be processed by an algorithm with time complexity $f(n)$ provided that $T(10) = 1$ minute.

	Length of time to run an algorithm					
$f(n)$	1 minute	1 hour	1 day	1 week	1 year	1 decade
n	10	600	14 400	100 800	5.26×10^6	5.26×10^7
$n \lg n$	10	250	3 997	23 100	883 895	7.64×10^6
$n^{1.5}$	10	153	1 275	4 666	65 128	302,409
n^2	10	77	379	1 003	7 249	22,932
n^3	10	39	112	216	807	1,738
2^n	10	15	20	23	29	32

Table 1.3 is even more expressive in showing how the time complexity of an algorithm affects the size of problems the algorithm can solve (we again use $\log_2 = \lg$). A linear algorithm solving a problem of size $n = 10$ in exactly one minute will process about 5.26 million data items per year and 10 times more if we can wait a decade. But an exponential algorithm with $T(10) = 1$ minute will deal only with 29 data items after a year of running and add only 3 more items after a decade. Suppose we have computers 10,000 times faster (this is approximately the ratio of a week to a minute). Then we can solve a problem 10,000 times, 100 times, or 21.5 times larger than before if our algorithm is linear, quadratic, or cubic, respectively. But for exponential algorithms, our progress is much worse: we can add only 13 more input values if $T(n)$ is $\Theta(2^n)$.

Therefore, if our algorithm has a constant, logarithmic, log-square, linear, or even "$n \log n$" time complexity we may be happy and start writing a program with no doubt that it will meet at least some practical demands. Of course, before taking the plunge, it is better to check whether the hidden constant c, giving the computation volume per data item, is sufficiently small in our case. Unfortunately, order relations can be drastically misleading: for instance, two linear functions $10^{-4}n$ and $10^{10}n$ are of the same order $O(n)$, but we should not claim an algorithm with the latter time complexity as a big success.

Therefore, we should follow a simple rule: *roughly estimate the computation volume per data item for the algorithms after comparing their time complexities in a*

"Big-Oh" sense! We may estimate the computation volume simply by counting the number of elementary operations per data item.

In any case we should be *very* careful even with simple quadratic or cubic algorithms, and especially with exponential algorithms. If the running time is speeded up in Table 1.3 so that it takes one *second* per ten data items in all the cases, then we will still wait about 12 *days* ($2^{20} \equiv 1,048,576$ seconds) for processing only 30 items by the exponential algorithm. Estimate yourself whether it is practical to wait until 40 items are processed.

In practice, quadratic and cubic algorithms cannot be used if the input size exceeds tens of thousands or thousands of items, respectively, and exponential algorithms should be avoided whenever possible unless we always have to process data of very small size. Because even the most ingenious programming cannot make an inefficient algorithm fast (we would merely change the value of the hidden constant c slightly, but not the asymptotic order of the running time), it is better to spend more time to search for efficient algorithms, even at the expense of a less elegant software implementation, than to spend time writing a very elegant implementation of an inefficient algorithm.

Worst-case and average-case performance

We have introduced asymptotic notation in order to measure the running time of an algorithm. This is expressed in terms of elementary operations. "Big Oh", "Big Omega" and "Big Theta" notations allow us to state upper, lower and tight asymptotic bounds on running time that are independent of inputs and implementation details. Thus we can classify algorithms by performance, and search for the "best" algorithms for solving a particular problem.

However, we have so far neglected one important point. In general, *the running time varies not only according to the size of the input, but the input itself.* The examples in Section 1.4 were unusual in that this was not the case. But later we shall see many examples where it does occur. For example, some sorting algorithms take almost no time if the input is already sorted in the desired order, but much longer if it is not.

If we wish to compare two different algorithms for the same problem, it will be very complicated to consider their performance on all possible inputs. We need a simple measure of running time.

The two most common measures of an algorithm are the ***worst-case running time***, and the ***average-case running time***.

The worst-case running time has several advantages. If we can show, for example, that our algorithm runs in time $O(n \log n)$ no matter what input of size n we consider, we can be confident that even if we have an "unlucky" input given to our program, it will not fail to run fairly quickly. For so-called "mission-critical" applications this is an essential requirement. In addition, an upper bound on the worst-case running time is usually fairly easy to find.

The main drawback of the worst-case running time as a measure is that it may be too pessimistic. The real running time might be much lower than an "upper bound", the input data causing the worst case may be unlikely to be met in practice, and the constants c and n_0 of the asymptotic notation are unknown and may not be small. There are many algorithms for which it is difficult to specify the worst-case input. But even if it is known, the inputs actually encountered in practice may lead to much lower running times. We shall see later that the most widely used fast sorting algorithm, quicksort, has worst-case quadratic running time, $\Theta(n^2)$, but its running time for "random" inputs encountered in practice is $\Theta(n \log n)$.

By contrast, the average-case running time is not as easy to define. The use of the word "average" shows us that probability is involved. We need to specify a probability distribution on the inputs. Sometimes this is not too difficult. Often we can assume that every input of size n is equally likely, and this makes the mathematical analysis easier. But sometimes an assumption of this sort may not reflect the inputs encountered in practice. Even if it does, the average-case analysis may be a rather difficult mathematical challenge requiring intricate and detailed arguments. And of course the worst-case complexity may be very bad even if the average case complexity is good, so there may be considerable risk involved in using the algorithm.

Whichever measure we adopt for a given algorithm, our goal is to show that its running time is $\Theta(f)$ for some function f *and* there is no algorithm with running time $\Theta(g)$ for any function g that grows more slowly than f when $n \to \infty$. In this case our algorithm is ***asymptotically optimal*** for the given problem.

Proving that no other algorithm can be asymptotically better than ours is usually a difficult matter: we must carefully construct a formal mathematical model of a computer and derive a lower bound on the complexity of every algorithm to solve the given problem. In this book we will not pursue this topic much. If our analysis does show that an upper bound for our algorithm matches the lower one for the problem, then we need not try to invent a faster one.

Exercises

Exercise 1.4.1. Add columns to Table 1.3 corresponding to one century (10 decades) and one millennium (10 centuries).

Exercise 1.4.2. Add rows to Table 1.2 for algorithms with time complexity $f(n) = \lg\lg n$ and $f(n) = n^2 \lg n$.

1.5 Basic recurrence relations

As we will see later, a great many algorithms are based on the following ***divide-and-conquer*** principle:

- divide a large problem into smaller subproblems and recursively solve each subproblem, then
- combine solutions of the subproblems to solve the original problem.

Running time of such algorithms is determined by a ***recurrence relation*** accounting for the size and number of the subproblems and for the cost of splitting the problem into them. The recursive relation defines a function "in terms of itself", that is, by an expression that involves the same function. The definition is not circular provided that the value at a natural number n is defined in terms of values at smaller natural numbers, and the recursion terminates at some base case below which the function is not defined.

Example 1.25 (Fibonacci numbers)**.** These are defined by one of the most famous recurrence relations: $F(n) = F(n-1) + F(n-2)$; $F(1) = 1$, and $F(2) = 1$. The last two equations are called the ***base of the recurrence*** or ***initial condition***. The recurrence relation uniquely defines the function $F(n)$ at any number n because any particular value of the function is easily obtained by generating all the preceding values until the desired term is produced, for example, $F(3) = F(2) + F(1) = 2$; $F(4) = F(3) + F(2) = 3$, and so forth. Unfortunately, to compute $F(10000)$, we need to perform 9998 additions.

Example 1.26. One more recurrence relation is $T(n) = 2T(n-1) + 1$ with the base condition $T(0) = 0$. Here, $T(1) = 2 \cdot 0 + 1 = 1$, $T(2) = 2 \cdot 1 + 1 = 3$, $T(3) = 2 \cdot 3 + 1 = 7$, $T(4) = 2 \cdot 7 + 1 = 15$, and so on.

Note. A recurrence relation is sometimes simply called a ***recurrence***. In engineering it is called a ***difference equation***.

We will frequently meet recurrences in algorithm analysis. It is more convenient to have an explicit expression, (or ***closed-form expression***) for the function in order to compute it quickly for any argument value n and to compare it with other functions. The closed-form expression for $T(n)$, that is, what is traditionally called a "formula", makes the growth of $T(n)$ as a function of n more apparent. The process of deriving the explicit expression is called "*solving the recurrence relation*".

Our consideration will be restricted to only the two simplest techniques for solving recurrences: (*i*) guessing a solution from a sequence of values $T(0)$, $T(1)$, $T(2)$, ..., and proving it by mathematical induction (a "bottom-up" approach) and (*ii*) "telescoping" the recurrence (a "top-down" approach). Both techniques allow us to obtain closed forms of some important recurrences that describe performance of sort and search algorithms. For instance, in Example 1.26 we can simply guess the closed form expression $T(n) = 2^n - 1$ by inspecting the first few terms of the sequence 0, 1, 3, 7, 15 because $0 = 1-1$, $1 = 2-1$, $3 = 4-1$, $7 = 8-1$, and $15 = 16-1$. But in other cases these techniques may fail and more powerful mathematical tools beyond the scope of this book, such as using characteristic equations and generating functions, should be applied.

Guessing to solve a recurrence

There is no formal way to find a closed-form solution. But after we have guessed the solution, it may be proven to be correct by mathematical induction (see Section D.2).

Example 1.27. For the recurrence $T(n) = 2T(n-1)+1$ with the base condition $T(0) = 0$ in Example 1.26 we guessed the closed-form relationship $T(n) = 2^n - 1$ by analysing the starting terms 0, 1, 3, 7, 15. This formula is obviously true for $n = 0$, because $2^0 - 1 = 0$. Now, by the induction hypothesis,

$$T(n) = 2T(n-1)+1 = 2(2^{n-1}-1)+1 = 2^n - 1$$

and this is exactly what we need to prove.

The Fibonacci sequence provides a sterner test for our guessing abilities.

Example 1.28. The first few terms of the sequence 1, 1, 2, 3, 5, 8, 13, 21, 34, ... give no hint regarding the desired explicit form. Thus let us analyse the recurrence $F(n) = F(n-1)+F(n-2)$ itself. $F(n)$ is almost doubled every time, so that $F(n) < 2^n$. The simplest guess $F(n) = c2^n$ with $c < 1$ fails because for any scaling factor c it leads to the impossible equality $2^n = 2^{n-1} + 2^{n-2}$, or $4 = 2+1$. The next guess is that the base 2 of the exponential function should be smaller, $\phi < 2$, that is, $F(n) = c\phi^n$. The resulting equation $\phi^n = \phi^{n-1} + \phi^{n-2}$ reduces to the quadratic one, $\phi^2 - \phi - 1 = 0$, with the two roots: $\phi_1 = 0.5(1+\sqrt{5})$ and $\phi_2 = 0.5(1-\sqrt{5})$. Because each root solves the recurrence, the same holds for any linear combination of them, so we know that $F(n) = c_1\phi_1^n + c_2\phi_2^n$ satisfies the recurrence. We choose the constants c_1 and c_2 to satisfy the base conditions $F(1) = 1$ and $F(2) = 1$: $F(1) = c_1\phi_1 + c_2\phi_2 = 1$ and $F(2) = c_1\phi_1^2 + c_2\phi_2^2 = 1$. Thus $c_1 = \frac{1}{\sqrt{5}}$ and $c_2 = -\frac{1}{\sqrt{5}}$ so that

$$F(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n \cong \frac{\phi^n}{\sqrt{5}}$$

where $\phi = \frac{1+\sqrt{5}}{2} \cong 1.618$ is the well-known "golden ratio". The term with $(1-\sqrt{5})/2 \cong -0.618$ tends to zero when $n \to \infty$, and so $F(n)$ is $\Theta(\phi^n)$.

"Telescoping" of a recurrence

This means a recursive substitution of the same implicit relationship in order to derive the explicit relationship. Let us apply it to the same recurrence $T(n) = 2T(n-1)+1$ with the base condition $T(0) = 0$ as in Examples 1.26 and 1.27:

Step 0 *initial recurrence* $T(n) = 2T(n-1)+1$

Step 1 *substitute* $T(n-1) = 2T(n-2)+1$, that is, replace $T(n-1)$:

$$T(n) = 2(2T(n-2)+1)+1 = 2^2T(n-2)+2+1$$

Step 2 *substitute* $T(n-2) = 2T(n-3)+1$:

$$T(n) = 2^2(2T(n-3)+1)+2+1 = 2^3T(n-3)+2^2+2+1$$

Step 3 *substitute* $T(n-3) = 2T(n-4)+1$:

$$\begin{aligned} T(n) &= 2^3(2T(n-4)+1)+2^2+2+1 \\ &= 2^4T(n-4)+2^3+2^2+2+1 \end{aligned}$$

Step ... ...

Step $n-2$ ...

$$T(n) = 2^{n-1}T(1)+2^{n-2}+\ldots+2^2+2+1$$

Step $n-1$ *substitute* $T(1) = 2T(0)+1$:

$$\begin{aligned} T(n) &= 2^{n-1}(2T(0)+1)+2^{n-2}+\ldots+2+1 \\ &= 2^nT(0)+2^{n-1}+2^{n-2}+\ldots+2+1 \end{aligned}$$

Because of the base condition $T(0) = 0$, the explicit formula is:

$$T(n) = 2^{n-1}+2^{n-2}+\ldots+2+1 \equiv 2^n-1$$

As shown in Figure 1.5, rather than successively substitute the terms $T(n-1)$, $T(n-2), \ldots, T(2), T(1)$, it is more convenient to write down a sequence of the scaled relationships for $T(n)$, $2T(n-1)$, $2^2T(n-2)$, ..., $2^{n-1}T(1)$, respectively, then individually sum left and right columns, and eliminate similar terms in the both sums (the terms are scaled to facilitate their direct elimination). Such solution is called ***telescoping*** because the recurrence unfolds like a telescopic tube.

Although telescoping is not a powerful technique, it returns the desired explicit forms of most of the basic recurrences that we need in this book (see Examples 1.29–1.32 below). But it is helpless in the case of the Fibonacci recurrence because after proper scaling of terms and reducing similar terms in the left and right sums, telescoping returns just the same initial recurrence.

Example 1.29. $T(n) = T(n-1)+n; T(0) = 1$.

This relation arises when a recursive algorithm loops through the input to eliminate one item and is easily solved by telescoping:

$$\begin{aligned} T(n) &= T(n-1)+n \\ T(n-1) &= T(n-2)+(n-1) \\ &\ldots \\ T(1) &= T(0)+1 \end{aligned}$$

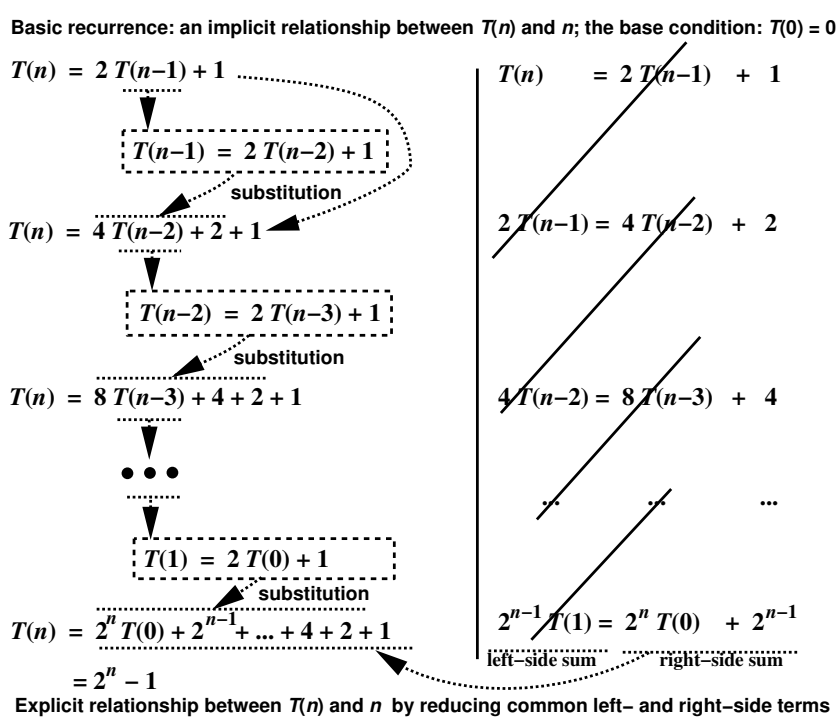


Figure 1.5: Telescoping as a recursive substitution.

By summing left and right columns and eliminating the similar terms, we obtain that $T(n) = T(0) + 1 + 2 + \ldots + (n-2) + (n-1) + n = \frac{n(n+1)}{2}$ so that $T(n)$ is $\Theta(n^2)$.

Example 1.30. $T(n) = T(\lceil n/2 \rceil) + 1; T(1) = 0$.

This relation arises for a recursive algorithm that almost halves the input at each step. Suppose first that $n = 2^m$. Then, the recurrence telescopes as follows:

$$\begin{aligned} T(2^m) &= T(2^{m-1}) + 1 \\ T(2^{m-1}) &= T(2^{m-2}) + 1 \\ &\cdots \\ T(2^1) &= T(2^0) + 1 \end{aligned}$$

so that $T(2^m) = m$, or $T(n) = \lg n$ which is $\Theta(\log n)$.

For general n, the total number of the halving steps cannot be greater than $m = \lceil \lg n \rceil$. Therefore, $T(n) \leq \lceil \lg n \rceil$ for all n. This recurrence is usually called the ***repeated halving principle***.

Example 1.31. Recurrence $T(n) = T(\lceil n/2 \rceil) + n; T(0) = 0$.

This relation arises for a recursive algorithm that halves the input after examining every item in the input for $n \geq 1$. Under the same simplifying assumption $n = 2^m$, the recurrence telescopes as follows:

$$\begin{aligned} T(2^m) &= T(2^{m-1}) + n \\ T(2^{m-1}) &= T(2^{m-2}) + n/2 \\ T(2^{m-2}) &= T(2^{m-3}) + n/4 \\ &\cdots \\ T(2) &= T(1) + 2 \\ T(1) &= T(0) + 1 \end{aligned}$$

so that $T(n) = n + \frac{n}{2} + \frac{n}{4} + \ldots + 1$ which is $\Theta(n)$.

In the general case, the solution is also $\Theta(n)$ because each recurrence after halving an odd-size input may add to the above sum at most 1 and the number of these extra units is at most $\lceil \lg n \rceil$.

Example 1.32. Recurrence $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n; T(1) = 0$.

This relation arises for a recursive algorithm that makes a linear pass through the input for $n \geq 2$ and splits it into two halves. Under the same simplifying assumption $n = 2^m$, the recurrence telescopes as follows:

$$\begin{aligned} T(2^m) &= 2T(2^{m-1}) + 2^m \\ T(2^{m-1}) &= 2T(2^{m-2}) + 2^{m-1} \\ &\cdots \\ T(2) &= 2T(1) + 2 \end{aligned}$$

so that

$$\begin{aligned} \frac{T(2^m)}{2^m} &= \frac{T(2^{m-1})}{2^{m-1}} + 1 \\ \frac{T(2^{m-1})}{2^{m-1}} &= \frac{T(2^{m-2})}{2^{m-2}} + 1 \\ &\cdots \\ \frac{T(2)}{2} &= \frac{T(1)}{1} + 1. \end{aligned}$$

Therefore, $\frac{T(n)}{n} = \frac{T(1)}{1} + m = \lg n$, so that $T(n) = n \lg n$ which is $\Theta(n \log n)$.

For general n, $T(n)$ is also $\Theta(n \log n)$ (see Exercise 1.5.2).

There exist very helpful parallels between the differentiation / integration in calculus and recurrence analysis by telescoping.

- The difference equation $T(n)-2T(n-1)=c$ rewritten as $\frac{T(n)-T(n-1)}{1}=T(n-1)+c$ resembles the differential equation $\frac{dT(x)}{dx}=T(x)$. Telescoping of the difference equation results in the formula $T(n)=c(2^n-1)$ whereas the integration of the differential equation produces the analogous exponential one $T(x)=ce^x$.

- The difference equation $T(n)-T(n-1)=cn$ has the differential analogue $\frac{dT(x)}{dx}=cx$, and both equations have similar solutions $T(n)=c\frac{n(n+1)}{2}$ and $T(x)=\frac{c}{2}x^2$, respectively.

- Let us change variables by replacing n and x with $m=\lg n$ and $y=\ln x$ so that $n=2^m$ and $x=e^y$, respectively. The difference equation $T(n)-T(\frac{n}{2})=c$ where $n=2^m$ and $\frac{n}{2}=2^{m-1}$ reduces to $T(m)-T(m-1)=c$. The latter has the differential analogue $\frac{dT(y)}{dy}=c$. These two equations result in the similar explicit expressions $T(m)=cm$ and $T(y)=cy$, respectively, so that $T(n)=c\lg n$ and $T(x)=c\ln x$.

The parallels between difference and differential equations may help us in deriving the desired closed-form solutions of complicated recurrences.

Exercise 1.5.1. Show that the solution in Example 1.31 is also in $\Omega(n)$ for general n.

Exercise 1.5.2. Show that the solution $T(n)$ to Example 1.32 is no more than $n\lg n+n-1$ for every $n\geq 1$. Hint: try induction on n.

Exercise 1.5.3. The running time $T(n)$ of a certain algorithm to process n data items is given by the recurrence $T(n)=kT\left(\frac{n}{k}\right)+cn$; $T(1)=0$ where k is a positive integer constant and n/k means either $\lceil n/k\rceil$ or $\lfloor n/k\rfloor$. Derive the explicit expression for $T(n)$ in terms of c, n, and k assuming $n=k^m$ with integer $m=\log_k n$ and k and find time complexity of this algorithm in the "Big-Oh" sense.

Exercise 1.5.4. The running time $T(n)$ of a slightly different algorithm is given by the recurrence $T(n)=kT\left(\frac{n}{k}\right)+ckn$; $T(1)=0$. Derive the explicit expression for $T(n)$ in terms of c, n, and k under the same assumption $n=k^m$ and find time complexity of this algorithm in the "Big-Oh" sense.

1.6 Capabilities and limitations of algorithm analysis

We should neither overestimate nor underestimate the capabilities of algorithm analysis. Many existing algorithms have been analysed with much more complex techniques than used in this book and recommended for practical use on the basis of these studies. Of course, not all algorithms are worthy of study and we should not suppose that a rough complexity analysis will result immediately in efficient algorithms. But computational complexity permits us to better evaluate basic ideas in developing new algorithms.

To check whether our analysis is correct, we may code a program and see whether its observed running time fits predictions. But it is very difficult to differentiate between, say, $\Theta(n)$ and $\Theta(n \log n)$ algorithms using purely empirical evidence. Also, "Big-Oh" analysis is not appropriate for small amounts of input and hides the constants which may be crucial for a particular task.

Example 1.33. An algorithm **A** with running time $T_{\mathbf{A}} = 2n \lg n$ becomes less efficient than another algorithm **B** having running time $T_{\mathbf{B}} = 1000n$ only when $2n \lg n > 1000n$, or $\lg n > 500$, or $n > 2^{500} \cong 10^{150}$, and such amounts of input data simply cannot be met in practice. Thus, although the algorithm **B** is better in the "Big Oh" sense, in practice we should use algorithm **A**.

Large constants have to be taken into account when an algorithm is very complex, or when we must discriminate between cheap or expensive access to input data items, or when there may be lack of sufficient memory for storing large data sets, etc. But even when constants and lower-order terms are considered, the performance predicted by our analysis may differ from the empirical results. Recall that for *very* large inputs, even the asymptotic analysis may break down, because some operations (like addition of large numbers) can no longer be considered as elementary.

In order to analyse algorithm performance we have used a simplified mathematical model involving elementary operations. In the past, this allowed for fairly accurate analysis of the actual running time of program implementing a given algorithm. Unfortunately, the situation has become more complicated in recent years. Sophisticated behaviour of computer hardware such as pipelining and caching means that the time for elementary operations can vary wildly, making these models less useful for detailed prediction. Nevertheless, the basic distinction between linear, quadratic, cubic and exponential time is still as relevant as ever. In other words, the crude differences captured by the Big-Oh notation give us a very good way of comparing algorithms; comparing two linear time algorithms, for example, will require more experimentation.

We can use worst-case and average-case analysis to obtain some meaningful estimates of possible algorithm performance. But we must remember that both recurrences and asymptotic "Big-Oh", "Big-Omega", and "Big-Theta" notation are just mathematical tools used to model certain aspects of algorithms. Like all models, they are not universally valid and so the mathematical model and the real algorithm may behave quite differently.

Exercises

Exercise 1.6.1. Algorithms **A** and **B** use $T_{\mathbf{A}}(n) = 5n \log_{10} n$ and $T_{\mathbf{B}}(n) = 40n$ elementary operations, respectively, for a problem of size n. Which algorithm has better performance in the "Big Oh" sense? Work out exact conditions when each algorithm outperforms the other.

Exercise 1.6.2. We have to choose one of two algorithms, **A** and **B**, to process a database containing 10^9 records. The average running time of the algorithms is $T_{\mathbf{A}}(n) = 0.001n$ and $T_{\mathbf{B}}(n) = 500\sqrt{n}$, respectively. Which algorithm should be used, assuming the application is such that we can tolerate the risk of an occasional long running time?

1.7 Notes

The word *algorithm* relates to the surname of the great mathematician Muhammad ibn Musa al-Khwarizmi, whose life spanned approximately the period 780–850. His works, translated from Arabic into Latin, for the first time exposed Europeans to new mathematical ideas such as the Hindu positional decimal notation and step-by-step rules for addition, subtraction, multiplication, and division of decimal numbers. The translation converted his surname into "Algorismus", and the computational rules took on this name. Of course, mathematical algorithms existed well before the term itself. For instance, Euclid's algorithm for computing the greatest common divisor of two positive integers was devised over 1000 years before.

The Big-Oh notation was used as long ago as 1894 by Paul Bachmann and then Edmund Landau for use in number theory. However the other asymptotic notations Big-Omega and Big-Theta were introduced in 1976 by Donald Knuth (at time of writing, perhaps the world's greatest living computer scientist).

Algorithms running in $\Theta(n \log n)$ time are sometimes called ***linearithmic***, to match "logarithmic", "linear", "quadratic", etc.

The quadratic equation for ϕ in Example 1.28 is called the ***characteristic equation*** of the recurrence. A similar technique can be used for solving any constant coefficient linear recurrence of the form $F(n) = \sum_{k=1}^{K} a_k F(n-k)$ where K is a fixed positive integer and the a_k are constants.

Chapter 2

Efficiency of Sorting

Sorting rearranges input data according to a particular linear order (see Section D.3 for definitions of order and ordering relations). The most common examples are the usual dictionary (lexicographic) order on strings, and the usual order on integers.

Once data is sorted, many other problems become easier to solve. Some of these include: finding an item, finding whether any duplicate items exist, finding the frequency of each distinct item, finding order statistics such as the maximum, minimum, median and quartiles. There are many other interesting applications of sorting, and many different sorting algorithms, each with their own strengths and weaknesses. In this chapter we describe and analyse some popular sorting algorithms.

2.1 The problem of sorting

The problem of sorting is to rearrange an input list of ***keys***, which can be compared using a total order $\leq$, into an output list such that if i and j are keys and i precedes j in the output list, then $i \leq j$. Often the key is a data field in a larger object: for example, we may wish to sort database records of customers, with the key being their bank balance. If each object has a different key, then we can simply use the keys as identifiers for the purpose of sorting: rather than moving large objects we need only keep a pointer from the key to the object.

There are several important attributes of sorting algorithms.

Definition 2.1. A sorting algorithm is called ***comparison-based*** if the only way to gain information about the total order is by comparing a pair of elements at a time via the order $\leq$.

A sorting algorithm is called ***stable*** if whenever two objects have the same key in the input, they appear in the same order in the output.

A sorting algorithm is called ***in-place*** if it uses only a fixed additional amount of working space, independent of the input size.

With a comparison-based sorting algorithm, we cannot use any information about the keys themselves (such as the fact that they are all small integers, for example), only their order relation. These algorithms are the most generally applicable and we shall focus exclusively on them in this book (but see Exercise 2.7.2).

We consider only two elementary operations: a ***comparison*** of two items and a ***move*** of an item. The running time of sorting algorithms in practice is usually dominated by these operations. Every algorithm that we consider will make at most a constant number of moves for each comparison, so that the asymptotic running time in terms of elementary operations will be determined by the number of comparisons. However, lower order terms will depend on the exact number of moves. Furthermore, the actual length of time taken by a data move depends on the implementation of the list. For example, moving an element from the end to the beginning of an array takes longer than doing the same for a linked list. We shall discuss these issues later.

The efficiency of a particular sorting algorithm may depend on many factors, for instance:

- how many items have to be sorted;
- are the items only related by the order relation, or do they have other restrictions (for example, are they all integers from the range 1 to 1000);
- to what extent they are pre-sorted;
- can they be placed into an internal (fast) computer memory or must they be sorted in external (slow) memory, such as on disk (so called ***external sorting***).

No one algorithm is the best for all possible situations, and so it is important to understand the strengths and weaknesses of several algorithms.

As far as computer implementation is concerned, sorting makes sense only for linear data structures. We will consider lists (see Section C.1 for a review of basic concepts) which have a first element (the head), a last element (the tail) and a method of accessing the next element in constant time (an iterator). This includes array-based lists, and singly- and doubly-linked lists. For some applications we will need a method of accessing the previous element quickly; singly-linked lists do not provide this. Also, array-based lists allow fast random access. The element at any given position may be retrieved in constant time, whereas linked list structures do not allow this.

Exercises

Exercise 2.1.1. The well-known and obvious ***selection sort*** algorithm proceeds as follows. We split the input list into a head and tail sublist. The head ("sorted") sublist is initially empty, and the tail ("unsorted") sublist is the whole list. The algorithm successively scans through the tail sublist to find the minimum element and moves it to the end of the head sublist. It terminates when the tail sublist becomes empty. (Java code for an array implementation is found in Section A.1).

How many comparisons are required by selection sort in order to sort the input list $(6,4,2,5,3,1)$?

Exercise 2.1.2. Show that selection sort uses the same number of comparisons on every input of a fixed size. How many does it use, exactly, for an input of size n?

Exercise 2.1.3. Is selection sort comparison-based? in-place? stable?

Exercise 2.1.4. Give a linear time algorithm to find whether a sorted list contains any duplicate elements. How would you do this if the list were not sorted?

2.2 Insertion sort

This is the method usually used by cardplayers to sort cards in their hand. Insertion sort is easy to implement, stable, in-place, and works well on small lists and lists that are close to sorted. However, it is very inefficient for large random lists.

Insertion sort is iterative and works as follows. The algorithm splits a list of size n into a head ("sorted") and tail ("unsorted") sublist.

- The head sublist is initially of size 1.
- Repeat the following step until the tail sublist is empty:
  - choose the first element x in the tail sublist;
  - find the last element y in the head sublist not exceeding x;
  - insert x after y in the head sublist.

Before each step $i = 1,2,\ldots,n-1$, the sorted and unsorted parts have i and $n-i$ elements, respectively. The first element of the unsorted sublist is moved to the correct position in the sorted sublist by exhaustive backward search, by comparing it to each element in turn until the right place is reached.

Example 2.2. Table 2.1 shows the execution of insertion sort. Variables C_i and M_i denote the number of comparisons and number of positions to move backward, respectively, at the ith iteration. Elements in the sorted part are italicized, the currently sorted element is underlined, and the element to sort next is boldfaced.

Table 2.1: Sample execution of insertion sort.

i	C_i	M_i	**Data to sort**									
			25	**8**	2	91	70	50	20	31	15	65
1	1	1	*8*	*25*	**2**	91	70	50	20	31	15	65
2	2	2	*2*	*8*	*25*	**91**	70	50	20	31	15	65
3	1	0	*2*	*8*	*25*	*91*	**70**	50	20	31	15	65
4	2	1	*2*	*8*	*25*	*70*	*91*	**50**	20	31	15	65
5	3	2	*2*	*8*	*25*	*50*	*70*	*91*	**20**	31	15	65
6	5	4	*2*	*8*	*20*	*25*	*50*	*70*	*91*	**31**	15	65
7	4	3	*2*	*8*	*20*	*25*	*31*	*50*	*70*	*91*	**15**	65
8	7	6	*2*	*8*	*15*	*20*	*25*	*31*	*50*	*70*	*91*	**65**
9	3	2	*2*	*8*	*15*	*20*	*25*	*31*	*50*	*65*	*70*	*91*

Analysis of insertion sort

Insertion sort is easily seen to be correct (see Exercise 2.2.2 for formal proof), since the head sublist is always sorted, and eventually expands to include all elements.

It is not too hard to find the worst case for insertion sort: when the input consists of distinct items in reverse sorted order, every element must be compared with every element preceding it. The number of moves is also maximized by such input. The best case for insertion sort is when the input is already sorted, when only $n-1$ comparisons are needed.

Lemma 2.3. The worst-case time complexity of insertion sort is $\Theta(n^2)$.

Proof. Fill in the details yourself — see Exercise 2.2.3. □

Since the best case is so much better than the worst, we might hope that on average, for random input, insertion sort would perform well. Unfortunately, this is not true.

Lemma 2.4. The average-case time complexity of insertion sort is $\Theta(n^2)$.

Proof. We first calculate the average number $\overline{C_i}$ of comparisons at the ith step. At the beginning of this step, i elements of the head sublist are already sorted and the next element has to be inserted into the sorted part. This element will move backward j steps, for some j with $0 \leq j \leq i$. If $0 \leq j \leq i-1$, the number of comparisons used will be $j+1$. But if $j = i$ (it ends up at the head of the list), there will be only i comparisons (since no final comparison is needed).

Assuming all possible inputs are equally likely, the value of j will be equally likely to take any value $0,\ldots,i$. Thus the expected number of comparisons will be

$$\overline{C_i} = \frac{1}{i+1}(1+2+\cdots+i-1+i+i) = \frac{1}{i+1}\left(\frac{i(i+1)}{2}+i\right) = \frac{i}{2}+\frac{i}{i+1}.$$

(see Section D.6 for the simplification of the sum, if necessary).

The above procedure is performed for $i = 1,2,\ldots,n-1$, so that the average total number $\overline{C}$ of comparisons is as follows:

$$\overline{C} = \sum_{i=1}^{n-1}\overline{C_i} = \sum_{i=1}^{n-1}\left(\frac{i}{2}+\frac{i}{i+1}\right) = \frac{1}{2}\sum_{i=1}^{n-1} i + \sum_{i=1}^{n-1}\frac{i}{i+1}$$

The first sum is equal to $\frac{(n-1)n}{2}$. To find the second sum, let us rewrite $\frac{i}{i+1}$ as $1-\frac{1}{i+1}$ so that

$$\sum_{i=1}^{n-1}\frac{i}{i+1} = \sum_{i=1}^{n-1}\left(1-\frac{1}{i+1}\right) = n-1-\sum_{i=1}^{n-1}\frac{1}{i+1} = n-\sum_{i=1}^{n}\frac{1}{i} = n-H_n$$

where H_n denotes the n-th ***harmonic number***: $H_n \approx \ln n$ when $n \to \infty$.

Therefore, $\overline{C} = \frac{(n-1)n}{4}+n-H_n$. Now the total number of data moves is at least zero and at most the number of comparisons. Thus the total number of elementary operations is $\Theta(n^2)$. □

The running time of insertion sort is strongly related to *inversions*. The number of inversions of a list is one measure of how far it is from being sorted.

Definition 2.5. An ***inversion*** in a list a is an ordered pair of positions (i,j) such that $i<j$ but $a[i]>a[j]$.

Example 2.6. The list $(3,2,5)$ has only one inversion corresponding to the pair $(3,2)$, the list $(5,2,3)$ has two inversions, namely, $(5,2)$ and $(5,3)$, the list $(3,2,5,1)$ has four inversions $(3,2)$, $(3,1)$, $(2,1)$, and $(5,1)$, and so on.

Example 2.7. Table 2.2 shows the number of inversions, I_i, for each element $a[i]$ of the list in Table 2.1 with respect to all preceding elements $a[0],\ldots,a[i-1]$ (C_i and M_i are the same as in Table 2.1).

Note that $I_i = M_i$ in Table 2.1. This is not merely a coincidence—it is always true. See Exercise 2.2.4.

The total number of inversions $I = \sum_{i=1}^{n-1} I_i$ in a list to be sorted by insertion sort is equal to the total number of positions an element moves backward during the sort. The total number of data comparisons $C = \sum_{i=1}^{n-1} C_i$ is also equal to the total number of inversions plus at most $n-1$. For the initial list in Tables 2.1 and 2.2,

Table 2.2: Number of inversions I_i, comparisons C_i and data moves M_i for each element $a[i]$ in sample list.

Index i	0	1	2	3	4	5	6	7	8	9
List element $a[i]$	25	8	2	91	70	50	20	31	15	65
I_i		1	2	0	1	2	4	3	6	2
C_i		1	2	1	2	3	5	4	7	3
M_i		1	2	0	1	2	4	3	6	2

$I = 21$, and insertion sort performs $C = 28$ comparisons and $M = 21$ data moves: in total, 49 elementary operations.

Swapping two neighbours that are out of order removes exactly one inversion, and a sorted list has no inversions. If an original list has I inversions, insertion sort has to swap I pairs of neighbours. Because of $\Theta(n)$ other operations in the algorithm, its running time is $\Theta(n+I)$. Thus, on nearly sorted lists for which I is $\Theta(n)$, insertion sort runs in linear time. Unfortunately this type of list does not occur often, if we choose one randomly. As we have seen above, the average number of inversions for a randomly chosen list must be in $\Theta(n^2)$. This shows that more efficient sorting algorithms must eliminate more than just one inversion between neighbours per swap. One way to do this is a generalization of insertion sort called *Shellsort* (see Exercise 2.2.7).

Implementation of insertion sort

The number of comparisons does not depend on how the list is implemented, but the number of moves does. The insertion operation can be implemented in a linked list in constant time, but in an array there is no option but to shift elements to the right when inserting an element, taking linear time in the worst and average case. Thus if using an array implementation of a list, we may as well move the element backward by successive swaps. If using a linked list, we can make fewer swaps by simply scanning backward. On the other hand, scanning backward is easy in an array but takes more time in a singly linked list. However, none of these issues affect the asymptotic Big-Theta running time of the algorithm, just the hidden constants and lower order terms. The main problem with insertion sort is that it takes too many comparisons in the worst case, no matter how cleverly we implement it.

Figure 2.1 shows basic pseudocode for arrays.

Exercises

Exercise 2.2.1. Determine the quantities C_i and M_i when insertion sort is run on the input list $(91, 70, 65, 50, 31, 25, 20, 15, 8, 2)$.

Exercise 2.2.2. Prove by induction that algorithm `insertionSort` is correct.

```
algorithm insertionSort
    Input: array a[0..n − 1]
begin
    for i ← 1 to n − 1 do
        k ← i − 1
        while k ≥ 0 and a[k] > a[k + 1] do
            swap(a, k, k + 1)
            k ← k − 1
        end while
    end for
end
```

Figure 2.1: Insertion sort for arrays.

Exercise 2.2.3. Prove that the worst-case time complexity of insertion sort is $\Theta(n^2)$ and the best case is $\Theta(n)$.

Exercise 2.2.4. Prove that the number of inversions, I_i, of an element $a[i]$ with respect to the preceding elements, $a[0],\ldots,a[i-1]$, in the initial list is equal to the number of positions moved backward by $a[i]$ in the execution of insertion sort.

Exercise 2.2.5. Suppose a sorting algorithm swaps elements $a[i]$ and $a[i+\texttt{gap}]$ of a list a which were originally out of order. Prove that the number of inversions in the list is reduced by at least 1 and at most $2\,\texttt{gap}-1$.

Exercise 2.2.6. ***Bubble sort*** works as follows to sort an array. There is a sorted left subarray and unsorted right subarray; the left subarray is initially empty. At each iteration we step through the right subarray, comparing each pair of neighbours in turn, and swapping them if they are out of order. At the end of each such pass, the sorted subarray has increased in size by 1, and we repeat the entire procedure from the beginning of the unsorted subarray. (Java code is found in Section A.1.)

Prove that the average time complexity of bubble sort is $\Theta(n^2)$, and that bubble sort never makes fewer comparisons than insertion sort.

Exercise 2.2.7. ***Shellsort*** is a generalization of insertion sort that works as follows. We first choose an ***increment sequence*** $\ldots h_t > h_{t-1} > \ldots > h_1 = 1$. We start with some value of t so that $h_t < n$. At each step we form the sublists of the input list a consisting of elements $h := h_t$ apart (for example, the first such list has the elements at position $0, h, 2h, \ldots$, the next has the elements $1, 1+h, 1+2h, \ldots$, etc). We sort each of these h lists using insertion sort (we call this the *h-sorting* step). We then reduce t by 1 and continue. Note that the last step is always a simple insertion sort.

Explain why Shellsort is not necessarily slower than insertion sort. Give an input on which Shellsort uses fewer comparions overall than insertion sort.

Exercise 2.2.8. Find the total numbers of comparisons and backward moves performed by Shellsort on the input list $(91, 70, 65, 50, 31, 25, 20, 15, 8, 2)$ and compare the total number of operations with that for insertion sort in Exercise 2.2.1.

2.3 Mergesort

This algorithm exploits a recursive divide-and-conquer approach resulting in a worst-case running time of $\Theta(n \log n)$, the best asymptotic behaviour that we have seen so far. Its best, worst, and average cases are very similar, making it a very good choice if predictable runtime is important. Versions of mergesort are particularly good for sorting data with slow access times, such as data that cannot be held in internal memory or are stored in linked lists.

Mergesort is based on the following basic idea.

- If the size of the list is 0 or 1, return.
- Otherwise, separate the list into two lists of equal or nearly equal size and recursively sort the first and second halves separately.
- Finally, merge the two sorted halves into one sorted list.

Clearly, almost all the work is in the merge step, which we should make as efficient as possible. Obviously any merge must take at least time that is linear in the total size of the two lists in the worst case, since every element must be looked at in order to determine the correct ordering. We can in fact achieve a linear time merge, as we see in the next section.

Analysis of mergesort

Lemma 2.8. Mergesort is correct.

Proof. We use induction on the size n of the list. If $n = 0$ or 1, the result is obviously correct. Otherwise, mergesort calls itself recursively on two sublists each of which has size less than n. By induction, these lists are correctly sorted. Provided that the merge step is correct, the top level call of mergesort then returns the correct answer. □

Almost all the work occurs in the merge steps, so we need to perform those efficiently.

Theorem 2.9. Two input sorted lists A and B of size n_A and n_B, respectively, can be merged into an output sorted list C of size $n_C = n_A + n_B$ in linear time.

Proof. We first show that the number of comparisons needed is linear in n. Let i, j, and k be pointers to current positions in the lists A, B, and C, respectively. Initially, the pointers are at the first positions, $i = 0$, $j = 0$, and $k = 0$. Each time the smaller of the two elements $A[i]$ and $B[j]$ is copied to the current entry $C[k]$, and the

corresponding pointers k and either i or j are incremented by 1. After one of the input lists is exhausted, the rest of the other list is directly copied to list C. Each comparison advances the pointer k so that the maximum number of comparisons is $n_A + n_B - 1$.

All other operations also take linear time. □

The above proof can be visualized easily if we think of the lists as piles of playing cards placed face up. At each step, we choose the smaller of the two top cards and move it to the temporary pile.

Example 2.10. If $a = (2,8,25,70,91)$ and $b = (15,20,31,50,65)$, then merge into $c = (2,8,15,20,25,31,50,65,70,91)$ as follows.

Step 1 $a[0] = 2$ and $b[0] = 15$ are compared, $2 < 15$, and 2 is copied to c, that is, $c[0] \leftarrow 2$, $i \leftarrow 0+1$, and $k \leftarrow 0+1$.

Step 2 $a[1] = 8$ and $b[0] = 15$ are compared to copy 8 to c, that is, $c[1] \leftarrow 8$, $i \leftarrow 1+1$, and $k \leftarrow 1+1$.

Step 3 $a[2] = 25$ and $b[0] = 15$ are compared and 15 is copied to c so that $c[2] \leftarrow 15$, $j \leftarrow 0+1$, and $k \leftarrow 2+1$.

Step 4 $a[2] = 25$ and $b[1] = 20$ are compared and 20 is copied to c: $c[3] \leftarrow 20$, $j \leftarrow 1+1$, and $k \leftarrow 3+1$.

Step 5 $a[2] = 25$ and $b[2] = 31$ are compared, and 25 is copied to c: $c[4] \leftarrow 25$, $i \leftarrow 2+1$, and $k \leftarrow 4+1$.

The process continues as follows: comparing $a[3] = 70$ and $b[2] = 31$, $a[3] = 70$ and $b[3] = 50$, and $a[3] = 70$ and $b[4] = 65$ results in $c[5] \leftarrow (b[2] = 31)$, $c[6] \leftarrow (b[3] = 50)$, and $c[7] \leftarrow (b[4] = 65)$, respectively. Because the list b is exhausted, the rest of the list a is then copied to c, $c[8] \leftarrow (a[3] = 70)$ and $c[9] \leftarrow (a[4] = 91)$.

We can now see that the running time of mergesort is much better asymptotically than the naive algorithms that we have previously seen.

Theorem 2.11. The running time of mergesort on an input list of size n is $\Theta(n \log n)$ in the best, worst, and average case.

Proof. The number of comparisons used by mergesort on an input of size n satisfies a recurrence of the form $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + a(n)$ where $1 \leq a(n) \leq n-1$. It is straightforward to show as in Example 1.32 that $T(n)$ is $\Theta(n \log n)$.

The other elementary operations in the divide and combine steps depend on the implementation of the list, but in each case their number is $\Theta(n)$. Thus these operations satisfy a similar recurrence and do not affect the $\Theta(n \log n)$ answer. □

Implementation of mergesort

It is easier to implement the recursive version above for arrays than for linked lists, since splitting an array in the middle is a constant time operation. Algorithm `merge` in Figure 2.3 follows the above description. The first half of the input array, a, from the leftmost index l to the middle index m acts as A, the second half from $m+1$ to the rightmost index r as B, and a separate temporary array t as C. After merging the halves, the temporary array is copied back to the original one, a.

algorithm `mergeSort`
  Input: array $a[0..n-1]$; array indices l, r; array $t[0..n-1]$
  sorts the subarray $a[l..r]$
begin
  if $l < r$ **then**
    $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$
    `mergeSort`(a, l, m, t)
    `mergeSort`$(a, m+1, r, t)$
    `merge`$(a, l, m+1, r, t)$
  end if
end

Figure 2.2: Recursive mergesort for arrays

It is easy to see that the recursive version simply divides the list until it reaches lists of size 1, then merges these repeatedly. We can eliminate the recursion in a straightforward manner. We first merge lists of size 1 into lists of size 2, then lists of size 2 into lists of size 4, and so on. This is often called ***straight mergesort***.

Example 2.12. Starting with the input list $(1,5,7,3,6,4,2)$ we merge repeatedly. The merged sublists are shown with parentheses.

Step 0:	(1)(5)(7)(3)(6)(4)(2)	Step 2:	(1,3,5,7)(2,4,6)
Step 1:	(1,5)(3,7)(4, 6)(2)	Step 3:	(1,2,3,4,5,6,7)

This method works particularly well for linked lists, because the merge steps can be implemented simply by redefining pointers, without using the extra space required when using arrays (see Exercise 2.3.4).

Exercises

Exercise 2.3.1. What is the minimum number of comparisons needed when merging two nonempty sorted lists of total size n into a single list?

Exercise 2.3.2. Give two sorted lists of size 8 whose merging requires the maximum number of comparisons.

```
algorithm merge
    Input: array a[0..n−1]; array indices l, r; array index s; array t[0..n−1]
           merges the two sorted subarrays a[l..s−1] and a[s..r] into a[l..r]
begin
    i ← l; j ← s; k ← l
    while i ≤ s−1 and j ≤ r do
        if a[i] ≤ a[j] then t[k] ← a[i]; k ← k+1; i ← i+1
        else                t[k] ← a[j]; k ← k+1; j ← j+1
        end if
    end while
    while i ≤ s−1 do                copy the rest of the first half
        t[k] ← a[i]; k ← k+1; i ← i+1
    end while
    while j ≤ r do                  copy the rest of the second half
        t[k] ← a[j]; k ← k+1; j ← j+1
    end while
    return a ← t
end
```

Figure 2.3: Linear time merge for arrays

Exercise 2.3.3. The 2-way merge in this section can be generalized easily to a k-way merge for any positive integer k. The running time of such a merge is $c(k-1)n$. Assuming that the running time of insertion sort is cn^2 with the same scaling factor c, analyse the asymptotic running time of the following sorting algorithm (you may assume that n is an exact power of k).

- Split an initial list of size n into k sublists of size $\frac{n}{k}$ each.
- Sort each sublist separately by insertion sort.
- Merge the sorted sublists into a final sorted list.

Find the optimum value of k to get the fastest sort and compare its worst/average case asymptotic running time with that of insertion sort and mergesort.

Exercise 2.3.4. Explain how to merge two sorted linked lists in linear time into a bigger sorted linked list, using only a constant amount of extra space.

2.4 Quicksort

This algorithm is also based on the divide-and-conquer paradigm. Unlike mergesort, quicksort dynamically forms subarrays depending on the input, rather than sorting and merging predetermined subarrays. Almost all the work of mergesort was

in the combining of solutions to subproblems, whereas with quicksort, almost all the work is in the division into subproblems.

Quicksort is very fast in practice on "random" data and is widely used in software libraries. Unfortunately it is not suitable for mission-critical applications, because it has very bad worst case behaviour, and that behaviour can sometimes be triggered more often than an analysis based on random input would suggest.

Basic quicksort is recursive and consists of the following four steps.

- If the size of the list is 0 or 1, return the list. Otherwise:
- Choose one of the items in the list as a ***pivot***.
- Next, ***partition*** the remaining items into two disjoint sublists: reorder the list by placing all items greater than the pivot to follow it, and all elements less than the pivot to precede it.
- Finally, return the result of quicksort of the "head" sublist, followed by the pivot, followed by the result of quicksort of the "tail" sublist.

The first step takes into account that recursive dynamic partitioning may produce empty or single-item sublists. The choice of a pivot at the next step is most critical because the wrong choice may lead to quadratic time complexity while a good choice of pivot equalizes both sublists in size (and leads to "$n \log n$" time complexity). Note that we must specify in any implementation what to do with items equal to the pivot. The third step is where the main work of the algorithm is done, and obviously we need to specify exactly how to achieve the partitioning step (we do this below). The final step involves two recursive calls to the same algorithm, with smaller input.

Analysis of quicksort

All analysis depends on assumptions about the pivot selection and partitioning methods used. In particular, in order to partition a list about a pivot element as described above, we must compare each element of the list to the pivot, so at least $n-1$ comparisons are required. This is the right order: it turns out that there are several methods for partitioning that use $\Theta(n)$ comparisons (we shall see some of them below).

Lemma 2.13. Quicksort is correct.

Proof. We use mathematical induction on the size of the list. If the size is 1, the algorithm is clearly correct. Suppose then that $n \geq 2$ and the algorithm works correctly on lists of size smaller than n. Suppose that a is a list of size n, p is the pivot element, and i is the position of p after partitioning. Due to the partitioning principle of quicksort, all elements of the head sublist are no greater than p, and all elements

of the tail sublist are no smaller than p. By the induction hypothesis, the left and right sublists are sorted correctly, so that the whole list a is sorted correctly. □

Unlike mergesort, quicksort does not perform well in the worst case.

Lemma 2.14. The worst-case time complexity of quicksort is $\Omega(n^2)$.

Proof. The worst case for the number of comparisons is when the pivot is always the smallest or the largest element, one of two sublists is empty, and the second sublist contains all the elements other than the pivot. Then quicksort is recursively called only on this second group. We show that a quadratic number of comparisons are needed; considering data moves or swaps only increases the running time.

Let $T(n)$ denote the worst-case running time for sorting a list containing n elements. The partitioning step needs (at least) $n-1$ comparisons. At each next step for $n \geq 1$, the number of comparisons is one less, so that $T(n)$ satisfies the easy recurrence, $T(n) = T(n-1) + n - 1$; $T(0) = 0$, similar to the basic one, $T(n) = T(n-1) + n$, in Example 1.29. This yields that $T(n)$ is $\Omega(n^2)$. □

However, quicksort is often used in practice, because it is fast on "random" input. The proof of this involves a more complicated recurrence than we have seen so far.

Lemma 2.15. The average-case time complexity of quicksort is $\Theta(n \log n)$.

Proof. Let $T(n)$ denote the average-case running time for sorting a list containing n elements. In the first step, the time taken to compare all the elements with the pivot is linear, cn. If i is the final position of the pivot, then two sublists of size i and $n-1-i$, respectively, are quicksorted, and in this particular case $T(n) = T(i) + T(n-1-i) + cn$. Therefore, the average running time to sort n elements is equal to the partitioning time plus the average time to sort i and $n-1-i$ elements, where i varies from 0 to $n-1$.

All the positions i may be met ***equiprobably***—the pivot picked with the same chance $\frac{1}{n}$—as the final pivot position in a sorted array. Hence, the average time of each recursive call is equal to the average, over all possible subset sizes, of the average running time of recursive calls on sorting both subsets:

$$T(n) = \frac{1}{n} \sum_{i=0}^{n-1} (T(i) + T(n-1-i)) + cn = \frac{2}{n} \sum_{i=0}^{n-1} T(i) + cn$$

To solve this more difficult recurrence, let us rewrite it as

$$nT(n) = 2 \sum_{i=0}^{n-1} T(i) + cn^2$$

and subtract the analogous one with n replaced by $n-1$:

$$(n-1)T(n-1) = 2\sum_{i=0}^{n-2} T(i) + c(n-1)^2.$$

After rearranging similar terms, we obtain $nT(n) = (n+1)T(n-1) + c(2n-1)$ Decomposing the right side via partial fractions we obtain

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{3c}{n+1} - \frac{c}{n}.$$

Telescoping of this recurrence results in the following relationship.

$$\begin{aligned}\frac{T(n)}{n+1} &= \frac{T(0)}{1} + 3c\left(\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\ldots+\frac{1}{n+1}\right) - c\left(\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\ldots+\frac{1}{n}\right)\\ &= c(3H_{n+1} - 3 - H_n)\end{aligned}$$

where H_n is the n-th harmonic number. Thus (see Section D.6) $T(n)$ is $\Theta(n\log n)$. □

This is the first example we have seen of an algorithm whose worst-case performance and average-case performance differ dramatically.

The finer details of the performance of quicksort depend on several implementation issues, which we discuss next.

Implementation of quicksort

There are many variants of the basic quicksort algorithm, and it makes sense to speak of "a quicksort". There are several choices to be made:

- how to implement the list;
- how to choose the pivot;
- how to partition around the pivot.

CHOOSING A PIVOT A ***passive pivot strategy*** of choosing a fixed (for example the first, last, or middle) position in each sublist as the pivot seems reasonable under the assumption that all inputs are equiprobable. The simplest version of quicksort just chooses the first element of the list. We call this the ***naive pivot selection rule***.

For such a choice, the likelihood of a random input resulting in quadratic running time (see Lemma 2.14 above) is very small. However, such a simple strategy is a bad idea. There are two main reasons:

- (nearly) sorted lists occur in practice rather frequently (think of a huge database with a relatively small number of updates daily);

- a malicious adversary may exploit the pivot choice method by deliberately feeding the algorithm an input designed to cause worst case behaviour (a so-called "algorithmic complexity attack").

If the input list is already sorted or reverse sorted, quadratic running time is obtained when actually quicksort "should" do almost no work. We should look for a better method of pivot selection.

A more reasonable choice for the pivot is the middle element of each sublist. In this case an already sorted input list produces the perfect pivot at each recursion. Of course, it is still possible to construct input sequences that result in quadratic running time for this strategy. They are very unlikely to occur at random, but this still leaves the door open for a malicious adversary.

As an alternative to passive strategies, an ***active pivot strategy*** makes good use of the input data to choose the pivot. The best active pivot is the exact median of the list because it partitions the list into (almost) equal sized sublists. But it turns out that the median cannot be computed quickly enough, and such a choice considerably slows quicksort down rather than improving it. Thus, we need a reasonably good yet computationally efficient estimate of the median. (See Section 2.6 for more on the problem of exact computation of the median of a list.)

A reasonable approximation to the true median is obtained by choosing the median of the first, middle, and last elements as the pivot (the so-called ***median-of-three*** method). Note that this strategy is also the best for an already-sorted list because the median of each subarray is recursively used as the pivot.

Example 2.16. Consider the input list of integers $(25,8,2,91,70,50,20,31,15,65)$. Using the median-of-three rule (with the middle of a size n list defined as the element at position $\lfloor n/2 \rfloor$), the first pivot chosen is the median of 25, 70 and 65, namely 65. Thus the left and right sublists after partitioning have sizes 7 and 2 respectively.

The standard choice for pivot would be the left element, namely 25. In this case the left and right sublists after partitioning have sizes 4 and 5 respectively.

The median-of-three strategy does not completely avoid bad performance, but the chances of such a case occurring are much less than for a passive strategy.

Finally, another simple method is to choose the pivot randomly. We can show (by the same calculation as for the average-case running time) that the expected running time on any given input is $\Theta(n \log n)$. It is still possible to encounter bad cases, but these now occur by bad luck, independent of the input. This makes it impossible for an adversary to force worst-case behaviour by choosing a nasty input in advance. Of course, in practice an extra overhead is incurred because of the work needed to generate a "random" number.

A similar idea is to first randomly shuffle the input (which can be done in linear time), and then use the naive pivot selection method. Again, bad cases still occur, but by bad luck only.

PARTITIONING There are several ways to partition with respect to the pivot element in linear time. We present one of them here.

Our partitioning method uses a pointer L starting at the head of the list and another pointer R starting at the end plus one. We first swap the pivot element to the head of the list. Then, while $L < R$, we loop through the following procedure:

- Decrement R until it meets an element less than or equal to p.
- Increment L until it meets an element greater than or equal to p.
- Swap the elements pointed to by L and R.

Finally, once $L = R$, we swap the pivot element with the element pointed to by L.

Example 2.17. Table 2.3 exemplifies the partitioning of a 10-element list. We choose the pivot $p = 31$. The bold element is the pivot; elements in italics are those pointed to by the pointers L and R.

Table 2.3: Partitioning in quicksort with pivot $p = 31$.

Data to sort										**Description**
25	8	2	91	15	50	20	31	70	65	Initial list
31	8	2	91	15	50	20	25	70	65	Move pivot to head
31	8	2	91	15	50	20	*25*	70	65	stop R
31	8	2	*91*	15	50	20	25	70	65	stop L
31	8	2	*25*	15	50	20	*91*	70	65	swap elements L and R
31	8	2	*25*	15	50	*20*	91	70	65	stop R
31	8	2	25	15	*50*	*20*	91	70	65	stop L
31	8	2	25	15	*20*	*50*	91	70	65	swap elements L and R
31	8	2	25	15	*20*	50	91	70	65	stop R
20	8	2	25	15	***31***	50	91	70	65	swap element L with pivot

Lemma 2.18. The partitioning algorithm described above is correct.

Proof. After each swap, we have the following properties:

- each element to the left of L (including the element at L) is less than or equal to p;
- each element to the right of R (including the element at R) is greater than or equal to p.

In the final swap, we swap the pivot element p with an element that is no more than p. Thus all elements smaller than p are now to its left, and all larger are to its right. □

We finish this section by discussing details associated with implementation of lists. Quicksort is much easier to program for arrays than for other types of lists. The pivot selection methods discussed above can be implemented in constant time for arrays, but not for linked lists (for example, try the median-of-three rules with linked lists). The partition step can be performed on linked lists, but the method we have presented requires a doubly-linked list in order to work well, since we need to scan both forward and backward.

In Figure 2.4 we present pseudocode for array-based quicksort. Here i denotes the position of the pivot p before the partition step, and j its position after partitioning. The algorithm works in-place using the input array.

```
algorithm quickSort
    Input: array a[0..n−1]; array indices l, r
           sorts the subarray a[l..r]
begin
    if l < r then
        i ← pivot(a, l, r)      return position of pivot element
        j ← partition(a, l, r, i)   return final position of pivot
        quickSort(a, l, j−1)    sort left subarray
        quickSort(a, j+1, r)    sort right subarray
    end if
    return a
end
```

Figure 2.4: Basic array-based quicksort.

Exercises

Exercise 2.4.1. Analyse in the same way as in Example 2.17 the next partitioning of the left sublist $(20,8,2,25,15)$ obtained in Table 2.3.

Exercise 2.4.2. Show in detail what happens in the partitioning step if all keys are equal. What would happen if we change the partition subroutine so that L skipped over keys equal to the pivot? What if R did? What if both did? (hint: the algorithm would still be correct, but its performance would differ).

Exercise 2.4.3. Analyse partitioning of the list $(25,8,8,8,8,8,2)$ if the pointers L and R are advanced while $L \leq p$ and $p \leq R$, respectively, rather than stopping on equality as in Table 2.3.

Exercise 2.4.4. Suppose that we implement quicksort with the naive pivot choice rule and the partitioning method of the text. Find an array of size 8, containing each

integer $1,\dots 8$ exactly once, that makes quicksort do the maximum number of comparisons. Find one that makes it do the minimum number of comparisons.

2.5 Heapsort

This algorithm is an improvement over selection sort that uses a more sophisticated data structure to allow faster selection of the minimum element. It, like mergesort, achieves $\Theta(n \log n)$ in the worst case. Nothing better can be achieved by an algorithm based on pairwise comparisons, as we shall see later, in Section 2.7.

A heap is a special type of tree. See Section D.7 if necessary for general facts about trees.

Definition 2.19. A ***complete binary tree*** is a binary tree which is completely filled at all levels except, possibly, the bottom level, which is filled from left to right with no missing nodes.

In such a tree, each leaf is of depth h or $h-1$, where h is the tree height, and each leaf of height h is on the left of each leaf of height $h-1$.

Example 2.20. Figure 2.5 demonstrates a complete binary tree with ten nodes. If the node J had been the right child of the node E, the tree would have not been complete because of the left child node missed at the bottom level.

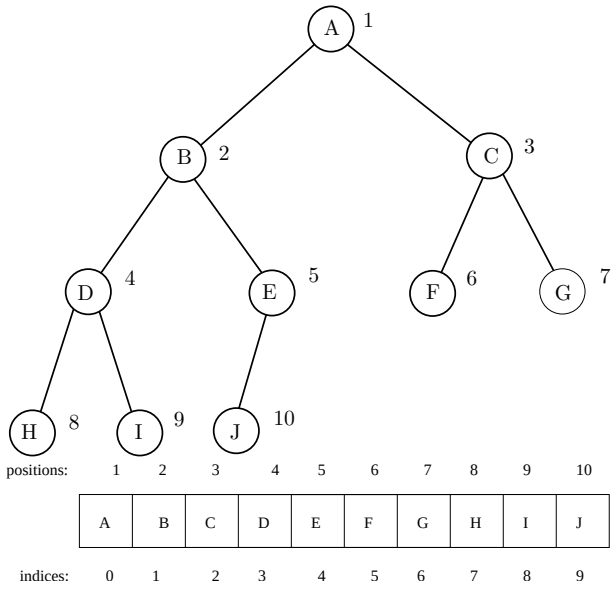


Figure 2.5: Complete binary tree and its array representation.

A level-order traversal of a complete binary tree is easily stored in a linear array as shown in Figure 2.5. In this book we mostly use indices 0 to $n-1$ for the array positions 1 to n in order to match popular programming languages such as Java, C, or C++. But the array representation of a binary tree is described more conveniently when both the node numbers and the array positions are changing from 1 to n. Therefore, when a position p in an array a is mentioned below, we bear in mind an array element $a[i]$ with the index $i = p-1$.

The node in position p has its parent node, left child, and right child in positions $\lfloor p/2 \rfloor$, $2p$, and $2p+1$, respectively. The leaves have no children so that for each leaf node q, the child position $2q$ exceeds the number of nodes n in the tree: $2q > n$.

Example 2.21. In Figure 2.5, the node in position $p = 1$ is the root with no parent node. The nodes in positions from 6 to 10 are the leaves. The root has its left child in position 2 and its right child in position 3. The nodes in positions 2 and 3 have their left child in position 4 and 6 and their right child in position 5 and 7, respectively. The node in position 4 has a left child in position 8 and a right child in position 9, and the node 5 has only a left child, in position 10.

Definition 2.22. A (maximum) ***heap*** is a complete binary tree having a key associated with each node, the key of each parent node being greater than or equal to the keys of its child nodes.

The heap order provides easy access to the maximum key associated with the root.

Example 2.23. Figure 2.6 illustrates a maximum heap. Of course, we could just as easily have a minimum heap where the key of the parent node is less than or equal to the keys of its child nodes. Then the minimum key is associated with the root.

The ***heapsort*** algorithm now works as follows. Given an input list, build a heap by successively inserting the elements. Then delete the maximum repeatedly (arranging the elements in the output list in reverse order of deletion) until the heap is empty. Clearly, this is a variant of selection sort that uses a different data structure.

Analysis of heapsort

Heapsort is clearly correct for the same reason as selection sort. To analyse its performance, we need to analyse the running time of the insertion and deletion operations.

Lemma 2.24. The height of a complete binary tree with n nodes is at most $\lfloor \lg n \rfloor$.

Proof. Depending on the number of nodes at the bottom level, a complete tree of height h contains between 2^h and $2^{h+1}-1$ nodes, so that $2^h \le n < 2^{h+1}$, or $h \le \lg n < h+1$. □

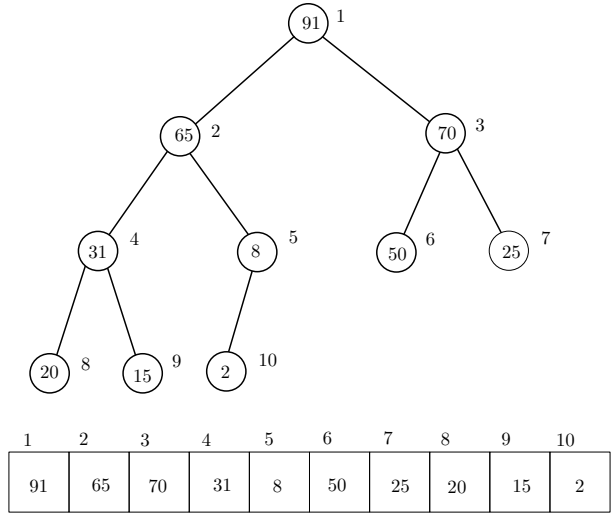


Figure 2.6: Maximum heap and its array representation.

Lemma 2.25. Insertion of a new node into a heap takes logarithmic time.

Proof. To add one more node to a heap of n elements, a new, $(n+1)$-st, leaf position has to be created. The new node with its associated key is placed first in this leaf. If the inserted key preserves the heap order, the insertion is completed. Otherwise, the new key has to swap with its parent, and this process of ***bubbling up***, (or *percolating up*) the key is repeated toward the root until the heap order is restored. Therefore, there are at most h swaps where h is the heap height, so that the running time is $O(\log n)$. □

Example 2.26. To insert an 11th element, 75, into the heap in Figure 2.6 takes three steps.

- Position 11 to initially place the new key, 75, is created.
- The new key is swapped with its parent key, 8, in position $5 = \lfloor 11/2 \rfloor$ to restore the heap order.
- The same type of swap is repeated for the parent key, 65, in position $2 = \lfloor 5/2 \rfloor$. Because the heap order condition is now satisfied, the process terminates.

This is shown in Table 2.4. The elements moved to restore the heap order are italicized.

Table 2.4: Inserting a new node with the key 75 in the heap in Figure 2.6.

Position	1	2	3	4	5	6	7	8	9	10	11
Index	0	1	2	3	4	5	6	7	8	9	10
Array at step 1	91	65	70	31	8	50	25	20	15	2	**75**
Array at step 2	91	65	70	31	**75**	50	25	20	15	2	*8*
Array at step 3	91	**75**	70	31	*65*	50	25	20	15	2	8

Lemma 2.27. Deletion of the maximum key from a heap takes logarithmic time in the worst case.

Proof. The maximum key occupies the tree's root, that is, position 1 of the array. The deletion reduces the heap size by 1 so that its last leaf node has to be eliminated. The key associated with this leaf replaces the deleted key in the root and then is ***percolated down*** the tree. First, the new root key is compared to each child and swapped with the larger child if at least one child is greater than the parent. This process is repeated until the order is restored. Therefore, there are h moves in the worst case where h is the heap height, and the running time is $O(\log n)$. □

Because we percolate down the previous leaf key, the process usually terminates at or near the leaves.

Example 2.28. To delete the maximum key, 91, from the heap in Figure 2.6, takes three steps, as follows.

- Key 2 from the eliminated position 10 is placed at the root.
- The new root key is compared to its children 65 and 70 in positions $2 = 2 \cdot 1$ and $3 = 2 \cdot 1 + 1$, respectively. To restore the heap order, it is swapped with its larger child, 70.
- The same swap is repeated for the children 50 and 25 in positions $6 = 2 \cdot 3$ and $7 = 2 \cdot 3 + 1$. Because the heap order is now correct, the process terminates.

Table 2.5: Deletion of the maximum key from the heap in Figure 2.6.

Position	1	2	3	4	5	6	7	8	9
Index	0	1	2	3	4	5	6	7	8
Array at step 1	**2**	65	70	31	8	50	25	20	15
Array at step 2	*70*	65	**2**	31	8	50	25	20	15
Array at step 3	*70*	65	*50*	31	8	**2**	25	20	15

See also Table 2.5. The leaf key replacing the root key is boldfaced, and the moves to restore the heap are italicized.

Lemma 2.29. Heapsort runs in time in $\Theta(n \log n)$ in the best, worst, and average case.

Proof. The heap can be constructed in time $O(n \log n)$ (in fact it can be done more quickly as seen in Lemma 2.31 but this does not affect the result). Heapsort then repeats n times the deletion of the maximum key and restoration of the heap property. In the best, worst, and average case, each restoration is logarithmic, so the total time is $\log(n) + \log(n-1) + \ldots + \log(1) = \log n!$ which is $\Theta(n \log n)$. □

Implementation of heapsort

There are several improvements that can be made to the basic idea above. First, the heap construction phase can be simplified. There is no need to maintain the heap property as we add each element, since we only require the heap property once the heap is fully built. A nice recursive approach is shown below. Second, we can eliminate the recursion. Third, everything can be done in-place starting with an input array.

We consider each of these in turn.

A heap can be considered as a recursive structure of the form left subheap ← root → right subheap, built by a recursive "heapifying" process. The latter assumes that the heap order exists everywhere except at the root and percolates the root down to restore the total heap order. Then it is recursively applied to the left and right subheaps.

Lemma 2.30. A complete binary tree satisfies the heap property if and only if the maximum key is at the root, and the left and right subtrees of the root also satisfy the heap property with respect to the same total order.

Proof. Suppose that T is a complete binary tree that satisfies the heap condition. Then the maximum key is at the root. The left and right subtrees at the root are also complete binary trees, and they inherit the heap property from T.

Conversely, suppose that T is a complete binary tree with the maximum at the root and such that the left and right subtrees are themselves heaps. Then the value at the root is at least as great as that of the keys of the children of the root. For each other node of T, the same property holds by our hypotheses. Thus the heap property holds for all nodes in the tree. □

For example, in Figure 2.6 we can see this fact clearly.

Lemma 2.31. A heap can be constructed from a list of size n in $\Theta(n)$ time.

Proof. Let $T(h)$ denote the worst-case time to build a heap of height at most h. To construct the heap, each of the two subtrees attached to the root are first transformed into heaps of height at most $h-1$ (the left subtree is always of height $h-1$, whereas the right subtree could be of lesser height, $h-2$). Then in the worst case the root percolates down the tree for a distance of at most h steps that takes time

$O(h)$. Thus heap construction is asymptotically described by the recurrence similar to Example 1.27, $T(h) = 2T(h-1) + ch$ and so $T(h)$ is $O(2^h)$. Because a heap of size n is of height $h = \lfloor \lg n \rfloor$, we have $2^h \leq n$ and thus $T(h)$ is $O(n)$. But since every element of the input must be inspected, we clearly have a lower bound of $\Omega(n)$, which yields the result. □

Now we observe that the recursion above can be eliminated. The key at each position p percolates down only after all its descendants have been already processed by the same percolate-down procedure. Therefore, if this procedure is applied to the nodes in reverse level order, the recursion becomes unnecessary. In this case, when the node p has to be processed, all its descendants have been already processed. Because leaves need not percolate down, a non-recursive heapifying process by percolating nodes down can start at the non-leaf node with the highest number. This leads to an extremely simple algorithm for converting an array into a heap (see the first **for** loop in Figure 2.7).

Figure 2.7 presents the basic pseudocode for heapsort (for details of the procedure `percolateDown`, see the Java code in Section A.1). After each deletion, the heap size decreases by 1, and the emptied last array position is used to place the just deleted maximum element. After the last deletion the array contains the keys in ascending sorted order. To get them in descending order, we have to build a minimum heap instead of the above maximum heap.

The first for-loop converts an input array a into a heap by percolating elements down. The second for-loop swaps each current maximum element to be deleted with the current last position excluded from the heap and restores the heap by percolating each new key from the root down.

```
algorithm heapSort
    Input: array a[0..n − 1]
begin
    for i ← ⌊n/2⌋ − 1 while i ≥ 0 step i ← i − 1 do
    percolateDown(a, i, n)          build a heap
    end for
    for i ← n − 1 while i ≥ 1 step i ← i − 1 do
        swap(a[0], a[i])            delete the maximum
        percolateDown(a, 0, i)      restore the heap
    end for
end
```

Figure 2.7: Heapsort.

Example 2.32. Table 2.6 presents successive steps of heapsort on the input array $a = [70, 65, 50, 20, 2, 91, 25, 31, 15, 8]$. In the table, the keys moved are italicized and the items sorted are boldfaced.

Table 2.6: Successive steps of heapsort.

Position	1	2	3	4	5	6	7	8	9	10
Index	0	1	2	3	4	5	6	7	8	9
Initial array	70	65	50	20	2	91	25	31	15	8
Building max heap	70	65	50	20	*8*	91	25	31	15	*2*
	70	65	50	*31*	8	91	25	*20*	15	2
	70	65	*91*	31	8	*50*	25	20	15	2
	70	*65*	91	31	8	50	25	20	15	2
	91	65	*70*	31	8	50	25	20	15	2
Max heap	91	65	70	31	8	50	25	20	15	2
Deleting max 1	*2*	65	70	31	8	50	25	20	15	**91**
Restoring heap 1-9	*70*	65	*50*	31	8	*2*	25	20	15	**91**
Deleting max 2	*15*	65	50	31	8	2	25	20	**70**	**91**
Restoring heap 1-8	*65*	*31*	50	*20*	8	2	25	*15*	**70**	**91**
Deleting max 3	*15*	31	50	20	8	2	25	**65**	**70**	**91**
Restoring heap 1-7	*50*	31	*25*	20	8	2	*15*	**65**	**70**	**91**
Deleting max 4	*15*	31	25	20	8	2	**50**	**65**	**70**	**91**
Restoring heap 1-6	*31*	*20*	25	*15*	8	2	**50**	**65**	**70**	**91**
Deleting max 5	*2*	20	25	15	8	**31**	**50**	**65**	**70**	**91**
Restoring heap 1-5	*25*	20	*2*	15	8	**31**	**50**	**65**	**70**	**91**
Deleting max 6	*8*	20	2	15	**25**	**31**	**50**	**65**	**70**	**91**
Restoring heap 1-4	*20*	*15*	2	*8*	**25**	**31**	**50**	**65**	**70**	**91**
Deleting max 7	*8*	15	2	**20**	**25**	**31**	**50**	**65**	**70**	**91**
Restoring heap 1-3	*15*	*8*	2	**20**	**25**	**31**	**50**	**65**	**70**	**91**
Deleting max 8	*2*	8	**15**	**20**	**25**	**31**	**50**	**65**	**70**	**91**
Restoring heap 1-2	*8*	*2*	**15**	**20**	**25**	**31**	**50**	**65**	**70**	**91**
Deleting max 9	*2*	**8**	**15**	**20**	**25**	**31**	**50**	**65**	**70**	**91**

PRIORITY-QUEUE SORT A heap is a particular implementation of the *priority queue* ADT (see Section C.1). There are many other such implementations. From an abstract point of view, heapsort simply builds a priority queue by inserting all the elements of the list to be sorted, then deletes the highest priority element repeatedly. Any priority queue implementation can be used to sort in the same way.

For example, a very simple implementation of a priority queue is via an unsorted list. In this case, building the queue is trivial, and the sorting algorithm is exactly selection sort. Another simple implementation is via a sorted list. In this case, the

algorithm is just insertion sort (we don't really need to delete the elements since the resulting list after building the priority queue is the output we are seeking).

There are many more sophisticated implementations of priority queues. They are useful not only for sorting, but for several important graph algorithms covered in this book, and also for applications such as discrete event simulation. There is still active research on finding better priority queue implementations.

Given that we can build a priority queue (such as a heap) in linear time, it is natural to ask whether we could implement a priority queue in such a way that the successive deletions can be done in better than $O(n \log n)$ time, thus yielding a sorting algorithm that is asymptotically faster in the worst case than any we have yet seen. Perhaps surprisingly, the answer is no for comparison-based algorithms, as we see in the next section.

Exercises

Exercise 2.5.1. Insert a 12th element, 85, into the final heap in Table 2.4.

Exercise 2.5.2. Delete the maximum key from the final heap in Table 2.5 and restore the heap order.

Exercise 2.5.3. Convert the array $[10, 20, 30, 40, 50, 60, 70, 80, 90]$ into a heap using the algorithm in Lemma 2.31.

Exercise 2.5.4. Present in the same manner the successive steps of heapsort on the already sorted input array $a = [2, 8, 15, 20, 25, 31, 50, 65, 70, 91]$.

Exercise 2.5.5. Determine whether heapsort is stable and compare it to insertion sort, mergesort, and quicksort regarding this property.

Exercise 2.5.6. Determine whether the running time of heapsort on an already sorted array of size n differs significantly from the average-case running time.

2.6 Data selection

Data selection is closely related to data sorting. The latter arranges an array in order whereas the former has to find only the kth smallest item (the item of ***rank*** k, also known as the kth ***order statistic***). We have seen (selection sort and heapsort) that if we can select, then we can sort. The converse is true too: given a list of n items, we can sort it and then select the desired order statistic easily. The obvious question is: can we perform selection without sorting, and asymptotically faster?

If for example $k = 1$ or $k = n$, then the answer is obviously yes (Exercise 2.6.1). However, if we require the median ($k = \lceil n/2 \rceil$) the answer is not at all obvious. For example, building a heap and repeatedly extracting the minimum would take time in $\Omega(n \log n)$, which is no better than sorting the entire list.

A variation of quicksort works well on this problem, *in the average case for random data*. However it has all the drawbacks of quicksort, and in the worst case its performance degenerates badly.

The idea of ***quickselect*** is that after the partitioning stage of quicksort, we know which of the two parts of the partition holds the element we are seeking, and so we can eliminate one recursive call. In other words, it proceeds as follows.

- If the size of the list is 0, return "not found"; if the size is 1, return the element of that list. Otherwise:
- Choose one of the items in the list as a pivot.
- Partition the remaining items into two disjoint sublists: reorder the list by placing all items greater than the pivot to follow it, and all elements less than the pivot to precede it. Let j be the index of the pivot after partitioning.
- If $k < j$, then return the result of quickselect on the "head" sublist; otherwise if $k = j$, return the element p; otherwise return the result of quickselect on the "tail" sublist.

Analysis of quickselect

Correctness of quickselect is established just as for quicksort (see Exercise 2.6.2). In the worst case, the running time can be quadratic; for example, if the input list is already sorted and we use the naive pivot selection rule, then to find the maximum element takes quadratic time.

Theorem 2.33. The average-case time complexity of quickselect is $\Theta(n)$.

Proof. Let $T(n)$ denote the average time to select the k-th smallest element among n elements, for fixed k where the average is taken over all possible input sequences. Partitioning uses no more than cn operations and forms two subarrays, of size i and $n-1-i$, respectively, where $0 \leq i < n$.

As in quicksort, the final pivot position in the sorted array has equal probability, $\frac{1}{n}$, of taking each value of i. Then $T(n)$ averages the average running time for all the above pairs of the subarrays over all possible sizes. Because only one subarray from each pair is recursively chosen, the average running time for the pair of size i and $n-1-i$ is $(T(i)+T(n-1-i))/2$ so that

$$T(n) = \frac{1}{2n}\sum_{i=0}^{n-1}(T(i)+T(n-1-i)) + cn = \frac{1}{n}\sum_{i=0}^{n-1}T(i) + cn$$

As for quicksort, the above recurrence can be rewritten as $nT(n) = \sum_{i=0}^{n-1}T(i) + cn^2$ and subtracting the analogous equation $(n-1)T(n-1) = \sum_{i=0}^{n-2}T(i) + c\cdot(n-1)^2$ and rearranging, we are eventually led to the familiar recurrence $T(n) = T(n-1) + c$ and can see that $T(n)$ is $\Theta(n)$. □

Implementation of quickselect

The only change from quicksort is that instead of making two recursive calls on the left and right subarrays determined by the pivot, quickselect chooses just one of these subarrays.

```
algorithm quickSelect
    Input: array a[0..n]; array indices l, r; integer k
           finds kth smallest element in the subarray a[l..r]
begin
    if l ≤ r then
        i ← pivot(a, l, r)      return position of pivot element
        j ← partition(a, l, r, i)   return final position of pivot
        q ← j − l + 1           the rank of the pivot in a[l..r]
        if k = q then return a[j]
        else if k < q then return quickSelect(a, l, j − 1, k)
        else return quickSelect(a, j + 1, r, k − q)
    end if
    else return "not found"
end
```

Figure 2.8: Basic array-based quickselect.

Figure 2.8 presents a basic array-based implementation. The algorithm processes a subarray $a[l..r]$, where $0 \le l \le r \le n-1$, of an input array a of size n, assuming that the desired rank k is in the correct range. A search in the whole array is performed with the input indices $l = 0$ and $r = n-1$. The search fails only if k is outside the correct range (including the case where the subarray is empty). Section A.1 contains a Java implementation that uses median-of-three pivoting.

Exercises

Exercise 2.6.1. Give a simple nonrecursive algorithm to find the maximum of a list of n elements. How many comparisons does it do in the worst case? Is this the best we can hope for?

Exercise 2.6.2. Prove that quickselect is correct.

Exercise 2.6.3. To find p keys of fixed ranks, $k_1, \ldots, k_p$, in an unordered array, we can either (i) run quickselect p times or (ii) use quicksort once to order the array and simply fetch the desired p keys. Find a condition (in terms of p and n) when on the average the second variant is more time-efficient (time to fetch array elements can be ignored). Which variant is better if n is 1 million and $p = 10$?

Exercise 2.6.4. Investigate "heapselect" and its time complexity. Do the same for "mergeselect".

2.7 Lower complexity bound for sorting

All the above sorting methods have average and worst time complexity bounds in $\Omega(n \log n)$. One of most fundamental theorems in algorithm analysis shows that no comparison-based sorting algorithm can have a better asymptotic lower bound. The proof uses the idea of a n-element binary ***decision tree*** representation of any sorting of n items by pairwise comparisons.

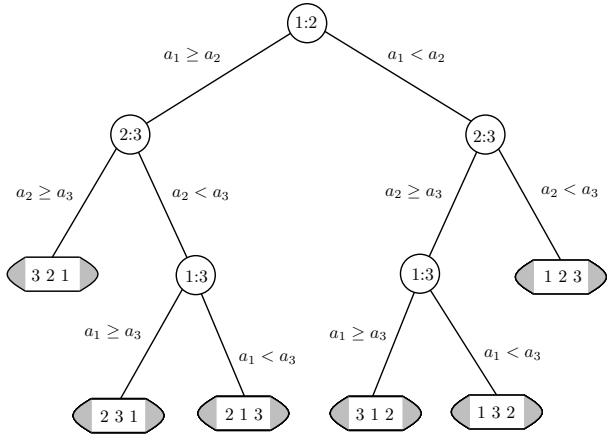


Figure 2.9: Decision tree for $n = 3$.

Example 2.34. A decision tree for three ($n = 3$) elements a_1, a_2, a_3 is shown in Figure 2.9. Each internal tree node represents a pairwise comparison and decision made at a particular step of sorting (in Figure 2.9, $i : j$ denotes the comparison between the elements $a[i]$ and $a[j]$). Two downward arcs from the node indicate two possible results of the comparison: $a[i] \geq a[j]$ or $a[i] < a[j]$. Leaves represent the sorted lists. In Figure 2.9, each triple ijk denotes the sorted list (a_i, a_j, a_k) obtained from the initial list (a_1, a_2, a_3). For instance, 123, 213, or 321 mean that the final list is (a_1, a_2, a_3), (a_2, a_1, a_3), or (a_3, a_2, a_1), respectively. Because any of the $n!$ permutations of n arbitrary items $a_1, a_2, \ldots, a_n$ may be obtained as the result of the sort, the decision tree must have at least $n!$ leaves.

The path length from the root to a leaf in the decision tree is equal to the number of comparisons to be performed in order to obtain the sorted array at the leaf. Therefore, the longest path (the height of the tree) equals the number of comparisons in the worst case. For example, 3 elements can be sorted with no more than 3 comparisons because the height of the 3-element tree in Figure 2.9 is equal to 3.

Theorem 2.35. Every comparison-based sorting algorithm takes $\Omega(n \log n)$ time in the worst case.

Proof. We first claim that each binary tree of height h has at most 2^h leaves. Once this claim is established, we proceed as follows. The least value h such that $2^h \geq n!$ has the lower bound $h \geq \lg(n!)$ which is in $\Omega(n \log n)$ (the asymptotic result follows from Section D.6). This will prove the theorem.

To prove the above claim about tree height, we use mathematical induction on h. A tree of height 0 has obviously at most 2^0 leaves. Now suppose that $h \geq 1$ and that each tree of height $h-1$ has at most 2^{h-1} leaves. The root of a decision tree of height h is linked to two subtrees, being each at most of height $h-1$. By the induction hypothesis, each subtree has at most 2^{h-1} leaves. The number of leaves in the whole decision tree is equal to the total number of leaves in its subtrees, that is, at most $2^{h-1} + 2^{h-1} = 2^h$. □

This result shows that heapsort and mergesort have asymptotically optimal worst-case time complexity for comparison-based sorting.

As for average-case complexity, one can also prove the following theorem by using the decision tree idea. Since we are now at the end of our introductory analysis of sorting, we omit the proof and refer the reader to the exercises, and to more advanced books.

Theorem 2.36. Every comparison-based sorting algorithm takes $\Omega(n \log n)$ time in the average case.

Proof. Try to prove this yourself (see Exercise 2.7.1). □

Exercises

Exercise 2.7.1. Prove Theorem 2.36. The following hints may be useful. First show that the sum of all depths of leaves in a binary decision tree with k leaves is at least $k \lg k$. Do this by induction on k, using the recursive structure of these trees. Then apply the above inequality with $k = n!$.

Exercise 2.7.2. Consider the following sorting method (often called ***counting sort***) applied to an array $a[n]$ all of whose entries are integers in the range 1..1000. Introduce a new array $t[1000]$ all of whose entries are initially zero. Scan through the array $a[n]$ and each time an integer i is found, increment the counter $t[i-1]$ by 1. Once this is complete, loop through $0 \leq i \leq 999$ and print out $t[i]$ copies of integer $i+1$ at each step.

What is the worst-case time complexity of this algorithm? How do we reconcile this with Theorem 2.35?

2.8 Notes

It was once said that sorting consumes 25% of all CPU time worldwide. Whatever the true proportion today, sorting clearly remains a fundamental problem to be solved in a wide variety of applications.

Given the rise of object-oriented programming languages, comparison-based sorting algorithms are perhaps even more important than in the past. In practice the time taken to perform a basic comparison operation is often much more than that taken to swap two objects: this differs from the case of, say, 32-bit integers, for which most analysis was done in the past.

Shellsort was proposed by D. Shell in 1959, quicksort by C. A. R. Hoare in 1960, mergesort in 1945 by J. von Neumann, and heapsort in 1964 by J. W. J. Williams. Insertion sort and the other quadratic algorithms are very much older.

At the time of writing, versions of mergesort are used in the standard libraries for the languages Python, C++ and Java, and a hybrid of quicksort and heapsort is used by C++.

We have not discussed whether there is an algorithm that will find the median (or any other given order statistic) in worst-case linear time. For a long time this was unknown, but the answer was shown to be yes in 1973 by Blum, Floyd, Pratt, Rivest and Tarjan. The algorithm is covered in more advanced books and is fairly complicated.

Chapter 3

Efficiency of Searching

Searching in a large database is a fundamental computational task. Usually the information is partitioned into ***records*** and each record has a ***key*** to use in the search. Suppose we have a data structure D of records. The search problem is as follows: given a search key k, either return the record associated with k in D (a ***successful search***), or indicate that k is not found, without altering D (an ***unsuccessful search***). If k occurs more than once, return any occurrence. The purpose of the search is usually to access data in the record (not merely the key). Simple examples are searching a phone directory or a dictionary.

In this chapter we discuss searching in a general data structure as above. The more specialized problem of searching for a pattern in a text string is covered in the printed version of this textbook.

3.1 The problem of searching

The most general data structure that allows searching is called an *associative array*, *table* or *dictionary*.

In the general case, a key and a value are linked by *association*. An ***associative array*** or ***dictionary*** is an abstract data type relating a disjoint set of keys to an arbitrary set of values. Keys of entries of an associative array may not have any ordering relation and may be of unknown range. There is no upper bound on the size of this structure, so that it can maintain an arbitrary number of different pieces of information simultaneously. The analogy with a conventional word dictionary may be misleading, because the latter has a lexicographical order while dictionary structures need not be ordered.

Another name for this data type is ***table***. It takes its name from program compilation where a symbol table is used to record variables of a program, together with their type, address, value, etc. Symbol table operations are also fundamental to database systems.

Definition 3.1. The ***table*** ADT is a set of ordered pairs (table entries) of the form (k, v) where k is an unique *key* and v is a data *value* associated with the key k. Each key uniquely identifies its entry, and table entries are distinguished by keys because no two distinct entries have identical keys.

Basic operations defined for this structure are: construct (initialize) an empty table, enumerate table entries, search for an entry, insert, delete, retrieve, and update table entries.

Abstractly, a table is a mapping (function) from keys to values. Given a search key k, ***table search*** has to find the table entry (k, v) containing that key. The found entry may be retrieved, or removed (deleted) from the table, or its value, v, may be updated. If the table has no such entry, a new entry with key k may be created and inserted in the table. Operations on a table also initialize a table to the empty one or indicate that an entry with the given key is absent. Insertions and deletions modify the mapping of keys onto values specified by the table.

Example 3.2. Table 3.1 presents a very popular (at least in textbooks on algorithms and data structures) table having three-letter identifiers of airports as keys and associated data such as airport locations, as values. Each identifier has a unique integer representation $k = 26^2 c_0 + 26c_1 + c_2$ where the c_i; $i = 0, 1, 2$, are ordinal numbers of letters in the English alphabet (A corresponds to 0, B to 1, ..., Z to 25). For example, AKL corresponds to $26^2 \cdot 0 + 26 \cdot 10 + 11 = 271$. In total, there are $26^3 = 17576$ possible different keys and entries.

Table 3.1: A map between airport codes and locations.

Key		**Associated value** v		
Code	k	City	Country	State / Place
AKL	271	Auckland	New Zealand	
DCA	2080	Washington	USA	District of Columbia (D.C.)
FRA	3822	Frankfurt	Germany	Hesse
GLA	4342	Glasgow	UK	Scotland
HKG	4998	Hong Kong	China	
LAX	7459	Los Angeles	USA	California
SDF	12251	Louisville	USA	Kentucky
ORY	9930	Paris	France	

As can be seen from this example, we may map the keys to integers.

We deal with both ***static*** (where the database is fixed in advance and no insertions, deletions or updates are done) and ***dynamic*** (where insertions, deletions or updates are allowed) implementations of the table ADT.

In all our implementations of the table ADT, we may simplify the analysis as follows. We use lists and trees as our basic containers. We treat each query or update of a list element or tree node, or comparison of two of them, as an elementary operation. The following lemma summarizes some obvious relationships.

Lemma 3.3. Suppose that a table is built up from empty by successive insertions, and we then search for a key k uniformly at random. Let $T_{ss}(k)$ (respectively $T_{us}(k)$) be the time to perform successful (respectively unsuccessful) search for k. Then

- the time taken to retrieve, delete, or update an element with key k is at least $T_{ss}(k)$;
- the time taken to insert an element with key k is at least $T_{us}(k)$;
- $T_{ss}(k) \leq T_{us}(k)$.

In addition

- the worst case value for $T_{ss}(k)$ equals the worst case value for $T_{us}(k)$;
- the average value of $T_{ss}(k)$ equals one plus the average of the times for the unsuccessful searches undertaken while building the table.

Proof. To insert a new element, we first try to find where it would be if it were contained in the data structure, and then perform a single insert operation into the container. To delete an element, we first find it, and then perform a delete operation on the container. Analogous statements hold for updating and retrieval. Thus for a given state of the table formed by insertions from an empty table, the time for successful search for a given element is the time that it took for unsuccessful search for that element, as we built the table, plus one. This means that the time for unsuccessful search is always at least the time for successful search for a given element (the same in the worst case), and the average time for successful search for an element in a table is the average of all the times for unsuccessful searches plus one. □

If the data structure used to implement a table arranges the records in a list, the efficiency of searching depends on whether the list is sorted. In the case of the telephone book, we quickly find the desired phone number (data record) by name (key). But it is almost hopeless to search directly for a phone number unless we have a special reverse directory where the phone number serves as a key. We discuss unsorted lists in the Exercises below, and sorted lists in the next section.

Exercises

Exercise 3.1.1. The ***sequential search*** algorithm simply starts at the head of a list and examines elements in order until it finds the desired key or reaches the end of the list. An array-based version is shown in Figure 3.1.

```
algorithm sequentialSearch
    Input: array a[0..n − 1]; key k
begin
    for i ← 0 while i < n step i ← i + 1 do
        if a[i] = k then return i
    end for
    return not found
end
```

Figure 3.1: A sequential search algorithm.

Show that both successful and unsuccessful sequential search in a list of size n have worst-case and average-case time complexity $\Theta(n)$.

Exercise 3.1.2. Show that sequential search is slightly more efficient for sorted lists than unsorted ones. What is the time complexity of successful and unsuccessful search?

3.2 Sorted lists and binary search

A sorted list implementation allows for much better search method that uses the divide-and-conquer paradigm. The basic idea of ***binary search*** is simple. Let k be the desired key for which we want to search.

- If the list is empty, return "not found". Otherwise:
- Choose the key $a[m]$ of the middle element of the list. If $a[m] = k$, return its record; if $a[m] > k$, make a recursive call on the head sublist; if $a[m] < k$, make a recursive call on the tail sublist.

Example 3.4. Figure 3.2 illustrates binary search for the key $k = 42$ in a list of size 16. At the first iteration, the search key 42 is compared to the key $a[7] = 53$ in the middle position $m = \lfloor (0+15)/2 \rfloor = 7$.

Because $42 < 53$, the second iteration explores the first half of the list with positions $0, \ldots, 6$. The search key is compared to the key $a[3] = 33$ in the middle position $m = \lfloor (0+6)/2 \rfloor = 3$. Because $42 > 33$, the third iteration explores the second half of the current sublist with positions $4, \ldots, 6$. The comparison to the key $a[5] = 49$ in the middle position $m = \lfloor (4+6)/2 \rfloor = 5$ reduces the search range to a single key, $a[4] = 42$, because now $l = r = m = 4$. Because the single entry is equal to the search key, the algorithm returns the answer 4.

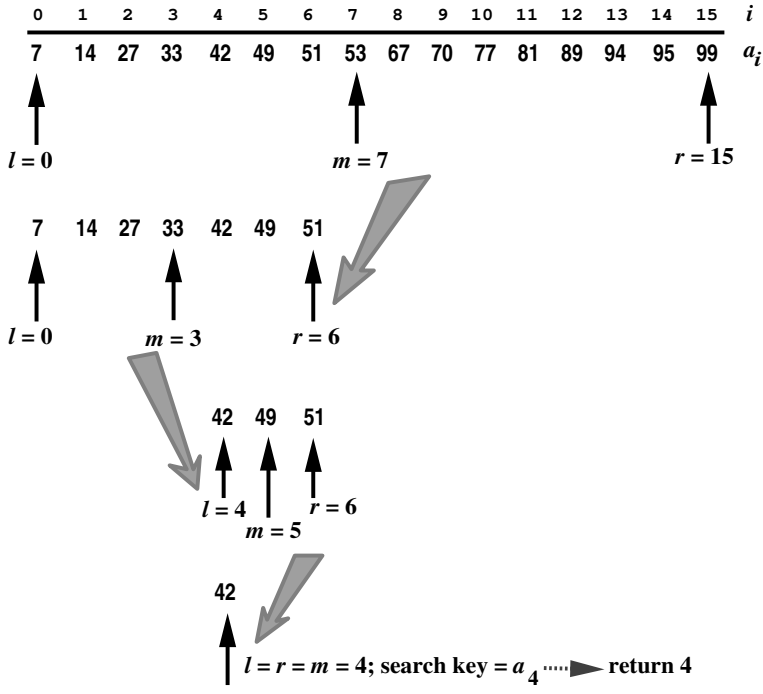


Figure 3.2: Binary search for the key 42.

Analysis of sorted list-based tables

Binary search is readily seen to be correct by induction on the size of the list (Exercise 3.2.3).

The performance of binary search on an array is much better than on a linked list because of the constant time access to a given element.

Lemma 3.5. Consider a sorted list implementation of the table ADT.

Using an array, both successful and unsuccessful search take time in $\Theta(\log n)$, in both the average and worst case, as do retrieval and updating. Insertion and deletion take time in $\Theta(n)$ in the worst and average case.

Using a linked list, all the above operations take time in $\Theta(n)$.

Proof. Unsuccessful binary search takes $\lceil \lg(n+1) \rceil$ comparisons in every case, which is $\Theta(\log n)$. By Lemma 3.3, successful search also takes time in $\Theta(\log n)$ on average and in the worst case. Insertion and deletion in arrays takes $\Theta(n)$ time on average and in the worst case.

For linked lists, the searches take time in $\Theta(n)$ and the list insertion and deletion take constant time. □

Binary search performs a predetermined sequence of comparisons depending on the data size n and the search key k. This sequence is better analysed when a sorted list is represented as a ***binary search tree***. For simplicity of presentation, we suppress the data records and make all the keys integers. Background information on trees is found in Section D.7.

Definition 3.6. A ***binary search tree*** (BST) is a binary tree that satisfies the following ordering relation: for every node v in the tree, the values of all the keys in the left subtree are smaller than (or equal, if duplicates allowed) to the key in v and the values of all the keys in the right subtree are greater than the key in v.

In line with the ordering relation, all the keys can be placed in sorted order by traversing the tree in the following way: recursively visit, for each node, its left subtree, the node itself, then its right subtree (this is the so-called ***inorder*** traversal). The relation is not very desirable for duplicate keys; for exact data duplicates, we should have one key and attach to it the number of duplicates.

The element in the middle position, $m_0 = \lfloor (n-1)/2 \rfloor$, of a sorted array is the root of the tree representation. The lower subrange, $[0,\ldots,m_0-1]$, and the upper subrange, $[m_0+1,\ldots,n-1]$, of indices are related to the left and right arcs from the root. The elements in their middle positions, $m_{\text{left},1} = \lfloor (m_0-1)/2 \rfloor$ and $m_{\text{right},1} = \lfloor (n+m_0)/2 \rfloor$, become the left and right child of the root, respectively. This process is repeated until all the array elements are related to the nodes of the tree. The middle element of each subarray is the left child if its key is less than the parent key or the right child otherwise.

Example 3.7. Figure 3.3 shows a binary search tree for the 16 sorted keys in Figure 3.2. The key $a[7] = 53$ is the root of the tree. The lower $[0..6]$ and the upper $[8..15]$ subranges of the search produce the two children of the root: the left child $a[3] = 33$ and the right child $a[11] = 81$. All other nodes are built similarly.

The tree representation interprets binary search as a pass through the tree from the root to a desired key. If a leaf is reached but the key is not found, then the search is unsuccessful. The number of comparisons to find a key is equal to the number of nodes along the unique path from the root to the key (the depth of the node, plus one).

A static binary search always yields a tree that is well-balanced: for each node in the tree, the left and right subtrees have height that differs by at most 1. Thus all leaves are on at most two levels. This property is used to define AVL trees in Section 3.4.

Implementation of binary search

Algorithm `binarySearch` in Figure 3.4 searches for key k in a sorted array a.

The search starts from the whole array from $l = 0$ to $r = n-1$. If l and r ever cross, $l > r$, then the desired key is absent, indicating an unsuccessful search. Otherwise,

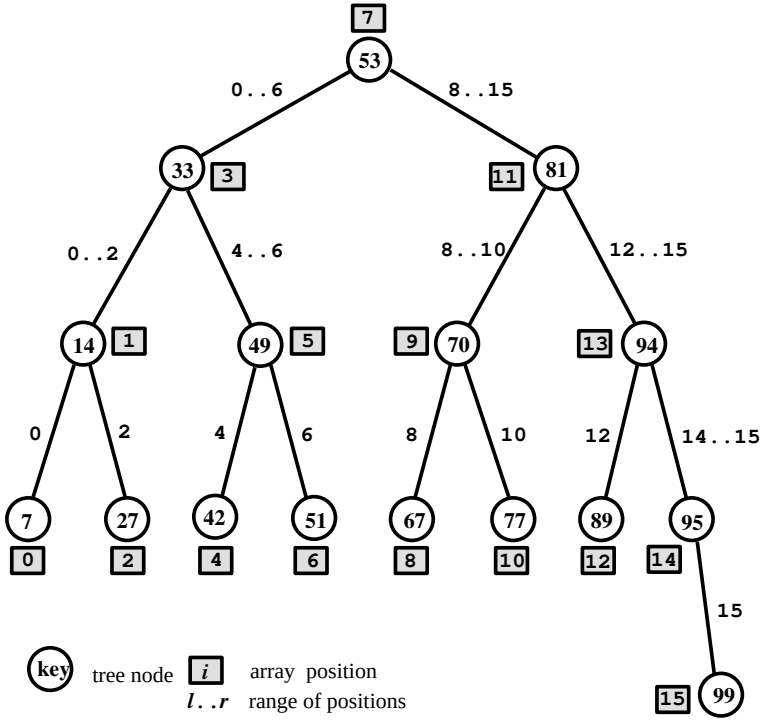


Figure 3.3: Binary tree representation of a sorted array.

the middle position of the odd range or rounded down "almost middle" position of the even one, $m = \lfloor \frac{l+r}{2} \rfloor$, is computed. The search key, k, is compared to the key $a[m]$ in this position:

- If $k = a[m]$, then return the found position m.
- If $k < a[m]$, then the key may be in the range l to $m-1$ so that $r \leftarrow m-1$ at next step.
- If $k > a[m]$, then it may be in the range $m+1$ to r so that $l \leftarrow m+1$ at next step.

Binary search is slightly accelerated if the test for a successful search is removed from the inner loop in Figure 3.4 and the search range is reduced by one in all cases. To determine whether the key k is present or absent, a single test is performed outside the loop (see Figure 3.5). If the search key k is not larger than the key $a[m]$ in the middle position m, then it may be in the range from l to m. The algorithm breaks the while-loop when the range is 1, that is, $l = r$, and then tests whether there is a match.

```
algorithm binarySearch
    Input: array a[0..n−1]; key k
begin
    l ← 0; r ← n−1
    while l ≤ r do
        m ← ⌊(l+r)/2⌋
        if a[m] < k then l ← m+1
        else if a[m] > k then r ← m−1 else return m
        end if
    end while
    return not found
end
```

Figure 3.4: Binary search with three-way comparisons.

```
algorithm binarySearch2
    Input: array a[0..n−1]; key k
begin
    l ← 0; r ← n−1
    while l < r do
        m ← ⌊(l+r)/2⌋
        if a[m] < k then l ← m+1
        else r ← m
    end while
    if a[l] = k then return l
    else return not found
end
```

Figure 3.5: Faster binary search with two-way comparisons.

Exercises

Exercise 3.2.1. Perform a search for the key 41 in the situation of Example 3.4.

Exercise 3.2.2. How many comparisons will binary search make in the worst case to find a key in a sorted array of size $n = 10^6$?

Exercise 3.2.3. Prove that both given array implementations of binary search correctly find an element or report that it is absent.

Exercise 3.2.4. If we have more information about a sorted list, ***interpolation search*** can be (but is not always) much faster than binary search. It is the method used when searching a telephone book: to find a name starting with "C" we open the book not in the middle, but closer to the beginning.

To search for an element between position l and r with $l < r$, we choose $\rho = \frac{k-a[l]}{a[r]-a[l]}$ and the next guess is $m = l + \lceil \rho(r-l) \rceil$.

Give an example of a sorted input array of 8 distinct integers for which interpolation search performs no better than sequential search.

Exercise 3.2.5. Determine how many positions interpolation search will examine to find the key 85 among the ten keys $(10, 20, 35, 45, 55, 60, 75, 80, 85, 100)$.

3.3 Binary search trees

In Section 3.2 we have already used a binary search tree to specify sequences of comparisons and simplify the performance analysis of binary search. Now we use a binary search tree not just as a mathematical object to aid our analysis, but as an explicit data structure that implements the table ADT.

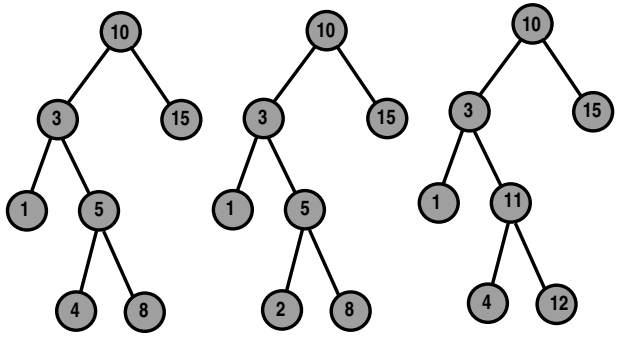


Figure 3.6: Binary trees: only the leftmost tree is a binary search tree.

Example 3.8. In Figure 3.6, two trees are not binary search trees because the key 2 in the middle tree is in the right subtree of the key 3, and the keys 11 and 12 in the rightmost tree are in the left subtree of the key 10.

Binary search trees implement efficiently the basic search, insert, and remove operations of the table ADT. In addition a BST allows for more operations such as sorting records by their keys, finding the minimum key, etc (see Exercises).

Binary search trees are more complex than heaps. Only a root or leaves are added to or removed from a heap, whereas any node of a binary search tree may be removed.

The search operation resembles usual binary search in that it starts at the root and moves either left or right along the tree, depending on the result of comparing the search key to the key in the node.

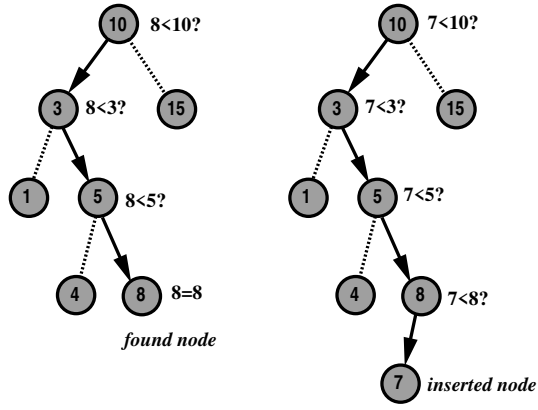


Figure 3.7: Search and insertion in the binary search tree.

Example 3.9. To find key 8 in the leftmost tree in Figure 3.6, we start at the root 10 and go left because $8 < 10$. This move takes us to 3, so we turn right (because $8 > 3$) to 5. Then we go right again ($8 > 5$) and encounter 8.

Example 3.10. To find key 7, we repeat the same path. But, when we are at node 8, we cannot turn left because there is no such link. Hence, 7 is not found.

Figure 3.7 illustrates both Examples 3.9 and 3.10. It shows that the node with key 7 can be inserted just at the point at which the unsuccessful search terminates.

The removal operation is the most complex because removing a node may disconnect the tree. Reattachment must retain the ordering condition but should not needlessly increase the tree height. The standard method of removing a node is as follows.

- A leaf node is simply deleted.
- An internal node with only one child is deleted after its child is linked to its parent node.
- If the node has two children, then it should be swapped with the node having the smallest key in its right subtree. The latter node is easily found (see Exercise 3.3.1). After swapping, the node can be removed as in the previous cases, since it is now in a position where it has at most one child.

This approach appears asymmetric but various modifications do not really improve it. The operation is illustrated in Figure 3.8.

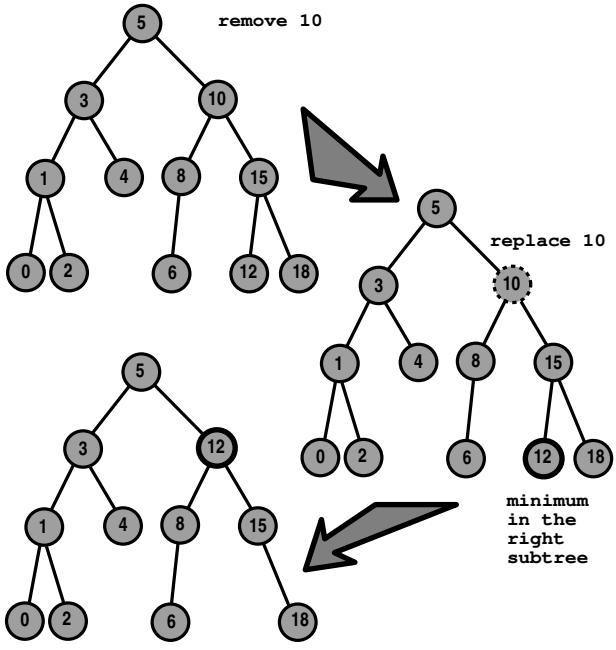


Figure 3.8: Removal of the node with key 10 from the binary search tree.

Analysis of BST-based tables

Lemma 3.11. The search, retrieval, update, insert and remove operations in a BST all take time in $O(h)$ in the worst case, where h is the height of the tree.

Proof. The running time of these operations is proportional to the number of nodes visited. For the find and insert operations, this equals 1 plus the depth of the node, but for the remove operation it equals the depth of the node plus at most the height of the node. In each case this is $O(h)$. □

For a well-balanced tree, all operations take logarithmic time. The problem is that insertions and deletions may destroy the balance, and in practice BSTs may be heavily unbalanced as in Figure 3.11. So in the worst case the search time is linear, $\Theta(n)$, and we have an inferior form of sequential search (because of the extra overhead in creating the tree arcs).

Figure 3.9 presents all variants of inserting the four keys 1, 2, 3, and 4 into an empty binary search tree. Because only relative ordering is important, this corresponds to any four keys $(a[i] : i = 1,\ldots,4)$ such that $a[1] < a[2] < a[3] < a[4]$.

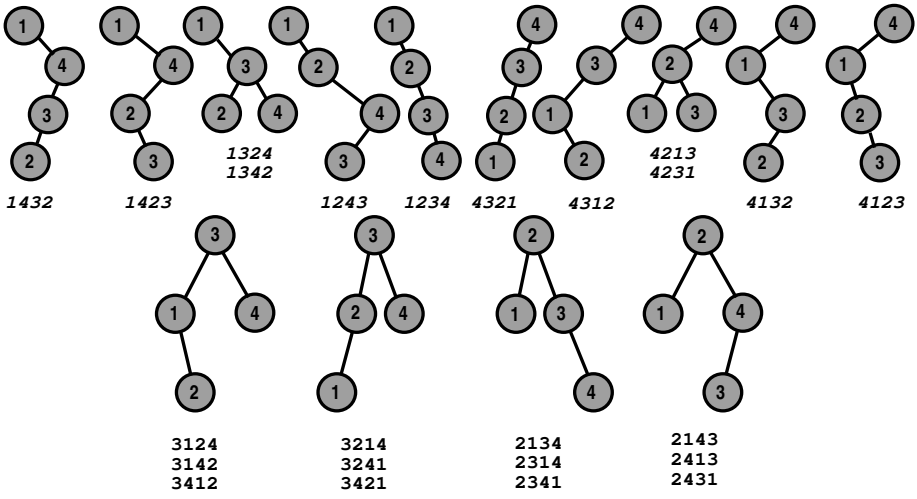


Figure 3.9: Binary search trees obtained by permutations of $1,2,3,4$.

There are in total $4! = 24$ possible insertion orders given in Figure 3.9. Some trees result from several different insertion sequences, and more balanced trees appear more frequently than unbalanced ones.

Definition 3.12. The ***total internal path length***, $S_\tau(n)$, of a binary tree τ is the sum of the depths of all its nodes.

The average node depth, $\frac{1}{n}S_\tau(n)$, gives the average time complexity of a successful search in a particular tree τ. The average-case time complexity of searching is obtained by averaging the total internal path lengths for all the trees of size n, that is, for all possible $n!$ insertion orders, assuming that these latter occur with equal probability, $\frac{1}{n!}$.

Lemma 3.13. Suppose that a BST is created by n random insertions starting from an empty tree. Then the expected time for successful and unsuccessful search is $\Theta(\log n)$. The same is true for update, retrieval, insertion and deletion.

Proof. Let $S(n)$ denote the average of S_τ over all insertion sequences, each sequence considered as equally likely. We need to prove that $S(n)$ is $\Theta(n \log n)$.

It is obvious that $S(1) = 0$. Furthermore, any n-node tree, $n > 1$, contains a left subtree with i nodes, a root at height 0, and a right subtree with $(n-i-1)$ nodes where $0 \le i \le n-1$, each value of i being by assumption equiprobable.

For a fixed i, $S(i)$ is the average total internal path length in the left subtree with respect to its own root and $S(n-i-1)$ is the analogous total path length in the right subtree. The root of the tree adds 1 to the path length of each other node. Because there

are $n-1$ such nodes, the following recurrence holds: $S(n) = (n-1) + S(i) + S(n-i-1)$ for $0 \le i \le n-1$. After summing these recurrences for all $i = 0, \ldots, n-1$ and averaging, just the same recurrence as for the average-case quicksort analysis (see Lemma 2.15) is obtained: $S(n) = (n-1) + \frac{2}{n}\sum_{i=0}^{n-1} S(i)$. Therefore, the average total internal path length is $\Theta(n \log n)$. The expected depth of a node is therefore in $\Theta(\log n)$. This means that search, update, retrieval and insertion take time in $\Theta(\log n)$. The same result is true for deletion (this requires the result that the expected height of a random BST is $\Theta(\log n)$, which is harder to prove—see Notes). □

Thus in practice, for random input, all BST operations take time about $\Theta(\log n)$. However the worst-case linear time complexity, $\Theta(n)$, is totally unsuitable in many applications, and deletions can also destroy balance.

We tried to eliminate the worst-case degradation of quicksort by choosing a pivot that performs well on random input data and relying on the very low probability of the worst case. Fortunately, binary search trees allow for a special general technique, called ***balancing***, that guarantees that the worst cases simply cannot occur. Balancing restructures each tree after inserting new nodes in such a way as to prevent the tree height from growing too much. We discuss this more in Section 3.4.

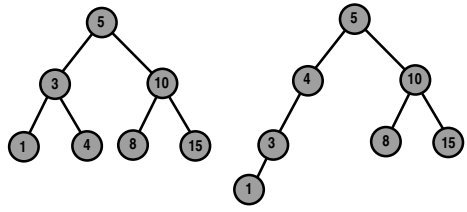


Figure 3.10: Binary search trees of height about $\log n$.

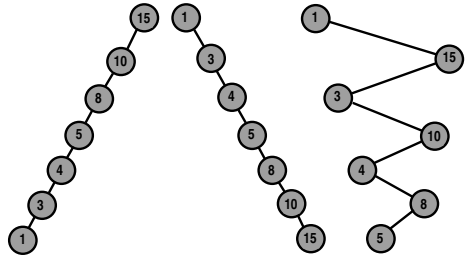


Figure 3.11: Binary search trees of height about n.

Implementation of BST

A concrete implementation of a BST requires explicit links between nodes, each of which contains a data record, and is more complicated than the other implemen-

tations so far. A programming language that supports easy manipulation of objects makes the coding easier. We do not present any (pseudo)code in this book.

Exercises

Exercise 3.3.1. Show how to find the maximum (minimum) key in a binary search tree. What is the running time of your algorithm? How could you find the median, or an arbitrary order statistic?

Exercise 3.3.2. Suppose that we insert the elements $1,2,3,4,5$ into an initially empty BST. If we do this in all 120 possible orders, which trees occur most often? Which tree shapes (in other words, ignore the keys) occur most often?

Exercise 3.3.3. Suppose that we insert the elements $1,2,3,4$ in some order into an initially empty BST. Which insertion orders yield a tree of maximum height? Which yield a tree of minimum height?

Exercise 3.3.4. Show how to output all records in ascending order of their keys, given a BST. What is the running time of your algorithm?

3.4 Self-balancing binary and multiway search trees

Because balanced binary search trees are more complex than the standard ones, the insertion and removal operations usually take longer time on the average. But because of the resulting balanced structure, the worst-case complexity is $O(\log n)$. In other words, the total internal path lengths of the trees are fairly close to the optimal value.

AVL, red-black, and AA-trees

Definition 3.14. An ***AVL tree*** is a binary search tree with the additional balance property that, for any node in the tree, the heights of the left and right subtrees can differ by at most 1.

This balance condition ensures height $\Theta(\log n)$ for an AVL tree despite being less restrictive than requiring the tree to be complete. Complete binary trees have too rigid a balance condition which is difficult to maintain when new nodes are inserted.

Lemma 3.15. The height of an AVL tree with n nodes is $\Theta(\log n)$.

Proof. Due to the possibly different heights of its subtrees, an AVL tree of height h may contain fewer than $2^{h+1}-1$ nodes. We will show that it contains at least c^h nodes for some constant $c>1$, so that the maximum height of a tree with n items is $\log_c n$.

Let S_h be the size of the smallest AVL tree of height h. It is obvious that $S_0=1$ (the root only) and $S_1=2$ (the root and one child). The smallest AVL tree of height h has subtrees of height $h-1$ and $h-2$, because at least one subtree is of height $h-1$ and the second height can differ at most by 1 due to the AVL balance condition. These

subtrees must also be the smallest AVL trees for their height, so that $S_h = S_{h-1} + S_{h-2} + 1$.

By mathematical induction, we show easily that $S_h = F_{h+3} - 1$ where F_i is the i-th Fibonacci number (see Example 1.28) because $S_0 = F_3 - 1$, $S_1 = F_4 - 1$, and $S_{i+1} = (F_{i+3} - 1) + (F_{i+2} - 1) + 1 \equiv F_{i+4} - 1$. Therefore, for each AVL tree with n nodes $n \geq S_h \approx \frac{\phi^{h+3}}{\sqrt{5}} - 1$ where $\phi \approx 1.618$. Thus, the height of an AVL tree with n nodes satisfies the following condition: $h \leq 1.44 \cdot \lg(n+1) - 1.33$, and the worst-case height is at most 44% more than the minimum height for binary trees. □

Note that AVL trees need in the worst case only about 44% more comparisons than complete binary trees. They behave even better in the average case. Although theoretical estimates are unknown, the average-case height of a randomly constructed AVL tree is close to $\lg n$.

All basic operations in an AVL tree have logarithmic worst-case running time. The difficulty is, of course, that simple BST insertions and deletions can destroy the AVL balance condition. We need an efficient way to restore the balance condition when necessary. All self-balancing binary search trees use the idea of ***rotation*** to do this.

Example 3.16. Figure 3.12 shows a left rotation at the node labelled p and a right rotation at the node labelled q (the labels are not related to the keys stored in the BST, which are not shown here). These rotations are mutually inverse operations. Each rotation involves only local changes to the tree, and only a constant number of pointers must be updated, independent of the tree size. If, for example, there is a subtree of large height below a, then the right rotation will decrease the overall tree height.

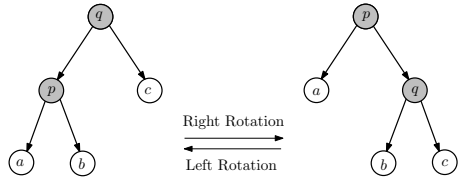


Figure 3.12: Left and right rotations of a BST.

The precise details of exactly when a rotation is required, and which kind, differ depending on the type of balanced BST. In each case the programming of the insertion and removal operations is quite complex, as is the analysis. We will not go into more details here—the reader should consult the recommended references.

Balancing of AVL trees requires extra memory and heavy computations. This is why even more relaxed efficient balanced search trees such as red-black trees are more often used in practice.

Definition 3.17. A ***red-black tree*** is a binary search tree such that every node is coloured either red or black, and every non-leaf node has two children. In addition, it satisfies the following properties:

- the root is black;
- all children of a red node must be black;
- every path from the root node to a leaf must contain the same number of black nodes.

Theorem 3.18. If every path from the root to a leaf contains b black nodes, then the tree contains at least $2^b - 1$ black nodes.

Proof. The statement holds for $b = 1$ (in this case the tree contains either the black root only or the black root and one or two red children). In line with the induction hypothesis, let the statement hold for all red-black trees with b black nodes in every path. If a tree contains $b+1$ black nodes in every path and has two black children of the root, then the tree contains two subtrees with b black nodes just under the root and has in total at least $1+2\cdot(2^b-1) = 2^{b+1}-1$ black nodes. If the root has a red child, the latter has only black children, so that the total number of the black nodes can become even larger. □

Each path cannot contain two consecutive red nodes and increase more than twice after all the red nodes are inserted. Therefore, the height of a red-black tree is at most $2\lceil \lg n \rceil$, and the search in it is logarithmic, $O(\log n)$.

Red-black trees allow for a very fast search. This data structure has still no precise analysis of its average-case performance. Its properties are found either experimentally or by analysing red-black trees containing random n keys. There are about $\lg n$ comparisons per search on the average and fewer than $2\lg n + 2$ comparisons in the worst case. Restoring the tree after insertion or deletion of single node requires $O(1)$ rotations and $O(\log n)$ colour changes in the worst case.

Another variety of balanced tree, the ***AA-tree***, becomes more efficient than a red-black tree when node deletions are frequent. An AA-tree has only one extra condition with respect to a red-black tree, namely that the left child may not be red. This property simplifies the removal operation considerably.

Balanced B-trees: efficiency of external search

The B-tree is a popular structure for ordered databases in external memory such as magnetic or optical disks. The previous "Big-Oh" analysis is invalid here because it assumes all elementary operations have equal time complexity. This does not hold for disk input / output where one disk access corresponds to hundreds of thousands of computer instructions and the number of accesses dominates running time. For a large database of many millions of records, even logarithmic worst-case performance of red-black or AA-trees is unacceptable. Each search should involve a very small number of disk accesses, say, 3–4, even at the expense of reasonably complex computations (which will still take only a small fraction of a disk access time).

Binary tree search cannot solve the problem because even an optimal tree has height $\lg n$. To decrease the height, each node must have more branches. The height of an optimal m-ary search tree (m-way branching) is roughly $\log_m n$ (see Table 3.2), or $\lg m$ times smaller than with an optimal binary tree (for example, 6.6 times for $m = 100$).

Table 3.2: Height of the optimal m-ary search tree with n nodes.

n	10^5	10^6	10^7	10^8	10^9
$\lceil \log_2 n \rceil$	17	20	24	27	30
$\lceil \log_{10} n \rceil$	5	6	7	8	9
$\lceil \log_{100} n \rceil$	3	3	4	4	5
$\lceil \log_{1000} n \rceil$	2	2	3	3	3

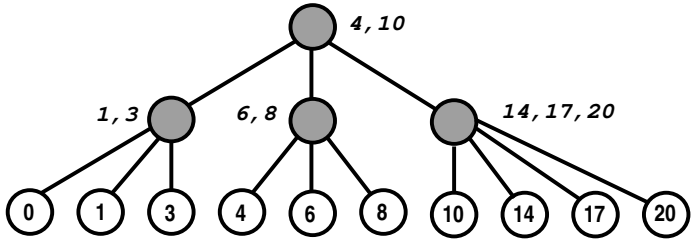


Figure 3.13: Multiway search tree of order $m = 4$.

Figure 3.13 shows that the search and the traversal of a multiway search tree generalize in a straightforward way the binary search tree operations. If in the latter case the search key is compared to a single key in a node in order to choose one of two branches or stop in the node, in an m-ary tree, the search key is compared to at most $m-1$ keys in a node to choose one of m branches. The major difference is that multiple data records are now associated only with leaves although some multiway

trees do not strictly follow this condition. Thus the worst-case and the average-case search involve the tree height and the average leaf height, respectively.

Example 3.19. Search for a desired key k in Figure 3.13 is guided by thresholds, for example at the root it goes left if $k < 4$, down if $4 \le k < 10$, and right if $k \ge 10$. The analogous comparisons are repeated at every node until the record with the key k is found at a leaf or its absence is detected. Let $k = 17$. First, the search goes right from the root (as $17 > 10$), then goes to the third child of the right internal node (as $17 \le 17 < 20$), and finally finds the desired record in that leaf.

Definition 3.20. A ***B-tree*** of order m is an m-ary tree such that:

- the root is either a leaf or it has between 2 and m children inclusive;
- each nonleaf node (except possibly the root) has between $\lceil m/2 \rceil$ and m children inclusive;
- each nonleaf node with μ children has $\mu - 1$ keys, $(\theta[i] : i = 1, \ldots, \mu - 1)$, to guide the search where $\theta[i]$ is the smallest key in subtree $i + 1$;
- all leaves are at the same depth;
- data items are stored in leaves, each storing between $\lceil l/2 \rceil$ and l items, for some l.

Other definitions of B-trees (mostly with minor changes) also exist, but the above one is most popular. The first three conditions specify the memory space each node needs (first of all, for m links and $m - 1$ keys) and ensure that more than half of it, except possibly in the root, will be used. The last two conditions form a well-balanced tree.

Note. B-trees are usually named by their ***branching limits***, that is, $\lceil m/2 \rceil$–m, so that 2–3 and 6–11 trees are B-trees with $m = 3$ and $m = 11$, respectively.

Example 3.21. In a $2-4$ B-tree in Figure 3.14 all nonleaf nodes have between $\lceil 4/2 \rceil = 2$ and 4 children and thus from 1 to 3 keys. The number l of data records associated with a leaf depends on the capacity of external memory and the record size. In Figure 3.14, $l = 7$ and each leaf stores between $\lceil 7/2 \rceil = 4$ and 7 data items.

Because the nodes are at least half full, a B-tree with $m \ge 8$ cannot be a simple binary or ternary tree. Simple ternary 2–3 B-trees with only two or three children per node are sometimes in use for storing ordered symbol tables in internal computer RAM. But branching limits for B-trees on external disks are considerably greater to make one node fit in a unit data block on the disk. Then the number of nodes examined (and hence the number of disk accesses) decreases $\lg m$ times or more compared with a binary tree search.

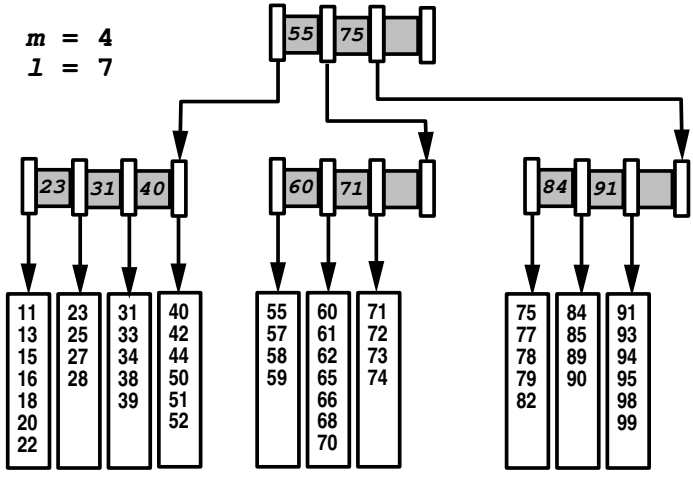


Figure 3.14: 2–4 B-tree with the leaf storage size 7 (2..4 children per node and 4..7 data items per leaf).

In each particular case, the tree order m and the leaf capacity l depend on the disk block size and the size of records to store. Let one disk block hold d bytes, each key be of κ bytes, each branch address be of b bytes, and the database contain s records, each of size r bytes. In a B-tree of order m, each nonleaf node stores at most $m-1$ keys and m branch addresses, that is, in total, $\kappa(m-1)+bm=(\kappa+b)m-\kappa$ bytes. The largest order m such that one node fits in one disk block, $(\kappa+b)m-\kappa\le d$, is $m=\left\lfloor\frac{d+\kappa}{b+\kappa}\right\rfloor$. Each internal node, except the root, has at least $\left\lceil\frac{m}{2}\right\rceil$ branches. At most $l=\frac{d}{r}$ records fit in one block, and each leaf addresses from $\frac{l}{2}$ to l records. Assuming each leaf is full, the total number of the leaves is $n=\left\lceil\frac{s}{l}\right\rceil$, so that in the worst case the leaves are at level $\lceil\log_{m/2}n\rceil+1$.

Example 3.22. Suppose the disk block is $d=2^{15}\equiv 32768$ bytes, the key size is $\kappa=2^6\equiv 64$ bytes, the branch address has $b=8$ bytes, and the database contains $s=2^{30}\cong 1.07\cdot 10^9$ records of size $r=2^{10}\equiv 1024$ bytes each. Then the B-tree order is $m=\left\lfloor\frac{32768+64}{8+64}\right\rfloor=\left\lfloor\frac{32832}{72}\right\rfloor=456$ so that each internal node, except the root, has at least 228 branches. One block contains at most $l=\frac{32768}{1024}=32$ records, and the number of leaves is at least $n=\lceil 2^{30}/32\rceil=2^{25}$. The worst-case level of the leaves in this B-tree is $\lceil\log_{228}2^{25}\rceil+1=\lceil 3.19\rceil+1=5$.

Generally, a search in or an insertion into a B-tree of order m with n data records requires fewer than $\lceil\log_{m/2}n\rceil$ disk accesses, and this number is practically constant if m is sufficiently big as shown in Table 3.2. The running time becomes even smaller

if the root and the upper two tree levels are stored in internal RAM and the slow disk accesses occur only for level 3 or higher. The three-level B-tree with $m = 456$ can handle up to 456^3, or $94,818,816$ entries. If in Example 3.22 each key uses only $\kappa = 24$ bytes, then $m = 1024$, and the three-level tree can handle over 10^9 entries.

Data insertion into a B-tree is simple if the corresponding leaf is not already full. A full leaf has to be split into two leaves, both having the minimum number of data items, and the parent node should be updated. If necessary, the splitting propagates up until it finds a parent that need not be split or reaches the root. Only in the extremely rare case that the root has to be split, the tree height increases and a new root with two children (halves of the previous root) is created. Data deletion is also simple until the leaf is empty and its neighbours must be combined to form a full leaf. Although the programming is not simple, all changes are well defined. Algorithm analysis, beyond the scope of this book, shows that both data insertion, deletion, and retrieval have only about $\log_{\frac{m}{2}} n$ disk accesses in the worst case.

Exercises

Exercise 3.4.1. Draw two different red-black trees containing at most two black nodes along every path from the root to a leaf.

Exercise 3.4.2. Draw two different AVL trees of size $n = 7$ and compare them to the complete binary tree of the same size. Is the latter also an AVL tree?

Exercise 3.4.3. Draw an AA-tree containing at most 2 black nodes along every path from a node to a leaf and differing from the complete binary tree of order $n = 7$.

Exercise 3.4.4. Draw a binary search tree of minimum size such that a left rotation reduces the height of the tree.

3.5 Hash tables

There are numerous ways to implement the table ADT. We have already seen that various search trees will do everything required, provided the keys are from some totally ordered set. If, say, the keys are dictionary words with the usual ordering, then it is not necessary to use any integer encoding—keys can be compared directly.

Suppose now that we have a very simple situation where the number of possible keys is small. Then we can just store the values in an array. One array entry can be reserved in advance for each possible key, and the key-to-value mapping ends up as a conventional array address. Searching then has worst-case constant time, as does insertion and deletion. This implementation of a table works well provided the number of possible keys is sufficiently small.

However, that nice state of affairs does not occur often (we could use it for the airport codes in Example 3.2). Usually there exists a very large number of possible keys although only a tiny fraction of them are actually put into use. For example, suppose that we have a database where each customer is identified by an 8-digit

telephone number. If we have 10 000 customers, only 0.01% of the array addresses are filled.

There is another technique to store and search for values in symbol tables, called ***hashing***, that uses less space and retains many (not all) of the benefits of direct array addressing.

Definition 3.23. Hashing computes an integer ***hash code*** for each object using a ***hash function*** that maps objects (for example, keys) to indices of a given linear array (the ***hash table***).

Hash functions are designed in such a manner that hash codes are computed quickly. The computation of an array index with a hash function, or "hashing a key to an index", depends only on the key to hash and is independent of other keys in the table. If $h(k)$ is the value of a hash function for k, then the key k should be placed at location $h(k)$.

The hash function is chosen so as to always return a valid index for the array. A ***perfect hash function*** maps each key to a different index. Unfortunately, it is difficult to find such a function in most cases.

Example 3.24. Let us map two-digit integer keys onto the ten array indices [0, 1, ..., 9] by a simple hash function $h(k) = \lfloor k/10 \rfloor$. Then the keys 21 and 27 both have the hash code 2 pointing to the same position in the array. Such a situation in which two different keys, $k_1 \neq k_2$, hash to the same index (table address), $h(k_1) = h(k_2)$, is called a ***collision***. Because both table entries (k_1, v_1) and (k_2, v_2) cannot be at the same address, we need a definite ***collision resolution policy***.

Different keys hashed to the same hash address are called ***synonyms***, and data items with synonymic keys are frequently also referred to as synonyms.

Collision resolution: OALP, OADH, and SC hash tables

There are many collision resolution policies. The main issues are:

- Do we use extra storage, or not?
- Which element moves when a collision occurs: the incumbent element or the newcomer (or both)?
- How do we decide where to move the evicted element to?

CHAINING In ***separate chaining*** synonyms with the same hash address are stored in a linked list connected to that address. We still hash the key of each item to obtain an array index. But if there is a collision, the new item is simply placed in this hash address, along with all other synonyms. Each array element is a head reference for the associated linked list, and each node of this list stores not only the key and data values for a particular table entry but also a link to the next node. The head node of the list referenced by the array element always contains the last inserted item.

OPEN ADDRESSING Open addressing uses no extra space for collision resolution. Instead, we move one of the colliding elements to another slot in the array. We may use LIFO (last-in, first out — the new element must move), FIFO (first in, first out — the old element must move), or more complicated methods such as Robin Hood or cuckoo hashing (see Notes). For our purposes here, we use LIFO.

Each collision resolution policy ***probes*** another array slot, and if empty inserts the currently homeless element. If the probed slot is not empty, we probe again to find a slot in which to insert the currently homeless element, and so on until we finish insertion. The ***probe sequence*** used can be a simple fixed sequence, or given by a more complicated rule (but is always deterministic). They all have the property that they "wrap around" the array when they reach the end. The two most common probing methods are:

- (Linear probing) always probe the element to the left;
- (Double hashing) probe to the left by an amount determined by the value of a secondary hash function.

Note. The choice of probing to the left versus probing to the right is clearly a matter of convention; the reader should note that other books may use rightward probing in their definitions.

Table 3.3: Open addressing with linear probing (OALP).

Data [key,value]	**Hash**: key/10	**Table address**	**Comments**
[20,A]	2	2	
[15,B]	1	1	
[45,C]	4	4	
[87,D]	8	8	
[39,E]	3	3	
[31,F]	3	0	try 3, 2, 1, 0
[24,G]	2	9	try 2, 1, 0, 9

Example 3.25. Table 3.3 shows how OALP fills the hash table of size 10 using the two-digit keys and the hash function of Example 3.24. The first five insertions have found empty addresses. However, the key–value pair [31, F] has a collision because the address $h(31) = 3$ is already occupied by the pair [39, E] with the same hash address, $h(39) = 3$. Thus, the next lower table address, location 2, is probed to see if it is empty, and in the same way the next locations 1 and 0 are checked. The address 0 is empty so that the pair [31, F] can be placed there.

A similar collision occurs when we try to insert the next pair, [24, G], because the hash address $h(24) = 2$ for the key 24 is already occupied by the previous pair [20, A].

Consequently, we probe successive lower locations 1 and 0, and since they both are already occupied, we wrap around and continue the search at the highest location 9. Because it is empty, the pair [24, G] is inserted in this location yielding the final configuration given in Table 3.3.

OALP is simple to implement but the hash table may degenerate due to ***clustering***. A ***cluster*** is a sequence of adjacent occupied table entries. OALP tends to form clusters around the locations where one or more collisions have occurred. Each collision is resolved using the next empty location available for sequential probing. Therefore, other collisions become more probable in that neighbourhood, and the larger the clusters, the faster they grow. As a result, a search for an empty address to place a collided key may turn into a very long sequential search.

Another probing scheme, ***double hashing***, reduces the likelihood of clustering. In double hashing, when a collision occurs, the key is moved by an amount determined by a secondary hash function Δ. Let h denote the primary hash function. Then for each key k we have the starting probe address $i_0 = h(k)$ and the probe decrement $\Delta(k)$. Each next successive probe position is $i_t = (i_{t-1} - \Delta(k)) \bmod m;\ t = 1, 2, \ldots$ where m is the table size.

Example 3.26. Table 3.4 shows how OADH fills the same hash table as in Example 3.25 if the hash function is given by $\Delta(k) = (h(k) + k) \bmod 10$.

Table 3.4: Open addressing with double hashing (OADH).

Data [key,value]	**Hash**: key/10	**Table address**	**Comments**
[20,A]	2	2	
[15,B]	1	1	
[45,C]	4	4	
[87,D]	8	8	
[39,E]	3	3	
[31,F]	3	9	using $\Delta(31) = 4$
[24,G]	2	6	using $\Delta(24) = 6$

Now when we try to place the key–value pair [31, F] into position $h(31) = 3$, the collision is resolved by probing the table locations with decrement $\Delta(31) = 4$. The first position, $(3-4) \mod 10 = 9$ is empty so that the pair [31, F] can be placed there. For the collision of the pair, [24, G], at location 2 the decrement $\Delta(24) = 6$ immediately leads to the empty location 6. The final table in Figure 3.4 contains three small clusters instead of one large cluster in Figure 3.3.

Generally, OADH results in more uniform hashing that forms more clusters than OALP but of smaller sizes. Linear probing extends each cluster from its end with the

lower table address, and nearby clusters join into larger clusters growing even faster. Double hashing does not extend clusters only at one end and does not tend to join nearby clusters.

Analysis of hash tables

The time complexity of searching in and inserting items in a hash table of size m with n already occupied entries is determined by the ***load factor***, $\lambda := \frac{n}{m}$. In open addressing, $0 \leq \lambda < 1$: λ equals the fraction of occupied slots in the array, and cannot be exactly equal to 1 because a hash table should have at least one empty entry in order to efficiently terminate the search for a key or the insertion of a new key.

Open addressing and separate chaining require n probes in the worst case, since all elements of the hash table may be synonyms. However the basic intuition is that provided the table is not too full, collisions should be rare enough that searching for an element requires only a constant number of probes on average.

Thus we want a result such as: "*Provided the load factor is kept bounded (and away from 1 in the case of open addressing), all operations in a hash table take* $\Theta(1)$ *time in the average case.*"

In order to have confidence in this result, we need to describe our mathematical model of hashing. Since a good hash function should scatter keys randomly, and we have no knowledge of the input data, it is natural to use the "random balls in bins" model. We assume that we have thrown n balls one at a time into m bins, each ball independently and uniformly at random.

For our analysis, it will be useful to use the function Q defined below.

Definition 3.27. For each integer m, n with $1 \leq n \leq m$, we define

$$Q(m,n) = \frac{m!}{(m-n)!m^n} = \frac{m}{m}\frac{m-1}{m}\cdots\frac{m-n+1}{m}.$$

Note that $Q(m,1) = 1$.

BASIC ANALYSIS OF COLLISIONS It is obvious that if we have more items than the size of the hash table, at least one collision must occur. But the distinctive feature of collisions is that they are relatively frequent even in almost empty hash tables.

The ***birthday paradox*** refers to the following surprising fact: *if there are 23 or more people in a room, the chance is greater than 50% that two or more of them have the same birthday.* Note: this is not a paradox in the sense of a logical contradiction, but just a "counter-intuitive" fact that violates "common sense".

More precisely, if each of 365 bins is selected with the same chance $\frac{1}{365}$, then after 23 entries have been inserted, the probability that at least one collision has occurred (at least one bin has at least two balls) is more than 50%. Although the table is only $23/365$ ($\cong 6.3\%$) full, more than half of our attempts to insert one more entry will result in a collision!

Let us see how the birthday paradox occurs. Let m and n denote the size of a table and the number of items to insert, respectively. Let $\Pr_m(n)$ be the probability of at least one collision when n balls are randomly placed into m bins.

Lemma 3.28. The probability of no collisions when n balls are thrown independently into m boxes uniformly at random is $Q(m,n)$. Thus $\Pr_m(n) = 1 - Q(m,n)$ and the expected number of balls thrown until the first collision is $\sum_{n \le m} Q(m,n)$.

Proof. Let $\pi_m(n)$ be the probability of no collisions. The "no collision" event after inserting ν items; $\nu = 2, \ldots, n$, is a joint event of "no collision" after inserting the preceding $\nu - 1$ items and "no collision" after inserting one more item, given $\nu - 1$ positions are already occupied. Thus $\Pr_m(\nu) = \Pr_m(\nu - 1) P_m(\text{no collision} \mid \nu - 1)$ where $P_m(\text{no collision} \mid \nu)$ denotes the conditional probability of no collision for a single item inserted into the table with $m - \nu$ unoccupied positions. This latter probability is simply $\frac{m-\nu}{m}$.

This then yields immediately

$$\pi_m(n) = \frac{m}{m} \frac{m-1}{m} \cdots \frac{m-n+1}{m} = \frac{m(m-1)\cdots(m-n+1)}{m^n} = \frac{m!}{m^n (m-n)!}$$

Therefore, $\Pr_m(n) = 1 - \frac{m!}{m^n (m-n)!} = 1 - Q(m,n)$ which gives the first result.

The number of balls is at least $n+1$ with probability $Q(m,n)$. Since the expected value of a random variable T taking on nonegative integer values can always be computed by $E[T] = \sum_{n \ge 1} i \Pr(T = i) = \sum_{j \ge 0} \Pr(T > j)$, and these latter probabilities are zero when $j > m$, the second result follows. □

Table 3.5 presents (to 4 decimal places) some values of $\Pr_m(n)$ for $m = 365$ and $n = 5 \ldots 100$. As soon as $m = 47$ (the table with 365 positions is only 12.9% full), the probability of collision is greater than 0.95. Thus collisions are frequent even in sparsely occupied tables.

Table 3.5: Birthday paradox: $\Pr_{365}(n)$.

n	5	10	15	20	22
$\Pr_{365}(n)$	0.0271	0.1169	0.2529	0.4114	0.4757
n	23	25	30	35	40
$\Pr_{365}(n)$	0.5073	0.5687	0.7063	0.8144	0.8912
n	45	50	55	60	65
$\Pr_{365}(n)$	0.9410	0.9704	0.9863	0.9941	0.9977
n	70	75	80	90	100
$\Pr_{365}(n)$	0.9992	0.9997	0.9999	1.0000	1.0000

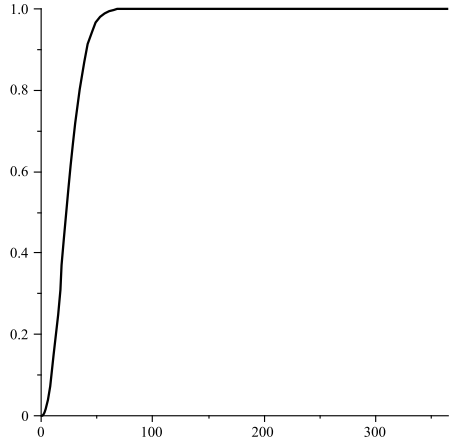


Figure 3.15: Birthday paradox: $\Pr_{365}(n)$.

Figure 3.15 shows the graph of $\Pr_{365}(n)$ as a function of n. The median of this distribution occurs around $n = 23$, as we have said above, and so 23 or more balls suffice for the probability of a collision to exceed $1/2$. Also, the expected number of balls before the first collision is easily computed to be 25.

When the load factor is much less than 1, the average number of balls per bin is small. If the load factor exceeds 1 then the average number is large. In each case analysis is not too difficult. For hashing to be practical, we need to be able to fill a hash table as much as possible before we spend valuable time ***rehashing*** — that is, allocating more space for the table and reassigning hash codes via a new hash function. Thus we need to analyse what happens when the load factor is comparable to 1, that is, when the number of balls is comparable to the number of bins. This also turns out to be the most interesting case mathematically.

THEORETICAL ANALYSIS OF HASHING In addition to the basic operations for arrays, we also consider the computation of the hash address of an item to be an elementary operation.

Chaining is relatively simple to analyse.

Lemma 3.29. The expected running time for unsuccessful search in a hash table with load factor λ using separate chaining is given by

$$T_{\text{us}}(\lambda) = 1 + \lambda.$$

The expected running time for successful search is $O(1 + \lambda/2)$.

Update, retrieval, insertion and deletion all take time that is $O(1 + \lambda)$.

Proof. In an unsuccessful search for a key k, the computation of the hash address, $h(k)$, takes one elementary operation. The average running time to unsuccessfully search for the key at that address is equal to the average length of the associated chain, $\lambda = \frac{n}{m}$. Thus in total $T_{\text{us}}(\lambda) = 1 + \lambda = 1 + n/m$.

The result for successful search and other operations now follows from Lemma 3.3, since update, retrieval and deletion from a linked list take constant time. □

To analyse open addressing, we must make some extra assumptions. We use the ***uniform hashing hypothesis***: each configuration of n keys in a hash table of size m is equally likely to occur. This is what we would expect of a truly "random" hash function, and it seems experimentally to be a good model for double hashing. Note that this is stronger than just requiring that each key is independently and uniformly likely to hash initially to each slot before collision resolution ("random balls in bins"). It also implies that all probe sequences are equally likely.

Lemma 3.30. Assuming the uniform hashing hypothesis holds, the expected number of probes for search in a hash table satisfy

$$T_{\text{us}}(\lambda) \leq \frac{1}{1-\lambda}$$

and

$$T_{\text{ss}}(\lambda) \leq \frac{1}{\lambda} \ln \frac{1}{1-\lambda}.$$

Proof. The average number of probes for an unsuccessful search is $T_{\text{us}}(\lambda) = \sum_{i=1}^{n} i p_{m,n}(i)$ where $p_{m,n}(i)$ denotes the probability of exactly i probes during the search. Obviously, $p_{m,n}(i) = \Pr(m,n,i) - \Pr(m,n,i+1)$ where $\Pr(m,n,i)$ is the probability of i or more probes in the search. By a similar argument to that used in the birthday problem analysis we have for $i \geq 2$

$$\Pr(m,n,i) = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+2}{m-i+2}$$

while $\Pr(m,n,1) = 1$. Note that clearly $\Pr(m,n,i) \leq (n/m)^{i-1} = \lambda^{i-1}$. Thus

$$\begin{aligned} T_{\text{us}}(\lambda) &= \sum_{i=1}^{n} i\,(\Pr(n,m,i) - \Pr(n,m,i+1)) \\ &\leq \sum_{i=1}^{\infty} i\,(\Pr(n,m,i) - \Pr(n,m,i+1)) = \sum_{i=1}^{\infty} \Pr(n,m,i) \leq \sum_{i=1}^{\infty} \lambda^{i-1} = \frac{1}{1-\lambda} = \frac{m}{m-n}. \end{aligned}$$

The result for successful search now follows by averaging. We have

$$\begin{aligned} T_{\mathrm{ss}}(\lambda) &\le \frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} \\ &= \frac{1}{\lambda}\sum_{j=m-n+1}^{m}\frac{1}{j} \\ &\le \frac{1}{\lambda}\int_{n-m}^{n} dx/x = \frac{1}{\lambda}\ln\left(\frac{m}{m-n}\right) = \frac{1}{\lambda}\ln\left(\frac{1}{1-\lambda}\right). \end{aligned}$$

□

Note. It can be shown that the exact values are

$$\begin{aligned} T_{\mathrm{us}}(\lambda) &= \frac{m+1}{m-n+1} \approx \frac{1}{1-\lambda} \\ T_{\mathrm{ss}}(\lambda) &= \frac{m+1}{n}\left(H_{m+1} - H_{m-n+1}\right) \approx \frac{1}{\lambda}\ln\left(\frac{1}{1-\lambda}\right) \end{aligned}$$

so that the upper bounds in the theorem are asymptotically precise as $m \to \infty$.

Good implementations of OADH have been shown in theory and practice to be well described by the simple uniform hashing model above, so we may safely use the above results.

However, OALP is not well described by the uniform hashing model, because of its clustering behaviour. It can be analysed, however, in a similar but more complicated manner.

Lemma 3.31. Assuming uniformly hashed random input, the expected number of probes for successful, $T_{\mathrm{ss}}(\lambda)$, and unsuccessful, $T_{\mathrm{us}}(\lambda)$, search in a hash table using OALP are, respectively,

$$T_{\mathrm{ss}}(\lambda) \approx 0.5\left(1 + \frac{1}{1-\lambda}\right) \quad \text{and} \quad T_{\mathrm{us}}(\lambda) \approx 0.5\left(1 + \left(\frac{1}{1-\lambda}\right)^2\right)$$

Proof. The proof is beyond the scope of this book (see Notes). □

The relationships in Lemma 3.31 and Lemma 3.30 completely fail when $\lambda = 1$. But the latter situation indicates a full hash table, and we should avoid getting close to it anyway.

Unlike OALP and OADH, the time estimates for separate chaining (SC) remain valid with data removals. Because each chain may keep several table elements, the load factor may be more than 1.

Table 3.6 presents the above theoretical estimates of the search time in the OALP, OADH, and SC hash tables under different load factors. Average time measurements

Table 3.6: Average search time bounds in hash tables with load factor λ.

λ	Successful search: $T_{ss}(\lambda)$			Unsuccessful search: $T_{us}(\lambda)$		
	SC	**OALP**	**OADH**	**SC**	**OALP**	**OADH**
0.10	1.05	1.06	1.05	1.10	1.12	1.11
0.25	1.12	1.17	1.15	1.25	1.39	1.33
0.50	1.25	1.50	1.39	1.50	2.50	2.0
0.75	1.37	2.50	1.85	1.75	8.50	4.0
0.90	1.45	5.50	2.56	1.90	50.5	10.0
0.99	1.49	50.5	4.65	1.99	5000.0	100.0

for actual hash tables [12] are close to the estimates for SC tables in the whole range $\lambda \leq 0.99$ and seem to be valid for larger values of λ, too. The measurements for OADH tables remain also close to the estimates up to $\lambda = 0.99$. But for OALP tables, the measured time is considerably less than the estimates if $\lambda > 0.90$ for a successful search and $\lambda > 0.75$ for an unsuccessful search.

Example 3.32. The expected performance of hashing depends only on the load factor. If $\lambda = 0.9$, OADH double hashing takes on the average 2.56 and 10 probes for successful and unsuccessful search, respectively. But if $\lambda = 0.5$, that is, the same keys are stored in a roughly twice larger table, the same numbers decrease to 1.39 and 2 probes.

Implementation of hashing

RESIZING One problem with open addressing is that successive insertions may cause the table to become almost full, which degrades performance. Eventually we will need to increase the table size. Doing this each time an element is inserted is very inefficient. It is better to use an upper bound, say 0.75, on the load factor, and to double the array size when this threshold is exceeded. This will then require recomputing the addresses of each element using a new hash function.

The total time required to resize, when growing a table from 0 to $m = 2^k$ elements, is of order $1 + 2 + 4 + 8 + \cdots + 2^{k-1} = 2^k - 1 = m - 1$. Since the m insertions take time of order m (recall the table always has load factor bounded away from 1), the average insertion time is still $\Theta(1)$.

DELETION It is quite easy to delete a table entry from a hash table with separate chaining (by mere node deletion from a linked list). However, open addressing encounters difficulties. If a particular table entry is physically removed from a OA-hash table leaving an empty entry in that place, the search for subsequent keys becomes invalid. This is because the OA-search terminates when the probe sequence encounters an empty table entry. Thus if a previously occupied entry is emptied, all probe sequences that previously travelled through that entry will now terminate before reaching the right location.

To avoid this problem, the deleted entry is normally marked in such a way that insertion and search operations can treat it as an empty and nonempty location, respectively. Unfortunately, such a policy results in hash tables packed with entries which are marked as deleted. But in this case the table entries can be rehashed to preserve only actual data and really delete all marked entries. In any case, the time to delete a table entry remains $O(1)$ both for SC and OA hash tables.

CHOOSING A HASH FUNCTION Ideally, the hash function, $h(k)$, has to map keys uniformly and randomly onto the entire range of hash table addresses. Therefore, the choice of this function has much in common with the choice of a generator of uniformly distributed pseudorandom numbers. A randomly chosen key k has to equiprobably hash to each address so that uniformly distributed keys produce uniformly distributed indices $h(k)$. A poorly designed hash function distributes table addresses nonuniformly and tends to cluster indices for contiguous clusters of keys. A well designed function scatters the keys as to avoid their clustering as much as possible.

If a set of keys is fixed, there always exists a ***perfect hash function*** that maps the set one-to-one onto a set of table indices and thus entirely excludes collisions. However, the problem is how to design such a function as it should be computed quickly but without using large tables. There exist techniques to design perfect hash functions for given sets of keys. But perfect hashing is of very limited interest because in most applications data sets are not static and the sets of keys cannot be pre-determined.

Four basic methods for choosing a hash function are ***division***, ***folding***, ***middle-squaring***, and ***truncation***.

Division assuming the table size is a prime number m and keys, k, are integers, the quotient, $q(k,m) = \lfloor \frac{k}{m} \rfloor$, and the remainder, $r(k,m) = k \mod m$, of the integer division of k by m specify the probe decrement for double hashing and the value of the hash function $h(k)$, respectively:

$$h(k) = r(k,m) \text{ and } \Delta(k) = \max\{1,\ q(k,m) \bmod m\} \quad .$$

The probe decrement is put to the range $[1,\ldots,m-1]$ because all decrements should be nonzero and point to the indices $[0,1,\ldots,m-1]$ of the table. The reason that m should be prime is that otherwise some slots may be unreachable by a probe sequence: for example if $m = 12$ and $\Delta(k) = 16$, only 3 slots will be probed before the sequence returns to the starting position.

Folding an integer key k is divided into sections and the value $h(k)$ combines sums, differences, and products of the sections (for example, a 9-digit decimal key, such as $k = 013402122$, can be split into three sections: 013, 402, and 122, to be added together for getting the value $h(k) = 537$ in the range $[0,\ldots,2997]$).

Middle-squaring a middle section of an integer key k, is selected and squared, then a middle section of the result is the value $h(k)$ (for example, the squared middle

section, 402, of the above 9-digit key, $k = 013402122$, results in 161604, and the middle four digits give the value $h(k) = 6160$ in the range $[0, \ldots, 9999]$).

Truncation parts of a key are simply cut out and the remaining digits, or bits, or characters are used as the value $h(k)$ (for example, deletion of all but last three digits in the above 9-digit key, $k = 013402122$, gives the value $h(k) = 122$ in the range $[0, \ldots, 999]$). While truncation is extremely fast, the keys do not scatter randomly and uniformly over the hash table indices. This is why truncation is used together with other methods, but rarely by itself.

Many real-world hash functions combine some of the above methods.

We conclude by discussing the idea of *universal hashing*. We have seen (in the section on quicksort) the idea of using randomization to protect against bad worst-case behaviour. An analogous idea works for hashing.

If a hash table is dynamically changing and its elements are not known in advance, any fixed hash function can result in very poor performance on certain inputs, because of collisions. Universal hashing allows us to reduce the probability of this occurrence by randomly selecting the hash function at run time from a large set of such functions. Each selected function still may be bad for a particular input, but with a low probability which remains equally low if the same input is met once again. Due to its internal randomisation, universal hashing behaves well even for totally nonrandom inputs.

Definition 3.33. Let K, m, and F denote a set of keys, a size of a hash table (the range of indices), and a set of hash functions mapping K to $\{0, \ldots, m-1\}$, respectively. Then F is a ***universal class*** if any distinct pair of keys $k, \kappa \in K$ collide for no more than $\frac{1}{m}$ of the functions in the class F, that is,

$$\frac{1}{|F|} \left| \{h \in F \mid h(k) = h(\kappa)\} \right| \leq \frac{1}{m} \quad .$$

Thus in the universal class all key pairs behave well and the random selection of a function from the class results in a probability of at most $\frac{1}{m}$ that any pair collide.

One popular universal class of hashing functions is produced by a simple division method. It assumes the keys are integers and cardinality of the set of keys K is a prime number larger than the largest actual key. The size m of the hash table can be arbitrary. This universal class is described by the next theorem.

Theorem 3.34 (Universal Class of Hash Functions)**.** Let $K = \{0, \ldots, p-1\}$ and $|K| = p$ be a prime number. For any pair of integers $a \in \{1, \ldots, p-1\}$ and $b \in \{0, \ldots, p-1\}$, let $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$. Then

$$F = \{h_{a,b} \mid 1 \leq a < p \text{ and } 0 \leq b < p\}$$

is a universal class.

Proof. It is easily shown that the number of collisions in the class F,

$$\left|\{h \in F \mid h(k) = h(\kappa);\ k, \kappa \in K\}\right|,$$

is the number of distinct numbers (x,y); $0 \le x,y < p$ such that $x \bmod m = y \bmod m$. Let us denote the latter property: $x \equiv y \pmod m$. It is evident that $h_{a,b}(k) = h_{a,b}(\kappa)$ iff

$$(ak+b) \bmod p \equiv (a\kappa+b) \bmod p \pmod m \quad .$$

Then for any fixed $k, \kappa < p$, there is one-to-one correspondence between the pairs (a,b) such that $0 < a < p$ and $0 < b < p$ and $h_{a,b}(k) = h_{a,b}(\kappa)$, and the pairs of distinct numbers (x,y) with the property that $0 \le x,y < p$ and $x \equiv y \pmod n$. The correspondence is given in one direction by

$$x = (ak+b) \bmod p;\ \ y = (a\kappa+b) \bmod p$$

where $x \neq y$ since

$$\{az+b \mid\ z = 0,\ldots,p-1\} = \{0,\ldots,p-1\}$$

when p is prime and $a \neq 0$. In the other direction the correspondence is given by the condition that a and b are the unique integers in $\{0,\ldots,p-1\}$ such that

$$ak+b \equiv x \pmod p \ \text{ and } \ a\kappa+b \equiv y \pmod p \quad .$$

These equations have a unique solution for a and b since p is prime, and $a \neq 0$ since $x \neq y$.

Clearly $|F| = p(p-1)$. Now let us find out how many pairs of distinct numbers (x,y) exist such that $0 \le x,y < p$ and $x \equiv y \pmod m$. For any fixed $s < m$ there are at most $\lceil \frac{p}{m} \rceil$ numbers $x < p$ such that $x \equiv s \pmod m$. Since p and m are integers, $\lceil \frac{p}{m} \rceil \le \frac{p-1}{m} + 1$. Therefore for each $x < p$ there are no more than $\frac{p}{m} - 1 \le \frac{p-1}{m}$ numbers $y < p$ distinct from x such that $x \equiv y \pmod m$, and the total number of such pairs (x,y) is at most $\frac{p(p-1)}{m}$. Hence for any fixed distinct pair of keys (k,κ) the fraction of F that cause k and κ to collide is at most $\frac{1}{m}$, so the class F is universal. □

This suggests the following strategy for choosing a hash function at run time: (i) find the current size of the set of keys to hash; (ii) select the next prime number p larger than the size of the key set found; (iii) randomly choose integers a and b such that $0 < a < p$ and $0 \le b < p$, and (iv) use the function $h_{a,b}$ defined in Theorem 3.34.

Exercises

Exercise 3.5.1. The Java programming language (as of time of writing) uses the following hash function h for character strings. Each character has a Unicode value represented by an integer (for example, the upper case letters $A, B, \ldots, Z$ correspond

to $65,66,\ldots,90$ and the lower case $a,b,\ldots,z$ correspond to $97,98,\ldots,122$). Then h is computed using 32-bit integer addition via

$$h(s) = s[0]*31^{n-1} + s[1]*31^{n-2} + \cdots + s[n-1]*31 + s[n].$$

Find two 2-letter strings that have the same hash value. How could you use this to make 2^{100} different strings all of which have the same hash code?

Exercise 3.5.2. Place the sequence of keys $k = 10,26,52,76,13,8,3,33,60,42$ into a hash table of size 13 using the modulo-based hash address $i = k \mod 13$ and linear probing to resolve collisions.

Exercise 3.5.3. Place the sequence of keys $k = 10,26,52,76,13,8,3,33,60,42$ into a hash table of size 13 using the modulo-based hash address $i = k \mod 13$ and double hashing with the secondary hash function $\Delta(k) = \max\{1, k/13\}$ to resolve collisions.

Exercise 3.5.4. Place the sequence of keys $k = 10,26,52,76,13,8,3,33,60,42$ into a hash table of size 13 using separate chaining to resolve collisions.

3.6 Notes

Binary search, while apparently simple, is notoriously hard to program correctly even for professional programmers: see [2] for details.

The expected height of a randomly grown BST was shown to be $\Theta(\log n)$ by J. M. Robson in 1979. After much work by many authors it is now known that the average value is tightly concentrated around $\alpha \ln n$ where α is the root of $x\ln(2e/x) = 1$, $\alpha \cong 4.311$.

The historically first balanced binary search tree was proposed in 1962 by G. M. Adelson-Velskii and E. M. Landis, hence the name AVL tree. Red-black trees were developed in 1972 by R. Bayer under the name "symmetric binary B-trees" and received their present name and definition from L. Guibas and R. Sedgewick in 1978. AA-trees were proposed by A. Anderson in 1993.

Multiway B-trees were proposed in 1972 by R. Bayer and E. McCreight.

According to D. Knuth, hashing was invented at IBM in early 1950's simultaneously and independently by H. P. Luhn (hash tables with SC) and G. M. Amdahl (OALP).

The analysis of OALP hashing was first performed by D. Knuth in 1962. This was the beginning of the modern research field of analysis of algorithms.

The random balls in bins model can be analysed in detail by more advanced methods than we present in this book (see for example [6]). Some natural questions are

- When do we expect all bins to have at least one ball?
- What proportion of boxes are expected to be empty when $n \approx m$?

- What is the expected maximum number of balls in a box when $n \approx m$?

The answers are applicable to our analysis of chaining: when are all chains expected to be nonempty? how many chains are empty when the average chain length is $\Theta(1)$? what is the maximum chain length when the average chain length is $\Theta(1)$? The answers are known to be, respectively: when $n \approx m \ln m$; about $e^{-\lambda}$; $\Theta(\log n / \log \log n)$. The last result is much harder to derive than the other two.

Part II

Introduction to Graph Algorithms

Chapter 4

The Graph Abstract Data Type

Many real-world applications, such as building compilers, finding routing protocols in communication networks, and scheduling processes, are well modelled using concepts from graph theory. A (directed) graph is an abstract mathematical model: a set of points and connection relations between them.

4.1 Basic definitions

Graphs can be studied from a purely mathematical point of view ("does something exist, can something be done?"). In this book we focus on algorithmic aspects of graph theory ("how do we do it efficiently and systematically?"). However, the mathematical side is where we must start, with precise definitions. The intuition behind the definitions is not hard to guess.

We start with the concept of digraph (the word stands for **di**rected **graph**). A good example to keep in mind is a street network with one-way streets.

Definition 4.1. A ***digraph*** $G = (V,E)$ is a finite nonempty set V of ***nodes*** together with a (possibly empty) set E of ordered pairs of nodes of G called ***arcs***.

Note. In the mathematical language of relations, the definition says that E is a relation on V. If $(u,v) \in E$, we say that v is ***adjacent*** to u, that v is an ***out-neighbour*** of u, and that u is an ***in-neighbour*** of v.

We can think of a node as being a point and an arc as an arrow from one node to another. This allows us to draw pictures that suggest ideas. The pictures cannot *prove* anything, however.

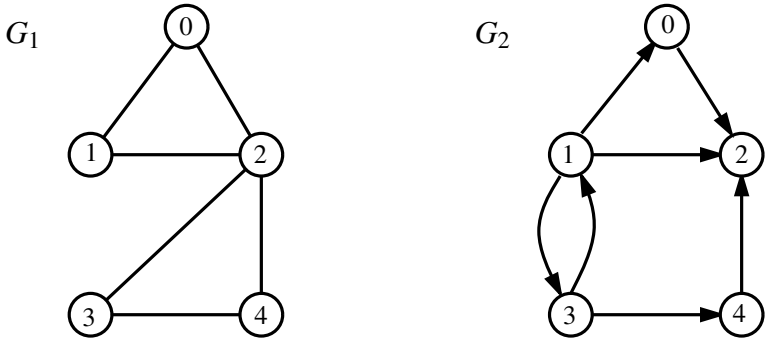


Figure 4.1: A graph G_1 and a digraph G_2.

Very often the adjacency relation is symmetric (all streets are two-way). There are two ways to deal with this. We can use a digraph that happens to be symmetric (in other words, (u,v) is an arc if and only if (v,u) is an arc). However, it is sometimes simpler to reduce this pair of arcs into a single undirected edge that can be traversed in either direction.

Definition 4.2. A ***graph*** $G=(V,E)$ is a finite nonempty set V of ***vertices*** together with a (possibly empty) set E of unordered pairs of vertices of G called ***edges***.

Note. Since we defined E to be a set, there are no multiple arcs/edges between a given pair of nodes/vertices.

Non-fluent speakers of English please note: the singular of "vertices" is not "vertice", but "***vertex***".

For a given digraph G we may also denote the set of nodes by $V(G)$ and the set of arcs by $E(G)$ to lessen any ambiguity.

Example 4.3. We display a graph G_1 and a digraph G_2 in Figure 4.1. The nodes/vertices are labelled $0,1,\ldots$ as in the picture. The arcs and edges are as follows.

$$E(G_1)=\{\{0,1\},\{0,2\},\{1,2\},\{2,3\},\{2,4\},\{3,4\}\}$$

$$E(G_2)=\{(0,2),(1,0),(1,2),(1,3),(3,1),(3,4),(4,2)\}$$

Note. Some people like to view a graph as a special type of digraph where every unordered edge $\{u,v\}$ is replaced by two directed arcs (u,v) and (v,u). This has the advantage of allowing us to consider only digraphs, and we shall use this approach in our Java implementation in Appendix B. It works in most instances.

However, there are disadvantages; for some purposes we must know whether our object is really a graph or just a symmetric digraph. Whenever there is (in our opinion) a potential ambiguity, we shall point it out.

Example 4.4. Every rooted tree (see Section D.7) can be interpreted as a digraph: there is an arc from each node to each of its children.

Every free tree is a graph of a very special type (see Appendix D.7).

Note. (Graph terminology) The terminology in this subject is unfortunately not completely standard. Some authors call a graph by the longer term "undirected graph" and use the term "graph" to mean what we call a directed graph. However when using our definition of a graph, it is standard practice to abbreviate the phrase "directed graph" with the word digraph.

We shall be dealing with both graphs and digraphs throughout these notes. In order to save writing "(di)graph" too many times, we make the following convention. We treat the digraph as the fundamental concept. In other words, we shall use the terminology of digraphs, nodes and arcs, with the understanding that if this is changed to graphs, edges, and vertices, the resulting statement is still true. However, if we talk about graphs, edges, and vertices, our statement is not necessarily true for digraphs. Whenever a result is true for digraphs but not for graphs, we shall say this explicitly (this happens very rarely).

There is another convention to discuss. An arc that begins and ends at the same node is called a ***loop***. We make the convention that *loops are not allowed in our digraphs*. Again, other authors may differ. If our conventions are relaxed to allow multiple arcs and/or loops, many of the algorithms below work with no modification or with only very minor modification required. However dealing with loops frequently requires special cases to be considered, and would distract us from our main goal of introducing the field of graph algorithms. As an example of the problems caused by loops, suppose that we represent a graph as a symmetric digraph as described above. How do we represent a loop in the graph?

Definition 4.5. The ***order*** of a digraph $G = (V, E)$ is $|V|$, the number of nodes. The ***size*** of G is $|E|$, the number of arcs.

We usually use n to denote $|V|$ and m to denote $|E|$.

For a given n, the value of m can be as low as 0 (a digraph consisting of n totally disconnected points) and as high as $n(n-1)$ (each node can point to each other node; recall that we do not allow loops). If m is toward the low end, the digraph is called ***sparse***, and if m is toward the high end, then the digraph is called ***dense***. These terms are obviously very informal. For our purposes we will call a class of digraphs sparse if m is $O(n)$ and dense if m is $\Omega(n^2)$.

Definition 4.6. A ***walk*** in a digraph G is a sequence of nodes $v_0 v_1 \ldots v_l$ such that, for each i with $0 \leq i < l$, (v_i, v_{i+1}) is an arc in G. The ***length*** of the walk $v_0 v_1 \ldots v_l$ is the number l (that is, the number of arcs involved).

A ***path*** is a walk in which no node is repeated. A ***cycle*** is a walk in which $v_0 = v_l$ and no other nodes are repeated.

Thus in a graph, we rule out a walk of the form uvu as a cycle (going back and forth along the same edge should not count as a cycle). A cycle in a graph must be of length at least 3.

It is easy to see that if there is a walk from u to v, then (by omitting some steps if necessary) we can find a path from u to v.

Example 4.7. For the graph G_1 of Figure 4.1 the following sequences of vertices are classified as being walks, paths, or cycles.

vertex sequence	walk?	path?	cycle?
032	no	no	no
01234	yes	yes	no
0120	yes	no	yes
123420	yes	no	no
010	yes	no	no

Example 4.8. For the digraph G_2 of Figure 4.1 the following sequences of nodes are classified as being walks, paths, or cycles.

node sequence	walk?	path?	cycle?
01234	no	no	no
312	yes	yes	no
131	yes	no	yes
1312	yes	no	no
024	no	no	no

Definition 4.9. In a graph, the ***degree*** of a vertex v is the number of edges meeting v. In a digraph, the ***outdegree*** of a node v is the number of out-neighbours of v, and the ***indegree*** of v is the number of in-neighbours of v.

A node of indegree 0 is called a ***source*** and a node of outdegree 0 is called a ***sink***.

If the nodes have a natural order, we may simply list the indegrees or outdegrees in a sequence.

Example 4.10. For our graph G_1, the degree sequence is $(2,2,4,2,2)$. The in-degree sequence and out-degree sequence of the digraph G_2 are $(1,1,3,1,1)$ and $(1,3,0,2,1)$, respectively. Node 2 is a sink.

Definition 4.11. The ***distance*** from u to v in G, denoted by $d(u,v)$, is the minimum length of a path from u to v. If no path exists, the distance is undefined (or $+\infty$).

For graphs, we have $d(u,v) = d(v,u)$ for all vertices u, v.

Example 4.12. In graph G_1 of Figure 4.1, we can see by considering all possibilities that $d(0,1) = 1$, $d(0,2) = 1$, $d(0,3) = 2$, $d(0,4) = 2$, $d(1,2) = 1$, $d(1,3) = 2$, $d(1,4) = 2$, $d(2,3) = 1$, $d(2,4) = 1$ and $d(3,4) = 1$.

In digraph G_2, we have, for example, $d(0,2) = 1, d(3,2) = 2$. Since node 2 is a sink, $d(2,v)$ is not defined unless $v = 2$, in which case the value is 0.

There are several ways to create new digraphs from old ones.

One way is to delete (possibly zero) nodes and arcs in such a way that the resulting object is still a digraph (there are no arcs missing any endpoints!).

Definition 4.13. A ***subdigraph*** of a digraph $G = (V,E)$ is a digraph $G' = (V',E')$ where $V' \subseteq V$ and $E' \subseteq E$. A ***spanning*** subdigraph is one with $V' = V$; that is, it contains all nodes.

Example 4.14. Figure 4.2 shows (on the left) a subdigraph and (on the right) a spanning subdigraph of the digraph G_2 of Figure 4.1.

Figure 4.2: A subdigraph and a spanning subdigraph of G_2.

Definition 4.15. The subdigraph ***induced*** by a subset V' of V is the digraph $G' = (V',E')$ where $E' = \{(u,v) \in E \mid u \in V' \text{ and } v \in V'\}$.

Example 4.16. Figure 4.3 shows the subdigraph of the digraph G_2 of Figure 4.1 induced by $\{1,2,3\}$.

We shall sometimes find it useful to "reverse all the arrows".

Definition 4.17. The ***reverse digraph*** of the digraph $G = (V,E)$, is the digraph $G_r = (V,E')$ where $(u,v) \in E'$ if and only if $(v,u) \in E$.

Example 4.18. Figure 4.4 shows the reverse of the digraph G_2 of Figure 4.1.

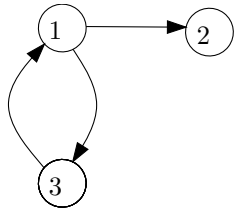


Figure 4.3: The subdigraph of G_2 induced by $\{1,2,3\}$.

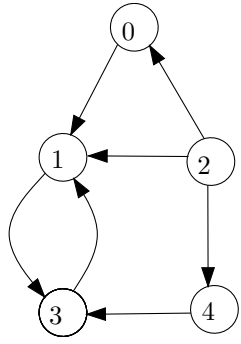


Figure 4.4: The reverse of digraph G_2.

It is sometimes useful to replace all one-way streets with two-way streets. The formal definition must take care not to introduce multiple edges. Note below that if (u,v) and (v,u) belong to E, then only one edge joins u and v in G'. This is because $\{u,v\}$ and $\{v,u\}$ are equal as sets, so appear only once in the set E'.

Definition 4.19. The ***underlying graph*** of a digraph $G = (V,E)$ is the graph $G' = (V,E')$ where $E' = \{\{u,v\} \mid (u,v) \in E\}$.

Example 4.20. Figure 4.5 shows the underlying graph of the digraph G_2 of Figure 4.1.

We may need to combine two or more digraphs $G_1, G_2, \ldots G_k$ into a single graph where the vertices of each G_i are completely disjoint from each other and no arc goes between the different G_i. The constructed graph G is called the ***graph union***, where $V(G) = V(G_1) \cup V(G_2) \cup \ldots \cup V(G_k)$ and $E(G) = E(G_1) \cup E(G_2) \cup \ldots \cup E(G_k)$.

Exercises

Exercise 4.1.1. Prove that in a digraph, the sum of all outdegrees equals the sum of all indegrees. What is the analogous statement for a graph?

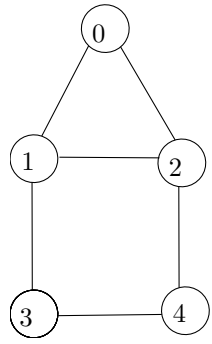


Figure 4.5: The underlying graph of G_2.

Exercise 4.1.2. Let G be a digraph of order n and u, v nodes of G. Show that $d(u,v) \leq n-1$ if there is a walk from u to v.

Exercise 4.1.3. Prove that in a sparse digraph, the average indegree of a node is $O(1)$, while in a dense digraph, the average indegree of a node is $\Omega(n)$.

4.2 Digraphs and data structures

In order to process digraphs by computer we first need to consider how to represent them in terms of data structures. There are two common computer representations for digraphs, which we now present. We assume that the digraph has the nodes given in a fixed order. Our *convention* is that the vertices are labelled $0, 1, \ldots, n-1$.

Definition 4.21. Let G be a digraph of order n. The ***adjacency matrix*** of G is the $n \times n$ boolean matrix (often encoded with 0's and 1's) such that entry (i, j) is true if and only if there is an arc from the node i to node j.

Definition 4.22. For a digraph G of order n, an ***adjacency lists*** representation is a sequence of n sequences, $L_0, \ldots, L_{n-1}$. Sequence L_i contains all nodes of G that are adjacent to node i.

The sequence L_i may or may not be sorted in order of increasing node number. Our *convention* is to sort them whenever it is convenient. (However, many implementations, such as the one given in Appendix B, do *not* enforce that their adjacency lists be sorted.)

We can see the structure of these representations more clearly with examples.

Example 4.23. For the graph G_1 and digraph G_2 of Example 4.3, the adjacency matrices are given below.

$$G_1: \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix} \qquad G_2: \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

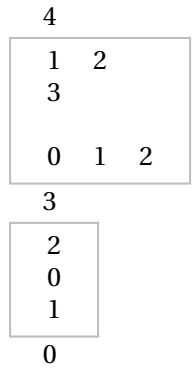
Notice that the number of 1's in a row (column) is the outdegree (indegree) of the corresponding node. The corresponding adjacency lists are now given.

$$G_1: \begin{array}{llll} 1 & 2 & & \\ 0 & 2 & & \\ 0 & 1 & 3 & 4 \\ 2 & 4 & & \\ 2 & 3 & & \end{array} \qquad G_2: \begin{array}{lll} 2 & & \\ 0 & 2 & 3 \\ & & \\ 1 & 4 & \\ 2 & & \end{array}$$

Note. Only the out-neighbours are listed in the adjacency lists representation. An empty sequence can occur (for example, sequence 2 of the digraph G_2). If the nodes are not numbered in the usual way (for example, they are numbered $0,\ldots,n-1$ or labelled $A,B,C,\ldots$), we may include these labels if necessary.

It is often useful to input several digraphs from a single file. Our standard format is as follows. The file consists of several digraphs one after the other. To distinguish the beginning of one and the end of the other we have a single line giving the order at the beginning of each graph. If the order is n then the next n lines give the adjacency matrix or adjacency lists representation of the digraph. The end of the file is marked with a line denoting a digraph of order 0.

Example 4.24. Here is a file consisting of 3 digraphs, of orders 4, 3, 0 respectively. The first contains a sink and hence there is a blank line.



There are also other specialized (di)graph representations besides the two mentioned in this section. These data structures take advantage of special structure for improved storage or access time, often for families of graphs sharing a common property. For such specialized purposes they may be better than either the adjacency matrix or lists representations.

For example, trees can be stored more efficiently. We have already seen in Section 2.5 how a complete binary tree can be stored in an array. A general rooted tree of n nodes can be stored in an array *pred* of size n. The value $pred[i]$ gives the parent of node i. The root is a special case and can be given value -1 (representing a NULL pointer), for example, if we number nodes from 0 to $n-1$ in the usual way. This of course is a form of adjacency lists representation, where we use in-neighbours instead of out-neighbours.

We will sometimes need to represent ∞ when processing graphs. For example, it may be more convenient to define $d(u,v) = \infty$ than to say it is undefined. From a programming point of view, we can use any positive integer that can not be confused with any other that might legitimately arise. For example, the distance between 2 nodes in a digraph on n nodes cannot be more than $n-1$ (see Exercise 4.1.2). Thus in this case we may use n to represent the fact that there is no path between a given pair of nodes. We shall return to this subject in Chapter 6.

Exercises

Exercise 4.2.1. Write down the adjacency matrix of the digraph of order 7 whose adjacency lists representation is given below.

2		
0		
0	1	
4	5	6
5		
3	4	6
1	2	

Exercise 4.2.2. Consider the digraph G of order 7 whose adjacency matrix representation is given below.

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

Write down the adjacency lists representation of G.

Exercise 4.2.3. Consider the digraph G of order 7 given by the following adjacency lists representation.

2		
0		
0	1	
4	5	6
5		
3	4	6
1	2	

Write down the adjacency matrix representation of the reverse digraph G_r.

Exercise 4.2.4. Consider the digraph G whose nodes are the integers from 1 to 12 inclusive and such that (i, j) is an arc if and only if i is a proper divisor of j (that is, i divides j and $i \neq j$).

Write down the adjacency matrix representation of G and of G_r.

Exercise 4.2.5. Write the adjacency lists and adjacency matrix representation for a complete binary tree with 7 vertices, assuming they are ordered $1, \ldots, 7$ as in Section 2.5.

Table 4.1: Digraph operations in terms of data structures.

Operation	**Adjacency Matrix**	**Adjacency Lists**
arc (i, j) exists?	is entry (i, j) 0 or 1	find j in list i
outdegree of i	scan row, count 1's	size of list i
indegree of i	scan column, count 1's	for $j \neq i$, find i in list j
add arc (i, j)	change entry (i, j)	insert j in list i
delete arc (i, j)	change entry (i, j)	delete j from list i
add node	create new row and column	add new list at end
delete node i	delete row/column i shuffle other entries	delete list i for $j \neq i$, delete i from list j

4.3 Implementation of digraph ADT operations

In this section we discuss the implementation of basic digraph operations we have seen in Section 4.1.

A matrix is simply an array of arrays. The lists representation is really a list of lists, but a list can be implemented in several ways, for example by an array or singly- or doubly-linked list using pointers. These have different properties (see Section C.1); for example, accessing the middle element is $\Theta(1)$ for an array but $\Theta(n)$ for a linked list. In any case, however, to *find* a value that may or may not be in the list requires sequential search and takes $\Theta(n)$ time in the worst case.

We now discuss the comparative performance of some basic operations using the different data structures. In Table 4.1 we show how the basic graph operations can be described in terms of the adjacency matrix or lists representations.

For example, suppose we wish to check whether arc (i, j) exists. Using the adjacency matrix representation, we are simply accessing an array element twice. However, with the lists representation, we will need to find j in list i. Adding a node in the lists case is easy, since we just add an empty list at the end, but adding a node with the matrix representation requires us to allocate an extra row and column of zeros. Deleting a node (which perhaps necessitates deleting some arcs) is trickier. In the matrix case, we must delete a row and column, and move up some elements so there are no gaps in the matrix. In the lists case, we must remove a list and also all references to the deleted node in other lists. This requires scanning each list for the offending entry and deleting it.

In Table 4.2 we compare the performance of two different data structures. There are several ways to implement a list, and hence the square of that number of ways to implement an adjacency lists representation. We have chosen one, an array of doubly linked lists, for concreteness.

For example, finding j in list i will take time in the worst case $\Theta(d)$, where d is the

Table 4.2: Comparative worst-case performance of adjacency lists and matrices.

Operation	**matrix**	**lists**
arc (i,j) exists?	$\Theta(1)$	$\Theta(d)$
outdegree of i	$\Theta(n)$	$\Theta(1)$
indegree of i	$\Theta(n)$	$\Theta(n+m)$
add arc (i,j)	$\Theta(1)$	$\Theta(1)$
delete arc (i,j)	$\Theta(1)$	$\Theta(d)$
add node	$\Theta(n)$	$\Theta(1)$
delete node i	$\Theta(n^2)$	$\Theta(n+m)$

size of list i, which equals the outdegree of i. In the worst case this might be $\Theta(m)$ since all nodes but i might be sinks. On the other hand, for a sparse digraph, the average outdegree is $\Theta(1)$, so arc lookup can be done on average in constant time. Note that if we want to print out all arcs of a digraph, this will take time $\Theta(n+m)$ in the lists case and $\Theta(n^2)$ in the matrix case.

Finding outdegree with the lists representation merely requires accessing the correct list (constant time) plus finding the size of that list (constant time). Finding indegree with the lists representation requires scanning all lists except one, and this requires us to look at every arc in the worst case, taking time $\Theta(n+m)$ (the n is because we must consider every node's list even if it is empty). If we wish to compute just one indegree, this might be acceptable, but if all indegrees are required, this will be inefficient. It is better to compute the reverse digraph once and then read off the outdegrees, this last step taking time $\Theta(n)$ (see Exercise 4.3.1).

One way around all this work is to use in our definition of adjacency lists representation, instead of just the out-neighbours, a list of in-neighbours also. This may be useful in some contexts but in general requires more space than is needed.

We conclude by discussing space requirements. The adjacency matrix representation requires $\Theta(n^2)$ storage: we simply need n^2 bits. It appears that an adjacency lists representation requires $\Theta(n+m)$ storage, since we must store an endpoint of each arc, and we need to allocate space for each node's list. However this is not strictly true for large graphs. Each node number requires some storage; the number k requires on average $\Theta(\log k)$ bits. If, for example, we have a digraph on n nodes where every possible arc occurs, then the total storage required is of order $n^2 \log n$, worse than with a matrix representation. For small, sparse digraphs, it is true that lists use less space than a matrix, whereas for small dense digraphs the space requirements are comparable. For large sparse digraphs, a matrix can still be more efficient, but this happens rarely.

The remarks above show that it is not immediately clear which representation to

use. We will mostly use adjacency lists, which are clearly superior for many common tasks (such as graph traversals, covered in Chapter 5) and generally better for sparse digraphs.

Any implementation of an abstract data type (for example as a Java class) must include objects and "methods". While most people would include methods for adding nodes, deleting arcs, and so on, it is not clear where to draw the line. In Appendix B, one way of writing Java classes to deal with graphs is presented in detail. There are obviously a lot of different choices one can make. In particular for our Java lists representation we use `ArrayList<ArrayList<Integer>>`.

Exercises

Exercise 4.3.1. Show how to compute the (sorted) adjacency lists representation of the reverse digraph of G from the (sorted) adjacency lists representation of G itself. It should take time $\Theta(n+m)$.

4.4 Notes

This chapter shows how to represent and process graphs with a computer. Although Appendix B uses the Java programming language, the ideas and algorithms are applicable to other industrial programming languages. For example, C++ has several standard graph algorithms libraries such as the Boost Graph Library [11], LEDA (Library of Efficient Data structures and Algorithms) [9] and GTL (Graph Template Library) [3]. All algorithms discussed here are provided in these libraries and in other mathematical interpreted languages like Mathematica and Maple.

Chapter 5

Graph Traversals and Applications

Many graph problems require us to visit each node of a digraph in a systematic way. For example, we may want to print out the labels of the nodes in some order, or perhaps we are in a maze and we have no idea where to find the door. More interesting examples will be described below. The important requirements are that we must be systematic (otherwise an algorithm is hard to implement), we must be complete (visit each node at least once), and we must be efficient (visit each node at most once).

There are several ways to perform such a traversal. Here we present the two most common, namely breadth-first search and depth-first search. We also discuss a more general but also more complicated and slower algorithm, priority-first search. First we start with general remarks applicable to all graph traversals.

5.1 Generalities on graph traversal

We first discuss a general skeleton common to all graph traversals. In Figure 5.1, this general skeleton is presented in pseudocode. We first describe it less formally.

Suppose that G is a digraph. Choose a node v as a starting point. We shall visit all nodes reachable from v in G using an obvious method.

At each stage, there are three possible types of nodes. *White* nodes are those that have not yet been visited. *Grey* nodes (or ***frontier nodes***) are those that have been visited but may have adjacent nodes that are white. Finally, *black* nodes are those nodes that have been visited, along with all their adjacent nodes. This is shown pictorially in Figure 5.2.

Initially, all nodes are white. The traversal algorithm first colours v grey. At each subsequent step, it chooses a grey node w and then a white node x adjacent to w

```
algorithm traverse
    Input: digraph G
begin
    array colour[0..n−1], pred[0..n−1]
    for u ∈ V(G) do
        colour[u] ← WHITE
    end for
    for s ∈ V(G) do
        if colour[s] = WHITE then
            visit(s)
        end if
    end for
    return pred
end

algorithm visit
    Input: node s of digraph G
begin
    colour[s] ← GREY; pred[s] ← NULL
    while there is a grey node do
        choose a grey node u
        if there is a white neighbour of u
            choose such a neighbour v
            colour[v] ← GREY; pred[v] ← u
        else colour[u] ← BLACK
        end if
    end while
end
```

Figure 5.1: Graph traversal schema.

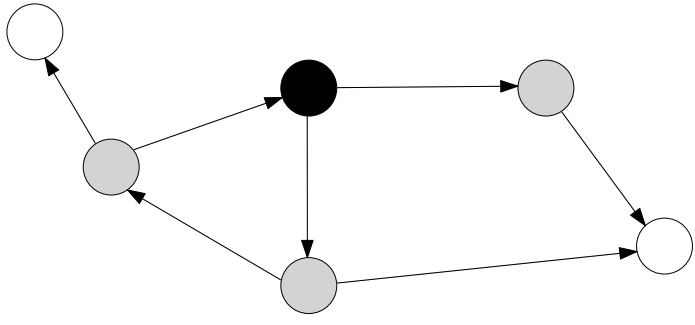


Figure 5.2: Node states in the middle of a digraph traversal.

(thereby changing the colour of x to grey). This possibly causes some grey nodes to become black. When all nodes are black, there can be no white nodes reachable from the current set of visited nodes. We have reached every node that can be reached from the root, and the traversal from v terminates.

At this stage we have created a subdigraph of G that is a tree: the nodes are precisely the black nodes, and the arcs are those arcs connecting a grey node to its white neighbour at the instant before the white node turns grey.

There may remain unvisited nodes in the digraph. In this case, we may choose another white node as root and continue. Eventually, we obtain a set of disjoint trees spanning the digraph, which we call the ***search forest***.

The above is formalized in Figure 5.1. Here `traverse` simply chooses a new root s when required. The main work is done by `visit`. In this procedure, the array *pred* stores the parent of each node; each root gets the value NULL, since it has no parent. Each time through the while-loop, a white node is turned grey or a grey node is turned black. Thus eventually there are no white nodes reachable from s and the procedure terminates. If a node u is reachable from s then it is visited during the call to `visit` with input s, unless it had already been visited in a previous tree.

Now once a traversal has been completed and a search forest obtained, we may classify the arcs into four distinct types. This classification will be useful later.

Definition 5.1. Suppose we have performed a traversal of a digraph G, resulting in a search forest F. Let $(u,v) \in E(G)$ be an arc. It is called a ***tree arc*** if it belongs to one of the trees of F. If the arc is not a tree arc, there are three possibilities. A non-tree arc is called a ***forward arc*** if u is an ancestor of v in F, a ***back arc*** if u is a descendant of v in F, and a ***cross arc*** if neither u nor v is an ancestor of the other in F.

In the first three cases in Definition 5.1, u and v must belong to the same tree of F. However, a cross arc may join two nodes in the same tree or point from one tree to another.

The following theorem collects all the basic facts we need for proofs in later sections. Figure 5.3 illustrates the first part.

Theorem 5.2. Suppose that we have carried out `traverse` on G, resulting in a search forest F. Let $v,w \in V(G)$.

- Let T_1 and T_2 be different trees in F and suppose that T_1 was explored before T_2. Then there are no arcs from T_1 to T_2.
- Suppose that G is a graph. Then there can be no edges joining different trees of F.
- Suppose that v is visited before w and w is reachable from v in G. Then v and w belong to the same tree of F.

- Suppose that v and w belong to the same tree T in F. Then any path from v to w in G must have all nodes in T.

Proof. If the first part were not true, then since w is reachable from v, and w has not been visited before T_1 is started, w must be reached in the generation of T_1, contradicting $w \in T_2$. The second part follows immediately for symmetric digraphs and hence for graphs. Now suppose that v is seen before w. Let r be the root of the tree T containing v. Then w is reachable from r and so since it has not already been visited when r is chosen, it belongs to T. Finally, if v and w are in the same tree, then any path from v to w in G going outside the tree must re-enter it via some arc; either the leaving or the entering arc will contradict the first part. □

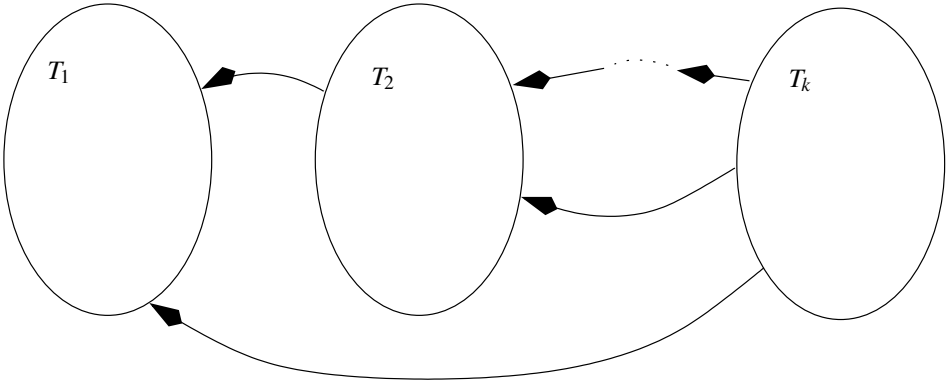


Figure 5.3: Decomposition of a digraph in terms of search trees.

We now turn to the analysis of `traverse`. The generality of our traversal procedure makes its complexity hard to determine. Its running time is very dependent on how one chooses the next grey node u and its white neighbour v. It also apparently depends on how long it takes to determine whether there exist any grey nodes or whether u has any white neighbours. However, any sensible rule for checking existence of either type of node should simply return false if there is no such node, and take no more time in this case than if it does find one. Thus we do not need to take account of the checking in our analysis.

Since the initialization of the array *colour* takes time $\Theta(n)$, the amount of time taken by `traverse` is clearly $\Theta(n+t)$, where t is the total time taken by all the calls to `visit`.

Each time through the while-loop of `visit` a grey node is chosen, and either a white node is turned grey or a grey node is turned black. Note that the same grey node can be chosen many times. Thus we execute the while-loop in total $\Theta(n)$ times since every node must eventually move from white through grey to black. Let a, A be

lower and upper bounds on the time taken to choose a grey node (note that they may depend on n and be quite large if the rule used is not very simple). Then the time taken in choosing grey nodes is $O(An)$ and $\Omega(an)$. Now consider the time taken to find a white neighbour. This will involve examining each neighbour of u and checking whether it is white, then applying a selection rule. If the time taken to apply the rule is at least b and at most B (which may depend on n), then the total time in choosing white neighbours is $O(Bm)$ and $\Omega(bm)$ if adjacency lists are used and $O(Bn^2)$ and $\Omega(bn^2)$ if an adjacency matrix is used.

In summary, then, the running time of `traverse` is $O(An+Bm)$ and $\Omega(an+bm)$ if adjacency lists are used, and $O(An+Bn^2)$ and $\Omega(an+bn^2)$ if adjacency matrix format is used.

A more detailed analysis would depend on the rule used. We shall see in Section 5.2 that BFS and DFS have a,b,A,B all constant, and so each yields a linear-time traversal algorithm. In this case, assuming a sparse input digraph, the adjacency list format seems preferable. On the other hand, for example, suppose that a is at least of order n (a rather complex rule for grey nodes is being used) and b,B are constant (for example, the first white node found is chosen). Then asymptotically both representations take time $\Omega(n^2)$, so using the adjacency matrix is not clearly ruled out (it may even be preferable if it makes programming easier).

Exercises

Exercise 5.1.1. Draw a moderately complicated graph representing a maze (corridors are edges and intersections are nodes). Label one node as the start and another as the end. One rule for getting through a maze is to try to go forward, always make a right turn when faced with a choice of direction, and back up as little as possible when faced with a dead end. Apply this method to your example. Interpret what you do in terms of the procedure `traverse`.

Exercise 5.1.2. Suppose that in `traverse`, the grey node is chosen at random and so is the white node. Find your way through your maze of the previous exercise using this method.

Exercise 5.1.3. Give an example for each of the following items.

- A search forest in which a cross arc points from one tree to another.
- A search forest in which a cross arc joins two nodes in the same tree.

5.2 DFS and BFS

So far everything has been discussed at a very general level. To proceed further we need more analysis of how to guide the traversal: what rule do we use for choosing the next grey and next white node? Different rules lead to very different results. The two main rules are ***breadth-first search*** (BFS) and ***depth-first search***

(DFS) which we discuss now. We shall also discuss the more complicated ***priority-first search*** (PFS) in Section 5.5.

Breadth-first and depth-first search are dual to each other. In BFS, when we want to visit a new node, the new grey node chosen is the one that has been grey for the *longest* time. By contrast, in DFS, we choose the one that has been grey for the *shortest* time.

In BFS we start at a node v and then go to each neighbour of v (in some order), then each neighbour of a neighbour of v that has not already been visited, and so on. The search chooses grey nodes across the entire frontier between the white and grey nodes, and takes us away from the root node as slowly as possible.

By contrast, in DFS we start at a node v, but this time we "deeply" search as far away from vertex v as possible, until we cannot visit any new nodes, upon which we backtrack as little as possible. The search keeps us away from the root as long as possible.

These search concepts are best illustrated with some examples.

Example 5.3. A graph G_1 and a digraph G_2 are displayed in Figure 5.4.

Breadth-first search trees (originating from node 0) of the graph G_1 and digraph G_2 are displayed in Figure 5.5. The dashed arcs indicate the original arcs that are not part of the BFS trees.

Note that even with DFS or BFS specified there still remains the choice of which white neighbour of the chosen grey node to visit. While it does not matter what choice is made, the choice should be algorithmic. Our *convention* is to choose the one with lowest index (the nodes being numbered $0,\ldots,n-1$).

This convention for choosing white nodes means that we can reconstruct the progress of the BFS traversal completely. For example, in G_1, node 0 is visited first, then node 1, node 2, node 3, node 4, node 8, and so on. Thus we can classify the edges: for example $\{1,2\}$ is a cross edge, $\{2,8\}$ a tree edge, and so on.

We also display depth-first search trees (originating from node 0) of the graph G_1 and digraph G_2 in Figure 5.6. Again, the dashed arcs indicate the original arcs that are not part of the DFS trees.

Here, for example, in G_2 we see that the order of visiting nodes is $0,1,6,2,4,5,3$. The arc $(5,0)$ is a back arc, $(3,5)$ is a cross arc, there are no forward arcs, etc.

In the examples above, all nodes were reachable from the root, and so there is a single search tree in the forest. For general digraphs, this may not be true. We should distinguish between BFS or DFS originating from a given node, or just BFS/DFS run on a digraph. The former we call `BFSvisit`/`DFSvisit` (a special case of `visit`) and the latter just `BFS`/`DFS` (a special case of `traverse`). The algorithm `DFS`, for example, repeatedly chooses a root and runs `DFSvisit` from that root, until all nodes have been visited. Pseudocode for these procedures is given in the next two sections.

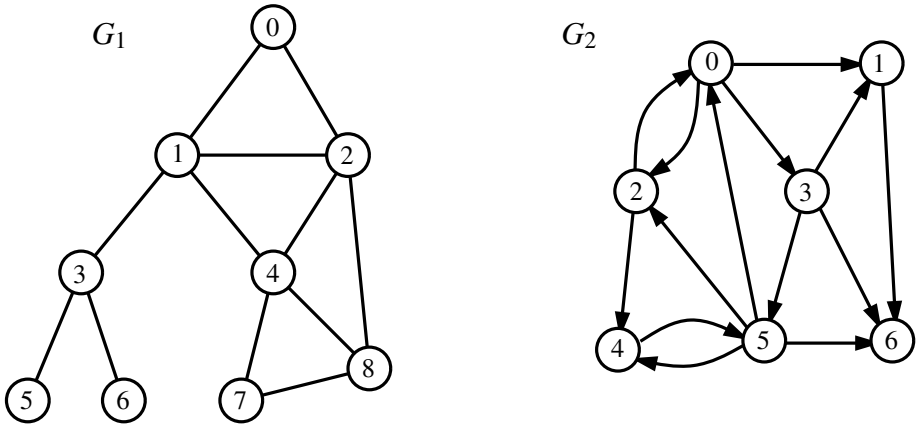


Figure 5.4: A graph G_1 and a digraph G_2.

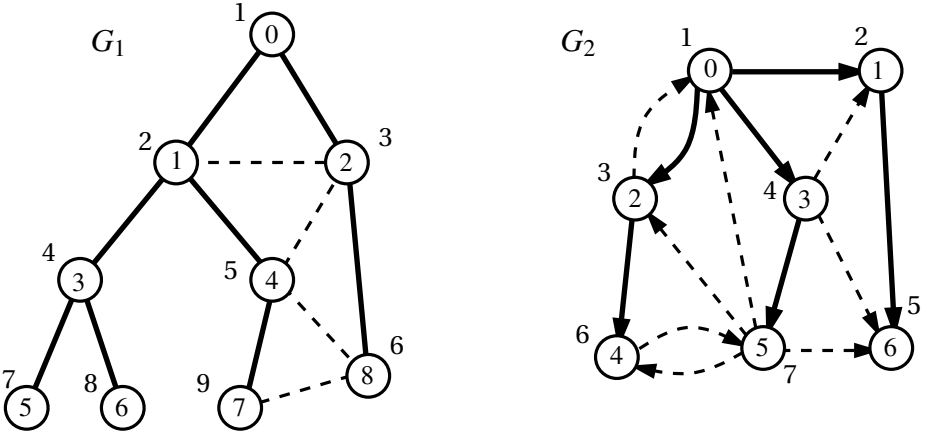


Figure 5.5: BFS trees for G_1 and G_2, rooted at 0.

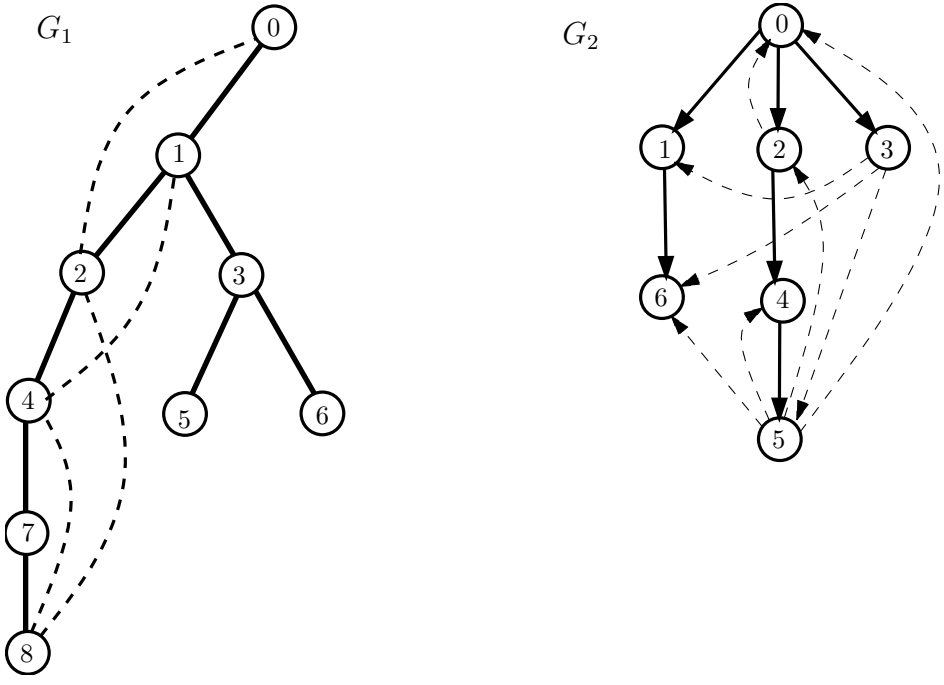


Figure 5.6: DFS trees for G_1 and G_2, rooted at 0.

Exercises

Exercise 5.2.1. Draw a graph for which DFS and BFS can visit nodes in the same order. Then draw one for which they must visit nodes in the same order. Make your examples as large as possible (maximize $n+m$).

Exercise 5.2.2. A technique called ***iterative deepening*** combines features of BFS and DFS. Starting at a root, for each $i = 0, 1, \ldots$ we search all nodes at distance at most i, using DFS (in other words, we limit the depth of the DFS to i). We do this for each i until the whole graph is explored or we reach a preassigned limit on i.

Note that we visit nodes near the root many times, so this search is not as efficient in terms of time. What is the advantage of this search technique when the graphs are too big to hold in memory?

5.3 Additional properties of depth-first search

In DFS, the next grey node chosen is the last one to be coloured grey thus far. The data structure for processing grey nodes in this "last in, first out" order is therefore a stack (see Section C.1 if necessary for the basic facts about stacks). We may store the

nodes in a stack as they are discovered. So the general `traverse` and `visit` procedures can be replaced by those in Figure 5.7. Note how the algorithm for `visit` has been altered. We loop through the nodes adjacent to the chosen grey node u, and as soon as we find a white one, we add it to the stack. We also introduce a variable *time* and keep track of the time a node was turned grey, and the time it was turned black, in the arrays $seen, done$. These will be useful later.

The complexity analysis of DFS is easy. Choosing a grey node u takes constant time since only the stack pop operation is required. The time taken to apply the selection rule to a white neighbour is also constant, since we take the first one found in the for-loop. Our analysis of `traverse` shows us that `DFS` runs in time $\Theta(n+m)$ if adjacency lists are used, and $\Theta(n^2)$ using an adjacency matrix. In summary, `DFS` is a linear-time traversal algorithm.

One nice feature of depth-first search is its recursive nature. The relationship between stacks and recursion is, we hope, well known to the reader. We can replace the `DFSvisit` procedure in this case by the recursive version in Figure 5.8.

We now note a few important facts about depth-first search that are useful in proving correctness of various algorithms later on.

Theorem 5.4. The call to `recursiveDFSvisit` with input s terminates only when all nodes reachable from s via a path of white nodes have been visited. The descendants of s in the DFS forest are precisely these nodes.

Proof. See Exercise 5.3.10. □

There are not as many possibilities for interleaving of the timestamps as there appear at first sight. In particular, we *cannot* have $seen[v] < seen[w] < done[v] < done[w]$. The following theorem explains why.

Theorem 5.5. Suppose that we have performed DFS on a digraph G, resulting in a search forest F. Let $v, w \in V(G)$ and suppose that $seen[v] < seen[w]$.

- If v is an ancestor of w in F, then

$$seen[v] < seen[w] < done[w] < done[v] \quad .$$

- If v is not an ancestor of w in F, then

$$seen[v] < done[v] < seen[w] < done[w] \quad .$$

Proof. The first part is clear from the recursive formulation of DFS. Now suppose that v is not an ancestor of w. Note that w is obviously also not an ancestor of v. Thus v lives in a subtree that is completely explored before the subtree of w is visited by `recursiveDFSvisit`. □

```
algorithm DFS
    Input: digraph G
begin
    stack S
    array colour[0..n−1], pred[0..n−1], seen[0..n−1], done[0..n−1]
    for u ∈ V(G) do
        colour[u] ← WHITE; pred[u] ← NULL
    end for
    time ← 0
    for s ∈ V(G) do
        if colour[s] = WHITE then
            DFSvisit(s)
        end if
    end for
    return pred, seen, done
end

algorithm DFSvisit
    Input: node s
begin
    colour[s] ← GREY
    seen[s] ← time; time ← time + 1
    S.insert(s)
    while not S.isEmpty() do
        u ← S.peek()
        if there is a neighbour v with colour[v] = WHITE then
            colour[v] ← GREY; pred[v] ← u
            seen[v] ← time; time ← time + 1
            S.insert(v)
        else
            S.delete()
            colour[u] ← BLACK
            done[u] ← time; time ← time + 1
        end if
    end while
end
```

Figure 5.7: Depth-first search algorithm.

```
algorithm recursiveDFSvisit
    Input: node s
begin
    colour[s] ← GREY
    seen[s] ← time; time ← time + 1
    for each v adjacent to s do
        if colour[v] = WHITE then
            pred[v] ← s
            recursiveDFSvisit(v)
        end if
    end for
    colour[s] ← BLACK
    done[s] ← time; time ← time + 1
end
```

Figure 5.8: Recursive DFS visit algorithm.

All four types of arcs in our search forest classification can arise with DFS. The different types of non-tree arcs can be easily distinguished while the algorithm is running. For example, if an arc (u,v) is explored and v is found to be white, then the arc is a tree arc; if v is grey then the arc is a back arc, and so on (see Exercise 5.3.3). We can also perform the classification after the algorithm has terminated, just by looking at the timestamps *seen* and *done* (see Exercise 5.3.4).

Exercises

Exercise 5.3.1. Give examples to show that all four types of arcs can arise when DFS is run on a digraph.

Exercise 5.3.2. Execute depth-first search on the digraph with adjacency lists representation given below. Classify each arc as tree, forward, back or cross.

0:	2		
1:	0		
2:	0	1	
3:	4	5	6
4:	5		
5:	3	4	6
6:	1	2	

Exercise 5.3.3. Explain how to determine, at the time when an arc is first explored by DFS, whether it is a cross arc or a forward arc.

Exercise 5.3.4. Suppose that we have performed DFS on a digraph G. Let $(v,w) \in E(G)$. Show that the following statements are true.

- (v,w) is a tree or forward arc if and only if

$$seen[v] < seen[w] < done[w] < done[v];$$

- (v,w) is a back arc if and only if

$$seen[w] < seen[v] < done[v] < done[w];$$

- (v,w) is a cross arc if and only if

$$seen[w] < done[w] < seen[v] < done[v].$$

Exercise 5.3.5.

Suppose that DFS is run on a digraph G and the following timestamps obtained.

v	0	1	2	3	4	5	6
$seen[v]$	0	1	2	11	4	3	6
$done[v]$	13	10	9	12	5	8	7

- List all tree arcs in the DFS forest.
- Suppose that $(6,1)$ is an arc of G. Which type of arc (tree, forward, back or cross) is it?
- Is it possible for node 2 to be an ancestor of node 3 in the DFS forest?
- Is it possible that G contains an arc $(5,3)$? If so, what type of arc must it be?
- Is it possible that G contains an arc $(1,5)$? If so, what type of arc must it be?

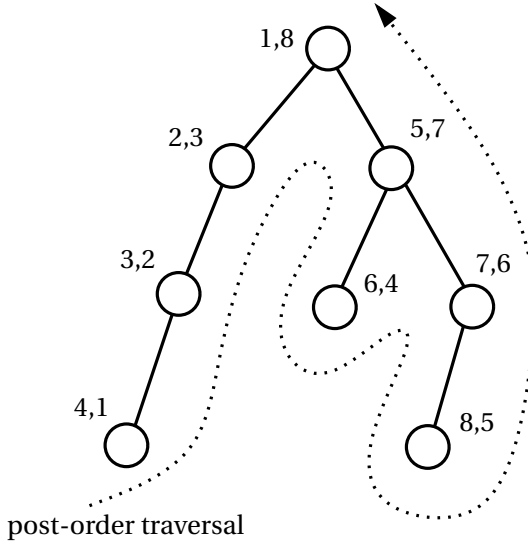
Exercise 5.3.6. Is there a way to distinguish tree arcs from non-tree arcs just by looking at timestamps after DFS has finished running?

Exercise 5.3.7. Suppose that DFS is run on a graph G. Prove that cross edges do not occur.

Exercise 5.3.8. Give an example to show that the following conjecture is *not true*: if w is reachable from v and $seen[v] < seen[w]$ then w is a descendant of v in the DFS forest.

Exercise 5.3.9. DFS allows us to give a so-called pre-order and post-order labelling to a digraph. The pre-order label indicates the order in which the nodes were turned grey. The post-order label indicates the order in which the nodes were turned black.

For example, each node of the following tree is labelled with a pair of integers indicating the pre- and post- orders, respectively, of the layout.



This is obviously strongly related to the values in the arrays *seen* and *done*. What is the exact relationship between the two?

Exercise 5.3.10. Prove Theorem 5.4 by using induction.

5.4 Additional properties of breadth-first search

The first-in first-out processing of the grey nodes in BFS is ideally handled by a queue. In Figure 5.9 we present the algorithm. The timestamps *seen*, *done* of DFS are of less use here; it is more useful to record the number of steps from the root in the array d.

A similar analysis to what we did for DFS also holds for BFS: it is also a linear time traversal algorithm, because the next grey and white node can again be chosen in constant time.

It is rather obvious that BFS processes all nodes at distance 1, then all nodes at distance 2, etc, from the root. The formal proof is below.

Theorem 5.6. Suppose we run `BFS` on a digraph G. Let $v \in V(G)$, and let r be the root of the search tree containing v. Then $d[v] = d(r,v)$.

Proof. Note that since $d[v]$ is the length of a path of tree arcs from r to v, we have $d[v] \geq d(r,v)$. We prove the result by induction on the distance. Denote the BFS search

```
algorithm BFS
    Input: digraph G
begin
    queue Q
    array colour[0..n−1], pred[0..n−1], d[0..n−1]
    for u ∈ V(G) do
        colour[u] ← WHITE; pred[u] ← NULL
    end for
    for s ∈ V(G) do
        if colour[s] = WHITE then
            BFSvisit(s)
        end if
    end for
    return pred, d
end

algorithm BFSvisit
    Input: node s
begin
    colour[s] ← GREY; d[s] ← 0
    Q.insert(s)
    while not Q.isEmpty() do
        u ← Q.peek()
        for each v adjacent to u do
            if colour[v] = WHITE then
                colour[v] ← GREY; pred[v] ← u; d[v] ← d[u] + 1
                Q.insert(v)
            end if
        end for
        Q.delete()
        colour[u] ← BLACK
    end while
end
```

Figure 5.9: Breadth-first search algorithm.

forest by F and let s be the root of a tree in F. Then $d[s] = 0 = d(s,s)$ so the result is true for distance zero. Suppose it is true for all v for which $d(s,v) < k$ and consider a node v such that $d(s,v) = k \geq 1$. Choose a shortest path from s to v in G and let u be the penultimate node in the path. Then $d(s,u) = k-1$ (it cannot be less, or it would contradict $d(s,v) = k$; on the other hand the subpath from s to u must be a shortest path from s to u, otherwise we could find a shorter one from s to v). By the inductive hypothesis, $d[u] = d(s,u) = k-1$. Now v must be seen after u (otherwise $d[v] < k$, but we know $d[v] \geq d(s,v) = k$). Thus v is seen in the loop through white neighbours of u, and so $d[v] = d[u]+1 = k$. □

We can classify arcs, but the answer is not as nice as with DFS.

Theorem 5.7. Suppose that we are performing `BFS` on a digraph G. Let $(v,w) \in E(G)$ and suppose that we have just chosen the grey node v. Then

- if (v,w) is a tree arc then $colour[w] = \text{WHITE}$, $d[w] = d[v]+1$
- if (v,w) is a back arc, then $colour[w] = \text{BLACK}$, $d[w] \leq d[v]-1$
- There are no forward arcs.
- if (v,w) is a cross arc then one of the following holds:
  - $d[w] < d[v]-1$, and $colour[w] = \text{BLACK}$;
  - $d[w] = d[v]$, and $colour[w] = \text{GREY}$;
  - $d[w] = d[v]$, and $colour[w] = \text{BLACK}$;
  - $d[w] = d[v]-1$, and $colour[w] = \text{GREY}$;
  - $d[w] = d[v]-1$, and $colour[w] = \text{BLACK}$.

Proof. The arc is added to the tree if and only if w is white. If the arc is a back arc, then w is an ancestor of v; the FIFO queue structure means w is black before the adjacency list of v is scanned.

Now suppose that (x,u) is a forward arc. Then since u is a descendant of x but not a child in the search forest, Theorem 5.6 yields $d[u] \geq d[x]+2$. But by the last theorem we have $d[u] = d(s,u) \leq d(s,x)+1 = d[x]+1$, a contradiction. Hence no such arc exists.

A cross arc may join two nodes on the same level, jump up one level, or jump up more than one level. In the last case, w is already black before v is seen. In the second case, w may be seen before v, in which case it is black before v is seen (recall w is not the parent of v), or it may be seen after v, in which case it is grey when (v,w) is explored. In the first case, w may be seen before v (in which case it is black before v is seen), or w may be seen after v (in which case it is grey when (v,w) is explored). □

In the special case of graphs we can say more.

Theorem 5.8. Suppose that we have performed BFS on a graph G. Let $\{v,w\} \in E(G)$. Then exactly one of the following conditions holds.

- $\{v,w\}$ is a tree edge, $|d[w] - d[v]| = 1$;
- $\{v,w\}$ is a cross edge, $d[w] = d[v]$;
- $\{v,w\}$ is a cross edge, $|d[w] - d[v]| = 1$.

Proof. By Theorem 5.7 there can be no forward edges, hence no back edges. A cross edge may not jump up more than one level, else it would also jump down more than one level, which is impossible by Theorem 5.6. □

For a given BFS tree, we can uniquely label the vertices of a digraph based on the time they were first seen. For the graph G_1 of Figure 5.4, we label vertex 0 with 1, vertices $\{1,2\}$ with labels $\{2,3\}$, vertices $\{3,4,8\}$ with labels $\{4,5,6\}$, and the last vertex level $\{5,6,7\}$ with labels $\{7,8,9\}$. These are indicated in Figure 5.5.

Exercises

Exercise 5.4.1. Carry out BFS on the digraph with adjacency list given below. Show the state of the queue after each change in its state.

0:	2		
1:	0		
2:	0	1	
3:	4	5	6
4:	5		
5:	3	4	6
6:	1	2	

Exercise 5.4.2. How can we distinguish between a back and a cross arc while BFS is running on a digraph?

Exercise 5.4.3. Explain how to determine whether the root of a BFS tree is contained in a cycle, while the algorithm is running. You should find a cycle of minimum length if it exists.

5.5 Priority-first search

Priority-first search is a more sophisticated form of traversal with many applications. For now, we consider it simply as a common generalization of breadth-first and depth-first search. Priority-first search may seem a little abstract compared to the more concrete DFS and BFS. We shall not need it until Chapter 6; however it will be essential then.

The important property is that each grey node has associated with it an integer ***key***. The interpretation of the key is of a priority: the smaller the key, the higher the priority. The rule for selecting a new grey node is to choose one with the smallest key.

In the simplest form of PFS, the key value is assigned when the node becomes grey, and never updated subsequently. More generally, the key may be further updated at other times. We shall see both types in this book. The second type of PFS is used in optimization problems as we shall discuss in Chapter 6.

The first type of PFS includes both BFS and DFS. In BFS, the key value of v can be taken as the time v was first coloured grey. Note that this means that a given grey node can be selected many times—until it becomes black, in fact, it will always have minimum key among the grey nodes. By contrast, in DFS we can take the key value to be $-seen[v]$. Then the last node seen always has minimum key. It cannot be chosen again until the nodes seen after it have become black.

The running time of PFS depends mostly on how long it takes to find the minimum key value, and how long it takes to update the key values.

In the array implementation mentioned above, finding the minimum key value takes time of order n at each step, so the quantity a is $\Omega(n)$. Thus a PFS of this type will take time in $\Omega(n^2)$. This is worse than the $\Theta(n+m)$ we obtain with BFS and DFS using adjacency lists and a queue or stack respectively. One reason is that a simple array is not a particularly good data structure for finding the minimum key. You have already seen a better one in Part I of this book—the binary heap. In fact PFS is best described via the priority queue ADT (see Section C.1).

Pseudocode demonstrating the first type of PFS is presented in Figure 5.10. The subroutine `setKey` there is the rule for giving the key value when a node is inserted. We do not include any code for `setKey`.

We proceed to show some applications of BFS and DFS in the following sections. Applications of PFS will be discussed in Chapter 6.

5.6 Acyclic digraphs and topological ordering

In this section we show how to arrange, when possible, the nodes of a digraph into a topological or precedence order. Many computer science applications require us to find precedences (or dependencies) among events, such as a compiler evaluating sub-expressions of an expression like that shown in Figure 5.11.

Here the compiler would need to compute, for example, both $(a+b)$ and c before it can compute $c-(a+b)$.

Definition 5.9. Let G be a digraph. A ***topological sort*** of G is a linear ordering of all its vertices such that if G contains an arc (u,v), then u appears before v in the ordering.

The term topological sort comes from the study of partial orders and is sometimes called a ***topological order*** or ***linear order***. If a topological sort exists, then it is

```
algorithm PFS
    Input: digraph G
begin
    priority queue Q
    array colour[0..n−1], pred[0..n−1]
    for u ∈ V(G) do
        colour[u] ← WHITE; pred[u] ← NULL
    end for
    for s ∈ V(G) do
        if colour[s] = WHITE then
            PFSvisit(s)
        end if
    end for
    return pred
end

algorithm PFSvisit
    Input: node s
begin
    colour[s] ← GREY
    Q.insert(s, setKey (s))
    while not Q.isEmpty() do
        u ← Q.peek()
        if u has a neighbour v with colour[v] = WHITE then
            colour[v] ← GREY
            Q.insert(v, setKey (v))
        else
            Q.delete()
            colour[u] ← BLACK
        end if
    end while  end
```

Figure 5.10: Priority-first search algorithm (first kind).

(a+b)*(c-(a+b))*(-c+d)

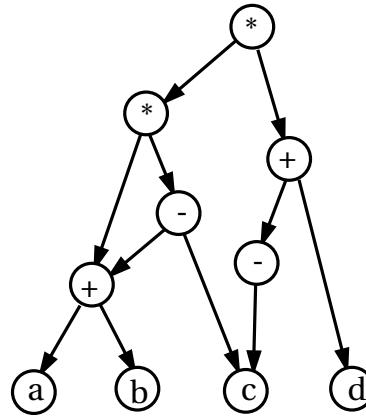


Figure 5.11: Digraph describing structure of an arithmetic expression.

possible to draw a picture of G with all nodes in a straight line, and the arcs "pointing the same way".

A digraph without cycles is commonly called a ***DAG***, an abbreviation for **d**irected **a**cyclic **g**raph. It is much easier for a digraph to be a DAG than for its underlying graph to be acyclic.

For our arithmetic expression example above, a linear order of the sub-expression DAG gives us an order (actually the reverse of the order) where we can safely evaluate the expression.

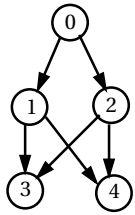
Clearly if the digraph contains a cycle, it is not possible to find such a linear ordering. This corresponds to inconsistencies in the precedences given, and no scheduling of the tasks is possible.

Example 5.10. In Figure 5.12 we list three DAGs and possible topological orders for each. Note that adding more arcs to a DAG reduces the number of topological orders it has. This is because each arc (u,v) forces u to occur before v, which restricts the number of valid permutations of the vertices.

The algorithms for computing *all* topological orders are more advanced than what we have time or space for here. We show how to compute one such order, however.

First we note that *if* a topological sort of a DAG G is possible, then there must be a source node in G. The source node can be first in a topological order, and no node that is not a source can be first (because it has an in-neighbour that must precede it in the topological order).

Theorem 5.11. A digraph has a topological order if and only if it is a DAG.

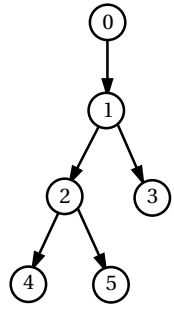


0,1,2,3,4

0,1,2,4,3

0,2,1,3,4

0,2,1,4,3



0,1,3,2,4,5

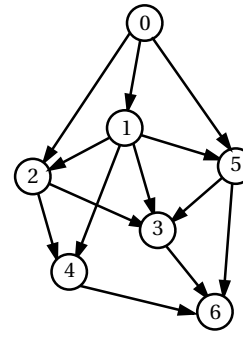
0,1,3,2,5,4

0,1,2,3,4,5

0,1,2,3,5,4

⋮

0,1,2,5,4,3



0,1,2,4,5,3,6

0,1,2,5,3,4,6

0,1,2,5,4,3,6

0,1,5,2,3,4,6

0,1,5,2,4,3,6

Figure 5.12: Topological orders of some DAGs.

Proof. First show that every DAG has a source (see exercise 5.6.2). Given this, we proceed as follows. Deleting a source node creates a digraph that is still a DAG, because deleting a node and some arcs cannot create a cycle where there was none previously. Repeatedly doing this gives a topological order. □

This theorem then gives an algorithm (***zero-indegree sorting***) for topologically sorting a DAG G.

So far we have not shown how to determine whether a given digraph is a DAG. But if we apply zero-indegree sorting to a digraph that is not a DAG, eventually it will stop because no source node can be found at some point (otherwise we would obtain a topological order and hence the digraph would be a DAG).

There is another way to determine acyclicity and topologically sort a DAG, based on DFS. If G contains a cycle, DFS must eventually reach a node that points to one that has been seen before. In other words, we will detect a back arc. The details and proof that this works now follow.

Theorem 5.12. Suppose that DFS is run on a digraph G. Then G is acyclic if and only if there are no back arcs.

Proof. Suppose that we run DFS on G. Note that if there is a back arc (v,u), then u and v belong to the same tree T, with root s say. Then there is a path from s to u, and there is a path from u to v by definition of back arc. Adding the arc (v,u) gives a cycle.

Conversely, if there is a cycle $v_0 v_1 \ldots v_n v_0$, we may suppose without loss of generality that v_0 is the first node of the cycle visited by the DFS algorithm. We claim that (v_n, v_0) is a back arc. To see why this is true, first note that during the DFS v_0 is linked to v_n via a path of unvisited nodes (possibly of length shorter than n). We have v_n as a descendant of v_0 in the DFS tree and a back arc (v_n, v_0). □

One valid topological order is simply the reverse of the DFS finishing times.

Theorem 5.13. Let G be a DAG. Then listing the nodes in reverse order of DFS finishing times yields a topological order of G.

Proof. Consider any arc $(u,v) \in E(G)$. Since G is a DAG, the arc is not a back arc by Theorem 5.12. In the other three cases, Exercise 5.3.4 shows that $done[u] > done[v]$, which means u comes before v in the alleged topological order. □

We can therefore just run DFS on G, and stop if we find a back arc. Otherwise printing the nodes in reverse order of finishing time gives a topological order. Note that printing the nodes in order of finishing time gives a topological order of the reverse digraph G_r.

Exercises

Exercise 5.6.1. Give an example of a DAG whose underlying graph contains a cycle. Make your example as small as possible.

Exercise 5.6.2. Prove that every DAG must have at least one source and at least one sink.

Exercise 5.6.3. Show that the following method for topologically sorting a DAG does not work in general: print the nodes in order of visiting time.

Exercise 5.6.4. Professor P has the following information taped to his mirror, to help him to get dressed in the morning.

Socks before shoes; underwear before trousers; trousers before belt; trousers before shoes; shirt before glasses; shirt before tie; tie before jacket; shirt before hat; shirt before belt.

Find an acceptable order of dressing for Professor P.

Exercise 5.6.5. What is the time complexity of zero-indegree sorting?

Exercise 5.6.6. Let G be a graph. There is an easy way to show that G is acyclic. It is not hard to show (see Section D.7) that a graph G is acyclic if and only if G is a forest, that is, a union of (free) trees.

Give a simple way to check whether a graph G is acyclic. Does the method for finding a DAG given above work for acyclic graphs also?

5.7 Connectivity

For many purposes it is useful to know whether a digraph is "all in one piece", and if not, to decompose it into pieces. We now formalize these notions. The situation for graphs is easier than that for digraphs.

Definition 5.14. A graph is ***connected*** if for each pair of vertices $u, v \in V(G)$, there is a path between them.

In Example 4.3 the graph G_1 is connected, as is the underlying graph of G_2.

If a graph is not connected, then it must have more than one "piece". More formally, we have the following.

Theorem 5.15. Let G be a graph. Then G can be uniquely written as a union of subgraphs G_i with the following properties:

- each G_i is connected
- if $i \neq j$, there are no edges from any vertices in G_i to any vertices in G_j

Proof. Consider the relation $\sim$ defined on $V(G)$, given by $u \sim v$ if and only if there is a path joining u and v (in other words, u and v are each reachable from the other). Then $\sim$ is an equivalence relation and so induces a partition of $V(G)$ into disjoint subsets. The subgraphs G_i induced by these subsets have no edges joining them by definition of $\sim$, and each is connected by definition of $\sim$. □

The subgraphs G_i above are called the ***connected components*** of the graph G. Clearly, a graph is connected if and only if it has exactly one connected component.

Example 5.16. The graph obtained by deleting two edges from a triangle has 2 connected components.

We can determine the connected components of a graph easily by using a traversal algorithm. The following obvious theorem explains why.

Theorem 5.17. Let G be a graph and suppose that DFS or BFS is run on G. Then the connected components of G are precisely the subgraphs spanned by the trees in the search forest.

Proof. The result is true for any traversal procedure, as we have already observed in Theorem 5.2. The trees of the search forest have no edges joining them, and together they span G. □

So we need only run BFS or DFS on the graph, and keep count of the number of times we choose a root—this is the number of components. We can store or print the vertices and edges in each component as we explore them. Clearly, this gives a linear time algorithm for determining the components of a graph.

So far it may seem that we have been too detailed in our treatment of connectedness. After all the above results are all rather obvious. However, now consider the situation for digraphs. The intuition of "being all in one piece" is not as useful here. In Example 4.3 the graph G_1 is connected, as is the underlying graph of G_2. They are "all in one piece", but not the same from the point of view of reachability. For example, in digraph G_2, node 2 is a sink. This motivates the following definition.

Definition 5.18. A digraph G is ***strongly connected*** if for each pair of nodes u, v of G, there is a path in G from u to v.

Note. In other words, u and v are each reachable from the other.

Suppose that the underlying graph of G is connected (some authors call this being ***weakly connected***), but G is not strongly connected. Then if G represents a road network, it is possible to get from any place to any other one, but at least one such route will be illegal: one must go the wrong way down a one-way street.

A strongly connected digraph must contain many cycles: indeed, if v and w are different nodes, then there is a path from v to w and a path from w to v, so v and w are contained in a cycle. Conversely, if each pair of nodes is contained in a cycle, then the digraph is clearly strongly connected.

Again, we can define ***strongly connected components*** in a way that is entirely analogous to component for graphs. The proof above for connected components generalizes to this situation.

Theorem 5.19. Let $G = (V, E)$ be a digraph. Then V can be uniquely written as a union of disjoint subsets V_i, with each corresponding induced subdigraph G_i being a strongly connected component of G.

Proof. Consider the relation $\sim$ defined on V, given by $u \sim v$ if and only if there is a path joining u and v and a path joining v and u (in other words, u and v are each reachable from the other). Then $\sim$ is an equivalence relation and so induces a partition of V into disjoint subsets. By definition, each subdigraph G_i is strongly connected and of maximal order. □

Example 5.20. A digraph and its three (uniquely determined) strongly connected components are displayed in Figure 5.13. Note that there are arcs of the digraph not included in the strongly connected components.

Note that if the underlying graph of G is connected but G is not strongly connected, then there are strong components C_1 and C_2 such that it is possible to get from C_1 to C_2 but not from C_2 to C_1. If C_1 and C_2 are different strong components, then any arcs between them must either all point from C_1 to C_2 or from C_2 to C_1. Suppose that we imagine each strong component shrunk to a single node (so we ignore the internal structure of each component, but keep the arcs between components).

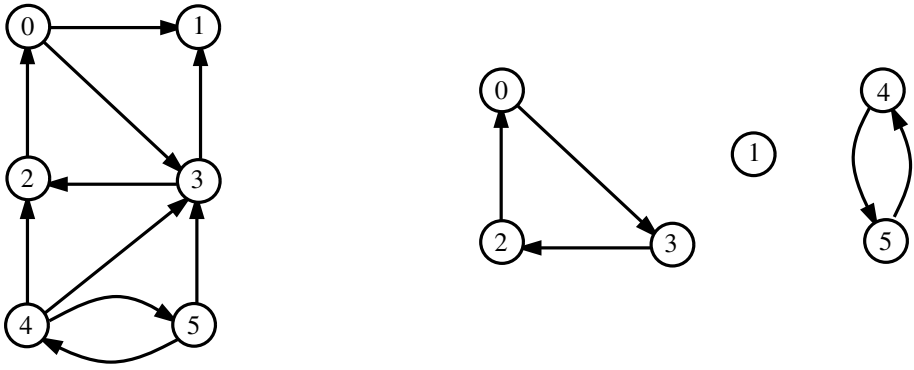


Figure 5.13: A digraph and its strongly connected components.

Then in the digraph resulting, if $v \neq w$ and we can get from v to w then we cannot get from w to v. In other words, no pair of nodes can belong to a cycle, and hence the digraph is acyclic. See Figure 5.14. Note that the converse is also true: if we have an acyclic digraph G and replace each node by a strongly connected digraph, the strongly connected components of the resulting digraph are exactly those digraphs that we inserted.

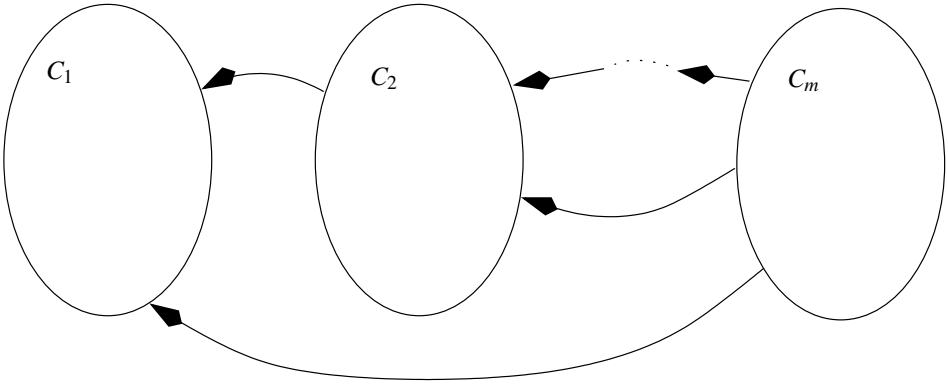


Figure 5.14: Structure of a digraph in terms of its strong components.

Note the similarity between this and the search forest decomposition in Figure 5.3. In that case, if we shrink each search tree to a point, the resulting digraph is also acyclic.

How to determine the strongly connected components? First we observe that the previous method for graphs definitely fails (see Exercise 5.7.1). To decide whether a digraph is strongly connected we could run `BFSvisit` or `DFSvisit` originating from

each node in turn and see whether each of the n trees so generated spans the digraph. However the running time of such an algorithm is $\Theta(n^2+nm)$.

We can do better by using DFS more cleverly.

Consider the reverse G_r. The strong components of G_r are the same as those of G. Shrinking each strong component to a point, we obtain acyclic digraphs H and H_r. Consider a sink S_1 in H_r. If we run DFS on G_r starting in the strong component S_1, we will reach every node in that component *and no other nodes of* G_r. The DFS tree will exactly span S_1. Now choose the next root to lie in the strong component S_2 node of H_r whose only possible out-neighbour is S_1 (this is possible by the same reasoning used for zero-indegree sort, except here we deal with outdegree). The DFS will visit all of S_2 and no other nodes of G_r because all other possible nodes have already been visited. Proceed in this way until we have visited all strong components.

We have shown that if we can choose the roots correctly, then we can find all strong components. Now of course we don't know these components *a priori*, so how do we identify the roots to choose?

Whenever there is a choice for the root of a new search tree, it must correspond to a new node of the DAG H_r. We want at least a reverse topological order of H_r. This is simply a topological order for H. Note that in the case where $H = G$ (each strong component has just one point), then G is a DAG. The method above will work if and only if we choose the roots so that each tree in the DFS for G_r has only one point. We just need a topological order for G, so run DFS on G and print the nodes in reverse order of finishing time. Then choose the roots for the DFS on G_r in the printed order.

It therefore seems reasonable to begin with a DFS of G. Furthermore, an obvious choice is: in the DFS of G_r, *choose each new root from among white nodes that finished latest in* F.

Then each DFS tree in the search of G_r definitely contains the strong component S of the root r. To see this, note that no node in that strong component could have been visited before in G_r, otherwise r would have already been visited. By Theorem 5.4, every node in the strong component of r is a descendant of r.

The only thing that could go wrong is that a search tree in G_r might contain more than one strong component. This cannot happen, as we now prove.

Theorem 5.21. If the following rule for choosing roots is used in the algorithm described above, then each tree in the second search forest spans a strong component of G, and all strong components arise this way.

Rule: use the white node whose finishing time in F was largest.

Proof. Suppose that a search tree in G_r does contain more than one strong component. Let S_1 be the first strong component seen in G_r and let S_2 be another, and let the roots be r,s respectively. Note that by the rule for choosing nodes r was the first node of S_1 seen in F (by Theorem 5.4, every node of S_1 is a descendant of the first one seen, which therefore has latest finishing time). The analogous statement holds for s and S_2.

By the rule for choosing roots, we have $done[r] > done[s]$ in F. We cannot have $seen[s] > seen[r]$ in F, for then s would be a descendant of r in F and in G_r, so they would belong to the same strong component. Thus $seen[r] > seen[s]$ in F. Hence S_2 was explored in F before r (and hence any node of S_1) was seen. But then r would have been reachable from s in G via a path of white nodes, so Theorem 5.4 shows that r and s are in the same strong component, another contradiction. □

The above algorithm runs in linear time with adjacency lists, since each DFS and the creation of the reverse digraph take linear time. We only need to remember while performing the first DFS to store the nodes in an array in order of finishing time.

Exercises

Exercise 5.7.1. Give an example to show that a single use of DFS does not in general find the strongly connected components of a digraph.

Exercise 5.7.2. Carry out the above algorithm by hand on the digraph of Example 5.20 and verify that the components given there are correct. Then run it again on the reverse digraph and verify that the answers are the same.

5.8 Cycles

In this section, we cover three varied topics concerning cycles.

The girth of a graph

The length of the smallest cycle in a graph is an important quantity. For example, in a communication network, short cycles are often something to be avoided because they can slow down message propagation.

Definition 5.22. For a graph (with a cycle), the length of the shortest cycle is called the ***girth*** of the graph. If the graph has no cycles then the girth is undefined but may be viewed as $+\infty$.

Note. For a digraph we use the term girth for its underlying graph and the (maybe non-standard) term ***directed girth*** for the length of the smallest directed cycle.

Example 5.23. In Figure 5.15 are three (di)graphs. The first has no cycles (it is a free tree), the second is a DAG of girth 3, and the third has girth 4.

How to compute the girth of a graph? Here is an algorithm for finding the length of a shortest cycle containing a given vertex v in a graph G. Perform `BFSvisit`. If we meet a grey neighbour (that is, we are exploring edge $\{x,y\}$ from x and we find that y is already grey), continue only to the end of the current level and then stop. For each edge $\{x,y\}$ as above on this level, if v is the lowest common ancestor of x and y in the BFS tree, then there is a cycle containing x,y,v of length $l = d(x)+d(y)+1$. Report the minimum value of l obtained along the current level.

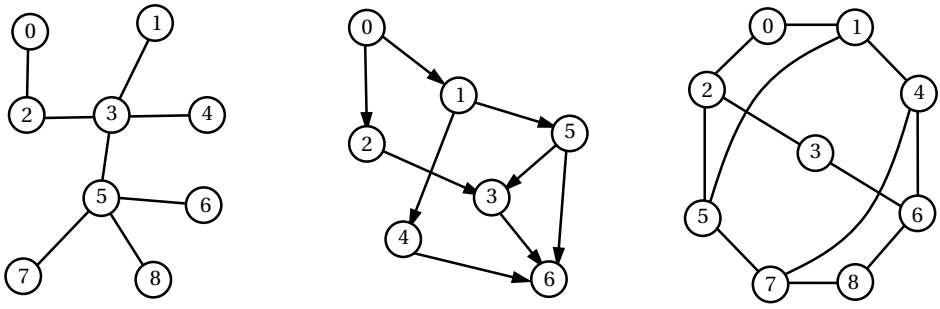


Figure 5.15: Some (di)graphs with different cycle behaviour.

Theorem 5.24. The above algorithm is correct.

Proof. Suppose that we arrive at vertex x, $d(x) = k$, and we have just encountered a grey neighbour y. Then $d[y] = d[x] + 1$ or $d[y] = d[x]$ (note that $d[x] = d[y] + 1$ is ruled out because then x would be a child of y and y would be black). This means that there is definitely a cycle containing x, y and z, where z is the lowest common ancestor of x and y in the BFS tree. Note that $z \neq x, z \neq y$ since neither x nor y is an ancestor of the other. The cycle consists of the tree edges from z to x, the cross edge $\{x, y\}$ and the tree edges from z to y. The length of the cycle is $l = d[x] + d[y] - 2d[z] + 1$.

Conversely, let C be any cycle and let z be the first vertex of the cycle reached by the search. Let x and y be two vertices in C whose distance from z is greatest, with $d[x] \leq d[y]$. Then the lowest common ancestor of x and y is exactly z, and the cycle found in the first paragraph is exactly C.

The vertex v belongs to the cycle C if and only if $v = z$. The length of the cycle is $2k+1$ if $d[x] = d[y]$ and $2k+2$ if $d[y] = d[x] - 1$. The minimum length of any cycle containing v found after this is $2(k+1)+1 = 2k+3$, so no better cycle can be found after the current level k is explored. □

Note. An easy-to-implement DFS idea may not work properly. For example, the DFS tree originating from vertex 0 of the third graph of Figure 5.15 finds only 6-cycles with the back edges (even though a 4-cycle exists using two of the back edges).

To compute the girth of a graph, we can simply run the above algorithm from each vertex in turn, and take the minimum cycle length achieved.

Bipartite graphs

Many graphs in applications have two different types of nodes, and no relations between nodes of the same type (this is a model of sexually reproducing organisms, for example).

Definition 5.25. A graph G is ***bipartite*** if $V(G)$ can be partitioned into two nonempty disjoint subsets V_0, V_1 such that each edge of G has one endpoint in V_0 and one in V_1.

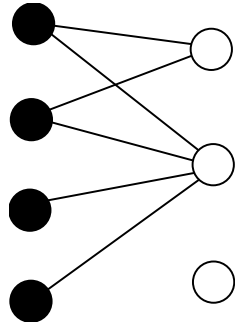


Figure 5.16: A bipartite graph.

Example 5.26. The graph in Figure 5.16 is bipartite. The isolated vertex could be placed on either side.

Showing that a graph is bipartite can be done by exhibiting a ***bipartition*** (a partition into two subsets as in the definition). Of course finding such a bipartition may not be easy. Showing a graph is not bipartite seems even harder. In each case, we certainly do not want to have to test all possible partitions of V into two subsets! There is a better way.

Definition 5.27. Let k be a positive integer. A graph G has a ***k-colouring*** if $V(G)$ can be partitioned into k nonempty disjoint subsets such that each edge of G joins two vertices in different subsets.

Example 5.28. The graph in Figure 5.16 has a 2-colouring as indicated.

It is not just a coincidence that our example of a bipartite graph has a 2-colouring.

Theorem 5.29. The following conditions on a graph G are equivalent.

- G is bipartite;
- G has a 2-colouring;
- G does not contain an odd length cycle.

Proof. Given a bipartition, use the same subsets to get a 2-colouring, and vice versa. This shows the equivalence of the first two conditions. Now suppose G is bipartite. A cycle must have even length, since the start and end vertices must have the same colour. Finally suppose that G has no odd length cycle. A 2-colouring is obtained as follows. Perform BFS and assign each vertex at level i the "colour" $i \bmod 2$. If we can complete this procedure, then by definition each vertex goes from a vertex of one colour to one of another colour. The only problem could be if we tried to assign a

colour to a node v that was adjacent to a node w of the same colour at the same level k. But then a cycle of length $2k+1$ is created. □

It is now easy to see that we may use the method in the proof above to detect an odd length cycle if it exists, and otherwise produce a 2-colouring of G. This of course runs in linear time.

Exercises

Exercise 5.8.1. Give an example to show that in the shortest cycle algorithm, if we do not continue to the end of the level, but terminate when the first cycle is found, we may find a cycle whose length is one more than the shortest possible.

Exercise 5.8.2. What is the time complexity of the shortest cycle algorithm?

Exercise 5.8.3. The ***n-cube*** (***hypercube***) is a graph on 2^n vertices, each labelled by a different bit vector of length n. If $\mathbf{v} = (v_0, \ldots, v_{n-1})$ and $\mathbf{w} = (w_0, \ldots, w_{n-1})$ are such bit vectors, then there is an edge between the vertex labelled $\mathbf{v}$ and a vertex labelled $\mathbf{w}$ if and only if $\mathbf{v}$ and $\mathbf{w}$ differ in exactly one component. For which values of n is the n-cube bipartite?

5.9 Maximum matchings

Next we want to introduce an important graph problem that can be solved in polynomial time by a clever path augmentation algorithm.

Definition 5.30. A ***matching*** in a graph is a set of pairwise non-adjacent edges (that is, each vertex can be in at most one edge of the matching). A ***maximal matching*** is a matching such that is not a proper subset of any other matching. A ***maximum matching*** is one with the largest possible number of edges (over all possible matchings).

Often, for many real-world problems, we want to find a maximum matching in bipartite graphs as illustrated by the next two examples.

Example 5.31. Suppose we have a set of workers V_0 and a set of tasks V_1 that need to be assigned. A given worker of V_0 is able to perform a subset of the tasks in V_1. Now with each worker capable of doing at most one task at a time, the boss would like to assign (match) as many workers as possible to as many of the tasks.

Example 5.32. Consider the ***marriage problem*** where we have a set of men and women (as vertices) and edges representing compatible relationships. The goal is to marry as many couples as possible, which is the same as finding a maximum matching in the relationship graph. If there are no homosexual interests then we have a bipartite graph problem.

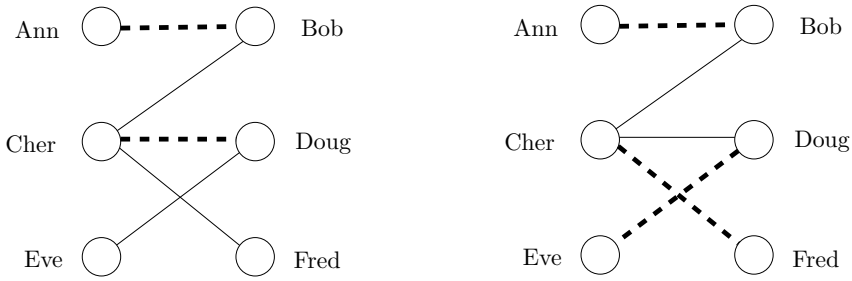


Figure 5.17: A maximal and maximum matching in a bipartite graph.

In Figure 5.17 we illustrate the difference between a maximal and maximum matching in the setting of Example 5.32. The matchings consist of bold-dashed edges (between females on the left and males on the right) in the drawings.

It is easy to find a maximal matching M in a graph. For example, a simple greedy approach of iterating over all edges and adding each edge to M if it is non-adjacent to anything already in M will work. As illustrated in Figure 5.17, a maximal matching may have fewer edges than a more desirable maximum matching.

Our algorithm to compute a maximum matching will be based on trying to improve an existing matching M by finding certain types of paths in a graph.

Definition 5.33. Given a matching M, an ***alternating path*** is a path in which the edges of the path alternate from being in the matching and not. An ***augmenting path*** is an alternating path that starts from and ends on unmatched vertices.

For example, consider the augmenting path Eve–Doug–Cher–Fred of Figure 5.17. It contains three edges but only the middle edge is part of a matching on the left case. We can get the better matching on the right if we add Eve-Doug, remove Doug–Cher, and add Cher–Fred to the existing matching. Thus, in general, we see that we can improve a matching if we can find an augmenting path. Note that there is always one more non-matching edge than matching edge in an augmenting path. Likewise, it is pretty easy to show that if there is no such augmenting path then we must have a maximum matching (see Exercise 5.9.1).

We next present a polynomal-time algorithm that finds an augmenting path if one exists. The basic idea is to start from an unmatched vertex v and build a tree (via a graph traversal algorithm such as BFS) of alternating paths away from v. If we reach another unmatched vertex then we have found an augmenting path. Otherwise, if we visit all vertices (in the same component as v) then we conclude that no augmenting path exists starting at v. This algorithm is given in Figure 5.18.

Theorem 5.34. There exists a polynomial-time algorithm to find a maximum matching in a bipartite graph.

```
algorithm findAugmentingPath
    Input: bipartite graph G; matching M; unmatched vertex x
begin
    queue Q
    array status[0..n − 1], pred[0..n − 1]
    for each u ∈ V(G) do
        status[u] ← WHITE; pred[u] ← NULL
    end for
    status[x] ← EVEN
    Q.insert(x)
    while not Q.isEmpty() do
        u ← Q.pop()
        if status[u] = EVEN then
            for each v adjacent to u do
                if status[v] = WHITE then
                    status[v] ← ODD; pred[v] ← u
                    if v is unmatched in M then
                        return path x, ..., pred[pred[u]], pred[u], u, v
                    else
                        Q.insert(v)
                    end if
                end if
            end for
        else                          that is, status[u] = ODD
            v ← matched vertex of u from M
            if status[v] = WHITE then
                status[v] ← EVEN; pred[v] ← u
                Q.insert(v)
            end if
        end if
    end while
    return false                      no augmenting paths containing x
end
```

Figure 5.18: An algorithm to find an augmenting path, given a matching and an unmatched starting vertex.

Proof. We first need to show that the algorithm `findAugmentingPath`, given in Figure 5.18, does find an augmenting path if one exists. It is sufficient to show that if there exists at least one augmenting path from vertex x to some other unmatched vertex that we find any one of them (in our case, by imitating BFS, it will be one of shortest length). `findAugmentingPath` builds a traversal tree starting at x using the following constraints.

- If a reachable vertex u is in the same partition as x (status will be set to EVEN by the algorithm) then we know (except for x) that there is an alternating path with the last edge including u being in the matching.
- If a reachable vertex v is not in the same partition as x (status will be set to ODD if it is matched) then we know that there is an alternating path with the last edge including v is not in the matching.

This process produces a tree with alternating paths rooted at x as illustrated in Figure 5.19. The status of the nodes are ODD or EVEN depending if they are an even or odd distance from x. If a vertex (first seen) is at an odd distance then we have seen an alternating path where the last edge is not in the matching. If this last vertex is unmatched then we have found an augmenting path, otherwise we extend the path by using the matched edge. If a vertex is at an even distance then we have seen an alternating path where the last edge is in the matching. Since the graph is bipartite the status of being ODD or EVEN is unambiguous.

Suppose the algorithm `findAugmentingPath` terminates without finding an augmenting path when one does exist. Let $x = v_0, v_1, \ldots, v_k$ be a counterexample. Consider the first index $0 < i \leq k$ such that $status[v_i] =$ WHITE. We know v_{i-1} was inserted in the queue Q. Consider the two cases. If $i-1$ is even then since v_i is a neighbour of v_{i-1} its status would have changed to ODD. If $i-1$ is odd then either (v_{i-1}, v_i) is in the matching or not. If so, the status of v_i would have changed; if not, a prefix of the counterexample is not an augmenting path. Thus, by contradiction, `findAugmentingPath` will find an augmenting path if one exists.

The running time of one invocation of `findAugmentingPath` is the same as the running time of BFS since each vertex is added to the queue Q at most once. For adjacency list representation of graphs this can be carried out in time $O(m)$. If we find an augmenting path then our best matching increases by one. Since a maximum matching is bounded by $\lfloor n/2 \rfloor$ we only need to find at most $O(n)$ augmenting paths. We potentially need to call `findAugmentingPath` once for each unmatched vertex, which is bounded by $O(n)$, and repeat the process for each modified matching. Therefore, the total running time to find a maximum matching is at most $O(n^2m)$. □

The algorithm presented here can easily be improved to $O(nm)$ by noting that it is only required to traverse and compute an "alternating path forest" in order to find an

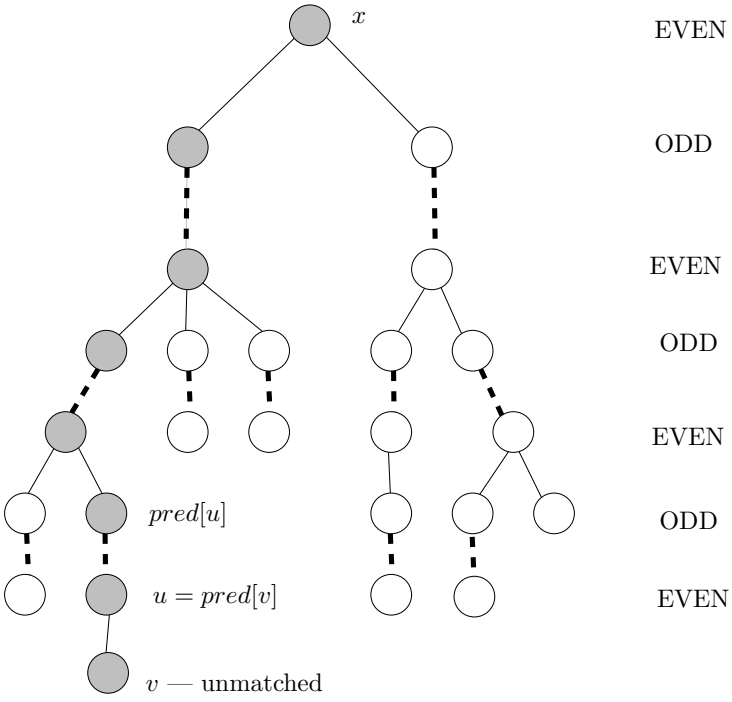


Figure 5.19: Structure of the graph traversal tree for finding augmenting paths.

augmenting path. That is, we do not need to originate a call to `findAugmentingPath` for all unmatched vertices. However the correctness is a bit tricky to justify.

There are other algorithms that find maximum matchings more efficiently than the one presented. One of these is the Hopcroft-Karp algorithm which runs in time $O(m\sqrt{n})$ and is based on finding a maximal flow in a network (by adding a source and sink vertex and directing all edges from one vertex partition to the other of the bipartite graph).

We conclude this section by mentioning that there also exist polynomial-time algorithms to find maximum matchings in arbitrary (non-bipartite) graphs. The details are beyond the scope of this book.

Exercises

Exercise 5.9.1. Prove that a matching for a graph is maximum if and only if it does not have any augmenting paths.

Exercise 5.9.2. Give an example of a bipartite graph of order 12 with a maximal matching that has an augmenting path of length 5 and a maximum matching of the same graph with two more edges.

Exercise 5.9.3. Show that the size of a maximum matching in a bipartite graph $G = (V,E)$ is the same as the size of the smallest ***vertex cover*** of G. A vertex cover is a subset $V' \subseteq V$ of vertices such that for all $(u,v) \in E$, at least one of u or v is in V'. Does the equality hold for arbitrary graphs?

5.10 Notes

The linear-time algorithm for finding strong components of a digraph was introduced by R. E. Tarjan in 1971.

One of the early polynomial-time algorithms for finding maximum matchings in bipartite graphs is based on the Ford–Fulkerson network flow algorithm [7] from 1956. The first polynomial-time algorithm for finding a maximum matching in an arbitrary graph was presented by J. Edmonds [5] in 1965.

Chapter 6

Weighted Digraphs and Optimization Problems

So far our digraphs have only encoded information about connection relations between nodes. For many applications it is important to study not only *whether* one can get from A to B, but *how much it will cost* to do so.

For example, the weight could represent how much it costs to use a link in a communication network, or distance between nodes in a transportation network. We shall use the terminology of cost and distance interchangeably (so, for example, we talk about finding a minimum weight path by choosing a shortest edge).

We need a different ADT for this purpose.

6.1 Weighted digraphs

Definition 6.1. A ***weighted digraph*** is a pair (G,c) where G is a digraph and c is a ***cost function***, that is, a function associating a real number to each arc of G.

We interpret $c(u,v)$ as the cost of using arc (u,v). An ordinary digraph can be thought of as a special type of weighted digraph where the cost of each arc is 1. A weighted graph may be represented as a symmetric digraph where each of a pair of antiparallel arcs has the same weight.

In Figure 6.1 we display a classic unweighted graph (called the 3-cube) of diameter 3, a digraph with arc weights, and a graph with edge weights.

There are two obvious ways to represent a weighted digraph on a computer. One is via a matrix. The adjacency matrix is modified so that each entry of 1 (signifying that an arc exists) is replaced by the cost of that arc. Another is a double adjacency

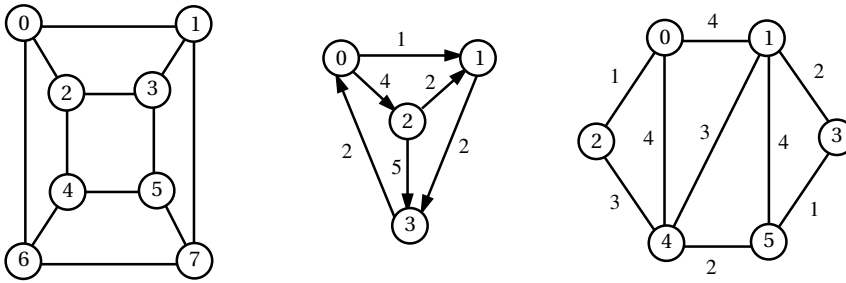


Figure 6.1: Some weighted (di)graphs.

list. In this case, the list associated to a node v contains, alternately, an adjacent node w and then the cost $c(v,w)$.

If there is no arc between u and v, then in an ordinary adjacency matrix the corresponding entry is 0. However, in a weighted adjacency matrix, the "cost" of a nonexistent arc should be set consistantly for most applications. We adopt the following *convention*. An entry of `null` or 0 in a weighted adjacency matrix means that the arc does not exist, and vice versa. In many of our algorithms below, such entries should be replaced by the programming equivalent of `null` for class objects, or ∞ for primitive data types. In the later case, we might use some positive integer greater than any expected value that might occur during an execution of the program.

Example 6.2. The two weighted (di)graphs of Figure 6.1 are stored as weighted adjacency matrices below.

$$\begin{bmatrix} 0 & 1 & 4 & 0 \\ 0 & 0 & 0 & 2 \\ 0 & 2 & 0 & 5 \\ 2 & 0 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 4 & 1 & 0 & 4 & 0 \\ 4 & 0 & 0 & 2 & 3 & 4 \\ 1 & 0 & 0 & 0 & 3 & 0 \\ 0 & 2 & 0 & 0 & 0 & 1 \\ 4 & 3 & 3 & 0 & 0 & 2 \\ 0 & 4 & 0 & 1 & 2 & 0 \end{bmatrix}$$

The corresponding weighted adjacency lists representations are

1	1	2	4
3	2		
1	2	3	5
0	2		

and

1	4	2	1	4	4		
0	4	3	2	4	3	5	4
0	1	4	3				
1	2	5	1				
0	4	1	3	2	3	5	2
1	4	3	1	4	2		

See Appendix B.4 for sample Java code for representing the abstract data type of edge-weighted digraphs.

6.2 Distance and diameter in the unweighted case

One important application for graphs is to model computer networks or parallel processor connections. There are many properties of such networks that can be obtained by studying the characteristics of the graph model. For example, how do we send a message from one computer to another, using the fewest intermediate nodes? This question is answered by finding a shortest path in the graph. We may also want to know what the largest number of communication links that may be required for any two nodes to talk with each other; this is equal to the diameter of the graph.

Definition 6.3. The ***diameter*** of a strongly connected digraph G is the maximum of $d(u,v)$ over all nodes $u, v \in V(G)$.

Note. If the digraph is not strongly connected then the diameter is not defined; the only "reasonable" thing it could be defined to be would be $+\infty$, or perhaps n (since no path in G can have length more than $n-1$).

Example 6.4. The diameter of the 3-cube in Figure 6.1 is easily seen to be 3. Since the digraph G_2 in Figure 4.1 is not strongly connected, the diameter is undefined.

The problem of computing distances in (ordinary, unweighted) digraphs is relatively easy. We already know from Theorem 5.6 that for each search tree, BFS finds the distance from the root s to each node in the tree (this distance equals the level of the node). If v is not in the tree then v is not reachable from s and so $d(s,v) = +\infty$ (or is undefined, depending on your convention).

It is often useful to have a readily available ***distance matrix***. The (i,j)-entry of this matrix contains the distance between node i and node j. Such a matrix can be generated by running `BFSvisit` from each node in turn; this gives an algorithm with running time in $\Theta(n^2+nm)$.

Example 6.5. An adjacency matrix and a distance matrix for the 3-cube shown in Figure 6.1 is given below. The maximum entries of value 3 indicate the diameter. The

reader should check these entries by performing a breadth-first search from each vertex.

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 1 & 1 & 2 & 2 & 3 & 1 & 2 \\ 1 & 0 & 2 & 1 & 3 & 2 & 2 & 1 \\ 1 & 2 & 0 & 1 & 1 & 2 & 2 & 3 \\ 2 & 1 & 1 & 0 & 2 & 1 & 3 & 2 \\ 2 & 3 & 1 & 2 & 0 & 1 & 1 & 2 \\ 3 & 2 & 2 & 1 & 1 & 0 & 2 & 1 \\ 1 & 2 & 2 & 3 & 1 & 2 & 0 & 1 \\ 2 & 1 & 3 & 2 & 2 & 1 & 1 & 0 \end{bmatrix}$$

It is more difficult to compute distance in weighted digraphs. In the next two sections we consider this problem.

Exercises

Exercise 6.2.1. Give an example of a weighted digraph in which the obvious BFS approach does not find the shortest path from the root to each other node.

Exercise 6.2.2. The ***eccentricity*** of a node u in a digraph G is the maximum of $d(u,v)$ over all $v \in V(G)$. The ***radius*** of G is the minimum eccentricity of a node. Write an algorithm to compute the radius of a digraph in time $\Theta(n^2+nm)$. How can we read off the radius from a distance matrix?

6.3 Single-source shortest path problem

The ***single-source shortest path*** problem (SSSP) is as follows. We are given a weighted digraph (G,c) and a source node s. For each node v of G, we must find the minimum weight of a path from s to v (by the weight of a path we mean the sum of the weights on the arcs).

Example 6.6. In the weighted digraph of Figure 6.1, we can see by considering all possibilities that the unique minimum weight path from 0 to 3 is 0 1 3, of weight 3.

We first present an algorithm of Dijkstra that gives an optimal solution to the SSSP whenever all weights are nonnegative. It does not work in general if some weights are negative—see Exercise 6.3.5.

Dijkstra's algorithm is an example of a ***greedy algorithm***. At each step it makes the best choice involving only local information, and never regrets its past choices. It is easiest to describe in terms of a set S of nodes which grows to equal $V(G)$.

Initially the only paths available are the one-arc paths from s to v, of weight $c(s,v)$. At this stage, the set S contains only the single node s. We choose the neighbour u with $c(s,u)$ minimal and add it to S. Now the fringe nodes adjacent to s and u must be updated to reflect the new information (it is possible that there exists a path from

```
algorithm Dijkstra
    Input: weighted digraph (G,c); node s ∈ V(G)
begin
    array colour[0..n−1], dist[0..n−1]
    for u ∈ V(G) do
        dist[u] ← c[s,u]; colour[u] ← WHITE
    end for
    dist[s] ← 0; colour[s] ← BLACK
    while there is a white node
        find a white node u so that dist[u] is minimum
        colour[u] ← BLACK
        for x ∈ V(G) do
            if colour[x] = WHITE then
                dist[x] ← min{dist[x], dist[u] + c[u,x]}
            end if
        end for
    end while
    return dist
end
```

Figure 6.2: Dijkstra's algorithm, first version.

s to v, passing through u, that is shorter than the direct path from s). Now we choose the node (at "level" 1 or 2) whose current best distance to s is smallest, and update again. We continue in this way until all nodes belong to S.

The basic structure of the algorithm is presented in Figure 6.2.

Example 6.7. An application of Dijkstra's algorithm on the second digraph of Figure 6.1 is given in Table 6.1 for each starting vertex s.

The table illustrates that the distance vector is updated at most $n-1$ times (only before a new vertex is selected and added to S). Thus we could have omitted the lines with $S = \{0,1,2,3\}$ in Table 6.1.

Why does Dijkstra's algorithm work? The proof of correctness is a little longer than for previous algorithms. The key observation is the following result. By an S-***path*** from s to w we mean a path all of whose intermediate nodes belong to S. In other words, w need not belong to S, but all other nodes in the path do belong to S.

Theorem 6.8. Suppose that all arc weights are nonnegative. Then at the top of the *while* loop, we have the following properties:

P1: if $x \in V(G)$, then $dist[x]$ is the minimum cost of an S-path from s to x;

P2: if $w \in S$, then $dist[w]$ is the minimum cost of a path from s to w.

Table 6.1: Illustrating Dijkstra's algorithm.

current $S \subseteq V$	**distance vector** $dist$
$\{0\}$	$0,1,4,\infty$
$\{0,1\}$	$0,1,4,3$
$\{0,1,3\}$	$0,1,4,3$
$\{0,1,2,3\}$	$0,1,4,3$
$\{1\}$	$\infty,0,\infty,2$
$\{1,3\}$	$4,0,\infty,2$
$\{0,1,3\}$	$4,0,8,2$
$\{0,1,2,3\}$	$4,0,8,2$
$\{2\}$	$\infty,2,0,5$
$\{1,2\}$	$\infty,2,0,4$
$\{1,2,3\}$	$6,2,0,4$
$\{0,1,2,3\}$	$6,2,0,4$
$\{3\}$	$2,\infty,\infty,0$
$\{0,3\}$	$2,3,6,0$
$\{0,1,3\}$	$2,3,6,0$
$\{0,1,2,3\}$	$2,3,6,0$

Note. Assuming the result to be true for a moment, we can see that once a node u is added to S and $dist[u]$ is updated, $dist[u]$ never changes in subsequent iterations. When the algorithm terminates, all nodes belong to S and hence $dist$ holds the correct distance information.

Proof. Note that at every step, $dist[x]$ does contain the length of *some* path from s to x; that path is an S-path if $x \in S$. Also, the update formula ensures that $dist[x]$ never increases.

To prove P1 and P2, we use induction on the number of times k we have been through the while-loop. Let S_k denote the value of S at this stage. When $k = 0$, $S_0 = \{s\}$, and since $dist[s] = 0$, P1 and P2 obviously hold. Now suppose that they hold after k times through the while-loop and let u be the next special node chosen during that loop. Thus $S_{k+1} = S_k \cup \{u\}$.

We first show that P2 holds after $k+1$ iterations. Suppose that $w \in S_{k+1}$. If $w \neq u$ then $w \in S$ and so P2 trivially holds for w by the inductive hypothesis. On the other hand, if $w = u$, consider any S_{k+1}-path γ from s to u. We shall show that $dist[u] \leq |\gamma|$ where $|\gamma|$ denotes the weight of γ. The last node before u is some $y \in S_k$. Let γ_1 be the subpath of γ ending at y. Then $dist[u] \leq dist[y] + c(y,u)$ by the update formula. Furthermore $dist[y] \leq |\gamma_1|$ by the inductive hypothesis applied to $y \in S_k$. Thus, combining

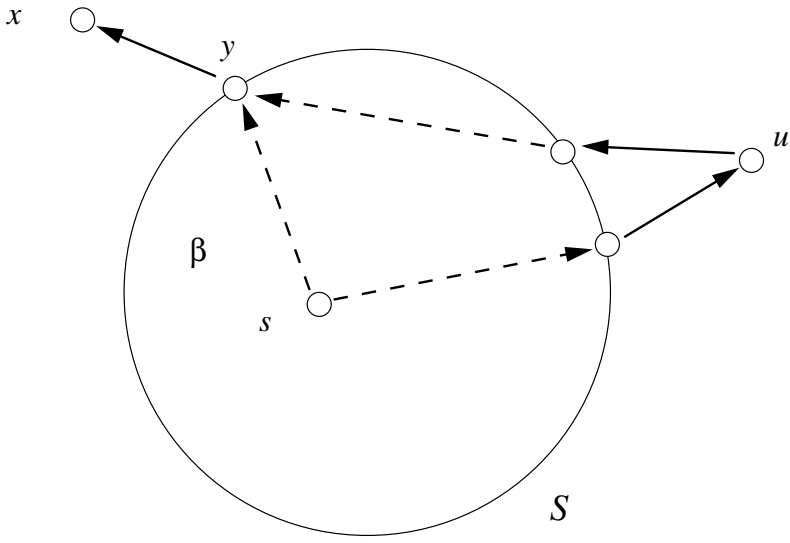


Figure 6.3: Picture for proof of Dijkstra's algorithm.

these inequalities, we obtain $dist[u] \leq |\gamma_1| + c(y,u) = |\gamma|$ as required. Hence P2 holds for every iteration.

Now suppose $x \in V(G)$. Let γ be any S_{k+1}-path to x. If u is not involved then γ is an S_k path and so $|\gamma| \leq dist[x]$ by the inductive hypothesis. Now suppose that γ does include u. If γ goes straight from u to x, we let γ_1 denote the subpath of γ ending at u. Then $|\gamma| = |\gamma_1| + c(u,x) \geq dist[x]$ by the update formula. Otherwise, after reaching u, the path returns into S_k directly, emerging from S_k again, at some node y before going straight to x (see Figure 6.3). Let γ_1 be the subpath of γ ending at y. Since P2 holds for S_k, there is a minimum weight S_k-path β from s to y of length $dist[y]$. Thus by the update formula,

$$|\gamma| = |\gamma_1| + c(y,x) \geq |\beta| + c(y,x) \geq dist[y] + c(y,x) \geq dist[x].$$

Hence P1 holds for all iterations. □

The study of the time complexity of Dijkstra's algorithm leads to many interesting topics.

Note that the value of $dist[x]$ will change only if x is adjacent to u. Thus if we use a weighted adjacency list, the block inside the second for-loop need only be executed m times. However, if using the adjacency matrix representation, the block inside the for-loop must still be executed n^2 times.

The time complexity is of order $an + m$ if adjacency lists are used, and $an + n^2$ with an adjacency matrix, where a represents the time taken to find the node with

```
algorithm Dijkstra2
    Input: weighted digraph (G,c); node s ∈ V(G)
begin
    priority queue Q
    array colour[0..n−1], dist[0..n−1]
    for u ∈ V(G) do
        colour[u] ← WHITE
    end for
    colour[s] ← GREY
    Q.insert(s, 0)
    while not Q.isEmpty() do
        u ← Q.peek(); t1 ← Q.getKey(u)
        for each x adjacent to u do
            t2 ← t1 + c(u,x)
            if colour[x] = WHITE then
                colour[x] ← GREY
                Q.insert(x,t2)
            else if colour[x] = GREY and Q.getKey(x) > t2 then
                Q.decreaseKey(x, t2)
            end if
        end for
        Q.delete()
        colour[u] ← BLACK
        dist[u] ← t1
    end while
    return dist
end
```

Figure 6.4: Dijkstra's algorithm, PFS version.

minimum value of *dist*. The obvious method of finding the minimum is simply to scan through array *dist* sequentially, so that a is of order n, and the running time of Dijkstra is therefore $\Theta(n^2)$. Dijkstra himself originally used an adjacency matrix and scanning of the *dist* array.

The above analysis is strongly reminiscent of our analysis of graph traversals in Section 5.1, and in fact Dijkstra's algorithm fits into the priority-first search framework discussed in Section 5.5. The key value associated to a node u is simply the value $dist[u]$, the current best distance to that node from the root s. In Figure 6.4 we present Dijkstra's algorithm in this way.

It is now clear from this formulation that we need to perform n delete-min operations and at most m decrease-key operations, and that these dominate the running

```
algorithm BellmanFord
    Input: weighted digraph (G,c); node s
begin
    array dist[0..n−1]
    for u ∈ V(G) do
        dist[u] ← ∞
    end for
    dist[s] ← 0
    for i from 0 to n−1 do
        for x ∈ V(G) do
            for v ∈ V(G) do
                dist[v] ← min(dist[v], dist[x] + c(x,v))
            end for
        end for
    end for
    return dist
end
```

Figure 6.5: Bellman–Ford algorithm.

time. Hence using a binary heap (see Section 2.5), we can make Dijkstra's algorithm run in time $O((n+m)\log n)$. Thus if every node is reachable from the source, it runs in time $O(m\log n)$.

The quest to improve the complexity of algorithms like Dijkstra's has led to some very sophisticated data structures that can implement the priority queue in such a way that the decrease-key operation is faster than in a heap, without sacrificing the delete-min or other operations. Many such data structures have been found, mostly complicated variations on heaps; some of them are called Fibonacci heaps and 2–3 heaps. The best complexity bound for Dijkstra's algorithm, using a Fibonacci heap, is $O(m+n\log n)$.

Bellman–Ford algorithm

This algorithm, unlike Dijkstra's handles negative weight arcs, but runs slower than Dijkstra's when all arcs are nonnegative. The basic idea, as with Dijkstra's algorithm, is to solve the SSSP under restrictions that become progressively more relaxed. Dijkstra's algorithm solves the problem one node at a time based on their current distance estimate.

In contrast, the Bellman–Ford algorithm solves the problem for all nodes at "level" $0,1,\ldots,n-1$ in turn. By level we mean the minimum possible number of arcs in a minimum weight path to that node from the source.

Theorem 6.9. Suppose that G contains no negative weight cycles. Then after the i-th

iteration of the outer for-loop, $dist[v]$ contains the minimum weight of a path to v for all nodes v with level at most i.

Proof. Note that as for Dijkstra, the update formula is such that $dist$ values never increase.

We use induction on i. When $i = 0$ the result is true because of our initialization. Suppose it is true for $i-1$. Let v be a node at level i, and let γ be a minimum weight path from s to v. Since there are no negative weight cycles, γ has i arcs. If y is the last node of γ before v, and γ_1 the subpath to y, then by the inductive hypothesis we have $dist[y] \leq |\gamma_1|$. Thus by the update formula we have $dist[v] \leq dist[y] + c(y,v) \leq |\gamma_1| + c(y,v) \leq |\gamma|$ as required. □

The Bellman–Ford algorithm runs in time $\Theta(nm)$ using adjacency lists, since the statement in the inner for-loop need only be executed if v is adjacent to x, and the outer loop runs n times. Using an adjacency matrix it runs in time $\Theta(n^3)$.

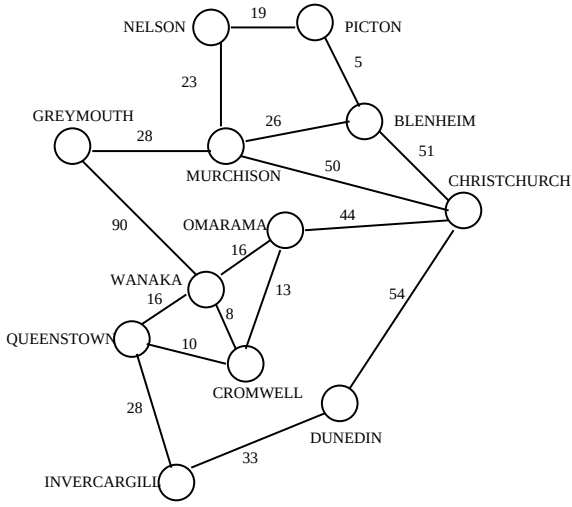
Exercises

Exercise 6.3.1. Run the Bellman–Ford algorithm on the digraph with weighted adjacency matrix given below. Choose each node as the source in turn as in Example 6.7.

$$\begin{bmatrix} 0 & 6 & 0 & 0 & 7 \\ 0 & 0 & 5 & -4 & 8 \\ 0 & -2 & 0 & 0 & 0 \\ 2 & 0 & 7 & 0 & 0 \\ 0 & 0 & -3 & 9 & 0 \end{bmatrix}$$

Exercise 6.3.2. Explain why the SSSP problem makes no sense if we allow digraphs with cycles of negative total weight.

Exercise 6.3.3. The graph shows minimum legal driving times (in multiples of 5 minutes) between various South Island towns. What is the shortest time to drive legally from Picton to (a) Wanaka, (b) Queenstown and (c) Invercargill? Explain which algorithm you use and show your work.



Exercise 6.3.4. Suppose the input to the Bellman–Ford algorithm is a digraph with a negative weight cycle. How does the algorithm detect this, so it can exit gracefully with an error message?

Exercise 6.3.5. Give an example to show that Dijkstra's algorithm may fail to give the correct answer if some weights are negative. Make your example as small as possible. Then run the Bellman–Ford algorithm on the example and verify that it gives the correct answer.

Exercise 6.3.6. Where in the proof of Dijkstra's algorithm do we use the fact that all the arc weights are nonnegative?

6.4 All-pairs shortest path problem

The problem is as follows: given a weighted digraph (G,c), determine for each $u,v \in V(G)$ (the length of) a minimum weight path from u to v.

It is easy to present this information in a distance matrix.

Example 6.10. For the digraph of Figure 6.1, we have already calculated the all-pairs distance matrix in Example 6.7:

$$\begin{pmatrix} 0 & 1 & 4 & 3 \\ 4 & 0 & 8 & 2 \\ 6 & 2 & 0 & 4 \\ 2 & 3 & 6 & 0 \end{pmatrix}.$$

```
algorithm Floyd
    Input: weighted digraph (G,c)
begin
    array d[0..n−1,0..n−1]
    for u ∈ V(G) do
        for v ∈ V(G) do
            d[u,v] ← c(u,v)
        end for
    end for
    for x ∈ V(G) do
        for u ∈ V(G) do
            for v ∈ V(G) do
                d[u,v] ← min(d[u,v], d[u,x] + d[x,v])
            end for
        end for
    end for
    return d
end
```

Figure 6.6: Floyd's algorithm.

Clearly we may compute this matrix as above by solving the single-source shortest path problem with each node taken as the root in turn. The time complexity is of course $\Theta(nA)$ where A is the complexity of our single-source algorithm. Thus running the adjacency matrix version of Dijkstra n times gives a $\Theta(n^3)$ algorithm, and the Bellman–Ford algorithm $\Theta(n^2m)$.

There is a simpler method discovered by R. W. Floyd. Like the Bellman-Ford algorithm, it is an example of an algorithm design technique called ***dynamic programming***. This is where smaller, less-difficult subproblems are first solved, and the solutions recorded, before the full problem is solved. Floyd's algorithm computes a distance matrix from a cost matrix in time $\Theta(n^3)$. It is faster than repeated Bellman–Ford for dense digraphs and unlike Dijkstra's algorithm, it can handle negative costs. For sparse graphs with positive costs repeated Dijkstra is competitive with Floyd, but for dense graphs they have the same asymptotic complexity. A key point in favour of Floyd's algorithm is its simplicity, as can be seen from the algorithm of Figure 6.6. Floyd's algorithm is basically a simple triple for-loop.

Note. Observe that we are altering the value of $d[u,v]$ in the update formula. If we already have a weighted adjacency matrix d, there is no need for the first double loop. We simply overwrite entries in d via the update formula, and everything works.

Example 6.11. An application of Floyd's algorithm on the third graph of Figure 6.1 is given below. The initial cost matrix is as follows.

$$\begin{bmatrix} 0 & 4 & 1 & \infty & 4 & \infty \\ 4 & 0 & \infty & 2 & 3 & 4 \\ 1 & \infty & 0 & \infty & 3 & \infty \\ \infty & 2 & \infty & 0 & \infty & 1 \\ 4 & 3 & 3 & \infty & 0 & 2 \\ \infty & 4 & \infty & 1 & 2 & 0 \end{bmatrix}$$

In the matrices below, the index k refers to the number of times we have been through the outer for-loop.

$$\begin{bmatrix} 0 & 4 & 1 & \infty & 4 & \infty \\ 4 & 0 & \mathbf{5} & 2 & 3 & 4 \\ 1 & \mathbf{5} & 0 & \infty & 3 & \infty \\ \infty & 2 & \infty & 0 & \infty & 1 \\ 4 & 3 & 3 & \infty & 0 & 2 \\ \infty & 4 & \infty & 1 & 2 & 0 \end{bmatrix}$$
$k=1$

$$\begin{bmatrix} 0 & 4 & 1 & \mathbf{6} & 4 & \mathbf{8} \\ 4 & 0 & 5 & 2 & 3 & 4 \\ 1 & 5 & 0 & \mathbf{7} & 3 & \mathbf{9} \\ \mathbf{6} & 2 & \mathbf{7} & 0 & \mathbf{5} & 1 \\ 4 & 3 & 3 & \mathbf{5} & 0 & 2 \\ \mathbf{8} & 4 & \mathbf{9} & 1 & 2 & 0 \end{bmatrix}$$
$k=2$

$$\begin{bmatrix} 0 & 4 & 1 & 6 & 4 & 8 \\ 4 & 0 & 5 & 2 & 3 & 4 \\ 1 & 5 & 0 & 7 & 3 & 9 \\ 6 & 2 & 7 & 0 & 5 & 1 \\ 4 & 3 & 3 & 5 & 0 & 2 \\ 8 & 4 & 9 & 1 & 2 & 0 \end{bmatrix}$$
$k=3$

$$\begin{bmatrix} 0 & 4 & 1 & 6 & 4 & \mathbf{7} \\ 4 & 0 & 5 & 2 & 3 & \mathbf{3} \\ 1 & 5 & 0 & 7 & 3 & \mathbf{8} \\ 6 & 2 & 7 & 0 & 5 & 1 \\ 4 & 3 & 3 & 5 & 0 & 2 \\ \mathbf{7} & \mathbf{3} & \mathbf{8} & 1 & 2 & 0 \end{bmatrix}$$
$k=4$

$$\begin{bmatrix} 0 & 4 & 1 & 6 & 4 & \mathbf{6} \\ 4 & 0 & 5 & 2 & 3 & 3 \\ 1 & 5 & 0 & 7 & 3 & \mathbf{5} \\ 6 & 2 & 7 & 0 & 5 & 1 \\ 4 & 3 & 3 & 5 & 0 & 2 \\ \mathbf{6} & 3 & \mathbf{5} & 1 & 2 & 0 \end{bmatrix}$$
$k=5$

$$\begin{bmatrix} 0 & 4 & 1 & 6 & 4 & 6 \\ 4 & 0 & 5 & 2 & 3 & 3 \\ 1 & 5 & 0 & \mathbf{6} & 3 & 5 \\ 6 & 2 & \mathbf{6} & 0 & \mathbf{3} & 1 \\ 4 & 3 & 3 & \mathbf{3} & 0 & 2 \\ 6 & 3 & 5 & 1 & 2 & 0 \end{bmatrix}$$
$k=6$

In the above matrices we list the entries that change in bold after each increment of k. Notice that undirected graphs, as expected, have symmetric distance matrices.

Why does Floyd's algorithm work? The proof is again by induction.

Theorem 6.12. At the bottom of the outer *for* loop, for all nodes u and v, $d[u,v]$ contains the minimum length of all paths from u to v that are restricted to using only intermediate nodes that have been seen in the outer *for* loop.

Note. Given this fact, when the algorithm terminates, all nodes have been seen in the outer *for* loop and so $d[u,v]$ is the length of a shortest path from u to v.

Proof. To establish the above property, we use induction on the outer for-loop. Let S_k be the set of nodes seen after k times through the outer loop, and define an S_k-path to be one all of whose intermediate nodes belong to S_k. The corresponding value of d is denoted d_k. We need to show that for all k, after k times through the outer for-loop, $d_k[u,v]$ is the minimum length of an S_k-path from u to v.

When $k=0$, $S_0=\emptyset$ and the result holds. Suppose it is true after k times through the outer loop and consider what happens at the end of the $(k+1)$-st time through the outer loop. Suppose that x was the last node seen in the outer loop, so $S_{k+1}=S_k\cup\{x\}$.

Fix $u, v \in V(G)$ and let L be the minimum length of an S_{k+1}-path from u to v. Obviously $L \leq d_{k+1}[u,v]$; we show that $d_{k+1}[u,v] \leq L$.

Choose an S_{k+1}-path γ from u to v of length L. If x is not involved then the result follows by inductive hypothesis. If x is involved, let γ_1, γ_2 be the subpaths from u to x and x to v respectively. Then γ_1 and γ_2 are S_k-paths and by the inductive hypothesis,

$$L \geq |\gamma_1| + |\gamma_2| \geq d_k[u,x] + d_k[x,v] \geq d_{k+1}[u,v].$$

□

The proof does not use the fact that weights are nonnegative—in fact Floyd's algorithm works for negative weights (provided of course that a negative weight cycle is not present).

Exercises

Exercise 6.4.1. Run Floyd's algorithm on the matrix of Exercise 6.3.1 and check your answer against what was obtained there.

Exercise 6.4.2. Suppose the input to Floyd's algorithm is a digraph with a negative weight cycle. How does the algorithm detect this, so it can exit gracefully with an error message?

Exercise 6.4.3.

The matrix M shows costs of direct flights between towns A, B, C, D, E, F (where ∞, as usual, means that no direct flight exists). You are given the job of finding the cheapest route between each pair of towns. Solve this problem. Hint: save your working.

$$M = \begin{bmatrix} 0 & 1 & 2 & 6 & 4 & \infty \\ 1 & 0 & 7 & 4 & 2 & 11 \\ 2 & 7 & 0 & \infty & 6 & 4 \\ 6 & 4 & \infty & 0 & \infty & 1 \\ 4 & 2 & 6 & \infty & 0 & 3 \\ \infty & 11 & 4 & 1 & 3 & 0 \end{bmatrix}.$$

The next day, you are told that in towns D, E, F, political troubles mean that no passenger is allowed to both take off and land there. Solve the problem with this additional constraint.

6.5 Minimum spanning tree problem

In this section, we use "tree" to mean "free tree" throughout. Recall that a tree is a connected acyclic graph. A ***spanning tree*** of a graph G is a spanning subgraph of G that is itself a tree.

Definition 6.13. Let G be a weighted graph. A ***minimum spanning tree*** (MST) is a spanning tree for G which has minimum total weight (sum of all edge weights).

Note. If all weights are nonnegative and we only want a spanning subgraph with minimum total weight, this must be a tree anyway (if not, delete an edge from a cycle and keep a spanning subgraph).

The problem we are concerned with is this: given a weighted graph G, find a MST for G. There are obvious practical applications of this idea. For example, how can we cheaply link sites with communication links so that they are all connected?

Example 6.14. In the third graph of Figure 6.1, the tree determined by the edges

$$\{0,2\},\{1,3\},\{3,5\},\{4,5\},\{2,4\}$$

has total weight 9. It is a tree and has the 5 smallest weight edges, so must be a MST.

One should not search naively through all possible spanning trees: it is known that there are n^{n-2} spanning trees for the complete graph K_n, for example!

In this section we present two efficient algorithms to find a MST that (like Dijkstra's algorithm) fall into the category of greedy algorithms.

Each builds up a MST by iteratively choosing an edge greedily, that is, choosing one with minimum weight, subject to not obviously ruining our chance of extending to a spanning tree. It turns out that this simple approach works for the MST problem (obviously, not for all graph optimization problems!). There are other algorithms with better theoretical complexity for the MST problem, but none is as simple to understand.

The algorithms can be described in an informal way very easily. The first, ***Prim's algorithm***, starts at a root vertex and chooses at each step an edge of minimum weight from the remaining edges, subject to: (a) adding the edge does not create a cycle in the subgraph built so far, and (b) the subgraph built so far is connected. By contrast, ***Kruskal's algorithm*** does not start at a root: it follows rule (a) and ignores (b). Prim's algorithm is perhaps easier to program, but Kruskal's is easier to perform by hand.

Each algorithm clearly terminates when no more edges can be found that satisfy the above condition(s). Since Prim's algorithm maintains acyclicity and connectedness, at each stage it has built a subgraph that is a tree. Kruskal's algorithm maintains acyclicity, so it has a forest at each step, and the different trees merge as the algorithm progresses.

We might first ask why does each algorithm even form a spanning tree (see Exercise 6.5.2). However, even given that a spanning tree is formed, it is not at all obvious that this spanning tree has minimum possible weight among all spanning trees of the graph.

We now show the correctness of these algorithms. We may suppose that the graph is connected. If it is not, we cannot find a spanning tree anyway, and must be satisfied with a spanning forest. Prim's algorithm will terminate when it has explored the first component and must be restarted from a new root in another component. Kruskal's algorithm will find a spanning forest without modification.

Theorem 6.15. Prim's and Kruskal's algorithms are correct.

Proof. Define a set of edges to be *promising* if it can be extended in some way to a MST. Then the empty set is promising since some MST exists. We claim that at each step, the algorithms above have chosen a promising set of edges. When they terminate, no further extension of the set is possible (by rule (a) above), and so we must have a MST.

To prove the claim efficiently, we need a technical fact, as follows. Suppose that B is a subset of $V(G)$, not containing all the vertices of G, and T a promising set of edges such that no edge in T leaves B. In other words, either both endpoints are in B or neither endpoint is in B. Then if e is a minimum weight edge that does leave B (it has one endpoint in B and one outside) then $T \cup \{e\}$ is also promising.

To see this fact, note that since T is promising, it is contained in some MST, U say. If $e \in U$ there is nothing to prove. Otherwise, when we add e to U we create exactly one cycle. There must be at least one other edge, say e', that leaves B, otherwise the cycle could not close. If we remove e' we obtain a new tree that spans G and whose total weight is no greater than the total weight of U. Thus V is also a MST, and since it contains $T \cup \{e\}$, that set is promising.

Now to prove the claim, suppose that our algorithm has maintained a promising set T of edges so far, and it has just chosen edge $e = \{u, v\}$. If we take B at each step to be the set of vertices in the tree (Prim) or the set of vertices in the tree containing u (Kruskal), then we may apply the fact above to conclude that $T \cup \{e\}$ is promising. This concludes the proof of correctness. □

The above informal descriptions of MST algorithms can be converted easily to an algorithm. In Figure 6.7 we present the algorithm. Note how similar Prim's algorithm is to Dijkstra's. The main difference is in the update formula. We also store the PFS tree, which we elected not to do for Dijkstra.

In Prim's algorithm, we checked whether a cycle would be created by adding an edge in the usual way: when exploring $\{u, v\}$ from u, if v has already been seen and is not the parent of u, then adding $\{u, v\}$ creates a cycle. With Kruskal's algorithm, we must use another method, since the above test does not work. Both u and v may have been seen, but may be in different trees.

Observe that if we try to add an edge both of whose endpoints are in the same tree in the Kruskal forest, this will create a cycle, so the edge must be rejected. On the other hand, if the endpoints are in two different trees, a cycle definitely will not be created; rather, the two trees merge into a single one, and we should accept the edge. We need a data structure that can handle this efficiently. All we need is to be able to find the tree containing an endpoint, and to merge two trees. The ***disjoint sets*** or ***union-find*** ADT is precisely what is needed. It allows us to perform the `find` and `union` operations efficiently.

The complexity of the algorithm depends to a great extent on the data structures used. The best known for Prim is the same as for Dijkstra, namely $O(m + n \log n)$, and

```
algorithm Prim
    Input: weighted graph (G,c); vertex s ∈ V(G)
begin
    priority queue Q
    array colour[0..n−1], pred[0..n−1]
    for u ∈ V(G) do
        colour[u] ← WHITE; pred[u] ← NULL
    end for
    colour[s] ← GREY
    Q.insert(s, 0)
    while not Q.isEmpty() do
        u ← Q.peek()
        for each x adjacent to u do
            t ← c(u,x)
            if colour[x] = WHITE then
                colour[x] ← GREY; pred[x] ← u
                Q.insert(x,t)
            else if colour[x] = GREY and Q.getKey(x) > t then
                Q.decreaseKey(x,t); pred[x] ← u
            end if
        end for
        Q.delete()
        colour[u] ← BLACK
    end while
    return pred
end
```

Figure 6.7: Prim's algorithm.

for Kruskal $O(m \log n)$. The disjoint sets ADT can be implemented in such a way that the union and find operations in Kruskal's algorithm runs in *almost* linear time (the exact bound is very complicated). So if the edge weights are presorted, or can be sorted in linear time (for example, if they are known to be integers in a fixed range), then Kruskal's algorithm runs for practical purposes in linear time.

Exercises

Exercise 6.5.1. Carry out each of these algorithms on the weighted graph of Figure 6.1. Do the two algorithms give the same spanning tree?

Exercise 6.5.2. Prove the assertion made above that when Kruskal's or Prim's algorithm terminates, the current set of edges forms a spanning tree.

```
algorithm Kruskal
    Input: weighted graph (G, c)
begin
    disjoint sets ADT A
    initialize A with each vertex in its own set
    sort the edges in increasing order of cost
    for each edge {u, v} in increasing cost order do
        if not A.set(u) = A.set(v) then
            add this edge
            A.union(A.set(u), A.set(v))
        end if
    end for
    return A
end
```

Figure 6.8: Kruskal's algorithm.

Exercise 6.5.3. Consider the following algorithm for the MST problem. Repeatedly delete edges from a connected graph G, at each step choosing the most expensive edge we can, subject to maintaining connectedness. Does it solve the MST problem sometimes? always?

6.6 Hard graph problems

We have presented several efficient algorithms for some common digraph problems. All of these algorithms have running time bounded above by a low degree polynomial in the size of the input. However, there are many essential problems that currently do not have known polynomial-time algorithms (so-called ***NP-hard*** problems). Some examples are:

- finding the longest path between two nodes of a digraph;
- finding a k-colouring of a graph, for fixed $k \geq 3$;
- finding a cycle that passes through all the vertices of a graph (a ***Hamiltonian cycle***);
- finding a minimum weight path that passes through all the vertices of a weighted digraph (the ***travelling salesperson problem*** or TSP);
- finding the largest ***independent set*** in a graph — that is, a subset of vertices no two of which are connected by an edge;
- finding the smallest ***vertex cover*** of a graph—that is, a special subset of vertices so that each vertex of the graph is adjacent to one in that subset. (However,

from Exercise 5.9.3, this problem is polynomial-time solvable when restricted to bipartite graphs.)

Investigating these problems is an active research area in computer science. In many cases the only approach known is essentially to try all possibilities, with some rules that prevent the listing of obviously hopeless ones. In some special cases (for example, graphs that are ***planar*** (they can be drawn in the plane without edges crossing) or are in some sense "close to" trees, much faster algorithms can be developed.

Exercises

Exercise 6.6.1. Find a Hamiltonian cycle of the graph in Exercise 6.3.3. Try to solve the TSP for this graph.

Exercise 6.6.2. What is the exact relation between the independent set and vertex cover problems?

6.7 Notes

Dijkstra's algorithm was proposed by E. W. Dijkstra in 1959. The Bellman–Ford algorithm was proposed independently by R. Bellman (1958) and L. R. Ford, Jr (1956). Floyd's algorithm was developed in 1962 by R. W. Floyd. Prim's algorithm was presented by R. C. Prim in 1957 and reinvented by E. W. Dijkstra in 1959, but had been previously introduced by V. Jarnik in 1930. Kruskal's algorithm was introduced by J. Kruskal in 1956.

Part III

Appendices

Appendix A

Java code for Searching and Sorting

This appendix contains Java implementations for many of the common search and sorting algorithms presented in the book.

A.1 Sorting and selection

The Java class below contains class methods for sorting integer arrays and for selecting an array element of a given rank. These algorithms insertion sort, Shellsort, mergesort, quicksort, heapsort, and quickselect are described in detail in Chapter 2. We may place them all in a Java public class or they may be cut-and-pasted, as needed, into a Java application.

```
//  Insertion sort of an input array a of size n
//
 public static void insertionSort( int [] a )
 {
      for(  int i = 1; i < a.length; i++ ) {
          int tmp = a [ i ];
          int k = i - 1;
          while( k >= 0 && tmp < a[ k ] ) {
               a [k + 1 ] = a[ k ];
               k--;
          }
          a[ k + 1 ] = tmp;
      }
}

//  Selection sort of an input array a of size n
//
public static void selectionSort( int [] a )
```

```
{
      for( int i = 0; i < a.length - 1; i++ ) {
          int posMin = i;
          for( int k = i + 1; k < a.length; k++ ) {
               if ( a[ posMin ] > a[ k ] ) posMin = k;
          }
          if ( posMin != i ) swap( a, i, posMin );
      }
}

//  Bubble sort of an input array a of size n
//
public static void bubbleSort( int [] a )
{
      for( int i = a.length - 1; i > 0; i-- ) {
          for( int k = 0; k < i; k++ ) {
               if ( a[ k ] > a[ k + 1 ] ) swap( a, k, k + 1 );
          }
     }
}

//  Insertion sort of an input array a of size n:
//  a private method used by quicksort and quickselect:
//  sorting between the indices lo and hi: 0 <= lo <= hi < n
//
private static void insertionSort( int [] a, int lo, int hi )
{
     if ( lo > hi || lo < 0 || hi >= a.length ) {
        lo = 0;
        hi = a.length - 1;
     }
     for ( int i = lo + 1; i <= hi; i++ ) {
          int tmp = a[i];
          int k = i - 1;
          while( k >= lo && tmp < a[ k ] ) {
                a[ k + 1 ] = a[ k ];
                k--;
          }
          a[ k + 1 ] = tmp;
      }
}

//  Shell's sort of an input array a of size n
//  using a sequence of gaps by G. Gonnet
//
public static void shellSort( int [] a )
{
    for( int gap = a.length/2; gap > 0;
                gap = (gap == 2) ? 1 : (int)(gap/2.2) )
         for( int i = gap; i < a.length; i++ ) {
             int tmp = a[ i ];
             int k = i;
             while( k >= gap && tmp < a [ k - gap ] ) {
```

```
                a[ k ] = a[ k - gap ];
                k -= gap;
            }
            a[ k ] = tmp;
        }
}

//  Mergesort of an input array a of size n
//  using a temporary array to merge data
//
public static void mergeSort( int [] a)
{
     int [] tmp = new int[ a.length ];
     mergeSort( a, tmp, 0, a.length - 1 );
}

private static void mergeSort( int [] a, int [] tmp,
                               int left, int right)
{
      if ( left < right ) {
          int centre = (left + right) / 2;
          mergeSort( a, tmp, left, centre);
          mergeSort( a, tmp, centre + 1, right);
          merge( a, tmp, left, centre + 1, right );
      }
}

private static void merge( int [] a, int [] tmp,
                           int left, int right, int rend )
{
     int lend = right - 1;
     int tpos = left;
     int lbeg = left;

     // Main loop
     while( left <= lend && right <= rend )
           if ( a[ left ] < a[ right ] )
              tmp[ tpos++ ] = a[ left++ ];
          else
             tmp[ tpos++ ] = a[ right++ ];

     // Copy the rest of the first half
    while( left <= lend )
          tmp[ tpos++ ] = a[ left++ ];

     // Copy the rest of the second half
     while( right <= rend )
           tmp[ tpos++ ] = a[ right++ ];

     // Copy tmp array back
     for( tpos = lbeg; tpos <= rend; tpos++ )
         a[ tpos ] = tmp[ tpos ];
}
```

```
//  Quicksort of an input array a of size n  using a median-of-three pivot
//  and insertion sort of subarrays of size less than CUTOFF threshold:
//
static public int CUTOFF = 10;

public static void quickSort( int [] a )
{
     quickSort( a, 0, a.length - 1 );
}

private static void quickSort( int [] a, int lo, int hi )
{
     if ( lo + CUTOFF > hi )
        insertionSort( a, lo, hi );
    else {
        // Sort low, middle, high
        int mi = ( lo + hi ) / 2;
        if ( a[ mi ] < a[ lo ] ) swap( a, lo, mi );
        if ( a[ hi ] < a[ lo ] ) swap( a, lo, hi );
        if ( a[ hi ] < a[ mi ] ) swap( a, mi, hi );

        // Place pivot p at position hi - 1
        swap( a, mi, hi - 1 );
        int p = a[ hi - 1];

        // Begin partitioning
       int i, j;
       for ( i = lo, j = hi - 1; ; ) {
            while( a[ ++i ] < p );
            while( p < a[ --j ] );
            if ( i < j ) swap( a, i, j );
            else break;
        }

        // Restore pivot
        swap( a, i, hi - 1 );
        // Sort small elements
        quickSort( a, lo, i - 1 );
         // Sort large elements
        quickSort( a, i + 1, hi );
    }
}

private static void swap( int [] a, int i, int j )
{
  int tmp = a[ i ];
  a[ i ] = a[ j ];
  a[ j ] = tmp;
}

//  Heapsort of an input array a of size n
//  using percolateDown() and swap() methods
```

```
//
public static void heapSort( int [] a )
{
     // build a heap
     for ( int i = a.length / 2 - 1; i >= 0; i-- )
          percolateDown( a, i, a.length );
     // successively delete the max and restore the heap
     for( int i = a.length - 1; i >= 1; i-- ) {
          swap( a, 0, i );
          percolateDown( a, 0, i );
     }
 }

 // Heapifying method to restore a heap a[0],...,a[size-1]
 // after changing a[ i ]; a child / parent position is one
 // greater than an index of the same array element
 private static void percolateDown( int [] a, int i, int size )
 {
      int child;
      int parent = i + 1;

      for ( child = parent * 2; child < size;  child = parent * 2 ) {
            if ( a[ parent - 1 ] < a[ child - 1 ] ||
                 a[ parent - 1 ] < a[ child ] ) {
               if ( a[ child - 1 ] < a[ child] ) {
                  swap( a, parent - 1, child );
                  parent = child + 1;
               } else {
                  swap( a, parent - 1, child - 1 );
                  parent = child;
               }
            }  else break;
      }
      if ( child == size && a[ parent - 1 ] < a[ child - 1 ] )
         swap( a, parent - 1, child - 1 );
  }

//  Counting sort of an input array a of size n
//  with elements such that min <= a[i] <= max
//  (assuming that this condition holds)
//
public static void countSort( int [] a, int min, int max )
{
      int i, j;
      int m = max - min + 1;
      int [] accum = new int[ m ];
      for ( j = 0; j < m; j++ ) accum[ j ] = 0;
      for ( i = 0;  i < a.length; i++ ) accum[ a[ i ] ]++;
      for ( i = j = 0; j < m; j++ ) {
             if ( accum[ j ] == 0 ) continue;
             while( ( accum[ j ]-- ) > 0 )
                   a[ i++ ] = j + min;
      }
```

```
}

// Quick select in an input array a of size n:
// returns the k-th smallest element in a[ k - 1 ]
//
public static void quickSelect( int[] a, int k )
{
     quickSelect( a, 0, a.length - 1, k );
}

private static void quickSelect( int[] a, int lo, int hi, int k )
{
      if ( lo + CUTOFF > hi ) {
          insertionSort( a, lo, hi );
      } else {
          // Sort low, middle, high
         int mi = ( lo + hi ) / 2;
         if ( a[ mi ] < a[ lo ] ) swap( a, lo, mi );
         if ( a[ hi ] < a[ lo ] ) swap( a, lo, hi );
         if ( a[ hi ] < a[ mi ] ) swap( a, mi, hi );

          // Place the pivot p into the rightmost place
          swap( a, mi, hi - 1 );
          int p = a[ hi - 1 ];

          // Begin partitioning
          int i, j;
          for( i = lo, j = hi - 1; ; ) {
              while( a[ ++i ] < p );
              while( p < a[ --j ] );
              if ( i < j ) swap( a, i, j ); else break;
          }

          // Restore pivot
          swap( a, i, hi - 1 );

          // Selection by recursion (the only changed part!)
         if ( k - 1 < i ) quickSelect( a, lo, i - 1, k );
         else if ( k - 1 > i ) quickSelect( a, i + 1, hi, k );
     }
  }
```

A.2 Search methods

We now present several Java methods for searching in an integer array. The algorithms (sequential search and binary search) are described in detail in Chapter 3.

```
// Sequential search for key in an array a
//
public static int sequentialSearch( int[] a, int key)
throws ItemNotFound
```

```
{
  for( int i = 0; i < a.length; i++ )
    if ( a[ i ] == key ) return i;
  throw new ItemNotFound( ``SequentialSearch fails'' );
}

// Binary search for key in a sorted array a
 //
public static int binarySearch( int[] a, int key)
throws ItemNotFound
{
   int lo = 0;
   int hi = a.length - 1;
   int mi;

   while( lo <= hi ) {
      mi = ( lo + hi ) / 2;
      if ( a[ mi ] < key ) lo = mi + 1;
      else if( a[ mi ] > key ) hi = mi - 1;
      else return mi;
   }
   throw new ItemNotFound( "BinarySearch fails" );
}

// Binary search using two-way comparisons
//
public static int binarySearch2( int[] a, int key)
throws ItemNotFound
{
  if ( a.length == 0 )
    throw new ItemNotFound( "Zero-length array" );

  int lo = 0;
  int hi = a.length - 1;
  int mi;

  while( lo < hi ) {
      mi = ( lo + hi ) / 2;
      if ( a[ mi ] < key ) lo = mi + 1;
      else                 hi = mi;
  }
  if ( a[ lo ] == key ) return lo;
  throw new ItemNotFound( "BinarySearch fails" );
}
```

Appendix B

Java graph ADT

This appendix presents a simplified abstract class for representing a graph abstract data type (ADT). Although it is fully functional, it purposely omits most exception handling and other niceties that should be in any commercial level package. These details would distract from our overall (introductory) goal of showing how to implement a basic graph class in Java.

Our plan is to have a common data structure that represents both graphs and digraphs. A graph will be a digraph with anti-parallel arcs; that is, if $(u,v) \in E$ then $(v,u) \in E$ also. The initial abstract class presented below requires a core set of methods needed for the realized graph ADT. It will be extended with the actual internal data structure representation in the form of adjacency matrix or adjacency lists (or whatever the designer picks).

```
package graphADT;

import java.util.ArrayList;
import java.io.BufferedReader;

/*
 * Current Abstract Data Type interface for (di)graph classes.
 */
public interface Graph
{
 // Need default, copy and BufferedReader constructors
 // (commented since Java doesn't allow abstract constructors!)
 //
 // public GraphADT();
```

```
 // public GraphADT(GraphADT);
 // public GraphADT(BufferedReader in);
```

Right from the beginning we get in trouble since Java does not allow abstract constructors. We will leave these as comments and hope the graph class designer will abide by them. We want to create graphs from an empty graph, copy an existing graph, or read in one from some external source. In the case of a `BufferedReader` constructor the user has to attach one to a string, file or keyboard. We will see examples later.

We now proceed by presenting the alteration methods required for our graph class interface.

```
  // data structure modifiers
  //
  void addVertices(int i);       // Add some vertices
  void removeVertex(int i);      // Remove vertex

  void addArc(int i, int j);     // Add directed edge
  void removeArc(int i, int j);  // Remove directed edge

  void addEdge(int i, int j);    // Add undirected edge
  void removeEdge(int i, int j); // Remove undirected edge
```

This small set of methods allows one to build the graph. We will soon explicitly define the methods for adding or deleting edges in terms of the two arc methods. An extended class can override these to improve efficiency if it wants. We now list a few methods for extracting information from a graph object.

```
// data structure queries
//
boolean isArc(int i, int j);    // Check for arcs
boolean isEdge(int i, int j);   // Check for edges

int degree(int i);         // Number of neighbours (outgoing)
int inDegree(int i);       // Number of incoming arcs

ArrayList<Integer> neighbours(int i);  // List of (out-)neighbours

int order();                  // Number of vertices
int size();                   // Number of edges

// output (default same as representation)
//
String toString();

} // end of interface Graph
```

For our implementation, we want a vertex's degree to equal the number of vertices returned by the `neighbours` method, which in our implementations will correspond to `degree(i)`. Also, the method `isEdge(i,j)` will most likely just check whether `isArc(i,j) && isArc(j,i)` is true.

Finally, one nice thing to offer is a method to view/save/print a graph. Traditionally in Java we define a `toString` method for this. Our two actual implementations will return human viewable adjacency lists or adjacency matrices, depending on the internal representation.

We have the `toString` method as an interface requirement for the derived classes to define. We want a `BufferedReader` constructor for a graph class to accept its own `toString` output. Two common external graph representations are handled by the methods given below.

```
public String toStringAdjMatrix()
{
 StringBuffer o = new StringBuffer();
 o.append(order()+"\n");

 for( int i = 0; i < order(); i++ )
 {
    for( int j = 0; j < order(); j++ )
    {
       if ( isArc(i,j) ) o.append("1 ");
       else o.append("0 ");
    }
    o.append("\n");
 }
 return o.toString();
}

public String toStringAdjLists()
{
 StringBuffer o = new StringBuffer();
 o.append(order()+"\n");

 for( int i = 0; i < order(); i++ )
 {
    for( int j = 0; j < order(); j++ )
    {
       if ( isArc(i,j) ) o.append(j+" ");
    }
    o.append("\n");
 }
 return o.toString();
}
```

To make things convenient for ourselves we require that the first line of our (two) external graph representations contain the number of vertices. Strictly speaking,

this is not needed for an 0/1 adjacency matrix. This makes our parsing job easier and this format allows us to store more than one graph per input stream. (We can terminate a stream of graphs with a sentinel graph of order zero.)

B.1 Java adjacency matrix implementation

We now define our first usable graph class based on an adjacency matrix representation (for graphs and digraphs). This class extends our graph interface `Graph`.

```
package graphADT;

import java.io.*;
import java.util.*;

/*  Current implementation uses adjacency matrix form of a graph.
 */
public class GraphAdjMatrix implements Graph
{

    //  Internal Representation and Constructors
    //
    protected int order;            // Number of vertices
    protected boolean adj[][];      // Adjacency matrix of graph

    public GraphAdjMatrix()         // default constructor
    {
        order = 0;
    }

    public GraphAdjMatrix(GraphAdjMatrix G) // copy constructor
    {
        int n = G.order();
        if ( n>0 ) { adj = new boolean[n][n]; order = n; }

        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                adj[i][j] = G.adj[i][j];
    }
    public GraphAdjMatrix(Graph G) // conversion constructor
    {
        int n = G.order();
        if ( n>0 ) { adj = new boolean[n][n]; order = n; }

        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                if (G.isArc(i,j)) adj[i][j] = true;
    }
```

The default constructor simply creates an empty graph and thus there is no need to allocate any space. The two copy constructors simply copy onto a new n-by-n matrix the boolean adjacency values of the old graph. Notice that we want new storage

and not an object reference for the copy.

An alternative implementation (as given in the first edition of this textbook) would also keep an integer variable `space` to represent the total space allocated. Whenever we delete vertices we do not want to reallocate a new matrix but to reshuffle the entries into the upper sub-matrix. Then whenever adding more vertices we just extend the dimension of the sub-matrix.

Our last input constructor for `GraphAdjMatrix` is now given.

```
public GraphAdjMatrix(BufferedReader buffer)
{
  try
  {
      String line = buffer.readLine().trim();
      String[] tokens = line.split("\\s+");

      if (tokens.length != 1)
      {
        throw new Error("bad format: number of vertices");
      }
      int n = order = Integer.parseInt(tokens[0]);

      if ( n>0 ) adj = new boolean[n][n];

      for (int i = 0; i < n; i++)
      {
         line = buffer.readLine().trim();
         tokens = line.split("\\s+");
         if (tokens.length != n)
         {
            throw new Error("bad format: adjacency matrix");
         }

         for (int j = 0; j < n; j++)
         {
            int entry = Integer.parseInt(tokens[j]);
            adj[i][j] = entry != 0;
         }
       }
  }
   catch (IOException x)
   { throw new Error("bad input stream"); }
}
```

We have tried to minimize the complexity of this `BufferedReader` constructor. We do however throw a couple of errors if something does go wrong. Otherwise, this method simply reads in an integer n denoting the dimension of the adjacency matrix and then reads in the 0/1 matrix. Notice how the use of the `String.split` method to

extract the integer inputs.

We next define several methods for altering this graph data structure. The first two methods allow the user to add or delete vertices from a graph.

```
//  Mutator Methods
//
public void addVertices(int n)
{
    assert(0 <= n );
    boolean matrix[][] = new boolean[order+n][order+n];

    for (int i = 0; i < order; i++)
    {
        for (int j = 0; j < order; j++)
        {
            matrix[i][j] = adj[i][j];
        }
    }
    order += n;
    adj = matrix;
}

public void removeVertex(int v)
{
    assert(0 <= v && v < order);
    order--;

    for (int i = 0; i < v; i++)
    {
        for (int j = v; j < order; j++)
        {
            adj[i][j] = adj[i][j+1];
        }
    }

    for (int i = v; i < order; i++)
    {
        for (int j = 0; j < v; j++)
        {
            adj[i][j] = adj[i+1][j];
        }
        for (int j = v; j < order; j++)
        {
            adj[i][j] = adj[i+1][j+1];
        }
    }
}
```

The `removeVertex` method is somewhat complicated in that we have to remove a row and column from the matrix corresponding to the vertex being deleted. We

decided to do this in two passes. The first pass (for variable $i < v$) simply shifts all column indices $j \geq v$ to the left. The second pass (for variable $i \geq v$) has to shift entries up by one while also shifting column indices $j \geq v$ to the left. The user of the graph should realize that the indices of the vertices change!

Next, we have four relatively trivial methods for adding and deleting arcs (and edges). Like the mutator methods for checking for valid vertex indices we add some important `assert` statements that can be turned on with an option to the java compiler for debugging graph algorithms.

```
// Mutator Methods (cont.)

 public void addArc(int i, int j)
 {
     assert(0 <= i && i < order);
     assert(0 <= j && j < order);
     adj[i][j] = true;
 }

 public void removeArc(int i, int j)
 {
     assert(0 <= i && i < order);
     assert(0 <= j && j < order);
     adj[i][j] = false;
 }

 public void addEdge(int i, int j)
 {
     assert(0 <= i && i < order);
     assert(0 <= j && j < order);
     adj[i][j] = adj[j][i] = true;
 }

 public void removeEdge(int i, int j)
 {
     assert(0 <= i && i < order());
     assert(0 <= j && j < order());
     adj[i][j] = adj[j][i] = false;
 }
```

The methods to access properties of the graph are also pretty straightforward.

```
 //  Access Methods
 //
 public boolean isArc(int i, int j)
 {
     assert(0 <= i && i < order);
     assert(0 <= j && j < order);
     return adj[i][j];
 }
```

```
public boolean isEdge(int i, int j)
{
    return isArc(i,j) && isArc(j,i);
}

public int degree(int i) // row count
{
    assert(0 <= i && i < order);
    int sz = 0;
    for (int j = 0; j < order; j++)
    {
        if (adj[i][j]) sz++;
    }
    return sz;
}

public int inDegree(int i) // column count
{
    assert(0 <= i && i < order);
    int sz = 0;
    for (int j = 0; j < order; j++)
    {
        if (adj[j][i]) sz++;
    }
    return sz;
}
```

Our constant-time method for checking whether an arc is present in a graph is given above in the method `isArc`. Unfortunately, we have to check all neighbours for computing the in- and out- degrees. Also the method, given below, for returning a list of neighbours for a vertex will need to scan all potential vertices.

```
public ArrayList<Integer> neighbours(int i)
{
    assert(0 <= i && i < order);
    ArrayList<Integer> nbrs = new ArrayList<Integer>();

    for (int j = 0; j < order; j++)
    {
        if (adj[i][j]) nbrs.add(j);
    }

    return nbrs;
}


public int order()
{
    return order;
}
```

```
    public int size() // Number of arcs (edges count twice)
    {
        int sz = 0;
//      boolean undirected = true;
        for (int i = 0; i< order; i++)
        {
            for (int j = 0; j< order; j++)
            {
                if ( adj[i][j]) sz++;
//              if ( adj[i][j] != adj[j][i] ) undirected = false;
            }
        }
        return sz;  // undirected ? sz / 2 : sz;
    }
```

The order of the graph is stored in an integer variable `_order`. However, we have to count all `true` entries in the boolean adjacency matrix to return the size. Notice that if we are working with an undirected graph this returns twice the expected number (since we store each edge as two arcs). If we specialize this class we may want to uncomment the indicated statements to autodetect undirected graphs (whenever the matrix is symmetric). It is probably safer to leave it as it is written, with the understanding that the user knows how *size* is defined for this implementation of `Graph`.

```
  // default output is readable by constructor
  //
  public String toString() { return toStringAdjMatrix(); }

} // end class GraphAdjMatrix
```

We finish our implementation by setting our output method `toString` to return an adjacency matrix. Recall the method `toStringAdjMatrix` was presented earlier on page 180.

B.2 Java adjacency lists implementation

We now present an alternate implementation of our graph ADT using the adjacency lists data structure. We will use the Java API class `Vector` to store these lists.

```
package graphADT;

import java.io.*;
import java.util.*;

/*  Current implementation uses adjacency lists form of a graph.
 */
public class GraphAdjLists implements Graph
{
    //  Internal Representation and Constructors
    //
```

```
protected ArrayList<ArrayList<Integer>> adj;

public GraphAdjLists()
{
    adj = new ArrayList<ArrayList<Integer>>();
}

public GraphAdjLists(Graph G)
{
    int n = G.order();
    adj = new ArrayList<ArrayList<Integer>>();
    for (int i = 0; i < n; i++)
    {
        adj.add(G.neighbours(i));
    }
}
```

We use an `ArrayList` that contains an `ArrayList` of `Integer` for our representation. We decided that the copy constructor for `Graph` is sufficient in terms of efficiency so do not need to define a specialized copy constructor for `GraphAdjLists`, handled automatically by the Java runtime environment. The default constructor creates a list of no lists (that is, no vertices). For better efficiency, the copy constructor takes over the role of our allocator and appends the neighbour lists of the graph parameter G directly onto a new adjacency list.

```
public GraphAdjLists(BufferedReader buffer)
{
    try
    {
        String line = buffer.readLine().trim();
        String[] tokens = line.split("\\s+");

        if (tokens.length != 1)
        {
            throw new Error("bad format: number of vertices");
        }

        adj = new ArrayList<ArrayList<Integer>>();
        int n = Integer.parseInt(tokens[0]);

        for (int u = 0; u < n; u++)
        {
            ArrayList<Integer> current = new ArrayList<Integer>();
            line = buffer.readLine().trim();
            int limit = 0;
            if (!line.equals(""))
            {
                tokens = line.split("\\s+");
                limit = tokens.length;
            }
```

```
            for (int i = 0; i < limit; i++)
            {
                current.add(Integer.parseInt(tokens[i]));
            }
            adj.add(current);
        }
    }
    catch (IOException x)
    { throw new Error("bad input stream"); }
}
```

Our stream constructor reads in an integer denoting the order n of the graph and then reads in n lines denoting the adjacency lists. Notice that we *do not* check for correctness of the data. For example, a graph of 5 vertices could have erroneous adjacency lists with numbers outside the range 0 to 4. We leave these robustness considerations for an extended class to fulfil, if desired. Also note that we do not list the vertex index in front of the individual lists and we use white space to separate items. A blank line indicates an empty list (that is, no neighbours) for a vertex.

```
//  Mutator Methods
//
public void addVertices(int n)
{
    assert(0 <= n);
    if ( n > 0 )
    {
        for (int i = 0; i < n; i++)
        {
            adj.add(new ArrayList<Integer>());
        }
    }
}

public void removeVertex(int i)
{
    assert(0 <= i && i < order());
    adj.remove(i);
    Integer I = new Integer(i);
    for (int u = 0; u < order(); u++)
    {
        ArrayList<Integer> current = adj.get(u);
        current.remove(I); // remove i from adj lists
        for (Integer num: current)
        {
            if (num > i)  // relabel larger indexed nodes
            {
                int index = current.indexOf(num);
                current.set(index, num-1);
            }
        }
    }
```

```
}
```

Adding vertices is easy for our adjacency lists representation. Here we just expand the internal `_adj` list by appending new empty lists. The `removeVertex` method is a little complicated in that we have to scan each list to remove arcs pointing to the vertex being deleted. We also have chosen to relabel vertices so that there are no gaps (that is, we want vertex indexed by i to be labeled `Integer(i)`). A good question would be to find a more efficient `removeVertex` method. One way would be to also keep an in-neighbour list for each vertex. However, the extra data structure overhead is not desirable for our simple implementation.

```
public void addArc(int i, int j)
{
    assert(0 <= i && i < order());
    assert(0 <= j && j < order());
    if (isArc(i,j)) return;
    (adj.get(i)).add(j);
}

public void removeArc(int i, int j)
{
    assert(0 <= i && i < order());
    assert(0 <= j && j < order());
    if (!isArc(i,j)) return;
    (adj.get(i)).remove(new Integer(j));
}

public void addEdge(int i, int j)
{
    addArc(i,j);
    addArc(j,i);
}

public void removeEdge(int i, int j)
{
    removeArc(i,j);
    removeArc(j,i);
}
```

Adding and removing arcs is easy since the methods to do this exist in the `Vector` class. All we have to do is access the appropriate adjacency list. We have decided to place a safeguard in the `addArc` method to prevent parallel arcs from being added between two vertices.

```
//  Access Methods
//
public boolean isArc(int i, int j)
{
    assert(0 <= i && i < order());
    assert(0 <= j && j < order());
```

```
    return (adj.get(i)).contains(new Integer(j));
}

public boolean isEdge(int i, int j)
{
    return isArc(i,j) && isArc(j,i);
}

public int inDegree(int i)
{
    assert(0 <= i && i < order());
    int sz = 0;
    for (int j = 0; j < order(); j++)
    {
        if (isArc(j,i))  sz++;
    }
    return sz;
}

public int degree(int i)
{
    assert(0 <= i && i < order());
    return (adj.get(i)).size();
}
```

Note how we assume that the `contains` method of a `Vector` object does a data equality check and not just a reference check. The `outDegree` method probably runs in constant time since we just return the list's size. However, the `inDegree` method has to check all adjacency lists and could have to inspect all arcs of the graph/digraph.

```
public ArrayList<Integer> neighbours(int i)
{
    assert(0 <= i && i < order());
    ArrayList<Integer> nei = new ArrayList<Integer>();
    for (Integer vert : adj.get(i))
    {
        nei.add(vert);
    }
    return nei;
    //return (ArrayList<Integer>)(adj.get(i)).clone();
}

public int order()
{
    return adj.size();
}

public int size()    // Number of arcs (counts edges twice)
{
    int sz = 0;
    for (int i=0; i<order(); i++)
```

```
        {
            sz += (adj.get(i)).size();
        }
        return sz;
    }
```

We do not want to have any internal references to the graph data structure being available to non-class members. Thus, we elected to return a clone of the adjacency list for our `neighbours` method. We did not want to keep redundant data so the order of our graph is simply the size of the `adj` list.

```
// default output readable by constructor
//
public String toString() { return toStringAdjLists(); }

} // end class GraphAdjLists
```

Again, we have the default output format for this class be compatible with the constructor `BufferedReader`. (The method `toStringAdjLists` is defined on page 180.)

B.3 Standardized Java graph class

We now have two implementations of a graph class as specified by our interface (abstract class) `Graph`. We want to write algorithms that can handle either format. Since Java is object-oriented we could have all our algorithms take a `Graph` object and the run-time dynamic mechanism should ascertain the correct adjacency matrix or adjacency lists methods. For example, we could write a graph coloring algorithm prototyped as `public int color(Graph G)` and pass it either a `GraphAdjMatrix` or a `GraphAdjLists`. And it should work fine!

We next present a simple test program for how one would use our graph implementations. We encourage the reader to trace through the steps and to try to obtain the same output.

```
import java.io.*; import graphADT.*;

public class test {

public static void main(String argv[])
{
   Graph G1 = new GraphAdjLists();

   G1.addVertices(5);
   G1.addArc(0,2); G1.addArc(0,3); G1.addEdge(1,2);
   G1.addArc(2,3); G1.addArc(2,0); G1.addArc(2,4);
   G1.addArc(3,2); G1.addEdge(4,1); G1.addArc(4,2);

   System.out.println(G1);
```

```
   Graph G2 = new GraphAdjMatrix(G1);

   G2.removeArc(2,0); G2.removeArc(4,1); G2.removeArc(2,3);

   System.out.println(G2);

   Graph G3 = new GraphAdjLists(G2);

   G3.addVertices(2);
   G3.addArc(5,4); G3.addArc(5,2); G3.addArc(5,6);
   G3.addArc(2,6); G3.addArc(0,6); G3.addArc(6,0);

   System.out.println(G3);

   Graph G4 = new GraphAdjLists(G3);

   G4.removeVertex(4); G4.removeEdge(5,0); G4.addVertices(1);
   G4.addEdge(1,6);

   System.out.println(G4);
}
} // test
```

The expected output, using JDK version 1.6, is given in Figure B.1. Note that the last version of the digraph G has a vertex of out-degree zero in the adjacency lists. (To compile our program we type 'javac test.java' and to execute it we type 'java test' at our command-line prompt '$'.)

B.4 Extended graph classes: weighted edges

The graph ADT presented in the previous sections can be easily extended to provide a customized data type. For example, if one only wants undirected graphs then a more restrictive class can be developed to prevent arc operations. In this section we want to illustrate how one can develop an ADT for arc-weighted graphs. We first want to define a new graph interface that allows for these weights.

```
public interface wGraph extends Graph
{
    class Weight<X>
    {
        private X value;
        public Weight(X arg)
        {
            value = arg;
        }
        public X getValue()
        {
            return value;
        }
        public void setValue(X arg)
        {
```

```
$ javac test.java
$ java test
5
2 3
2 4
1 3 0 4
2
1 2

5
0 0 1 1 0
0 0 1 0 1
0 1 0 0 1
0 0 1 0 0
0 0 1 0 0

7
2 3 6
2 4
1 4 6
2
2
4 2 6
0

7
2 3
2 6
1 5
2
2 5

1
```

Figure B.1: Sample output of the graph `test` program.

```
            value = arg;
        }
        public String toString()
        {
            return value.toString();
        }
    }

    void addArc(int i, int j);  // overridden with "default weight"
    void addArc(int i, int j, Weight weight);

    void setArcWeight(int i, int j, Weight weight);
    //assumes edge i-j exists; and replaces the weight of edge i-j

    Weight getArcWeight(int i, int j);

    ArrayList<Weight> neighbourWeights(int i);
    // If you call neighbours(i) and neighbourWeights(i) then
    // the k-th element of both lists are correlated
}
```

In the above interface, we define a class to represent arbitrary weights. Usually one uses `Weight<Integer>` as the arc attributes. Since `wGraph` extends `Graph`, the existing graph algorithms, written for non-weighted graphs, should still work.

We can implement a `wGraph` in several ways just like we had `GraphAdjMatrix` and `GraphAdjLists` for the interface `Graph`. We can then create a `wGraph` object by creating an implementation object, such as:

```
wGraph wG = new wGraphMatrix(Buffer);
```

An adjacency matrix implementation for this interface is given below. Note that the `BufferedReader` constructor assumes weights of type `Integer`. If one wants a floating point data type then another constructor (or creation method) is required.

```
package graphADT;

import java.util.ArrayList; import java.io.*;

public class wGraphMatrix implements wGraph
{
    protected int order;
    protected Weight[][] adjW;  // null entry means no arc

    public wGraphMatrix()
    {
       order = 0;
    }

    public wGraphMatrix(wGraphMatrix G)
    {
        int n = order = G.order();
        if ( n > 0 )
```

```
    {
        adjW = new Weight[n][n];
    }

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            adjW[i][j] = G.adjW[i][j];
        }
    }
}

public wGraphMatrix(wGraph G) //convert implementation
{
    int n = order = G.order();
    adjW = new Weight[n][n];

    for (int i = 0; i < n; i++)
    {
        ArrayList<Integer> nbrs = G.neighbours(i);
        ArrayList<Weight> wNbrs = G.neighbourWeights(i);
        for (int j = 0; j < nbrs.size(); j++)
        {
            int index = nbrs.get(j);
            adjW[i][index] = wNbrs.get(j);
        }
    }
}

public wGraphMatrix(Graph G) // promote and/or copy
{
    int n = order = G.order();
    if ( n > 0 )
    {
        adjW = new Weight[n][n];
    }

    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            if (G.isArc(i, j))
            {
                adjW[i][j] = new Weight<Integer>(1);
            }
        }
    }
}

public wGraphMatrix(BufferedReader buffer)
{
    try
```

```
    {
        String line = buffer.readLine().trim();
        String[] tokens = line.split("\\s+");

        if (tokens.length != 1)
        {
            throw new Error("bad format: number of vertices");
        }
        int n = order = Integer.parseInt(tokens[0]);

        if ( n > 0 )
        {
            adjW = new Weight[n][n];
        }

        for (int i = 0; i < n; i++)
        {
            line = buffer.readLine().trim();
            tokens = line.split("\\s+");
            if (tokens.length != n)
            {
              throw new Error("bad format: adjacency matrix");
            }

            for (int j = 0; j < n; j++)
            {
                int entry = Integer.parseInt(tokens[j]);
                if (entry != 0)
                {
                  adjW[i][j] = new Weight<Integer>(entry);
                }
            }
        }
    }
    catch (IOException x)
    {
        throw new Error("bad input stream");
    }

}

// mutator methods

public void addVertices(int n)
{
    assert(0 <= n );
    Weight weights[][] = new Weight[order+n][order+n];

    for (int i = 0; i < order; i++)
    {
        for (int j = 0; j < order; j++)
        {
        weights[i][j] = adjW[i][j];
```

```
        }
    }
    order += n;
    adjW = weights;
}

public void removeVertex(int v)
{
    assert(0 <= v && v < order);
    order--;

    for (int i = 0; i < v; i++)
    {
        for (int j = v; j < order; j++)
        {
            adjW[i][j] = adjW[i][j+1];
        }
    }

    for (int i = v; i < order; i++)
    {
        for (int j = 0; j < v; j++)
        {
            adjW[i][j] = adjW[i+1][j];
        }
        for (int j = v; j < order; j++)
        {
            adjW[i][j] = adjW[i+1][j+1];
        }
    }
}

public void addArc(int i, int j)
{
    assert(0 <= i && i < order());
    assert(0 <= j && j < order());
    adjW[i][j] = new Weight<Integer>(1); //default weight
}

public void removeArc(int i, int j)
{
    assert(0 <= i && i < order());
    assert(0 <= j && j < order());
    adjW[i][j] = null;
}

public void addEdge(int i, int j)
{
    addArc(i,j); addArc(j,i);
}

public void removeEdge(int i, int j)
{
```

```
    removeArc(i,j); removeArc(j,i);
}

public void addArc(int i, int j, Weight weight)
{
    assert(0 <= i && i < order());
    assert(0 <= j && j < order());
    adjW[i][j] = weight;
}

public void setArcWeight(int i, int j, Weight weight)
{
    assert(isArc(i, j));
    adjW[i][j] = weight;
}

public Weight<?> getArcWeight(int i, int j)
{
    assert(isArc(i, j));
    return adjW[i][j];
}

// accessor methods

public boolean isArc(int i, int j)
{
    assert(0 <= i && i < order);
    assert(0 <= j && j < order);
    return adjW[i][j] != null;
}

public boolean isEdge(int i, int j)
{
    return isArc(i,j) && isArc(j,i);
}

public int inDegree(int i) // column count
{
    assert(0 <= i && i < order);
    int sz = 0;
    for (int j = 0; j < order; j++)
    {
        if (adjW[j][i] != null) sz++;
    }
    return sz;
}

public int degree(int i) // row count
{
    assert(0 <= i && i < order);
    int sz = 0;
    for (int j = 0; j < order; j++)
    {
```

```
        if (adjW[i][j] != null) sz++;
    }
    return sz;
}

public int order()
{
    return order;
}

public int size()  // Number of arcs (edges count twice)
{
    int sz = 0;
    for (int i = 0; i< order; i++)
    {
        for (int j = 0; j< order; j++)
        {
            if (adjW[i][j] != null) sz++;
        }
    }
    return sz;  // undirected ? sz / 2 : sz;
}

public ArrayList<Integer> neighbours(int i)
{
    assert(0 <= i && i < order);
    ArrayList<Integer> nbrs = new ArrayList<Integer>();

    for (int j = 0; j < order; j++)
    {
        if (adjW[i][j] != null) nbrs.add(j);
    }
    return nbrs;
}

public ArrayList<Weight> neighbourWeights(int i)
{
    ArrayList<Weight> nbrsWei = new ArrayList<Weight>();

    for (int j = 0; j < order(); j++)
    {
        if (adjW[i][j] != null)
        {
           nbrsWei.add(adjW[i][j]); // corresponding weight
        }
    }
    return nbrsWei;
}

public String toString()  // print weights in n-by-n matrix
{
    StringBuffer o = new StringBuffer();
    o.append(order()+"\n");
```

```
        for (int i = 0; i < order(); i++)
        {
            for (int j = 0; j < order(); j++)
            {
                if (adjW[i][j] != null)
                {
                    o.append(adjW[i][j] + " ");
                }
                else
                {
                    o.append(0 + " ");
                }
            }
            o.append("\n");
        }
        return o.toString();
    }

}
```

One thing to note is that if one wants to output the underlying graph representation (that is, without weights) one can simply call the `toString` method of `Graph` reference.

We conclude by mentioning that the details for an adjacency lists implementation, `wGraphLists`, are included in the graph library accompanying this book. We note that this adjacency lists version of the ADT is more suitable when one expects weights of numerical value 0 or has sparse graphs.

Appendix C

Background on Data Structures

We assume that the reader is familiar with basic data structures such as arrays and with the basic data types built in to most programming languages (such as integer, floating point, string, etc). Many programming applications require the programmer to create complicated combinations of the built-in structures. Some languages make this easy by allowing the user to define new data types (for example Java or C++ classes), and others do not (for example C, Fortran). These new data types are concrete implementations in the given language of ***abstract data type***s (***ADT***s), which are mathematically specified.

C.1 Informal discussion of ADTs

An ADT consists of a set with certain operations on it. How those operations are to be carried out is not our concern here. It is up to the programmer to choose an implementation that suits the given application.

Some of the key ADTs are: list, stack, queue, priority queue, dictionary, disjoint sets. We discuss them each below in turn, semi-formally.

A ***container*** is a collection of objects from some universal set U. Basic operations are to create a new empty container, insert an element, check whether the container is empty (the `isEmpty` operation). We assume that each element when inserted returns a locator that identifies its position uniquely.

We can then try to `find` an element. Depending on the additional operations defined, this may be easy or difficult. We may need to enumerate all locators. We can also find the `size` of a container by enumerating all locators. Again, this may be very inefficient, and for certain applications a special size operation may be defined.

Similarly, many container ADTs have a `delete` operation. Some of these allow quick removal of an element, while others have to find it first, which may be slower.

Other operations can be expressed in terms of the basic ones. For example, we can sometimes modify or update an element by finding it, remembering its locator, deleting it, then inserting the new value at the given location. This procedure is sometimes very inefficient, so a special update operation may be required in some situations. We normally try to have as few basic operations as possible, and other operations such as sorting are expressed in terms of these (this is the aim of "generic programming").

As we see a container is very general. Some of the important container ADTs are listed below.

A ***list*** is a container that stores elements in a linear sequence. Some basic operations are to `insert` in a given position in the sequence, to `delete` an element at a given position. To find an element requires sequential search, enumerating all locators until the element is found or we run out of locators. The first element of a list is called the ***head*** and the last is the ***tail***. A ***sublist*** is a contiguous piece of the list, that can be traversed by the iterator with no gaps. If we divide the list into two sublists by choosing an element x and letting the head sublist consist of all elements before x, and the tail sublist consist of all elements after it (either sublist, or neither, may contain x, depending on the situation).

The main data structures used for implementing a list are arrays and singly- and doubly-linked lists. They have different properties. For example, to find the middle element of a list implemented as an array is a constant-time operation but this is not true for the linked lists, since one must traverse the list, not having direct access to the middle element as in an array. On the other hand, to insert an element in a given (nonempty) position in an array takes longer than with the linked list implementation.

A ***sorted list*** is a container of elements from a totally ordered set U, with the same basic operations as a list, except that the elements are kept sorted in ascending order.

A ***stack*** is a restricted kind of list in which insertion and deletion occur at the same end ("top") of the list, and at no other position. The basic operations are `insert` (also called `push`), `delete` (also called `pop`), and `getTop` (also called `peek`) which returns the element at the top of the list without removing it.

A ***queue*** is a restricted kind of list in which insertion occurs at one end (the "tail") and deletion occurs at the other end (the "head"). The basic operations are `insert` (also called `enqueue` or `push`), `delete` (also called `dequeue` or `pop`), and `getHead` (also called `peek`) which returns the element at the head without removing it.

A ***priority queue*** is a container of elements from a totally ordered set U that allows us to `insert` an element, to find the smallest element with `peek`, and to remove the smallest element with `delete`. A more advanced operation is `decreaseKey` which finds and makes an element smaller. A general delete operation is not usually defined.

A priority queue can be well implemented using a binary heap, if the operation `decreaseKey` is not required to be particularly efficient. Otherwise, more sophisticated implementations are normally used.

A ***dictionary*** (also called ***table***, ***associative array*** or ***map***) is a a container with basic operations `find`, `insert`, and `delete`. It is usually also desired to perform an `update` operation many times, since dictionaries are often used for dynamic databases.

Some of these ADTs can be used to simulate other ones. For example, a dictionary can be implemented using a list. Insertion occurs after the last element, and finding is via sequential search. For practical situations where a dictionary is used, the `find` operation is used a lot, so such an inefficient implementation would not normally be used. Similarly, one can use a list to simulate a priority queue. Insertion occurs at one end and finding the maximum element by sequential search. Or we could use a sorted list, where insertion occurs at the correct point and removing the minimum element is particularly easy, since it is just removing the first element.

C.2 Notes on a more formal approach

The discussion above still doesn't define the various ADTs in a completely satisfactory way. As in abstract algebra, we must specify not only the basic operations but also their properties using a set of axioms. It can be quite difficult to do this succinctly. Also, whereas in the case of algebra the basic structures (group, ring, field ...) have been agreed on for many decades, the axiomatic definitions of the main ADTs do not seem to be completely standardized yet. So we shall not give a complete axiomatic presentation, but limit ourselves to an example.

The stack ADT could be defined as follows.

A stack on a set U is a set S with operations `push`, `pop`, `peek`, `isEmpty`. There is an empty stack called ε which corresponds to S being the empty set. These operations do the following: `push` takes an ordered pair consisting of a stack and an element of U as an argument, and returns a stack; `pop` takes a stack as an argument and returns an element of U; `peek` takes a stack as an argument and returns either an element of u or "ERROR"; `isEmpty` returns either 0 (false) or 1 (true). The axioms for every stack S and element x of U are as follows.

- `isEmpty`(ε) = 1
- `pop`(`push`(S, x)) = S
- `peek`(`push`(S, x)) = x

Appendix D

Mathematical Background

We collect here some basic useful facts, all of which can be found in standard textbooks on calculus and discrete mathematics, to which the reader should refer for proofs.

D.1 Sets

A ***set*** is a collection of distinguishable objects (its ***elements***) whose order is unimportant. Two sets are the same if and only if they have the same elements. We denote the statement that x is an element of the set X by $x \in X$ and the negation of this statement by $x \notin X$. We can list finite sets using the braces notation: for example, the set S consisting of all integers between 1 and 10 that are divisible by 3 is denoted $\{3,6,9\}$. A ***subset*** of a set X is a set all of whose elements are also elements of X. Each set has precisely one subset with zero elements, the ***empty set*** which is denoted $\emptyset$. A subset can be described by set-builder notation; for example, the subset of S consisting of all multiples of 3 between 1 and 7 can be written $\{s \mid s \in S \text{ and } s \leq 7\}$.

For sets X and Y, the ***union*** and ***intersection*** of X and Y are, respectively, the sets defined as follows (note that the "or" is inclusive, so "P or Q" is true if and only if P is true, Q is true, or both P and Q are true):

$$X \cup Y = \{x \mid x \in X \text{ or } x \in Y\}$$
$$X \cap Y = \{x \mid x \in X \text{ and } x \in Y\}.$$

The ***set difference*** $X \setminus Y$ is defined by $X \setminus Y = \{x \mid x \in X \text{ and } x \notin Y\}$. The ***complement*** of a subset S of X is the subset $\overline{S} = X \setminus S$ of S. The number of elements (often called its ***cardinality***) of a set X is often denoted by $|X|$.

D.2 Mathematical induction

A common way of proving that a statement $P(n)$ is true for all integers n after some threshold n_0 is as follows. It is called the ***principle of mathematical induction*** and is equivalent to the fact that there are no infinite chains of nonnegative integers $x_1 > x_2 > \dots$.

The principle of mathematical induction states that *if*

- $P(n_0)$ is true *and*
- *for each* $n \geq n_0$, *if* $P(n)$ is true *then* $P(n+1)$ is true

then $P(n)$ is true for all $n \geq n_0$.

For example, suppose we wish to prove that there exists some constant $c > 0$ such that $(n+1)^4/n^4 \leq c$ for all $n \geq 1$ (see below for motivation). We notice that when $n = 1$ the ratio is 16 and that it appears to get smaller with increasing n (the next few values of the ratio are $81/16 \cong 5$ and $256/81 \cong 3$). So we may guess that $c = 16$ will work (or any larger number). So we have that $n_0 = 1$, $P(n)$ is the statement "$(n+1)^4/n^4 \leq 16$ for all $n \geq 1$".

We already know that $P(n_0)$ is true. Now suppose that $P(n)$ is true for some $n \geq n_0$. We need to show that $P(n+1)$ is true. We know that

$$(n+1)^4 \leq 16n^4$$

and wish to show that

$$(n+2)^4 \leq 16(n+1)^4.$$

Taking 4th roots we have $n+1 \leq 2n$ and we want $n+2 \leq 2(n+1)$. Now using the fact that $P(n)$ is true (this is called the ***inductive hypothesis***), we have $n+2 = n+1+1 \leq 2n+1 < 2(n+1)$ so the result follows. Thus $P(n)$ is true for all n by the principle of mathematical induction.

An alternative form of the principle of mathematical induction, which is equivalent to it, is called ***complete induction***:

- $P(n_0)$ is true *and*
- *for each* $n \geq n_0$, *if* $P(i)$ is true for each i with $n_0 \leq i \leq n$, *then* $P(n+1)$ is true

then $P(n)$ is true for all $n \geq n_0$.

D.3 Relations

A ***relation*** on a set S is a set R of ordered pairs of elements of S, that is, a subset of $S \times S$. If (x,y) is a such a pair, we say that x is related to y and sometimes write xRy. An example is the relation of divisibility on the positive integers; xRy if and only if y is a multiple of x. Here $2R12$, $1Rx$ for every x, and $xR1$ only if $x = 1$.

There are some special types of relations that are important for our purposes.

An ***equivalence relation*** is a relation that is ***reflexive, symmetric*** and ***transitive***. That is, we have for every $x, y, z \in S$

- xRx
- if xRy then yRx
- if xRy and yRz then xRz

An equivalence relation amounts to the same thing as a ***partition***: a decomposition of S as a union of disjoint subsets. Each subset consists of all elements that are related to any one of them, and no elements in different subsets of the partition are related.

Examples of equivalence relations: "having the same mother" on the set of all humans; "being divisible by 7" on the set of all positive integers; "being mutually reachable via a path in a given graph".

Another important type of relation is a ***partial order***. This is a relation that is reflexive, ***antisymmetric*** and transitive. Antisymmetry means that if xRy and yRx then $x = y$. Examples are: xRy if and only if x is a factor of y , where x and y are positive integers.

A ***linear order*** or ***total order*** is a partial order where every pair of elements is related. For example, the usual relation $\leq$ on the real numbers. The elements of a finite set with a total order can be arranged in a line so that each is related to the next and none is related to any preceding element.

D.4 Basic rules of logarithms

For $x > 1, y > 0$, the logarithm $\log_x y$ to base x of y satisfies the equality: $x^{\log_x y} = y$ and has the following properties:

- $\log_x(a \cdot b) = \log_x a + \log_x b$
- $\log_x(a/b) = \log_x a - \log_x b$
- $\log_x(a^b) = b \log_x a$

Using the definition and the properties of the logarithm, it is easy to show that the following rules hold for $x > 1, y > 0, z > 0$.

$$\begin{aligned} \log_x y &= \frac{1}{\log_y x} \\ \log_x z &= \frac{\log_y z}{\log_y x} \\ z^{\log_x y} &= y^{\log_x z} \end{aligned}$$

The last rule is easily proven by taking logarithm to base x of each side of the equality.

The notation $\log_e = \ln$ is commonly used, as also is $\log_2 = \lg$.

Often we want to convert the real values returned from functions like logarithms to integers. Let x be a real number. The ***floor*** $\lfloor x \rfloor$ of x is the greatest integer not greater than x, and the ***ceiling*** $\lceil x \rceil$ of x is the least integer not less than x. If x is an integer, then $\lfloor x \rfloor = x = \lceil x \rceil$.

D.5 L'Hôpital's rule

This rule was in fact proved by J. Bernoulli and sold to G. de L'Hôpital for inclusion in the first calculus textbook, published in 1696. The form that we need for asymptotics is as follows.

Theorem D.1. If $\lim_{x\to\infty} f(x) = \infty = \lim_{x\to\infty} g(x)$ and f, g are positive differentiable functions for $x > 0$, then

$$\lim_{x\to\infty} f(x)/g(x) = \lim_{x\to\infty} f'(x)/g'(x).$$

As an example, to compute the limit of e^x/x^3 as $x \to \infty$, we use the rule repeatedly:

$$\lim_{x\to\infty} \frac{e^x}{x^3} = \lim_{x\to\infty} \frac{e^x}{3x^2} = \lim_{x\to\infty} \frac{e^x}{6x} = \lim_{x\to\infty} \frac{e^x}{6} = \infty.$$

D.6 Arithmetic, geometric, and other series

A general arithmetic series is specified by the recurrence $a_n = a_{n-1} + c$, where c is a constant. The sum of its n terms is

$$a_1 + a_2 + \ldots + a_n = \frac{n}{2}(a_1 + a_n) = na_1 + c\frac{n(n-1)}{2}.$$

When $a_1 = 1$ and $c = 1$,

$$1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

A general geometric series is specified by the recurrence $a_n = ca_{n-1}$, where $c \neq 1$ is a constant. The sum of its n terms is

$$a_1 + a_2 + \ldots + a_n = a_1\frac{c^n - 1}{c - 1}$$

If $0 < c < 1$, it is better to rewrite the sum as

$$a_1 + a_2 + \ldots + a_n = a_1\frac{1 - c^n}{1 - c}$$

When n goes to infinity, the sum of the latter infinite geometric series is

$$\sum_{k=1}^{\infty} a_k = \frac{a_1}{1 - c}$$

provided $|c| < 1$, otherwise the infinite sum does not exist.

The sum of squares has an easily guessed explicit formula

$$\sum_{i=1}^{n} i^2 = 1 + 2^2 + 3^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

which can be proved by induction. Similar explicit formulae hold for the sum $\sum_{i=1}^{n} i^p$ where p is a fixed positive integer, but they become rather complicated. More useful for our purposes is the asymptotic formula for large n

$$\sum_{i=1}^{n} i^p \in \Theta(n^{p+1})$$

which also holds for negative integers p provided $p \neq -1$. When $p = -1$ we have an asymptotic statement about the ***harmonic numbers*** $H_n = \sum_{i=1}^{n} i^{-1}$,

$$\sum_{i=1}^{n} 1/i \in \Theta(\log n).$$

A similar sum of interest to us is

$$\log(n!) = \sum_{i=1}^{n} \log i.$$

This is in $\Theta(n \log n)$ for any base of the logarithm.

A more complicated formula (***Stirling's approximation***), which we do not prove here, is

$$n! > n^n e^{-n} \sqrt{2\pi n}$$

and in fact, as $n \to \infty$, we have

$$n! \approx n^n e^{-n} \sqrt{2\pi n},$$

where $f \approx g$ is a stronger result than $f \in \Theta(g)$, namely $\lim_{n\to\infty} f(n)/g(n) = 1$.

The above asymptotic results can all be proved in the following way using integral calculus. The method works for any continuous function that is increasing or decreasing (a monotone function) for $x > 0$. For example, consider $f(x) = x^3$ which is increasing. Then we may approximate the integral $\int_0^n f(x)\,dx$ from below by the sum of rectangles with base length 1 and height $f(i)$, the sum going from 0 to $n-1$. See Figure D.1. Similarly we can approximate it from above by the same sum from 1 to n. This gives after rearrangement:

$$\int_1^n x^3\,dx \leq \sum_{i=1}^{n} i^3 \leq \int_1^{n+1} x^3\,dx$$

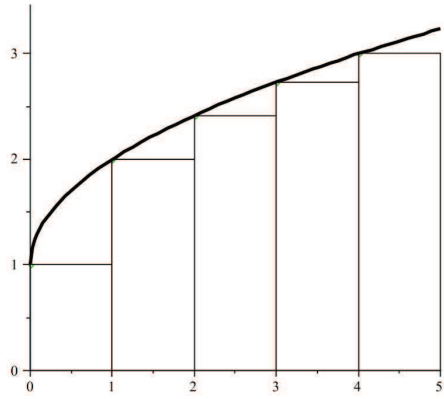


Figure D.1: Approximation of an integral by lower rectangles.

and so we have

$$\frac{n^4-1}{4} \leq \sum_{i=1}^{n} i^3 \leq \frac{(n+1)^4-1}{4}.$$

This easily yields that $\sum_{i=1}^{n} i^3$ is $\Theta(n^4)$ since $(n+1)^4/n^4 \leq 16$ for $n \geq 1$ and $n^4 - 1 \geq 15n^4/16$ for $n \geq 2$.

D.7 Trees

A ***rooted ordered tree*** is what computer scientists call simply a "tree". These trees are defined recursively. An ordered rooted tree is either a single node or a distinguished node (the ***root***) attached to some ordered rooted trees given in a fixed order (hence such a tree is defined recursively). In a picture, these subtrees are usually drawn from left to right below the parent node. The ***parent*** of a node is defined as follows. The root has no parent. Otherwise the node was attached in some recursive call, and the root it was attached to at that time is its parent. The roots of the subtrees in the definition are the ***children*** of the root. A rooted ordered tree can be thought of as a digraph in which there is an arc from each node to each of its children.

A node with no children is called a ***leaf***. The ***depth*** of a node is the distance from the root to that node (the length of the unique path between them). The ***height of a node*** is the length of a longest path from the node to a leaf. The ***height*** of tree is the height of the root. Note that a tree with a single node has height zero. Some other books use a definition of height whose value equals the value given by our definition, plus one.

A ***binary tree*** is an ordered rooted tree where the number of children of each node is always $0, 1$, or 2.

A ***free tree*** (what mathematicians call a tree) has no order (so a mirror image of a picture of a tree is a picture of the same tree) and no distinguished root. Every free tree can be given a root arbitrarily (in n ways, if the number of nodes is n), and ordered in many different ways.

A free tree can be thought of as the underlying graph of an ordered rooted tree. A free tree is a very special type of graph. First, if n is the number of nodes and e the number of edges, then $e = n - 1$. To see this, note that in the underlying graph of an ordered rooted tree, each edge connects a node with its parent. Each node except one has a parent. Thus there is a one-to-one correspondence between nodes other than the root and edges, yielding the result.

One can easily show that the following are equivalent for a graph G:

- G is a free tree.
- G is a connected graph with $e = n - 1$.
- G is an acyclic graph with $e = n - 1$.

Appendix E

Solutions to Selected Exercises

SOLUTION TO EXERCISE 1.1.1 ON PAGE 19:
The equation $T(n) = cn^2$ has only one unknown, c, to be found from $T(10) = c \times 10^2 = 500$, so $c = 500/100 = 5$. Then $T(1000) = 5 \times (1000)^2 = 5 \times 10^6$, that is, the algorithm takes 5 million elementary operations to process 1000 data items.

In fact we do not need to compute c. We first compute $T(1000)/T(10)$ which equals $10^6 c/100c = 10^4$. Thus the answer is 500×10^4, or 5 million.

SOLUTION TO EXERCISE 1.1.2 ON PAGE 19:
As above, the constants c_A and c_B have to be found in order to work out how many elementary operations each algorithm takes with $n = 2^{20}$ and find the fastest algorithm for processing 2^{20} data items. For $n = 2^{10}$, $T_A(2^{10}) = c_A \times 2^{10} \lg(2^{10}) = 10$, so $c_A \times 2^{10} \times 10 = 10$, or $c_A = 1/2^{10} = 2^{-10}$, and $T_B(2^{10}) = c_B \times (2^{10})^2 = 1$, so $c_B = 2^{-20}$. Hence,

$$\begin{array}{rcrcrcl} T_A(2^{20}) & = & 2^{-10} \times 2^{20} \times \lg(2^{20}) & = & 2^{10} \times 20 & < & 2^{15} \\ T_B(2^{20}) & = & 2^{-20} \times (2^{20})^2 & = & 2^{20} & & \end{array}$$

and $T_A(2^{20}) < T_B(2^{20})$, so the algorithm A processes 2^{20} data items the fastest.

SOLUTION TO EXERCISE 1.2.1 ON PAGE 21:
The running time is linear because when $j = m = 1$ the assignment statement makes m equal to $n-1$. Then when $j = n-1$, the assignment statement makes m equal to $(n-1)^2$. As the inner loop runs once every time $j = m$, it runs in total only two times and does n operations each loop. The outer loop runs n times. Let c_i be the constant number of elementary operations in the inner loop, and let c_o be the number of el-

ementary operations in the outer loop other than the operations in the inner loop. Then $T(n) = c_o n + 2c_i n \in O(n)$.

SOLUTION TO EXERCISE 1.3.1 ON PAGE 27:
We have $10n^3 - 5n + 15 \in O(n^2)$ if and only if (iff) there exist a positive real constant c and a positive integer n_0 such that the inequality $10n^3 - 5n + 15 \leq cn^2$ holds for all $n \geq n_0$. Reducing it to $n - 0.5n^{-1} + 1.5n^{-2} \leq 0.1c$ shows that for any value of c this inequality does not hold true for all $n > k+1$, where k is the closest integer greater than $0.1c$. Therefore, $10n^3 - 5n + 15 \notin O(n^2)$.

SOLUTION TO EXERCISE 1.3.2 ON PAGE 27:
As above, $10n^3 - 5n + 15 \in \Theta(n^3)$ iff there exist positive real constants c_1 and c_2 and a positive integer n_0 such that the inequalities $c_1 n^3 \leq 10n^3 - 5n + 15 \leq c_2 n^3$, or what is the same, $c_1 \leq 10 - 5n^{-2} + 15n^{-3} \leq c_2$, hold true for all $n \geq n_0$. We know that $\lim_{n\to\infty}(10 - 5n^{-2} + 15n^{-3}) = 10$, so there always will be a value $n_0 > 3$ such that for $n > n_0$, $c_1 \leq 10 - 5n^{-2} + 15n^{-3} \leq c_2$ where $c_1 \leq 10 - \varepsilon$ and $c_2 \geq 10 - \varepsilon$ where $\varepsilon = 5n_0^{-2} - 15n_0^{-3} > 0$, say $c_1 = 1$ and $c_2 = 20$. Therefore, $10n^3 - 5n + 15 \in \Theta(n^3)$.

Note that Lemma 1.19, the Limit Rule, can be used instead. In this case $f(n) = 10n^3 - 5n + 15$; $g(n) = n^3$, and $f(n) \in \Theta(g(n))$ because $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 10$.

SOLUTION TO EXERCISE 1.3.3 ON PAGE 27:
As above, $10n^3 - 5n + 15 \in \Omega(n^4)$ iff there exist a positive real constant c and a positive integer n_0 such that the inequality $10n^3 - 5n + 15 \geq cn^4$ holds for all $n > n_0$. We need to show that for any value of c and n_0 this inequality, or what is the same, the reduced one, $10n^{-1} - 5n^{-3} + 15n^{-4} \geq c$, does not hold true for all $n > n_0$. We know $\lim_{n\to\infty}(10n^{-1} - 5n^{-3} + 15n - 4) = 0$, so no matter which values c and n_0 are picked, the inequality cannot be true for all $n > n_0$. Therefore, $10n^3 - 5n + 15 \notin \Omega(n^4)$.

SOLUTION TO EXERCISE 1.3.5 ON PAGE 27:
To show that each function $f(n)$ in Table 1.2 stands in "Big Oh" relation to the preceding one, $g(n)$, that is, $f(n) \in O(g(n))$, it is sufficient to use the Limit Rule (Lemma 1.19) and show that $\lim_{n\to\infty} f(n)/g(n) = 0$:

$n \in O(n \log n)$ because $n/(n \log n) = (\log n)^{-1}$ and $\lim_{n\to\infty}(\log n)^{-1} = 0$;

$n \log n \in O(n^{1.5})$ because $n \log n / n^{1.5} = \log n / n^{0.5}$ and any positive power of n grows faster than any logarithm (Example 1.14): $\lim_{n\to\infty} \log n / n^{0.5} = 0$;

$n^{1.5} \in O(n^2)$ and $n^2 \in O(n^3)$ because higher powers of n grow faster than lower powers (Example 1.20);

$n^3 \in O(2^n)$ because exponential functions with base greater than 1 grow faster than any positive power of n (Example 1.13): so $\lim_{n\to\infty} n^3/2^n = 0$.

SOLUTION TO EXERCISE 1.3.6 ON PAGE 27:
Lemma 1.16

Proof. It follows from $h(n) \in O(g(n))$ and $g(n) \in O(f(n))$ that $h(n) \leq c_1 g(n)$ for all $n > n_1$ and $g(n) \leq c_2 f(n)$ for all $n > n_2$ where c_1 and c_2 are positive real constants and n_1 and n_2 are positive integers. Then $h(n) \leq c_1 g(n) \leq c_1 c_2 f(n)$ for $n \geq \max\{n_1, n_2\}$, so the relationship $h(n) \leq c f(n)$ for all $n \geq n_0$ is also true for $c = c_1 c_2$ and $n_0 = \max\{n_1, n_2\}$. Therefore "Big Oh" is transitive. □

Lemma 1.17

Proof. It follows from $g_1(n) \in O(f_1(n))$ and $g_2(n) \in O(f_2(n))$ that $g_1(n) \leq c_1 f_1(n)$ for all $n > n_1$ and $g_2(n) \leq c_2 f_2(n)$ for all $n > n_2$, respectively, with positive real constants c_1 and c_2 and positive integers n_1 and n_2. Then for all $n \geq \max\{n_1, n_2\}$ this is also true:

$$g_1(n) + g_2(n) \leq c_1 f_1(n) + c_2 f_2(n) \leq \max\{c_1, c_2\} (f_1(n) + f_2(n))$$

But $f_1(n) + f_2(n) \leq 2\max\{f_1(n), f_2(n)\}$, so $g_1(n) + g_2(n) \leq c \max\{f_1(n), f_2(n)\}$ where $c = 2\max\{c_1, c_2\}$. Therefore, $g_1(n) + g_2(n) \in O(\max\{f_1(n), f_2(n)\})$, and the rule of sums for "Big Oh" is true. □

Lemma 1.18

Proof. It follows from $g_1(n) \in O(f_1(n))$ and $g_2(n) \in O(f_2(n))$ that $g_1(n) \leq c_1 f_1(n)$ for all $n > n_1$ and $g_2(n) \leq c_2 f_2(n)$ for all $n > n_2$, respectively, with positive real constants c_1 and c_2 and positive integers n_1 and n_2. Then for all $n \geq \max\{n_1, n_2\}$ this is also true: $g_1(n) g_2(n) \leq c f_1(n) f_2(n)$ where $c = c_1 c_2$. Therefore the rule of products for "Big Oh" is true. □

SOLUTION TO EXERCISE 1.3.7 ON PAGE 27:
The rule of sums for "Big Omega" and "Big Theta" is similar to that for "Big Oh", namely,

If $g_1 \in \Omega(f_1)$ and $g_2 \in \Omega(f_2)$, then $g_1 + g_2 \in \Omega(\max\{f_1, f_2\})$, and

If $g_1 \in \Theta(f_1)$ and $g_2 \in \Theta(f_2)$, then $g_1 + g_2 \in \Theta(\max\{f_1, f_2\})$.

SOLUTION TO EXERCISE 1.3.8 ON PAGE 27:
The Lemmas and proofs are similar to the "Big Oh" ones, except the inequalities are "greater than" instead of "less than".
Lemma 1.15 (Scaling). For all constants $c > 0$, $cf \in \Omega(f)$, in particular, $f \in \Omega(f)$.

Proof. The relationship $cf(n) \geq cf(n)$ holds for all $n > 0$. Thus, constant factors are ignored. □

Lemma 1.16 (Transitivity). If $h \in \Omega(g)$ and $g \in \Omega(f)$, then $h \in \Omega(f)$.

Proof. It follows from $h(n) \in \Omega(g(n))$ and $g(n) \in \Omega(f(n))$ that $h(n) \geq c_1 g(n)$ for all $n > n_1$ and $g(n) \geq c_2 f(n)$ for all $n > n_2$ where c_1 and c_2 are positive real constants and n_1 and n_2 are positive integers. Then $h(n) \geq c_1 g(n) \geq c_1 c_2 f(n)$, or $h(n) \geq c f(n)$ is also true for $c = c_1 c_2$ and all $n \geq n_0 = \max\{n_1, n_2\}$. Therefore Big Omega is transitive. □

Lemma 1.17 (Rule of sums). If $g_1 \in \Omega(f_1)$ and $g_2 \in \Omega(f_2)$, then $g_1 + g_2 \in \Omega(\max\{f_1, f_2\})$.

Proof. It follows from $g_1(n) \in \Omega(f_1(n))$ and $g_2(n) \in \Omega(f_2(n))$ that $g_1(n) \geq c_1 f_1(n)$ for all $n > n_1$ and $g_2(n) \geq c_2 f_2(n)$ for all $n > n_2$, respectively, with positive real constants c_1 and c_2 and positive integers n_1 and n_2. Then for all $n \geq \max\{n_1, n_2\}$ this is also true:

$$g_1(n) + g_2(n) \geq c_1 f_1(n) + c_2 f_2(n) \geq \min\{c_1, c_2\} (f_1(n) + f_2(n))$$

But $f_1(n) + f_2(n) \geq \max\{f_1(n), f_2(n)\}$, so $g_1(n) + g_2(n) \geq c \max\{f_1(n), f_2(n)\}$ where $c = \min\{c_1, c_2\}$. Therefore, $g_1(n) + g_2(n) \in \Omega(\max\{f_1(n), f_2(n)\})$, and the rule of sums for "Big Omega" is true. □

Lemma 1.18 (**Rule of products**). Similar to the above lemmas.

SOLUTION TO EXERCISE 1.4.1 ON PAGE 30:
You can make n the subject of the equation for all $f(n)$ except for $n \lg n$. To work out $n \lg n$, simply guess n until you find the correct value for 1 millennium.

	Length of time to run an algorithm	
$f(n)$	1 century	1 millennium
n	5.26×10^9	5.26×10^9
$n \lg n$	6.72×10^7	5.99×10^8
$n^{1.5}$	1.40×10^6	$6.51 \cdot 10^6$
n^2	72 522	229 334
n^3	3 746	8 071
2^n	35	39

SOLUTION TO EXERCISE 1.4.2 ON PAGE 30:

Time complexity		**Input size** n				**Time** $T(n)$
Function	*Notation*	*10*	*30*	*100*	*1000*	
"$\log\log n$"	$\lg\lg n$	1	1.23	1.42	1.68	$\lg\lg n / \lg\lg 10$
"$n^2 \log n$"	$n^2 \lg n$	1	13.29	200	30000	$n^2 \lg n / 100 \lg 10$

SOLUTION TO EXERCISE 1.5.1 ON PAGE 36:
The recurrence $T(n) = T(n-1) + n$; $T(0) = 0$, in Example 1.29 implies that $T(n) \geq 0$ for $n \geq 0$, so $T(n) \geq n$ for all $n > 0$. Therefore, $T(n) \in \Omega(n)$ for general n. Similarly, $T(n) \geq 0$

for all $n \geq 0$, so $T(n) \geq n$ for all $n > 0$ and therefore, $T(n) \in \Omega(n)$ in Examples 1.31 and 1.32.

Conversely, in Example 1.30 $T(n) \notin \Omega(n)$ because $T(n) \leq \lceil \lg n \rceil < \lg(n+1)$ for all $n \geq 2$, and the logarithmic function grows slower than n (Example 1.14).

SOLUTION TO EXERCISE 1.5.2 ON PAGE 36:
The base case holds for $n = 2$: $T(2) = T(1) + T(1) + 2 = 2 < 3 = 2\lg 2 + 2 - 1$. By the inductive hypothesis, $T(m) \leq m \lg m + m - 1 = m(\lg m + 1) - 1$ for all $m < n$. For an even n, $\lceil \frac{n}{2} \rceil = \lfloor \frac{n}{2} \rfloor = \frac{n}{2}$, so

$$T(n) \leq 2\left(\tfrac{n}{2}\left(\lg\left(\tfrac{n}{2}\right)+1\right)-1\right) = n\lg n - 2 \leq n\lg n + n - 1;\ n \geq 4$$

For an odd n, $\lceil \frac{n}{2} \rceil = \frac{n+1}{2}$; $\lfloor \frac{n}{2} \rfloor = \frac{n-1}{2}$, and $\lg(n-1) < \lg(n+1) < \lg n + 1$, so

$$\begin{aligned} T(n) &\leq \tfrac{n+1}{2}\lg\left(\tfrac{n+1}{2}\right) + \tfrac{n-1}{2}\lg\left(\tfrac{n-1}{2}\right) + \tfrac{n+1}{2} + \tfrac{n-1}{2} - 2 \\ &\leq \tfrac{n+1}{2}\lg(n+1) + \tfrac{n-1}{2}\lg(n-1) - 2 \\ &\leq \tfrac{n+1}{2}(\lg n + 1) + \tfrac{n-1}{2}(\lg n + 1) - 1 = n\lg n + n - 1;\ n \geq 3 \end{aligned}$$

Therefore, for all $n \geq 2$, $T(n) \leq n\lg n + n - 1$.

SOLUTION TO EXERCISE 1.5.3 ON PAGE 36:
Substituting $n = k^m$ into the recurrence $T(n) = kT\left(\frac{n}{k}\right) + cn;\ T(1) = 0$ gives $T(k^m) = kT(k^{m-1}) + ck^m;\ T(k^0) = 0$. Telescoping the latter recurrence yields:

$$\begin{aligned} T(k^m) &= kT(k^{m-1}) + ck^m \\ &= k\left(kT(k^{m-2}) + ck^{m-1}\right) + ck^m \\ &= k^2T(k^{m-2}) + 2ck^m \\ T(k^m) &= k^2\left(kT(k^{m-3}) + ck^{m-2}\right) + 2ck^m \\ &= k^3T(k^{m-3}) + 3ck^m \\ &\cdots \\ T(k^m) &= k^mT(1) + cmk^m \\ &= cmk^m \end{aligned}$$

or, what is the same, $T(n) = cn\log_k n$. Therefore, $T(n) \in O(n\log n)$.

SOLUTION TO EXERCISE 1.5.4 ON PAGE 36:
Just as in the previous solution, substituting $n = k^m$ into the recurrence $T(n) = kT(\frac{n}{k}) + ckn;\ T(1) = 0$, produces $T(k^m) = kT(k^{m-1}) + ck^{m+1};\ T(1) = 0$. Telescoping the latter

recurrence gives:

$$\begin{aligned}
T(k^m) &= kT(k^{m-1}) + ck^{m+1} \\
&= k\left(kT(k^{m-2}) + ck^m\right) + ck^{m+1} \\
&= k^2T(k^{m-2}) + 2ck^{m+1} \\
T(k^m) &= k^2\left(kT(k^{m-3}) + ck^{m-1}\right) + 2ck^{m+1} \\
&= k^3T(k^{m-3}) + 3ck^{m+1} \\
&\ldots. \\
T(k^m) &= k^mT(1) + cmk^{m+1} \\
&= cmk^{m+1} = ckmk^m
\end{aligned}$$

or, what is the same, $T(n) = ckn\log_k n$. Therefore, $T(n) \in O(n\log n)$.

SOLUTION TO EXERCISE 1.6.1 ON PAGE 37:
Because $n \in O(n\log n)$, in "Big-Oh" sense the linear algorithm **B** has better performance than the "$n\log n$" algorithm **A**. But for small enough n, the latter algorithm is faster, e.g. $T_{\mathbf{A}}(10) = 50$ and $T_{\mathbf{B}}(10) = 400$ elementary operations. The cutoff point is when $T_{\mathbf{A}}(n) = T_{\mathbf{B}}(n)$, that is: $5n\log_{10} n = 40n$, or $\log_{10} n = 8$, or $n = 10^8$. Therefore, even though the algorithm **B** is faster in "Big Oh" sense, this only occurs when more than 100 million data items have to be processed.

SOLUTION TO EXERCISE 1.6.2 ON PAGE 38:
In "Big-Oh" sense, the average-case time complexity of the linear algorithm **A** is larger than of the "$\sqrt{n}$" algorithm **B**. But for a database of the given size, $T_{\mathbf{A}}(10^9) = 10^6$ and $T_{\mathbf{B}}(10^9) = 1.58 \times 10^7$ elementary operations. So in this case the algorithm **A** is, in the average, over ten times faster than the algorithm **B**. Because we can tolerate the risk of an occasional long running time that might occur more likely with the more complex algorithm, the algorithm **A** should be used.

SOLUTION TO EXERCISE 2.1.2 ON PAGE 41:
Regardless of the initial ordering of the list, selection sort searches at each iteration i through the entire unsorted part of the size $n-i$ and makes $n-i-1$ comparisons to find the minimum element, so in total, $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in \Theta(n^2)$ comparisons in the worst, average, and best case. The maximum number of data moves is n, because each iteration moves at most one element into its correct position, and their average number is $\frac{n}{2}$. Thus, both the maximum and the average individual time complexity in selection sort is $\Theta(n)$ for data moves and $\Theta(n^2)$ for comparisons.

SOLUTION TO EXERCISE 2.2.1 ON PAGE 44:
Adding up the columns in the next table gives, in total, 90 comparisons plus data moves.

i	C_i	M_i	Data to sort									
			91	**70**	65	50	31	25	20	15	8	2
1	1	1	70	91	**65**	50	31	25	20	15	8	2
2	2	2	65	70	91	**50**	31	25	20	15	8	2
3	3	3	50	65	70	91	**31**	25	20	15	8	2
4	4	4	31	50	65	70	91	**25**	20	15	8	2
5	5	5	25	30	50	65	70	91	**20**	15	8	2
6	6	6	20	25	30	50	65	70	91	**15**	8	2
7	7	7	15	20	25	30	50	65	70	91	**8**	2
8	8	8	8	15	20	25	30	50	65	70	91	**2**
9	9	9	2	8	15	20	25	30	50	65	70	91

SOLUTION TO EXERCISE 2.2.2 ON PAGE 44:
The inductive hypothesis is that each inner-loop iteration $i = 1,\ldots,n-1$ of insertion sort increases by one the size of the already sorted part $a[0],\ldots,a[i-1])$ of size i in the list under consideration, while keeping it sorted. The base case for the math induction is for $i = 0$ when the sorted part $(a[0])$ of size 1 is sorted by definition. At iteration i, an element $temp$ from the unsorted part of the list is inserted into the already sorted part between the elements $a[j-1]$ and $a[j]$ such that $a[j-1] \le temp < a[j]$. Either the left-hand or right-hand inequality may be absent if $j = 0$ or $j = i$, respectively. The obtained new part of size $i+1$ is sorted because all the previous elements smaller than or equal to $temp$ are before it and stay sorted, and all elements greater than $temp$ are after it and also stay sorted. So because the iterations terminate when $i > n-1$, insertion sort is correct.

Moreover, duplicates will be lumped together and their relative order in the initial unsorted array will not change, so insertion sort is *stable*.

SOLUTION TO EXERCISE 2.2.3 ON PAGE 44:
Insertion sort runs the slowest on the totally reverse ordered list that has the maximum number of inversions: $\binom{n}{2} = \frac{n(n-1)}{2} \in \Theta(n^2)$. The worst-case time complexity of insertion sort is $\Theta(n^2)$ because each swap removes only one inversion.

SOLUTION TO EXERCISE 2.2.4 ON PAGE 45:
Obviously, sorting of elements that precede $a[i]$ (i.e. with positions less than i) does not change their inversions with $a[i]$. So the number ν of inversions between $a[i]$ and the preceding elements is equal to the total number of elements greater than $a[i]$ among the elements $a[0],\ldots,a[i-1]$. Just before the iteration i places the element $a[i]$ into its correct position, the preceding sorted part will have the ν elements; $0 \le \nu \le i$, being greater than $a[i]$ and ordered immediately before $a[i]$ at the positions $i-1,\ldots,i-\nu$. During execution of insertion sort on an array, every element of the array that is greater than $a[i]$ must be moved up once. Thus the total number of data moves to insert $a[i]$ is equal to the total number ν of inversions with respect to the preceding elements in the initial array.

SOLUTION TO EXERCISE 2.2.5 ON PAGE 45:
The out-of-order elements $a[i]$ and $a[i+gap]$ are not equal one to another, so $a[i] > a[i+gap]$. The elements at positions less than i or greater than $i+gap$ do not change their inversions with respect to either $a[i]$ or $a[i+gap]$ after the latter are swapped. The swap adds no new inversions with the elements, $a[k]$; $i < k < i+gap$, between these positions because all the already ordered pairs such that $a[i] < a[k]$ and $a[k] < a[i+gap]$ remain ordered after this swap. Since no inversions are added but one inversion is removed by placing $a[i]$ and $a[i+gap]$ in order, the minimum number of the inversions removed is 1.

The maximum number of the inversions is removed if for every pair of positions (i,k) or $(k,i+gap)$ where $i < k < i+gap$ there was an inversion before, but no inversion after the swap. There are $2 \times (gap-1)$ such pairs, so the maximum number of the inversions removed is $2 \times gap - 1$. Thus, swapping the out-of-order elements $a[i]$ and $a[i+gap]$ of a list a removes at least 1 and at most $2 \times gap - 1$ inversions.

SOLUTION TO EXERCISE 2.2.6 ON PAGE 45:
The while loop runs until there are no data swaps in the inner for-loop, so the while loop stops just after the list is sorted. Each swap of elements $a[i]$ and $a[i-1]$ in the inner for-loop removes exactly one inversion, does not affect their inversions with the preceding or subsequent elements in the list, and obviously does not create any new inversion. Because the average number of inversions in a list is $\frac{1}{2}\binom{n}{2} = \frac{n(n-1)}{4} \in O(n^2)$, the average time complexity of bubble sort is $O(n^2)$.

SOLUTION TO EXERCISE 2.2.8 ON PAGE 46:

h	i	C_i	M_i	Data to sort									
5				91	70	65	50	31	25	20	15	8	2
	5	1	1	**25**					**91**				
	6	1	1		**20**					**70**			
	7	1	1			**15**					**65**		
	8	1	1				**8**					**50**	
	9	1	1					**2**					**31**
2				25	20	15	8	2	91	70	65	50	31
	2	1	1	**15**		**25**							
	3	1	1		**8**		**20**						
	4	2	2	**2**		**15**		**25**					
	5	1	0				20		91				
	6	1	0					25		70			
	7	2	1				20		**65**		**91**		
	8	2	1					25		**50**		**70**	
	9	3	2				20		**31**		**65**		**91**
1				2	8	15	20	25	31	50	65	70	91
	1	1	0	2	8								
	2	1	0		8	15							
	3	1	0			15	20						
	4	1	0				20	25					
	5	1	0					25	31				
	6	1	0						31	50			
	7	1	0							50	65		
	8	1	0								65	70	
	9	1	0									70	91

By adding up the columns, we get a total of 40 comparisons plus data moves. This is less than half that of insertion sort's 90, so even with a low n value, Shellsort is better than insertion sort.

SOLUTION TO EXERCISE 2.3.3 ON PAGE 49:
Insertion sort runs in $\Theta(n^2)$ in the average and worst case, so the total time for this algorithm is:

$$T(n) = kc\left(\frac{n}{k}\right)^2 + c(k-1)n = c\frac{n^2}{k} + c(k-1)n = cn\left(\frac{n}{k} + (k-1)\right)$$

Assuming the constant n and the variable k, $T(n)$ is minimal for the same value of k as the function $f_n(k) = \frac{n}{k} + k - 1$. Because $1 \leq k \leq n$, at the boundaries $f_n(1) = n$ and $f_n(n) = n$, and the function is not equal to n at every other k, at least one local optimum exists in the interval between $k = 1$ and $k = n$. For this optimum, the first derivative by k equals to 0: $\frac{df_n(k)}{dk} = -\frac{n}{k^2} + 1$, so $\frac{n}{k^2} = 1$, or $k = \sqrt{n}$. Since $f_n(\sqrt{n}) = 2\sqrt{n} - 1 < n$, it is a local minimum. Thus, when $k = \sqrt{n}$, $T(n) = cn(2\sqrt{n} - 1)$ is minimal, too.

It is not as fast as mergesort's $\Theta(n \log n)$ but quicker than insertion sort's $\Theta(n^2)$.

SOLUTION TO EXERCISE 2.4.1 ON PAGE 55:
Partitioning of a 5-element list after choosing the pivot $p = 15$ (the bold element is the pivot; elements in italics are those pointed to by the pointers L and R).

Data to sort					Description
20	8	2	25	15	Initial array
15	8	2	25	20	move pivot to head
15	8	*2*	25	20	stop R
15	8	*2*	25	20	stop L (as $L = R$)
2	8	*15*	25	20	swap element L with pivot

SOLUTION TO EXERCISE 2.5.1 ON PAGE 63:

Position	1	2	3	4	5	6	7	8	9	10	11	12
Index	0	1	2	3	4	5	6	7	8	9	10	11
Array at step 1	91	75	70	31	65	50	25	20	15	2	8	**85**
Array at step 2	91	75	70	31	65	**85**	25	20	15	2	8	50
Array at step 3	91	75	**85**	31	65	70	25	20	15	2	8	50

SOLUTION TO EXERCISE 2.5.2 ON PAGE 63:
Restoring the heap after deleting the maximum element:

Position	1	2	3	4	5	6	7	8
Index	0	1	2	3	4	5	6	7
Step 1	**15**	65	50	31	8	2	25	20
Step 2	**65**	*15*	50	31	8	2	25	20
Step 3	65	**31**	50	*15*	8	2	25	20
Step 4	65	31	50	**20**	8	2	25	*15*

SOLUTION TO EXERCISE 2.5.3 ON PAGE 63:

Position	1	2	3	4	5	6	7	8	9
Index	0	1	2	3	4	5	6	7	8
Initial array	10	20	30	40	50	60	70	80	90
$i = 3$	10	20	30	**90**	50	60	70	80	*40*
$i = 2$	10	20	**70**	90	50	60	*30*	80	40
$i = 1$	10	**90**	70	*20*	50	60	30	80	40
	10	90	70	**80**	50	60	30	*20*	40
$i = 0$	**90**	*10*	70	80	50	60	30	20	40
	90	**80**	70	*10*	50	60	30	20	40
	90	80	70	**40**	50	60	30	20	*10*
Max heap	90	80	70	40	50	60	30	20	10

SOLUTION TO EXERCISE 2.5.4 ON PAGE 63:

Position	1	2	3	4	5	6	7	8	9	10
Index	0	1	2	3	4	5	6	7	8	9
Initial array	2	8	15	20	25	31	50	65	70	91
Building the maximum heap										
$i=4$	2	8	15	20	**91**	31	50	65	70	*25*
$i=3$	2	8	15	**70**	91	31	50	65	*20*	25
$i=2$	2	8	**50**	70	91	31	*15*	65	20	25
$i=1$	2	**91**	50	70	*8*	31	15	65	20	25
	2	91	50	70	**25**	31	15	65	20	*8*
$i=0$	**91**	*2*	50	70	25	31	15	65	20	8
	91	**70**	50	*2*	25	31	15	65	20	8
	91	70	50	**65**	25	31	15	*2*	20	8
Max heap	91	70	50	65	25	31	15	2	20	8
Deleting max 1	8	70	50	65	25	31	15	2	20	**91**
Restoring heap 1-9	70	65	50	20	25	31	15	2	8	**91**
Deleting max 2	8	65	50	20	25	31	15	2	**70**	**91**
Restoring heap 1-8	65	25	50	20	8	31	15	2	**70**	**91**
Deleting max 3	2	25	50	20	8	31	15	**65**	**70**	**91**
Restoring heap 1-7	50	25	31	20	8	2	15	**65**	**70**	**91**
Deleting max 4	15	25	31	20	8	2	**50**	**65**	**70**	**91**
Restoring heap 1-6	31	25	15	20	8	2	**50**	**65**	**70**	**91**
Deleting max 5	2	25	15	20	8	**31**	**50**	**65**	**70**	**91**
Restoring heap 1-5	25	20	15	2	8	**31**	**50**	**65**	**70**	**91**
Deleting max 6	8	20	15	2	**25**	**31**	**50**	**65**	**70**	**91**
Restoring heap 1-4	20	8	15	2	**25**	**31**	**50**	**65**	**70**	**91**
Deleting max 7	2	8	15	**20**	**25**	**31**	**50**	**65**	**70**	**91**
Restoring heap 1-3	15	8	2	**20**	**25**	**31**	**50**	**65**	**70**	**91**
Deleting max 8	2	8	**15**	**20**	**25**	**31**	**50**	**65**	**70**	**91**
Restoring heap 1-2	8	2	**15**	**20**	**25**	**31**	**50**	**65**	**70**	**91**
Deleting max 9	2	**8**	**15**	**20**	**25**	**31**	**50**	**65**	**70**	**91**

SOLUTION TO EXERCISE 2.5.5 ON PAGE 63:
No, because the creation of a heap does not preserve the order of equal keys. Similarly, quicksort is also unstable, but insertion sort and mergesort are stable.

SOLUTION TO EXERCISE 2.5.6 ON PAGE 63:
The only significant increase in running time is at the very beginning when the heap is first created: since data items are in the wrong order, each element has to be percolated down to the leaves. But since this step is of $O(n)$ complexity, the $\Theta(n \log n)$ running time for the deletion of all the max values dominates. Hence, the running time does not differ significantly from the average-case one.

SOLUTION TO EXERCISE 2.6.3 ON PAGE 65:
In the average, quickselect does p linear operations, $T_{select}(n,p) = pc_1 n$, while quicksort does one $O(n \log n)$ operation, $T_{sort}(n,p) = c_2 n \lg n$. So quicksort is quicker in finding p keys, $T_{sort}(n,p) < T_{select}(n,p)$, when $c_2 n \lg n < pc_1 n$, or $p > \frac{c_2}{c_1} \lg n$. Because quickselect is similar to quicksort except of skipping one half at each step, we can assume that $c_1 \approx c_2$, so that quicksort is better if $p > \lg n$. When $n = 10^6$ and $p = 10$, $10 < \lg n = 19.9$. Therefore, the variant with quickselect should be used.

SOLUTION TO EXERCISE 2.6.4 ON PAGE 66:
Both heapselect and mergeselect have to order the list first before selecting the kth smallest item. So in both the cases the average-case and the worst-case time complexity is $\Theta(n \log n)$.

SOLUTION TO EXERCISE 2.7.1 ON PAGE 67:
Let us show that the sum of all heights of leaves of a decision tree with k leaves is at least $k \lg k$. The smallest height is when the tree is balanced so that the number of leaves on the left subtree is equal to the number of leaves on the right subtree. Let $H(k)$ be the sum of all heights of k leaves in such a tree. Then the left and the right subree attached to the root have $\frac{k}{2}$ leaves each and $H(k) = 2H\left(\frac{k}{2}\right) + k$ because the link to the root adds one to each height. Therefore, $H(k) = k \lg k$. In any other decision tree, the sum of heights cannot be smaller than $H(k)$. When $k = n!$, or the number of leaves is equal to the number of permutations of an array of n keys, $H(n!) = n! \log n!$. The average height of a leaf, given that each permutation has equal chance, is obtained by dividing the total of all heights by the total number of leaves:

$$H_{avg}(n!) = \frac{H(n!)}{n!} = \log n! \approx n \log n - 1.44n$$

This means that the lower bound of the average-case complexity of sorting n items by pairwise comparisons is $\Omega(n \log n)$.

SOLUTION TO EXERCISE 2.7.2 ON PAGE 67:
The time complexity is linear, $\Theta(n)$, as it takes n steps to scan through array a, and then a further n steps to print out the contents of t. Theorem 2.35 says that any algorithm that sorts by comparing *only pairs of elements* must use at least $\lceil \log n! \rceil$ comparisons in the worst case. This algorithm uses the specific knowledge that the contents of the array a are integers in the range 1..1000. Thus, this algorithm would not work if the keys can only be compared to each other because contrary to this case their absolute values are totally unknown.

SOLUTION TO EXERCISE 3.2.1 ON PAGE 76:
It will be identical to Figure 3.3 except the very last step will not return 4, but find instead that $a[m] > k$ so $r \leftarrow m - 1$ and $l > r$, so that the loop will terminate and return "not found".

SOLUTION TO EXERCISE 3.2.2 ON PAGE 76:
Binary search halves the array at each step, thus the worst case is when it does not find the key until there is only one element left in the range. Using the improved binary search that only does one comparison to split the array, we are looking for the smallest integer k such that $2^k \geq 10^6$, or $k = \lceil \lg 10^6 \rceil = 20$. Thus 20 comparisons are needed to reduce the range to 1, and in total there are 21 comparisons as at the end the comparison to the key is done.

SOLUTION TO EXERCISE 3.2.5 ON PAGE 77:

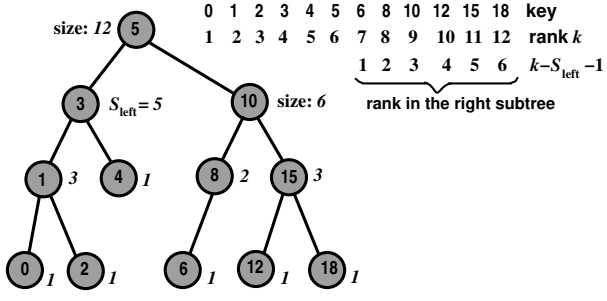
Index	0	1	2	3	4	5	6	7	8	9	**next index** m
Step 1	10	20	35	45	55	60	75	80	**85**	100	$8 = 0 + \lceil \frac{85-10}{100-10} \cdot 9 \rceil$
Step 2									**85**	100	$8 = 8 + \lceil \frac{85-85}{100-85} \cdot 1 \rceil$
Step 3									85		return value: 8

Interpolation search will search through three positions.

SOLUTION TO EXERCISE 3.3.1 ON PAGE 82:
In line with the ordering relation of a BST, the search for the maximum key starts at the root, repeatedly goes right while the right child exists, and stops at the node with the largest key. The search for the smallest key is similar, except it moves left instead of right.

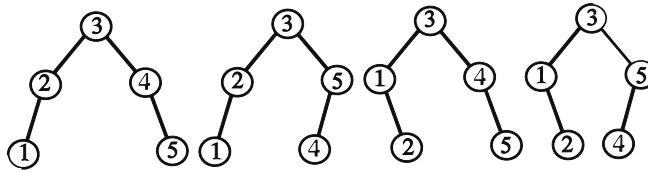
The running time of these algorithms for a BST with n nodes depends on the tree shape: it is $\Theta(n)$ in the worst case and $\Theta(\log n)$ in the best and the average cases.

To find the median or any other rank statistic, notice that the root of a BST behaves like the pivot in quickselect or quicksort because all the keys to the left or to the right are smaller or greater, respectively. In general, the rank of a key is equal to the ***rank*** of its node defined as 1 plus the number of nodes above or to the left including the number of nodes in its left subtree. A key of rank k, i.e. the k-th smallest key, $1 \le k \le n$, is found by a recursive tree search controlled, like in quickselect, by the rank r of the root. If $r = k$ then the goal key is in the root. Otherwise, it is the k-th smallest key in the left subtree if $k < r$ or the $(k-r)$-th smallest key in the right subtree if $k > r$. The figure below illustrates this search. The size of the subtree below a node (including the node itself) is in *italic*. The root contains the 6th smallest key, since its left subtree (not including the root itself) is of size 5. The 4th smallest key, 3, has left subtree of size 3. The 9th smallest key, 10, is the 3rd smallest key in the right subtree $(3 = 9-5-1)$ and has left subtree of size 2.



SOLUTION TO EXERCISE 3.3.2 ON PAGE 82:
Under all possible insertion sequences, more balanced trees appear more frequently

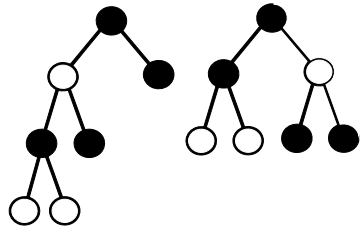
than unbalanced ones. Thus the balanced trees (and therefore their shapes) displayed below will occur most often.



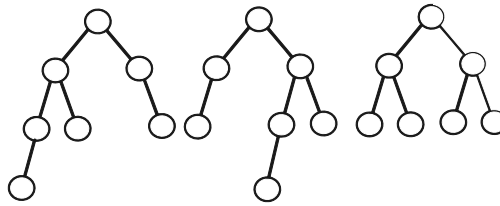
SOLUTION TO EXERCISE 3.3.3 ON PAGE 82:
As was shown in Figure 3.9, the insertion orders 1423, 1432, 1243, 1234, 4312, 4321, 4132, and 4123 yield a tree of the maximal height 3, while 2134, 2143, 2314, 2341, 2413, 2431, 3124, 3142, 3214, 3241, 3412, and 3421 yield a tree of the minimal height 2.

SOLUTION TO EXERCISE 3.3.4 ON PAGE 82:
Just as in the above solution to Exercise 3.3.1, notice that according to the ordering relation, the root of a BST behaves like the pivot in quicksort because all the keys to its left or to its right are smaller or greater, respectively. Therefore, all the keys can be sorted in ascending order by a recursive tree traversal similar to quicksort in that the left subtree is visited first, then the root, then the right subtree, so the output of records in order of visiting the nodes of the BST yields the ascending order of their keys.

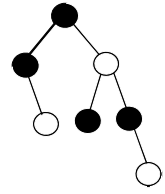
SOLUTION TO EXERCISE 3.4.1 ON PAGE 88:
Several solutions exist, and two red-black trees with two black nodes along "root–leaf" paths are shown here.



SOLUTION TO EXERCISE 3.4.2 ON PAGE 88:
Several solutions exist, and two AVL trees with 7 nodes are shown below along with the complete binary tree, which is also an AVL tree.



SOLUTION TO EXERCISE 3.4.3 ON PAGE 88:
One AA tree with two black nodes along "root-leaf" paths, of several possible solutions, is shown below.



SOLUTION TO EXERCISE 3.5.1 ON PAGE 100:
For example, two strings $s_1 =$ bC and $s_2 =$ ab have just the same hash code $31 \times 98 + 67 = 31 \times 97 + 98 = 3105$. Generally, the same hash code is for a pair of 2-letter strings, s_1 and s_2, such that $h(s_1) = h(s_2)$, that is, $31s_1[0] + s_1[1] = 31s_2[0] + s_2[1]$, or $31(s_1[0] - s_2[0]) = s_2[1] - s_1[1]$. Let for definiteness, $s_1[0] > s_2[0]$. Positive differences between the Unicode values of two capitals (A,B,...,Z) or two small letters (a,b,...,z) are in the range of $[0..25]$, and between the values of a small and a capital letter are in the range of $[7..57]$. Therefore, only a single pair of the differences, $s_1[0] - s_2[0] = 1$ and $s_2[1] - s_1[1] = 31$, results in the same hash codes of two different 2-letter strings. Because there are $25 + 25 = 50$ pairs of the first letters such that $s_2[0] = s_1[0] - 1$ and 25 pairs of the second letters such that $s_2[1] = s_1[1] + 31$, in total $50 \times 25 = 750$ pairs out of all possible $52^2 = 2704$ 2-letter strings have the same hash code.

For the even n, the hash function is reduced to the weighted sum of hash codes for the successive 2-letter substrings:

$$h(s) = 31^{n-2}(31s[0] + s[1]) + 31^{n-4}(31s[2] + s[3]) + \ldots + (31s[n-2] + s[n-1])$$

Let $S_m = \{(s_{i:0}; s_{i:1}) : i = 1, \ldots, m\}$ be an arbitrary subset of m different 2-letter pairs having each the same hash code h_i. Then 2^m strings s, size of $2m$ letters each, with the same hash code $h(s) = 31^{2m-2}h_0 + 31^{2m-4}h_1 + \ldots + 31h_{m-2} + h_{m-1}$ can be built by concatenating the m substrings, one from each pair: $s = s_{1:\alpha_1} s_{2:\alpha_2} \ldots s_{m:\alpha_m}$ providing $s_{i:0}$ or $s_{i:1}$ are selected in accord with the i-th binary digit α_i of the binary number between 0 and $2^m - 1$. To form 2^{100} such strings, we need to select $m = 100$ arbitrary 2-letter pairs, S_{100}, having each the same hash code.

SOLUTION TO EXERCISE 3.5.2 ON PAGE 101:

Hash table index	0	1	2	3	4	5	6	7	8	9	10	11	12
Insert 10											**10**		
Insert 26	**26**										10		
Insert 52, collision at 0	26										10		**52**
Insert 76	26										10	**76**	52
Insert 13, collision at 0	26									**13**	10	76	52
Insert 8	26								**8**	13	10	76	52
Insert 3	26			**3**					8	13	10	76	52
Insert 33	26			3				**33**	8	13	10	76	52
Insert 60, collision at 8	26			3			**60**	33	8	13	10	76	52
Insert 42, collision at 3	26		**42**	3			60	33	8	13	10	76	52
Resulting hash table	26		42	3			60	33	8	13	10	76	52

SOLUTION TO EXERCISE 3.5.3 ON PAGE 101:

Hash table index	0	1	2	3	4	5	6	7	8	9	10	11	12
Insert 10											**10**		
Insert 26	**26**										10		
Insert 52, collision at 0, $\Delta = 4$	26									**52**	10		
Insert 76	26									52	10	**76**	
Insert 13, collision at 0, $\Delta = 1$	26									52	10	76	**13**
Insert 8	26								**8**	52	10	76	13
Insert 3	26			**3**					8	52	10	76	13
Insert 33	26			3				**33**	8	52	10	76	13
Insert 60, collision at 8, $\Delta = 4$	26			3	**60**			33	8	52	10	76	13
Insert 42, collision at 3, $\Delta = 3$	26	**42**		3	60			33	8	52	10	76	13
Resulting hash table	26	42		3	60			33	8	52	10	76	13

SOLUTION TO EXERCISE 3.5.4 ON PAGE 101:

Hash table index	0	1	2	3	4	5	6	7	8	9	10	11	12
Insert 10											**10**		
Insert 26	**26**										10		
Insert 52, collision at 0	{26,**52**}										10		
Insert 76	{26,52}										10	**76**	
Insert 13, collision at 0	{26,52,**13**}										10	76	
Insert 8	{26,52,13}								**8**		10	76	
Insert 3	{26,52,13}			**3**					8		10	76	
Insert 33	{26,52,13}			3				**33**	8		10	76	
Insert 60, collision at 8	{26,52,13}			3				33	{8,**60**}		10	76	
Insert 42, collision at 3	{26,52,13}			{3,**42**}				33	{8,60}		10	76	
Resulting hash table	{26,52,13}			{3,42}				33	{8,60}		10	76	

SOLUTION TO EXERCISE 4.1.1 ON PAGE 109:

Consider a set of arcs E and nodes V, we want to construct the digraph $G = (V, E')$ where $E' = E$. We do this by taking the empty digraph $G' = (V, \{\})$ and for each arc in

E add it to the graph G'. Each time we add an arc $(u,v) \in E$ to G', we are adding one to the outdegree of node u and one to the indegree of v. When we have added all the arcs we get the graph G and since every time the outdegree of a node increased, the indegree of a node also increased. Hence, the sum of the outdegrees equal the sum of the indegrees.

For a graph, an analogous statement would be that the sum of all the degrees of all vertices is equal to twice the number of edges in the graph. This is because each edge contains two different vertices.

SOLUTION TO EXERCISE 4.1.2 ON PAGE 110:
Given that the path exists between nodes u and v, then there exists a sequence of nodes $u, v_1, v_2, \ldots, v_k, v$ such that no node is repeated. If no node is repeated then $k \leq n-2$ where n is the number of nodes in the digraph. If k was any greater, than a node would be repeated, and it would contain a cycle that can be eliminated. As $d(u,v) \leq k+1$, $d(u,v) \leq n-1$.

SOLUTION TO EXERCISE 4.1.3 ON PAGE 110:
We have defined sparse being if the number of edges being $O(n)$ where n is the number of nodes. And dense is when the number of edges are $\Omega(n^2)$.

The sum of the indegrees of the digraph is also the number of arcs m. The average indegree of a node is $\frac{m}{n}$.

For sparse graphs, the number of arcs $m \in O(n)$, or $m \leq cn$ for some fixed constant c independent of n. Thus, the average indegree is less than or equal to c, which is $O(1)$.

For dense graphs, the number of edges $m \in \Omega(n^2)$, or $m \geq cn^2$ for some fixed constant c independent of n. Thus, the average is greater than or equal to cn, which is $\Omega(n)$.

SOLUTION TO EXERCISE 4.2.1 ON PAGE 112:

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

SOLUTION TO EXERCISE 4.2.2 ON PAGE 113:

```
7
1 4 5
0 3
0 6
0 5
5

5
```

SOLUTION TO EXERCISE 4.2.3 ON PAGE 113:

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

SOLUTION TO EXERCISE 4.2.4 ON PAGE 113:

G:

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

G_r :

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

SOLUTION TO EXERCISE 5.1.2 ON PAGE 121:
At each time we turn right, else if its a dead end back up as little as possible. What we are doing is applying to the *visit* algorithm (Figure 5.1) a heuristic, a method of solving a problem, to the *'choose a grey node* u*'* statement. The *visit* algorithm allows for any way of choosing which grey node to choose next, so the way specified in the exercise is a valid way that will, eventually, result in finding the exit.

SOLUTION TO EXERCISE 5.3.2 ON PAGE 127:
When DFS is run on the graph, the follow timestamps are obtained.

v	0	1	2	3	4	5	6
$seen[v]$	0	2	1	6	7	8	9
$done[v]$	5	3	4	13	12	11	10

Tree arcs: (0,2),(2,1),(3,4),(4,5),(5,6)
Forward arcs: (3,5),(3,6)
Back arcs: (1,0),(2,0),(5,3),(5,4)
Cross arcs: (6,1),(6,2)

SOLUTION TO EXERCISE 5.3.4 ON PAGE 127:
By using the defintions of these arcs and also Theorem 5.5, we can prove each of the three statements.

Let $(v,w) \in E(G)$ be an arc. A forward arc is an arc such that v is an ancestor of w in the tree and is not a tree arc. If the arc (v,w) is in the tree (a tree arc), then v is still an ancestor of w. Thus, the arc (v,w) is a tree or forward arc if and only if v is an ancestor of w.

By Theorem 5.5, v is an ancestor of w is equivalent to:

$$seen[v] < seen[w] < done[w] < done[v]$$

The arc (v,w) is a back arc if w is an ancestor of v in the tree and is not a tree arc. If w is an ancestor of v then it cannot be a tree arc by the above proof. This means that the arc is a back arc if and only if w is an ancestor of v.

By Theorem 5.5, w is an ancestor of v is equivalent to:

$$seen[w] < seen[v] < done[v] < done[w]$$

The arc (v,w) is a cross arc if neither v nor w are the ancestor of the other. But, we only need to check if w is not an ancestor of v. This is because of the order that DFS visits these nodes, if it visits v before w ($seen[v] < seen[w]$) then this will be a tree or forward arc. So to be a cross arc, $seen[w] < seen[v]$ must be true. This means that the arc (v,w) is a cross arc if and only if w is not an ancestor of v.

By Theorem 5.5, w is not an ancestor of v is equivalent to:

$$seen[w] < done[w] < seen[v] < done[v]$$

SOLUTION TO EXERCISE 5.3.5 ON PAGE 128:

(i) Simply apply the rules found in Exercise 5.3.4, the table entries are the type of arc (u,v) would be if the arc existed.

u	v						
	0	1	2	3	4	5	6
0		Tree	Forward	Tree	Forward	Forward	Forward
1	Back		Tree	Cross	Forward	Forward	Forward
2	Back	Back		Cross	Forward	Tree	Forward
3	Back	Cross	Cross		Cross	Cross	Cross
4	Back	Back	Back	Cross		Back	Cross
5	Back	Back	Back	Cross	Tree		Tree
6	Back	Back	Back	Cross	Cross	Back	

Of course, if some of these arcs existed, and DFS was run on the graph, some of the timestamps would change.

(ii) It would be a back arc.

(iii) No, because they are on different branches of the tree (hence if $(3,2)$ was an arc in the graph, it would be a cross arc)

(iv) No, because then when DFS is at time 4, instead of expanding node 4, it would expand node 3, and the DFS tree would be entirely different.

(v) Yes, because it is DFS the tree would be the same (but not if it was BFS), the arc would be a forward arc.

SOLUTION TO EXERCISE 5.3.9 ON PAGE 129:
The order of the nodes in the digraph in the *seen* array is equal to the preorder labeling of the nodes, and the order of the nodes in the diagraph in the *done* array is equal to the post order labeling of the nodes.

SOLUTION TO EXERCISE 5.3.10 ON PAGE 129:
To prove via induction, we need both a base case and an inductive step.

The base case is when there are no white nodes as neighbours of node s, then the algorithm does no recursion and returns. In this case `recursiveDFSvisit` only visits node s, which is intended.

The inductive step is that given all the white nodes that are neighbours of node s our theorem is true for them, then it is also true for node s. For each node reachable from s via a path of white nodes, the start of every path is one of the neighbours of s. Because the call to `recursiveDFSvisit` with input s only terminates when the recursive calls with input of each of the white neighbours of s finish. All the recursive calls then cover each of the paths from s to each node reachable by a path of white nodes, and thus satisfies the inductive step.

Finally, because each path cannot have a loop in it, there is a finite number of recursions and `recursiveDFSvisit` is guaranteed to terminate.

Therefore, by mathematical induction, Theorem 5.4 is true.

SOLUTION TO EXERCISE 5.6.2 ON PAGE 137:
By Theorem 5.11, every DAG has a topological ordering $(v_1, v_2, \ldots v_n)$ such that there are no arcs $(v_i, v_j) \in E(G)$ such that $i < j$. This means that there are no arcs going from right to left in the topological ordering. This means that node v_1 has no arcs going into it and node v_n has no nodes going away from it, they are respectively, a source and sink node. Therefore for every DAG there is at least one source and sink node.

SOLUTION TO EXERCISE 5.6.4 ON PAGE 137:
Shirt, hat, tie, jacket, glasses, underwear, trousers, socks, shoes, belt.

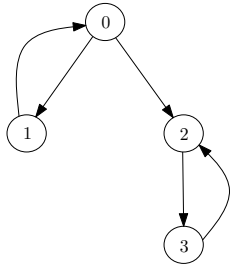
SOLUTION TO EXERCISE 5.6.5 ON PAGE 137:
The standard implementation uses an array of indegrees. This can be computed in time $O(m)$ from either adjacency lists or adjacency matrices. The algorithm can find a node v of degree 0 in time $O(n)$ and can decrement the indegrees of the neighbours of v in constant time for adjacency lists. Since we have at most m decrements of elements of the array of indegree, the running time is at most $O(n^2 + m)$. If a priority queue is used to extract nodes of indegree 0 the running time slightly improves.

SOLUTION TO EXERCISE 5.6.6 ON PAGE 137:
Simply delete vertices with 0 or 1 edges on them from the graph (including the edges), if at anytime there are no vertices with the number of edges less than 2, then the graph has a cycle. Otherwise, if the entire graph can be deleted by only deleting vertices with 0 or 1 edges, then the graph is acyclic.

SOLUTION TO EXERCISE 5.7.1 ON PAGE 142:

Adjacency list:

4
1,2
0
3
2

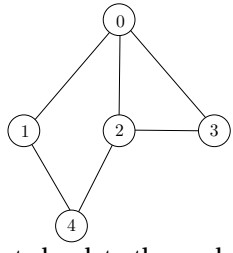


There are two strongly connected components in this graph, DFS only finds one tree.

SOLUTION TO EXERCISE 5.8.1 ON PAGE 145:

Adjacency list:

5
1,2,3
0,4
0,4
0,2



If the algorithm does not check to the end of the level, it will return that the shortest cycle is {0,1,4,2} instead of {0,2,4}.

SOLUTION TO EXERCISE 5.8.3 ON PAGE 145:

We need to find two disjoint subsets. Consider the number of 1's (it can just as easily be the number of 0's) to be k in a bit vector of length n, an edge can only be between another bit vector with either $k-1$ or $k+1$ 1's. This is because if the number of 1's is less than $k-1$ or greater than $k+1$ then there will be more than 1 difference in the bits. Also, two different bit vectors with the same number of 1's will not have an edge because they will differ in two places exactly (not the required one).

One way of satisfying this condition is if all the odd number of 1's are on one side, and all the even number of 1's are on the other. This means that for any n-cube you can find a bipartite consisting of the odd number of 1's bit vectors in one group, and the even number of 1's in the other.

SOLUTION TO EXERCISE 6.2.2 ON PAGE 154:

The running time is the same as the time to compute the distance matrix. The eccentricity of a node v is simply the maximum entry of row v of the distance matrix. The radius is the minimum over all maximum values per row. This can be computed in time $\Theta(n^2)$ if we have access to a distance matrix.

SOLUTION TO EXERCISE 6.3.2 ON PAGE 160:
If a cycle $v_1, v_2, \ldots, v_k$ exists with the sum of its arc weights is less than zero then we can find a walk of total weight as small as we want from v_1 to v_2 by repeating the cycle as many times as we want before stopping at v_2.

SOLUTION TO EXERCISE 6.3.6 ON PAGE 161:
Property P2 fails if we allow arcs of negative weight. Suppose u is the next vertex added to S. If arc (u,w) is of negative weight for some other vertex w that is currently in S, then the previous distance from s to w, $dist[w]$, may no longer be the smallest.

SOLUTION TO EXERCISE 6.4.2 ON PAGE 164:
If a diagonal entry in the distance matrix ever becomes less than zero when using Floyd's algorithm then know that a negative weight cycle has been found.

SOLUTION TO EXERCISE 6.5.1 ON PAGE 167:
For this weighted graph, both Prim's and Kruskal's algorithms will find the unique minimum spanning tree of weight 9.

Bibliography

[1] A. V. Aho, J. D. Ullman. *Foundations of Computer Science*, Computer Science Press, 1992.

[2] J. Bentley. *Programming Pearls*, Second Edition. Addison-Wesley, Inc., 2000. 101

[3] F. J. Brandenburg. *The Graph Template Library—GTL*, (see http://www.infosun.fim.uni-passau.de/GTL/) 116

[4] T. H. Cormen, C. E. Leiserson and R. L. Rivest. *Introduction to Algorithms*, McGraw-Hill, New York, 1990.

[5] J. Edmonds. "Paths, trees, and flowers," *Canadian Journal of Mathematics* 17 (1965), pages 449–467. 150

[6] R. Sedgewick and P. Flajolet. *Introduction to the Analysis of Algorithms*, Addison-Wesley, Inc., 1996. 101

[7] L. R. Ford and D. R. Fulkerson "Maximal flow through a network," *Canadian Journal of Mathematics* **8** (1956), pages 399–404. 150

[8] M. T. Goodrich and R. Tamassia. *Data Structures and Algorithms in Java*, John Wiley and Sons, Inc., 2001.

[9] K. Mehlhorn and St. Nadher. *The LEDA Platform of Combinatorial and Geometric Computing*, Cambridge University Press, 1999. (see http://www.mpi-sb.mpg.de/LEDA/leda.html) 116

[10] J. Orwant, J. Hietaniemi and J. Macdonald. *Mastering Algorithms with Perl*, O'Reilly, August 1999.

[11] J.G. Siek, L-Q. Lee and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual* Addison-Wesley, 2001. (see http://www.boost.org) 116

[12] T. A. Standish. *Data Structures in Java*^{TM}, Addison-Wesley, 1998. 97

Index