Lecture notes for 2nd term of module

Foundations of Computer Science

Martín Escardó

School of Computer Science
University of Birmingham

This is the version of 11th January 2005.

You have been given the version of 11th January 2005.
Any significant differences will be documented in Appendix B.

Contents

Chapter 1

Introduction

In this part of the module we study *data structures* and *algorithms*. You have already seen some data structures (e.g. arrays, lists, stacks, queues) and some algorithms (e.g. linear search and binary search) in the first term.

1.1 Algorithms as opposed to programs

There is a difference between *algorithm* and *program*. An algorithm is a mechanical procedure written in such a way that *human beings* can understand it, but a program is a mechanical procedure written in such a way that a *computer* can execute it. Since computers are quite inflexible compared to the human mind, programs usually contain more details than algorithms. Because in this module we want to ignore such details, we consider the design of algorithms rather than programs. Programming is the topic of the software workshop module. In fact, we will often see the same topics in both modules from different perspectives. Very often, however, we will write down actual programs in order to clarify certain aspects. But, as discussed above, the task of *implementing* the algorithms discussed here as computer programs is left to the software workshop module.

1.2 Fundamental questions of interest about algorithms

Given an algorithm, we are led to ask:

1. What is it supposed to do?
2. Does it really do what it is supposed to do?
3. How efficiently does it do it?

The technical jargon is:

1. Specification.
2. Verification.
3. Performance analysis.

This may sound a bit abstract, so let's consider an example.

1.3 An example: searching

One important problem in computing is that of *locating information*. More succinctly, this is problem is known as *searching*. The information to be searched has to be *represented* (or *encoded*) somehow. This is were *data structures* come in. As you have already seen, in a computer, everything is ultimately represented as a sequence of binary digits (bits). But this is too low level for our purposes. What we wish is to develop and study powerful data structures which are closer to the way we think (or at least more structured than mere sequences of bits). After all, it is us, humans, who have to develop software systems — computers merely run them.

After we have chosen a suitable representation, the information has to be processed. In this case, the process is that of searching. This is where algorithms come in. To begin with, let's consider

1. One particular representation and a variation, and

2. two algorithms for that representation.

In order to simplify matters, let's suppose that we want to search a collection of integer numbers (we could have equally well deal with strings of characters or any other data type of interest). As you have already seen, arrays are one of the simplest possible ways of representing collections.

It will be convenient to be able to write down arrays on paper. We just write its items in order, separated by commas and enclosed by square brackets. Thus,

$$[1, 4, 17, 3, 90, 79, 4, 6, 81]$$

is an example of an array. Let's call this array a:

$$a = [1, 4, 17, 3, 90, 79, 4, 6, 81]$$

Then the array a has 9 items, and hence we say that its size is 9. In everyday life, we usually start counting from 1. When we work with arrays, we more often start from 0. Thus, for our array a, its positions are $0, 1, 2, \ldots, 7, 8$. The element in the 8^{th} position is 81. Here we use the notation $a[8]$ to denote this element. More generally, for any integer i denoting a position, we write $a[i]$ to the denote the element in the i^{th} position. This position i is called an *index* (the plural is *indices*). Then, for example, $a[0] = 1$ and $a[2] = 17$.

If we ask where 17 is in the array a, the answer is 2. If we ask where 91 is, the answer is *nowhere*. Because we are not using -1 as a possible index (as we start counting from 0), we may, as we do, use the number -1 to represent *nowhere*. Other (perhaps better) conventions are possible, but we'll stick to this here.

1.4 Specification of our search problem

We are now ready to formulate the specification of our search problem:

> *Given an array a and integer x, find an integer i such that*
>
> 1. *if there is no j such that $a[j]$ is x, then i is -1,*
> 2. *otherwise, $a[i]$ is x.*

The first clause says that if x doesn't occur in the array then i should be -1, and the second says that if it does then i should be a position where it occurs. If there is more than one position where x occurs, then this specification allows you to return any of them — this would be the case e.g. if a were $[17, 13, 17]$ and x were 17. Thus, the specification is ambiguous. Hence different algorithms with different behaviours can satisfy the same specification — for example, one algorithm may return the smallest position at which x occurs and an another the largest. There is nothing wrong with ambiguous specifications. In fact, they do occur in practice often.

1.5 A possible algorithm: linear search

In the lecture we used a flowchart to convey the algorithm. Here we use a different way of expressing the same algorithm, which is very close to conventional languages such as C and Java, but still missing some details that a computer usually needs.

```
We assume that an array a of size n and a key x are given.
We are supposed to find an i satisfying the above specification.

For i = 0,1,...,n-1,
    if a[i] is equal to x
       then we have found a suitable i and hence we stop.

If we reach this point
   then x is not in a and hence we terminate with i = -1.
```

The ellipsis "..." is potentially ambiguous, but we, human beings, know exactly what is meant here and so we don't worry. In a language such as C or Java, one would write something like

```
for (i = 0; i < n; i++)
    if (a[i] == x)
       return i;

return -1;
```

In the case of Java, this would be within a method of a class, and more details are needed, such as the parameter a for the method and a declaration of the auxiliary variable i. In the case of C, this would be within a function declaration, and similar missing details are needed.

In this example, it is rather easy to see that the algorithm satisfies the specification — we have to just observe that, because we start counting from zero, the last position of the array is its size minus one. *If we forget this, and let i run from 0 to n instead, we get an incorrect algorithm.* The practical effect of this mistake is that the execution of this algorithm gives rise to an error when the item to be located in the array is actually not there, because a non-existing location is attempted to be accessed. Depending on the particular language, operating system and machine you are using, the actual effect of this error will be different. For example, in C running under unix in hardware that supports memory protection, you may (but won't necessarily) get execution aborted followed by a message "segmentation fault". In Java, you'll always get an error message.

For more elaborate specifications and/or algorithms, the fact that an algorithm satisfies its specification may be not obvious at all. In this case, we need to spend some effort verifying that the algorithm is indeed correct — if it is at all (and very often it is not). In general, testing can be enough for finding out that the algorithm is incorrect. However, since the number of inputs for most algorithms is infinite in theory and huge in practice, more than just testing is needed in order to be sure that an algorithm satisfies its specification. We need a *correctness proof*. Although we'll discuss proofs in this module, we'll do that in a rather informal way (but, of course, we'll attempt to be rigorous). The reason is that we want to emphasize data structures and algorithms. You'll meet formal verification techniques in other modules.

1.6 A more sophisticated algorithm: binary search

Can we improve the performance of the previous? In the worst case, searching an array of size n takes n steps. On average, it takes $n/2$. For a big collection of data, e.g. the world wide web, this is unacceptable in practice. Thus, it is necessary to organize the collection in such a way that a more efficient algorithm is possible. As we shall see, there are many possibilities. And the more we demand in terms of efficiency the more complicated the data structures representing the collections become. Here we consider one of the simplest: we represent collections by arrays, again, but we enumerate the elements in ascending order.

(The problem of obtaining an ordered list from any given list is known as *sorting* and is studied in a later chapter.)

Thus, instead of working with the previous example $[1, 4, 17, 3, 90, 79, 4, 6, 81]$, we would work with $[1, 3, 4, 5, 6, 17, 79, 81, 90]$, which has the same items but listed in ascending order.

You've met the following algorithm in the first part of this module (1st term).

```
We assume that a sorted array a of size n and a key x are given.
We are supposed to find an i satisfying the above specification.

Work with integers l and r initially set of 0 and n-1 respectively.

Work also with an integer m.

While l < r-1
     set m to the integer part of (l+r)/2, and
     if x < a[m]
        then      set r to m,
        otherwise set l to m.

if a[l] is equal to x
   then      set i to l,
   otherwise set i to -1.
```

The idea is to split the array into two segments, one going from l to m and the other from m to r, where m is the position half way from l to r, and where, initially, l and r are the leftmost and rightmost positions of the array. Because the array is sorted, the item to be searched can be in only one of the two parts. Because the size of the sub-array going from locations l to r is halved at each iteration of the while-loop, we only need $\log_2 n$ steps in the worst case (when the item we are looking for is not in the array). To see this the different in practice, notice that $\log_2 1000000$ is approximately 20, so that for an array of size 1000000 only 20 iterations are needed in the worst case of the binary-search algorithm, whereas 1000000 are needed in that of the linear-search algorithm.

This time, it is not obvious that we have taken proper care of the boundary condition in the while loop. Also, strictly speaking, this algorithm is not correct because it doesn't work for the empty array (that of size zero) — can you fix this problem in an elegant way? Apart from that, it is correct — first try to convince yourself. After you succeed, try to explain your argument to a colleague. Finally, try to *write down* a convincing argument. Most programmers stop at the first stage. But experience shows that it is only when we attempt to write down seemingly convincing arguments that we actually find subtle mistakes. It does require a lot of training to be able to write down convincing arguments, and, sadly, many programmers are simply unable to do that (in my opinion simply for lack of training).

1.7 Summary

This part of the module is about data structures and algorithms. Data structures are needed in order to represent information so that algorithms can conveniently manipulate it. The fundamental topics of practical interest concerning algorithms are specification, verification and performance analysis. We'll look at each of these for topics for each problem at hand.

The data structures covered in this module are lists, binary search trees, heap three, priority queues, hash tables and graphs.

1.8 Module web page

You can find information about this part of the module at

    http://www.cs.bham.ac.uk/˜mhe/teaching.html which is regularly updated.

1.9 Textbooks

You *must* complement the material of these notes with a textbook. For instance, we have lectures about linked lists, which are examinable and are not covered in these notes.

Some text books are suggested in the syllabus page for the module. Students often ask me: what is the best text book? The simple answer is that there is no best book and that there are very many excellent books. I suggest that you go to the main library and browse the shelves of books on data structures and algorithms. If you like any of them, order a copy for you. But make sure that all topics listed in the above summary are covered.

This is a classical topic, so there is no need to buy a book published recently. Books published in the seventies are quite good, and good books continue to be published every year. The reason is that this is a fundamental topic taught in all computer science university degrees.

I strongly suggest that you buy some book of your choice (possibly among the ones suggested in the syllabus page). In any case, if you don't own a book, you should go to the library. I emphasize that these notes contain only a summary of the topics covered in the lectures, and that you should consult a textbook for full details. For example, some students complain that the explanation of Dijkstra's algorithm for finding shortest paths in graphs, given here, is very concise. Indeed it is. You should read a book for more details (and go to the lectures).

Chapter 2

Binary search trees

We have already seen a couple of searching algorithms where the search space is represented as an array. We now study another important way of storing data which allows us to find a particular item in as little time as possible. This topic will take several lectures. In practice, when efficiency is a crucial factor, further elaborations of this way of storing data are used, consisting of so-called B-trees and AVL trees, among others, at the expense of using more sophisticated algorithms.

2.1 The problem: Searching

As we have already seen, many applications involve finding a particular item in a collection of data. If the data is stored as an unsorted list or array, then to find the item in question, one obviously has to check each entry (until the correct one is found or the collection is exhausted). On average, if there are n items, this will take $n/2$ attempts and in the worst case, all n items will have to be checked. If the collection is large, a lot of time will be spent doing this (think, for example, of the collection of items accessible via the World Wide Web).

2.2 Search keys

The solution considered here consists of two parts. Firstly, if the items are labelled via comparable *keys*, one can store them such that they are sorted already (being 'sorted' may mean different things for different keys, and which key to choose is an important design decision).

In our examples, the search keys will often be numbers, for simplicity, but other choices occur in practice. For example, the comparable keys can be words. In this case, comparability usually refers to the alphabetical order. If w and t are words, we write $w < t$ to mean that w precedes t in the alphabetical order. If $w = bed$ and $t = sky$ then the relation $w < t$ holds, but this is not the case if $w = bed$ and $t = abacus$. An example of a collection to be searched is a dictionary. Each entry of the dictionary is a pair consisting of a word and a definition. The definition is a sequence of words and punctuation symbols. The search key, in this example, is the word (to which a definition is attached in the dictionary entry). Thus, abstractly, a dictionary is a sequence of entries, were an entry is a pair consisting of a word and definition. This is what matters from the point of view of the search algorithms we are going to consider. Notice the use of the word "abstract": what we mean is that we abstract from details that are irrelevant from the point of view of the algorithms. For example, a dictionary usually comes in the form a book, which is a sequence of pages — for us, the distribution of dictionary entries into pages is an accidental feature of the dictionary. All what matters for us is that the dictionary is a sequence of entries. So "abstraction" means "getting rid of irrelevant details".

For our purposes, only the search key is important, and we will ignore the fact that the entries of the collection will typically be more complex objects (as in the example of a dictionary or a phone book).

Secondly, one can employ a data structure to hold the items which performs best for the typical application. There is no easy answer as to what the best choice is—the circumstances have to be inspected, and a decision has to be made based on that. A very common solution which we study in this chapter is based on data structures called *trees*. (Recall that you have seen examples of trees in the first term for arithmetic expressions.)

2.3 Trees

A **tree** consists of *nodes* and *branches*. An example is given in Figure 2.1. Nodes are usually *labelled* with a data item (typically a *search key*).

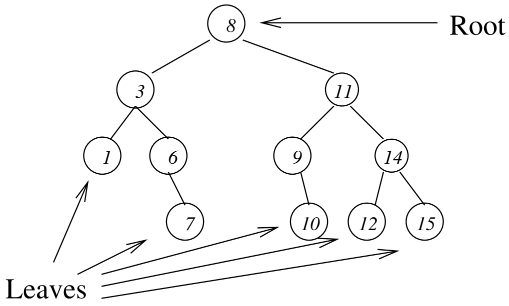


Figure 2.1: A tree

In our examples, we will use nodes labelled by integers, but one could just as easily have chosen something else, *e.g.* strings of characters. We also refer to the label of a node as its *value*. For the moment, we assume that each possible value occurs at most once in the tree. It is easiest to represent trees pictorially. A formal definition will be given later.

In order to talk about trees it is convenient to have some terminology: We demand that there is always a unique 'top level' node, the *root*. (Our trees are typically thought of as upside-down with the root forming the top level.) Given a node, every node on the next level 'down' which is connected to the given node via a branch is a *child* of that node. In the example, the children of 8 are 3 and 11. Conversely, the node (there's at most one) on the level above connected to the given node (via a branch) is its *parent*. So 11 is the parent of 9 and 14. Nodes that have the same parent are known as *siblings*—siblings are always on the same level.

If a node is the child of a child of ... of a another node then we say that the first node is a *descendent* of the second node. Conversely, the second node is an *ancestor* of the first node. Nodes which do not have any children are known as *leafs*. (Think of the upside-down tree.)

A *path* is a sequence of connected branches from one node to another (branching is not allowed). Trees have the property that for every node there is a unique path connecting it with the root. (In fact, that is one possible definition of a tree.) The *depth* or *level* of a node is given by the length of this path. Hence the root has level 0, its children have level 1, *etc.* The maximal length of a path in a tree is also called the *height* of the tree. A path of maximal length always goes from the root to a leaf. The tree in the picture above has height 3, a tree consisting just of one node has height 0, and we define (somewhat artificially) the empty tree to have height -1. The *size* of a tree on the other hand is given by the number of nodes it contains. The size of the tree in the picture is 11.

2.4 Binary trees

Binary trees are the ones typically used for our purposes. A *binary tree* is a tree where every node has at most two children. This is not a formal definition, since we haven't said formally what a tree is. We can define binary trees "inductively" via the following rules.

Definition. A *binary tree* is either

(Rule 1) empty,

(Rule 2) or consists of a node and two binary trees, the *left* and *right* subtree.

Rule 1 is called the "base case" and Rule 2 is called the "induction step". This definition may seem circular, but it actually isn't. You can imagine that the (infinite) collection of (finite) trees is created in a sequence of days. Day 0 is when you "get off the ground". You declare an empty tree to exist by fiat, using Rule 1. At any other day, you are allowed to use the trees that you have created in the previous days, using Rule 1, in order to construct new trees. Thus, for example, at day 1 you can create exact the trees that have a root but no children (i.e. both the left and right subtrees are the empty tree, created at day 1). At day 2 you can use the empty tree and the one-node tree, to create more trees. (Question: what trees are created at day 2. How about day 3? As another exercise, consider how you would prove that the tree in the picture above is in fact a binary tree. At which day was it created?) Thus, the binary trees are the objects created by the above two rules in a finite number of steps. The height of a tree, defined below, is the number of days it takes to create it using the above two rules, where we assume that only one rule is used per day, as we have just discussed.

2.5 The height and size of a binary tree

Binary trees are somewhat limited (which is good for the applications we have in mind) when it comes to relating their size n and height h. The maximum height of a binary tree with n nodes is $(n-1)$—it is reached when all nodes have precisely one child, forming something that looks like a chain.

On the other hand, assume we have n nodes and want to build a tree with minimal height out of them. We achieve this by 'filling' each position on each successive level, starting from the root. In other words, we make sure that the successive subtrees are non-empty, until we run out of nodes. It does not matter where we place the nodes on the last (bottom) level of the tree. We say that such trees are *perfectly balanced*, and we will see below why they are optimal for our purposes—basically, searching in a binary tree takes as many steps as the height of the tree, so minimizing the height minimizes the time spent searching in the tree.

Let us calculate how many nodes fit into a binary tree of a given height h. Calling this function f, we obtain:

h	$f(h)$
0	1
1	3
2	7
3	15

In fact, it is not difficult to see that $f(h) = 1 + 2 + 4 + \cdots + 2^h$. This, in turn, is equal to $2^{h+1} - 1$. (This can be proved by induction on the definition of a binary tree as follows. The base case applies to the empty tree, and indeed, we can store $2^{-1+1} - 1 = 2^0 - 1 = 1 - 1 = 0$ nodes in an empty tree. Now for the induction step: A tree of height $h + 1$ has two subtrees of height h, and by the induction hypothesis, each of these subtrees can store $2^{h+1} - 1$ nodes. In addition, there is the new root which adds an extra node, so all in all a tree of height $h + 1$ can have as many as $2 \times (2^{h+1} - 1) + 1 = 2^{h+2} - 2 + 1 = 2^{(h+1)+1} - 1$ nodes, which is what we wanted to show.)

Hence a perfectly balanced tree consisting of n nodes has height approximately $\log n$. This is good, because $\log n$ is very small even for relatively large n, as the following table shows:

n	$\log n$
1	0
10	4
$1,000$	10
$1,000,000$	20

Below we will describe how to use these trees to hold data such that any search has at most as many steps as the height of the tree. Therefore for perfectly balanced trees we have reduced the search time considerably as the table to the left demonstrates. It is not easy, however, to create perfectly balanced trees, as we shall see.

We shall be using logarithms (to the base 2) rather often in our lectures. If you are not very familiar with logarithms (or not familiar at all), don't despair. An easy and intuitive way of understanding what a logarithm is is the following. The logarithm of a number n is the number of times by which we can divide n by 2 before it becomes 1 or a number smaller than 1. So for example, to calculate the logarithm of 20 we perform successive divisions by 2, getting the sequence 20, 10, 5, 2.5, 1.25, 0.625. Since it took 5 divisions to get a number smaller than one, the logarithm is 5. Sometimes, we need rules for calculating with logarithms. In Chapter A you find some formulas that we use often.

This doesn't account for fractional logarithms (which you can get using a calculator), but, on the other hand, we are hardly ever interested in fractional logarithms, as we shall be using them for counting (e.g. the minimum height of a binary tree, as we have done above).

2.6 The size of a binary tree

Here is an example for a simple recursive algorithm for binary trees. Given a tree, we want to know its size, i.e. the number of nodes it contains.

The base case is very simple once again: The empty tree has size 0. The inductive step works as follows: If the tree is assembled from a left subtree l, a right subtree r and a node then its size will be the size of l plus the size of r plus 1. Let us use `EmptyTree` for the empty tree and $\mathtt{MkTree}(v, l, r)$ for a tree consisting of a node v, a left subtree l and a right subtree r. Then we can define the function size, which takes a tree and returns its size, as follows:

$$\begin{aligned} \mathrm{size}(\mathtt{EmptyTree}) &= 0 \\ \mathrm{size}(\mathtt{MkTree}(v, l, r)) &= 1 + \mathrm{size}(l) + \mathrm{size}(r) \end{aligned}$$

This is an inductive definition of a function, which can be regarded as a "recursive algorithm".

2.7 Recursive algorithms

Sometimes students have difficulties with recursion. A source of confusion is that it appears that "the algorithm calls itself". This way of putting things, although suggestive, can be misleading the first time we encounter recursion.

The algorithm itself is a passive entity, which actually can't do anything at all, let alone call itself. What happens is that a *processor* (which can be a machine or a person) *executes* the algorithm. So what goes on when a processor executes a recursive algorithm such as the above? An easy way of understanding this is to imagine that whenever a recursive call is encountered, new processors are given the task with a copy of the same algorithm.

For example, suppose that I (the first processor in this task) want to compute the size of a tree t using the above recursive algorithm. Then, according to the above algorithm, I first check whether it is empty. If it is, I return zero and finish my computation. If not, then my tree t has to have left and right subtrees l and r and some value v at its root node; that is, it is of the form $t = \mathtt{MkTree}(v, l, r)$. In this case, I ask two students, say John and Mary, to execute the same algorithm, but for the trees l and r. When they finish, say with results m and n respectively, I compute $m + n + 1$, because my tree has a root node in addition to the left and right subtrees. John and Mary themselves may delegate executions of the same algorithm, with further subtrees, to other people. Thus, the algorithm is not calling itself, what happens is that there are many people running copies of the same algorithm.

In our example, in order to make things understandable, we assume that each person executes a single copy of the algorithm. However, the same processor, with some difficulty, can impersonate several processors, in such a way that a recursive algorithm can be executed by a single processor achieving the same result as the above execution involving many processors. This is achieved via the use of a stack that keeps track of the various positions of the same algorithm that are currently being executed — but this knowledge is not needed for our purposes.

2.8 Binary trees (of integer items) as Java data structures including a size method

In the Java program given below, a tree is given by a particular node, namely the root. The implementation of the class `Node` reflects the above considerations for representing trees, and the method `size` is just the above definition.

```
class BinTree
{ private Node root;
  public BinTree()
  { root = null;
  }
  public int size()
  { return Node.size(root);
  }

  private class Node
  { private int value;
    private Node left;
    private Node right;

    Node(int v, Node l, Node r)
    { value=v;
      left=l;
      right=r;
    }

    static int size(Node n)
    { return (n==null)?
        0 : size(n.left)+size(n.right)+1;
    }
  }
}
```

2.9 Binary search trees

This is a solution to our problem: We will consider how to store our data using a binary tree such that searching for a particular item takes a minimal effort. The underlying idea is simple: At each node, we want the value of this node to tell us whether we should search in the corresponding left or right subtree. Hence we define:

Definition. A *binary search tree* is a binary tree that is either empty or satisfies the following conditions:

- All values occurring in the left subtree are smaller than that of the root.
- All values occurring in the right subtree are larger than that of the root.
- The left and the right subtree are themselves binary search trees.

An example is given in Figure 2.1.

2.10 Building binary search trees

When building a binary search tree it is easier to start with the root and then insert new nodes as needed. The following cases arise:

- If the given tree is empty then just assign the new value to the root, the left and right subtrees remain empty.
- If the given tree is non-empty then to insert a node with value v proceed as follows:
  - If v is equal to the value of the root, do nothing (or raise an exception).
  - If v is smaller than the value of the root: Insert v into the left subtree.
  - If v is larger than the value of the root: Insert v into the right subtree.

Using the same notation as before, we thus get:

$$\begin{array}{rcl} \mathrm{insert}(v, \mathtt{EmptyTree}) & = & \mathtt{MkTree}(v, \mathtt{EmptyTree}, \mathtt{EmptyTree}) \\ \mathrm{insert}(v, \mathtt{MkTree}(w,l,r)) & = & \begin{cases} \mathtt{MkTree}(w,l,r) & \text{if } v = w, \\ \mathtt{MkTree}(w, \mathrm{insert}(v,l), r) & \text{if } v < w, \\ \mathtt{MkTree}(w, l, \mathrm{insert}(v,r)) & \text{if } v > w. \end{cases} \end{array}$$

Note that the node added is always a leaf. The resulting tree is once again a binary search tree. This can be rigorously proved via an inductive argument.

There is something that should be emphasized here. The procedure that we are describing creates a new tree out of a given tree, with the value inserted at the right position. In a way, the given tree is not modified, it is just inspected. When the tree represents a large database, it would be better to modify the given tree rather than to construct a new tree again. This involves the use of pointers (as you have seen in linked lists). For the moment, we shall not be concerned with such details. It seems preferable to introduce only a few concepts at a time.

2.11 An implementation of the insert method in Java.

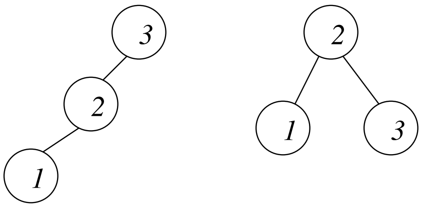
```
class BinTree
{ private Node root;

  public void insert(int v)
  { root=Node.insert(v,root)
  }

  private class Node
  { private int value;
    private Node left;
    private Node right;

    static Node insert(int v,Node n)
    { if (n==null)
        return new Node(v,null,null);
      else
      { if (v<n.value)
          n.left=insert(v,n.left);
        else n.right=insert(v,n.right);
        return n;
      }
    }
  }
}
```

2.12 Searching a binary search tree

Searching a binary search tree is not dissimilar to the process performed when inserting an item: Compare the item that you are looking for with the root, and then 'push' the comparison down into the left or right subtree depending on the result of this comparison, until a match is found or a leaf is reached. This takes at most as many comparisons as the height of the tree. At worst, this will be the number of nodes in the tree minus one, as we have seen.

But how many comparisons are required on average? In order to answer this question, we need the average height of a binary tree. This is calculated by taking all binary trees of a given size n and calculating their height, which is by no means easy. The trouble is that there are many ways of building the same binary search tree by successive insertions. As we have seen above, perfectly balanced trees achieve minimal height for a given number of nodes, and it turns out that the more balanced a tree, the more ways there are of building it.

This is demonstrated in the figure below: The only way of getting the tree on the left hand side is by inserting 3, 2, 1 into the empty tree in that order. The tree on the right, however, can be reached in two ways: Inserting in the order 2, 1, 3 or in the order 2, 3, 1.



Ideally, of course, one would only use well-balanced trees (they don't have to be perfectly balanced to perform better than binary search trees without restrictions) to keep the height minimal.

We won't do the actual calculations here (see [Aho, Hopcroft, Ullman, *Data Structures and Algorithms*, 1983], for more details). If we assume that all the possible orders in which the nodes might be inserted are equally likely then the average number of comparisons needed is $\log n$, since the average height of a binary search tree falls into that class. Inserting a node into a binary search tree also requires $\log n$ steps.

Aside. Somewhat surprisingly, this is quite a bit better than the average height of a binary search tree with n nodes, where we don't assume that the tree has been built as a search tree. The average height of a binary tree is approximately $\sqrt{n}$.

2.13 Deleting nodes from a binary search tree

This is more complicated than one might assume at first sight. It turns out that the following algorithm works as desired:

- If the node in question is a leaf, just remove it.
- If only one of the node's subtrees is non-empty, 'move up' the remaining subtree.
- If the node has two non-empty subtrees, find the 'left-most' node occurring in the right subtree (this is the smallest item in the right subtree). Use this node to overwrite the one that is to be deleted. Replace the left-most node by its right subtree, if this exists); otherwise just delete it. Of course, we are using the fact that the left-most node has to have an empty left subtree.

2.14 A Java implementation of the deletion algorithm

```
class BinTree
```

```
{ private Node root;

  public void delete(int v)
  { root=Node.delete(v,root);
  }

  private class Node
  { private int value;
    private Node left;
    private Node right;

    static Node delete(int v,Node n)
    { if(n!=null)
      { if(n.value==v)
        { if(n.left==null) return n.right;
          else if(n.right==null)
            return n.left;
          else
          { n.value=fetch_left(n.right);
            n.right=del_left(n.right);
          }
        }
        else if(v<n.value)
          n.left=delete(v,n.left);
        else n.right=delete(v,n.right);
      }
      return n;
    }

    static int fetch_left(Node n)
    { while(n.left!=null) n=n.left;
      return n.value;
    }

    static Node del_left(Node n)
    { if(n.left!=null)
      { n.left=del_left(n.left);
        return n;
      }
      else return n.right;
    }
  }
}
```

Deletion of a node requires the same number of steps as searching or inserting a new node.

2.15 Searching again

Algorithms can be expressed in many ways. Here is a concise description of the search algorithm that we have already discussed.

> In order to search for a value v in a binary search tree t, proceed as follows. If t is empty then v doesn't occur in t and hence we stop. Otherwise, if v is equal to the root of t then v does occur in t and hence we again stop. If, on the other hand, v is smaller than the root, then, by

definition of binary search tree, it is enough to search on the left subtree of t. Hence replace t by its left subtree and carry on in the same way. The case in which v is bigger than the root is handled in a similar way.

Notice that such a description of an algorithm embodies both the steps that need to be carried out *and* the reason why this gives a correct solution to the problem. This way of describing algorithms is very common when we don't intend to run them in a computer. When we do, further refinement is necessary. A next step in the refinement could be:

```
boolean isin(value v, tree t) {
  if (t is empty)
     return False;
  else
     if (v is equal to the root node of t)
        return true;
     else
        if (v < root node of t)
           return isin(v, left subtree of t);
        else
           return isin(v, right subtree of t);
}
```

This is another example of a recursive algorithm. The recursion eventually terminates, because every recursive call takes a smaller tree, so that we eventually find what we are looking for or reach an empty tree. In this example, the recursion is easily transformed into a while-loop:

```
boolean isin(value v, tree t) {
  while (t is not empty  and  v is different from the root of t)
        if (v < root node of t)
           t = left subtree of t;
        else
           t = right subtree of t;

  if (t is empty)
     return false;
  else
     if (v is equal to the root of t)
        return true;
     else
        return false;
}
```

Using boolean expressions, the part below the while-loop can be simplified to

```
  return ((t is not empty) and (v is equal to the root of t));
```

The idea is that each of the expressions is evaluated to a boolean value, then its logical conjunction ("and") is taken, and finally the obtained truth value is returned.

2.16 Primitive operations on trees

We have deliberately omitted implementation details for trees, and you have been referred to software workshop module which you are taking (different students study different programming languages). However, in order to have a more rigorous approach, still ignoring implementation details, we formally introduce the primitive operations that we have been using for manipulating trees in algorithms.

Firstly, we have the *constructors*, which are used to *build* trees:

$$\begin{aligned} \texttt{EmptyTree} &: \texttt{tree}, \\ \texttt{MkTree} &: \texttt{value} \times \texttt{tree} \times \texttt{tree} \to \texttt{tree}. \end{aligned}$$

The first constructs an empty tree. The second, given a value v and two trees l and r, constructs a tree, which we write $\texttt{MkTree}(v, l, r)$, with root v and left and right subtrees l and r respectively.

For convenience, we define a derived constructor

$$\texttt{Leaf} \quad : \quad \texttt{value} \to \texttt{tree}$$

that creates a tree consisting of a single node, which is the root and the unique leaf of the tree at the same time:

$$\texttt{Leaf}(v) = \texttt{MkTree}(v, \texttt{EmptyTree}, \texttt{EmptyTree}).$$

For example, the tree displayed Figure 1 can be constructed as follows:

```
t = MkTree(8,
          MkTree(3,
                 Leaf(1),
                 MkTree(6,
                       EmptyTree,
                        Leaf(7)
                       )
               ),
          MkTree(11,
                 MkTree(9,
                        EmptyTree,
                        Leaf(10)
                       ),
                 MkTree(14,
                        Leaf(12),
                        Leaf(15)
                       )
                 )
          )
```

where we have used indentation for clarity — an equivalent unreadable version is the following:

```
t=MkTree(8,MkTree(3,Leaf(1),MkTree(6,EmptyTree,Leaf(7))),MkTree(11,
MkTree(9,EmptyTree,Leaf(10)),MkTree(14,Leaf(12),Leaf(15))))
```

Secondly, we have *destructors*, which are used in order to break a previously constructed tree into smaller pieces:

$$\begin{aligned} \texttt{IsEmpty} &: \texttt{tree} \to \texttt{boolean} \\ \texttt{Root} &: \texttt{tree} \to \texttt{value} \\ \texttt{Left} &: \texttt{tree} \to \texttt{tree} \\ \texttt{Right} &: \texttt{tree} \to \texttt{tree} \end{aligned}$$

The first tests which of the above two constructions apply. For a tree that is not empty, the other three destructors extract the root and the left and right subtrees respectively. For example, for the tree t defined above, we have that

```
Root(Left(Left(t)) = 1,
```

but the expression

```
Root(Left(Left(Left(t))))
```

doesn't make sense because

```
Left(Left(Left(t))) = EmptyTree
```

and the empty tree doesn't have a root. In a language such as Java, this typically would rise an exception, whereas in a language such as C, if the programmer is not careful enough, this would cause an unpredictable behaviour (if you are lucky, a core dump will be produced and the program will be aborted with no further harm).

The following equations should be obvious:

```
Root(MkTree(v,l,r)) = v
Left(MkTree(v,l,r)) = l
Right(MkTree(v,l,r)) = r
IsEmpty(EmptyTree) = True
IsEmpty(MkTree(v,l,r)) = False
```

The following makes sense only under the assumption that t is a non-empty tree:

```
MkTree(Root(t),Left(t),Right(t)) = t
```

It just says that if we break apart a non-empty tree and use the pieces to build a new tree, then we get the same tree back.

A data type for which we exhibit the constructors and destructors and describe their behaviour (using e.g. equations as above), but for which we explicitly hide the implementation details, is called an *abstract data type*. The concrete data type used in the implementation is called a *data structure*. For example, the usual data structures used to implement the tree data type are records and pointers — but other implementations are possible. One advantage of abstract data types is that we can develop algorithms without worrying about representation details of data. In fact, everything is ultimately represented as a sequence of bits in a computer, as we have already mentioned, but we don't want to think in such low level terms.

2.17 Searching yet again

Using these primitive functions on trees, our searching algorithm can be now written as

```
boolean isin(value v, tree t) {

if (EmptyTree(t))
   return False;
else
   if (v = Root(t))
      return true;
   else
      if (v < Root(t))
         return isin(v, Left(t));
      else
         return isin(v, Right(t));
}
```

Recursive algorithms of this kind have a slight variant known as *inductive* definitions, which take the form of equations (for the base case and the inductive step). The inductive definition corresponding to the above recursive algorithm is the following:

```
isin(v,EmptyTree) = False
isin(v,MkTree(w,l,r)) = if (v = w)
                        then True
```

```
                        else if (v < w)
                             then isin(v,l)
                             else isin(v,r)
```

Here we have used a variant of the `if-then-else` control operator of the form

```
if <boolean-expression> then <value> else <value>
```

rather than the more usual

```
if <boolean-expression> then <command> else <command>
```

In C and Java, the form using expressions rather than commands have to be written in the following more obscure way:

```
<boolean-expression> ? <value> : <value>
```

For example, the expression

```
 (2 == 4) ? 1 : 3
```

evaluates to 3.

Inductive definitions are usually easier to read than recursive algorithms. However, languages such as C and Java don't have mechanisms for writing down inductive definitions (the usual mechanism is known as *pattern-matching*) and hence we first have to transform them into recursive algorithms. But other languages, such as Prolog, ML and Haskell do allow such definitions. In fact, the inductive definition displayed above is almost literally a program in ML or Haskell.

2.18 Deletion revisited

We have already given an algorithm for deleting nodes of a binary search tree in a rather concise way. Here is a more detailed algorithm:

```
tree delete (value v, tree t) {
  if (IsEmpty(t))
     return t;    // because there is nothing to be deleted
  else
     if (v < Root(t))
        return MkTree(Root(t), delete(v,Left(t)), Right(t));
     else
     if (v > Root(t))
        return MkTree(Root(t), Left(t), delete(v,Right(t)));
     else
     // (v = Root(t))
        if (IsEmpty(Left(t)))
           return Right(t);
        else
           if (IsEmpty(Right(t)))
              return Left(t);
           else
           // both subtrees non-empty: difficult case
              return MkTree(smallest_node(Right(t)),
                            Left(t),
                            remove_smallest_node(Right(t));

}
```

This uses two sub-algorithms `smallest_node` and `remove_smallest_node`. Here is the first:

```
value smallest_node(tree t) {
// Precondition: t is a non-empty binary search tree

  if (IsEmpty(Left(t))
     return(Root(t));
  else
     return smallest_node(Left(t));
}
```

This algorithm assumes that the input tree t is non-empty. In fact, the first thing that it does is no extract the left subtree of t and then test whether it is empty. It is the responsibility of the programmer to ensure that the precondition is met when the algorithm is used — hence it is important to explicitly say what the precondition is. The algorithm uses the fact that, by definition of binary search tree, the smallest node of t is the leftmost node. The second sub-algorithm uses the same idea:

```
tree remove_smallest_node(tree t) {
// Precondition: t is a non-empty binary search tree

  if (IsEmpty(Left(t))
     return(Right(t));
  else
     return MkTree(Root(t),remove_smallest_node(Left(t)),Right(t));
}
```

2.19 Checking whether a given tree is a binary search tree

We have already given a definition of binary search tree. This definition is sufficiently precise so that people can understand it and work with it. It can be made even more precise so that machines can check whether or not a given binary tree is a binary search tree:

```
boolean isbst(tree t) {
  return (      IsEmpty(t)
           or
               (     allsmaller(Left(t),Root(t))
                 and allbigger(Right(t),Root(t))
                 and isbst(Left(t))
                 and isbst(Right(t))
               )
         );
}

boolean allsmaller(tree t, value v) {

return (    IsEmpty(t)
        or
          (    Root(t) < v
           and allsmaller(Left(t),v)
           and allsmaller(Right(t),v)
          )
       );
}

boolean allbigger(tree t, value v) {
```

```
return (   IsEmpty(t)
         or
           (    Root(t) > v
            and allbigger(Left(t),v)
            and allbigger(Right(t),v)
           )
        );
}
```

This says that a binary tree is a binary search tree if it is either empty or else all nodes on the left subtree are smaller than the root, all nodes on the right subtree are bigger than the root, and both the right and left subtrees are themselves binary search trees.

2.20 Induction over binary trees

We have already seen an example of an inductive definition over trees. Here are another two, corresponding to notions that we have already discussed:

```
size(EmptyTree) = 0
size(MkTree(v,l,r)) = 1 + size(l) + size(r)

height(EmptyTree) = -1
height(MkTree(v,l,r)) = 1 + max(height(l),height(r))
```

You may wish to pause to think why we have decided to define the height of an empty tree to be -1 rather than 0. Sometimes it is necessary to establish some properties of trees such as

$$\texttt{height}(t) < \texttt{size}(t) \leq 2^{\texttt{height}(t)+1} - 1.$$

This is intended to hold for *all* binary trees. Since there are infinitely many trees, how could we possibly prove this? The answer lies in the fact that, although there are infinitely many trees, there are only *two* rules for creating trees, as we have seen:

1. **(base case)** `EmptyTree` is a tree.
2. **(induction step)** If `v` is a value and `l` and `r` are trees then `MkTree(v,l,r)` is a tree.

Thus, each tree is obtained by starting from the base case and then applying the induction step finitely many times. This gives rise to the following proof rule, which plays a key rôle in correctness proofs of algorithms that manipulate trees:

> **Induction principle:** In order to prove that a property holds for *all* binary trees, show that
>
> 1. **(base case)** it holds for the empty tree `EmptyTree`, and
> 2. **(induction step)** it holds for the tree `MkTree(v,l,r)` whenever it holds for given trees `l` and `r`.

It can be proved by induction that

$$\texttt{size}(t) \leq 2^{\texttt{height}(t)+1} - 1.$$

This is left as an exercise. We now prove by induction that

$$\texttt{height}(t) < \texttt{size}(t).$$

The base case is easy:

$$\texttt{height}(\texttt{EmptyTree}) = -1 < 0 = \texttt{size}(\texttt{EmptyTree}).$$

The induction step is not hard either:

1. Assume that we are given trees `l` and `r` such that

$$\begin{aligned} \texttt{height}(\texttt{l}) &< \texttt{size}(\texttt{l}), && (2.1)\\ \texttt{height}(\texttt{r}) &< \texttt{size}(\texttt{r}). && (2.2) \end{aligned}$$

2. We have to conclude that, no matter what `v` is,

$$\texttt{height}(\texttt{MkTree}(\texttt{v},\texttt{l},\texttt{r})) < \texttt{size}(\texttt{MkTree}(\texttt{v},\texttt{l},\texttt{r})).$$

   We do this as follows.

3. Combining the two equations (1) and (2), we get

$$\texttt{height}(\texttt{l}) + \texttt{height}(\texttt{r}) < \texttt{size}(\texttt{l}) + \texttt{size}(\texttt{r}).$$

4. Using this and the fact that

$$\max(\texttt{height}(\texttt{l}), \texttt{height}(\texttt{r})) \leq \texttt{height}(\texttt{l}) + \texttt{height}(\texttt{r}),$$

   we conclude that

$$\max(\texttt{height}(\texttt{l}), \texttt{height}(\texttt{r})) < \texttt{size}(\texttt{l}) + \texttt{size}(\texttt{r}).$$

5. Adding one to each side of the inequality, we get

$$1 + \max(\texttt{height}(\texttt{l}), \texttt{height}(\texttt{r})) < 1 + \texttt{size}(\texttt{l}) + \texttt{size}(\texttt{r}).$$

6. By the above definitions of height and size, we conclude that

$$\texttt{height}(\texttt{MkTree}(\texttt{v},\texttt{l},\texttt{r})) < \texttt{size}(\texttt{MkTree}(\texttt{v},\texttt{l},\texttt{r})),$$

   as we wished to conclude.

2.21 Printing the nodes of a binary search tree in order

The nodes of a binary search tree can be printed in ascending order as follows:

```
print_in_order(tree t) {

if (not IsEmpty(t)) {
   print_in_order(Left(t));
   print(Root(t));
   print_in_order(Right(t));
   }
}
```

2.22 Sorting

We shall present and discuss many sorting algorithms, that is, algorithms that arrange a collection of elements in order. We are ready to present one of them. Let's assume that the collection is given as an array.

```
sort(array a of size n) {

tree t = EmptyTree;
```

```
for i=0,1,...,n-1
    t = insert(a[i],t);

print_in_order(t);
}
```

Can you modify this algorithm so that instead of printing the values we store them back in the array in ascending order?

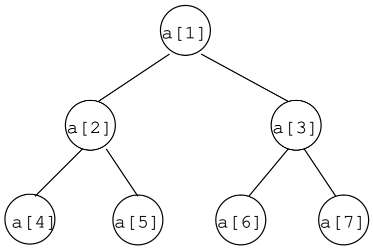
Chapter 3

Heap trees

3.1 Trees stored in arrays

In Chapter 2 we mentioned that binary trees can be stored with the help of a pointer-like structure, where one item contains references to its children. If the tree in question is a *complete* binary tree, there is an alternative.

Definition. A binary tree is *complete* if it has leaves on at most two adjacent levels and if all the leaves on the last level are placed as far to the left as possible. Intuitively, a complete binary tree is one that is obtained by filling the nodes starting with the root, then filling level one, then level 2, *etc*, always from the left, until one runs out of nodes. So complete binary trees are 'full' on every level but (possibly) the last one. Complete binary trees always have minimal height for their size, namely $\log n$, and they are always perfectly balanced (but not every perfectly balanced tree is complete in the sense of the above definition). But, more importantly, they can suitably be stored in arrays like this:



This way of storing a binary tree is not very efficient if the tree is not complete—we would reserve space in the array for each node, after all. For complete binary trees, on the other hand, an array gives us a very tight representation. Another reason this method is not very suitable for binary search trees is that inserting and/or deleting a node would mean that large portions of the array had to be shifted. Since keeping binary search trees balanced is a difficult problem, we cannot hope to adapt our algorithms for binary search trees so as to make storing them as arrays a viable option.

Notice that this time we have chosen to start with index 1 rather than 0. This has some advantages. The nodes on level i have indices $2^i, \cdots, 2^{i+1} - 1$. The level of a node with index i is $\log i$. The children of a node with index i, if they exist, have indices $2i$ and $2i + 1$. The parent of a child with index i has index $i/2$ (using integer division). This gives rise to the following simple algorithms:

```
boolean isroot(int i) {
  return truth value of "i = 1";
}

int parent(int i) {
```

```
  return i / 2;
}

int left(int i) {
  return 2 * i;
}

int right(int i) {
  return 2 * i + 1;
}

int level(int i) {
  return log(i);
}
```

3.2 Priority queues and heap-trees

The structure typically implemented as a complete binary tree is something called a *priority queue*. While most queues in every-day life operate on a first come, first served basis, it is sometimes important to be able to assign a *priority* to the items in the queue, and to always serve the item next that has the highest priority. And example for this would be a hospital casualty department, where life-threatening injuries have to be treated first.

It turns out that such queues can be implemented efficiently by particular binary trees, so called *heap-trees*. The idea is that what used to be the search key when we were talking about binary search trees is now replaced by a number giving the priority of the item in question (higher numbers meaning a higher priority in our examples). It is not necessary to keep a heap-tree sorted the way a binary search tree is to be able to insert and remove elements efficiently—this is due to the fact that we only ever want to remove one element, namely the one with the highest priority present, the root.

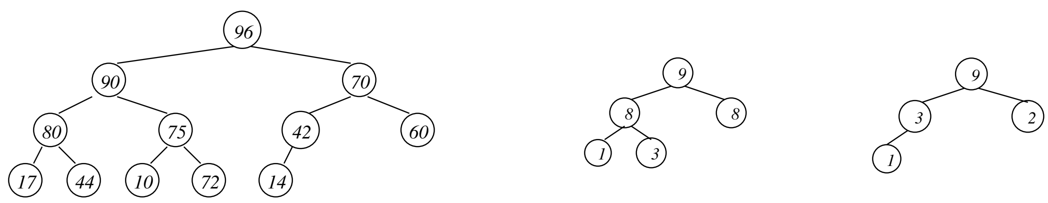
Definition. A *heap-tree* is a complete binary tree which is either empty or satisfies the following conditions:

- The priority of the root is higher than (or equal to) that of its children.
- The left and right subtree of the root are heap-trees.

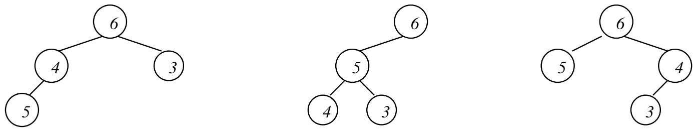
An alternative definition can be given as follows: A heap-tree is a complete binary tree such that the priority of every node is higher than (or equal to) that of all its descendents. Yet another way of saying the same thing is to say that for every path, as one goes down in the tree, the values become smaller.

The first obvious difference to binary search trees is that the biggest number now occurs at the root (rather than the right-most node). Secondly, whereas with binary search trees, the left and right child of a node played very different rôles, whereas now they will be interchangeable.

Here are a few examples



and here a few non-examples.



3.3 Basic operations on heap-trees

What would be expect to be able to do with a heap-tree? We would like to be able to remove the top-priority node (that is, 'serve' the next customer) and we would like to insert new nodes (or customers) with a given priority. Assuming that priorities are given by integers, we need operations

```
boolean heapempty();
int root();
delete_root();
insert(int v);
```

In order to develop algorithms using our array representation, we keep the largest position that has been filled so far, which is the same as the current number of nodes of the heap-tree. The first two are very easy:

```
int MAX = 100;     // Maximum number of nodes allowed.
int heap[MAX];     // Stores priority values of the nodes of the heap-tree.
int n = 0;         // Largest position that has been filled so far.

boolean heapempty() {
  return truth value of "n = 0";
}

int root() {
  if (heapempty())
     raise an exception;
  else
     return heap[1];
}
```

3.4 Inserting a new node

Inserting into a heap-tree is relatively straightforward. Because we keep of the largest position `n` which has been filled so far, we can insert the new element at position $\mathtt{n}+1$, provided there is still room in the array. This will once again give us a complete binary tree, but the heap-tree property might, of course, be violated now. Hence we may have to 'bubble up' the new element. This is done by comparing its priority with that of its parent, and if the new element has higher priority larger, then it is exchanged with its parent. We may have to repeat this process, but once we reach a parent that has bigger or equal priority, we can stop. Inserting a node takes at most $\log n$ steps because the number of times that we may have to 'bubble up' the new element depends on the height of the tree. Here is an algorithm

```
insert(int v) {
  if (n < MAX) {
     n = n + 1;
     heap[n] = v;
     bubble_up(n);
     }
  else
     raise an exception because we've run out of space;
```

```
}

bubble_up(int i) {
  while (not isroot(i) and  heap[i] > heap[parent(i)]) {
        swap heap[i] and heap[parent(i)];
        i = parent(i);
    }
  }
```

3.5 Deleting from a heap-tree

As we said before, for heap-trees we only want to be able to delete the root, i.e. the node with the highest priority. We're then left with something which isn't a binary tree at all. Just as we tried to insert by putting a node at the 'last' position, that is the right-most leaf, we use that node to fill the new vacancy at the root. This once again gives us a binary tree, but once again we have lost the heap property. We can now 'trickle down' the new root by comparing it to both its children and exchanging it for the largest. This process is then repeated until this element has found its place. Again, this takes at most $\log n$ steps.

Note that this algorithm does not try to be *fair* in the sense that if two nodes have the same priority, it is not necessarily the case that the one that has been waiting longer is removed first. This could be fixed by keeping some kind of time-stamp on arrivals, or by giving them numbers.

Chapter 4

Sorting

In computer science, 'sorting' usually refers to bringing a number of items into some order.

In order to be able to do this, we need to consider a notion of order on the set of items we are considering. For example, for numbers we can use the usual order and for strings the so-called *lexicographic* order, which is the one dictionaries and encyclopaedias use.

Usually, what is meant by *sorting* is that once the process is finished, there is a simple way of 'visiting' all the items in order, for example to print out the contents of a data-base. This may well mean different things depending on how the data is being stored: If all the objects are stored in an array a of size n, then

```
for i=1,...,n-1
   print(a[i])
```

would print the items in ascending number. If the objects are stored in a linked list, we would expect that the first entry is the smallest, the next the second-smallest, and so on. Often, more complicated structures such as binary search trees or heap-trees are used to sort the items, which can then be written into an array, or list, as desired.

Why is sorting so important? Having the items in order makes it much easier to *find* a given item. We saw that, by having the data items sorted as a binary search tree, we could reduce the average (and worst case) complexity of searching a particular item to $\log n$ steps whereas those had been in n steps without sorting. So if we often have to look up items, it pays off to sort the whole collection first. Imagine using a dictionary or phone book where the entries don't appear in some (known) order.

Therefore sorting algorithms are important tools for program designers. Different algorithms are suited to different situations (as we shall see, there is no 'best' sorting algorithm), and therefore a number of them will be introduced in this course. It is worth pointing out that we be far from covering *all* existing sorting algorithms—in fact, the field is still very much alive, and new developments take place all the time. However, the general strategies can now be considered to be well-understood (although for some sorting algorithms we still don't have an accurate measure of its performance), and a lot of what is newly done these days consists of tweaking existing principles.

4.1 Common sorting strategies

One way of organizing the various sorting algorithms is by classifying the underlying idea, or 'strategy'. Things you might come up with when asked for such ideas are:

selection sorting Find the smallest item, put it in the first position, find the smallest of the remaining items, put it in the second position ...

insertion sorting Take the items one at a time and insert them into an initially empty data structure such that at each stage, the data structure we are considering is sorted.

exchange sorting If two items are found to be out of order, exchange them.

enumeration sorting If we know that there are 27 items which are smaller than the one we are currently considering then its final position will be at number 28.

divide and conquer This is the recursive approach: split the problem up into smaller problems. The sorting of just one item is trivial. In this case, you have to put the sorted 'parts' together somehow.

All these strategies are based on *comparing* items and then rearranging them accordingly. We will later consider methods which make use of specific knowledge about the items that can occur which do not fall under this category. The ideas given above are also all based on the assumption that the items to be sorted all fit into the computer's internal memory, which is why we speak of *internal sorting algorithms*. These days, given the growing power of computers, *external* storing devices become less and less frequent in sorting. If not all the items can be stored in the internal memory at one time, different techniques have to be used. The underlying idea is to sort however many one can handle at a time and then carefully merge the results. We will not consider external sorting algorithms in any detail in this course. You can find some material on this topic in [Iain T. Adamson *Data Structures and Algorithms: A First Course*, Springer 1996] or [Donald E. Knuth, *The Art of Computer Programming: Sorting and Searching*, Addison-Wesley, 1998].

4.2 How many comparisons does it take?

A way of judging the complexity of a sorting algorithm is to count the number of comparisons it will carry out (as a function of the number of items to be sorted). Obviously, if our algorithm is particularly stupid, there will not be an *upper bound* on the comparisons it uses (it might compare the same two items more than once). Here we are therefore interested in a *lower bound* for the comparisons needed. In other words: How many comparisons do we need *at least* to have all the information needed to sort the collection of items in question?

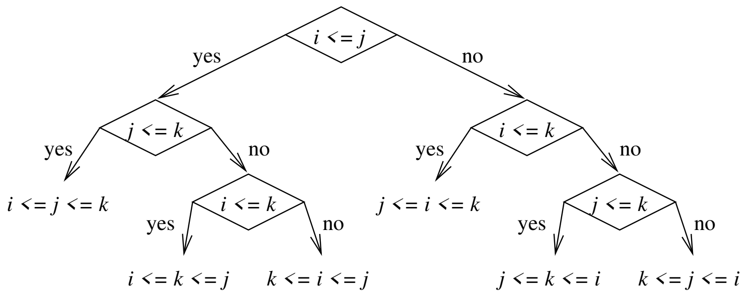
In general, questions of this kind are rather hard, because we have to consider *all* possible algorithms. In fact, for some problems, optimal lower bounds are not known. One example is the so-called *Traveller Salesman Problem* (TSP), for which all known algorithms are extremely inefficient (to the extent of being useless in practice), but for which it is an open problem whether there exists a feasible algorithm.

For sorting based on comparisons, however, it turns out that a tight lower bound exists: the number of comparisons in the worst case is proportional to $n \log n$ where n is the number of items to be sorted. Before giving a proof of this fact, let's examine this from an intuitive point of view.

Firstly, of course "zero steps" is trivially a lower bound: We certainly need to do something. Even if the given collection is sorted, we must check the items one at a time to see whether they are in the correct order. Thus, a better lower bound is n: we need at least n steps, because every element has to be examined. If there were a sorting algorithm that works in n steps, we would be done: n would be both a lower bound and an upper bound to the number of steps, and hence an *exact bound*. However, as we shall see later, no known algorithm takes fewer than $n \log n$ steps in the worst case. In principle, the possibility that a better algorithm is eventually found is open. But, if we show that, no matter what algorithm we choose, we need at least $n \log n$ steps, then we will have obtained an exact bound. This is what we do now.

To begin with, assume we have three items, i, j, and k. If we know that $i \leq j$ and that $j \leq k$ then we know that the sorted order is: i, j, k. So it took us two comparisons to find this out. In some cases, however, we will need as many as three: If the first two comparisons tell us that $i > j$ and that $j \leq k$ then we know that j is the smallest of the two, but we cannot say from this data how i and k relate. So what is the *average* number of comparisons needed then? This can best be decided with a so-called *decision tree*, where we keep track of the information gathered so far and count the number of comparisons needed. Here's the one for the example we were discussing:

How about the general case? The decision tree is obviously a binary tree. What can we say about its size? It is clear that its *height* will tell us how many comparisons will be needed at most, and that the average length of a path from the root to a leaf will give us the average number of comparisons required. The leaves of the decision tree are the possible *outcomes*. These are given by the different possible orders we can have on n items, so we are asking how many ways there are of arranging n items. The first item can be any of n items, the second any of the remaining $n - 1$ items, and so forth, so their total number is



$n! = n(n-1)(n-2)\cdots 3\cdot 2\cdot 1$. So we want to know the height h of a binary tree that can accommodate as many as $n!$ leaves. The number of leaves of a tree of height h is at most 2^h, so we want to find h such that (all the logarithms are supposed to be to base 2, as always for this course)

$$2^h \geq n! \quad \text{or} \quad h \geq \log(n!)$$

So we want to determine $\log(n!)$, or rather, we at least want to get a better idea of what to expect for h, the number of comparisons needed. If n is even, say $n = 2k$, then

$$n! \geq (2k)(2k-1)\cdots(k+1)k \geq k^{k+1} > k^k \geq \left(\frac{1}{2}n\right)^{\frac{1}{2}n}.$$

If, on the other hand, n is odd, so $n = 2k+1$ then

$$n! \geq (2k+1)(2k)(2k-1)\cdots(k+1) > (k+1/2)^{k+1} > (k+1/2)^{k+1/2} = \left(\frac{1}{2}n\right)^{\frac{1}{2}n}.$$

Hence in either case we get $n! > ((1/2)n)^{(1/2)n}$, and therefore $\log(n!) > (1/2)n\log((1/2)n)$. If $n \geq 4$ we have $(1/2)\log n \geq 1$ and so $\log((1/2)n) = \log n - 1 \geq (1/2)\log n$. All in all, for $n \geq 4$ we get

$$h \geq \log(n!) \geq (1/4)n\log n.$$

Hence no sorting algorithm based on comparing items can have a better average or worst case performance than $n\log n$ comparisons. It remains to see whether this can actually achieved. To do this, we have to exhibit at least one algorithm with this performance behaviour (and convince ourselves that it does have this behaviour). In fact, there are several algorithms with this behaviour. This is the topic of the next section.

Notice that we have ignored the factor $1/4$. Why? There are many reasons for that. Firstly, we haven't considered how much time each comparison step takes. If we stipulate that it takes 1 unit of time on a certain machine, then we could have said $1/4n\log n$ instead of $n\log n$. But, if we had a machine 4 times slower, the answer would have to be $n\log n$. Thus, in order to be machine-independent, we don't take factors into account. It has to be emphasized that exact knowledge of the factors is often important in practice. Secondly, by ignoring factors, it is possible to considerably simplify messy calculations. Thus, from the practical point of view of *obtaining* lower and upper bounds, it is easier to ignore constant factors.

4.3 Big-oh notation and complexity classes

There is a notation for expressing the fact that we are ignoring constant factors and other irrelevant details: we write $O(n\log n)$ instead of $n\log n$, and we say that "the complexity class of sorting is big-oh $n\log n$". The big-oh notation is discussed in detail in Chapter 7.

4.4 A few algorithms: Selection Sort, Insertion Sort, Bubblesort

Let us consider particular examples for some of the strategies mentioned above; the strategy implemented is given in brackets after the name of the algorithm. As has been indicated already, the way they work depends on what kind of data structure contains the items that we wish to sort. One option we will consider for our first examples is that we have an array which we wish to sort in the sense explained above.

Selection Sort. Selection Sort first finds the item with the smallest item and puts it into `data[0]` by exchanging it with whichever entry is in that position at the time. Then it finds the entry with the second-smallest entry and exchanges it with the entry in `data[1]`. It continues this way until the array is sorted. More generally, at the ith stage, Selection Sort finds the ith-smallest item and swaps it with the item in `data[i-1]`.

Insertion Sort. Insertion Sort checks the second entry and compares it with the first one. If they're in the wrong order, it swaps the two. That leaves `data[0], data[1]` sorted. Then it takes the third entry and positions it in the right place, leaving `data[0], data[1], data[2]` sorted, and so on. More generally, at the beginning of the ith stage, Insertion Sort has the entries `data[0], ..., data[i-1]` sorted and inserts `data[i]`, giving sorted entries `data[0], ..., data[i]`.

Bubblesort. Bubble sort starts by comparing `data[size]` with `data[size-1]` and swaps them if they're in the wrong order. It then compares `data[size-1]` and `data[size-2]` and swaps those if need be, and so on. This means that once it reaches `data[0]`, the smallest entry will be in the right place. It then starts from the back again, comparing pairs of 'neighbours', but leaving the zeroth entry alone (which is known to be correct). After it has reached the front again, the second-smallest entry will be in place. It keeps making 'passes' over the array until it is sorted. More generally, at the ith stage Bubblesort compares neighbouring entries 'from the back', swapping them as needed. The item with the lowest index that is compared to its right neighbour is `data[i-1]`. After the ith stage, the entries `data[0], ..., data[i-1]` are in their final position.

Example. We now produce a 'test-run' demonstrating the way the algorithms work, using the following array:

4	1	3	2

Selection sort. For the first step, the Selection Sort finds the smallest of `data[1], data[2], data[3]`, which is `data[1]=1`. It swaps this value with that in `data[0]=4` (since it is smaller than the value held there) giving $\boxed{1 \mid 4 \mid 3 \mid 2}$ For the second step, it finds the smallest of `data[2], data[3]`, that is `data[3]=2`. Since this value is smaller than `data[1]` the two are again swapped, giving $\boxed{1 \mid 2 \mid 3 \mid 4}$

Finally, the algorithm checks whether `data[3]` is smaller than `data[2]` which is not the case, so no change occurs. $\boxed{1 \mid 2 \mid 3 \mid 4}$

Insertion sort. Insertion Sort treats `data[0]` as an already sorted array. At the first step, it picks up `data[1]` and 'inserts it' into the already sorted array, increasing the size of same. Since `data[1]` is smaller than `data[0]`, it has to be inserted in the zeroth slot, but that slot is holding a value already. So we first move `data[0]` 'up' one slot into `data[1]` (care must be taken to remember `data[1]` first!), and then we can move the old `data[1]` to `data[0]`, giving $\boxed{1 \mid 4 \mid 3 \mid 2}$ At the next step, the algorithm treats `data[0], data[1]` as an already sorted array and tries to insert `data[2]=3`. This value obviously has to fit between `data[0]=1` and `data[1]=4`. This is achieved by moving `data[1]` 'up' one slot to `data[2]` (the value of which we assume we have remembered), allowing us to move the current value into `data[1]`, giving $\boxed{1 \mid 3 \mid 4 \mid 2}$

Finally, `data[3]=2` has to be inserted into the sorted array `data[0],..., data[2]`. Since `data[2]=4` is bigger than 2, it is moved 'up' one slot, and the same happens for `data[1]=3`. Comparison with `data[0]=1` shows that `data[1]` was the slot we were looking for, giving $\boxed{1 \mid 2 \mid 3 \mid 4}$

Bubblesort. Bubblesort starts by comparing `data[3]=2` with `data[2]=3`. Since they are not in order, it swaps them, giving $\boxed{4 \mid 1 \mid 2 \mid 3}$ It then compares `data[2]=2` with `data[1]=1`. Since those are in order, it leaves them be. Then it compares `data[1]=1` with `data[0]=1`, and those are not in order once again, so they have to be swapped. We get $\boxed{1 \mid 4 \mid 2 \mid 3}$ Note that the smallest entry has reached its final place. This will *always* happen after Bubblesort has done its first 'pass' over the array.

Now that the algorithm has reached the zeroth entry, it starts at the back again, comparing `data[3]=3` with `data[2]=2`. These entries are in order, so nothing happens. (Note that these numbers have been compared before! Also note that in general, we can't avoid repeating comparisons.) Then it compares `data[2]=2` and `data[1]=4`. These are not in order, so they have to be swapped, giving $\boxed{1 \mid 2 \mid 4 \mid 3}$ Since we already know that `data[0]` contains the smallest item, we leave it alone, and the second pass is finished. Note that now the second-smallest entry is in place, too.

Bubblesort now starts the third and final pass, comparing `data[3]=3` and `data[2]=4`. Again these are out of order and have to be swapped, giving $\boxed{1 \mid 2 \mid 3 \mid 4}$ Since it is known that `data[0]` and `data[1]` contain the correct items already, they are not touched. Furthermore, the third-smallest item is in place now, which means that the fourth-smallest *has to be* correct, too.

Implementations of these algorithms.

Selection Sort:

```
for (int i=0; i<size-1; i++) {
  int k=i;

  for (int j=i+1; j<size, j++)
    if (data[j]<data[k])
        k = j;

  swap data[i] and data[k];
}
```

Insertion Sort:

```
for (int i=1; i<size; i++) {
  int k = data[i];
  int j = i-1;

  while ((k<data[j]) & (j>=0)) {
    data[j+1]=data[j];
    j--;
  }

  data[j+1]=k;
}
```

Bubblesort:

```
 for (int i=1; i<size; i++)
   for (int j=size-1; j>=i; j--)
     if (data[j]<data[j-1])
       swap data[j] and data[j-1];
```

Complexity. Let us now consider the number of steps required by the three given algorithms. As is usual for comparison-based sorting algorithms, we will count the number of comparisons that are being made. Assume that the size n of the problem is equal to the variable `size` in the program.

Complexty of selection sort. The outer loop is carried out $n-1$ times. In the inner loop, which is carried out $(n-1)-i = n-1-i$ times, one comparison occurs. Hence the number of comparisons is:

$$\begin{aligned}
&\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 \\
&= \sum_{i=0}^{n-2} n-1-i \\
&= (n-1)+\cdots+2+1 \\
&= \frac{n(n-1)}{2}.
\end{aligned}$$

Therefore the number of comparisons for Selection Sort is proportional to n^2, in the worst as well as in the average case (since these comparisons are carried out no matter how the input data is structured).

Complexity of insertion sort. The outer loop is carried out $n-1$ times. How many times the inner loop is carried out now depends on circumstances: In the worst case, it will be carried out i times; on average, it will be half that often. Hence the number of comparison in the worst case is

$$\begin{aligned}
&\sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 \\
&= \sum_{i=1}^{n-1} i \\
&= 1+2+\cdots+(n-1) \\
&= \frac{n(n-1)}{2};
\end{aligned}$$

that in the average case is half that, that is $n(n-1)/4$. The average and worst case number of steps for of Insertion Sort are both proportional to n^2.

Complexity of bubblesort. The outer loop is carried out $n-1$ times. The inner loop is carried out $(n-1)-(i-1) = n-i$ times. Again the number of comparisons is the same in each case, namely

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

Therefore the worst as well as average number of comparisons is also proportional to n^2

Discussion. So is it the case then that it doesn't make any difference which of these algorithms is used? Here's a table measuring running times of the three, applied to arrays of integers of the size given in the top row. Here O1024 denotes an array with 1024 entries which is sorted already, whereas R1024 is an array which is sorted in the *reverse* order, that is, from biggest to smallest. All other arrays were determined at random. (Warning: You will find tables which differ from this one if you choose a different book. This serves to show that such measurements are *always* dependent on the machine and the implementation of the language involved.)

	128	256	512	1024	O1024	R1024	2048
Bubblesort	54	221	881	3621	1285	5627	14497
Insertion Sort	15	69	276	1137	6	2200	4536
Selection Sort	12	45	164	634	643	833	2497

So where do these differences come from? Note that Selection Sort, while always making $(1/2)n(n-1)$ comparisons, carries out *at most* $n-1$ swaps. Each swap requires three assignments and takes, in fact, more time than a comparison. Bubblesort, on the other hand, does a lot of swaps. Insertion sort does particularly well on data which is sorted already—in such a case, it only makes $n-1$ comparisons. You should bear this in mind for applications—if only a few entries are out of place, Insertion Sort can be very quick. This should serve to show that complexity considerations can be rather delicate, and require good judgement.

Finally, the above numbers give you some idea why under program designers, the general rule is *never* to use Bubblesort. It is easy to program, but that is about all it has going for it. You are better off by avoiding it altogether.

4.5 A different idea—Heapsort

Let us consider another way of of implementing a selection sorting algorithm. The underlying idea is that it would help if we could pre-arrange the data such that selecting the smallest/biggest entry becomes easier. For that, remember the idea of a *priority queue* discussed earlier. We can take the item of an item to give it a priority. Then if we remove the item with the highest priority at each step we can fill an array 'from the rear', starting with the biggest item.

Now priority queues can be implemented in different ways and we discussed an implementation using heap-trees. Another way of implementing them would be using a sorted array, so that the entry with the highest priority appears in `data[size]`. Removing this item would be very simple, but inserting a new one would always involve shifting a number of items to the right to make room for it:

n	0	1	2	3	4	5
`data[n]`	1	2	4			
n	0	1	2	3	4	5
`data[n]`	1	2		4		
n	0	1	2	3	4	5
`data[n]`	1	2	3	4		

A third way would be to use an unsorted array: A new item would be inserted by just putting it into `data[size+1]`, but to delete the entry with the highest priority one would have to find it first. After that, the items with a higher index would have to be 'shifted down'.

Of those three representations, only one is of use in carrying out the above idea: An unsorted array is what we started from, so that isn't any help, and ordering the array is what we are trying to achieve.

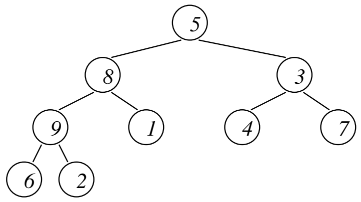
To make use of heap-trees, we first of all have to think of a way of taking an unsorted array and re-arranging it in such a way that it becomes a heap-tree. One possibility would be to insert the items one by one, using the `insert` algorithm discussed earlier. It turns out, however, that this can be done more efficiently. First of all note that if we have n items in the array `data` in positions $1, \ldots$, n, then all the items with an index greater than $n/2$ will be leaves. Therefore if we 'trickle down' all the items `data[n/2], ..., data[1]` by exchanging them with the larger of their children until they either are positioned at a leaf, or until their children are both smaller, we obtain a heap-tree.

5	8	3	9	1	4	7	6	2

We know that the last 5 entries (those are the indices greater than $9/2 = 4$) are leaves of the tree (see the picture).

So the algorithm starts by trickling down 9, which turns out not to be nececssary, so the array remains the same. Next 3 is trickled down, giving:

5	8	7	9	1	4	3	6	2



Next 8 is trickled down, giving:

5	9	7	8	1	4	3	6	2

Finally, 5 is trickled down to give first

9	5	7	8	1	4	3	6	2

then

9	8	7	5	1	4	3	6	2

and finally

9	8	7	6	1	4	3	5	2

The time complexity of this algorithm is as follows: it trickles down $\lfloor n/2 \rfloor$ items, those with indices 1, $\ldots$, $\lfloor n/2 \rfloor$. Each of those trickle operations involve two comparisons at each stage. Now an item with index i will will be on level $\log i$, which means that there are $\log n - \log i = \log(n/i)$ steps until a leaf is reached, so that the trickle process for the item at position i may stop. Hence the total number of comparisons carried out to trickle `data[i]` into position is at most $2\log(n/i)$. So the number of comparisons involved at most is

$$2\Big(\log(n/(n/2)) + \log(n/((n/2)-1)) + \cdots + \log(n/1)\Big) = 2(n/2)\log n - 2\log((n/2)!).$$

This can be shown to be smaller than $2.5n$.

Now that we have a heap-tree, we want to get a sorted array out of it. In the heap-tree, the item with the highest priority, that is the item with the largest item, in `data[1]`. In a sorted array, it should be in position `data[size]`. We then swap the two—which is almost the same as removing the root of the heap-tree, since `data[size]` is precisely the item that would be moved into the root position at the next step. Since now `data[size]` contains the correct item, we will never have to look at it again. Instead, we take the items `data[1], ..., data[size-1]` and rearrange them into a heap-tree with the trickle procedure, which we know to have complexity $O(\log n)$.

Now the second largest item is in position `data[1]`, and we know its final position will be `data[size-1]`, so we now swap these two items. Then we rearrange `data[1], ..., data[size-2]` back into a heap-tree using the trickle procedure.

When the ith step has been completed, the items `data[n-i+1], ..., data[n]` will have the correct entries, and there will be a heap-tree in the items `data[1], ..., data[n-i]`. (Note that the size, and therefore the height, of the heap-tree decreases.) As a part of the ith step, we will have to trickle the new root down. This will take at most twice as many comparisons as the heap-tree is high at the time, which is the logarithm (to base 2) of the number of items in the heap-tree at the time, that is $n - i$.

Hence the complexity function for this phase of the algorithm will be at most

$$2\Big(\log(n-1) + \log(n-2) + \cdots + \log 2 + \log 1\Big) = 2\log((n-1)!).$$

This function can be shown to be smaller than $2n\log n$.

So the worst-case complexity of the entire sorting algorithm, that is *first* rearranging the (unsorted) array into a heap-tree (which is proportional to n) and *secondly* making a sorted array out of the heap-tree (which is proportional to $n\log n$) is given by the *sum* of the two complexity functions. Since the term $n\log n$ grows faster than n, we can simplify $n + n\log n$ to $n\log n$.

4.6 Divide and conquer algorithms

These algorithms all are recursive, the idea being that it is easier to sort a smaller collection of items, so we repeatedly split up the given collection. The 'base case' is the case of a one-element collection, which is trivially sorted. If our collection is bigger than that then we want to split it into two smaller parts.

There are two main ideas for achieving this: Assuming we are working on an array `data` with entries `data[0], ..., data[size-1]` then we can split the set of indices. That is, we consider the two arrays `data[0], ..., data[(size-1)/2]` and `data[(size -1)/2 + 1], ..., data[size-1]`. This method has the advantage that the splitting of the collection into two collections of equal (or nearly equal) size is easy. However, the two sorted arrays that result from this split have to be *merged* together carefully to maintain the ordering. On the other hand, we could try to split the array such that we know that all items in the first collection are smaller than the items in the second collection. Then all we have to do to put them together is to take the first sorted array followed by the second sorted array. This is the underlying idea for an algorithm called Quicksort.

4.7 Quicksort

To save space, we don't actually split the array into smaller arrays, instead, we *rearrange* it to reflect the splitting. We say that we *partition* the array. Quicksort is then applied to sub-arrays of this partitioned array. In order for the algorithm to recursively call itself to sort ever smaller parts of the original collection, we have to tell in *which part* is currently under consideration. Therefore Quicksort calls itself giving the lowest and highest index of the sub-array under consideration. The initial call would then be `Quicksort(data,0,size-1)`.

```
quicksort(array a, int left, int right) {
  if (left < right) {
      int mid = partition(a,left,right);  // see below
      quicksort(a,left,mid-1);
      quicksort(a,mid+1,right);
  }
}
```

The partitioning. The question now is how we can actually perform this kind of split. Ideally, we would like to get two lists of equal size (namely that of half of the collection given) since that is the most efficient way of doing this. (Imagine if we're unfortunate and split off only one item at a time. Then we get n recursive calls where n is the number of entries. If we halve the list at each stage, on the other hand, then we only need $\log n$ recursive calls. If this reminds you of searching in a binary search tree, you are quite right. We can draw a tree whose nodes are labelled by the sub-collections that have been split off, which demonstrates this.)

If our array is very simple, for example [4,3,7,8,1,6], then a good split would be to put all the items smaller than 5 into one part, giving [4,3,1] and all items bigger than or equal to 5 into another, that is [7,8,6]. Indeed, moving all items with a smaller key than some given value into one sub-collection, and all entries with a bigger or equal key to that value into another sub-collection is the standard strategy. This given value is called the *pivot*.

Choosing the pivot. If we get the pivot 'just right' (choosing 5 in the above example, or even 6), then the split will be even. However, there is no guaranteed and quick way of finding the optimal pivot. If the keys are integers, one can take the arithmetic mean of all the keys (that is add them all up and divide by the number of items). However, that requires visiting *all* the entries first to sample their key, and adds considerable overhead to the algorithm (imagine thousands of items…). More importantly, if the keys are, say, strings, you can't do this at all.

Common strategies therefore are:

- Use a random number generator to produce an index k and then use `a[k]`. (If the collection is not given as an array, find another way of randomly choosing an entry.)

- Take a key from 'the middle' of the array, that is `a[(size-1)/2]`. (If the collection isn't given as an array, ignore this suggestion.)

- Take a small sample (usually three items) and take the 'middle' key of those.

You shouldn't choose the first key in the array as the pivot, because if your array is almost sorted, this is going to be a bad choice, and this situation is fairly common.

There is another thing we absolutely have to avoid, namely that of splitting the collection into an *empty* collection and the whole collection again—if we do this then the algorithm won't even terminate, and not just perform badly. So we have to make sure that *at least* one key in the collection is smaller than the pivot. Since we have to ensure this at all cost (an algorithm that doesn't terminate at all obviously fails disastrously), it is worth going through the whole array if need be. What is usually done is to check the entries in the array. If one is found that is smaller than the pivot, one can proceed. Otherwise, if one is found that is bigger than the pivot, make that the new pivot. (Then at least the item(s) which gave rise to the old pivot will be in the 'smaller' sub-collection.) One can make another strategy out of this:

- Check the keys in the array until you find two that differ. Take the larger as the pivot.

We will not give an implementation for any of these strategies and just assume that there is a `choosepivot` algorithm. This algorithm will return the index of the pivot, not the pivot itself.

The partitioning. In order to carry out the partitioning within the given array, some thought is required as to how this may be achieved. This is more easily demonstrated by an example than put into words. For a change, we will consider an array of strings: [c, fortran, java, ada, pascal, basic, haskell, miranda]. The ordering we choose is the lexicographic one. Let the pivot be "fortran".

We will once again use | to denote a partition of the array. To the left of the left marker, there will be items we know to have a smaller key than the pivot. To the right of the right marker, there will be items we know to have a key bigger than or equal to the pivot. In the middle, there will be items we have not yet considered. (Note that therefore this algorithm proceeds to investigate the items in the array *from two sides*.

We start with the array looking like this: [|c, fortran, java, ada, pascal, basic, haskell, miranda|]. Starting from the left, we find that "c" is less than "fortran", so we move the left pointer one step to the right to give [c | fortran, java, ada, pascal, basic, haskell, miranda|]. Now "fortran" is (greater than or) equal to "fortran", so we stop on the left and proceed from the right instead. We cannot move the left marker.

Proceeding from the right now, we find "miranda" to be than "fortran", so we move the right marker to the left by one, and and again for "haskell", giving[c | fortran, java, ada, pascal, basic | haskell, miranda]. Now we have found two keys, "fortran" and "basic", which are 'on the wrong side'. We therefore swap them, which allows us to move both the left and the right marker further towards the middle. This brings us to [c, basic | java, ada, pascal | fortran, haskell, miranda].

Now we proceed from the left once again, but "java" is bigger than "fortran", so we stop there and switch to the right. Now "pascal" is bigger than "fortran", so we move the right marker again. We find "ada", which is smaller than the pivot. We've now got [c, basic | java, ada | pascal, fortran, haskell, miranda]. As before, we want to swap "java" and "ada", which leaves the left and the right markers in the same place: [c, basic, ada || java, pascal, fortran, haskell, miranda]. Therefore we are done.

Here is the `partition` algorithm. Since we cannot have a marker point 'between' array entries, we will assume the left marker is on the left of `a[left]` and the right marker is to the right of `a[right]`. The markers are therefore 'in the same place' once `right` becomes smaller than `left`, which is when we stop. Again we assume that keys are integers (we need to have a class for the pivot, or we could keep this implementation more general).

```
int partition(array a, int left, int right) {

  int m = choosepivot(a, left, right); // get index of pivot
```

```
  int p = a[m]; // pivot itself

  swap a[m] and a[right]; // it is convenient to have the pivot
                          // in the last position of the subarray
                          //   (so that excluding the pivot we
                          //    still have a contiguous subarray)

  // now scan from both ends

  int l = left;
  int r = right - 1; // but leave the pivot alone

  while (l <= r) {

    // now find an element larger than the pivot
    while (l <= r  and  a[l] <= p)
          l = l + 1;

    // now find an element smaller than the pivot
    while (l <= r  and  a[r] >= p)
          r = r - 1;

    // we now check which condition caused the while loops to finish
    if (l < r)
      swap a[l] and a[r];
  }

  // NB. We now must have l>r

  swap a[l] and a[right]; // we put the pivot where it should be

  return l; // the position where the pivot is now


  // What we achieved with the partitioning is:
  //  (0) the elements of the subarray are the same, but in a different order,
  //  (1) for every i in the range left to l-1,   we have a[i] <= a[l],
  //  (2) for every i in the range l+1  to right, we have a[l] <= a[i].
}
```

The complexity of Quicksort. Once again we shall make our formal complexity considerations based on the number of comparisons the algorithm performs. The partitioning step compares each item against the pivot (technically, we could save one comparison if the pivot was one of the entries, and if we remembered which one), and therefore has complexity function $f(n) = n$ (or $f(n) = n - 1$, but that doesn't make a big difference).

In the worst case, whenever we partition a (sub)array we end up with two arrays one of which has size 1. If this happens at each step then we apply the partitioning method to arrays of size n, then $n - 1$, then $n - 2$, until we reach 1. Those complexity functions then add up to

$$n + (n - 1) + \cdots 2 + 1 = n(n + 1)/2 = n^2/2 + n/2.$$

Ignoring the factor and the term $n/2$, this shows that, in the worst case, the number of comparisons performed by Quicksort is approximately to n^2. In the average case, however, Quicksort does much better (unlike some of the algorithms considered above). In the *best* case, whenever we partition the array, the resulting list will differ in size by at most one. Then we have n comparisons in the first case, then twice $\lfloor n/2 \rfloor$ comparisons for the two sub-arrays, then four times $\lfloor n/4 \rfloor$, eight times $\lfloor n/8 \rfloor$, that is

$$n + 2\lfloor n/2 \rfloor + \cdots + \lfloor n/2 \rfloor 2 = n \log n.$$

More interesting is, of course, how Quicksort does in the *average case*. However, that is also much harder to analyse. The strategy in choosing a pivot goes into that (though as long as it is chosen reasonably, that is bearing in mind the problems outlined above, that doesn't change the complexity class), and it does make a difference whether all keys are different, or whether there can be duplicates. In the end in *all* these cases, the number of comparisons in the average case turns out to be proportional to $n \log n$.

Improving Quicksort. It is always worthwhile to spend some time on the strategy for defining the pivot, since the particular problem in question might well allow for a more refined approach. Generally, the pivot will be better the more keys are being sampled before it is being chosen. For example, one could check seven keys and take the 'middle' one of those (the so called *median*). Note that in order to find the true median we actually have to make n^2 comparisons, so we cannot do that without making Quicksort unattractively slow.

Quicksort is not a very suitable algorithm if the problem size is small. The reason for this is that the overhead from the recursion (the return address and formal parameters all need to be stored). Hence once the problem gets 'small' (a size of 16 is found in the literature), Quicksort could stop calling itself and instead use, for example, Selection Sort to sort any lists of this or smaller size.

4.8 Mergesort

If we want to apply the other strategy for splitting a collection of items in half mentioned above, we will not obtain sorted collections that we can just append to each other. Hence mergesort needs an algorithm which merges two sorted collection into another sorted collection. Here is an algorithm, although we assume for the sake of comparability with the other algorithms that we wish to apply mergesort to an array once again. Again the integer variables `m` and `n` refer to the lower and upper index of the sub-array to be sorted.

```
void mergesort(array a, int left, int right) {
if (left < right) {
    int mid = (left + right) / 2;     // integer part of division
    mergesort(a, left, mid);
    mergesort(a, mid + 1, right);
    merge(a, left, mid, right);       // see below
  }
}
```

The `merge` algorithm. Note that it would be simple to write a mergesort algorithm that operates on linked lists (of a known length) rather than arrays. To 'split' two lists in half, all that has to be done is to create two new lists by setting the pointer of the $\lfloor n/2 \rfloor$th list entry to null, and use the $\lfloor n/2 \rfloor + 1$th entry to head off the new list. (Care should be taken to keep the size information intact, however.)

The principle of merging two sorted collections (whether they be lists, arrays, or something else) is quite simple: The smallest key over all will either be the smallest key in the first collection or the smallest key in the second collection. Let's assume it's the smallest key in the first collection. Now the second smallest key over all will either be the second-smallest key in the first collection, or the smallest key in the second collection, and so forth. In other words we work our way through both collections and at each stage, the 'next' key is the current key in either the first or the second collection.

The implementation will be quite different, however, depending on which data structure we wish to use. In the case where the structure is given by an array, it is actually necessary for the `merge` algorithm to create a new array to hold the result of the operation at least temporarily.

In contrast, when using linked lists, it would be possibly for `merge` to work by just changing the reference to the next node. This does make for somewhat more confusing code, however.

```
void merge(array a, int left, int mid, int right) {

  create new array b of size right-left+1;

      // We will merge the elements of the given subarrays
      // storing the result in the array b.
      // After we finish we'll copy this back to the array a.

  int l=0;
  int i=left;
  int j=mid+1;

  while ((i <= mid) and (j<=right)) {
    if (a[i] <= a[j]) {
      b[l]=a[i];
      i++;
      l++;
    }
    else {
      b[l]=a[j];
      j++;
      l++;
    }
  }

  // One sub-array may have a leftover.
  // Check which and copy the remainder to b.

  if (i > mid) {
    for (i=j; i<=right; i++) {
      b[l]=a[i];
      l++;
    }
  }
  else {
  if (j > right) {
      for (j=i; j<=mid; j++) {
        b[l]=a[j];
        l++;
      }
    }
  }

  //Transfer from b back to a.
  for (l=left; l<=right; l++)
      a[l]=b[l-left];
}
```

The complexity of mergesort. The number of comparisons needed by mergesort is proportional to $n \log n$. This holds for the worst as well as the average case. Like Quicksort, mergesort can be speeded up if the recursive algorithm is abandoned once the size of the collections considered becomes small. For arrays, 16 would once again be a suitable size to switch to an algorithm like Selection Sort.

4.9 Treesort

Another possibility for a comparison based sorting algorithm is to insert the items into an initially empty binary search tree. When all trees have been inserted, we know that an infix order traversal of the binary search tree will visit the records in the right order. This algorithm is called Treesort, and it implements the insertion sorting strategy. For Treesort we require that all keys be different.

Treesort has complexity class $n \log n$. When we try to find the complexity function based on the number of comparisons, we note that we are repeatedly inserting into a growing tree. Recall that for Heapsort, we were rearranging (by trickling down) an always decreasing tree. The complexity of both these functions is dependent on the *height* of the tree at the time. Two cases arise here: When considering the average case, we once again find ourselves having to estimate an expression containing $\log(n!)$. In this case, $n \log n$ again gives the right answer. However, in the worst case, the entries are already sorted, and we obtain a tree of height $n - 1$, in which case the insertion process has complexity n^2.

Treesort is difficult to compare with the other sorting algorithms we have discussed, since it does not return an array as the sorted data structure. It should be chosen if it is desirable to have the items stored in a binary search tree. This is usually the case if items are frequently deleted or inserted, since a binary search tree allows these operations to be implemented efficiently (compare with binary search trees). It should be pointed out, however, that searching can be done for a sorted array in $\log n$ steps, the same complexity class as that for binary search trees: The item in question is compared to the item with index $\lfloor n/2 \rfloor$: If it is equal, we are done. If it is smaller, we next compare it to the item with index $\lfloor n/4 \rfloor$, if larger, we next compare it to the entry with index $\lfloor 3n/4 \rfloor$, and so on.

4.10 Summary of comparison-based sorting algorithms

Here is a summary of the algorithms we have considered:

algorithm	strategy employed	objects manipulated	worst case complexity	average case complexity
Selection Sort	Selection	arrays	$O(n^2)$	$O(n^2)$
Insertion Sort	Insertion	arrays/lists	$O(n^2)$	$O(n^2)$
Bubblesort	Exchange	arrays	$O(n^2)$	$O(n^2)$
Heapsort	Selection	arrays	$O(n \log n)$	$O(n \log n)$
Quicksort	D & C	arrays	$O(n^2)$	$O(n \log n)$
Mergesort	D & C	lists/arrays	$O(n \log n)$	$O(n \log n)$
Treesort	Insertion	lists	$O(n^2)$	$O(n \log n)$

But what does it all mean in practice? Here is a table comparing the performance of those of the above algorithms that operate on arrays. It should again be pointed out that these numbers are only of restricted significance, since they can vary somewhat from machine (and language implementation) to machine. Quicksort2 and mergesort2 are algorithms where the recursive procedure is abandoned in favour of Selection Sort once the size of the array falls to 16 and below.

algorithm/size	128	256	512	1024	O1024	R1024	2048
Selection Sort	12	45	164	634	643	833	2497
Insertion Sort	15	69	276	1137	6	2200	4536
Bubblesort	54	221	881	3621	1285	5627	14497
Heapsort	21	45	103	236	215	249	527
Quicksort	12	27	55	112	1131	1200	230
Quicksort2	6	12	24	57	1115	1191	134
Mergesort	18	36	88	188	166	170	409
Mergesort2	6	22	48	112	94	93	254

What has to be stressed is that there is no 'best sorting algorithm'. There usually is a best sorting algorithm *for particular circumstances* and it is up to the program designer to make sure the appropriate one is picked.

4.11 Bin, Bucket, Radix Sorts

These are all names for the same algorithm. Imagine you are given a number of dates, by day and month, and wish to sort those. One way of doing this would be to create a queue for each day, and placing the dates into the queue of their dates (without sorting them further). Now form one big queue out of these, starting with day 1 up to day 31. In the next step, create a queue for each month, and place the dates into the queue *in the order that they appear in the queue created by the first run*. Again form a big queue out of these. This is now sorted in the intended order.

This may seem surprising at first sight, so let us consider an example.

[25/12, 28/08, 29/05, 01/05, 24/04, 03/01, 04/01, 25/04, 26/12, 26/04, 05/01, 20/04].

We create queues for the days as follows:

01: [01/05]
03: [03/01]
04: [04/01]
05: [05/01]
20: [20/04]
24: [24/04]
25: [25/12, 25/04]
26: [26/12, 26/04]
28: [28/08]
29: [29/05]

(Empty queues are not shown - there's no need to create them before we hit an item that belongs to said queue.) Concatenated, this gives:

[03/01, 04/01, 05/01, 20/04, 24/04, 25/12, 25/04, 26/12, 26/04, 28/08, 29/05].

Now we create queues for the months that are present, getting:

01: [03/01, 04/01, 05/01]
04: [20/04, 24/04, 25/04, 26/04]
05: [01/05, 29/05]
08: [28/08]
12: [25/12, 26/12]

Concatenating all these queues gives the expected

[03/01, 04/01, 05/01, 20/04, 24/04, 25/04, 26/04, 01/05, 29/05, 28/08, 25/12, 26/12].

This is called *two-phase Radix Sorting*, since there clearly are two phases to it. Note that *at no point*, the algorithm actually *compares* any keys at all. So what is going on here?

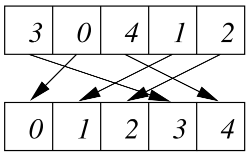
This kind of algorithm makes use of the fact that the keys are *from a strictly restricted set*, or, in other words, of a particular form which is known *a priori*. The complexity class of this algorithm is n, since at every pass, each item is looked at precisely once (the creation of queues does cause some overhead, of course, but this is not worse than linear).

If you know that your keys are all (at most) two-digit integers, you can use Radix Sort to sort them, by creating queues for the last digit first and then doing the same for the first one. Similarly, if you know that your keys are all strings consisting of three characters, you can apply Radix Sort. You would first sort according to the third character, then the second, and finally the first (giving a *three phase* Radix Sort). It is important that you always sort according to the *least significant* criterion first.

4.12 Other methods not based on comparison

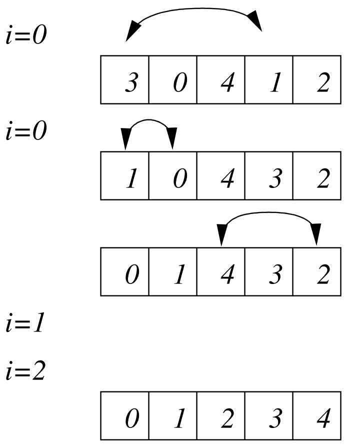
It always pays to think about the data you are trying to sort. Imagine you know the keys are the numbers from 0 to 99. How would you sort those? Take a moment to think about it.

The answer is surprisingly simple. We know that we have 100 entries in the array and we know *exactly which keys should go there and in which order.* This is a very unusual situation as far as general sorting is concerned, yet it may well come up in every-day life. Rather than employing one of the comparison-bases sorting algorithms, in this situation we can do something much simpler as demonstrated by the picture.




```
allocate array ‘result’ of size ‘size’;

for (i=0; i<size; i++)
    result[a[i]] = a[i];

return result;
```

For once we did create a second array to hold the result but, as a matter of fact, we can do without that, although we have to be slightly more inventive for that.




```
for (i=0; i<size; i++) {
  while (a[i] != i) {
    item temp = a[a[i]];
    a[a[i]] = a[i];
    a[i] = temp;
  }
}
```

This looks worse than it is, complexity-wise: the algorithm performs at most $n-1$ swaps where n is the size of the array, since one item, namely `a[a[i]]` is always swapped into its final position. So at worst, it has complexity n.

This should make it clear that in particular situations, sorting might be performed by much simpler (and quicker) means. Once again it is the responsibility of the program designer to take this possibility into account.

Chapter 5

Hash tables

We have already seen a number of ways of storing items on a computer: Arrays, and variants thereof (sorted and unsorted arrays as well as heap-trees, for example), linked lists, and (binary search) trees. We have seen that all these perform quite differently when it comes to particular tasks we expect to carry out, and that *the best way* of achieving this does not exist in general, but differs from situation to situation.

This chapter provides another way of performing this task. It is a method quite different from the ones we have seen so far. Its performance is quite impressive as far as time is concerned. This is payed for, however, by needing more (memory) *space* as well as being quite sophisticated, and therefore not entirely straightforward to describe or implement.

We first fix what we expect to be able to do with this way of storing data without considering how it is actually implemented. In other words, we outline an *abstract data type*. This is similar to what you might do when first trying to implement a class in Java: You will think about the operations you wish to perform on objects of that class. (You may also want to fix a few variables you know you will definitely need for that class, but this does not usually come into defining an abstract data type.) However, for an abstract data type, the way we do this in this course is by describing these operations in plain English, trusting to the idea that they are simple enough not to need further explanations. In any case, what is needed is a *specification* for the abstract data type in question. Once you become familiar with software engineering, you will learn about more formal approaches to this way of operating.

After we have decided what our specification is, we choose a *data structure* in order to *implement* the table abstract data type. The data structure considered in this chapter is known as a *Hash table*.

5.1 The *table* abstract data type

The specification of our *table* abstract data type is this:

1. A table can be used to store objects.
2. The objects can be quite complicated. However, for our purposes, the only relevant detail is that each object has a unique *key*, and that keys can be compared for equality. The keys are used in order to identify objects.
3. We assume that there are methods for
    (a) determining whether the table is empty or full;
    (b) inserting of a new object into the table, provided the table is not full already;
    (c) given a key, retrieving the object with that key;
    (d) given a key, updating the item with that key (usually by replacing the item with a new one with the same key, which is what we will assume here, or by overwriting some of the items variables);
    (e) given a key, deleting the object with that key, provided such object is stored into the table;

(f) listing all the items in the table (if there is an order on the keys then we would expect this to occur in increasing order).

Notice that we are assuming that each object is uniquely identified by its key.

In a language such as Java, we could write an interface for this abstract data type as follows, where we assume here that keys are objects of a class we call `Key`:

```
interface Table
{ Boolean IsEmpty();
  Boolean IsFull();
  void Insert(Record);
  Record Retrieve(Key);
  void Update(Record);
  void Delete{Key};
  void Traverse();
}
```

Note that we have not fixed how exactly the storage of records should work—that is something that goes with the *implementation*. Also note that you could give an interface to somebody else, who could then write a program which performs operations on tables *without ever knowing how they are implemented*. You could certainly carry out all those operations with binary search trees and sorted or unsorted arrays if you wished. The former even has the advantage that a binary search tree never becomes full as such, it is only limited by the size of the memory.

This is also a sensible way to go about defining a Java class: First think about what you want to do with the class, and then wonder about how exactly you might implement the methods. Thus, languages such as Java support mechanisms for defining abstract data types. But notice that, as opposed to a specification in plain English such as the above, a definition of an interface is only a partial specification of an abstract data type, as it doesn't explain *what* the methods are supposed to do; it only explains how they are called.

5.2 Implementations of the table data structure

Implementation via sorted arrays. Let's assume we implement the above via a sorted array. Whether it is full or empty can be determined in constant time if we add a variable for the size. Then to insert an element we first have to find its proper position, which will take on average the same time as finding an element. To find an element (which is necessary for all other operations apart from traversal), we can use binary search as described in the first part of the module, so this takes $O(\log n)$. This is also the complexity for retrieval and update. However, if we wish to delete or insert an item, we will have to shift what is 'to the right' of the location in question by one, either to the left (deletion) or to the right (insertion). This will take on average $n/2$ steps, so these operations have linear complexity. Traversal in order is simple, and of linear complexity as well.

Implementation via binary search trees Another possible implementation would be via binary search trees. However, we know already that in the worst case, the tree will be very deep and narrow so that these will have linear complexity when it comes to looking up an entry. We will see a variant of binary search trees later in the course which keeps the worst case the same as the average case (but which is more complicated to both understand and program), so-called *balanced binary search trees*. For those, search, retrieval, update, insertion and deletion can be done in time $O(\log n)$, and traversal takes $O(n)$.

Implementation via Hash tables This is the topic of this chapter. The idea is that, at the expense of using more space than strictly needed, we speed-up the table operations.

5.3 Hash tables

The underlying idea for hash tables is simple, and quite appealing: Assume that given a key, there was a way of jumping straight to the entry for that key. Then we would never have to search at all, we could just go there!

Of course, we have not said yet how that could be achieved. Assume that we have an array `data` to hold our entries. Now if we had a function h that assigns a key to the index (an integer) where it will be stored, then we could just look up `data[h(k)]` to find the entry with the key `k`.

It would be easier if we could just make our array big enough to hold *all* the keys that might appear. For example, if we knew that our keys were the numbers from 0 to 99 we could just create an array of size 100 and store the entry with key 67 in `data[67]`. In this case the function h would be the identity function; that is, the function defined by $h(k) = k$.

However, this idea is not very practical if we are dealing with a relatively small number of keys out of a huge collection of possible keys. For example, many American companies use their employees' 9-digit social security number as a key (British ones don't work quite as well because they are usually a mixture of characters and numbers). Obviously there are hugely more such numbers than any individual company will have employees, and it would probably be impossible (and not very clever) to reserve space for all the 100,000,000 social security number which might occur.

Instead, we do use a non-trivial function h, the so-called *hash function*, to map the space of possible keys to the set of indices of our array. For example, if we had 500 employees we might create an array with 1000 entries and use three digits from their social security number to determine the place in the array where the records for a particular employee should be stored.

There is an obvious problem with this technique which becomes apparent at once: What if two employees have the same such digits? This is called a *collision* between the two keys. Much of the remainder of this chapter will be spent on the various strategies for dealing with collisions.

First of all, of course, one would try to avoid collisions. If the keys that are likely to actually occur are not evenly spread in the space of all possible keys, particular attention should be spent on choosing the function h in such a way that collisions among those are unlikely to occur. If, for example, the first three digits of a social security number had geographical meaning then employees are particularly likely to have the three digits signifying the region where the company resides, and so choosing the first three digits as a hash function might result in many collisions which could have been avoided by a more prudent choice.

5.4 Likehood of collisions — the load factor of a hash table

One might be tempted to assume that collisions do not occur very often if a small subset of the set of possible keys is chosen, but this assumption is mistaken.

Assume we have a hash table of size m, and that it currently has n entries. Then we call $\lambda = n/m$ the *load factor* of the hash table. The load factor can be seen as describing how full the table currently is: A hash table with load factor 0.25 is 25% full, one with load factor 0.50 is 50% full, and so forth. If we have a hash table with load factor λ then the probability that for the next key we wish to insert a collision occurs is λ. Thus assumes that each key from the key space is equally likely, and that the hash function h spreads the key space evenly over the set of indices of our array. If these optimistic assumptions fail, then the probability can be higher.

Therefore to minimize collision, it is prudent to keep the load factor low, fifty percent being an often quoted figure. We will see later what effect the table's load factor has on the speed of the operations we are interested in.

5.5 An example

Let us assume we have a small array we wish to use, of size 11, and that our set of possible keys is the set of 3-character strings, where each character is in the range from A to Z .

We now have to define a hash function which maps each string to an integer from 0 to 10. For that, we first map each string to a number as follows: We can map each character to an integer from 0 to 25 by giving its place in the alphabet (A is the first letter, so it goes to 0, B the second so it goes to 1, and so on, with Z getting value 25). The string $X_1X_2X_3$ therefore gives us three numbers from 1 to 26, say k_1, k_2, and k_3. We map the whole string to the number calculated as

$$k = k_1 * 26^2 + k_2 * 26^1 + k_3 * 26^0 = k_1 * 26^2 + k_2 * 26 + k_3.$$

(That is, we think of strings as coding numbers in base 26.)

Now it is quite easy to go from any *number* k (rather than a string) to a number from 0 to 11: For example, we can take the remainder the number leaves when divided by 11. This is the Java operation `k % 11`. So our hash function is

$$h(X_1X_2X_3) = (\mathtt{k}_1 * \mathtt{26}^2 + \mathtt{k}_2 * \mathtt{26} + \mathtt{k}_3)\%\mathtt{11} = \mathtt{k}\%\mathtt{11}.$$

Assume we wish to insert the following three-letter airport acronyms as keys into our hash table: PHL, ORY, GCM, HKG, GLA, AKL, FRA, LAX, DCA. To make this easier, we'll just catalogue the values the hash function takes for these:

Code	PHL	ORY	GCM	HKG	GLA	AKL	FRA	LAX	DCA
$h(X_1X_2X_3)$	4	8	6	4	8	7	5	1	1

The number associated with PHL is 4, so we place it at index 4, giving

				PHL						

Next ORY, which gives us the number 8, so that's where we put it:

				PHL				ORY		

Then we get GCM, with value 6, giving

				PHL		GCM		ORY		

Now HKG which also has value 4 causing our first collision since the corresponding position has already been filled with PHL. Now we could, of course, try to deal with this by saying the table is full, but this gives such poor performance (due to the frequency in which collisions occur) that it is unacceptable.

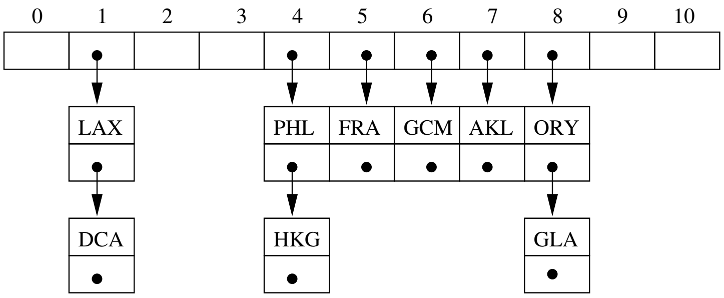
5.6 Strategies for dealing with collisions

First of all, note that we have to be able to tell whether a particular location in the array is still empty, or whether it has been filled yet. We therefore assume that there is a *unique* key (which is *never* associated with a record) which denotes that the position has not been filled yet. This key will not appear in the pictures we use.

Buckets. One option is to, in fact, reserve a two-dimensional array from the start. (Think of a column as a bucket in which we throw all the elements which give a particular result when the hash function is supplied, so the fifth column is that of all the keys for which the hash function evaluates to 4.) Then we could put HKG into the slot 'beneath' PHL, and GLA in the one beneath ORY, and continue filling the table in the order given until we reach

0	1	2	3	4	5	6	7	8	9	10
	LAX			PHL	FRA	GCM	AKL	ORY		
	DCA			HKG				GLA		

This method has to reserve quite a bit more space than will be eventually required (since it must take into account the likely maximal number of collisions). When searching for a particular key, it will be necessary to search the entire column associated with its expected position (at least until we reach an empty slot). If there is an order on keys, we can store them in ascending order, which means that we know a particular key is not present once we reach a key which is larger. The average complexity of searching for a particular item depends on how many entries in the array have been filled already. This method turns out to be slower than the other techniques we shall consider, so we shall not spend more time on it apart from remarking that it has proven useful when the entries are held in slow external storage. Note, however, that while the table is still quite empty over all, collisions have become increasingly likely.



Direct chaining. Rather than reserving entire sub-arrays (the rows above) for keys that collide, one can make a linked list for the entries corresponding to every key. The result can be pictured something like this.

This method does not reserve any space that will not be taken up, but has the disadvantage that in order to find a particular item, lists will have to be traversed. However, adding the hashing step speeds this up considerably. The average non-empty list occurring in the hash table, will have size $1 + ((n - 1)/m)$, so searching for a particular entry will require calculating its hash function and then finding the element in the corresponding list. Finding an item in a list of size k takes on average

$$\frac{1}{k}\Big(1 + 2 + \cdots + k\Big) = \frac{k(k+1)}{2k} = \frac{k+1}{2}$$

comparisons, so for the given average length, we will have

$$\frac{1 + (n-1)/m + 1}{2} = 1 + \frac{n-1}{2m} \approx 1 + \frac{\lambda}{2}$$

comparisons on average in a successful search. The average list (including empty ones), on the other hand, will have length λ, and in an unsuccessful search the number of comparisons made to decide the item in question is not present will therefore be λ. Note that this does not at all depend on the number of keys in the table, but on the load factor, which can be held low by increasing the size of the hash table if it should get too full. Also note that insertion is done even more speedily, since all we have to do is to insert a new element at the front of the appropriate list. Hence, apart from traversal, the complexity class of all operations is *constant*. For the last we need to sort the keys, which can be done in $O(n \log n)$, as we know from Chapter 4. A variant would be to make the lists sorted, which will speed up finding an item as well as traversal slightly, although it won't put these operations into a different complexity class. This is paid for by making insertion more expensive, *ie.* that operation will take slightly longer, but still have constant complexity. This is very impressive compared to our considerations for sorted arrays or (balanced) binary search trees.

Open addressing. The last fundamentally different approach to this is to find another open location for an entry which cannot be placed where its hash function tells us to. We refer to that position as a key's *primary position* (so in the example, ORY, GLA and JFK all have the same primary position). The simplest strategy for achieving this is to search for empty locations by decreasing the index considered by one until we find an empty space. If we reach the beginning of the array, *ie.* the index 0, we start again at the back. This is called *linear probing*. Here's the example from above. We had reached the stage

				PHL		GCM		ORY		

and then wanted to put HKG at index 4, where we found PHL. We therefore now try index 3, and find an empty location, so we put HKG there giving

			HKG	PHL		GCM		ORY		

Next we reach GLA, with value 8. But the location with that index is already filled (by ORY), as is the one to the left. The slot to the left again is free, however, (that is two steps to the left from the primary position) so we insert GLA there to give

			HKG	PHL		GCM	GLA	ORY		

Now we have AKL, and despite the fact that we have not had the value 7 before we find that the corresponding location is filled (by GLA), so we try the next one down which has GCM, but that at index 5 is empty, so that's where we put AKL.

			HKG	PHL	AKL	GCM	GLA	ORY		

We now continue with the remaining keys, eventually reaching

DCA	LAX	FRA	HKG	PHL	AKL	GCM	GLA	ORY		

This looks quite convincing, and seems to make good use of the space we have reserved. However, what happens now if we wish to find a particular key? It will not be good enough any longer to apply the hash function to it and check there: Instead, we will have to follow its possible insertion locations until we hit an empty one, which tells us that the key we were looking for is not present, after all, because it would have been inserted there. This is why every hash table that uses open addressing should have at least one empty slot at any time, and be declared full when only one empty location is left. However, as we shall see hash tables lose much of their speed advantage if they have a high load factor, so as a matter of policy, more locations should be kept empty.

So to find the key AKL, we will first check at index 7, then at 6, and 5, where we are successful. Searching for JFK, on the other hand, we would start with its proper position, given by the value 8, so we would check indices 8, 7, 6, …, 1, 0, 10 in that order until we find an empty space which tells us that JFK is, in fact, not present at all. This looks pretty bad at first sight, but bear in mind that we said that we will aim towards keeping the load factor at around 50 percent, so there would be many more empty slots which effectively stop any further search.

But this idea brings another problem with it. Assume we first delete GCM and the search for AKL again. Then we would find the array empty at index 6 and stop searching, therefore conclude wrongly that AKL is not present. This is not acceptable, but equally we do not wish to have to search through the entire array to be sure that an entry is not there.

The solution is that instead, we reserve another key to mean that a position is empty, but that it did hold a key at some point. Let's assume that for that, we use the character '!'. Then after deleting GCM, the array would be

DCA	LAX	FRA	HKG	PHL	AKL	!	GLA	ORY		

and when searching for AKL we would know to continue beyond the exclamation mark. If, on the other hand, we are trying to *insert* a key then we ignore any exclamation marks and fill the position once again. This now does take care of all our problems, although if we do a lot of deleting and inserting, we will end up with a table which is a bit of a mess. A large number of exclamation marks means that we have to keep looking for a long time to find a particular entry despite of the fact that the load factor may not be all that high. This happens if deletion is a frequent operation. In such a case it may be better to compute the hash table anew, or use another implementation.

The complexity of open addressing with linear probing is rather delicate, and we will not attempt to give an account of it here. If λ is once again the load of the table, then a successful search takes on average $(1/2)(1+(1/(1-\lambda))$ comparisons, while an unsuccessful one takes approximately $(1/2)(1+(1/(1-\lambda))^2$. For relatively small load factors, this is quite impressive (and even for larger ones, it's not bad).

5.7 Open addressing—alternatives to linear probing

Clustering. There is one problem with linear probing as introduced above, namely what is known as *primary* and *secondary clustering*. Note what happens if we try to insert two keys that have the same result when the hash function is applied to them: Take the above hash table at the stage where we just inserted GLA:

			HKG	PHL		GCM	GLA	ORY		

If we next try to insert JFK we note that the hash function evaluates to 8 once again. So we keep checking *the same* locations we only just checked in order to insert GLA. This seems a rather inefficient way of doing this. This effect is known as *primary clustering* because the new key JFK will be inserted close to the previous key with the same primary position, GLA. It means that we get a continuous 'block' of filled slots, and whenever we try to insert any key which is sent into the block by the hash function, we will have to test all locations until we hit the end of the block, and then make such block even bigger by appending another entry at its end. So these blocks, or clusters, keep growing not only if we hit the same primary location repeatedly, but also if we hit anything that's part of the same cluster. The last effect is called *secondary clustering*. Note that searching for keys is also adversely affected by these clustering effects.

Double hashing. The solution is to do something slightly more sophisticated than trying every position until we find an empty one. This is known as *double hashing*. We apply a *second* hash function to tell us how to go about finding an empty slot if a key's primary position has been filled already. In the above example, something we can do is to take the same number associated with the three-character code k, but now we use the result of integer division by 11 (instead of the remainder). However, the resulting value might be bigger than 10, so to make it fit into our table, we *first* take the result of integer division by 11, and *then* take the remainder this result leaves when again divided by 11. We would like to use as our second hash function $h'(n) = (\mathtt{k/11})\%\mathtt{11}$, but this has one problem: it might give 0 at some point, and we cannot test 'every zeroth location'. Hence we make our second hash function one if the above would evaluate to 0, that is

$$h'(n) = \begin{cases} (\mathtt{k/11})\%\mathtt{11} & \text{if } (\mathtt{k/11})\%\mathtt{11} \neq \mathtt{0}, \\ 1 & \text{otherwise.} \end{cases}$$

Its values are given in the table below:

Code	PHL	ORY	GCM	HKG	GLA	AKL	FRA	LAX	DCA
$h'(X_1X_2X_3)$	4	1	1	3	9	2	6	7	2

We then proceed as follows: Let us return to the position we were in when the first collision occurred,

				PHL		GCM		ORY		

with HKG the next key to insert, which gives a collision with PHL. Since $h'(\text{HKG}) = 3$ we now try *every third location* in order to find a free slot. This brings us

	HKG			PHL		GCM		ORY		

Note that this did not create a block. When we now try to insert GLA, we once again find its primary location blocked by ORY. Since $h'(\text{GLA}) = 9$, we now try every ninth location. Counting to the left from ORY, that gets us (starting again from the back when we reach the first slot) to the last location over all

	HKG			PHL		GCM		ORY		GLA

Note that we still have not got any blocks, which is good. Further note that most keys which share the same primary location with ORY and GLA will follow a different route when trying to find an empty slot (ORY every slot rather than every ninth one), thus avoiding primary clustering. Here is the result when filling the table with the remaining keys given:

	HKG	DCA		PHL	FRA	GCM	AKL	ORY	LAX	GLA

Our example is too small to show convincingly that this method also avoids secondary clustering. Another example which demonstrates the difference between linear probing and double hashing is spread over Exercises 1 and 3.

The efficiency of double hashing is as difficult to judge as that of linear probing, and therefore we will just give the result, without an explanation. A successful search requires on average $(1/\lambda)\ln(1/(1-\lambda))$ comparisons, an unsuccessful one $1/(1-\lambda)$. Note that the natural logarithm—that is to base e—occurs here.

5.8 Choosing hash functions

What is important when choosing a hash function is to make sure that it evenly spreads the space of possible keys onto the set of indices for the hash table. Secondly, it is advantageous if clusters in the space of possible keys are broken up (something that the remainder in a division will *not* do), in case we are likely to get a 'continuous run'. Therefore, when defining hash functions of strings of characters, it is never a good idea to make the last (or even the first) few characters decisive.

When choosing secondary hash functions, in order to avoid primary clustering, one has to make sure that different keys with the same primary position give *different* results when the second hash function is applied. Secondly, one has to be careful to ensure that the second hash function cannot result in a number which has a non-trivial common divisor with the size of the hash table: If the hash table has size 10, and we get a second hash function which gives 2 as a result, then only *half* of the locations will be checked, which might result in failure (an endless loop, for example) while the table is still half empty. A simple remedy for this is to always make the size of the table a prime number.

5.9 Complexity considerations for hash tables

This table gives the average number of locations that have to be checked to conduct successful and unsuccessful searches in hash tables with different collision handling strategies, depending on the load factor given in the top column.

	0.10	0.25	0.50	0.75	0.90	0.99
Successful Search						
Direct chaining	1.05	1.12	1.25	1.37	1.45	1.48
Linear probing	1.06	1.17	1.50	2.50	5.50	50.5
Double hashing	1.05	1.15	1.39	1.85	2.56	4.65
Unsuccessful search						
Direct chaining	0.10	0.25	0.50	0.75	0.90	0.99
Linear probing	1.12	1.39	2.50	8.50	50.5	5000
Double hashing	1.11	1.33	2.00	4.00	10.0	100

This shows the considerable advantage double hashing has over linear probing. Whether or not the former is preferable to direct chaining is dependent on the circumstances.

The following table shows a comparison for different implementation of the `table` interface. Hash tables perform rather well. In practice, when deciding what to use, it will have to depend on the mix of operations typically performed. Lots of insertions and in particular deletions don't sit very well with hash tables, as explained above. The complexity of updating and retrieving is the same as that of searching.

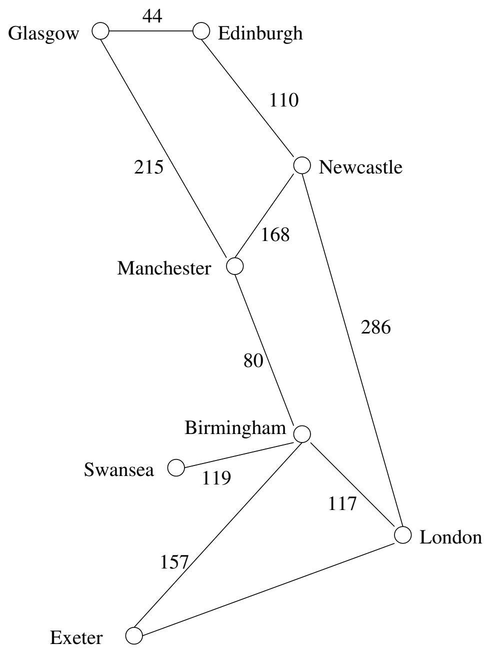
	Search	Insert	Delete	Traverse
Sorted array	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$
Balanced bst	$O(\log n)$	$O(n \log n)$	$O(\log n)$	$O(n)$
Hash table	$O(1)$	$O(1)$	$O(1)$	$O(n \log n)$

Just to give an example, if there are 4096 entries in a balanced binary search tree, it takes on average 12.25 comparisons to complete a successful search. On the other hand, we can get as few as 1.39 comparisons by using a hash table, provided that we keep its load factor below 50 percent.

Chapter 6

Graphs

Many times when information is being represented, we find it useful to do so like this:



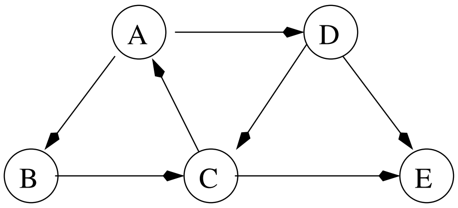
With a similar structure (leaving out the distances, or replacing them by something else), we could represent many other situations, like an underground network, or a network of pipes (where a number might give the diameter), or a railway map, or an indication for which cities are linked by flights, or ferries. Even if we assume it's a network of paths or roads, the numbers needn't give a distance: They might be an indication for how long it takes to cover the distance in question on foot, so up a steep hill an equal distance would take longer than on even ground.

There is more to be done with such a picture of a situation than just reading off which place is directly connected with another place: For example, we can ask ourselves whether there is a way of getting from A to B at all, or which the shortest way is. And then there is the famous 'Travelling Salesman Problem', which asks to find the shortest way through the structure which visits each city precisely once.

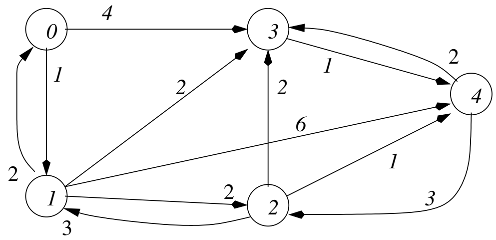
6.1 Graphs

The kind of structure in the above figure is known as a *graph*. Formally, a graph consists of so-called *vertices*, the little dots, or nodes, which are connected by so-called *edges*. If there are labels on the edges (usually non-negative real numbers) we say that the graph is *weighted*.

We distinguish between *directed* and *undirected* graphs. For the former (also called digraphs), each edge comes with a direction (usually indicated by an arrow as in the figures below). Think of them as representing roads, where some roads may be one-way only. Or think of the numbers coming with them as something that applies to travel in one-way only: Going up a hill will take longer than coming down. The following is an example of a digraph:



A graph is called *weighted* if it has labels on its edges, e.g.



In undirected graphs we assume that every edge can be viewed as going both ways, that is, an edge between A and B goes from A to B as well as from B to A. The first graph given at the beggining of this chapter is unweighted and undirected.

While we're about it, let's get some more terminology out of the way. A *path* is a sequence of vertices $v_1, v_2, \ldots, v_n$ such that v_i and v_{i+1} are connected by an edge for $1 \leq i \leq n-1$. (Note that in a directed graph, an edge from v_i to v_{i+1} is one which has the corresponding direction.)

You may have noticed that this is not at all dissimilar to a *tree*. In fact, trees are graphs of a particular kind: A *circle* is a path whose first vertex is the same as its last one. An undirected graph is *connected* if between every two vertices there is a path connecting them. A tree is nothing but a graph which is connected and contains no circles.

For directed graphs, the notion of connectedness becomes two different ones: We say that a digraph is *weakly connected* if for every two vertices A and B there is either a path from A to B or a path from B to A. We say it is *strongly connected*, on the other hand, if there are paths leading both ways. So in a weakly connected digraph there may be two vertices i and j such that there exists no path from i to j. A path is *simple* if no vertex appears on it twice (with the exception of a circle, where the first and last vertex may be the same—this is because we have to 'cut open' the circle at some point to get a path, so this is inevitable).

Because a graph, unlike a tree, does not come with a natural 'starting point' from which there is a unique path to each vertex, it does not make sense to speak of parents and children in a graph. Instead, if two vertices A and B are connected by an edge e, we say that they are *neighbours*, and the edge connecting them is said to be *incident* to A and B. Two edges that have a vertex in common (for example, one connecting A and B and one connecting B and C) are said to be *adjacent*.

6.2 Implementing graphs

Before we start considering the implementation of graphs we should pause for a moment. All the data structures we have considered so far were to hold information, and we wanted to perform certain actions which mostly centred around inserting new items, deleting particular items, searching for particular items, and sorting the collection. At no time was there ever a *connection* between all those items, apart from the order in which their keys appeared. But that latter was never something that was inherent to the structure and that we therefore tried to represent somehow—it was just a property we used to store items in a way which made sorting and searching quicker. (While we created connections in binary search trees, they were important in themselves, and destroyed again when items were added or deleted.) Now, on the other hand, we are *given* a structure which comes with these connections, and it is our job to make sure that the implementation keeps track of those. After all, these *constitute* the information we are trying to encode!

Array-based implementation. The first underlying idea for this approach is that we might as well rename the vertices of the graph so that they are labelled by non-negative integers, say from 0 to $n-1$, if they don't have these labels already. However, this only works if the graph is given *explicitly*, that is if we know in advance how many vertices there will be, and which pairs will have edges between them. Then all we need to keep track of is which vertex has an edge to which other vertex. We can do this quite easily in a $n \times n$ two-dimensional array `adj`, also called a *matrix*, the so-called *adjacency matrix* (or, in the case of weighted graphs, the *weight matrix* `weights`), for the graph. Here are the matrix representations for two graphs given above:

		A	B	C	D	E
		0	1	2	3	4
A	0	0	1	0	1	0
B	1	0	0	1	0	0
C	2	1	0	0	0	1
D	3	0	0	1	0	1
E	4	0	0	0	0	0

	0	1	2	3	4
0	0	1	∞	4	∞
1	2	0	2	2	6
2	∞	3	0	2	1
3	∞	∞	∞	0	1
4	∞	∞	3	2	0

In the first case, a '0' in position `adj[i][j]` reads as `false`, that is, there is no edge from the vertex i to the vertex j. A '1', on the other hand, or `true`, indicates that there is an edge. It is quite useful to use booleans here rather than the numbers 0 and 1 because it allows us to carry out operations on the booleans. We just use the numbers here because they make for a more compact presentation.

In the second case we have a weighted graph, and we have the weights in the matrix instead, using the symbol ∞ to indicate that there is no edge.

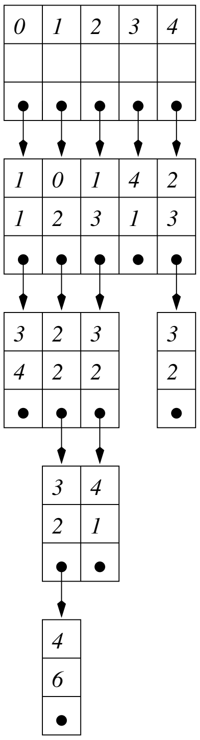
For an undirected graph, if there is a 1 in the ith column and the jth row, we know that there is an edge from vertex i to the vertex with the number j, which means there is also an edge from vertex j to vertex i. This means that `adj[i][j] == adj[j][i]` will hold for all i and j from 0 to $n-1$, so there is some redundant information here. We say that such matrices are *symmetric*, since the matrix is equal to its own mirror-image when reflected on the diagonal.

Mixed implementation. There is one problem with the adjacency matrix representation: If the graph has very many vertices, this becomes quite a big array (ten thousand entries are needed if we have just 100 vertices), and if there are very few edges (in which case the adjacency matrix contains many 0s and only a few 1s, and we say it is *sparse*), it seems a waste of space to reserve so much for so little information.

A solution for this case is to still number all the vertices as before, but rather than using a two-dimensional array, we use a one-dimensional array to point at a linked list of neighbours of a particular vertex. For example, one of the above graphs can be represented as follows:

Pointer-based implementations. This generalizes the pointer-based implementation for binary trees (which in turn generalizes linked lists). In a language such as Java, a class `Graph` might have the following as an internal class:

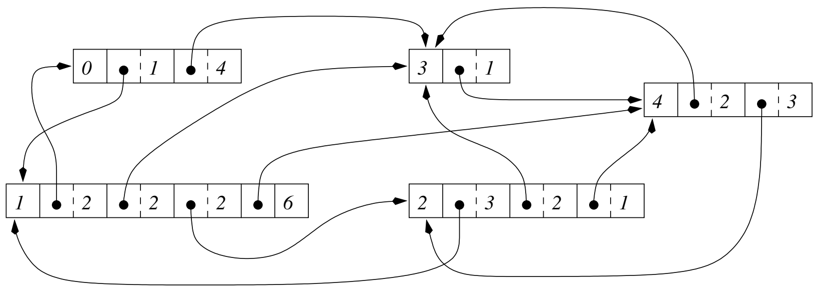
```
class Vertex
{ string name;
  Vertex[] neighbours;
}
```



If there are very few edges, we will have very short lists at each entry of the array, thus saving space over the adjacency matrix representation. This implementation is one using so-called *adjacency lists*. Note that if we are considering undirected graphs then there is still a certain amount of redundancy in this representation, since every edge is represented twice, once in each list corresponding to the two vertices it connects. In Java, this could be accomplished as follows:

```
class Graph
{ Vertex[] heads;
  private class Vertex
  { int name;
    double weight;
    Vertex next;
  ...//methods for vertices
  }
 ...//methods for graphs
}
```

When we create a vertex, we allocate an array `neighbours` big enough to accomodate (pointers to) its the neighbours. We place the (pointer to the) neighbours of the vertex into that array in some arbitrary order. Any non-needed entries in the array will hold a `null` pointer as usual. For example, one of the above graphs would look as:



6.3 Traversals—visiting all vertices in a systematic fashion

In order to solve this problem, we need a strategy for exploring graphs which guarantees that we don't miss an edge or vertex. Because, unlike trees, graphs don't have a root vertex, there is no natural place to start a traversal, and therefore we assume that we are given a starting vertex i. There are two strategies for performing this task.

The first is known as *breadth first traversal*: We start with the given vertex i. Then we visit its neighbours one by one (which must be possible no matter which implementation we use), placing them in an initially empty queue. We then remove the first vertex from the queue and one by one put its neighbours at the end of the queue. We then visit the next vertex in the queue and again put its neighbours at the end of the queue. We do this until the queue is empty.

However, there is no reason why this algorithm should ever terminate. If there is a circle in the graph, like A, B, C in in one of the above graphs, we would enqueue a vertex *we have already visited*, and thus we would run into an infinite loop (visiting A's neighbours puts B onto the queue, visiting that (eventually) gives us C, and once we reach C in the queue, we get A again). To avoid this we create a second array `done` of booleans, where `done[j]` is `true` if we have already visited the vertex with number j, and it is `false` otherwise. In the above algorithm, we only enqueue a vertex j if `done[j]` is `false`. Then we mark it as done by setting `done[j] = true`. This way, we won't enqueue any vertex more than once, and for a finite graph, our algorithm is bound to terminate. In the example we are discussing, breadth first search starting at A might yield: A, B, D, C, E.

To see why this is called breadth first search, imagine a binary tree being implemented in this way, where the starting vertex is the root. We would then first follow all the edges emanating from the root, leading to all the vertices on level 1, then find all the vertices on the level below, and so on, until we find all the vertices on the 'lowest' level.

The second is known as *depth-first traversal*: Given a vertex i to start from we now put it on a stack rather than a queue (recall that in a stack, the only item that can be removed at any time is the last one to be put on the stack), and mark it as done as for breadth first traversal. We then look up its neighbours one after the other, mark them as done and put them onto the stack. We then pop the next vertex from the stack and visit its neighbours in turn—provided they have not been marked as done, just as as we did for breadth first traversal. For the example discussed above, we might get (again starting from A): A, B, C, E, D.

Note that in both cases, the order of vertices depends on the implementation. There's no reason why A's neighbour B should be visited before D in the example. So it is better to speak of *a* result of depth-first or breadth-first traversal. Also note that the only vertices that will be listed are those in the same *connected component* as A. If we have to ensure that all vertices are visited we may need to start the process over with a different starting vertex: choose one that has not been marked as done when the algorithm terminates.

Exercises. Write algorithms, in pseudocode, to (1) visit *all* nodes of a graph, and (2) decide whether a given graph is connected or not. In (2), consider in fact two algorithms, one for the strong and the other for the weak notion of connectedness.

6.4 Shortest path between two vertices.

Suppose we are given a weighted digraph with vertices labelled by non-negative numbers. We want to answer the following question: Given two vertices, what is the shortest route from one to the other?

Here by "shortest" we mean a path which, when we add up the weights along its edges, gives the smallest weight for the path overall. This number is called the *length* of the path. Thus, a shortest path is one with minimal length. Note that there need not be a unique shortest path, since several paths might have the same length. In a disconnected graph there will not be a path between vertices in different *components*, but we can take care of this by using ∞ once again to stand for "no path at all".

Notice that weights don't need to be distances; they can, for example, be time (in which case we could speak of "quickest paths") or money (in which case we could speak of "cheapest paths"), among other possibilities. By considering "abstract" graphs in which the numerical weights are left uninterpreted, we can take care of all such situations and others. But notice that we do need to restrict to non-negative numbers, because if there are negative numbers and cycles, we can have paths with smaller and smaller lengths, but no path with minimal length.

Applications of shortest-path algorithms include train-ticket reservations, driving directions, internet packet routing (if you send an email message from your computer to someone else, it has to go through various computer routers, until it reaches it final destination).

Dijkstra's algorithm. In turns out that if we want to compute the shortest path from a given start node s to a given end node z, it is actually convenient to compute the shortest paths from s to all other nodes, not just the given node z that we are interested in. Given the start node, Dijkstra's algorithm computes shortest paths starting from s and ending at each possible node. Because the algorithm, although elegant and short, is fairly complicated, we attack it in small steps.

Overestimation of shortest paths. We keep an array D indexed by vertices. The idea is that $D[z]$ will hold the distance of the shortest path from s to z *when the algorithm finishes*. However, *before the algorithm finishes*, $D[z]$ is an overestimate of the distance from s to z. We initially set $D[s] = 0$ and $D[z] = \infty$ for all vertices z other than the start node s. What the algorithm does is to decrease this overestimate until it is no longer possible to do so. When this happens, the algorithm terminates, and the estimates are now tight.

Improving estimates. Of course, the idea is to find shortcuts. Suppose that, for two given vertices u and z, it happens that $D[u] + \text{weight}[u][z] < D[z]$. Then there is a way of going from s to u and then to z whose total length is smaller than the overestimate $D[z]$ of the distance from s to z, and hence we can replace $D[z]$ by this better estimate. This corresponds to the fragment

```
if (D[u] + weight[u][z] < D[z])
   D[z] = D[u] + weight[u][z]
```

of the full algorithm given below.

The problem now is to systematically apply this improvement so that (1) we eventually get the tight estimates promised above and (2) this is done efficiently.

Dijkstra's algorithm, version I. The first version of the algorithm is not as efficient as it can be, but it is correct and relatively simple. The idea is that, at each stage of the algorithm, if an entry $D[u]$ of the array D has minimal value among all values recorded in D, then the overestimate $D[u]$ is actually tight, because the improvement algorithm discussed above won't succeed to find a shortcut. But this is not the whole story — see below.

```
// Input:  A directed graph with weight matrix 'weight' and
//         a start vertex 's'.

// Output: An array 'D' as explained above.

// We begin by buiding the overestimates.

D[s] = 0;    // Surely the shortest path from s to itself has length zero.

for (each vertex z of the graph)
    if (z isn't the start vertex s)
        D[z] = infinity; // This is certainly an overestimate.

// We use an auxiliary array 'tight' indexed by vertices,
// which records for which nodes the shortest paths
// estimates are ``known'' to be tight by the algorithm.

for (each vertex z of the graph)
    tight[z] = false;


// We now update the arrays 'D' and 'tight' until
// all entries in the array 'tight' hold the value true.

repeat as many times as there are vertices in the graph {
   find a vertex u with tight[u] false and with minimal estimate D[u];

   tight[u] = true;

   for (each vertex z adjacent to u)
       if (D[u] + weight[u][z] < D[z])
```

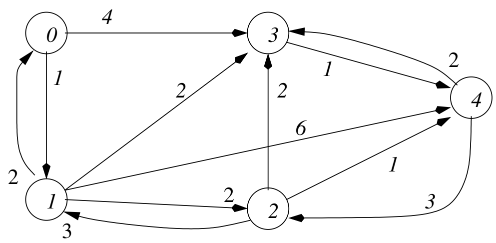
```
            D[z] = D[u] + weight[u][z]; // a lower overestimate exists;
    }

    // At this point, all entries of the array 'D' hold tight estimates.
```

It is clear that when the algorithm finishes, the entries of D will hold overestimates of the lengths of the shortest paths. What is perhaps not clear is why these estimates are actually tight. In order to understand why, first notice that an initial sub-path of a shortest path is itself a shortest path. To see this, suppose that you wish to navigate from a vertex s to a vertex z, and that the shortest path from s to z happens to go through a certain vertex u. Then your path from s to z can be split into two paths, one going from s to u (an initial sub-path) and the other going from u to z (a final sub-path). Given that the whole, unsplit path is a shortest path from s to z, the initial sub-path has to be a shortest path from s to u, for if not, then you could shorten your path from s to z by replacing the initial sub-path to u by a shorter path, which would not only give a shorter path from s to u but also from s to the final destination z. Now it follows that for any start vertex, there is a tree of shortest paths from that vertex to all other vertices. The reason is that shortest paths can't have cycles. Implicitly, Dijkstra's algorithm constructs this tree starting from the root, that is, the start vertex.

If we wish to also compute the shortest path, rather than just its length, we need to introduce a second array `pred` keeping track of the 'previous vertex' (exercise).

The time complexity of this algorithm is quite clearly $O(n^2)$ where n is the number of vertices.

An example. Suppose we want to compute the shortest path from 0 to 4 in the following graph:



An implementation of the above algorithm, with some coded added to write down the intermediate calculations and keep track of predecessors as needed for the exercise, gives the following output:

```
Computing shortest paths from 0


        |0      1       2       3       4
--------+-----------------------------------------------------------------
D       |0      oo      oo      oo      oo
tight   |no     no      no      no      no
pred.   |none   none    none    none    none

Vertex 0 has minimal estimate, and so is tight.

Its neighbour 1 has its estimate decreased from oo to 1 taking a shortcut via 0.
Its neighbour 3 has its estimate decreased from oo to 4 taking a shortcut via 0.


        |0      1       2       3       4
--------+-----------------------------------------------------------------
D       |0      1       oo      4       oo
tight   |yes    no      no      no      no
pred.   |none   0       none    0       none

Vertex 1 has minimal estimate, and so is tight.
```

```
Its neighbour 0 is already tight.
Its neighbour 2 has its estimate decreased from oo to 3 taking a shortcut via 1.
Its neighbour 3 has its estimate decreased from 4 to 3 taking a shortcut via 1.
Its neighbour 4 has its estimate decreased from oo to 7 taking a shortcut via 1.

        |0      1       2       3       4
--------+------------------------------------------------------------------
D       |0      1       3       3       7
tight   |yes    yes     no      no      no
pred.   |none   0       1       1       1

Vertex 2 has minimal estimate, and so is tight.

Its neighbour 1 is already tight.
Its neighbour 3 has its estimate unchanged.
Its neighbour 4 has its estimate decreased from 7 to 4 taking a shortcut via 2.

        |0      1       2       3       4
--------+------------------------------------------------------------------
D       |0      1       3       3       4
tight   |yes    yes     yes     no      no
pred.   |none   0       1       1       2

Vertex 3 has minimal estimate, and so is tight.

Its neighbour 4 has its estimate unchanged.

        |0      1       2       3       4
--------+------------------------------------------------------------------
D       |0      1       3       3       4
tight   |yes    yes     yes     yes     no
pred.   |none   0       1       1       2

Vertex 4 has minimal estimate, and so is tight.

Its neighbour 2 is already tight.
Its neighbour 3 is already tight.

        |0      1       2       3       4
--------+------------------------------------------------------------------
D       |0      1       3       3       4
tight   |yes    yes     yes     yes     yes
pred.   |none   0       1       1       2

End of Dijkstra's computation.
The shortest path from 0 to 4 is:
0 1 2 4 .
```

Dijkstra's algorithm, version II. This time complexity can be improved with the use of priority queues. Here it is convenient to use the convention that low numbers are higher priority.

```
// Input:  A directed graph with weight matrix 'weight' and
//         a start vertex 's'.

// Output: An array 'D' as explained above.
```

```
// We begin by buiding the overestimates.

D[s] = 0;    // Surely the shortest path from s to itself has length zero.

for (each vertex z of the graph)
    if (z isn't the start vertex s)
        D[z] = infinity; // This is certainly an overestimate.

let a priority queue contain all vertices of the graph using
the entries of D as the priorities;

// Now the idea is to build the path three discussed above.
// But this is not done explicitly. Initially, the implicit
// path tree is empty.

while (priority queue is not empty) {
   // The next vertex of the path tree is called u.

   u = remove vertex with smallest priority from queue;

   for (each vertex z in the queue which is adjacent to u)
       if (D[u] + weight[u][z] < D[z]) {
          D[z] = D[u] + weight[u][z]; // a lower overestimate exists;

       Change to D[z] the priority of vertex z in the priority queue;
       }
}

// At this point, all entries of the array 'D' hold tight estimates.
```

Notice that changing the priority of an element of a priority queue requires some rearrangement of the tree that takes $O(\log n)$ steps where n is the size of the queue (exercise). With this, it is relatively easy to see that the time complexity of the second version of the algorithm is $O((m + n) \log n)$, where m is the number of edges and n is the number of vertices of the graph. In a full graph, $m = O(n^2)$ and hence the time complexity reduces to $O(n^2)$ as n^2 grows faster than $n \log n$. Thus, in this case, the time complexity is the same as that of the previous algorithm. However, in practice, it is often the case that $m = O(n)$, that is, there aren't many more edges than nodes, and in this case the time complexity reduces to $O(n \log n)$.

6.5 Shortest path for all vertices.

If we are not just interested in finding the shortest path from one specific vertex to all others, but between every pair of vertices we could, of course, apply Dijkstra's algorithm to every starting vertex. But there is a simpler way of doing this. We keep a matrix 'distance' which contains overestimates of the lengths for every pair of vertices. We decrease overestimates using the same idea as above. In the algorithm given below, we attempt to decrease the estimate of the distance from a vertex s to a vertex z by going systematically via each possible vertex u to see whether there is a shortcut; if there is, the overestimate of the distance is decreased to a smaller overestimate.

```
// Store initial estimates.

for (each vertex s)
    for (each vertex z)
        distance[s][z] = weight[s][z];
```

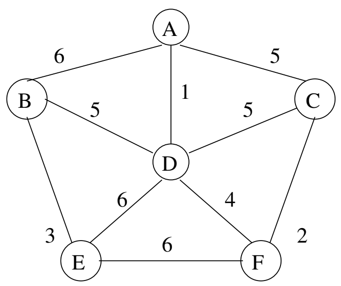
```
// Improve them until they are tight.
for (each vertex s)
    for (each vertex z)
        for (each vertex u)
             if (distance[s][u]+distance[u][z] < distance[s][z])
                distance[s][z] = distance[s][u]+distance[u][z];
```

If we wish to also keep track of which the shortest path is, rather than just its length, we need to introduce a second matrix keeping track of the 'previous vertex' (exercise).

This is known as *Floyd's algorithm*. Its complexity class is quite obviously $O(n^3)$. Again it can be adapted to non-weighted graphs by choosing a suitable weight matrix. In general, Floyd's algorithm will be faster than Dijkstra's, although both are in the same complexity class. This is due to the fact that the former performs fewer instructions in each run through the loops. If the number of edges is small against the number of vertices, however, then, Dijkstra's algorithm can be made to perform in $O(nm + \log n)$, and be faster if one uses the adjacency list representation.

6.6 Minimal spanning trees

Assume you are given a weighted undirected graph like the one to the below:



We could think of the vertices as houses, and the weights as the distances between them. Now imagine you are tasked with supplying all these houses with water, gas, or electricity. For obvious reasons, you will want to keep the amount of digging and laying of pipes or cable to a minimum. What is the best system (that is the one with the shortest length overall) to choose?

Obviously, we will have to choose some of the edges to dig along, but not all of them. If we already have chosen the one between A and D and the one between B and D, there is no reason to also have the one between A and B. More generally, it is clear that we want to avoid *circles*. Secondly, assuming we have just one feeding point (it is of no importance which of the vertices that is), we want the whole system to be *connected*. We have seen that a connected graph without circles is a tree.

Hence we call what we are looking for a (it need not be unique, as we shall see) *minimal spanning tree* of the graph. 'Spanning' here refers to the fact that the tree must contain all the vertices of the original graph (but will make do with fewer edges), so it 'spans' the original graph. (Think of the vertices as being carried by the edges, and the edges as being made of stiff material which will keep its shape.) Minimal refers to the sum of all the weights of the edges contained in that tree.

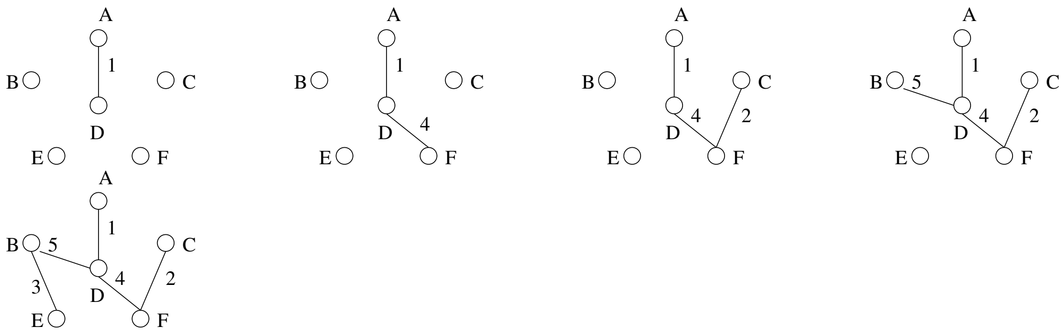
Observations on spanning trees. For the other graph algorithms we have covered so far, we made some observations which allowed us to come up with an algorithm as well as with a strategy for a proof that the algorithm did indeed perform as desired. So to come up with an observation which allows us to develop an algorithm for the minimal spanning tree problem, we shall have to make a few observations about minimal spanning trees. Assume for the time being that all the weights in the above graph were equal, to give us some idea of what kind of shape a minimal spanning tree might have under those circumstances. Here are some examples.

We note that their general shape is such that if we add any of the remaining edges, we will create a circle. Secondly, we notice that going from one spanning tree to another can be achieved by removing an

edge and replacing it by another (two the vertex which would otherwise be unconnected) such that no circle is created. This isn't quite good enough to come up with an algorithm, but it is good enough to let us prove that the ones we find do actually work.

A vertex-based greedy approach. We call an algorithm greedy if it makes decisions based on what is best based on 'local' considerations, with no concern about how this might effect the overall picture. Thus, the idea is to start with an approximation, as in Dijkstra's algorithm, and then refine it in a series of steps. A greedy approach to the minimal spanning tree problem works as follows:

Assume we already have a spanning tree connecting the vertices of some set S. Consider all the edges which connect a vertex in S with one outside of S. Among all these, choose one with a minimal weight. This cannot create a circle, since it must reach a vertex not yet in S. To get started, choose any vertex to be the sole element of the set S, which trivially has a minimal spanning tree containing no edges. For the above graph, starting with A, we get:

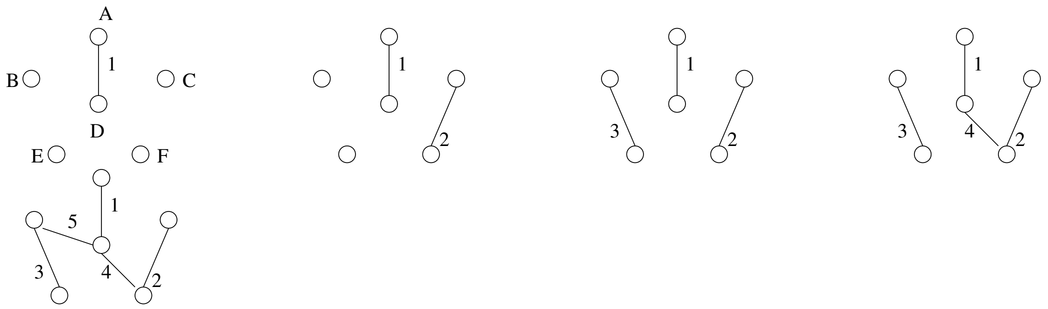


This algorithm is called *Prim's Algorithm*. When implementing it, one can use either an array to keep track of the vertices reached so far (the set S from above), or a list. One could then maintain another array `closest` which for a vertices i not in S yet keeps track of the vertex in S closest to i. (That is the vertex in S which has an edge to i with minimal weight). Further `closest` should keep track of the weight of that edge. That way we only have to check weights mentioned in this array. The complexity of Prim's algorithm is $O(n^2)$: At each step, we add another vertex to S, and then have to update the `closest` array, not dissimilar to Dijkstra's algorithm.

It is slightly more challenging to convince ourselves that this algorithm works than it has been for others we have seen so far. The proof proceeds via induction, and the induction hypothesis is that at stage i, the set of edges chosen so far can be completed to a minimal spanning tree. This is obviously true for the starting set consisting of a single vertex. Assume we have reached stage i, and the current set is S. If we now choose a new edge e as indicated, we will add a new vertex to S. Either e is part of the minimal spanning tree that we know can be made from the edges we had chosen prior to e, and we are done, or e added to that tree would give a circle. But then we can remove an edge from the minimal spanning tree (the other one connecting a vertex in S to one outside which makes up the circle) to remove the circle. Since that other edge has greater or equal weight to e, the resulting tree is minimal, and it is spanning because it still reaches all edges.

Just as with Floyd's versus Dijkstra's algorithm, there is a question whether it is really necessary to consider all vertices at every stage, or whether it is not possible to only check actually existing edges. We therefore introduce another strategy.

An edge-based greedy approach. Our second algorithm ignores vertices entirely, but just keeps building up a minimal spanning tree at each step as follows: Assume we have a collection of edges T already. Now among all the edges not yet in T, choose one with minimal weight such that its addition to T does not produce a circle. For our example from above, the algorithm proceeds as follows:



This technique is known as *Kruskal's algorithm*. It is trickier to implement. We will just note here that this can be achieved in $O(m \log m)$, where m is once again the number of edges. Once again, if the number of edges is much smaller than that of vertices, this algorithm is much faster. If, on the other hand, there are many edges, it becomes slower because it has to perform more instructions than Prim's algorithm.

6.7 Appendix: A full implementation of Dijkstra's algorithm in C

In this program, written in the programming language C, we use "assertions". The C syntax for assertions (as defined in the header file `assert.h`) is

```
assert(<condition>);
```

The meaning of this is that the condition is evaluated (to either true or false) and if the result of the evaluation is false, then the execution of the program aborts (after printing "Assertion failed: `<condition>`").

There are three main uses of assertions: One is documentation. The programmers write what they believe to be true at certain points of the execution of the algorithm. The idea is that this may help (human) readers (including the writer) of the program to understand its logic.

A second is debugging. It is a fact of life that, very often, what the programmers believe to be true is actually plainly wrong. Indeed, when I run the first two versions of this program, assertions did fail, and I had to carefully read the program and try to figure out why this was the case. In later versions, assertions didn't fail, but the program didn't do what I intended it to do either. Hence I included new assertions, some of which proved to be wrong and hence I could see the symptoms of the problem. By means of this device, I could eventually see what was wrong. There were silly mistakes, which nevertheless completely compromised the correctness of the program.

A third use of assertions concerns certain assumptions that are deliberately made when the program is written, and the program makes no promises if these assumptions are violated. For example, I've assumed (quite arbitrarily) that my program is going to cope only with paths with at most 10000 vertices. Hence I included an assertion that will fail if your input violates this assumption.

In this program, I haven't included assertions all over the place, but I have included some that I believe to be crucial.

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *                                                             *
 * Dijkstra's algorithm in C.                                  *
 *                                                             *
 * By MHE, 4th Mar 2003.                                       *
 *                                                             *
 * If you don't know C but know some Java, you should be able  *
 * to read this by making some educated guesses.               *
 *                                                             *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * **/

#include <stdio.h>
#include <limits.h>
#include <assert.h>

/* C lacks some very basic types, but we can easily define them. */

typedef enum {false, true} bool; /* The order is rather important to
                                    get the boolean values right. */
typedef char *string;

/*
  We take the maximum unsigned integer (as defined in the standard
  header value.h) as the value of infinity. So this shouldn't be the
  number of vertices of any path in our graph, because the algorthm
  won't work. (But I haven't been careful enough to include assertions
  to make sure that inputs that violate this assumptions are detected.
  If it is violated, only God knows what is going to happen when you
  run the program. This is definitely not good practice, but, because
  I am not going to sell this program, I just ask the users not to
  attempt bad inputs at home.)

  We define a type of values for weights, which can be changed (to
  e.g. floats) if required. But, as discussed in the lectures, the
  algorithm works for nonnegative values only.
*/

#define oo UINT_MAX /* infinity is represented as the maximum unsigned int. */

typedef unsigned int value;

/*
   In order to avoid complications with IO, we include the graph in
   the program directly. Of course, one wouldn't do that in practice.
*/


/* Firstly, the type of vertices for our particular graph (which was
   given as an example in the lectures). The (non)vertex 'nonexistent'
   is used in order to indicate that a vertex hasn't got a predecessor
   in the path under consideration. See below. */

typedef enum { HNL, SFO, LAX, ORD, DFW, LGA, PVD, MIA, nonexistent} vertex;

/* If you modify the above, in order to consider a different graph,
   then you also have to modify the following. */

const vertex first_vertex = HNL;
```

```
const vertex last_vertex = MIA;

#define no_vertices 8 /* NB. This has to be the same as the number
                         (last_vertex + 1), which unfortunately
                         doesn't compile... */

/* Now a dictionary for output purposes. The order has to match the
   above, so that, for example, name[SFO]="SFO". */

string name[] = {"HNL","SFO","LAX","ORD","DFW","LGA", "PVD", "MIA"};

/* Now the weight matrix of our graph (as given in the
   lectures). Because the graph is undirected, the matrix is
   symmetric. But our algorithm is supposed to work for directed
   graphs as well (and even for disconnected ones - see below). */

value weight[no_vertices][no_vertices] =
  {
   /*  HNL    SFO    LAX    ORD    DFW    LGA   PVD    MIA  */
    {    0,    oo,  2555,    oo,    oo,    oo,    oo,    oo}, /* HNL */
    {   oo,     0,   337,  1843,    oo,    oo,    oo,    oo}, /* SFO */
    { 2555,   337,     0,  1743,  1233,    oo,    oo,    oo}, /* LAX */
    {   oo, 1843,  1743,     0,   802,    oo,   849,    oo}, /* ORD */
    {   oo,    oo,  1233,   802,     0,  1387,    oo,  1120}, /* DFW */
    {   oo,    oo,    oo,    oo,  1387,     0,   142,  1099}, /* LGA */
    {   oo,    oo,    oo,   849,    oo,   142,     0,  1205}, /* PVD */
    {   oo,    oo,    oo,    oo,  1120,  1099,  1205,     0}  /* MIA */
  };


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * The implementation of Dijkstra's algorithm starts here. *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/* We first declare the array 'D' of overestimates, the array 'tight'
   which records which of those estimates are actually tight, and the
   array 'predecessor'. The last has the property that predecessor[z]
   is the last vertex visited on the way from the start node to z. */

value D[no_vertices];
bool tight[no_vertices];
vertex predecessor[no_vertices];

/*
  We begin with the following auxiliary function, which should be
  called only when we know for sure that there is at least one node u
  with tight[u] false. Otherwise, the program aborts; this happens
  when the assertion below fails.
 */

vertex minimal_nontight() {

  vertex j, u;

  for (j = first_vertex; j <= last_vertex; j++)
     if (!tight[j])
        break;
```

```
  assert(j <= last_vertex);

  u = j;

  /* Now u is the first vertex with nontight estimate, but not
     necessarily the one with minimal estimate, so we aren't done
     yet. */

  for (j++; j <= last_vertex; j++)
     if (!tight[j]  &&  D[j] < D[u])
          u = j;

  /* Having checked all vertices, we now know that u has the required
     minimality property. */

  return u;
}

/*
   The following definition assumes that the graph is directed in
   general, which means that it would be wrong to have weight[z][u]
   instead. The intended meaning is that the vertex u has the vertex z
   as one of its successors. Recall that such a vertex z was
   called a neighbour of the vertex u, but this latter terminology is
   best reserved for undirected graphs. Notice that the example given
   above happens to be undirected. But, as remarked above, this
   program is intended to work for directed graphs as well.
 */

bool successor(vertex u, vertex z) {
  return (weight[u][z] != oo  &&  u != z);
}


/*
  We finally arrive at the main algorithm. This is the same as given
  above, now written in C, with the computation of the actual
  paths included. The computation of paths is achieved via the use of
  the array 'predecessor' declared above. Strictly speaking,
  Dijkstra's algorithm only records what is needed in order to recover
  the path. The actual computation of the path is performed by the
  algorithm that follows it.
 */

void dijkstra(vertex s) {

  vertex z, u;
  int i;

  D[s] = 0;

  for (z = first_vertex; z <= last_vertex; z++) {

    if (z != s)
      D[z] = oo;

    tight[z] = false;
    predecessor[z] = nonexistent;
```

```
  }

  for (i = 0; i < no_vertices; i++) {

    u = minimal_nontight();
    tight[u] = true;

    /* In a disconnected graph, D[u] can be oo. But then we just move
       on to the next iteration of the loop. (Can you see why?) */

    if (D[u] == oo)
      continue; /* to next iteration ofthe for loop */

    for (z = first_vertex; z <= last_vertex; z++)
      if (successor(u,z) && !tight[z] && D[u] + weight[u][z] < D[z]) {
        D[z] = D[u] + weight[u][z]; /* Shortcut found. */
        predecessor[z] = u;
      }
  }
}

/* The conditions (successor(u,z) && !tight[z]) can be omitted from
   the above algorithm without changing its correctness. But I think
   that the algorithm is clearer with the conditions included, because
   it makes sense to attempt to tighten the estimate for a vertex only
   if the vertex is not known to be tight and if the vertex is a
   successor of the vertex under consideration. Otherwise, the
   condition (D[u] + weight[u][z] < D[z]) will fail anyway. However,
   in practice, it may be more efficient to just remove the conditions
   and only perform the test (D[u] + weight[u][z] < D[z]). */

/* We can now use Dijkstra's algorithm to compute the shortest path
   between two given vertices. It is easy to go from the end of the
   path to the beginning.

   To reverse the situation, we use a stack. Therefore we digress
   slightly to implement stacks (of vertices). In practice, one is
   likely to use a standard library instead, but, for teaching
   purposes, I prefer this program to be selfcontained. */

#define stack_limit 10000 /* Arbitrary choice. Has to be big enough to
                             accomodate the largest path. */

vertex stack[stack_limit];
unsigned int sp = 0; /* Stack pointer. */

void push(vertex u) {
  assert(sp < stack_limit); /* We abort if the limit is exceeded. This
                               will happen if a path with more
                               vertices than stack_limit is found. */
  stack[sp] = u;
  sp++;
}

bool stack_empty() {
  return (sp == 0);
}
```

```
vertex pop() {
  assert(!stack_empty()); /* We abort if the stack is empty. This will
                             happen if this program has a bug. */
  sp--;
  return stack[sp];
}

/* End of stack digression and back to printing paths. */

void print_shortest_path(vertex origin, vertex destination) {

  vertex v;

  assert(origin != nonexistent  &&  destination != nonexistent);

  dijkstra(origin);

  printf("The shortest path from %s to %s is:\n\n",
         name[origin], name[destination]);

  for (v = destination; v != origin; v = predecessor[v])
    if (v == nonexistent) {
      printf("nonexistent (because the graph is disconnected).\n");
      return;
    }
    else
      push(v);

  push(origin);

  while (!stack_empty())
    printf(" %s",name[pop()]);

  printf(".\n\n");
}


/* We now run an example. */

int main() {

  print_shortest_path(SFO,MIA);

  return 0; /* Return gracefully to the caller of the program
               (provided the assertions above haven't failed). */
}

/* End of program. */
```

This is the output you get when you run the program:

```
The shortest path from SFO to MIA is:

 SFO LAX DFW MIA.
```

Chapter 7

How to measure efficiency

7.1 Time versus space complexity

When creating software for serious application, there often is a need to judge how quickly a program can fulfil a number of tasks. It certainly has to be ensured that the waiting time is reasonable—if, for example, you are programming a flight booking system it is not acceptable if the travel agent and customer have to wait for half an hour for transactions. We call this performance criterion 'time complexity'.

The second criterion that is customarily used to determine efficiency is concerned with how much of the memory a given program will need for a particular task. Here we speak of 'space complexity'. For a given task, there typically are algorithms which trade time for space, or vice versa. For example, we have seen that hash tables have a very good time complexity at the expense of using more memory than is needed by other algorithms. It is up to the program designer to decide which is the better trade-off for the situation at hand.

7.2 Worst versus average

The other thing that has to be decided when making these consideration is whether what is of interest is the *average* performance of a program, or whether it is important to guarantee that even in the *worst* case, performance obeys certain rules. In many every-day applications, the average case will be the more important one, and saving time there may be more important than guaranteeing good behaviour in the worst case. On the other hand, when considering time-critical problems (think of a program that keeps track of the planes in a certain sector), it may be totally unacceptable for the software to take very long if the worst comes to the worst.

Again, algorithms often trade efficiency of the average case versus efficiency of the worst case—*ie.* the most efficient program on average might have a particularly bad worst case. We have seen concrete examples for these when we considered algorithms for searching.

7.3 Concrete measures for performance

For the purpose of this module, we have been mostly interested in *time complexity*. For this, we have to decide how to measure same. Something one might try to do is to just write the program and run it—but this method has a number of pitfalls.

For one, if this is a big application and there are several algorithms, they would have to be programmed first before they can be compared. So a considerable amount of time would be spent on writing programs which won't get used in the final product. Also, the machine the program is run on might influence its running time, or even the compiler used. You would also have to make sure that the *data* with which you test your software is typical for the application it is created for. Again, in particular with big applications, this is not feasible. This empirical method has another disadvantage: it won't tell you anything about the next time you are considering a similar problem.

Therefore (time) complexity is measured differently. Firstly, in order not to be bound to a particular programming language/machine architecture, we typically measure the efficiency of *algorithms* rather than that of *implementations*. For this to be possible, however, the algorithm has to be described in a way which very much *looks* like a program, and is often expressed in what we call 'pseudo-code'.

7.4 What to measure/count

What we do to determine the (time) complexity of an algorithm is then to count the number of operations that will occur, depending on the 'size' of the problem. (The 'size' of a problem is typically given as an integer, and is the number of items that are manipulated. When describing a search algorithm, it is the number of items amongst which we are searching, when describing a sorting algorithm, it is the number of items to be sorted.) So the complexity of an algorithm will be given by a function which maps the number of items to the (often) approximate number of steps the algorithm will take when performed on that many items. Because we often have to approximate such numbers (such as considering an average) we do not demand that the complexity function give us an integer as the result. Instead we will use *real numbers* for that.

In the early days of computers, operations were counted according to their 'cost', and *eg.* typically, multiplication of integers was considered much more expensive than their addition. In today's world, where computers have become much faster, and often have dedicated floating-point hardware, these differences have become less important. Still you should be careful when deciding to consider all operations equally costly—applying some function, for example, will take (sometimes considerably) longer than adding two numbers. For some discussion of this particular issue, see Section 10.8 in [Standish, *Data Structures in Java*, 1998], and don't worry if you don't understand the algorithm he is using there.

If our algorithm is such that we may indeed consider all steps equally costly, then usually the time complexity of the algorithm is determined by the number of loops, as well as how often these loops are being executed (the reason for this is that adding a constant number of instructions which does not change with the size of the problem has no effect on the overall complexity, as we will see in Section 3). We shall consider an example for this in the last section of this chapter.

7.5 Some common complexity classes

Very often we are not interested in the actual function that describes the (time) complexity of an algorithm, but just in its *complexity class*. This basically tells us about the growth of the complexity function, and therefore something about the performance of the algorithm on large numbers of items.

Before we define complexity classes in a more formal manner, let us first gain some intuition for what they actually mean. For this purpose, we choose one function as representative for the classes we wish to consider. We shall see in the next section to which degree a representative can be considered 'typical' for its class. (Recall that we are considering functions which map natural numbers (the size of the problem) to the set of real numbers $\mathbb{R}$.)

The most common classes (in rising order) are the following:

- $O(1)$, 'Oh of one', or *constant* complexity;
- $O(\log \log n)$, 'Oh of log log en';
- $O(\log n)$, 'Oh of log en', or *logarithmic* complexity;
- $O(n)$, 'Oh of en', or *linear* complexity;
- $O(n \log n)$, 'Oh of en log en';
- $O(n^2)$, 'Oh of en squared', or *quadratic* complexity;

- $O(n^3)$, 'Oh of en cubed', or *cubic* complexity;
- $O(2^n)$, 'Oh of two to the en', or *exponential* complexity.

As a representative, we choose the function which gives the class its name—*ie.* for $O(n)$ we choose the function $n \longmapsto n$, also written as $f(n) = n$, and for $O(\log n)$ we choose $n \longmapsto \log n$, also written as $f(n) = \log n$ (this is the logarithm to base 2). So assume we have algorithms with these functions describing their complexity. In the table below we list how many operations it will take them to deal with a problem of a given size.

$f(n)$	$n=2$	$n=16$	$n=256$	$n=1024$	$n=1048576$
1	1	1	1	1.0×10^0	1.0×10^0
$\log\log n$	0[a]	2	3	3.32×10^0	4.32×10^0
$\log n$	1	4	8	1.00×10^1	2.00×10^1
n	2	16	2.56×10^2	1.02×10^3	1.05×10^6
$n\log n$	2	64	2.05×10^3	1.02×10^4	2.10×10^7
n^2	4	256	6.55×10^4	1.05×10^6	1.10×10^{12}
n^3	8	410	1.68×10^7	1.07×10^9	1.15×10^{18}
2^n	4	65536	1.16×10^{77}	1.80×10^{308}	6.74×10^{315652}

[a]Obviously, this doesn't make sense as a complexity, and for this problem size we would have to make an exception.

These numbers get so big that it is somewhat difficult to decide just how long a time span they would describe. Hence here's a table which gives time spans rather than instruction counts, based on the assumption that we have a computer which can operate at a speed of 1 MIP, where one MIP = a million instructions per second.

$f(n)$	$n=2$	$n=16$	$n=256$	$n=1024$	$n=1048576$
1	1 μsec[a]	1 μsec	1 μsec	1 μsec	1 μsec
$\log\log n$		2 μsec	3 μsec	3.32 μsec	4.32 μsec
$\log n$	1 μsec	4 μsec	8 μsec	10 μsec	20 μsec
n	2 μsec	16 μsec	256 μsec	1.02 msec[b]	1.05 sec
$n\log n$	2 μsec	64 μsec	2.05 msec	1.02 msec	21 sec
n^2	4 μsec	256 μsec	65.5 msec	1.05 sec	1.8 wks
n^3	8 μsec	4.1 msec	16.8 sec	17.9 min	$36,559$ yrs
2^n	4 μsec	65.5 msec	3.7×10^{63} yrs	5.7×10^{294} yrs	$2.1 \times 10^{315639} yrs$[c]

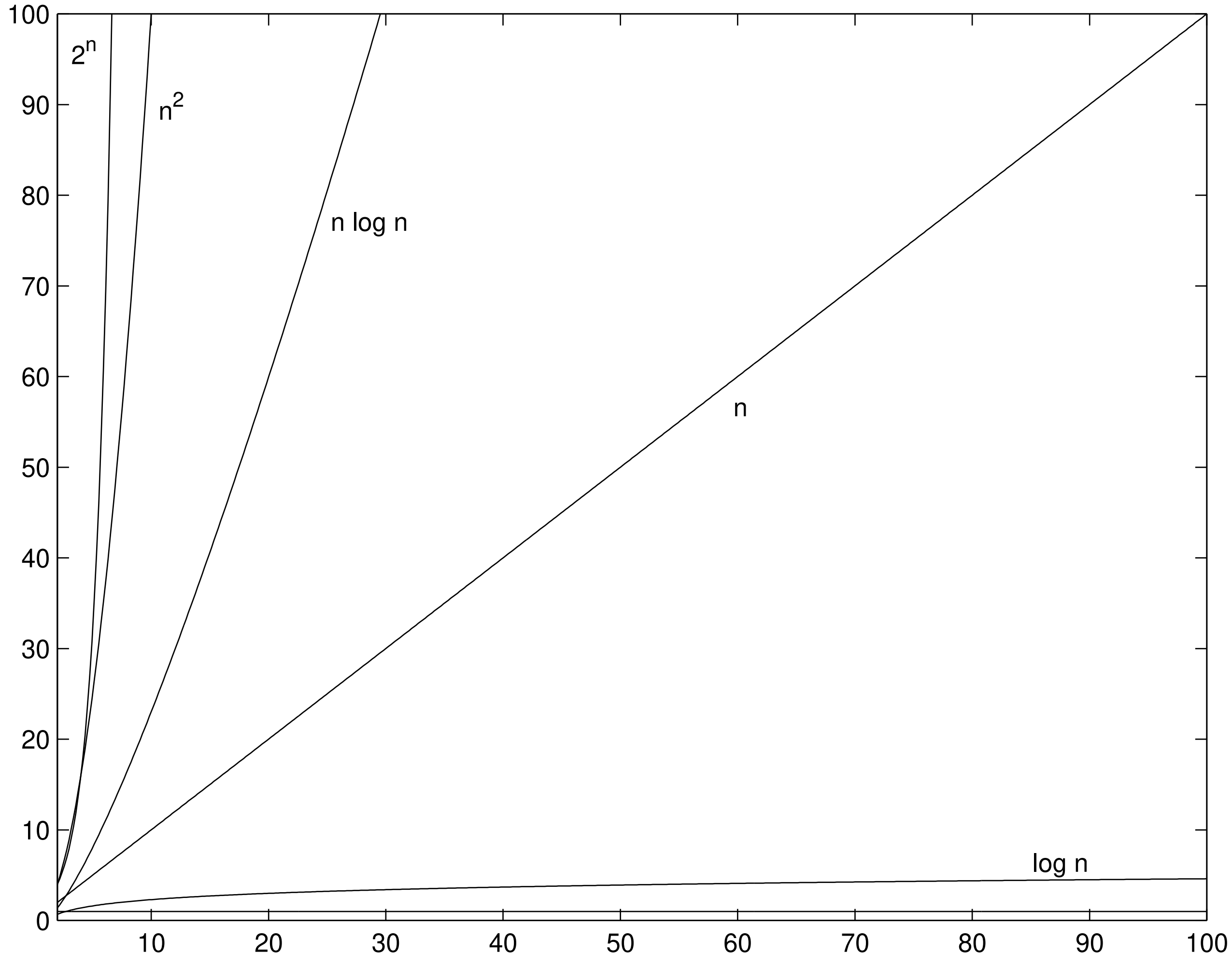
[a]1 μsec is one millionth of a second, or one microsecond
[b]1 msec is one thousandth of a second, or one millisecond
[c]This is an extremely long period of time. Current estimates say that the sun will burn out in another 5×10^9 years. And, of course, we don't have any computers these days which are likely to run uninterrupted for even a single year.

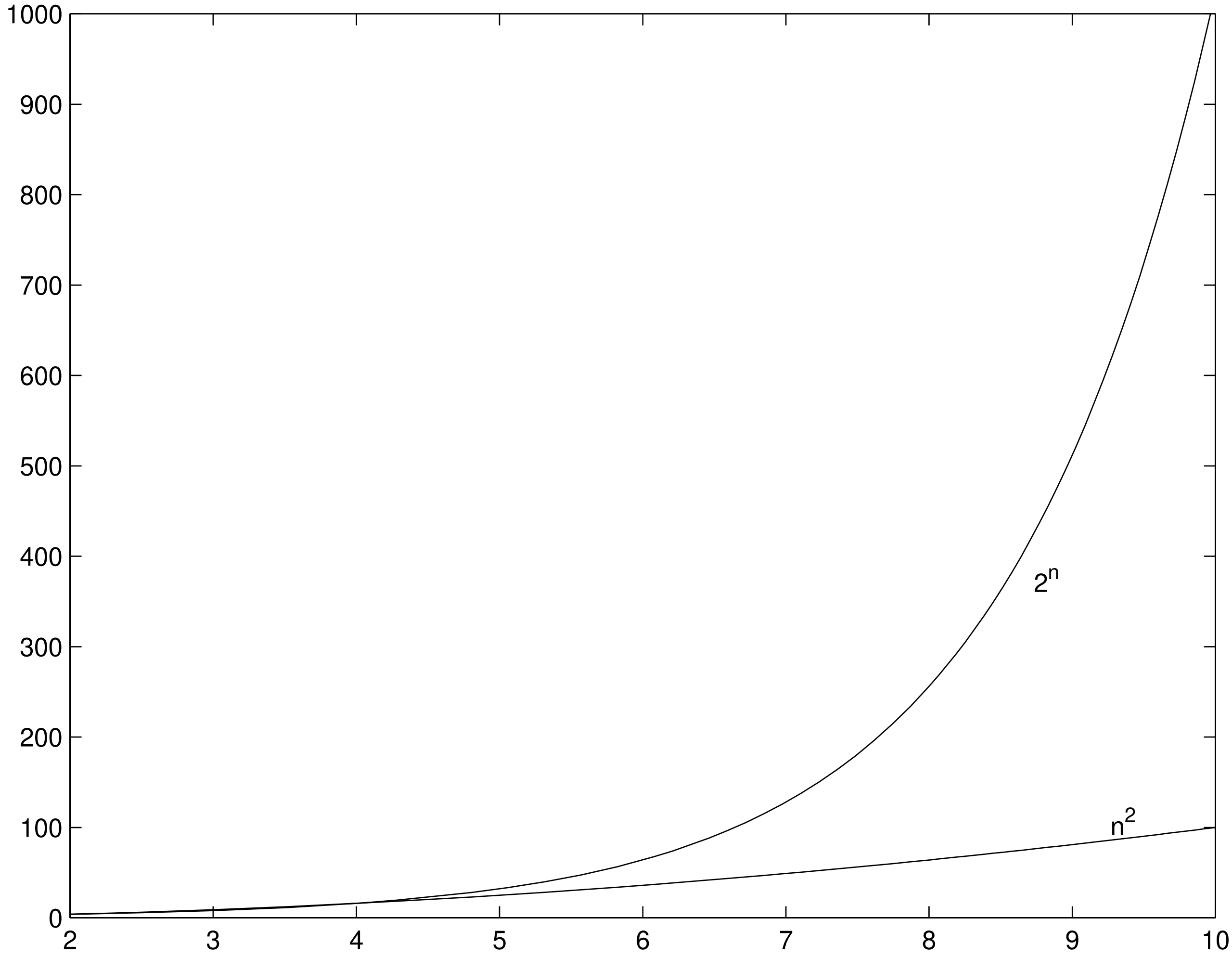
As we can see, if the size of our problem gets really big, there is a huge difference in the time it takes to perform an algorithm of a given class. This is why complexity classes are so important—they tell us how feasible it is to run a program with a big number of data items. Typically, people do not worry much about complexity below the sizes of 10 or maybe 20, but the above numbers should tell you why it is worth thinking about complexity classes where bigger applications are concerned.

Another useful way of thinking about growth classes is thinking about what happens if the problem size doubles. These considerations are given in the following table:

$f(n)$	If the size of the problem doubles then $f(n)$ will be	
1	the same,	$f(2n) = f(n)$
$\log \log n$	almost the same,	$(\log(\log(2n)) = \log(\log(n) + 1);$
$\log n$	more by $1 = \log 2$,	$f(2n) = f(n) + 1;$
n	twice as big as before,	$f(2n) = 2f(n);$
$n \log n$	a bit more than twice as big as before,	$2n \log(2n) = 2(n \log n) + 2n;$
n^2	four times as big as before,	$f(2n) = 4f(n);$
n^3	nine times as big as before,	$f(2n) = 8f(n);$
2^n	the square of what it was before,	$f(2n) = (f(n))^2.$

We now give a plot of some of the functions from the table. Note that although these functions are only defined on natural numbers, we draw them as if they were defined for all real numbers, because that makes it easier to take in the information presented.





7.6 Complexity classes

We have said that complexity classes are concerned with *growth*, and the tables above have given you some idea of what different behaviour means when it comes to growth. There we have chosen a representative each of the complexity classes considered, but we have'nt said anything about just how 'representative' such an element is. Let us start with the formal definition of a 'big Oh' class.

Definition. A function g belongs to the *complexity class* $O(f)$ if there is a number $n_0 \in \mathbb{N}$ and a constant $c > 0$ such that for all $n \geq n_0$, we have that $g(n) \leq c * f(n)$. We say that the function g is 'eventually smaller' than the function $c * f$.

What does it mean? Firstly, we are ignoring *when* g becomes smaller than $c * f$. We are only interested in the *existence* of n_0 such that, from then on, g is smaller than $c * f$. Secondly, we wish to consider the efficiency of an algorithm independently of the speed of the computer that is going to execute it. This is why f is multiplied by a constant. The idea is that when we measure the time of the steps of a particular algorithm, we are not sure how long each of them takes. By definition, $g \in O(f)$ means that eventually (namely beyond the point n_0), the growth of g will be at most as much as the growth of $c * f$. This definition also makes it clear that constant *factors* do not change the growth class (or O-class) of a function. Hence $n \longmapsto n^2$ is in the same growth class as $n \longmapsto 1/1000000n^2$ or $n \longmapsto 1000000n^2$. So we can write $O(n^2) = O(1000000n^2) = O((1/1000000)n^2)$. Typically, however, we choose the *simplest* representative, as we did in the tables above. In this case it is $O(n^2)$.

The classes we mentioned above are related as follows:

$$O(1) \subseteq O(\log\log n) \subseteq O(\log(n)) \subseteq O(n) \subseteq O(n\log n) \subseteq O(n^2) \subseteq O(n^3) \subseteq O(2^n)$$

When adding functions from different growth classes, their sum will always be in the larger growth class. This allows us to simplify terms. For example, the growth class of $n \longmapsto 500000\log n + 4n^2 + .3n + 100$ can be determined as follows. The summand with the largest growth class is $4n^2$ (we say that this is the 'dominating sub-term' of the function), and we are allowed to drop constant factors, so this function is in the class $O(n^2)$.

Now have another look at Figure 1. Can you see why we talk about 'different growth classes' now?

When we say that an algorithm 'belongs to' some class $O(f)$, we mean that it is *at most* as fast growing as f. We have seen that 'linear searching' (where one searches in a collection of data items which is unsorted) has linear complexity, *ie.* it is in growth class $O(n)$. This holds for the *average* as well as the *worst* case: The operations needed are comparisons of the key we are searching for with the keys appearing in our data collection. In the worst case, we have to check all n entries until we find the right one, which means we make n comparisons. On average, however, we will only have to check $n/2$ entries until we hit the correct one, leaving us with $n/2$ operations. Both those functions, $n \longmapsto n$ and $n \longmapsto n/2$ belong to the same complexity class, namely $O(n)$. However, it would be equally correct to say that the algorithm belongs to $O(n^2)$, since that class contains all of $O(n)$. But this would be *less informative*, and we would not say that an algorithm has quadratic complexity if we know that, in fact, it is linear. Sometimes it is difficult to be sure what the exact complexity is (as is the case with the famous P=NP problem), in which case one might say that an algorithm is 'at most', say, quadratic.

7.7 How to determine the complexity of an algorithm

So how does one go about counting the number of operations an algorithm needs? We have already seen many examples. Let's consider insertion sort again.

If we have an (unsorted) array `a[]` of integers of length n, the algorithm searches for the smallest element of the array and then swaps it with the element in the first position, `a[0]`. Then it seeks the smallest element in the array from position 1 to position $n-1$ and swaps it with $a[1]$, and so on.

The sorting method would have a definition like this:

```
for (int i = 0; i< n; i++) {
               // find the smallest element, a[k], in a[i] to a[n-1]
               // swap a[k] with a[i]
}
```

Filling in just a bit more code we get:

```
for (int i = 0; i< n; i++) {
  int k;
  for (int j=i+1; j<n, j++) {
    k = i;
    if (a[j] < a[k]) then k = j;
  }
  /* k now contains the index of the smallest element from
     a[i] to a[n-1]
  */
  int l = a[i];
  a[i] = a[k];
  a[k] = l;  // this does the swapping
}
```

So how many operations does this program execute? There is first of all an outer loop which is run through n times. Then there is the inner loop, and it depends on the counter of the outer loop how often we go through that. For each run of the inner loop, we have one comparison, but we don't always get an assignment, due to the 'if' clause. If we are interested in the average case, we would assume that about half the time, the 'if' clause is true. And then we get the three assignments each time we complete the outer loop.

This gives us the following number of operations carried out on average (where we ignore the fact that incrementing loop variables, etc, costs a bit of time, too):

$$
\begin{aligned}
f(n) &= \sum_{i=0}^{n-1}((\sum_{i+1}^{n-1} 1 + \frac{1}{2}) + 3) \\
&= \sum_{i=0}^{n-1}((n-i-1)\frac{3}{2} + 3) \\
&= \frac{3}{2}\sum_{i=0}^{n-1}(n-i-1) + 3n \\
&= \frac{3}{2}((n-1) + (n-2) + \cdots + 1) + 3n \\
&= \frac{3}{2}n(n-1) + 3n \\
&= \frac{3}{2}n^2 + \frac{3}{2}n
\end{aligned}
$$

Therefore this algorithm has complexity $O(n^2)$ which, as we know, isn't a very good complexity class, and we have seen that there are better sorting algorithms. One thing to note is that when considering sorting algorithms one often measures complexity via the number of *comparisons* that occur, ignoring things such as assignments, *etc*. Note that if we only count comparisons in the example above, we get $f(n) = (n-1) + (n-2) + \cdots + 1 = n(n-1)/2 = (1/2)n^2 + (1/2)n$. So the complexity class stays the same. This is typical for the (so-called 'internal') sorting algorithms, and the reason why people like to restrict themselves to counting the number of comparisons involved is that it yields a good measure for comparing the various sorting algorithms. What exactly to count when considering the complexity of a particular algorithm is always a judgement call. You will have to gain more experience before you feel comfortable with making such calls yourself.

Something to be aware of when making these calculations is that it isn't a bad idea to keep track of any factors, in particular those that go with the dominating sub-term. In the above example, the factor applied to the dominating sub-term, namely n^2, was $3/2$ (and by coincidence, this was also the factor that came with the second term, namely n). It is good and well to know that an algorithm that is linear will perform better than a quadratic one *provided the size of the problem is large enough*, but if you know that your problem has a size of, say, at most 100 then a complexity of $(1/10)n^2$ might be preferable to one of $1000000n$. Or if you know that your program is only ever used on fairly small samples, then using the simplest algorithm you can find might be beneficial over all—it is easier to program, and there is not a lot of time to be saved.

Appendix A

Some formulas

The symbols r and s stand for real numbers, m and n for natural ones. There are variables a, b and c assumed to range over the real numbers.

A.1 Binomial formulas and the like

$(a+b)^2 = a^2 + 2ab + b^2$ $\qquad$ $(a+b)(a-b) = a^2 - b^2$

$(a-b)^2 = a^2 - 2ab + b^2$

$(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$ $\qquad$ $(a-b)^3 = a^3 - 3a^2b + 3ab^2 - b^3$

A.2 Powers and roots

$a^0 = 1$ $\qquad$ $a^1 = a$

$a^{-r} = 1/(a^r)$ $\qquad$ $a^{1/n} = \sqrt[n]{a}$

$a^r a^s = a^{r+s}$ $\qquad$ $a^r/a^s = a^{r-s}$

$a^s b^s = (ab)^s$ $\qquad$ $a^s/b^s = (a/b)^s$

$(a^r)^s = a^{rs} = a^{sr} = (a^s)^r$

The following are special cases of the above:

$\sqrt[n]{a}\,\sqrt[n]{b} = \sqrt[n]{ab}$ $\qquad$ $a^{m/n} = \sqrt[n]{a^m} = \sqrt[n]{a}^m$

$\sqrt[n]{a}/\sqrt[n]{b} = \sqrt[n]{a/b}$ $\qquad$ $a^{-(m/n)} = 1/(\sqrt[n]{a^m}) = 1/(\sqrt[n]{a})^m$

A.3 Logarithms

Definition: The *logarithm of c to base a*, written as $\log_a c$, is the real number b satisfying the equation $c = a^b$. For that we assume that $c > 0$ and $a > 1$.

There is a special case, namely $\log_a 1 = 0$ since $a^0 = 1$. Further we have that $\log_a a = 1$ since $a^1 = a$. From the definition, we get immediately that

$$a^{\log_a c} = c \text{ and } \log_a a^b = b.$$

We can move from one base to another via

$$\log_{a'} b = \log_{a'} a * \log_a b.$$

A.4 Rules for logarithms

$\log_a(bb') = \log_a b + \log_a b'$ $\log_a(b/b') = \log_a b - \log_a b'$
$\log_a(b^r) = r \log_a b$

The following are special cases of the above:

$\log a^n = n \log a$ $\log \sqrt[n]{a} = (1/n) \log a$

A.5 Sums

We often find it useful to abbreviate a sum $a_1 + \cdots + a_n$ with $\sum_{i=1}^{n} a_i$.

We can view this as a little algorithm if we wish: Let us use `s` to hold the above sum at the end of the program, and assume that `double[] a` is an array holding the numbers we wish to add, that is: `a[i] =` a_i.

```
double s=0;
for (i=1; i<=n; i++)
    s = s + a[i];
```

The most common use of sums for our purposes is when investigating the complexity of a program. For that, we usually have to count a variant of $1 + 2 + \cdots + n$. Hence it is helpful to know that

$$\sum_{i=1}^{n} i = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}.$$

In order to illustrate this, consider the program

```
for (i=0; i<n; i++)
{ for(j=0; j<=i; j++)
  { k++ //dummy instruction 1
    l++ //dummy instruction 2
    m++ //dummy instruction 3
  }
}
```

By the above discussion, its complexity is given as follows:

$$\sum_{i=0}^{n-1} \sum_{j=0}^{i} 3 = \sum_{i=0}^{n-1} (i+1)3 = 3 \sum_{i=0}^{n-1} (i+1) = 3 \sum_{i=1}^{n} i = 3\frac{n(n+1)}{2}.$$

Appendix B

Differences from the printed version you have

There are none at the moment, but any significant changes will be explained here so that it is not necessary to print the notes again.