ALGORITHMS AND DATA STRUCTURES

Design, Correctness, Analysis

ALGORITHMS AND DATA STRUCTURES

Design, Correctness, Analysis

Jeffrey H. Kingston

University of Sydney

Sydney · Wokingham, England · Reading, Massachusetts
Menlo Park, California · New York · Don Mills, Ontario
Amsterdam · Bonn · Singapore · Tokyo · Madrid · San Juan

Cover designed by Crayon Design
Typeset by Times Graphics
Printed and bound in Singapore

First printed 1990.

National Library of Australia Cataloguing in Publication Data
Kingston, Jeffrey H. (Jeffrey Howard).
Algorithms and data structures : design, correctness, analysis.

Bibliography.
Includes index.
ISBN 0 201 41705 7.

1. Algorithms. 2. Data structures (Computer science).
I. Title.

005.1

Library of Congress Cataloging-in-Publication Data
Kingston, Jeffrey H.
Algorithms and data structures : design, correctness, analysis
Jeffrey H. Kingston.
p. cm.
ISBN 0-201-41705-7
1. Data structures (Computer science) 2. Algorithms. I. Title.
QA76.9.D35K524 1990
005.7'3--dc20

90-163
CIP

Preface

This book is intended as a text for a one-semester second or third year course on algorithms and data structures. It aims to present the central topics of the subject, coherently organized, with emphasis more on depth of treatment than on broad survey.

My motive in choosing depth over breadth was a desire to involve the student by making all the material fully accessible. For example, an average complexity analysis is the best way to justify the use of Quicksort, but since this is not easy to perform, some preparation in analysis techniques is needed. Similarly, the correctness of Dijkstra's algorithm is not clear, and it requires a proof using loop invariants. Thus are we led naturally to depth of treatment.

I have retained the traditional organization by application area for most of the book (Chapters 6–10). This brings together alternative solutions to the various problems, and makes manifest to the student the general scope of the subject in a way that a text structured around design or analysis techniques cannot do. Chapters 1–4 are devoted to techniques: correctness, analysis, the use of abstract data types, and algorithm design.

Finding the right level of treatment for correctness is difficult. Proofs of correctness are essential for some of the graph algorithms of Chapter 10, and the presentation of even quite simple algorithms can be improved by giving their loop invariants; but a formal treatment including predicate transformers would easily fill an entire book. I have compromised, using informal arguments to establish formal invariants, and including an introductory chapter that could be assigned as reading.

By counting the number of times that a characteristic operation is performed, the analyses give quite precise results, without excessive detail. Amortized complexity, an unusual feature of this book, is the key to some exciting new data structures – notably Fibonacci heaps, which lead to an optimal implementation of Dijkstra's algorithm.

Abstract data types have helped greatly in organizing the subject matter, both by classifying and specifying data structures, and by removing them from the algorithms. They permeate the book, and there are whole chapters devoted to the implementation of three important ones: the symbol table, the priority queue, and the disjoint sets structure.

For algorithm design, the usual list of strategies is presented, and the reader is invited to consider applying each to the problem at hand. Backtracking and branch-and-bound have been omitted, since they are most often applied to NP-hard and artificial intelligence problems that lie outside the scope of this text.

As I wrote this book, I perceived a need for a more systematic classification of iterative algorithms than is usually given. To this end, I have identified two distinctive kinds of loop invariant, the first occurring in such trivial algorithms as summing an array and insertion sort, and the second in more subtle algorithms, including the greedy algorithms. This classification is presented in Section 4.2.

The book is entirely self-contained in its treatment of correctness, analysis of algorithms (except basic probability theory), data abstraction, and algorithm design. Some knowledge of the kind usually imparted in a second programming course is assumed: familiarity with a Pascal-like programming language, linked structures, and recursion.

Executable Modula-2 code is given in nearly all cases; it has been compiled and tested. The major technical issue in choosing a programming language for presenting this material must be the degree of support provided for data abstraction. Modula-2 has the necessary modules and opaque types; regrettably, the absence of type inheritance, generic modules, procedure name overloading, and function value dereferencing leads to some loss of generality or readability in some programs.

Specific attributions are given throughout the text. More generally, I am indebted to a number of previous authors, especially to Aho *et al.* (1983) for my choice of subject areas, as well as many of the individual topics; to Tarjan (1983), whose monograph provided a model for my organization, and some of its most interesting material; and to Knuth (1973a, b) for general inspiration, as well as most of the analyses in Chapters 5–8.

Several people generously gave their time during the development of the book at Sydney University. Greg Ryan carefully read the manuscript; Stephen Russell and John Gough assisted with Modula-2; and Bryden Allen, Greg Butler, Norman Foo, and Alan Fekete gave reviews and advice. My thanks also to my thesis supervisor, Allan Bromley, for his encouragement over the years.

This book has grown from courses taught while visiting the University of Iowa in 1984–7, and, in a general way, owes much to my colleagues there, especially Donald Alton, Keith Brinck, and Douglas Jones, and to the congenial environment I found at Iowa. Accordingly, I dedicate the book with gratitude to my many American friends.

Contents

Preface		v
Chapter 1	**Algorithm Correctness**	**1**
1.1	Problems and Specifications	1
1.2	Recursive Algorithms	2
1.3	Iterative Algorithms	4
	Exercises	9
Chapter 2	**Analysis of Algorithms**	**13**
2.1	Characteristic Operations and Time Complexity	13
2.2	Recursive Algorithms	16
2.3	Iterative Algorithms	21
2.4	Evaluating Efficiency, and the O-notation	26
	Exercises	28
Chapter 3	**Data Abstraction**	**36**
3.1	Abstract Data Types	36
3.2	Lists, Stacks, and Queues	40
3.3	Correctness of ADT Implementations	43
3.4	Analysis of ADT Implementations	47
3.5	Amortized Analysis	50
	Exercises	55
Chapter 4	**Algorithm Design**	**62**
4.1	The Design Process	62
4.2	Incremental Algorithms	64
4.3	Divide-and-Conquer	66
4.4	Dynamic Programming	71
	Exercises	77
Chapter 5	**Properties of Binary Trees**	**82**
5.1	Definitions	82

5.2	Height and Path Length	83
5.3	Skew Trees and Complete Trees	85
5.4	Huffman Trees	88
	Exercises	92
Chapter 6	**Symbol Tables**	**96**
6.1	Specification	96
6.2	Linked Lists	97
6.3	Locality of Reference and Self-adjusting Lists	98
6.4	Binary Search Trees	103
6.5	Analysis of Binary Search Tree Insertions	105
6.6	Splay Trees	110
6.7	B-trees	116
6.8	Hashing	122
6.9	Choosing a Symbol Table Implementation	126
	Exercises	127
Chapter 7	**Priority Queues**	**142**
7.1	Specification	142
7.2	Heap-ordered Trees	144
7.3	The Heap	147
7.4	Heapsort	149
7.5	Binomial Queues	153
7.6	Fibonacci Heaps	156
7.7	Choosing a Priority Queue Implementation	163
	Exercises	164
Chapter 8	**Sorting**	**175**
8.1	Insertion Sorting	176
8.2	Selection Sorting	181
8.3	Merging and Mergesort	182
8.4	Quicksort	187
8.5	Radix Sorting	190
8.6	Choosing a Sorting Algorithm	193
	Exercises	194
Chapter 9	**Disjoint Sets**	**202**
9.1	Specification	202
9.2	The Galler–Fischer Representation	206
9.3	Union by Size	208
9.4	Path Compression	210
	Exercises	216

Chapter 10 Graph Algorithms 219

10.1 Definitions 220
10.2 Specification and Representation 221
10.3 Relations and Digraphs 223
10.4 Directed Acyclic Graphs 228
10.5 Shortest Paths and Breadth-first Search 236
10.6 Depth-first Search 245
10.7 Strongly Connected Components 248
10.8 Biconnectivity 250
10.9 Minimum Spanning Trees and Kruskal's Algorithm 254
10.10 Prim's Algorithm 261
10.11 The Travelling Salesperson Problem 267
Exercises 272

Chapter 11 Lower Bounds 284

11.1 Models of Computation 285
11.2 Adversary Bounds 286
11.3 Decision Trees 288
11.4 Entropy 292
11.5 Transformations 297
Exercises 301

Recommended Further Reading 304

References 305

Index 309

Chapter 1

Algorithm Correctness

There are several good reasons for studying the correctness of algorithms, for example, to improve the quality of the programs we write, or of the languages we use. But the reasons for doing it here, in this book, are as follows.

First, some algorithms are so mysterious as to defy intuition. To understand these algorithms formal methods must be used.

Second, although it is true that every algorithm depends for its correctness on specific properties of the problem at hand, there must also be a strategy for putting those properties to work. To prove an algorithm correct is to reveal this strategy: it can then be used in comparisons with other algorithms, and in the development of new algorithms.

The study of correctness, as we will go about it, is known as *axiomatic semantics*, and it is principally owing to Floyd (1967) and Hoare (1969). It is possible, using the methods of axiomatic semantics, to prove that a program is correct as rigorously as one can prove a theorem in logic. This will not be attempted here, because it is an entire subject in itself (Dijkstra, 1976; Gries, 1981); instead, a less rigorous approach will be taken which is compatible with the fully rigorous one, but which is more appropriate to our aims of understanding, comparing, and developing algorithms.

1.1 Problems and Specifications

A *problem* is a general question to be answered, usually possessing several *parameters*. A problem is specified by describing the form we expect the parameters to take and the question we ask about them. For example, the *minimum-finding problem* is "S is a set of numbers. What is a minimum element of S?" It has one parameter, S.

An *instance* of a problem is an assignment of values to the parameters. For example, '$S = \{5, 2, 6, 9\}$' is an instance of the minimum-finding problem.

An *algorithm* for a problem is a step-by-step procedure for taking any instance of the problem and producing a correct answer for that instance. If several answers are equally correct, as often happens, the algorithm is free to produce any one. An algorithm is *correct* if it is guaranteed to produce a correct answer to every instance of the problem.

Specifying a problem can be a difficult task in itself, because there is a need for great precision. For example, the empty set has no minimum element, so the specification given above for the minimum-finding problem is flawed. A good way to state a specification precisely is to give two Boolean expressions, or *conditions*: the first, called the *precondition*, states what may be assumed to be true initially; the second, called the *postcondition*, states what is to be true about the result. For example, the minimum-finding problem could be specified like this:

Pre: S is a finite, non-empty set of integers
Post: m is a minimum element of S.

We could write (there exists $x \in S$ such that $m = x$) **and** (for all $x \in S$, $m \leq x$) to express more formally what it means for m to be a minimum element of S. By assuming that all instances are non-empty, we are saying that we don't care what an algorithm for this problem does if it is given the empty set. It is the user's responsibility to supply only instances in accord with the precondition.

1.2 Recursive Algorithms

Newcomers to recursion are often confused by the apparent circularity of the method: to solve a problem, first solve the problem. But it is misleading to view recursion in this way; rather, one *instance* is solved by solving one or more different, and smaller, instances. When proving that a recursive algorithm finds the correct solution to some instance, we therefore need to assume that it finds correct solutions to these smaller instances, and this suggests immediately that we should use induction on the size of the instance to prove correctness.

As a first example, consider the problem of calculating $n!$, whose specification is

```
(* Pre: n is an integer such that n ≥ 0 *)
x := Factorial(n);
(* Post: x = n! *)
```

The convention of including the conditions as comments in a program fragment at the points where they should be true has been adopted. The usual recursive algorithm for this problem is

```
procedure Factorial(n: integer): integer;
begin
  if n = 0 then
    return 1;
  else
    return n*Factorial(n − 1);
  end;
end Factorial;
```

and its proof of correctness is as follows:

Theorem 1.1: For all integers $n \geq 0$, *Factorial*(n) returns $n!$.

Proof: by induction on n.

Basis step: $n = 0$. Then the test $n = 0$ succeeds, and the algorithm returns 1. This is correct, since $0! = 1$.

Inductive step: The inductive hypothesis is that *Factorial*(j) returns $j!$, for all j in the range $0 \leq j \leq n - 1$. It must be shown that *Factorial*(n) returns $n!$. Since $n > 0$, the test $n = 0$ fails and the algorithm returns n**Factorial*($n - 1$). By the inductive hypothesis, *Factorial*($n - 1$) returns $(n - 1)!$, so *Factorial*(n) returns $n \times (n - 1)!$, which equals $n!$. ■

Notice that the proof is only possible because the recursive call is given a smaller instance than the original, so that the inductive hypothesis may be applied to it. Also, the theorem says nothing about the behaviour of *Factorial*(n) for $n < 0$, and in fact the algorithm never halts for these n.

The second example is the *binary search algorithm*. Its goal is to determine whether a number x is present in the sorted array $A[a..b]$:

```
(* Pre: a ≤ b + 1 and A[a..b] is a sorted array *)
found := BinarySearch(A, a, b, x);
(* Post: found = x ∈ A[a..b] and A is unchanged *)
```

Binary search first compares x with the middle element of the array, $A[mid]$. If $x < A[mid]$, it must lie in the left half of the array if it is present at all; if $x > A[mid]$, it must lie in the right half. The algorithm is naturally expressed recursively:

```
procedure BinarySearch(var A: AType; a, b: integer; x: KeyType):
boolean;
var mid: integer;
begin
  if a > b then return false;
  else
    mid := (a + b) div 2;
    if    x = A[mid] then return true;
     elsif x < A[mid] then return BinarySearch (A, a, mid - 1, x);
     else                  return BinarySearch (A, mid + 1, b, x);
     end;
  end;
end BinarySearch;
```

A has been made a **var** parameter for efficiency's sake; its value does not change. The proof of correctness is by induction on the size of the array $A[a..b]$:

Theorem 1.2: For all $n \geq 0$, where $n = b - a + 1$ equals the number of elements in the array $A[a..b]$, *BinarySearch*(A, a, b, x) correctly returns the value of the condition $x \in A[a..b]$.

Proof: by induction on n.

Basis step: $n = 0$. The array is empty, so $a = b + 1$, the test $a > b$ succeeds, and the algorithm returns **false**. This is correct, because x cannot be present in an empty array.

Inductive step: $n > 0$. The inductive hypothesis is that, for all j such that $0 \leq j \leq n - 1$, where $j = b' - a' + 1$, *BinarySearch*(A, a', b', x) correctly returns the condition $x \in A[a'..b']$. From the calculation $mid := (a + b)$ **div** 2 it may be concluded that $a \leq mid \leq b$. If $x = A[mid]$, clearly $x \in A[a..b]$ and the algorithm correctly returns **true**. If $x < A[mid]$, since A is sorted it may be concluded that $x \in A[a..b]$ if and only if $x \in A[a..mid - 1]$. By the inductive hypothesis, this second condition is returned by *BinarySearch*$(A, a, mid - 1, x)$. The inductive hypothesis does apply, since $0 \leq (mid - 1) - a + 1 \leq n - 1$. The case $x > A[mid]$ is similar, and so the algorithm works correctly on all instances of size n. ■

1.3 Iterative algorithms

In this section the technique used to prove the correctness of an algorithm containing a **while** loop is explained. The following algorithm, which determines the sum of the elements of $A[a..b]$, will be used as the first example:

```
(* Pre: a ≤ b + 1 *)
i := a; sum := 0;
while i ≠ b + 1 do
  sum := sum + A[i];
  i := i + 1;
end;
(* Post: sum = Σ_{j=a}^{b} A[j] *)
```

As usual, the precondition and postcondition are included as comments at the points where they should be true. Note that, by definition, $A[a..a-1]$ denotes an empty array whose sum is 0, and this algorithm calculates this empty sum correctly.

The key step in the proof of correctness is the invention of a condition, called the *loop invariant* of the algorithm, which is supposed to be true at the beginning and end of each iteration of the **while** loop. By the 'beginning of an iteration' is meant the moment just before the boolean test at the top of the loop is executed.

There may be several loop invariants, such as **true** or (in the example above) $a \leq b + 1$; but to be useful a loop invariant must capture the relationship among the variables that change in value as the loop progresses. The loop invariant for the algorithm above is $sum = \Sigma_{j=a}^{i-1} A[j]$, which expresses the relationship between the variables *sum* and *i*; the reader may easily verify intuitively that this condition holds at the beginning and end of each iteration.

For the record, and as a model for the more difficult proofs in later sections of this book, here is a proof that the condition really is a loop invariant:

Theorem 1.3: (Loop invariant of summing algorithm) At the beginning of the kth iteration of the summing algorithm above, the condition $sum = \Sigma_{j=a}^{i-1} A[j]$ holds.

Proof: by induction on k.

Basis step: $k = 1$. At the beginning of the first iteration, the initialization statements clearly ensure that $sum = 0$ and $i = a$. Since $0 = \Sigma_{j=a}^{i-1} A[j]$, the condition holds.

Inductive step: The inductive hypothesis is that $sum = \Sigma_{j=a}^{i-1} A[j]$ at the beginning of the kth iteration. Since it has to be proved that this condition holds after one more iteration, it is also assumed that the loop is not about to terminate, that is, that $i \neq b + 1$. Let sum' and i' be the values of *sum* and *i* at the beginning of the $(k+1)$st iteration. We are required to show that $sum' = \Sigma_{j=a}^{i'-1} A[j]$. Since $sum' = sum + A[i]$, and $i' = i + 1$, we have

$$\begin{aligned} sum' &= sum + A[i] \\ &= \sum_{j=a}^{i-1} A[j] + A[i] \\ &= \sum_{j=a}^{i} A[j] \\ &= \sum_{j=a}^{i'-1} A[j], \end{aligned}$$

and so the condition holds at the beginning of the $(k + 1)$st iteration. ■

Establishing the loop invariant is invariably the hard part of the proof, but there are two easier steps remaining. First of all, the postcondition must be shown to hold at the end. Consider the last iteration of the loop in the summing algorithm. At the end of it, the loop invariant holds, as we have shown. Then the test $i \neq b + 1$ is made, fails, and execution passes to the point after the loop. Clearly, at that moment the condition

$$sum = \sum_{j=a}^{i-1} A[j] \textbf{ and } i = b + 1$$

holds. But

$$sum = \sum_{j=a}^{i-1} A[j] \textbf{ and } i = b + 1$$

$$\Rightarrow sum = \sum_{j=a}^{b} A[j]$$

which is the desired postcondition; so that *Post* has been shown to hold when the algorithm terminates. Notice that this conclusion could not have been reached so simply if $i \leq b + 1$ had been used as the condition at the top of the loop. In general, just after the completion of the execution of the loop '**while** B . . .', with loop invariant I, the condition I **and not** B holds, and it is necessary to prove that this implies *Post*.

The final step is to show that there is no risk of an infinite loop. The method of proof is to identify some integer quantity that is strictly increasing (or decreasing) from one iteration to the next, and to show that when this becomes sufficiently large (or small) the loop must terminate.

For the summing algorithm, i is strictly increasing, and when it reaches $b + 1$ the loop must terminate. Note that this argument depends on i being no greater than $b + 1$ initially; in other words, the condition $a \leq b + 1$ must be true initially in order for termination to be guaranteed.

To summarize, then, the steps required to prove that the iterative algorithm

```
(* Pre *)
...
while B do
   ...
end;
(* Post *)
```

is correct are as follows:

(1) Guess a condition I.

(2) Prove by induction that I is a loop invariant.

(3) Prove that I **and not** $B \Rightarrow$ *Post*.

(4) Prove that the loop is guaranteed to terminate.

With practice, a clear intuitive understanding of the correctness of an algorithm will lead immediately to the loop invariant. Remember that it involves all the variables whose values change within the loop, but that it expresses an unchanging relationship among those variables. It must also contain complete information about what the algorithm has achieved.

For example, the loop invariant $sum = \Sigma_{j=a}^{i-1} A[j]$ makes good intuitive sense. It simply says that, at the beginning of each iteration, *sum* contains the sum of all the values examined so far.

Some good guidance on the general form of the loop invariant may be obtained from *Post*, since I must satisfy I **and not** $B \Rightarrow$ *Post*, where B and *Post* are known. Indeed, it is good policy to take *Post* and generalize it in some way to obtain I. For example, in the summing algorithm above, I is just *Post* with b replaced by $i - 1$. This simple relationship ensures that the condition I **and not** $B \Rightarrow$ *Post* is readily proven.

At the other extreme, check that the initialization statements establish I. If they do, and I **and not** $B \Rightarrow$ *Post*, it is probably worthwhile to proceed with the main part of the induction.

Correctness of binary search. This section will be concluded with a study of the correctness of the following non-recursive binary search algorithm:

```
procedure BinarySearch(var A: AType; a, b: integer; x: KeyType):
boolean;
var i, j, mid: integer;
  found: boolean;
begin
  (* Pre: a ≤ b + 1 and A[a] ≤ ... ≤ A[b] *)
  i := a; j := b; found := false;
  while (i ≠ j + 1 and not found do
    mid := (i + j) div 2;
    if    x = A[mid] then found := true;
    elsif x < A[mid] then j := mid - 1;
    else                  i := mid + 1;
    end;
  end;
  (* Post: found = x ∈ A[a..b] *)

  return found;
end BinarySearch;
```

From the discussion of the recursive binary search algorithm in Section 1.2, it is fairly evident that the loop invariant should state that $x \in A[a..b]$ if and only if $x \in A[i..j]$. This takes care of the variables i and j.

The harder question is how to bring *found* and *mid* into the loop invariant, especially since *mid* is undefined at the beginning of the first iteration. Perhaps the best way to handle these two is to imagine another version of the algorithm in which the index of x is actually returned if found. For this version, $found \Rightarrow (a \le mid \le b$ **and** $x = A[mid])$ must be added to the postcondition, and this immediately suggests that it be included in the loop invariant:

$$(x \in A[a..b] \text{ if and only if } x \in A[i..j])$$
$$\textbf{and } (found \Rightarrow (a \le mid \le b \textbf{ and } x = A[mid]))$$

The initialization $i := a; j := b;$ *found* := **false**; clearly establishes this invariant. At termination the loop invariant and $i = j + 1$ or *found* both hold. If *found* is true, the loop invariant tells us that $a \le mid \le b$ **and** $x = A[mid]$, so we must have $x \in A[a..b]$; on the other hand, if *found* is false, then $i = j + 1$ and so $x \notin A[i..j]$ and therefore $x \notin A[a..b]$. Thus the postcondition holds. The rest of the proof is left as an exercise; it is quite similar to the argument used to prove that the recursive binary search algorithm was correct.

EXERCISES

1.1 Consider the following recursive algorithm:

```
procedure g(n: integer): integer;
begin
  if n ≤ 1 then
    return n;
  else
    return 5*g(n − 1) − 6*g(n − 2);
  end;
end g;
```

Prove by induction on n that $g(n)$ returns $3^n - 2^n$ for all $n \geq 0$.

1.2 Prove that the specification

```
(* Pre: a ≤ b + 1 *)
SelectionSort(A, a, b);
(* A[a] ≤ A[a + 1] ≤ . . . ≤ A[b] *)
```

is satisfied by the procedure

```
procedure SelectionSort(var A: AType; a, b: integer);
var i: integer;
begin
  if a = b + 1 then
    (* do nothing *)
  else
    i := MinIndex(A, a, b);
    if i ≠ a then
      Swap(A[i], A[a]);
    end;
    SelectionSort(A, a + 1, b);
  end;
end SelectionSort;
```

You may assume that $MinIndex(A, i, j)$ returns the index of a minimum element of the non-empty array $A[i..j]$, and that $Swap(A[i], A[j])$ swaps the two indicated elements.

1.3 The specification given for *SelectionSort* in the preceding question is satisfied by the following procedure: