Introduction To Algorithms 2nd Edition Cormen Solutions Manual

Instructor's Manual

by Thomas H. Cormen
Clara Lee
Erica Lin

to Accompany

Introduction to Algorithms

Second Edition

by Thomas H. Cormen
Charles E. Leiserson
Ronald L. Rivest
Clifford Stein

The MIT Press
Cambridge, Massachusetts London, England

McGraw-Hill Book Company
Boston Burr Ridge, IL Dubuque, IA Madison, WI
New York San Francisco St. Louis Montréal Toronto

Visit TestBankDeal.com to get complete for all chapters

Instructor's Manual
by Thomas H. Cormen, Clara Lee, and Erica Lin
to Accompany
Introduction to Algorithms, Second Edition
by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein

Published by The MIT Press and McGraw-Hill Higher Education, an imprint of The McGraw-Hill Companies, Inc., 1221 Avenue of the Americas, New York, NY 10020. Copyright © 2002 by The Massachusetts Institute of Technology and The McGraw-Hill Companies, Inc. All rights reserved.

No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written consent of The MIT Press or The McGraw-Hill Companies, Inc., including, but not limited to, network or other electronic storage or transmission, or broadcast for distance learning.

Contents

Revision History ***R-1***

Preface ***P-1***

Chapter 2: Getting Started
Lecture Notes *2-1*
Solutions *2-16*

Chapter 3: Growth of Functions
Lecture Notes *3-1*
Solutions *3-7*

Chapter 4: Recurrences
Lecture Notes *4-1*
Solutions *4-8*

Chapter 5: Probabilistic Analysis and Randomized Algorithms
Lecture Notes *5-1*
Solutions *5-8*

Chapter 6: Heapsort
Lecture Notes *6-1*
Solutions *6-10*

Chapter 7: Quicksort
Lecture Notes *7-1*
Solutions *7-9*

Chapter 8: Sorting in Linear Time
Lecture Notes *8-1*
Solutions *8-9*

Chapter 9: Medians and Order Statistics
Lecture Notes *9-1*
Solutions *9-9*

Chapter 11: Hash Tables
Lecture Notes *11-1*
Solutions *11-16*

Chapter 12: Binary Search Trees
Lecture Notes *12-1*
Solutions *12-12*

Chapter 13: Red-Black Trees
Lecture Notes *13-1*
Solutions *13-13*

Chapter 14: Augmenting Data Structures
Lecture Notes *14-1*
Solutions *14-9*

Chapter 15: Dynamic Programming
Lecture Notes *15-1*
Solutions *15-19*

Chapter 16: Greedy Algorithms
Lecture Notes *16-1*
Solutions *16-9*

Chapter 17: Amortized Analysis
Lecture Notes *17-1*
Solutions *17-14*

Chapter 21: Data Structures for Disjoint Sets
Lecture Notes *21-1*
Solutions *21-6*

Chapter 22: Elementary Graph Algorithms
Lecture Notes *22-1*
Solutions *22-12*

Chapter 23: Minimum Spanning Trees
Lecture Notes *23-1*
Solutions *23-8*

Chapter 24: Single-Source Shortest Paths
Lecture Notes *24-1*
Solutions *24-13*

Chapter 25: All-Pairs Shortest Paths
Lecture Notes *25-1*
Solutions *25-8*

Chapter 26: Maximum Flow
Lecture Notes *26-1*
Solutions *26-15*

Chapter 27: Sorting Networks
Lecture Notes *27-1*
Solutions *27-8*

Index ***I-1***

Revision History

Revisions are listed by date rather than being numbered. Because this revision history is part of each revision, the affected chapters always include the front matter in addition to those listed below.

- 18 January 2005. Corrected an error in the transpose-symmetry properties. Affected chapters: Chapter 3.
- 2 April 2004. Added solutions to Exercises 5.4-6, 11.3-5, 12.4-1, 16.4-2, 16.4-3, 21.3-4, 26.4-2, 26.4-3, and 26.4-6 and to Problems 12-3 and 17-4. Made minor changes in the solutions to Problems 11-2 and 17-2. Affected chapters: Chapters 5, 11, 12, 16, 17, 21, and 26; index.
- 7 January 2004. Corrected two minor typographical errors in the lecture notes for the expected height of a randomly built binary search tree. Affected chapters: Chapter 12.
- 23 July 2003. Updated the solution to Exercise 22.3-4(b) to adjust for a correction in the text. Affected chapters: Chapter 22; index.
- 23 June 2003. Added the link to the website for the `clrscode` package to the preface.
- 2 June 2003. Added the solution to Problem 24-6. Corrected solutions to Exercise 23.2-7 and Problem 26-4. Affected chapters: Chapters 23, 24, and 26; index.
- 20 May 2003. Added solutions to Exercises 24.4-10 and 26.1-7. Affected chapters: Chapters 24 and 26; index.
- 2 May 2003. Added solutions to Exercises 21.4-4, 21.4-5, 21.4-6, 22.1-6, and 22.3-4. Corrected a minor typographical error in the Chapter 22 notes on page 22-6. Affected chapters: Chapters 21 and 22; index.
- 28 April 2003. Added the solution to Exercise 16.1-2, corrected an error in the first adjacency matrix example in the Chapter 22 notes, and made a minor change to the accounting method analysis for dynamic tables in the Chapter 17 notes. Affected chapters: Chapters 16, 17, and 22; index.
- 10 April 2003. Corrected an error in the solution to Exercise 11.3-3. Affected chapters: Chapter 11.
- 3 April 2003. Reversed the order of Exercises 14.2-3 and 14.3-3. Affected chapters: Chapter 13, index.
- 2 April 2003. Corrected an error in the substitution method for recurrences on page 4-4. Affected chapters: Chapter 4.

- 31 March 2003. Corrected a minor typographical error in the Chapter 8 notes on page 8-3. Affected chapters: Chapter 8.
- 14 January 2003. Changed the exposition of indicator random variables in the Chapter 5 notes to correct for an error in the text. Affected pages: 5-4 through 5-6. (The only content changes are on page 5-4; in pages 5-5 and 5-6 only pagination changes.) Affected chapters: Chapter 5.
- 14 January 2003. Corrected an error in the pseudocode for the solution to Exercise 2.2-2 on page 2-16. Affected chapters: Chapter 2.
- 7 October 2002. Corrected a typographical error in EUCLIDEAN-TSP on page 15-23. Affected chapters: Chapter 15.
- 1 August 2002. Initial release.

Preface

This document is an instructor's manual to accompany *Introduction to Algorithms*, Second Edition, by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. It is intended for use in a course on algorithms. You might also find some of the material herein to be useful for a CS 2-style course in data structures.

Unlike the instructor's manual for the first edition of the text—which was organized around the undergraduate algorithms course taught by Charles Leiserson at MIT in Spring 1991—we have chosen to organize the manual for the second edition according to chapters of the text. That is, for most chapters we have provided a set of lecture notes and a set of exercise and problem solutions pertaining to the chapter. This organization allows you to decide how to best use the material in the manual in your own course.

We have not included lecture notes and solutions for every chapter, nor have we included solutions for every exercise and problem within the chapters that we have selected. We felt that Chapter 1 is too nontechnical to include here, and Chapter 10 consists of background material that often falls outside algorithms and data-structures courses. We have also omitted the chapters that are not covered in the courses that we teach: Chapters 18–20 and 28–35, as well as Appendices A–C; future editions of this manual may include some of these chapters. There are two reasons that we have not included solutions to all exercises and problems in the selected chapters. First, writing up all these solutions would take a long time, and we felt it more important to release this manual in as timely a fashion as possible. Second, if we were to include all solutions, this manual would be longer than the text itself!

We have numbered the pages in this manual using the format *CC-PP*, where *CC* is a chapter number of the text and *PP* is the page number within that chapter's lecture notes and solutions. The *PP* numbers restart from 1 at the beginning of each chapter's lecture notes. We chose this form of page numbering so that if we add or change solutions to exercises and problems, the only pages whose numbering is affected are those for the solutions for that chapter. Moreover, if we add material for currently uncovered chapters, the numbers of the existing pages will remain unchanged.

The lecture notes

The lecture notes are based on three sources:

- Some are from the first-edition manual, and so they correspond to Charles Leiserson's lectures in MIT's undergraduate algorithms course, 6.046.
- Some are from Tom Cormen's lectures in Dartmouth College's undergraduate algorithms course, CS 25.
- Some are written just for this manual.

You will find that the lecture notes are more informal than the text, as is appropriate for a lecture situation. In some places, we have simplified the material for lecture presentation or even omitted certain considerations. Some sections of the text—usually starred—are omitted from the lecture notes. (We have included lecture notes for one starred section: 12.4, on randomly built binary search trees, which we cover in an optional CS 25 lecture.)

In several places in the lecture notes, we have included "asides" to the instructor. The asides are typeset in a slanted font and are enclosed in square brackets. *[Here is an aside.]* Some of the asides suggest leaving certain material on the board, since you will be coming back to it later. If you are projecting a presentation rather than writing on a blackboard or whiteboard, you might want to mark slides containing this material so that you can easily come back to them later in the lecture.

We have chosen not to indicate how long it takes to cover material, as the time necessary to cover a topic depends on the instructor, the students, the class schedule, and other variables.

There are two differences in how we write pseudocode in the lecture notes and the text:

- Lines are not numbered in the lecture notes. We find them inconvenient to number when writing pseudocode on the board.
- We avoid using the *length* attribute of an array. Instead, we pass the array length as a parameter to the procedure. This change makes the pseudocode more concise, as well as matching better with the description of what it does.

We have also minimized the use of shading in figures within lecture notes, since drawing a figure with shading on a blackboard or whiteboard is difficult.

The solutions

The solutions are based on the same sources as the lecture notes. They are written a bit more formally than the lecture notes, though a bit less formally than the text. We do not number lines of pseudocode, but we do use the *length* attribute (on the assumption that you will want your students to write pseudocode as it appears in the text).

The index lists all the exercises and problems for which this manual provides solutions, along with the number of the page on which each solution starts.

Asides appear in a handful of places throughout the solutions. Also, we are less reluctant to use shading in figures within solutions, since these figures are more likely to be reproduced than to be drawn on a board.

Source files

For several reasons, we are unable to publish or transmit source files for this manual. We apologize for this inconvenience.

In June 2003, we made available a `clrscode` package for LaTeX 2_ε. It enables you to typeset pseudocode in the same way that we do. You can find this package at `http://www.cs.dartmouth.edu/~thc/clrscode/`. That site also includes documentation.

Reporting errors and suggestions

Undoubtedly, instructors will find errors in this manual. Please report errors by sending email to `clrs-manual-bugs@mhhe.com`

If you have a suggestion for an improvement to this manual, please feel free to submit it via email to `clrs-manual-suggestions@mhhe.com`

As usual, if you find an error in the text itself, please verify that it has not already been posted on the errata web page before you submit it. You can use the MIT Press web site for the text, `http://mitpress.mit.edu/algorithms/`, to locate the errata web page and to submit an error report.

We thank you in advance for your assistance in correcting errors in both this manual and the text.

Acknowledgments

This manual borrows heavily from the first-edition manual, which was written by Julie Sussman, P.P.A. Julie did such a superb job on the first-edition manual, finding numerous errors in the first-edition text in the process, that we were thrilled to have her serve as technical copyeditor for the second-edition text. Charles Leiserson also put in large amounts of time working with Julie on the first-edition manual.

The other three *Introduction to Algorithms* authors—Charles Leiserson, Ron Rivest, and Cliff Stein—provided helpful comments and suggestions for solutions to exercises and problems. Some of the solutions are modifications of those written over the years by teaching assistants for algorithms courses at MIT and Dartmouth. At this point, we do not know which TAs wrote which solutions, and so we simply thank them collectively.

We also thank McGraw-Hill and our editors, Betsy Jones and Melinda Dougharty, for moral and financial support. Thanks also to our MIT Press editor, Bob Prior, and to David Jones of The MIT Press for help with TeX macros. Wayne Cripps, John Konkle, and Tim Tregubov provided computer support at Dartmouth, and the MIT sysadmins were Greg Shomo and Matt McKinnon. Phillip Meek of McGraw-Hill helped us hook this manual into their web site.

THOMAS H. CORMEN
CLARA LEE
ERICA LIN
Hanover, New Hampshire
July 2002

Lecture Notes for Chapter 2: Getting Started

Chapter 2 overview

Goals:

- Start using frameworks for describing and analyzing algorithms.
- Examine two algorithms for sorting: insertion sort and merge sort.
- See how to describe algorithms in pseudocode.
- Begin using asymptotic notation to express running-time analysis.
- Learn the technique of "divide and conquer" in the context of merge sort.

Insertion sort

The sorting problem

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \le a'_2 \le \cdots \le a'_n$.

The sequences are typically stored in arrays.

We also refer to the numbers as ***keys***. Along with each key may be additional information, known as ***satellite data***. *[You might want to clarify that "satellite data" does not necessarily come from a satellite!]*

We will see several ways to solve the sorting problem. Each way will be expressed as an ***algorithm***: a well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

Expressing algorithms

We express algorithms in whatever way is the clearest and most concise.

English is sometimes the best way.

When issues of control need to be made perfectly clear, we often use ***pseudocode***.

- Pseudocode is similar to C, C++, Pascal, and Java. If you know any of these languages, you should be able to understand pseudocode.
- Pseudocode is designed for *expressing algorithms to humans*. Software engineering issues of data abstraction, modularity, and error handling are often ignored.
- We sometimes embed English statements into pseudocode. Therefore, unlike for "real" programming languages, we cannot create a compiler that translates pseudocode to machine code.

Insertion sort

A good algorithm for sorting a small number of elements.

It works the way you might sort a hand of playing cards:

- Start with an empty left hand and the cards face down on the table.
- Then remove one card at a time from the table, and insert it into the correct position in the left hand.
- To find the correct position for a card, compare it with each of the cards already in the hand, from right to left.
- At all times, the cards held in the left hand are sorted, and these cards were originally the top cards of the pile on the table.

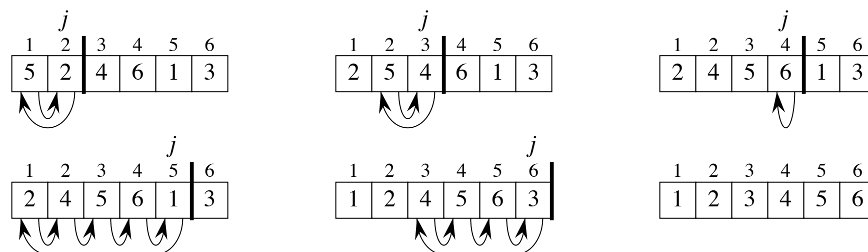
Pseudocode: We use a procedure INSERTION-SORT.

- Takes as parameters an array $A[1 \, . \, . \, n]$ and the length n of the array.
- As in Pascal, we use ". ." to denote a range within an array.
- *[We usually use 1-origin indexing, as we do here. There are a few places in later chapters where we use 0-origin indexing instead. If you are translating pseudocode to C, C++, or Java, which use 0-origin indexing, you need to be careful to get the indices right. One option is to adjust all index calculations in the C, C++, or Java code to compensate. An easier option is, when using an array $A[1 \, . \, . \, n]$, to allocate the array to be one entry longer—$A[0 \, . \, . \, n]$—and just don't use the entry at index 0.]*
- *[In the lecture notes, we indicate array lengths by parameters rather than by using the length attribute that is used in the book. That saves us a line of pseudocode each time. The solutions continue to use the length attribute.]*
- The array A is sorted ***in place***: the numbers are rearranged within the array, with at most a constant number outside the array at any time.

INSERTION-SORT(A)	*cost*	*times*
for $j \leftarrow 2$ **to** n	c_1	n
do $key \leftarrow A[j]$	c_2	$n-1$
$\triangleright$ Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
$i \leftarrow j-1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
do $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
$A[i+1] \leftarrow key$	c_8	$n-1$

[Leave this on the board, but show only the pseudocode for now. We'll put in the "cost" and "times" columns later.]

Example:



[Read this figure row by row. Each part shows what happens for a particular iteration with the value of j indicated. j indexes the "current card" being inserted into the hand. Elements to the left of $A[j]$ that are greater than $A[j]$ move one position to the right, and $A[j]$ moves into the evacuated position. The heavy vertical lines separate the part of the array in which an iteration works—$A[1\,..\,j]$—from the part of the array that is unaffected by this iteration—$A[j+1\,..\,n]$. The last part of the figure shows the final sorted array.]

Correctness

We often use a ***loop invariant*** to help us understand why an algorithm gives the correct answer. Here's the loop invariant for INSERTION-SORT:

> **Loop invariant:** At the start of each iteration of the "outer" **for** loop—the loop indexed by j—the subarray $A[1\,..\,j-1]$ consists of the elements originally in $A[1\,..\,j-1]$ but in sorted order.

To use a loop invariant to prove correctness, we must show three things about it:

Initialization: It is true prior to the first iteration of the loop.

Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.

Termination: When the loop terminates, the invariant—usually along with the reason that the loop terminated—gives us a useful property that helps show that the algorithm is correct.

Using loop invariants is like mathematical induction:

- To prove that a property holds, you prove a base case and an inductive step.
- Showing that the invariant holds before the first iteration is like the base case.
- Showing that the invariant holds from iteration to iteration is like the inductive step.
- The termination part differs from the usual use of mathematical induction, in which the inductive step is used infinitely. We stop the "induction" when the loop terminates.
- We can show the three parts in any order.

For insertion sort:

Initialization: Just before the first iteration, $j = 2$. The subarray $A[1 \,..\, j-1]$ is the single element $A[1]$, which is the element originally in $A[1]$, and it is trivially sorted.

Maintenance: To be precise, we would need to state and prove a loop invariant for the "inner" **while** loop. Rather than getting bogged down in another loop invariant, we instead note that the body of the inner **while** loop works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$, and so on, by one position to the right until the proper position for *key* (which has the value that started out in $A[j]$) is found. At that point, the value of *key* is placed into this position.

Termination: The outer **for** loop ends when $j > n$; this occurs when $j = n+1$. Therefore, $j-1 = n$. Plugging n in for $j-1$ in the loop invariant, the subarray $A[1 \,..\, n]$ consists of the elements originally in $A[1 \,..\, n]$ but in sorted order. In other words, the entire array is sorted!

Pseudocode conventions

[Covering most, but not all, here. See book pages 19–20 for all conventions.]

- Indentation indicates block structure. Saves space and writing time.
- Looping constructs are like in C, C++, Pascal, and Java. We assume that the loop variable in a **for** loop is still defined when the loop exits (unlike in Pascal).
- "$\triangleright$" indicates that the remainder of the line is a comment.
- Variables are local, unless otherwise specified.
- We often use ***objects***, which have ***attributes*** (equivalently, ***fields***). For an attribute *attr* of object x, we write *attr*$[x]$. (This would be the equivalent of x. *attr* in Java or x -> *attr* in C++.)
- Objects are treated as references, like in Java. If x and y denote objects, then the assignment $y \leftarrow x$ makes x and y reference the same object. It does not cause attributes of one object to be copied to another.
- Parameters are passed by value, as in Java and C (and the default mechanism in Pascal and C++). When an object is passed by value, it is actually a reference (or pointer) that is passed; changes to the reference itself are not seen by the caller, but changes to the object's attributes are.
- The boolean operators "and" and "or" are ***short-circuiting***: if after evaluating the left-hand operand, we know the result of the expression, then we don't evaluate the right-hand operand. (If x is FALSE in "x and y" then we don't evaluate y. If x is TRUE in "x or y" then we don't evaluate y.)

Introduction To Algorithms 2nd Edition Cormen Solutions Manual

Analyzing algorithms

We want to predict the resources that the algorithm requires. Usually, running time. In order to predict resource requirements, we need a computational model.

Random-access machine (RAM) model

- Instructions are executed one after another. No concurrent operations.
- It's too tedious to define each of the instructions and their associated time costs.
- Instead, we recognize that we'll use instructions commonly found in real computers:
  - Arithmetic: add, subtract, multiply, divide, remainder, floor, ceiling). Also, shift left/shift right (good for multiplying/dividing by 2^k).
  - Data movement: load, store, copy.
  - Control: conditional/unconditional branch, subroutine call and return.

  Each of these instructions takes a constant amount of time.

The RAM model uses integer and floating-point types.

- We don't worry about precision, although it is crucial in certain numerical applications.
- There is a limit on the word size: when working with inputs of size n, assume that integers are represented by $c \lg n$ bits for some constant $c \geq 1$. ($\lg n$ is a very frequently used shorthand for $\log_2 n$.)
  - $c \geq 1 \Rightarrow$ we can hold the value of $n \Rightarrow$ we can index the individual elements.
  - c is a constant $\Rightarrow$ the word size cannot grow arbitrarily.

How do we analyze an algorithm's running time?

The time taken by an algorithm depends on the input.

- Sorting 1000 numbers takes longer than sorting 3 numbers.
- A given sorting algorithm may even take differing amounts of time on two inputs of the same size.
- For example, we'll see that insertion sort takes less time to sort n elements when they are already sorted than when they are in reverse sorted order.

Input size: Depends on the problem being studied.

- Usually, the number of items in the input. Like the size n of the array being sorted.
- But could be something else. If multiplying two integers, could be the total number of bits in the two integers.
- Could be described by more than one number. For example, graph algorithm running times are usually expressed in terms of the number of vertices and the number of edges in the input graph.

Visit TestBankDeal.com to get complete for all chapters