算法题解

(英文版)

Exercises & Solutions on Algorithms

霍红卫　编著

高等教育出版社
Higher Education Press

算 法 题 解

（英文版）

Exercises & Solutions on Algorithms

霍红卫　编著

高 等 教 育 出 版 社
HIGHER EDUCATION PRESS

内 容 简 介

本书提供了在学习现代计算机算法时经常会遇到的许多问题的答案，可以帮助读者更好地理解、掌握算法分析与设计课程。本书包括 360 多个练习题，这些练习题不仅涉及到一些经典问题，而且包括一些重要的应用程序中的热点问题，如文字处理中的段落排版、数据压缩、数据库系统和 Internet 搜索引擎等方面的算法问题，这些都是现代软件系统的基本组成部分。

本书可作为高等学校计算机科学与技术类专业、数学及信息与计算科学专业本科生和研究生“算法分析与设计”课程的辅助教材，也可供其他专业涉及算法设计与应用的研究和开发人员学习参考。

图书在版编目（CIP）数据

算法题解：英文版 / 霍红卫编著. 一北京：高等教育出版社，2004.7
ISBN 7－04－014619－3

Ⅰ.算... Ⅱ.霍... Ⅲ.电子计算机一算法设计一高等学校一解题一英文 Ⅳ.TP301.6

中国版本图书馆CIP数据核字（2004）第060855号

策划编辑 刘建元 **责任编辑** 郭福生 **封面设计** 张 楠
市场策划 陈 振 **责任印制** 韩 刚

出版发行	高等教育出版社	**购书热线**	010-64054588
社　　址	北京市西城区德外大街 4 号	**免费咨询**	800-810-0598
邮政编码	100011	**网　　址**	http://www.hep.edu.cn
总　　机	010-82028899		http://www.hep.com.cn

经　　销 新华书店北京发行所
印　　刷 北京市鑫霸印务有限公司

开　　本	787×1092　1/16	**版　　次**	2004 年 7月第 1 版
印　　张	24.5	**印　　次**	2004 年 7月第 1 次印刷
字　　数	510 000	**定　　价**	35.00 元

Foreword

Computer hardware technologies will be limited by the speed of electron motion, although the performances of microprocessors have been continuously improved as the development of microelectronics and computer architecture technology for the last 30 years. We expect that future improvements are not so much in the speed of computer devices as in the complexity of computation, which is determined by algorithms of computation. So further improvements in algorithms will offer a possible route to continue to make computers faster. Therefore better exploitation of parallel operations, pre-computation of parts of a problem and heuristic technologies are viable ways of increasing computing efficiency.

The study of algorithms is an essential part of computer science. Significant advances in this field have been made. Meanwhile, the applied realm of algorithms is being enlarged continuously. A large group of recognized algorithms have become essential parts of modern software systems.

As an outstanding and significant work in this book, the solutions to over 360 exercises will help the readers understand the main principles and concepts of algorithms design courses. This book is especially valuable for the readers who are beginning to learn algorithm or wishing to have up-to-date reference material at hand.

The research and design of algorithms is a field that is full of challenge and excitement. I hope that this book could bring more researchers, students and engineers envolving in both computer hardware and software to the competition in this field.

Member of the Chinese Academy of Sciences

Shen XuBang

May 2004

Preface

This book provides solutions to many problems encountered in the modern study of computer algorithms. It helps us better understand the course design and analysis of algorithms. The first section in each chapter provides some relative algorithms, which are described in pseudo-code designed to be readable by anyone who has done some programming. And the last section in each chapter provides answers to the exercises. This book includes over 360 exercises, some exercises test basic concept mastery of the chapter and others are about more complicated problems. The exercises involve not only in classical problems, but also the interesting and hotspot problems in some important applications. From typesetting a paragraph in word processing to data compression, from database systems to the Internet search engines, all have become essential parts of modern software systems.

The exercises in this book relate to the problems on elementary data structures, dynamic sets and searching, B-trees, binomial heaps, Fibonacci heaps, data structures for disjoint sets, divide and conquer, dynamic programming, greedy, amortized analysis, DFS, BFS, topological sort, strongly connected components, minimum spanning tree, single-source shortest paths, all-pairs shortest paths, maximum flow, etc.

The study of algorithms is at the very heart of computer science. During the last 50 years, a certain amount of significant advances in the field of algorithms have been made. This book aims to serve as a supplement for students and professionals interested in knowing and making intelligent use of some fundamental algorithms to a broad variety of applications. This book can be used as complementary material for graduate and undergraduate courses in the design and analysis of algorithms. It will also provide useful background and reference information for those who are working with ACM programming contest.

If any reader finds errors or omissions in this book, please inform the author (preferably by electronic mail). All constructive criticism and suggestions for answers to more exercises are most welcome.

Acknowledgements. Many people provided me helpful references and materials. Also, some students of the spring graduate and fall undergraduate courses on design and analysis of algorithms (http://www.lib.xidian.edu.cn/newindex/huohongwei/kejian.htm) at Xidian University provided me with helpful suggestions. I have gained two mentors' support in writing this book, and particularly want to express my appreciation to them. They are Academician Shen Xubang,

Chinese Academy of Science and Professor Han Jungang, Dean of the Department of Computer Science and Technology, Xi'an University of Posts and Telecommunications. It has been a pleasure working with Higher Education Press in the development of this book. I especially thank Liu Jianyuan, Ni Wenhui, and Guo Fusheng for their support and patience. Finally, I thank my husband and my son for their love and support during the writing of this book.

Huo Hongwei
School of Computer Science
Xidian University
May 2004

Contents

Chapter 1

Mathematical Foundation

An algorithm is any well-defined computational procedure that takes some values as input and produces some values as output. To characterize an algorithm's efficiency, we introduce asymptotic notation to describe the asymptotic running time of an algorithm. The notation is a function of the input size of one's problem. We are concerned with how the running time of an algorithm increases with the size of the input, as the size of the input increases without bound. We will study methods for developing efficient algorithms and make mathematical comparison of algorithms (without actually implementing them).

The algorithms we discuss will be in "pseudo-code", so we will not worry about certain details of implementation.

1.1 Growth of Functions

1.1.1 O-notation (Big-O)

$$\mathrm{O}(g(n)) = \{f(n) : \exists c,\, n_0 > 0 \text{ such that } f(n) \le cg(n)\ \forall\, n \ge n_0\}.$$

We use O-notation to give an upper bound on a function and bound the worst-case running time.

Examples:

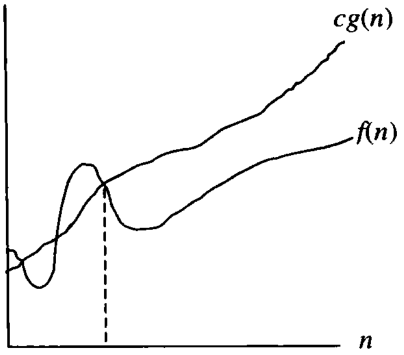
1. $\frac{1}{3}n^2 - 3n \in \mathrm{O}(n^2)$ because $\frac{1}{3}n^2 - 3n \le cn^2$ if $c \ge \frac{1}{3} - \frac{3}{n}$ which holds for $c = \frac{1}{3}$ and $n > 1$.
2. $k_1n^2 + k_2n + k_3 \in \mathrm{O}(n^2)$ because $k_1n^2 + k_2n + k_3 \le (\, k_1 + |k_2| + |k_3|\,)n^2$ and for $c > k_1 + |k_2| + |k_3|$ and $n \ge 1$, $k_1n^2 + k_2n + k_3 \le cn^2$.
3. $k_1n^2 + k_2n + k_3 \in \mathrm{O}(n^3)$ as $k_1n^2 + k_2n + k_3 \le (\, k_1 + |k_2| + |k_3|\,)n^3$. (upper bound)

Note:

- When we say "the running time is $\mathrm{O}(n^2)$" we mean that the worst-case running time is

$O(n^2)$ — the best case might be better.

- Use of O-notation often makes it much easier to analyze algorithms; we can easily prove the $O(n^2)$ insertion-sort time bound.
- We Sometime abuse the notation, for instance:
  - Write $f(n) = O(g(n))$ instead of $f(n) \in O(g(n))$.
  - Use $O(n)$ in equations: e.g. $2n^2 + 3n + 1 = 2n^2 + O(n)$ (meaning that $2n^2 + 3n + 1 = 2n^2 + f(n)$ where $f(n)$ is some function in $O(n)$).
  - Use $O(1)$ to denote a constant time.



$f(n) = O(g(n))$

1.1.2 Ω-notation (Big-Omega)

$$\Omega(g(n)) = \{f(n) : \exists c, n_0 > 0 \text{ such that } cg(n) \le f(n)\ \forall\ n \ge n_0\}$$

We use Ω-notation to give a lower bound on a function.

Examples:

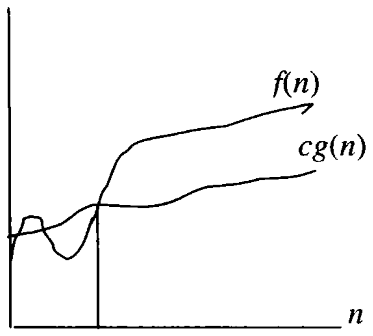
1. $\frac{1}{3}n^2 - 3n = \Omega(n^2)$ because $\frac{1}{3}n^2 - 3n \ge cn^2$ if $c \le 1/3 - 3/n$ which is true if $c = 1/6$ and $n > 18$.
2. $k_1n^2 + k_2n + k_3 = \Omega(n^2)$.
3. $k_1n^2 + k_2n + k_3 = \Omega(n)$.　(lower bound)

Note:

- When saying "the running time is $\Omega(n^2)$" we mean that the best-case running time is

$\Omega(n^2)$—the worst case might be worse.

- Insertion-sort:
  - Best-case: $\Omega(n)$—when the input array is already sorted.
  - Worst-case: $O(n^2)$—when the input array is reverse sorted.
  - We can also say that the worst case running time is $\Omega(n^2)$.



n_0 $f(n) = \Omega(g(n))$

1.1.3 Θ-notation (Big-Theta)

$$\Theta(g(n)) = \{f(n) : \exists c_1, c_2, n_0 > 0 \text{ such that } c_1 g(n) \le f(n) \le c_2 g(n)\ \forall\ n \ge n_0\}$$

We use Θ-notation to give a tight bound on a function.

$$f(n) = \Theta(g(n)) \text{ if and only if } f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)).$$

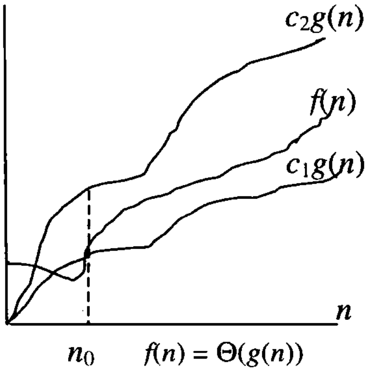
Examples:

1. $k_1 n^2 + k_2 n + k_3 = \Theta(n^2)$.
2. The worst-case running time of insertion-sort is $\Theta(n^2)$.
3. $6n \lg n + \sqrt{n} \lg^2 n = \Theta(n \lg n)$.

We need to find $c_1, c_2, n_0 > 0$ such that $c_1 n \lg n \le 6n \lg n + \sqrt{n} \lg^2 n \le c_2 n \lg n$ for $n \ge n_0$.

$c_1 n \lg n \le 6n \lg n + \sqrt{n} \lg^2 n \Rightarrow c_1 \le 6 + \lg n / \sqrt{n}$, which is true if we choose $c_1 = 6$ and $n_0 = 1$.

$6n \lg n + \sqrt{n} \lg^2 n \le c_2 n \Rightarrow 6 + \lg n / \sqrt{n} \le c_2$, which is true if we choose $c_2 = 7$ and $n_0 = 2$. This is because $\lg n \le \sqrt{n}$ if $n \ge 2$. So $c_1 = 6$, $c_2 = 7$ and $n_0 = 2$ works.



1.2 Recurrences

To solve a problem P, divide and conquer paradigm involves three steps:

1. Divide P into smaller sub problems $P_1, P_2, \ldots, P_k$.
2. Conquer by solving the smaller subproblems recursively.
3. Combine solutions to $P_1, P_2, \ldots, P_k$ into the solution for original P.

The analysis of divide-and-conquer algorithms leads to recurrences. Mergesort has the recurrence $T(n) = 2\mathrm{T}(n/2) + n$. There are four methods for solving recurrence—that is, substitution method, iteration method, recursion-tree method and master method.

1.2.1 Substitution Method

The *substitution method* for solving recurrences consists of two steps. First, guess the form of solution. Then use mathematical induction to find the constants and show that the solution works.

Example: Solve $T(n) = 2T(n/2) + n$ using substitution method.
Proof: We guess $T(n) \le cn \lg n$ for some constant c (that is, $T(n) = O(n \lg n)$).
Base case: Function constant for small constant n.
Inductive step: Assume that it holds for $n/2$: $T(n/2) \le cn/2 \lg n/2$. We try to show that holds $T(n) \le cn \lg n$ for an appropriate choice of the constant c. Substituting our guess in the

recurrence, we have

$$\begin{aligned}T(n) &= 2T(n/2) + n\\ &\le 2(cn/2 \lg n/2) + n\\ &= cn \lg n/2 + n\\ &= cn \lg n - cn \lg 2 + n\\ &= cn \lg n - cn + n\end{aligned}$$

where the last step holds as long as $c \ge 1$.
The hard part of the substitution method is often requires to make a good guess.

1.2.2 Iteration Method

In the *iteration method* we iteratively expand the recurrence until we "see the pattern". The iteration method does not require to make a good guess like the substitution method (but it is often more involved than using induction).

Example: Solve $T(n) = 8T(n/2) + n^2$ $(T(1) = 1)$

$$\begin{aligned}T(n) &= n^2 + 8T(n/2)\\ &= n^2 + 8(8T(n/2^2) + (n/2)^2)\\ &= n^2 + 8^2T(n/2^2) + 8(n^2/4)\\ &= n^2 + 2n^2 + 8^2T(n/2^2)\\ &= n^2 + 2n^2 + 8^2(8T(n/2^3) + (n/2^2)^2)\\ &= n^2 + 2n^2 + 8^3T(n/2^3) + 8^2(n^2/4^2))\\ &= n^2 + 2n^2 + 2^2n^2 + 8^3T(n/2^3)\\ &= \ldots\\ &= n^2 + 2n^2 + 2^2n^2 + 2^3n^2 + 2^4n^2 + \ldots\end{aligned}$$

How long does it continue? i times where $n/2^i = 1 \Rightarrow i = \lg n$.
What is the last term? $8^iT(1) = 8^{\lg n}$.

$$\begin{aligned}T(n) &= n^2 + 2n^2 + 2^2n^2 + 2^3n^2 + 2^4n^2 + \ldots + 2^{\lg n - 1}n^2 + 8^{\lg n}\\ &= \textstyle\sum_{k=0..\lg n-1} 2^k n^2 + 8^{\lg n}\\ &= n^2 \textstyle\sum_{k=0..\lg n-1} 2^k + (2^3)^{\lg n}\end{aligned}$$

Now $\sum_{k=0..\lg n-1} 2^k$ is a geometric sum so we have $\sum_{k=0..\lg n-1} 2^k = \Theta(2^{\lg n - 1}) = \Theta(n)$.

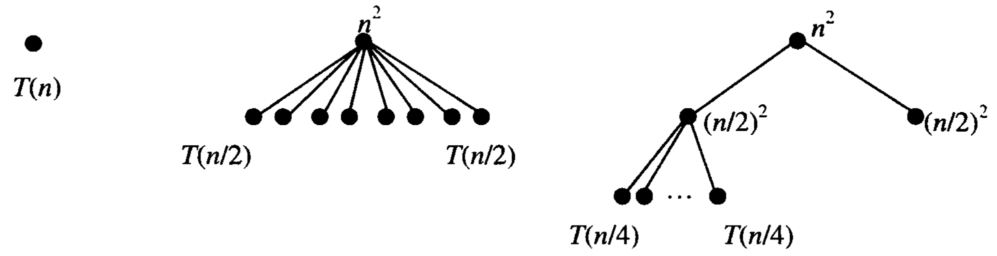
$$(2^3)^{\lg n} = (2^{\lg n})^3 = n^3$$

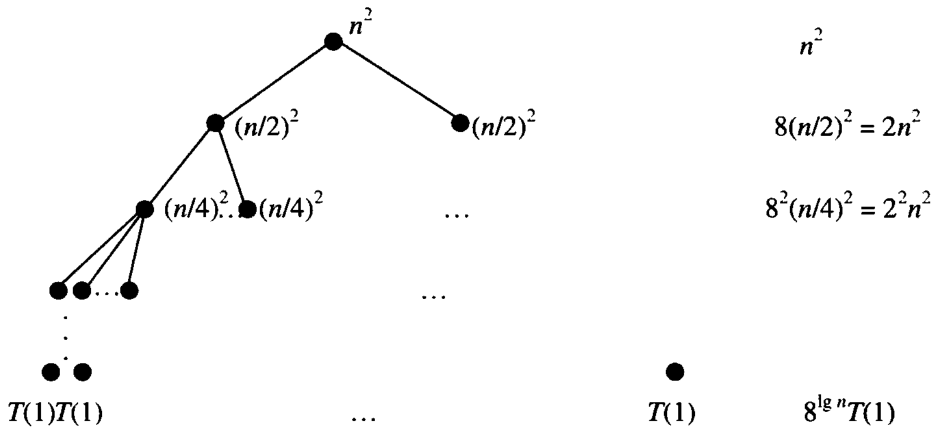
$$\begin{aligned}T(n) &= n^2 \cdot \Theta(n) + n^3\\ &= \Theta(n^3)\end{aligned}$$

1.2.3 Recursion-tree Method

In the *recursion-tree method*, each node in the tree represents the cost of a single subproblem somewhere in the set of recursive function invocation. We sum the costs within each level of the tree to obtain a set of per-level costs, and then we sum all the per-level costs to determine the total cost of all levels of the recursion.

Example: Solve $T(n) = 8T(n/2) + n^2$ $(T(1) = 1)$





$$
\begin{aligned}
T(n) &= n^2 + 2n^2 + 2^2n^2 + 2^3n^2 + 2^4n^2 + \ldots + 2^{\lg n - 1} n^2 + 8^{\lg n} \\
&= \textstyle\sum_{k=0..\lg n - 1} 2^k n^2 + 8^{\lg n} \\
&= n^2 \textstyle\sum_{k=0..\lg n - 1} 2^k + (2^3)^{\lg n} \\
&= n^2 \cdot \Theta(n) + n^3 \\
&= \Theta(n^3)
\end{aligned}
$$

1.2.4 Master Method

We have solved several recurrences using substitution, iteration and recursion-tree. in the form of $T(n) = aT(n/b) + n^c$. $(T(1) = 1)$.

The solution to the recurrence $T(n) = 2T(n/2) + n$ is $O(n \lg n)$ ($a = 2$, $b = 2$, and $c = 1$) .

The solution to the recurrence $T(n) = 8T(n/2) + n^2$ is $\Theta(n^3)$ ($a = 8$, $b = 2$, and $c = 2$) .

It would be nice to have a general solution to the recurrence $T(n) = aT(n/b) + n^c$. Yes, we do.

$$T(n) = aT(n/b) + n^c \quad a \geq 1, b \geq 1, c > 0.$$

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^c, \\ \Theta(n^c \log_b n) & \text{if } a = b^c, \\ \Theta(n^c) & \text{if } a < b^c. \end{cases}$$

1.2.5 Other Recurrences

Some important/typical bounds on recurrences have not been covered by the master method:

- Logarithmic: $\Theta(\lg n)$
  Recurrence: $T(n) = 1 + T(n/2)$
  Typical example: recurse on a half input (and throw half away)
  Variations: $T(n) = 1 + T(99n/100)$
- Linear: $\Theta(n)$
  Recurrence: $T(n) = 1 + T(n-1)$
  Typical example: single loop
  Variations: $T(n) = 1 + 2T(n/2)$, $T(n) = n + T(n/2)$, $T(n) = T(n/5) + T(7n/10 + 6) + n$
- Quadratic: $\Theta(n^2)$
  Recurrence: $T(n) = n + T(n-1)$
  Typical example: nested loops
- Exponential: $\Theta(2^n)$
  Recurrence: $T(n) = 2T(n-1)$

1.3 Exercises & Solutions

Exercise 1-1: Give two functions f_1 and f_2 so that the following equation is true for f_1, and false for f_2.

$$\Sigma_{i=1..n} f(i) = \Theta(f(n)).$$

Prove your answers.

Solution: $\Sigma_{i=1..n} 2^i = 2^{n+1} - 2 = \Theta(2^n)$. $\Sigma_{i=1..n} i = n(n+1)/2 = \Theta(n^2) \neq \Theta(n)$.

Exercise 1-2: Prove that

$$\lg(n!) = \Theta(n \lg n).$$

without appealing to Stirling's approximation of the factorial function.

Solution: Since $\lg(n!) = \Sigma_{i=1..n} \lg i \le \Sigma_{i=1..n} \lg n = n \lg n$, it is immediate that $\lg(n!) = O(n \lg n)$. It remains to be shown that $\lg(n!) = \Omega(n \lg n)$. We assume that the base of log is 2, and show that $\lg(n!) \ge n (\lg n)/4$ for all evens $n \ge 4$. For even n, we have $\lg(n!) = \Sigma_{i=1..n} \lg i \ge \Sigma_{i=n/2..n} \lg i \ge \Sigma_{i=n/2..n} \lg n/2 \ge n/2 (\lg n/2) = (n \lg n)/2 - n/2$. For $n \ge 4$, we have $(n \lg n)/2 - n/2 \ge (n \lg n)/4$.

Exercise 1-3: Consider the following problem: given an array $A[1..n]$ of distinct integers, and a number $1 \le k \le n$, find any one of the k largest elements in A. For example, if $k = 2$, it is ok to return the largest or second largest integer in A, without knowing if the return value is the largest or if it is the second largest array element.

(a) Give an algorithm that solves this problem using no more that $n - k$ comparisons of array elements.

(b) Argue that every algorithm that solves this problem must, in the worst case, perform at least $n-k$ comparisons.

Solution:

(a) Find-one of the k largest

```
FIND-ONE-K-LARGEST( )
1  $x \rightarrow A[1]$
2  for $i = 2$ to $n - k + 1$
3      do if $A[i] > x$
4          then $x \rightarrow A[i]$
5  return $x$
```

(b) Suppose an algorithm performs only $n - k - 1$ comparisons. Then at the end, there are at least