算法基础

附：算法基础自学考试大纲

组编／上海市高等教育自学考试办公室
主编／夏宽理

上海市高等教育自学考试指定教材　计算机软件专业（专科本科段）

社

上海市高等教育自学考试指定教材
计算机软件专业(独立本科段)

算 法 基 础

(附:算法基础自学考试大纲)

上海市高等教育自学考试办公室 组编

夏宽理 主编

高等教育出版社

内容简介

本书系统地介绍了常用算法和算法设计方法：排序算法、集合运算和检索算法、基于图的算法、字符串匹配算法、常用算法设计技术和索引技术。全书采用C语言作为数据结构和算法的描述语言。本书的重点是介绍算法的设计过程，注重算法设计能力的培养。全书内容丰富、概念清晰、逻辑性强、通俗易懂，既便于教学，也适宜于自学。

本书可作为计算机相关专业独立本科段自考教材，亦可作为全日制高等院校计算机类相关专业的本科教材或教学参考书。

图书在版编目(CIP)数据

算法基础/夏宽理主编．—北京：高等教育出版社，2003.10

ISBN 7-04-013760-7

Ⅰ.算… Ⅱ.夏… Ⅲ.电子计算机—算法理论—高等学校—教材 Ⅳ.TP301.6

中国版本图书馆CIP数据核字(2003)第084434号

责任编辑 司马镭 **封面设计** 吴 昊 **责任印制** 蔡敏燕

书　　名 算法基础
主　　编 夏宽理

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		021-56964871
邮政编码	100011	**免费咨询**	800-810-0598
总　　机	010-82028899	**网　　址**	http://www.hep.edu.cn
传　　真	021-56965341		http://www.hep.com.cn
			http://www.hepsh.com

排版校对 南京展望照排印刷有限公司
印　　刷 江苏如皋市印刷有限公司

开　　本	787×1092　1/16	**版　　次**	2003年10月第1版
印　　张	15	**印　　次**	2003年10月第1次
字　　数	356 000	**定　　价**	21.00元

凡购买高等教育出版社图书，如有缺页、倒页、脱页等质量问题，请在所购图书销售部门联系调换。

郑 重 声 明

编者的话

随着计算机科学技术的迅猛发展，以及计算机应用的日益广泛，对计算机应用软件开发人员的需求也与日俱增。为了尽快和多渠道培养应用软件开发人员，开设自学考试软件专业（独立本科段）是势在必行之举。

算法设计技术是软件核心技术之一，是计算机软件开发人员必须掌握的知识和技能。

本书是根据上海市高等教育自学考试软件专业（独立本科段）的“算法基础”课程自学考试大纲编写的教材，所选内容符合大纲的要求。全书共8章：第1章简单介绍了问题、算法和程序的基本概念，算法效率与算法分析方法；第2章介绍递归的基本概念和用递归描述算法的实例；第3章讨论多种常用的排序算法；第4章介绍集合的表示，集合操作的实现算法，以及在不同数据结构上应用的多种检索算法；第5章介绍了图在计算机中的表示方法，以及基于图的一些常见算法；第6章介绍了算法设计中最经常采用的多种方法；第7章介绍字符串的匹配算法；第8章介绍了已被广泛应用的索引技术。

本书在内容的选取上参考了算法设计和分析的教材，并结合“算法基础”课程的要求，更强调算法的设计思想和算法的设计过程，特别对某些问题还给出了多个设计思想不同的算法。这不仅有助于读者理解算法，更有助于读者了解算法设计方法和技能，从而使算法设计能力得到培养。书中每章有足够数量的习题供读者练习和进一步思考。

本书力求叙述通俗易懂、示例丰富，便于自学。本书除可作为自学考试教材外，也可作为高等院校计算机类相关专业的本科教材或教学参考书，亦可供从事计算机应用工作的科技人员参考。

在本书编写过程中，得到上海市高等教育自学考试委员会、复旦大学继续教学学院有关领导的支持和帮助；全书内容的选取，还承蒙复旦大学计算机科学与工程系招兆铿教授、谢铭培教授提出了宝贵建议，在此一并致以诚挚的谢意。

由于水平有限、时间仓促，书中难免会有错误存在，恳请读者及同行们批评指正。

编　者

2003年8月

目　录

算法基础

算法基础(6369)自学考试大纲

第1章 算法的基本概念

随着计算机应用的日益广泛，对信息表示和信息处理技术的研究也日益深入。当使用计算机解决问题时，需用数据结构表示信息，用相应的算法处理信息。因此，数据结构和算法的有关知识对计算机应用工作者来说是必须具备的。

计算机科学家按数据结构中元素之间的关系对数据结构进行分类，并总结出各种数据结构表示信息的方法，以及相应的解决常见问题的处理技术，但还有更多处理技术有待于人们去研究和发现。本书的目的是介绍已总结出来的现成算法，以便日后用这些算法去解决问题，或利用已掌握的知识设计新的算法。

1.1 问题、算法和程序

当计算机应用人员要用计算机进行事务处理时，就会面对问题、算法和程序。

1. 问题

在这里，问题是一个需要用计算机解决的任务。通常有一组数据输入，通过计算机处理后，就会产生对应的一组数据输出。要让计算机解决一个新问题，必须先为计算机寻找或设计一个解决该问题的算法。为了确定该算法，首先要彻底地理解该问题，并给出正确的定义。如确定任何可行方案所需的资源，确定解决问题的各种限制条件，包括计算机资源和运行时间等。问题的陈述和限制条件构成问题的定义域，问题的解是问题的值域，算法和程序完成定义域到值域的一种映射。

2. 算法

算法就是问题的求解方法，由一系列求解步骤组成。对于计算机来说，算法是一个由有限条指令组成的有限集合，这些指令确定了解决问题的运算或操作的序列。计算机算法与让人执行的方法不同，构成计算机算法的每条指令必须是确定的与可行的，每条指令必须清楚明了，没有二义性。算法的执行是一个将输入转化为对应输出的处理过程。算法描述由经明确说明的一组简单指令和规则组成，计算机按规则执行其中的指令就能在有限的步骤内解决一个问题或者完成一个函数的计算。一种特定的算法解决一个特定的问题，但解决同一个问题可以有多种算法。当知道有多种算法能解决同一个问题时，选择算法的标准首先是算法的正确性、可靠性、简单性和易理解性，即实现简单也是选择算法的重要标准。其他的标准还有算法所需要的存储空间少、执行速度快等。一般来说，一个算法应该包含以下

多个性质：

(1) 正确性。算法必须完成所期望的功能，能对每一次有效的输入，总能在有限的时间内给出正确的输出。

(2) 可操作性。一个复杂算法会由许多步骤组成，每一个步骤所描述的行为对于完成算法的计算机来说是可读的、可执行的，并且每一步必须能在有限的时间内完成。算法执行的快慢将影响一个算法是否被实际应用所采纳，为此在设计算法时，要对算法的执行时间给出某种程度上的度量信息。由于算法的实际执行时间可能会与具体的输入有关，这种度量只能在某种程度上反映算法的实际执行时间。

(3) 确定性。算法是一个操作步骤序列，正确的算法要求组成算法的规则和步骤的意义应是惟一确定的，没有二义性，并且每个步骤的后继步骤必须是明确的。因此，计算机按算法指定的操作步骤顺序执行，能在执行有限步骤后给出问题的结果。

(4) 有限性。一个算法必须由有限个操作步骤组成。

(5) 终止性。一般来说，执行算法是为了求得一个结果，算法的执行应在有限的时间内终止，不能进入死循环。

3. 程序

计算机程序是使用某种程序设计语言对某个算法的具体实现。多数情况会考虑到实现效率，程序所实现的功能可能会与算法的要求有一些差异。对于程序员来说，尽量要做到程序与算法一致，在确定了算法足够多的细节后，才按算法编写程序。为了简化表达，本书将混用算法和程序这两个名词。

由于算法的终止性，不是所有的计算机程序都是算法，如操作系统是一个程序，但它不是算法，不过操作系统中实现每个功能的函数多数实现了一个特定的算法。

1.2　算法的效率和算法分析

当有多个正确算法可以解决问题时，选择哪一个算法常常可由以下两个互相冲突的目标来确定：

(1) 易理解、易编码和易调试；

(2) 有效利用计算机资源。

理想的情况是程序能同时达到这两个目标，但在多数情况下，不能同时最佳地达到上述两个目标，程序员会根据具体情况，采用某种折中的办法实现自己的程序。与目标(1)有关的问题主要属于软件工程研究的内容；而与目标(2)有关的问题是算法分析的重要内容之一。

估量一个算法或程序的效率的方法称为算法分析，可以采用算法消耗的资源来度量算法的复杂性。用这种方法可以清楚地看到解决同一个问题的不同算法在效率上的差异。

当解决同一个问题有多个算法时，应选用消耗资源少的那一个算法。算法执行所需的资源主要包括时间和空间两方面，即执行程序需要的时间和算法中数据结构所需的内存空间。通常算法执行所需的时间是最关键的。

影响算法执行时间的因素是多方面的，如与执行程序的计算机的主频和外部设备输入/输出速度等有关。若程序用高级语言编写，则还会与所用的语言和编译系统生成的代码的质量及运行环境有关。如果在一台指定的计算机上，在给定的时间和空间限制下运行一个程序，以上因素都会对程序的执行效率有影响。但是这些因素与算法或数据结构的差异无关。为此，评价同一问题的两个算法所对应的程序，应该在同样的条件下用同一个编译器，并根据在同一台计算机上运行的效率来评价执行一个程序所要消耗的资源。由于很难准确计算出一个算法所要消耗的资源，并考虑到算法所消耗的资源通常与问题的规模有关，算法消耗的资源定量估算用问题规模的渐近函数来表示，这种方法称为渐近算法分析，也简称为算法分析。算法分析的目的就是估算出问题规模与算法的效率或开销的渐近函数关系。

采用上述渐近方法，可以这样来判断算法性能的估算标准，即对于一定规模的输入，算法必须要作的基本操作步骤数。其中输入规模用输入数据量来表示，如排序问题中，输入 n 个整数进行排序，就用 n 来表示输入规模。一般说来，完成操作所需的时间与操作数的值无关的一组操作都可以当作一个基本操作。如两个变量的值交换、两个整数的加法计算等。但求 n 个数之和的计算就不是基本操作，因为该操作的执行时间与操作数的个数 n 有关。

【例 1.1】 求有 n 个元素的一维数组的所有元素之和。

```
int sum(int *a, int n)
{   int s, i;
    for(s = 0, i = 0; i < n; i++)
     s += a[i];
    return s;
}
```

上述函数 sum()依次遍历数组中的元素，将其值累计于工作变量 s 中，待遍历结束，返回累计结果。其中，问题的规模是 n，基本操作是将数组元素累计到变量 s 中。可以认为，累计所需的时间是一定的，与被累计数的大小或其在数组中的位置无关。

因为影响时间消耗的最主要因素是输入规模，常把执行算法所需时间 T 写成输入规模 n 的函数，记作 $T(n)$，称 $T(n)$ 是运行时间函数。

把 sum()函数中的基本操作所需时间记为 c。c 还包括变量 i 的增值和检查循环是否继续等。如果不考虑 c 的实际值、函数调用和返回所需时间，只要求执行该算法的一个估计的近似时间，则运行 sum()函数的总时间可近似地认为是 cn，或写成 $T(n) = cn$。

【例 1.2】 求有 n 个元素的一维数组的中间位置的那个元素的值。

```
int aMiddle(int *a, int n)
{
    return a[n/2];
}
```

数组的中间位置上的元素 $a[n/2]$ 可以直接引用，并认为完成这个操作的时间与数组元素个数及该元素的值无关，在一台特定的计算机中是确定的，记作 c。因此，该算法的运行时间函数是 $T(n) = c$。

【例 1.3】 求 $n \times n$ 两维数组元素和。

```
#define  N  50
int sumDArray(int a[][N], int n)
{  int s, i, j;
   for(s = 0, i = 0; i < n; i++)
     for(j = 0; j < n; j++)
       s += a[i][j];
   return s;
}
```

上述函数 sumDArray()的基本操作也是累计，可以认为完成一次累计的操作所需要时间也是一定的，记为 c。同样，也可以忽略循环变量的累加和循环是否继续的测试。要执行的基本操作总次数为 n^2。因此，运行时间函数是 $T(n) = cn^2$。

由于算法的运行时间代价是一个渐近估计，对于两个算法的运行时间代价函数差异较大的算法，它们的运行时间代价的比较可简化成关于问题规模增长率的比较。设问题的规模为 n，时间代价记为 $T(n)$。若 $T(n)$是 n 的线性函数，则称 $T(n)$是线性增长率。若 $T(n)$是 n 的一个多项式函数，其中最高次幂是 p，则称 $T(n)$是多项式 p 次增长率，特别当 p 等于 2 时，称 $T(n)$是二次增长率。若 $T(n)$是 n 的一个指数函数，如 $T(n) = ck^n$，则称 $T(n)$是指数增长率。有不同增长率的两个算法，随着问题规模的增加，运行时间代价的变化会有很大的差别。当问题规模足够大时，指数增长率算法的运行时间会大大地超过多项式增长率算法的运行时间；而多项式增长率的算法的运行时间又要超过线性增长率算法的运行时间。

1. 最佳、最差和平均情况

由于算法的处理对象是数据，常常会有这样的情况，对不同的值要作不同的处理，所以即使问题的规模相同，如果输入数据不同，执行算法的实际时间代价也可能会不同。例如在一个顺序存储有 n 个元素的线性表中找一个值为 key 的元素在线性表中的位置。若采用顺序检索算法，从线性表的第一个元素开始，顺序检索直止找到值为 key 的元素，或找遍整个线性表都没有找到值为 key 的元素结束该算法。对于具体的输入，算法实际运行的时间代价可能在一个很大的范围内浮动。例如线性表的首元素的值等于 key，算法只检索了一个元素，实际运行时间最短，也是该算法运行时间代价的最佳情况。如只有线性表的最后一个元素的值等于 key，其他元素都不等于 key，则算法必须作 n 次检索，也是该算法运行时间代价的最差情况。或从一般情况来讨论这个算法，即对于给定的规模不同的线性表，或者对于同一个线性表检索不同的 key，在各元素的检查概率相同的情况下，算法平均检索到整个线性表的一半元素就能找到值为 key 的元素。也就是说，该算法平均要检索 $n/2$ 个元素，这就是算法运行时间代价的平均情况。

最佳情况是算法最理想的情况，一般说来，它不能代表一个算法的性能。最差情况也不能正确反映一个算法的性能，但能知道一个算法至少能做得多快。在实时要求很高的应用中，若不能及时处理可能发生的极端情况，则这个算法就不能采用。平均情况能反映一个算法的典型表现。然而，平均情况分析并不是总是可行的，正确的平均情况分析，需要清楚知道数据是如何分布的。如上述线性表顺序检索中的平均情况分析中得出的结论，这是基于

key 在线性表中的每个位置出现的机率相等的假设之上的。如果这个假设不成立，则该算法的平均情况还需更进一步的讨论。

由以上分析得出这样的一般结论：在实时系统中，更关注最差情况的算法分析，在其他情况下，特别是知道输入数据分布情况的时候，通常更注重平均情况。但若不能对输入数据的分布给出一种合理的假设的情况下，也只能求助于最差情况分析。

2. 渐近分析

两个算法执行的时间消耗可由它们的时间增长率函数的性质来判定。如有两个算法，它们的时间增长率函数分别为 $T_1(n)=c_1n^2$，$T_2(n)=c_2n^3$，则当 n 足够大时，几乎与 c_1 和 c_2 值无关，T_2 将一定会大于 T_1。因此，为了简化算法分析，在粗略估算算法的时间增长率函数时常忽略它的系数，这种分析方法是一种近似算法分析。简单地说，渐近算法分析是当输入规模很大时，对一种算法复杂性的估计。由于这种分析方法相对简单，常被用于算法时间性能的比较。

算法的实际执行除与输入数据的规模有关外，也可能会与输入值的分布规律有关。同一规模的两组不同输入，算法的执行时间也可能会有很大差异。在算法研究中，算法的时间复杂性常用时间函数增长率的上限来表示。算法的时间消耗函数的上限与输入规模有关，有最佳上限、最差上限和平均情况上限之分。为了简化“增长率的上限为 $f(n)$”的叙述，采用一种特殊的表示法，即大“O”表示法。如果某个算法的时间消耗增长率上限（最差情况）是 $f(n)$，则称这个算法的某种时间消耗函数“在集合 $O(f(n))$ 中”，简称“在 $O(f(n))$ 中”。例如，某算法在最差情况下 $T(n)$ 增长率速度与 n^3 相同，则称该算法最差情况代价“在 $O(n^3)$ 中”。

令 $T(n)$ 表示算法的实际运行时间，$f(n)$ 则是上限的一个函数表达式，若存在两个正常数 c 和 n_0，对任意 $n>n_0$，有 $|T(n)|\leqslant c|f(n)|$，则称 $T(n)$ 在集合 $O(f(n))$ 中。记作 $T(n)=O(f(n))$。

上述定义是说，对于问题的某种类别（比如最差情况）的输入，只要输入规模足够大（即 $n>n_0$），该算法总是能在 $c|f(n)|$ 步内完成，c 是某个确定的正常数。

当评价一个算法的时间性能时，简单地可用算法的渐近分析方法，即时间复杂度的数量级来衡量。例如，设某个问题的求解方法已有两个算法 A 和 B，它们的时间复杂度分别是 $T_A(n)=100n^2$，$T_B(n)=4n^3$，则当输入规模 $n<25$ 时，有 $T_A(n)>T_B(n)$，即算法 B 花费的时间较少。但是，随着问题规模 n 的增大，算法 A 较之算法 B 要快得多。所以，算法渐近分析方法正是从问题规模较大情况下评价算法时间性能的一种方法。

为了能从不同侧面分析算法的时间复杂性，除记号 O 之外，另引入记号 Θ 和 Ω。

定义 $T(n)=\Omega(f(n))$，如果存在正常数 c 和 n_0，使得当 $n\geqslant n_0$ 时，$T(n)\geqslant cf(n)$ 成立，则称 $T(n)$ 是 $\Omega(f(n))$，记作 $T(n)=\Omega(f(n))$。$T(n)=\Omega(f(n))$ 蕴含 $f(n)$ 在 $n\geqslant n_0$ 上给出 $T(n)$ 增长率的下界。

定义 $T(n)=\Theta(f(n))$，如果 $T(n)=O(f(n))$ 和 $f(n)=\Omega(T(n))$ 同时成立。称 $T(n)$ 是 $\Theta(f(n))$，记作 $T(n)=\Theta(f(n))$。当 $T(n)=\Theta(f(n))$ 时，$T(n)$ 和 $f(n)$ 的增长率是同阶的。

当一个算法的时间复杂性有 $T(n)=O(f(n))$ 时，就能保证函数 $T(n)$ 的增长率不会比

$f(n)$ 更快，这样，$f(n)$ 是 $T(n)$ 的上界。因此，这就蕴含着 $f(n) = \Omega(T(n))$，即 $T(n)$ 是关于 $f(n)$ 的下界。

例如，n^3 比 n^2 增长得快，可以有 $n^2 = O(n^3)$，或 $n^3 = \Omega(n^2)$。$f(n) = n^2$ 和 $g(n) = 2n^2$ 的增长率相同，所以 $f(n) = O(g(n))$ 与 $f(n) = \Omega(g(n))$ 成立。如果 $g(n) = 2n^2$，则虽有 $g(n) = O(n^4)$、$g(n) = O(n^3)$ 和 $g(n) = O(n^2)$，但是 $g(n) = O(n^2)$ 是最好的答案。

最后，指出有关算法时间复杂性渐近分析方法的几个重要规则。

【规则 1.1】 如果 $T_1(n) = O(f(n))$ 和 $T_2(n) = O(g(n))$，则 ① $T_1(n) + T_2(n) = \max(O(f(n)), O(g(n)))$；② $T_1(n) * T_2(n) = O(f(n) * g(n))$。

【规则 1.2】 如果 $T(n)$ 是 k 阶多项式，则 $T(n) = \Theta(n^k)$。

【规则 1.3】 对任何正常数 k，$\log_k n = O(n)$。这条规则说明对数增长率是非常慢的。

另有几个应用规则：

【例 1.4】 应用规则 1——单循环。

一个循环的运行时间等于内循环语句的运行时间乘上循环的次数。

【例 1.5】 应用规则 2——嵌套循环。

嵌套循环内的一组语句的运行时间是该组语句的运行时间乘上全部循环次数的积。

如下面的程序段的运行时间是 $O(n^2)$。

```
for(p = k = 0; k < n; k++)
    for(j = 0; j < n; j++)
        P++;
```

【例 1.6】 应用规则 3——多个连续语句。

由规则 1.1 可知：多个连续语句运行时间等于其中那个运行时间最长的。

如下面的程序段，第一个语句的运行时间是 $O(n)$，接着是一个运行时间为 $O(n^2)$ 的语句，则它们的运行时间是 $O(n^2)$。

```
for(k = 0; k < n; k++)
    a[k] = 0;
for(k = 0; k < n; k++)
    for(j = 0; j < n; j++)
        a[k] += a[j] + k + j;
```

【例 1.7】 应用规则 4——if/else 语句。

以下结构的 if/else 语句的运行时间不会比 S1 和 S2 中较大的运行时间加上测试时间的和更大。

```
if(条件)
  S1
else
  S2
```

1.3 算法设计实例

本小节以求解最大和子序列问题为例，说明算法设计是一件技巧性很强的工作。

【例 1.8】 设有一个整数序列，要求从给定的序列中找出和最大的部分子序列。在这里，子序列是指序列中从某个元素开始的连续若干个元素。由于序列的元素可能有负数，找出其中子序列元素和最大的子序列就变成是一件相对比较困难的工作。这里给出四种求解算法，请读者注意每种算法的思考过程。

【算法 1.1】 求最大和子序列之一。

求序列的最大和子序列有多种方法，在不深入考虑最大和子序列性质的情况下，以一般的找最大值问题处理，简单地可采用穷举法。即先预设一个最小值作为临时最大和的初值，然后逐一列举所有可能的子序列，对每个子序列，将它的和与临时最大和比较，若比临时最大和更大，则用当前和更新临时最大和。由于一个子序列可由序列的开始位置和结束位置确定，穷举所有可能的子序列，可对所有可能的子序列的开始位置和结束位置作穷举。对确定的子序列，先求这个子序列的和，然后与早先求得的临时最大和的子序列的和作比较，若当前子序列是一个其和更大的子序列，就用它的和更新临时子序列最大和。将上述结果写成算法如下：

```
{  为临时最大和子序列的和 maxSum 置初值;
   对所有可能的子序列的开始位置循环;
      对所有可能的子序列的结束位置循环 {
        为当前子序列的和 thisSum 置初值;
        循环累计当前子序列的元素于 thisSum;
        if(thisSum>maxSum)
           maxSum = thisSum;   /* 用 thisSum 更新 maxSum */
    }
}
```

将上述算法用 C 语言写成程序，即：

```
int maxSubSeq1(int a[], int n)
{  /* 已知序列的首元素位置，以及序列的元素个数，函数返回最大和子序列的和 */
      int maxSum, k, j, p, thisSum;
      maxSum = a[0];  /* 用首元素作为最大和子序列的初值 */
      for(k = 0; k < n; k++)  /* 子序列的开始位置 */
         for(j = k; j < n; j++)  /* 子序列的结束位置 */
      {    thisSum = 0;
           for(p = k; p <= j; p++)  /* 求子序列的和 */
              thisSum += a[p];
           if(thisSum > maxSum)
              maxSum = thisSum;   /* 用 thisSum 更新 maxSum */
      }
      return maxSum;
}
```

在上述算法中，置最大和子序列的初值用时 O(1)，第一个循环运行 n 次，第二个循环运

行 $n-k$ 次，略微放大后，假定运行 n 次，第三个循环运行 $j-k+1$ 次，再次假定为 n 次。循环内的其他语句的运行时间是 O(1)。所以算法总的运行时间是 $O(1*n*n*n)=O(n^3)$。

如要更精确的分析，计算实际循环的执行次数，则有以下结果：最内循环运行 $j-k+1$ 次，第二个循环运行 $n-k$ 次，对上式 j 从 k 到 $n-1$ 求和，这样两个内循环一起，元素共累计次数为：

$$(n-k+1)(n-k)/2 \tag{1.1}$$

最后，考虑到外循环要运行 n 次，对式 1.1 中 k 等于 0 至 $n-1$ 求和，得元素累计总次数为：

$$(n^3+3n^2+2n)/6$$

【算法 1.2】 求最大和子序列之二。

对应开始位置，所有可能的结束位置将逐一发生递增变化，所以求当前子序列的和可采用递推法，即由前一个子序列的和加上新的结束位置上的元素来实现。考虑到这个改进，上述算法又可优化成以下形式：

```
{   为临时最大和子序列的和 maxSum 置初值；
    对所有可能的子序列的开始位置循环 {
      为当前子序列的和 thisSum 置初值 0；
      对应开始位置，对所有可能的子序列的结束位置循环 {
        循环累计当前结束位置上的元素于 thisSum；
        if(thisSum > maxSum)
          maxSum = thisSum;   /*用 thisSum 更新 maxSum*/
    }
    }
}
```

将上述算法用 C 语言写成程序，即：

```
int maxSubseq2(int *a, int n)
{   int i, j, maxSum, thisSum;
    maxSum = a[0];
    for(i = 0; i < n; i++)
    {   thisSum = 0;
        for(j = i; j < n; j++)
        {   thisSum += a[j];
            if(thisSum > maxSum)
              maxSum = thisSum;
        }
        }
        return maxSum;
}
```

经上述改进，算法的运行时间是 $O(n^2)$。

【算法 1.3】 求最大和子序列之三。

采用递归方法，可以将相对比较复杂的算法的运行时间加快到 O($n$lb n[①])。用分治方法，将问题分划成两个几乎相等的子问题，然后用递归方法求解这两个子问题。

若将序列分成几乎相等的左右两部分，则最大和子序列有三种出现可能，即全在左边部分，全在右边部分，或一部分在左边部分，另一部分在右边部分。其中前两种情况可以直接用递归方法求解。第三种情况可以在这样两种特别的子序列中寻找，一个是左边部分包含该部分最末元素的最大和子序列，另一个是右边部分包含该部分首元素的最大和子序列，第三种情况就是这样两个子序列的和。如序列 4→—3→5→—2→—1→2→6→—2 中，左边部分最大和子序列是由前 3 个数组成，其和是 6；右边部分最大和子序列是由它的两个中间数组成，其和是 8。对于左边部分包含它的末元素的最大和子序列是全部 4 个数，其和是 4。同样，右边包含它的首元素的最大和子序列是前 3 个数，其和是 7。这样，跨左右两部分的最大和子序列的和是 11。从上述三个子序列中选取和最大的子序列就是问题的解，最终求得所给序列的最大和子序列的和是 11。按上述思想编写成算法如下：

```
int maxSubSeq3(int a[], int left, int right, int *sp, int *mp)
{   /* 已知序列,及子序列的开始下标和结束下标,函数返回最大和子序列的和,最大和子序列的开始位置和元素个数由参数带回 */
    int maxSum, start, m, k, maxBorderSum, maxRightSum, center,
        tempStart, tempEnd, tempM, thisSum;
    if(left == right) {   /* 当数组段只有一个元素时,立即得到结果 */
        *sp = left; *mp = 1;
        return a[left];
    }
    center = (left + right)/2;   /* 求序列的中点位置 */
    maxSum = maxSubSum3(a, left, center, &start, &m);
                                                    /* 求左边最大和子序列 */
    maxRightSum = maxSubSum3(a, center+1, right, &tempStart, &tempM);
    if(maxSum < maxRightSum) {   /* 从左右两部分中确定一个更大的 */
      maxSum = maxRightSum; start = tempStart; m = tempM;
    }
    maxBorderSum = a[center]; thisSum = 0;
    for(k = center; k >= left; k--) {
                                        /* 找左边包含最左元素的最大和子序列 */
      thisSum += a[k];
      if(thisSum > maxBorderSum) {
        maxBorderSum = thisSum;
        tempStart = k;
```

① lb $n = \log_2 n$。