算法设计分析的理论与方法

顾立尧　霍义兴　编著

上海交通大学出版社

算法设计分析的理论与方法

顾立克　霍义兴　编著

上海交通大学出版社

算法设计分析的理论与方法

出　版：上海交通大学出版社
（淮海中路1984弄19号）
发　行：新华书店上海发行所
印　刷：常熟文化印刷厂印刷
开　本：787×1092（毫米）1/16
印　张：12
字　数：297000
版　次：1989年3月第1版
印　次：1989年4月第1次
印　数：1—1100
科　目：193—276
ISBN 7-313-00441-9/TP·39

定价：2.45元

前　言

算法研究是计算机科学的核心内容之一，也是有关计算机专业学生的必修课程之一。人们早就认识到，为加快程序的执行，不能光指望计算机硬件速度的提高，还需着力于研制快速的算法。采用高效快速算法往往比提高硬件速度更能节省时间。因而算法研究这个领域引起了人们强烈的兴趣，这门课程主要通过对常见的典型算法的剖析，介绍一些算法设计及时空复杂性分析的基本方法。现有的这方面的书籍与"数据结构"的界面不很清晰，另外书中由于没有补充一些数学准备知识，影响问题的深入分析。笔者多年来从事"数据结构"及"算法设计与分析"方面的教学，希望有一本既与"数据结构"界面清晰，又有必备数学基础，自成系统的书籍，这是编写本书的主要目的。

本书讨论的重点放在非数值算法方面。近年来由于平行处理方面的进展，平行算法的研究也发展得很快，考虑到串行算法是平行算法的基础，另外串行算法一套分析方法也较为成熟，作为教材，本书只讨论串行算法。

本书内容的选取参考了美国IEEE—83计算机教育大纲中有关"算法设计与分析"方面的内容，因而可作为计算机科学与工程系不同专业不同程度学生的教材或教学参考书。使用时，可根据不同的要求选取部分或全部内容予以讲解。全部讲解估计需要60个学时。前已提及本书自成系统，只要具备高级语言(最好是PASCAL语言)及数据结构知识，自学本书是不成问题的，因而很适宜作为没学过"算法设计与分析"的计算机工作者的参考书。

本书第一章介绍三种计算模型RAM，RASP，TURING机及计算复杂性的测量。第二章介绍递归技术与分治法，这是常用的算法设计方法，掌握后便于解决各种类型的问题，并且可遵循一定的方法来进行时间复杂性的分析。为便于读者解递归方程，本章中补充了一些生成函数及常系数线性递推方程方面的数学知识。最后举了几个分治法解题的例子。这一章的内容既基本又非常重要务必深刻理解，牢固掌握。第三章在数据结构中学过排序的基础上再深入一步讨论排序问题，主要是深入地分析算法复杂性，并再介绍几种数据结构中没有介绍过的排序算法，例如分段逆序插入排序，二分查找合并排序。第四章介绍集合运算问题及与其相宜的数据结构。由于对一个给定问题设计有效算法首先要考虑该问题的基本性质，有一类问题的对象可用数学上的集合来代表。在这一章中将介绍集合上的七种基本运算，并提出不同的数据结构以适于不同的运算序列。第五章介绍图的算法。工程与科学上的很多问题可形式化地用无向图或有向图来表示。在本章中主要介绍一些有多项式解的基本的图论问题。第六章介绍一些很重要的设计策略，如贪心法，动态规划，回溯法及分技限界法。通过一些实例来深入介绍如何应用这些设计方法，关键不是单纯地理解这些算例，而是要在碰到各种具体问题时如何得心应手地使用这些设计方法。这一章的内容丰富而又重要，务必深入理解掌握。第七章讨论一个重要的问题类——NP-完全问题，这一类中的所有问题在计算困难性上是等价的，只要这类问题中的一个有了一个有效(多项式时间为界)的算法，那么这类问题中的所有问题均有了有效的解。仅管这类问题计算上是困难的，但实际碰到的问题中有相当一部分是这类问题，所以讨论这类问题是有实用意义的。在这一章中将介绍非确定型图灵机，P类和NP类，

NP-完全性，Cook 定理，NP-完全问题及其近似解法。此章有一定难度。

考虑到目前国内计算机教育把 PASCAL 语言作为工具语言，因而本书中用 Pidgin PASCAL 语言来描述算法，不直接用 PASCAL 语言可避免陷在 PASCAL 语言细节中而忽略了算法的核心部分。如用到特殊符号则随时加以解释。

从理解书中的内容到自己能设计分析算法，其间有一个很大的台阶，弥补这个台阶的最好方法便是通过习题练习设计分析算法。这将花去读者大量的时间，但非常必要。

上海交通大学霍义兴编写了第五、七章，第一、二、三、四、六章由上海机械学院顾立尧编写。在本书编写过程中得到了上海机械学院高建、奚霞明、沈宇豪的大力帮助。

如书中有什么错误或不妥之处，望读者批评指正，以便笔者不断提高自己的水平。

编　者

内 容 简 介

算法的设计与分析是计算机科学的重要内容之一。本书是作者在多年教学实践基础上，参考了IEEE教程大纲后编写的。

全书共分七章：计算模型、递归技术与分治法、排序、集合运算问题及其合适的数据结构、图的算法、有效算法的设计及NP-完全问题。各章前后呼应，是一个有机的整体；但各部分又相对独立，需要时，也可单独使用。另外，每章都附有大量的例题和习题，便于教学或自学。

本书可供大专院校的本科生、研究生作为教材使用，也可供从事计算机科学的科研人员和工程技术人员使用。

目　录

第一章　计算模型 ……1
1.1　算法及算法的计算复杂性……1
1.2　随机存取模型 RAM 及 RAM 程序的计算复杂性 ……6
1.3　随机存取存贮程序模型 RASP ……13
1.4　RAM 模型的简化 ……16
1.5　图灵机模型 ……19
1.6　图灵机模型与 RAM 模型间的关系 ……23
第二章　递归技术与分治法……27
2.1　分治法 ……27
2.2　递归方程及其求解 ……29
2.3　生成函数 ……31
2.4　常系数线性递推方程 ……35
2.5　分治法举例 ……40
第三章　排序……48
3.1　排序问题 ……48
3.2　比较方法排序的时间下限 ……49
3.3　基数排序 ……50
3.4　分段逆序插入排序 ……55
3.5　二分查找合并排序 ……59
3.6　堆排序 ……65
3.7　顺序统计 ……69
第四章　集合运算问题及其合适的数据结构……73
4.1　集合的基本运算 ……73
4.2　散列 ……74
4.3　最优二叉查找树 ……76
4.4　简单不相交集的合并算法 ……83
4.5　UNION-FIND 问题的树结构及其应用……86
4.6　平衡树模式 ……
4.7　字典及优先队列 ……
4.8　可并堆 ……
4.9　可连接队列 ……
第五章　图的算法 ……
5.1　最小耗费生成树……
5.2　无向图的先深搜索……

5.3 无向图的双连通性……105
5.4 有向图的先深搜索……110
5.5 有向图的强连通性……111
5.6 路径寻找问题……114
5.7 传递闭包的算法……115
5.8 最短路径算法……116
第六章 有效算法的设计……122
6.1 贪心法……122
6.2 动态规则……133
6.3 回溯法……143
6.4 分枝限界法……152
第七章 NP-完全问题……173
7.1 基本概念……173
7.2 非确定性图灵机(NDTM)……174
7.3 P类和NP类……178
7.4 NP-完全性……179
7.5 可满足性问题的NP-完全性……180

第一章 计算模型

1.1 算法及算法的计算复杂性

对于一个给定的问题，如何寻找一个解决这个问题的有效算法？一旦找到了一个算法，如何来与能解决同一问题的其他算法作比较？到底如何来判定一个算法的好坏？计算机科学家与程序设计员对这些问题很感兴趣。

首先我们必须对算法有一个明确的概念。一个算法，就是一个有穷规则的集合，其中的规则规定了一组解决某一特定类型的问题的运算序例。

一个算法应该具有如下五个重要的特性。

(1) 有穷性　一个算法必须总是在执行有穷次之后结束(如果有算法的其他所有特性，而且不具有穷性，则可以叫做"计算方法")。

(2) 确定性　算法的每一个步骤必须是确切地定义的。

(3) 输入　一个算法有0个或多个输入，它就是在算法执行之前，初始给出的量。

(4) 输出　一个算法有一个或多个输出，它是同输入有某种特定关系的量。

(5) 能行性　一般地讲，希望一个算法是能行的，也就是说它们原则上都是能够精确地进行的，而且人们用笔和纸做有穷次即可完成。

对于一个具体的应用，通常可能有若干个算法可以选用，我们很需要知道哪一个算法是最好的。我们来分析一个求极大值的算法：给定几个元素$x[1]$，$x[2]$，…，$x[n]$，现要求出m和j，使得

$$m=x[j]=\underset{1\leqslant k\leqslant n}{\mathrm{MAX}}\{x[k]\},$$

而且使其中的j尽可能地大。

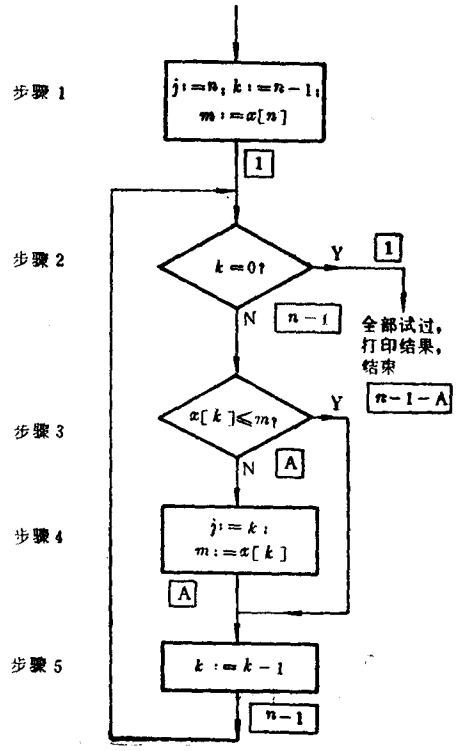

图1.1　求极大值算法流程图

上述算法要求的存贮量是固定的，因此可以仅仅分析执行需要的时间，为此要计算执行每一步的次数：

步　骤	次　数
1	1
2	n
3	$n-1$
4	A
5	$n-1$

再细分的话，用算法流程图中方框内的数字代表该处执行的次数。知道了每步执行的次数，就为我们提供了在一个具体计算机上运行该算法所花费时间的必要信息。

在上例中除A外的每一个数量都是已知的，为了完成整个分析，必须研究这个有趣的数量A。这个分析包括求A的极小值、极大值和平均值。由A的极小值、极大值和平均值便可容易地求出总的次数的极小值、极大值和平均值。

如$x[n]=\underset{1\leq k\leq n}{\mathrm{MAX}}\{x[k]\}$时，$A$的极小值为0。因在步骤1中，$m:=x[n]$后，$x[1]$，$x[2]$，…，$x[n-1]$均比$x[n]$小，故$A=0$。如$x[1]>x[2]>\cdots>x[n]$，$m$将改变$(n-1)$次，故$A$的极大值是$(n-1)$。求$A$的平均值比求$A$的极小值及极大值要麻烦。可肯定的是$A$的平均值在0和$(n-1)$之间，平均值定义为

$$A_n=\sum_k k\cdot p_{nk},\quad 0\leqslant k\leqslant n-1,$$

其中假定各个$x[k]$的值是各不相同的，$x[1]$，$x[2]$，…，$x[n]$有$n!$种排列，假定$n!$种排列是等概率的。p_{nk}为$A=k$的概率。

$p_{nk}=(n!$种排中满足$A=k$的排列数$)/n!$。

先看一个具体例子，例如$n=3$，假定以下各种情况$(n!=3!=3\times2\times1)$的每一种都是等概率的。

情　况	A的值
$x[3]>x[2]>x[1]$	0
$x[2]>x[3]>x[1]$	1
$x[3]>x[1]>x[2]$	0
$x[1]>x[3]>x[2]$	1
$x[2]>x[1]>x[3]$	1
$x[1]>x[2]>x[3]$	2

当$n=3$，满足$A=0$的排列数有2，$p_{30}=\frac{1}{3}$；满足$A=1$的排列数有3，$p_{31}=\frac{1}{2}$；满足$A=2$的排列数有1，$p_{32}=\frac{1}{6}$。故

$$A_n=\sum_{k=0}^{2}k\cdot p_{3k}=0\times\frac{1}{3}+1\times\frac{1}{2}+2\times\frac{1}{6}=\frac{5}{6}。$$

可见，A的平均值与各个$x[k]$的绝对值大小无关。

可以证明

$$A_n=H_n-1,$$

其中

$$H_n=1+\frac{1}{2}+\cdots+\frac{1}{n}=\sum_{1\leq k\leq n}\frac{1}{k},\quad n\geqslant0。$$

对A这个数分析清楚了，总的次数也就清楚了。但若要精确探究所花费的时间，必须具有下列信息：所使用的计算机；所使用计算机的机器语言指令系统；每一机器指令所需的时间；把源程序翻译成机器语言的编译程序。现在还有平行处理的计算机，如ILLIACIV等。因而要取得精确的计时数字是困难的，且也是没有意义，由于计时数字受了上述诸多条件的限制，也不能很确切地反映算法的优劣。因此我们必须建立某些共同的语言来描述算法的优劣，并要在既简单而具有代表性的计算模型上讨论这些问题。

评价算法可用不同的准则。最经常用的是在解越来越大的问题时空间与时间需求的增长速度。当涉及一个具体问题时，用称做问题规模的整数来度量输入数据的数量。例如矩阵相乘

问题的规模可以相乘矩阵的最大阶数来衡量；图论问题的规模可以边数来衡量。

一个算法所需的时间被表达为问题规模的函数，即称为算法的时间复杂性 (time complexity)，记作 $T(n)$。当问题规模逐渐增大，时间杂度的极限称作渐近时间复杂性 (asymptotic time complexity)。同样可定义空间复杂性(space complexity)及渐近空间复杂性 (asymptotic space complexity)。

渐近时间复杂性最终决定了该算法能解的问题的规模。对于常数 $C>0$，如一个算法能在时间 Cn^2 内处理完规模大小为 n 的输入，该算法的时间复杂性为 $O(n^2)$，称作"n^2 级"。("order n^2")。

更精确的定义如下：

$f(n)=O(g(n))$，当且仅当存在两个正的常数 C 及 n_0，对于所有 $n\geqslant n_0$，$|f(n)|\leqslant C|g(n)|$ 成立。

亦可理解成函数 $f(n)$ 以函数 $g(n)$ 为界。我们说一个算法的时间复杂性 $T(n)=O(g(n))$。

例 1.1.1　如算法的时间复杂性为

$$T(n)=a_m n^m+\cdots+a_1 n+a_0,$$

$T(n)$ 为一 m 阶的多项式，那么 $T(n)=O(n^m)$。

证明：
$$\begin{aligned}|A(n)|&\leqslant|a_m|n^m+\cdots+|a_1|n+|a_0|\\&\leqslant(|a_m|+|a_{m-1}|/n+\cdots+|a_0|/n^m)n^m\\&\leqslant(|a_m|+\cdots+|a_0|)n^m。\end{aligned}$$

$\because$ $n\geqslant 1$，

选择　　$C=|a_m|+\cdots+|a_0|$ 及 $n_0=1$，

因而存在正的常数 C 及 n_0，对于所有 $n\geqslant n_0$，

$$|f(n)|\leqslant |n^m| \text{ 成立，}$$

$\therefore$ $T(n)=O(n^m)$ 证毕。

我们主要在数量级上讨论算法的时间复杂性，对 C 的大小不是最感兴趣，只要找到有这么一个 $C>0$ 存在便可以了。实际上在例 1.1.1 中有很多常数可选作 C，只要这常数大于 $|a_m|$ 便行。

假如一算法中有 k 条语句，它们的执行次数各是 $C_1n^{m_1}$，$C_2n^{m_2}$，…，$C_kn^{m_k}$，整个算法执行的次数可由

$$C_1n^{m_1}+\cdots+C_kn^{m_k}$$

给出。那么该算法的时间复杂性是 $O(n^m)$，其中 $m=\mathrm{MAX}\{mi\}$，$1\leqslant i\leqslant k$。

当 n 非常大时，也就是在渐近时间复杂性的概念下，下面的不等式显然成立：

$$O(1)<O(\log_2 n)<O(n)<O(n\log_2 n)<O(n^2)<O(n^3)<O(2^n)。$$

$O(1)$ 的意思为需要一些固定的基本操作次数，所以总的执行时间需要是一个常数。

当 n 不是非常大时，到底哪个算法快，就要看具体情况了。例如第一个算法的时间复杂性 $T_1(n)=O(n)$，其常数 $C_1=10^4$，即计算时间是 10^4n 步。第二个算法的时间复杂性 $T_2(n)=O(n^2)$，其常数 $C_2=1$，即计算时间是 n^2 步。到底哪个算法快？当 $n<10^4$ 时，第二个算法快；当 $n>10^4$ 时，第一个算法快。

如一个算法的时间复杂性 $T(n)=O(2^n)$，当问题规模 n 增长时，它增长得非常非常快。假如把一个本来需要指数时间的算法改设计成只需多项式时间的算法，那就已完成了一项巨大

的工作，这个问题在第七章中还要仔细讨论。

表 1.1 常见计算函数值的比较

$\log_2 n$	n	$n\log_2 n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

O 符号用来表示上界，$T(n)=O(g(n))$的意义是当问题规模无限增大时，$T(n)$ 以函数 $g(n)$为上界。我们同样希望有一函数可表示下界。

定义：$f(n)=\Omega(g(n))$，当且仅当存在正的常数C及n_0，对于所有 $n>n_0$，$|f(n)|\geqslant C|g(n)|$。

在有些情况下，一个算法的时间复杂性 $T(n)=\Omega(g(n))$，并且 $T(n)=O(g(n))$。这时我们用下列符号。

定义：$f(n)=\theta(g(n))$，当且仅当存在正的常数 C_1、C_2 和 n_0，对于所有 $n>n_0$，$C_1|g(n)|\leqslant|f(n)|\leqslant C_2|g(n)|$。

如一个算法的时间复杂性 $T(n)=\theta(g(n))$，那么 $g(n)$同时是 $T(n)$的上界及下界。这就意味着在最坏与最好情况下需要同样数量级的时间。可举一个简单的例子说明确实存在这样的情况。例如在几个数中找一个最大数，最坏情况最好情况均是要作$(n-1)$次比较，

$\therefore$ $T(n)=O(n)=\Omega(n)$，也就是说 $T(n)=\theta(n)$。

在 n 个数中查找一个特定的数，最好情况在作第一次比较时便已找到，即 $T(n)=\Omega(1)$。最坏情况在作最后一次比较时才找到，即 $T(n)=O(n)$。

下面给出一个更强的数学符号。

定义：$f(n)\sim o(g(n))$(读作 $f(n)$是渐近于 $g(n)$)，当且仅当存在正的常数 n_0，

$$\lim_{n>n_0\text{且}n\to\infty} f(n)/g(n)=1。$$

如 $T(n)=a_kn^k+\cdots+a_0$，那么

$$T(n)=O(n^k)\quad 及\quad T(n)\sim o(a_kn^k)。$$

由于高速巨型计算机的出现(CRAY-II 的运行速度已达 10 亿次/s)，也许有人会提出是否有必要还需对算法的优劣进行研究，可能有人认为在高速巨型机前面，一切问题都可快速地迎刃而解。这是一种非常片面的看法，通过下面的分析可见，即便巨型机的运行速度非常高，对算法优劣的研究仍是至关重要的。

假如有 5 个算法 $A_1\sim A_5$，它们的时间复杂性如下：

算　法	时间复杂性
A_1	n
A_2	$n\log_2 n$

A_3	n^2
A_4	n^3
A_5	2^n

这里的时间复杂性是指处理完一个规模为 n 的输入所需要的单位时间数。假如我们取一个单位时间数为 1 ms。算法 A_1 在 1s 内能处理完规模为 1000 的输入，算法 A_3 在 1 s 内只能处理完规模为 31 的输入，而算法 A_5 在 1 s 内至多只能处理完规模为 9 的输入。表 1.1 给出了这五个算法在 1 s 内，1 min 内及 1 h 内能处理完的最大的输入量。

表 1.2　不同算法在不同单位时间里所能处理的最大输入量

算　法	时间复杂性	最大输入量		
		1s 内	1 min 内	1 h 内
A_1	n	1000	6×10^4	3.6×10^6
A_2	$n\log n$	140	4893	2.0×10^5
A_3	n^2	31	244	1897
A_4	n^3	10	39	153
A_5	2^n	9	15	21

如以 1 min 为时间单位，当算法 A_5 由 A_2 来取代时，可处理的输入量要增加到 326 倍。当以 1 h 为时间单位，算法 A_5 由 A_2 来取代时，可处理的输入量要增加到将近一万倍。

表 1.3 说明计算机的速度提高 10 倍后使每个算法的处理能力改变的情况。

表 1.3　计算机速度提高 10 倍的效果

算　法	时间复杂性	速度提高前所能处理的输入量	速度提高后所能处理的输入量
A_1	n	S_1	$10\,S_1$
A_2	$n\log n$	S_2	对于大的 S_2，接近 $10\,S_2$
A_3	n^2	S_3	$3.16\,S_3$
A_4	n^3	S_4	$2.15\,S_4$
A_5	2^n	S_5	$S_5+3.3$

就算法 A_5 而言，当计算机的速度提高 10 倍，算法 A_5 在某一单位时间内所能处理的输入量只增加了 3 个。即便计算机的速度提高一万倍，算法 A_5 在某一单位时间内所能处理的输入量也只增加了 13 个。由表 1.1 及表 1.2 可清晰地看出，计算机速度提高的效果还不如用算法 A_2 来取代算法 A_5 的效果明显。由此可见改进算法的时间复杂性能极大地增进计算机解决问题的能力。现代社会是一个信息社会，要求处理的信息量越来越多，虽然计算机的处理速度越来越快，但高效率算法的设计仍显得越为重要。

在进一步讨论算法及其复杂性前，必须先规定一个执行算法的计算模型，并且要定义一个基本的计算步。现实世界中没有一台计算机可适合各种情况，如任何一台计算机的字长总是

有限的，它不可能容纳下任意大的一个整数，等等。因而必须选择一个合适的计算模型，它既能解决现实计算机所遇到的困难，又能正确地反映在真实计算机上所进行的计算时间。

下面几节要讨论几个基本的计算模型，最为重要的是随机存取模型 RAM (randon access machine)，随机存取存贮程序模型 RASP (random access stored program machine) 以及图灵机(turing machine)。这三个计算模型在计算能力上是相等的，但在速度上是不同的。

1.2 随机存取模型 RAM 及 RAM 程序的计算复杂性

随机存取模型 RAM 是台单累加器计算机，不允许在其中自行修改其指令。

RAM 包含有只读输入带，只写输出带，程序存储部件，存储器及指令计数器。存储器中的 0 号寄存器作为累加器(accumulator)。

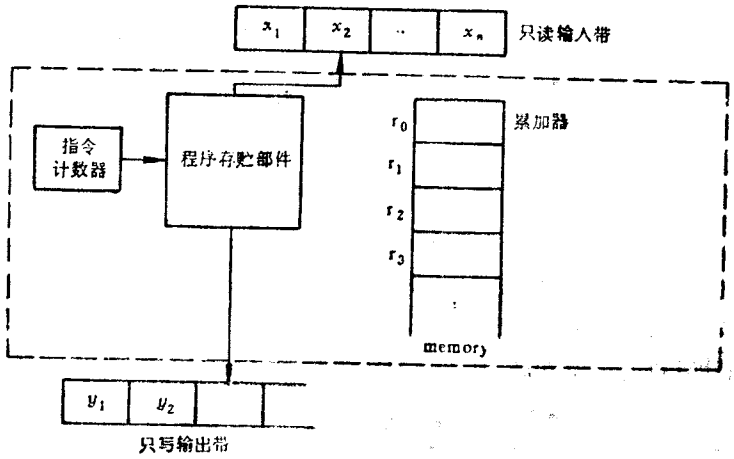

图 1.2 随机存取机器

只读输入带由一系列方格组成，每格可存放一个整数(可能是负的)，读一个数，带头后移一格。只写输出带的每个方格中初始为空，执行一条写的指令后，在输出带头下的输出带中的空格中打印出一个整数，然后带头右移一格。输出信号一经写出，不能再修改。

存贮器包括一系列寄存器 $r_0, r_1, \cdots r_i, \cdots$。$r_0$ 为累加器，每个单元能存放一个任意大小的整数。内存单元的上界不受限定，也就是说任何程序可使用任意多个内存单元。这样的抽象在下述情况下便是真实的：

(1) 问题规模不超过一台计算机的存贮容量；

(2) 计算中所出现的任何整数的大小不会超过计算机的字长。

RAM 的程序不是存放在存贮器中，这样程序便不能自行修改其本身。程序是一个带标号的指令序列，与真实计算机中所用的指令相仿。设有算术运算指令，输入输出指令，存取数指令及转移指令。

寻址方式有直接寻址和间接寻址两种基本寻址方式。

所有计算在累加器 r_0 中进行，它像其他存贮器中的寄存器一样，能容纳一个任意大小的整数。

原则上，可根据实际需要增设实际计算机所使用的其他指令，如增设逻辑运算指令，字符

操作指令或位操作指令。

操作地址有以下三种形式：

(1) $=i$，直接数型，操作数是整数 i 本身，为无地址指令；

(2) i，直接地址型，i 是一非负整数，操作数是内存单元 r$_i$ 的内容。

(3) $*i$，间接地址型，i 为非负整数。j 是内存单元 i 中所存贮的那个整数，而操作数是内存单元 j 中的内容 $c(j)$，即操作数

$c(j)=c(c(i))$。（当 $c(i)<0$ 时，$c(j)=c(c(i))$无定义。）

RAM 对所有无定义的指令作停机处理。

设操作数 a 的值为 $v(a)$，故

$$v(=i)=i,$$
$$v(i)=c(i),$$
$$v(*i)=c(c(i))。$$

	操作码	操作地址	注　　解
(1)	LOAD	$=i/i/*i$	取操作数入累加器
(2)	STORE	$i/*i$	累加器中数存入内存
(3)	ADD	$=i/i/*i$	加法运算
(4)	SUB	$=i/i/*i$	减法运算
(5)	MULT	$=i/i/*i$	乘法运算
(6)	DIV	$=i/i/*i$	除法运算
(7)	READ	$i/*i$	读入
(8)	WRITE	$=i/i/*i$	输出
(9)	JUMP	标号	无条件转移到标号语句
(10)	JGTZ	标号	正转移到标号语句
(11)	JZERO	标号	零转移到标号语句
(12)	HALT		停机

举二个简单的例子：

LOAD 2　从第二个内存单元取数到累加器 $c(0)\longleftarrow c(2)$

ADD=1　加 1，$c(0)\longleftarrow c(0)+1$

STORE 2　再把结果从累加器存入第二个内存单元中，$c(2)\longleftarrow c(0)$

再举一个间接地址取数的简单例子：

LOAD *2　把第 $c(2)$个内存单元中的数 $c(c(2))$取到累加器 $c(0)\longleftarrow c(c(2))$

ADD =3　加 3，$c(0)\longleftarrow c(0)+3$

STORE 1　再把结果从累加器存回到第一个内存单元中，$c(1)\longleftarrow c(0)$

这些指令与实际计算机中的指令是很相近的，为帮助一部分不熟悉指令的读者熟悉指令，再把各指令的语义仔细阐明如表 1.4 所示。

总的来讲，一个 RAM 程序定义了从输入带到输出带的一个映射。对某些输入，映射可能是无定义的，故它是一个部分映射。我们可对这种映射关系作不同的解释。两个重要的解释：或是作为一个函数，或是作为一种语言来解释。

表 1.4 RAM 指令的意义(操作数 a 可以是 $=i$, i, $*i$ 三者之一)

指　令	语　义
(1) LOAD a	$c(0)\longrightarrow v(a)$
(2) STORE i	$c(i)\longleftarrow c(0)$
STORE $*i$	$c(c(i))\longleftarrow c(0)$
(3) ADD a	$c(0)\longleftarrow c(0)+v(a)$
(4) SUB a	$c(0)\longleftarrow c(0)-v(a)$
(5) MULT a	$c(0)\longleftarrow c(0)\times v(a)$
(6) DIV a	$c(0)\longleftarrow \lfloor c(0)/v(a)\rfloor$
(7) SEAD i	$c(i)\longleftarrow$当前输入符号
READ $*i$	$c(c(i))\longleftarrow$当前输入符号,读完后在上两种情况下均是带头右移一格
(8) WRITE a	$v(a)$输出到输出带上带头对应的那一格,然后带头右移一格
(9) JUMP b	指令计数器置到标号为 b 的指令
(10) JGTZ b	假如 $c(0)>0$, 指令计数器置到标号为 b 的指令;否则,指令计数器置到下一指令。
(11) JZERO b	假如 $c(0)=0$ 指令计数器置到标号为 b 的指令;否则,指令计数器置到下一指令。
(12) HALT	执行停止。

假如一个程序 P 不断地从输入带上读入 n 个整数,并且在输出带上至多输出一个整数。当 $x_1, x_2, \cdots, x_n$ 是输入带上开始 n 个方格上的整数,程序 P 在输出带的第一个方格上输出 y,然后停止,那么便说程序 P 计算了函数 $f(x_1, x_2, \cdots, x_n)=y$。

例 1.2.1　函数

$$f(n)=\begin{cases} n^n, \text{对所有 } n\geqslant 1 \text{ 的整数,} \\ 0, \text{其余情形。} \end{cases}$$

计算 $f(n)$的 Pidgin Pascal 程序如下,我们忽略程序的细节。如变量说明等。

```
begin
    read (r1)
    if r1≤0 then write 0
    else
        begin
            r2←r1;
            r3←r1−1;
            while r3>0 do
                begin
                    r2←r2*r1;          {共乘(n−1)次}
                    r3←r3−1
                end;
            write(r2)
        end
end
```

计算 n^n 的 RAM 程序如下：

	RAM 程序		相应的 Pidgin Pascal 语句
	READ	1	read r1
	LOAD	1	if r1≤0 then write 0
	JGTZ	pos	
	WRITE	=0	
	JUMP	endif	
POS:	LOAD	1	r2←r1
	STORE	2	
	LOAD	1	r3←r1−1
	SUB	=1	
	STORE	3	
while	LOAD	3	while r3>0 do
	JGTZ	continue	
	JUMP	endwhile	
continue:	LOAD	2	r2←r2*r1
	MULT	1	
	STORE	2	
	LOAD	3	r3←r3−1
	SUB	=1	
	STORE	3	
	JUMP	while	
endwhile	WRITE	2	write r2
endif:	HALT		

解释一个 RAM 程序的另一个方法是把它当成一个语言接受器。一个字母表是符号的有穷集合，而语言是字母表上字符串的集合。字母表的符号能用整数 1, 2, …, k 表示。一个 RAM 机器能以下列方式接受语言。在输入带上放一个输入字符串 $S=a_1a_2\cdots a_n$（可用 n 个整数 x_1, x_2, …, x_n 表示），在输入带的第一方格上放符号 a_1，第二方格上放符号 a_2，……，在第$(n+1)$个方格上放 0，作为输入串的结束标志符。

如一个 RAM 程序 P 读了字符串 S 全体及结束标志符 0 后，在输出带的第一格输出一个 1 并停机，就说程序 P 接受字符串 S。P 可接受的语言 L 是 P 可接受的字符串的集合。对于不在 P 可接受的语言中的输入串，程序 P 在输出带上输出一个不同于 1 的符号并停机，或者程序 P 永远不停机。

例 1.2.2 讨论一个 RAM 程序，它接受由输入字母表{1, 2}上所有由同样多个 1 与 2 组成的字符串。这程序把每个输入符号读入到寄存器 r_1 中，在寄存器 r_2 中记下到目前为止读入的 1 的个数与 2 的个数之差。当读入结束标志符 0 时，程序检查出 1 与 2 的差数为 0 便输出 1 并停机。在此我们假设了仅有可能的输入符号是 0、1 及 2。

识别具有同样个数的 1 与 2 的字符串的 Pidgin Pascal 程序如下：

```
begin
    d←0;
    read x;
    while x≠0 do
        begin
            if x≠1 then d←d−1 else d←d+1;
```