§7.7 消除算法中的递归

一．递归算法

1.什么是递归算法？

一个算法在运行过程中直接或间接地调用自身，则该算法称为递归算法。

2.使用和消除递归

在本课程中，我们常用递归算法来描述一个解决问题的算法，特别是用分治策略设计算法时，用递归来描述算法十分自然。此外，用递归来描述算法常使算法简洁易懂。

然而在一些语言中，过程调用(特别是递归过程调用)比一般的赋值语句代价高得多。因此，如果我们删除程序中的递归，常常使一个程序运行速度比原来快一个大的常数因子。

二．递归算法的例子

例子：给定一个目标值t和一组正整数权，问是否存在这些正整数的一个子集，使得该子集中正整数之和为t。这个问题称为背包问题。例如：给定正整数权和t值，选取上述正整数中第个正整数，有。

解背包问题的递归算法为：

FUNCTION Knapsack (target : integer; candidate : integer) : boolean;

BEGIN

1. IF target = 0 THEN
2. RETURN(true)
3. ELSE IF (target < 0) OR (candidate > n) THEN
4. RETURN(false)

ELSE

1. IF Knapsack( target-weights[candidate], candidate+1)

THEN BEGIN

6. WRITELN( weights[candidate] );

7. RETURN(true)

END

8. ELSE RETURN( Knapsack(target, candidate+1) )

END;

三．消除尾递归

常常我们有一个机械的方法，消除过程中最后一个调用自身的语句。

如果过程在其最后一个语句调用，那么我们用代替过程调用语句，在该赋值语句后加上一条转移到过程代码的开始的转移语句，就可以消除尾递归。当过程有多个参数，可以类似和那样处理。

例如：上述的尾递归语句可以改为

candidate := candidate + 1;

goto beginning;

四．完全消除递归

原则上任意一个递归程序都可以转换为等价的非递归程序。

方法：

用类似编译中运行时存储组织那样，用户自己定义一个活动记录栈，栈里的每一个活动记录存放一个过程每一次运行时，其所有变量的现行值，参数的现行值以及该过程的返回地址。每当调用一个过程，就为该过程的本次运行在栈顶建立一个新的活动记录，当该过程返回时，栈顶的相应活动记录就撤消。

上例中，我们为过程knapsack建立一个状态栈，非递归过程如下：

PROCEDURE Knapsack (target : integer);

VAR

candidate : integer;

winflag : boolean;

S : STACK;

BEGIN

candidate := 1;

winflag := false;

MAKENULL(S);

Push(none, S);

REPEAT

IF winflag THEN

BEGIN /\*pop stack, printing weights included in solution\*/

IF Top(S) = included THEN

WRITELN(weights[candidate]);

candidate := candidate1;

Pop(S)

END

ELSE IF target = 0 THEN /\*solution found\*/

BEGIN

winflag := true;

candidate := candidate1;

Pop(S)

END

ELSE IF (target < 0) OR (candidate > n) THEN

BEGIN /\*no solution possible with choice made\*/

candidate := candidate1;

Pop(S)；

IF Top(S) = included THEN

target := target + weights[candidate]

END

ELSE IF Top(S) = none THEN

BEGIN /\*first try including candidate\*/

target := target weights[candidate];

candidate := candidate + 1;

Pop(S);

Push(included, S);

Push(none, S)

END

ELSE IF Top(S) = included THEN

BEGIN /\*try excluding candidate\*/

IF candidate = 1 THEN

target := target weights[candidate];

candidate := candidate + 1;

Pop(S);

Push(excluded, S);

Push(none, S)

END

ELSE BEGIN /\*Top(S) = excluded; give up on current choice\*/

Pop(S);

candidate := candidate1

END

UNTIL Empty(S);

END;

第八章 平摊分析

在平摊分析中，对一个数据结构执行一系列操作所花的时间被平均地分配在每个操作上，从而证明每个操作的平均时间代价很小，即使某个操作代价可能较大。

平摊分析与平均时间复杂度不同，它不需要假设每一个操作的概率，然后求数学期望值。

§8.1 合计方法

在平摊分析的合计方法中，我们证明：对任意n，一系列n个操作在最坏情况下一共需要T(n)时间，因此每个操作的平均代价或平摊代价是T(n)/n。

以下用两个例子说明。

一．栈操作

栈有两种基本操作：

1. Push(S, x) : 将元素x压入栈S中。
2. Pop(S) : 将栈顶元素从S中弹出，并返回该元素。

这两种操作每一个只需O(1)时间。执行n个这两种操作需时间。

假设我们还有一种操作：

1. Multipop(S, k) : 将S中弹出k个元素，如果S中元素个数小于k个，则将S弹空为止。

它的算法如下：

Multipop(S, k)

1. while not Empty(S) and (k 0) do

BEGIN

1. Pop(S);
2. k := k1;

END;

假设S中实际上保存了s个元素，那么Multipop(S, k)的实际执行时间为O(min{s, k})。

1. 一系列n个操作的分析

假设初始栈S为空，执行一系列n个Push, Pop和Multipop操作，最坏运行时间是多少？

因为Multipop最坏情况下需O(n)时间，故执行一系列n个Push, Pop，Multipop需O()时间。

用合计方法，我们可以得到更好的上界。

因为初始栈为空，每个元素被压入栈一次，它只能被弹出栈一次。因为n个Push, Pop和Multipop操作，至多将n个元素压入栈，因此，至多弹出n个元素出栈。因此，时间复杂度为O(n)。而平均每个操作需O(n)/n = O(1)时间。

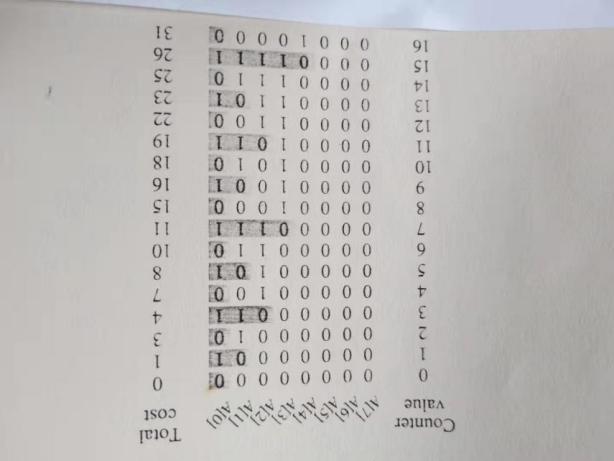
二．二进制增量计数器

设有一个k位的二进制计数器，用数组A[]表示，其中length[A] = k。设最低位是A[0]，最高位是。一个二进制数

保存在A中，则。

假设初始时，每一次的值加1，一直加到n。其时间复杂性用A中的二进制位变化的次数之和来表示。

例如：A中共有8位，从0加到16的计算时间和过程见图8.1。

图8.1

A中的值每增加1，用以下算法求得。

Increment(A)

1. ;
2. while do

BEGIN

1. ;
2. ;

END;

1. if then
2. ;
3. 粗略分析

在上述算法中，2-4行循环，我们要在位置加1，如果这时，则要将改为0，在下一次循环时，在中加1，重复上述过程，直到且，这时将置为1，算法结束。

最坏情况下，A中有k位变化，Increment(A)操作需时间。A中初始值为0，做n次加1操作，共需时间。

但这个分析不精确。

1. 精确分析

实际上做n次加1操作只需O(n)时间。

由图8.1可以看出，每次加1，A[0]位都会变化，故变化了次，而位，每两次加1，变化1次，故变化了次，再看A[2]位，每4次加1，变化1次，共变化了次，，变化了次。对于，不会变化。故所有位的变化次数之和为：

从而当A初值为0，做n次加1，加到n，所有位变化次数之和为O(n)，而每次加1的时间复杂度为O(n)/n = O(1)。