

**常用算法设计方法**

适用班级：软件设计师

主讲：邓少勋

网址：[www.bitpx.com](http://www.bitpx.com)

E-Mail:bitpx@163.com

分值说明：早上考2-5分,下午考15分

比特培训中心

贵州·贵阳

[第1节 计算机算法概述 1](#_Toc398816256)

[1.1 算法的五个特性 1](#_Toc398816257)

[1.2 算法设计的要求 1](#_Toc398816258)

[1.3 算法效率的度量 1](#_Toc398816259)

[第2节 各种常规算法 2](#_Toc398816260)

[2.1 迭代法 2](#_Toc398816261)

[2.2 穷举搜索法 3](#_Toc398816262)

[2.3 递推法 3](#_Toc398816263)

[2.4 递归法 3](#_Toc398816264)

[2.5 分治法 4](#_Toc398816265)

[2.5.1 分治法思想 4](#_Toc398816266)

[2.5.2 分治法时间复杂度计算 5](#_Toc398816267)

[2.6 动态规划法 7](#_Toc398816268)

[2.7 回溯法 9](#_Toc398816269)

[2.8 贪心法 10](#_Toc398816270)

[2.9 分支限界法 11](#_Toc398816271)

[2.10 概率算法 11](#_Toc398816272)

[2.11 字符串的模式匹配 11](#_Toc398816273)

[第3节 附录部分 12](#_Toc398816274)

[3.1 使用递推法求n的阶乘程序代码 12](#_Toc398816275)

# 计算机算法概述

计算机算法是对特定问题求解步骤的描述，它是指令的有限序列。为解决某问题的算法与为该问题编写的程序含义是相同的。

常用的表示算法的语言有：自然语言、流程图、盒图、程序设计语言和伪代码。

## 算法的五个特性

#### 有限性：算法必须在执行有限条指令之后结束，每条指令执行的时间也必须是有限的。

#### 确定性：算法中每一条指令必须有确切的含义，读者和计算机在理解时不会产生二义性，并且在相同条件下，相同的输入只能得到相同的输出。

#### 可行性：算法能把问题真正的解决。即不能是理论正确但无法在计算机上实现的算法。

#### 输入：一个算法有零个或多个输入。

#### 输出：一个算法有一个或多个输出。

●(2004年下)下面的程序段违反了算法的 (54) 原则。

void sam()

{ int n=2；

while(!odd(n))

n+=2；

printf(n)；

}

(54)A.有穷性 B.确定性 C.可行性 D.健壮性

## 算法设计的要求

#### 正确性：算法应当满足具体问题的需求。

#### 可读性：算法应该能让人读懂，能被计算机运行。

#### 健壮性：算法应该具有容错处理能力，不容易被击垮。

#### 高效率与低存储量要求：效率指程序的执行时间（越短越好），算法要占用计算机一定的存储量（越小越好）。

## 算法效率的度量

#### 时间复杂度

根据不同的输入，将算法的时间复杂度分为三种情况：

##### 最佳情况：使算法执行时间最少的输入。一般不进行算法在最佳情况下的时间复杂度分析。

##### 最坏情况：使算法执行时间最多的输入。一般会进行算法在最坏时间复杂度的分析，因为最坏情况是在任何输入下运行时间的一个上限，而且对于某些算法来说，最坏情况是相当频繁的。

##### 平均情况：算法的平均运行时间，可按三个步骤进行分析：将所有的输入按其执行时间分类；确定每类输入发生的概率；确定每类输入的执行时间。

##### 时间复杂度的表示

算法时间复杂度符号O、Ω、Θ的定义分别如下。

* O记号：给出一个函数的渐进上界。给定一个函数g(n)，O(g(n))表示为一个函数集合的{f(n)：存在正常数c和n0，使得对所有的n≥n0，有O≤f(n)≤c·g(n)}。
* Ω记号：给出一个函数的渐进下界。给定一个函数g(n),Ω(g(n))表示为一个函数集合的{f(n)：存在正常数c和n0，使得对所有的n≥n0，有O≤c·g(n)≤f(n)。
* Θ记号：给出一个函数的渐进上界和下界，即渐进确界。给定一个函数g(n)，Θ(g(n))表示为一个函数集合{f(n)：存在正常数c1、c2和n0，使得对所有的n≥n0，有O≤c1·g(n)≤f(n)≤c2·g(n)}。

常用大O记法，假如一个程序的实际执行时间为T（n）= 2n3+n2+5，则T（n）=O（n3）。

##### 时间复杂度常用表示方法

* 大O记法:假如一个程序的实际执行时间为T（n）= 2n3+n2+5，则T（n）=O（n3），只用最高的次幂表示。
* O(1)：当算法耗费的时间没有达到n这个数据级别时，就使用O(1)表示（其中的“1”并非表示数量1，而表示时间未至n这个数量级）。

常见的时间复杂度如图 1.1所示。



图 1.1 常见时间复杂度的比较

●(2012年上)以下关于渐进符号的表示中，不正确的是 (62) 。

(62)A.n2= (n2) B.n2=O(n2) C.n2=O(n) D.n2=O(n3)

●(2004年下)下面函数中渐进时间最小的是(53)。

(53)A．T1(n)=n+nlogn　B．T2(n)=2n+nlogn 　 C．T3(n)=n2-logn D．T4(n)=n+100logn

#### 空间复杂度

一个算法的空间复杂度是指程序运行从开始到结束所需的存储量。主要包含算法自身实现所需要的辅助存储空间，不包含程序中原存储数据的空间。

* 固定空间：在硬盘上的存储空间。
* 可变空间：程序运行时占用的内存空间。

●(2007年下)关于算法与数据结构的关系， （64）是正确的。

A. 算法的实现依赖于数据结构的设计 B. 算法的效率与数据结构无关

C. 数据结构越复杂，算法的效率越高 D. 数据结构越简单，算法的效率越高

●(2014年上)某个算法的时间复杂度递归式T(n)=T(n-l)+n，其中n为问题的规模，则该算法的渐进时间复杂度为（62），若问题的规模增加了16倍，则运行时间增加（63）倍。

（62）A.Θ(n) B.Θ(nlgn) C.Θ(n2) D.Θ(n2lgn)

（63）A.16 B.64 C.256 D.1024

●(2006年上)设某算法的计算时间可用递推关系式T(n)=2T(n/2)+n表示，则该算法的时间复杂度为(59)。

A．O(lgn) B．O(nlgn) C．O(n) D．O(n2)

●(2008年下)设某算法的计算时间表示为递推关系式T(n)= T(n-1)+ n(n>0)及T(0)=1，则该算法的时间复杂度为（65）。

（65）A.O(lgn) B.O(nlgn) C.O(n) D.O(n2)

●(2010年上)若某算法在问题规模为n时，其基本操作的重复次数可由下式表示，则该算法的时间复杂度为（64）。

T（n）=

（64）A.O(n) B.O(n2) C.O(logn) D.O(nlogn)

●(2009年下)某算法的时间复杂度表达式为T(n)=an2+bnlgn+cn+d，其中，n为问题规模，a、b、c和d为常数，用O表示其渐进意义时间复杂度为（63）。

（63）A.O(n2) B.O(n) C.O(nlgn) D. O(1)

●(2010年上)若对一个链表最常用的操作是在末尾插入结点和删除结点，则采用仅设尾指针的单向循环链表(不含头结点)时，（65）。

A.插入和删除操作的时间复杂度都为O(1)

B.插入和删除操作的时间复杂度都为O(n)

C.插入操作的时间复杂度为O(1)，删除操作的时间复杂度为O(n)

D.插入操作的时间复杂度为O(n)，删除操作的时间复杂度为O(1)

●(2011年下)对n个元素值分别为-1、0或1的整列数组A进行升序排序的算法描述如下：统计A中-1、0和1的个数，设分别为n1、n2和n3，然后将A中的前n1个元素赋值为-1，第n1+1到n1+n2个元素赋值为0，最后n3个元素赋值为1。该算法的时间复杂度和空间复杂度分别为（64） 。

**（64）A. B. C. D.**

●(2012年上)现要对n个实数（仅包含正实数和负实数）组成的数组A进行重新排列，使得其中所有的负实数都位于正实数之前。求解该问题的算法的伪代码如下所示，则该算法的时间和空间复杂度分别为 (65) 。

i=O；j=n-l；

while i<j do

while A[i]<0 do

i= i+l：

while A[j]>O do

j =j-l；

if i<j do

交换A[i]和A[j]

(65)A.  (n)和 (n) B． (1)和 (n) C.  (n)和 (1) D． (1)和 (1)

●(2013年上)采用顺序表和单链表存储长度为n的线性序列，根据序号查找元素，其时间复杂度分别为(51)。

(51) A.0(l)、0(l) B.0(l)、0(n) C.0(n)、0(l) D.00(n)、0(n)

● (53) 算法采用模拟生物进化的三个基本过程“繁殖(选择)→交叉(重组)→变异(突变)”。

(53)A.粒子群 B.人工神经网络 C.遗传 D.蚁群

# 各种常规算法

## 迭代法

思想：从某个点出发，通过某种方式求出下一个点，使得其离要求的点（方程的解）更近一步；当两点之差接近到可接受的精度范围时，就认为找到了问题的解。

注意：如果方程无解或者迭代方法不够适用，那么近似根序列将不会收敛，迭代过程会成为“死循环”。所以使用迭代法之前要进行是否有解的判断，并且限制迭代的次数；而方程即使有解，也要选择合适的迭代公式，否则迭代失败。

例子：用迭代法求x=sqrt(n)。根据牛顿迭代法，构造出求平方根的迭代公式为:x=(x+n/x)/2要求前后两次求出的x的差的绝对值小于10e-6。

#include<stdio.h>

double fabs(double n)

{ //求n的绝对值

if (n < 0) return -n;

return n;

}

void main()

{

int n;

double x0,x1;

scanf("%d",&n);

while (n>=0)

{

x0 = 1;

x1 = (x0+n/x0)/2;

while(fabs(x1-x0)>=0.000001)

{

x0 = x1;

x1 = (x0+n/x0)/2;

}

printf("%.3lf\n",x1);

scanf("%d",&n);

}

}

迭代法是用于求方程或方程组近似根的一种常用算法设计方法。

## 穷举搜索法

思想：对可能的众多候选解按某种顺序进行逐一枚举和检验，并从中找出那些符合要求的候选解作为问题的解。

特点：这种方法通常采用多重循环（循环次数取决于变量个数）来实现，对每个变量的每个值都测试是否满足所给定的条件，如果满足则认为找到问题的一个可行解。

缺点：只能解决候选解个数非常有限且容易枚举这些候选解的问题。

例子：找出n个自然数(1,2,3,…,n)中r个数的组合，要求r为3，且第一位数大于第二位数，第二位数大于第三位数。

程序：

#include<stdio.h>

void main()

{

int i,j,k,n;

printf("Please input n\n");

scanf("%d",&n);

for(i=n;i>0;i--)

for(j=n;j>0;j--)

for(k=n;k>0;k--)

if((i>j)&&(j>k))printf("%d %d %d\n",i,j,k);

}

## 递推法

思想：设要求解的问题规模为N，当N≤某个常数C时，解或为已知，或能非常方便地得到解。能采用递推法构造算法的问题有重要的递推性质：当得到问题规模为i-1的解后，由问题的性质，能从已求得的规模为1、2、3…、i-1的一系列解，构造出问题规模为i的解。这样，程序可从i=0或i=1出发，重复地，由已知的i-1规模的解，通过递推，获得规模为i的解，直至规模为n的解。

递推法典型用法是求整数的阶乘问题。求n！可转换为(n-1)!\*n，以下是使用递推法求解整数5的阶乘的算法描述：

##### 求出1的阶乘结果为1（1的阶乘结果是已知的）；

##### 根据1的阶乘结果递推2的阶乘，即2！=1！\*2;

##### 根据2的阶乘结果递推3的阶乘，即3！=2！\*3;

##### 根据3的阶乘结果递推4的阶乘，即4！=3！\*4;

##### 根据4的阶乘结果递推5的阶乘，即5！=4！\*5;

递推法特点：执行效率高，能解决有前向相关性的问题，但是要求前向相关顺序确定而且个数不多的问题。任何能使用递推法解决的问题，都能使用递归法解决，但反之不然。

## 递归法

递归描述的算法通常有这样的特征：为求解规模为N的问题，设法将它分解成规模较小的一些问题，然后从这些小问题的解方便地构造出大问题的解，并且这样规模较小的问题也能采用同样的分解和综合方法，分解成规模更小的问题，并从这些更小问题的解构造出其分解之前问题的解。特别地，当规模N≤C(C为常数)时，能直接得到解。

阶乘函数可递归地定义如图 2.1所示：

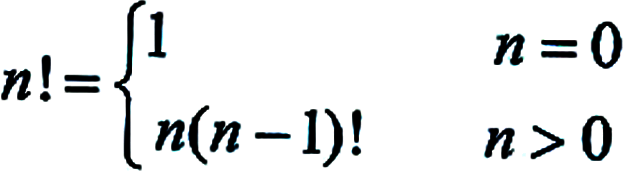


图 2.1 阶乘递归式

阶乘函数的自变量n的定义域是非负整数。递归式的第一式给出了这个函数的一个初始值，是非递归定义的。每个递归函数都必须有非递归定义的初始值，否则，递归函数就无法计算。递归式的第二式是用较小自变量的函数值来表示较大自变量的函数值的方式来定义n的阶乘。n！可以递归地计算如下：

int Factorial（int n）

{

if (n == 0) return 1;

if (n > 0) return n\*Factorial（n - 1）；

}

图 2.2是当n＝3的递归图：

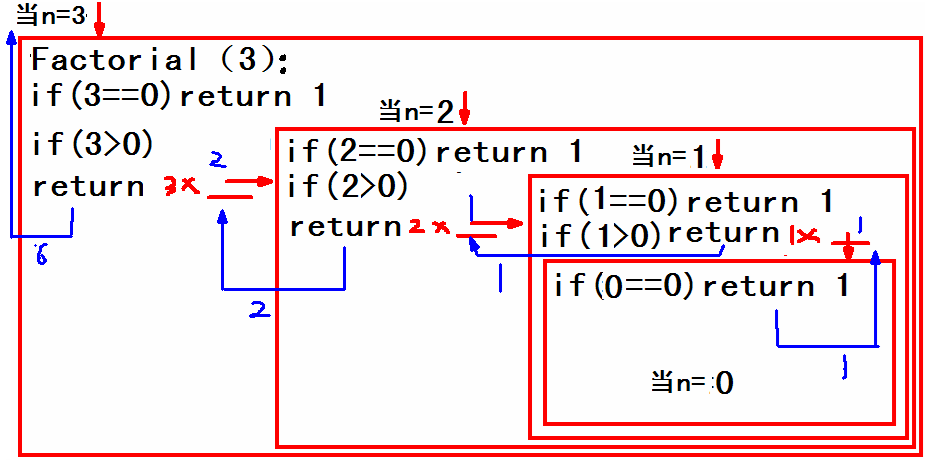


图 2.2 求3！的递归图

特点:由于递归程序需要调用工作栈，故其运行效率较低，无论是耗费的计算时间还是占用的存储空间都比非递归算法要多。递归程序分为递推和回归两个过程：

* 递推过程：把较复杂的问题的求解推到比原来问题简单一些的问题的求解；
* 回归阶段：当获得最简单情况的解后，逐级返回，依次获得稍复杂问题的解。

Fibonacci数列和背包问题等都是递归算法的典型应用。

递推法和递归法的关系：任何能使用递推法解决的问题，都能使用递归法解决，反之不成立。因为某些递归算法所要求的子问题的形式，不一定能够预先计算好结果，因此只能递归，无法递推。经典的例子是合并排序既能够递推，也能够递归，而快速排序无法递推，只能递归。

●(2007年上)若一个问题既可以用迭代方式也可以用递归方式求解，则（65）方法具有更高的时空效率。

A.迭代 B.递归 C.先递归后迭代 D.先迭代后递归

●(2005年下)设求解某问题的递归算法如下：

F(int n){

　if n==1 {

　　Move(1)

　}else{

　　F(n-1);

　　Move(n);

　　F(n-1);

　}

}

求解该算法的计算时间,仅考虑算法Move所做的计算为主要计算,且Move 为常数级算法。则算法F的计算时间T(n)的递推关系式为 (53) ；设算法Move的计算时间为k，当n=4时，算法F的计算时间为 (54) 。

（53）A.T(n)=T(n-1)+1 B.T(n)=2T(n-1) C.T(n)=2T(n-1)+1 D.T(n)=2T(n+1)+1

（54）A.14k B.15k C.16k D.17k

## 分治法

### 分治法思想

思想：将一个难以直接解决的大问题，分解成一些规模较小的相同问题，以便各个击破，分而治之。如果规模为n的问题可以分解成k个子问题，1<k<=n,且这些子问题互相独立且与原问题相同，然后递归求解这些子问题，最后将各子问题的解合并成原大问题的解。

Hanoi塔问题、比赛日程安排是该算法的典型应用场所。

例子：最大子段和问题。给定由n个整数(可能为负整数)组成的序列a1、a2、a3、…、an,求该序列形如的子段和的最大值。当所有整数均为负整数时定义其最大子段和为0，依次定义，所求的最优值为:,例如，当(a1, a2, a3, a4, a5, a6)=( -2,11,-4,13,-5,-2)时，最大子段和为：，如果将所给的序列a[1:n]分为长度相等的两端a[1:n/2]和a[n/2+1:n],分别求出这两端的最大子段和，则a[1:n]的最大子段和有3种情形：

##### a[1:n]的最大子段和与a[1:n/2]的最大子段和相同；

##### a[1:n]的最大子段和与a[n/2+1:n]的最大子段和相同；

##### a[1:n]的最大子段和为，且1<=i<= n/2, n/2+1<=j<= n；

(1)和(2)这两种情形可递归求得。对于情形（3），容易看出a[n/2]和a[n/2+1]在最优子序列中。容易得出情形（3）的最优值为Sum：

s1= s2=

Sum = s1+s2;

算法：

#include<stdio.h>

int MaxSubsum(int a[], int left, int right)

{

int sum = 0, center, LeftSum, RightSum;

int s1, lefts, s2, rights, i;

if (left == right)

sum = a[left] > 0 ? a[left] : 0;

else

{

center = (left + right) / 2;

//求出第(1)种情况的最大字段和

LeftSum = MaxSubsum(a, left, center);

//求出第(2)种情况的最大字段和

RightSum = MaxSubsum(a, center + 1, right);

//求出第(3)种情况的最大字段和

s1 = 0, lefts = 0;

for (i = center; i >= left; i--)

{

lefts += a[i];

if (lefts > s1)s1 = lefts;

}

s2 = 0, rights = 0;

for (i = center + 1; i <= right; i++)

{

rights += a[i];

if (rights > s2)s2 = rights;

}

sum = s1 + s2;//sum保存了第三种情况字段和

if (sum < LeftSum)sum = LeftSum;

if (sum < RightSum)sum = RightSum;

}

return sum;

}

void main()

{

int a[7] = { 0, -2, 11, -4, 13, -5, -2 }, sum;

sum = MaxSubsum(a, 1, 6);

printf("%d\n", sum);

}

### 分治法时间复杂度计算

一个分治法将规模为n的问题分成k个规模为n/m的子问题去解。用T(n)表示该分治法解规模为n的问题所需的计算时间，则有:

，通过迭代法求得方程解: 

●(2013年上)给定n个整数构成的数组｛a1,a2，…，an｝和整数x，判断A中是否存在两个元素ai和aj，使得ai+aj=x。为了求解该问题，首先用归并排序算法对数组A进行从小到大排序；然后判断是否存在ai十aj=x,具体的方法如下列伪代码所示。则求解该问题时排序算法应用了 (62) 算法设计策略，整个算法的时间复杂度为(63) 。

…

i=1；j=n

while i<j

if ai+aj=x return true

else if ai+aj>x

j--；

else

i++；

return false；

…

(62)A.分治 B.贪心 C.动态规划 D.回溯

(63)A.0(n) B.0(nlgn) C.O(n2) D.O(nlg2n)

●(2011年上)分治算法设计技术（63）。

（63）A.一般由三个步骤组成：问题划分、递归求解、合并解 B.一定是用递归技术来实现

C.将问题划分为K个规模相等的子问题 D.划分代价很小而合并代价很大

●(2011年下)设算法A的时间复杂度可用递归式表示，算法B的时间复杂度可用递归式表示，若要使得算法B渐进地快于算法A，则a的最大整数为 （65） 。

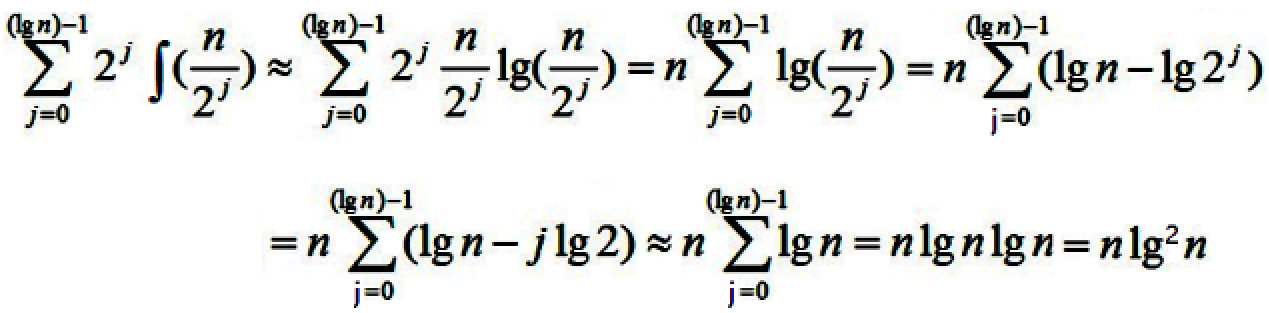
（65）A.48 B.49 C.13 D.14

●(2011年上)某算法的时间复杂度可用递归式表示，若用表示，则正确的是（64）。

（64）A. B. C. D.

●(2010年下)某算法的时间复杂度可用递归式T（n）=表示，若用Θ表示该算法的渐进时间复杂的紧致界，则正确的是（62）。

（62）A.Θ（nlg2n） B.Θ（nlgn） C.Θ（n2） D.Θ（n3）

解析：

●(2011年下)在有n个无序无重复元素值的数组中查找第i小的数的算法描述如下：任意取一个元素r，用划分操作确定其在数组中的位置，假设元素r为第k小的数。若i等于k，则返回该元素值；若i小于k，则在划分的前半部分递归进行划分操作找第i小的数；否则在划分的后半部分递归进行划分操作找第k-i小的数。该算法是一种基于 （63） 策略的算法。

（63）A.分治 B.动态规则 C.贪心 D.回溯

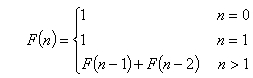
●(2009年上)归并排序采用的算法设计方法属于（65）。

（65）A.归纳法 B.分治法 C.贪心法 D.回溯方法

●(2009年上)现有16枚外形相同的硬币，其中有一枚比真币的重量轻的假币，若采用分治法找出这枚假币，至少比较（63）次才能够找出该假币。

（63）A.3 B.4 C.5 D.6

●(2008年上)斐波那契(Fibonacci)数列可以递归地定义为：



●用递归算法求解F(5)时需要执行 (63) 次“+”运算，该方法采用的算法策略是 (64) 。

(63)A. 5 B. 6 C. 7 D. 8

(64)A.动态规划 B.分治 C.回溯 D.分支限界

●(2008年上)给定一组长度为n的无序序列，将其存储在一维数组a[0..n-1]现采用如下方法找出其中的最大元素和最小元素：比较a[0]和a[n-1]，若a[0]较大，则将二者的值进行交换；再比较a[1]和a[n-2],若a[1]较大，则交换二者的值；然后依次比较a[2]和a[n-3]、a[3]和a[n-4]、...，使得每一对元素中的较小者被交换到低下标端。重复上述方法，在数组的前n/2个元素中查找最小元素，在后n/2个元素中查找最大元素。上述方法采用的算法设计策略是（64）。

（64）A.动态规划法 B.贪心法 C.分治法 D.回溯法

## 动态规划法

#### 基本思想

动态规划算法与分治法类似，其基本思想也是将待求解问题分解成若干个子问题，先求解子问题，然后从子问题的解中得到原问题的解。

与分治法不同的是，适合用于动态规划法求解的问题，经分解得到的子问题往往不是独立的，若用分治法来解决这类问题，则分解得到的重复子问题太多。

动态规划算法通常用于求解具有某种最优性质的问题，在这类问题中，可能会有许多可行解，每一个解都对应一个值，我们希望找到具有最优值（最大或最小）那个解，设计一个动态规划算法，通常可按以下几个步骤进行：

##### 找出最优解的性质，并刻画其结构特征

##### 递归的定义最优值；

##### 以自底向上的方式计算出最优值；

##### 根据计算最优值时得到的信息，构造一个最优解。

步骤（1）－（3）是动态规划算法的基本步骤，在只需要求出最优值的情形下步骤4可以省略，若需要求出问题的一个最优解，则必须执行步骤（4）。此时，在步骤3中计算最优值时，通常需记录更多的信息，以便在步骤4中根据所记录的信息，快速构造出一个最优解。

#### 动态规划适合解决的问题

##### 最优子结构：如果一个问题的最优解中包含了其子问题的最优解，就说该问题具有最优子结构。当一个问题具有最优子结构时，可考虑动态规划算法，但贪心策略也可能使用。

##### 重叠子问题：在用分治法求解时，有些子问题被重复求解许多次，如果我们能够保存已解决的子问题的解，而在需要时再找出已求得的解，这样就可以避免大量的重复计算，从而得到更高效的算法。为了达到这个目的，可以用一个表来记录所有已解决的子问题的答案。不管该子问题以后是否被用到，只要它被计算过，就将其结果记录入表中，这就是动态规划算法的基本思想。

我们同样考虑最大子段和问题，设，即bi是a序列以ai结尾的子段和中，最大的那个子段和。

则原问题的解maxSum=max{bi}。

在求解bi的时候，若bi-1<=0，则以ai结尾的最大子段和不应该包含ai之前的元素，即bi=ai。否则，bi=ai+bi-1。

由上可得出动态规划递归式：

b[i]=max{b[i-1]+a[i],a[i]},a<=i<=n

sum=max{ max{b[i-1]+a[i],a[i]},sum}

C语言描述的算法为：

#include<stdio.h>

int MaxSum(int a[],int n,int &Start,int &End)

{

int sum = 0;

int \*b,t;

b=new int[n+1];b[0]=0;

for(int i=1;i<=n;i++)

{

if(b[i-1]>0)b[i]=b[i-1]+a[i];

else {b[i] = a[i];t=i;};

if(b[i]>sum){sum=b[i];Start=t;End=i;};

}

delete []b;

return sum;

}

void main()

{

int a[7]={0,-2,11,-4,13,-5,-2},sum,Start,End,i;

sum = MaxSum(a,6,Start,End);

for(i=Start;i<=End;i++)

{

printf("%d ",a[i]);

}

printf("\n%d\n",sum);

}

●(2010年下)用动态规划策略求解矩阵连乘问题M1\*M2\*M3\*M4，其中M1（20\*5）、M2（5\*35）、M3（35\*4）和M4（4\*25），则最优的计算次序为（63）。

（63）A.((M1\*M2)\*M3)\*M4 B.(M1\*M2)\*(M3\*M4) C.(M1\*(M2\*M3)) \*M4 D.M1\*（M2\*（M3\*M4））

●(2005年上)利用动态规划方法求解每对结点之间的最短路径问题(all pairs shortest path problem)时，设有向图G=<V,E>共有n个结点，结点编号1～n，设C是G的成本邻接矩阵，用Dk(i,j)即为图G 中结点i到j并且不经过编号比k还大的结点的最短路径的长度（Dn(i,j)即为图G中结点i到j的最短路径长度），则求解该问题的递推关系式为 (56) 。

（56）A.Dk(i,j)=Dk-1(i,j)+C(i,j) B.Dk(i,j)=min{Dk-1(i,j),Dk-1(i,j)+C(i,j)}

C.Dk (i,j)=Dk-1(i,k)+Dk-1(k,j) D.Dk(i,j)=min{Dk-1(i,j),Dk-1(i,k)+Dk-1(k,j)}

●(2004年下)采用动态规划策略求解问题的显著特征是满足最优性原理，其含义是 (52)。

(52)A．当前所做出的决策不会影响后面的决策 B．原问题的最优解包含其子问题的最优解

C．问题可以找到最优解，但利用贪心法不能找到最优解 D．每次决策必须是当前看来最优的决策才可以找到最优解

●(2006年上)考虑一个背包问题，共有n=5个物品，背包容量为W=10,物品的重量和价值分别为：w={2，2，6，5，4}，v={6，3，5，4，6}，求背包问题的最大装包价值。若此为0-1背包问题，分析该问题具有最优子结构，定义递归式为c(i,j)= ,

其中c(i，j)表示i个物品、容量为j的0-1背包问题的最大装包价值，最终要求解c(n,W)。

采用自底向上的动态规划方法求解，得到最大装包价值为 (62) ，算法的时间复杂度为 (63) 。

若此为部分背包问题，首先采用归并排序算法，根据物品的单位重量价值从大到小排序，然后依次将物品放入背包直至所有物品放入背包中或者背包再无容量，则得到的最大装包价值为 (64) ，算法的时间复杂度为(65) 。

(62)A.11 B.14 C.15 D.16.67

(63)A.Θ(nW) B.Θ(nlgn) C.Θ() D.Θ(nlgnW)

(64)A.11 B.14 C.15 D.16.67

(65)A.Θ(nW) B.Θ(nlgn) C.Θ() D.Θ(nlgnW)

●(2006年下)两个矩阵 和相乘，用基本的方法进行，则需要的乘法次数为m\*n\*p。多个矩阵相乘满足结合律。不同的乘法顺序所需要的乘法次数不同。考虑采用动态规划方法确定，,…，多个矩阵连乘的最优顺序，即所需要的乘法次数最少。最少乘法次数用m[i,j]表示，其递归式定义为：

m[i,j]=

其中，i, j和k为矩阵下标，矩阵序列中的维度为()\*，采用自底向上的方法实现该算法来确定n个矩阵相乘的顺序，其时间复杂度为\_(64)\_。若四个矩阵、、、相乘的维度序列为2、6、3、10、3，采用上述算法求解，则乘法次数为\_(65)\_。

(64)A.O() B.O(1gn) C.O() D. O(1gn)

(65)A.156 B.144 C.180 D.360

●(2007年下)求解两个长度为n的序列X和Y的一个最长公共子序列(如序列ABCBDAB和BDCABA的一个最长公共子序列为BCBA)可以采用多种计算方法。如可以采用蛮力法，对X的每一个子序列，判断其是否也是Y的子序列，最后求出最长的即可，该方法的时间复杂度为 (62) 。经分析发现该问题具有最优子结构，可以定义序列长度分别为i和j的两个序列X和Y的最长公共子序列的长度为c[ij]，如下式所示。

采用自底向上的方法实现该算法，则时间复杂度为 (63) 。

(62)A.O() B.O(1gn) C.O() D.O(n)

(63)A.O() B.O(1gn) C.O() D.O(n)

●(2018年11月)在一条笔直公路的一边有许多房子，现要安装消防栓，每个消防栓的覆盖范围远大于房子的面积，如下图所示。现求解能覆盖所有房子的最少消防栓数和安装方案(问题求解过程中，可将房子和消防栓均视为直线上的点)。



该问题求解算法的基本思路为:从左端的第一栋房子开始，在其右侧m米处安装一个消防栓，去掉被该消防栓覆盖的所有房子。在剩余的房子中重复上述操作，直到所有房子被覆盖。算法采用的设计策略为 (62) ;对应的时间复杂度为(63) 。

假设公路起点A的坐标为0，消防栓的覆盖范围(半径)为20米，10栋房子的坐标为(10, 20, 30, 35, 60, 80,160, 210, 260, 300)，单位为米。根据上述算法，共需要安装 (64) 个消防栓。以下关于该求解算法的叙述中，正确的是(65) 。

(62)A分治 B.动态规划 C.贪心 D.回溯

(63)A.O(lgn). B.O(n) C.O(nlgn) D.O(n2)

(64)A.4 B.5 C.6 D.7

(65)A.肯定可以求得问题的一个最优解 B.可以求得问题的所有最优解 C.对有些实例，可能得不到最优解 D.只能得到近似最优解

●(2019年5月)已知矩阵和相乘的时间复杂度为O(mnp)。矩阵相乘满足结合律，如三个矩阵A、B、C相乘的顺序可以是(A\*B)\*C也可以是A\*(B\*C)。不同的相乘顺序所需进行的乘法次数可能有很大的差别。因此确定n个矩阵相乘的最优计算顺序是一个非常重要的问题。已知确定n个矩阵....相乘的计算顺序具有最优子结构，即....的最优计算顺序包含其子问题....和.... (lkn)的最优计算顺序。

可以列出其递归式为：

其中，的维度为,m[i,j]表示....最优计算顺序的相乘次数。先采用自底向上的方法求n个矩阵相乘的最优计算顺序。则求解该问题的算法设计策略为 (62) 。算法的时间复杂度为 (63) ，空间复杂度为 (64) 。

给定一个实例，(....)=(20,15,4,10,20,25),最优计算顺序为 (65) 。

(62)A.分治法 B.动态规划法 C.贪心法 D.回溯法

(63)A.O() B. O(lgn) C.O() D.O()

(64)A.O() B. O(lgn) C.O() D.O()

(65)A.(((A1\*A2)\*A3)\*A4)\*A5 B. A1\*(A2\*(A3\*(A4\*A5))) C.((A1\*A2)\*A3)\*(A4\*A5) D.(A1\*A2)\*((A3\*A4)\*A5)

## 回溯法

* 有“通用的解题法”之称，也称为试探法，用它可以系统地搜索一个问题的所有解或任一解。
* 回溯法是一个既有系统性又带有跳跃性的搜索算法。它在包含问题所有解的解空间树中，按照深度优先的策略，从根结点出发搜索解空间树，算法搜索至解空间树的任一节点时，总是先判断该点是否肯定不包含问题的解。如果肯定不包含，则跳过以该节点为根的子树的系统搜索，逐层向其祖先节点回溯，否则，进入该子树，继续按深度优先的策略进行搜索。回溯法在求问题的所有解时，要回溯到根，且根节点的所有子树都已被搜索遍才结束。而在求解任一解时，只要搜索到任一解就可以结束。这种以深度优先的方式系统地搜索问题的解的算法称为回溯法，它适用于解一些组合数较大的问题。

#### 问题的解空间

应用回溯法解问题时，首先应明确定义问题的解空间。例如，对于有n种可选物品的0-1背包问题，其解空间由长度为n的0-1向量组成。该解空间包含了对变量的所有可能的0-1赋值。当n＝3时，其解空间为：

{(0,0,0),(0,0,1),(0,1,0),(0,1,1),(1,0,0),(1,0,1),(1,1,0),(1,1,1)}

此解空间对应以下完全二叉树如图 2.3所示：

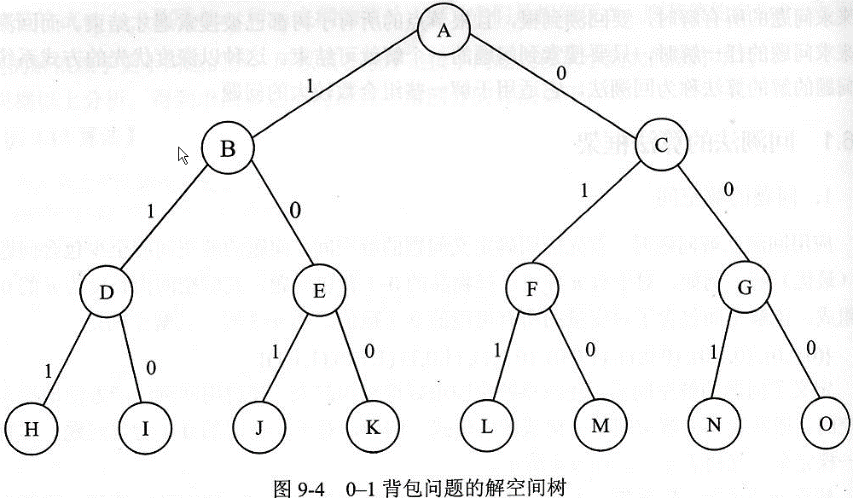


图 2.3 0-1背包问题的解空间树

#### 回溯法的基本思想

确定了解空间的组织结构后，回溯法就从开始节点（根节点）出发，以深度优先的方式搜索整个解空间。这个开始节点就成为一个活节点，同时也成为当前的扩展节点。在当前的扩展节点处，搜索向纵深方向移至一个新节点，这个新节点就成为一个新的活节点，并成为当前扩展节点。如果在当前扩展节点处不能再向纵深方向移动，则当前的扩展节点就成为死节点。换句话说，这个节点不再是一个活节点。此时，应往回移动（回溯）至最近的一个活节点处，并使这个节点成为当前的扩展节点。回溯法即以这种工作方式递归地在解空间中搜索，直至找到所要求的解或解空间中已无活节点时为止。

特例：用回溯法解决0-1背包问题

问题描述：对于n＝3时的0-1背包问题，考虑下面的具体实例：

w={16,15,15} p={45,25,25},c=30。其中w表示物品的重量，p表示物品的价值，c表示背包能够容纳的最大重量，请问怎么进行物品装包，使得背包没有被撑破，而又获得最大的价值？

首先从根节点开始搜索其解空间。具体过程看课堂演示！！

运用回溯法解题通常包含以下3个步骤：

##### 针对所给问题，定义问题的解空间

##### 确定易于搜索的解空间结构

##### 以深度优先的方式搜索解空间

##### 回溯法搜遍整个解空间，最后得到最优解。

N皇后问题也是回溯方法的典型应用。

注意：背包问题和0-1背包问题是不同的，0-1背包问题表示一个物体要么不放进背包，要么整体放进背包，物体不能切割；而背包问题可将物理切割后放进背包。

●(2013年下)在求解某问题时，经过分析发现该问题具有最优子结构性质，求解过程中子问题被重复求解，则采用（64）算法设计策略，以深度优先的方法搜索解空间，则采用（65）算法设计策略。

（64）A.分治 B.动态规则 C.贪心 D.回溯

（65）A.动态规则 B.贪心 C.回溯 D.分治限界

●(2011年上)要在8\*8的棋盘上摆放8个“皇后”，要求“皇后”之间不能发生冲突，即任何两个“皇后”不能在同一行、同一列和相同的对角线上，则一般采用 （62）来实现。

（62）A.分治法 B.动态规划法 C.贪心法 D.回溯法

●(2009年上)以下的算法设计方法中，（64）以获取问题最优解为目标。

（64）A.回溯方法 B.分治法 C.动态规划 D.递推

●(2018年上)现需要申请一些场地举办一批活动，每个活动有开始时间和结束时间。在同一个场地，如果一个活动结束之前，另一个活动开始，即两个活动冲突。若活动A从1时间开始，5时间结束，活动B从5时间开始，8时间结束，则活动A和B不冲突。现要计算n个活动需要的最少场地数。

求解该问题的基本思路如下(假设需要场地数为m，活动数为n，场地集合为，初始条件均无活动安排：

(1)采用快速排序算法对n个活动的开始时间从小到大排序，得到活动。对每个活动，i从1到n，重复步骤（2)、（3)和（4)；

(2)从开始，判断与的最后一个活动是否冲突，若冲突，考虑下一个场地,…；

(3)一旦发现与某个的最后一个活动不冲突，则将安排到,考虑下一个活动；

(4)若与所有已安排活动的的最后一个活动均冲突，则将安排到一个新的场地,考虑下一个活动；

(5)将n减去没有安排活动的场地数即可得到所用的最少场地数。

算法首先采用了快速排序算法进行排序，其算法设计策略是 (62) ;后面步骤采用的算法设计策略是 (63) 。整个算法的时间复杂度是 (64) 。下表给出了n=11的活动集合，根据上述算法，得到最少的场地数为 (65) 。

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| 开始时间 | 0 | 1 | 2 | 3 | 3 | 5 | 5 | 6 | 8 | 8 | 12 |
| 结束时间 | 6 | 4 | 13 | 5 | 8 | 7 | 9 | 10 | 11 | 12 | 14 |

(62)A.分治 B.动态规划 C.贪心 D.回溯

(63)A.分治 B.动态规划 C.贪心 D.回溯

(64)A.Θ(lgn) B.Θ(n) C.Θ(nlgn) D.Θ()

(65)A.4 B.5 C.6 D.7

## 贪心法

贪心法是一种不追求最优解，只希望得到较为满意的解的方法。贪心法一般可以快速得到满意的解，因为它省去了为找最优解要穷尽所有可能解而必须耗费的大量时间。

贪心法常以当前情况为基础作最优选择，而不考虑各种可能的整体情况，所以贪心法不要回溯。

例如平时购物找钱时，为使找回的零钱硬币最少，不考虑找零钱所有的各种方案，而是从最大面值的币种开始，按递减的顺序考虑各种币种，先尽量用大面值的币种，当不足大面值币种的金额时才去考虑下一种较小面值的币值。这就是采用贪心法，这种方法在这里总是最优，是因为银行对其发行的硬币种类和硬币面值的巧妙安排。如果只有面值分别为1，5和11单位的硬币，而希望找回总额为15单位的硬币。按贪心算法，应找1个11单位面值的硬币和4个1单位面值的硬币，共找回5个硬币。但最优的解答应是3个5单位面值的硬币。

装箱问题、马踏棋盘问题是贪心法的典型应用场合。背包问题可用贪心法，但是0-1背包问题不能用贪心法求最优解。

● 考虑下述背包问题的实例。有5件物品，背包容量为100，每件物品的价值和重量如下表所示，并已经按照物品的单位重量价值从大到小排好序。根据物品单位重量价值大优先的策略装入背包中，则采用了 (60) 设计策略。考虑0/1背包问题（每件物品或者全部装入背包或者不装入背包）和部分背包问题（物品可以部分装入背包），求解该实例得到的最大价值分别为 (61) 。

|  |  |  |
| --- | --- | --- |
| 物品编号 | 价值 | 重量 |
| 1 | 50 | 5 |
| 2 | 200 | 25 |
| 3 | 180 | 30 |
| 4 | 225 | 45 |
| 5 | 200 | 50 |

(60)A.分治 B.贪心 C.动态规划 D.回溯

(61)A.605和630 B.605和605 C.430和630 D.630和430

●(2012年上)某货车运输公司有一个中央仓库和n个运输目的地，每天要从中央仓库将货物运输到所有的运输目的地，到达每个运输目的地一次且仅一次，最后回到中央仓库。在两个地点i和j之间运输货物存在费用Cij。为求解旅行费用总和最小的运输路径，设计如下算法：首先选择离中央仓库最近的运输目的地1，然后选择离运输目的地1最近的运输目的地2，…，每次在未访问过的运输目的地中选择离当前运输目的地最近的运输目的地，最后回到中央仓库。则该算法采用了 (63) 算法设计策略，其时间复杂度为 (64) 。

(63)A．分治 B．动态规划 C．贪心 D．回溯

(64)A.(n2) B.  (n) C.  (nlgn) D.  (1)

●(2010年下)（65）不能保证求得0-1背包问题的最优解。

（65）A.分支限界法 B.贪心算法 C.回溯法 D.动态规划策略

●(2007年上)设商店有10元、5元、2元和1元的零币，每种零币的数量充足。售货员给顾客找零钱时，零币的数量越少越好。例如给顾客找零29元：先找2张10元币，然后选择1张5元币，再选择两张2元币。以上的找零钱方式采用了 (62) 策略。

A．分治 B.贪心 C.动态规划 D.回溯

●(2006年下)（58）算法策略与递归技术的联系最弱。

A.动态规划 B.贪心 C.回溯 D.分治

●(2005年下)利用贪心法求解0/1背包问题时， (55) 能够确保获得最优解。用动态规划方法求解0/1 背包问题时，将"用前i个物品来装容量是X的背包"的0/1背包问题记为KNAP(1,i,X),设fi(X)是KNAP(1,i,X)最优解的效益值，第j个物品的重量和放入背包后取得效益值分别为Wj和pj(j=1~n)。则依次求解f0(X) 、f1(X)、...、fn(X)的过程中使用的递推关系式为 (56) 。

（55）A.优先选取重量最小的物品 B.优先选取效益最大的物品 C.优先选取单位重量效益最大的物品 D.没有任何准则

（56）A.fi(X)=min{fi-1(X),fi-1(X)+pi} B.fi(X)=max{fi-1(X),fi-1(X-Wi)+pi}

C.fi(X)=min{fi-1(X-Wi),fi-1(X-Wi)+pi} D.fi(X)=max{fi-1(X-Wi),fi-1(X)+pi}

解析：KNAP(1,i,X)的最优值只能来源于以下两种情况：

* 第i个物品不装入背包，此时最优解的值就是子问题KNAP(1,i-1,X)的最优解的效益值，即为：fi-1（X）
* 第i个物品放入背包，此时最优解的值为第i个物品的效益值与子问题KNAP(1,i-1,X-Wi)的最优值之和，即为：fi-1（X-Wi）+Pi

●(2004年下)在下列算法设计方法中，(57)在求解问题的过程中并不从整体最优上加以考虑，而是做出在当前看来是最好的选择。利用该设计方法可以解决(58)问题。

(57)A．分治法 B．贪心法 C．动态规划方法　D．回溯法

(58)A．排序　　　　　B．检索　　　C．背包　　　　　D．0/1背包

## 分支限界法

分支限界法类似于回溯法，也是一种在问题的解空间树T上搜索问题解的算法。但在一般情况下，分支限界法与回溯法的求解目标不同。回溯法的求解目标是找出T中满足约束条件的所有解，而分支限界法的求解目标则是找出满足约束条件的一个解，或是在满足约束条件的解中找出使某一目标函数值达到极大或极小的解，即在某种意义下的最优解。

由于求解目标不同，导致分支界限法与回溯法在解空间树T上的搜索方式也不相同。回溯法以深度优先的方式搜索解空间树T，而分支限界法则以广度优先或以最小耗费优先的方式搜索解空间树T。分支限界法的搜索策略是，每一个活节点只有一次机会成为扩展节点。活节点一旦成为扩展节点，就一次性产生其所有儿子节点。在这些儿子节点中，凡是导致不可行解或非最优解的儿子节点被舍弃，其余儿子节点被加入活节点表中，此后，从活节点表中取下一节点成为当前扩展节点，并重复上述节点扩展过程。这个过程一直持续到所需的解或活节点表为空时为止。

分支限界法已被利用来解决了大量离散最优化的实际问题。

问题描述：对于n＝3时的0-1背包问题，考虑下面的具体实例：

w={16,15,15} p={45,25,25},c=30.其中w表示物品的重量，p表示物品的价值，c表示背包能够容纳的最大重量，请问怎么进行物品装包，使得背包没有被撑破，而又获得最大的价值？

首先从根节点开始搜索其解空间。具体看课堂演示！！

●(2004年下)在分支-限界算法设计策略中，通常采用(56)搜索问题的解空间。

(56)A．深度优先　B．广度优先 C．自底向上　D．拓扑序列

## 概率算法

概率算法的一个基本特征是对所求解问题的同一实例用同一概率算法求解两次，可能得到完全不同的效果。这两次求解所需的时间甚至所得到的结果可能有相当大的差别。有四种概率算法：

#### 数值概率算法：常用于数值问题的求解。这种算法所得到的往往是近似解，且近似解的精度随计算时间的增加而不断提高。但多数情况下只能得到近似解（满意解），得不到精确解。

#### 蒙特卡罗算法：用于求问题的精确解，但这个解未必是正确的。求得正确解的概率依赖于算法所用的时间。算法所用的时间越多，得到正确解的概率越高。缺点是：无法判断所得到的解是否正确。

#### 拉斯维加斯算法：得出问题的正确解，此算法一旦得出结果，这个结果肯定是正确解。找到正确解的概率随着它所用的计算时间的增加而提高。

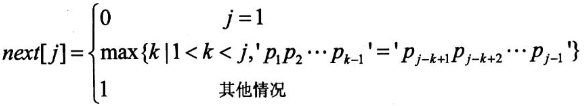
#### 舍伍德算法：总能求得问题的一个解，且所求得的解总是正确的。当一个确定性算法在最坏情况下的计算复杂度与其在平均情况下的计算复杂度有较大差别时，可在这个确定性算法中引入随机性将它改造成一个舍伍德算法，消除或减少问题的好坏实例间的这种差别。舍伍德算法的精髓不是避免算法的最坏情况行为，而是设法消除这种最坏情形行为与特定实例之间的关联性。

●(2004年下)拉斯维加斯(Las Vegas)算法是一种常用的(55)算法。

(55)A.确定性 B.近似　C.概率 D.加密

## 字符串的模式匹配

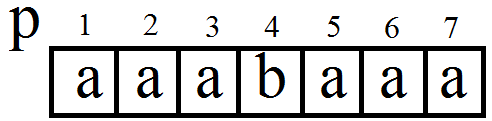
●(2012年上)在字符串的KMP模式匹配算法中，需要求解模式串p的next函数值，其定义如下所示。若模式串p为“aaabaaa”，则其next函数值为 (58) 。



(58)A.0123123 B.0123210 C.0123432 D.0123456

解析：

根据题意，字符串“aaabaaa”存在于名为p的数组中，存储单元从1开始编号，如图所示，其中j就是数组的存储单元的下标，值为1～7。



此题关键是理解next[j]=max{k|1<k<j, “p1p2…pk-1”=“pj-k+1pj-k+2…pj-1”}，next[j]的值其实是求一个满足条件1<k<j且“p1p2…pk-1”=“pj-k+1pj-k+2…pj-1”的k值集合中的最大一个数。本题的求解步骤如下：

#### 当j=1时，根据题意next[j]=next[1]=0;故next函数值为0；

#### 当j=2时，根据题意next[2]= max{k|1<k<2,“p1p2…pk-1”=“pj-k+1pj-k+2…pj-1”}，易知没有满足条件1<k<2的k的值，故只能next[2]=1(其他情况)；到此，next函数值为01(包含j=1时的next值)；

#### 当j=3时，根据题意next[3]=max{k|1<k<3,“p1p2…pk-1”=“p3-k+1p3-k+2…p3-1”}，易知满足条件1<k<3的k值为2。当k值为2时，“p1…p2-1”部分为“p1”，即字符串“a”；“p3-2+1…p3-1”部分为“p3-2+1…p3-1”为“p2”，即字符串“a”；此时可知当k=2时，满足1<k<3且“p1p2…pk-1”=“pj-k+1pj-k+2…pj-1”，则2可留在集合中，此时集合中只有2，故next[3]= max{2}=2。到此为止，next函数值为012(包含j=1、2时的next值)；

#### 当j=4时，根据题意next[4]=max{k|1<k<4,“p1p2…pk-1”=“p4-k+1p4-k+2…p4-1”}，易知满足条件1<k<4的k值为2、3。当k值为2时，“p1…p2-1”部分为“p1”，即字符串“a”；“p4-2+1…p4-1”部分为“p3…p3”为“p3”，即字符串“a”；此时可知当k=2时，满足1<k<4且“p1p2…pk-1”=“pj-k+1pj-k+2…pj-1”，则2可留在集合中；当k值为3时，“p1…p3-1”部分为“p1p2”，即字符串“aa”；“p4-3+1…p4-1”部分为“p2…p3”为“p2p3”，即字符串“aa”；此时可知当k=3时，满足1<k<4且“p1p2…pk-1”=“pj-k+1pj-k+2…pj-1”，则3可留在集合中；目前，集合中有两个数2与3，故next[4]=max{2,3}=3；到此，next函数值为0123(包含j=1、2、3时的next值)；

#### 当j=5时，根据题意next[5]=max{k|1<k<5,“p1p2…pk-1”=“p5-k+1p5-k+2…p5-1”}，易知满足条件1<k<5的k值为2、3、4。当k值为2时，“p1…p2-1”部分为“p1”，即字符串“a”；“p5-2+1…p5-1”部分为“p4…p4”为“p4”，即字符串“b”；此时可知当k=2时，满足1<k<5但不满足“p1p2…pk-1”=“pj-k+1pj-k+2…pj-1”条件，则2不能留在集合中；当k值为3时，“p1…p3-1”部分为“p1p2”，即字符串“aa”；“p5-3+1…p5-1”部分为“p3…p4”为“p3p4”，即字符串“ab”；此时可知当k=3时，满足1<k<5但不满足“p1p2…pk-1”=“pj-k+1pj-k+2…pj-1”条件，则3不能留在集合中；当k值为4时，“p1…p4-1”部分为“p1p2p3”，即字符串“aaa”；“p5-4+1…p5-1”部分为“p2…p4”为“p2p3p4”，即字符串“aab”；可知当k=4时，满足1<k<5但不满足“p1p2…pk-1”=“pj-k+1pj-k+2…pj-1”条件，则4不能留在集合中；到此，找不到满足1<k<5且“p1p2…pk-1”=“p5-k+1p5-k+2…p5-1”条件的k值，故next[5]为1(其他情况)，next函数值为01231(包含j=1、2、3、4时的next值)；

#### 当j=6时，根据题意next[6]=max{k|1<k<6,“p1p2…pk-1”=“p6-k+1p6-k+2…p6-1”}，易知满足条件1<k<6的k值为2、3、4、5。当k值为2时，“p1…p2-1”部分为“p1”，即字符串“a”；“p6-2+1…p6-1”部分为“p5…p5”为“p5”，即字符串“a”；此时可知当k=2时，满足1<k<6和“p1p2…pk-1”=“pj-k+1pj-k+2…pj-1”条件，则2留在集合中；当k值为3时，“p1…p3-1”部分为“p1p2”，即字符串“aa”；“p6-3+1…p6-1”部分为“p4…p5”为“p4p5”，即字符串“ba”；此时可知当k=3时，满足1<k<6但不满足“p1p2…pk-1”=“pj-k+1pj-k+2…pj-1”条件，则3不能留在集合中；当k值为4时，“p1…p4-1”部分为“p1p2p3”，即字符串“aaa”；“p6-4+1…p6-1”部分为“p3…p4”为“p3p4p5”，即字符串“aba”；可知当k=4时，满足1<k<6但不满足“p1p2…pk-1”=“pj-k+1pj-k+2…pj-1”条件，则4不能留在集合中；当k值为5时，“p1…p5-1”部分为“p1p2p3p3”，即字符串“aaab”；“p6-5+1…p6-1”部分为“p2…p5”为“p2p3p4p5”，即字符串“aaba”；可知当k=5时，满足1<k<6但不满足“p1p2…pk-1”=“pj-k+1pj-k+2…pj-1”条件，则5不能留在集合中；到此，集合中有一个数2，故next[6]=max{2}=2；则next函数值为012312(包含j=1、2、3、4、5时的next值)；

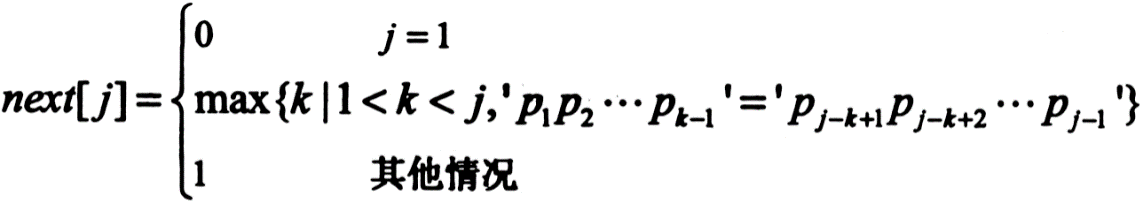
#### 当j=7时，根据题意next[7]=max{k|1<k<7,“p1p2…pk-1”=“p7-k+1p7-k+2…p7-1”}，易知满足条件1<k<7的k值为2、3、4、5、6。当k值为2时，“p1…p2-1”部分为“p1”，即字符串“a”；“p7-2+1…p7-1”部分为“p6…p6”为“p6”，即字符串“a”；此时可知当k=2时，满足1<k<7和“p1p2…pk-1”=“pj-k+1pj-k+2…pj-1”条件，则2留在集合中；当k值为3时，“p1…p3-1”部分为“p1p2”，即字符串“aa”；“p7-3+1…p7-1”部分为“p5…p6”为“p5p6”，即字符串“aa”；此时可知当k=3时，满足1<k<7和“p1p2…pk-1”=“pj-k+1pj-k+2…pj-1”条件，则3留在集合中；当k值为4时，“p1…p4-1”部分为“p1p2p3”，即字符串“aaa”；“p7-4+1…p7-1”部分为“p4…p5”为“p4p5p6”，即字符串“baa”；可知当k=4时，满足1<k<7但不满足“p1p2…pk-1”=“pj-k+1pj-k+2…pj-1”条件，则4不能留在集合中；当k值为5时，“p1…p5-1”部分为“p1p2p3p4”，即字符串“aaab”；“p7-5+1…p7-1”部分为“p3…p6”为“p3p4p5p6”，即字符串“abaa”；可知当k=5时，满足1<k<7但不满足“p1p2…pk-1”=“pj-k+1pj-k+2…pj-1”条件，则5不能留在集合中；当k值为6时，“p1…p6-1”部分为“p1p2p3p4p5”，即字符串“aaaba”；“p7-6+1…p7-1”部分为“p2…p6”为“p2p3p4p5p6”，即字符串“aabaa”；可知当k=6时，满足1<k<7但不满足“p1p2…pk-1”=“pj-k+1pj-k+2…pj-1”条件，则6不能留在集合中；到此，集合中有两个数2与3，故next[7]=max{2，3}=3；则next函数值为0123123(包含j=1、2、3、4、5、6时的next值)。

到此，j的值已经枚举完了数组p的所有单元下标，最后next的值为0123123，本题选择A选项。

●(2012年下)在字符串的模式匹配过程中，如果模式串的每个字符依次和主串中一个连续的字符序列相等，则称为匹配成功。如果不能在主串中找到与模式串相同的子串，则称为匹配失败。在布鲁特—福斯模式匹配算法（朴素的或基本的模式匹配）中，若主串和模式串的长度分别为n和m（且n远大于m），且恰好在主串末尾的m个字符处匹配成功，则在上述的模式匹配过程中，字符的比较次数最多为 (57)

(57)A．n\*m B．(n-m+1)\*m C．(n-m-1)\*m D．(n-m)\*n

●(2011年下)在KMP模式匹配算法中，需要求解模式串p的next的函数值，其定义如下（其中，j为模式串字符的序号）。对于模式串“abaabaca”，其next函数值序列为 （57） 。



（57）A.01111111 B.01122341 C.01234567 D.01122334

●在字符串的KMP模式匹配算法中，需先求解模式串的next函数值，其定义如下式所示，j表示模式串中字符的序号(从l开始)。若模式串p为“abaac”，则其next函数值为 (60) 。

(60)A.01234 B.01122 C.01211 D.01111

# 附录部分

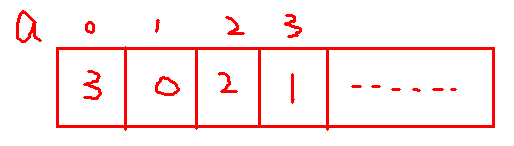
## 使用递推法求n的阶乘程序代码

例子：对给定的n(n<=100),计算并输出k的阶乘k!(k=1,2,3,…,n)的全部有效数字。

由于要求的整数可能大大超出一般整数的位数，程序用一维数组存储长整数，存储长整数数组的每个元素只存储长整数的一位数字。如有m位长整数N用数组a[]存储：

N=a[m]\*10m-1+a[m-1]\*10m-2+…+a[2]\*101+a[1]\*100

用a[0]存储长整数N的位数m，即a[0]=m。则数组的每个元素存储k的阶乘k!的一位数字，并从低位到高位依次存于数组的第二个元素，第三个元素…例如，5!=120,在数组中的存储形式为：



首元素表示长整数是一个3位数，接着是低位到高位依次是0、2、1，表示长整数是120.

计算阶乘k!可采用对已求的阶乘(k-1)!连续再累加k-1次后求出。例如，已知4！=24，计算5！，可对原来的24再累加4个24后得到120.

程序：

#include<stdio.h>

#include<malloc.h>

#define MAXN 100

void pnext(int a[],int k)

{

int \*b,m=a[0],i,j,r,carry;

b = (int \*)malloc(sizeof(int)\*(m+1));

for(i=1;i<=m;i++)b[i]=a[i];

for(j=1;j<k;j++)//保证k-1的阶乘累加k-1次

{

for(carry=0,i=1;i<=m;i++)

{//保证b数组中每位数(k-1的阶乘)都参与累加运算

r = (i<=a[0]?a[i]+b[i]:a[i])+carry;

a[i]=r%10;

carry=r/10;

}

if(carry)a[++m]=carry;

}

free(b);

a[0]=m;

}

void write(int \*a,int k)

{

int i;

printf("%4d!=",k);

for(i=a[0];i>0;i--)printf("%d",a[i]);

printf("\n\n");

}

void main()

{

int a[MAXN],n,k;

printf("Enter the number n:\n");

scanf("%d",&n);

a[0]=1;

a[1]=1;

write(a,1);

for(k=2;k<=n;k++)

{

pnext(a,k);

write(a,k);

}

}

运行结果：

