第 14 章：算法设计　　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

考点突破

根据考试大纲，本章要求考生掌握以下几个方面的知识点。

（1）数据结构设计：线性表、查找表、树、图的顺序存储结构和链表存储结构的设计和实现。

（2）算法设计：迭代、穷举搜索、递推、递归、回溯、贪心、动态规划、分治等算法设计。

从历年的考试情况来看，本章的考点主要集中以下方面。

在数据结构设计中，主要考查基本数据结构如栈，二叉树的常见操作代码实现。

在算法设计中，主要考查动态规划法、分治法、回溯法、递归法、贪心法。

上一节　本书简介　下一节

第 14 章：算法设计　　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

考点精讲

1. 算法特性

算法有一些基本特性要求掌握，表14-1是对这些特性的总结。

表14-1 算法的特性

特性	说明	违反示例

有穷性	必须总是在执行有穷步之后结束，且每一步都可在有穷时间内完成	void sam(){
		int n=2
		while (!odd(n))　n+=2;
		printf(n);}

确定性	算法中每一条指定都必须有确切的含义，而且只有唯一条执行路径，对相同的输入只能得出相同输出	有函数 f(x)： 当 x≤4 时，f(x)=0 当 x>3 时，f(x)=1 ---x=4 时，取值将不确定
输入	一个算法有 0 个或多个输入，以刻画对象的初始情况	即当算法本身未定出初始条件，而且还没有输入项
输出	一个算法有一个或多个输出，以反映对输入数据加工后的结果	没有输出的算法是没有意义的
可行性	一个算法是可行的，即算法中描述的操作都是可以通过已经实现的基本运算执行有限次来实现的	例如，打印出无限不循环小数的完整值，显然是不可行的。 ——因为它将是无限性

2. 算法复杂度分析

算法复杂性包括两个方面，一个是算法效率的度量（时间复杂度），一个是算法运行所需要的计算机资源量的度量（空间复杂度），这也是评价算法优劣的重要依据。

时间复杂度

一个程序的时间复杂度是指程序运行从开始到结束所需要的时间。通常分析时间复杂度的方法是从算法中选取一种对于所研究的问题来说是基本运算的操作，以该操作重复执行的次数作为算法的时间度量。一般来说，算法中原操作重复执行的次数是规模n的某个函数T(n)。由于许多情况下要精确计算T(n)是困难的，因此引入了渐进时间复杂度在数量上估计一个算法的执行时间。

我们通常使用“O()”来表示时间复杂度，其定义如下：

如果存在两个正常数 c 和 m，对于所有的 n，当 n≥m 时有 f(n)≤cg(n)，则有 f(n)=O(g(n))

也就是说，随着n的增大，f(n)渐进地不大于g(n)。例如，一个程序的实际执行时间为$T(n)=3n^3+2n^2+n$，则$T(n)=O(n^3)$。T(n)和n^3的值随n的增长渐近地靠拢。常见的渐进时间复杂度有：$O(1)<O(\log_2 n)<O(n)<O(n\log_2 n)<O(n^2)<O(n^3)<O(2^n)$。

下面以几个实例来说明具体程序段的时间复杂度。

例1：求以下程序段的时间复杂度。

```
temp=i;
i=j;
j=temp;
```

分析该程序段会发现，程序的功能是将i与j的值交换。程序一共3条语句，每条语句的执行次数都为1。即：T(n)=1+1+1，所以整个程序段的时间复杂度为O(1)。

例2：求以下程序段的时间复杂度。

```
sum=0;  //执行1次
for(i=1;i<=n;i++) //执行n次
for(j=1;j<=n;j++) //执行n^2次
sum++; //执行n^2次
```

本程序段的$T(n)=2n^2+n+1$，时间复杂度应取指数级别最高的，所以为$O(n^2)$。

空间复杂度

一个程序的空间复杂度是指程序运行从开始到结束所需的存储量。它通常包括固定部分和可变部分两个部分。

在算法的分析与设计中，经常会发现时间复杂度和空间复杂度之间有着微妙的关系，经常可以相互转换，也就是可以利用空间来换时间，也可以用时间来换空间。

渐近符号

前面讲到时间复杂度时，已经提到了“通常我们使用O()来表示时间复杂度”，但在考试时，有时会出现Θ符号，所以在此介绍一下常用的三种渐近符号在算法复杂度中所代表的含义。

O（f（n）），给出了算法运行时间的上界，一般用来表达最坏情况下的时间复杂度，这也是平时最常见的一种表达表式；

Ω（f（n）），给出了算法运行时间的下界，一般用来表达最好情况下的时间复杂度；

Θ（f（n）），给出了算法运行时间的上界和下界，其实并不是所有的算法都能求出Θ（f（n））的。

第 14 章：算法设计 作者：希赛教育软考学院 来源：希赛网 2014年05月06日

一点一练

试题1

以下关于渐近符号的表示中，不正确的是__(1)__。

（1）A．$n^2=\Theta(n^2)$ B．$n^2=O(n^2)$ C．$n^2=O(n)$ D．$n^3=O(n^3)$

试题2

某算法的时间复杂度可用递归式 $T(n)=\begin{cases}O(1) & ,n=1\\ 2T(n/2)+n\lg n & ,n>1\end{cases}$ 表示，若用 Θ 表示该算法的渐进时间复杂度的紧致界，则正确的是__(2)__。

（2）A．$\Theta(n\lg^2 n)$ B．$\Theta(n\lg n)$

C．$\Theta(n^2)$ D．$\Theta(n^3)$

试题3

某算法的时间复杂度可用递归式 $T(n)=\begin{cases}\Theta(1) & ,n=1\\ 6T(n/5)+n & ,n>1\end{cases}$ 表示，若用 Θ 表示，则正确的是__(3)__。

（3）A．$\Theta(n^{\log_5^6})$ B．$\Theta(n^2)$ C．$\Theta(n)$ D．$\Theta(n^{\log_6^5})$

第 14 章：算法设计 作者：希赛教育软考学院 来源：希赛网 2014年05月06日

解析与答案

试题1分析

C选项中的n^2与O(n)，明显不在一个数量级之上，所以不对等。

试题1答案

（1）C

试题2分析

在本题中，我们关键要理解算法的渐进紧致界的概念，举个例子来说吧，假设当N>N0时，函数f(N)在一个常数因子范围内等于g(N)，则称g(n)是f(n)的一个渐近紧致界。

$$T(n)=2T(n/2)+n\lg n$$
$$=2(2T(n/4)+\frac{n}{2}\lg\frac{n}{2})+n\lg n$$
$$=4T(n/4)+n\lg\frac{n}{2}+n\lg n$$
$$=\ldots=n(\lg n+\lg\frac{n}{2}+\lg\frac{n}{4}+\ldots)+1$$

本题中给出的递归式的渐进紧致界应该是A。

试题2答案

（2）A

试题3分析

本题题型与试题2完全一致，推导过程请参看试题2，推导结果为： $\Theta(n^{\log_3^6})$ 。

试题3答案

（3）A

上一节　本书简介　下一节

查找与排序

查找与排序是软件设计师考试中常考的知识点，不仅在上午综合知识部分考查，下午软件设计部分也会涉及。

查找与排序都会涉及到一些算法，例如查找，最笨的方法就是顺序查找，即从头开始，逐一对比，直到找到目标为止，这样如果要找的元素比较靠后，则需要消耗大量时间。如果用折半查找，则可以快速找到目标。折半的方式为：一开始就跟目标序列中的中部元素对比，如果要找的值小于中部元素，则说明要找的元素在前半个队列中，如此一来，一次对比，实排除了一半的元素，所以很高效。本节将详细介绍这些查找与排序的算法。

上一节　本书简介　下一节

考点精讲

1. 顺序查找

顺序查找又称线性查找，它是最基本的查找技术，它的查找过程是：从表中的第一个记录开始，逐个进行记录的关键字和给定值比较，若某个记录的关键字和给定值相等，则查找成功，找到所查的记录；如果直到最后一个记录，都无匹配的，则查找失败。

顺序查找方法既适用于线性表的顺序存储结构，也适用于线性表的链式存储结构。

成功时的顺序查找的平均查找长度如下：

$$\mathrm{ASL}=\sum_{i=1}^{n}p_i c_i=\sum_{i=1}^{n}(n-i+1)=np_1+(n-1)p_2+\cdots+2p_{n-1}+p_n$$

在等概率情况下，$p_i=1/n(1\le i\le n)$，故成功的平均查找长度为$(n+\ldots+2+1)/n=(n+1)/2$，

即查找成功时的平均比较次数约为表长的一半。若k值不在表中，则需进行（n+1）次比较之后

才能确定查找失败（所以表中元素过多时，不宜采用该方法进行查找）。

该算法的时间复杂度为：O(n)。

2. 二分查找法

二分查找又称折半查找，优点是比较次数少，查找速度快，平均性能好；其缺点是要求待查表为有序表，且插入删除困难。因此，折半查找方法适用于不经常变动而查找频繁的有序列表。

下面具体来看一看二分法查找的具体操作流程：（设R[low,…,high]是当前的查找区间）

（1）确定该区间的中点位置：mid=[(low+high)/2]（注意：此处会进行取整操作，在考试时，这些细节往往直接影响得分）；

（2）将待查的k值与R[mid].key比较，若相等，则查找成功并返回此位置，否则需确定新的查找区间，继续二分查找，具体方法如下。

若R[mid].key>k，则由表的有序性可知R[mid,…,n].key均大于k，因此若表中存在关键字等于k的结点，则该结点必定是在位置mid左边的子表R[low,…,mid−1]中。因此，新的查找区间是左子表R[low,…,high]，其中high=mid−1。

若R[mid].key<k，则要查找的k必在mid的右子表R[mid+1,…,high]中，即新的查找区间是右子表R[low,…,high]，其中low=mid+1。

若R[mid].key=k，则查找成功，算法结束。

（3）下一次查找是针对新的查找区间进行，重复步骤（1）和（2）。

（4）在查找过程中，low逐步增加，而high逐步减少。如果high<low，则查找失败，算法结束。

二分查找法采用了分治法的思想，后面将详细说明分治法的工作方式。

3. 散列表

散列技术是在记录的存储位置和它的关键字之间建立一个确定的对应关系f，使得每个关键字key对应一个存储位置f(key)。采用散列技术将记录存储在一块连续的存储空间中，这块连续存储空间称为散列表或哈希表。

使用散列表技术使得查询能在O(1)时间内完成查找，这是非常难得的。那么他是怎么做到的呢？

简单一点讲就是：在存储时，我们会建立一个函数，利用函数算出存储空间，而取的时候，仍然用同样的方式来取出数据，这样就达到了目的。

下面通过一个实例来理解这样的过程。

例：将一个数列存储起来：52，37，28，41，79，85，93，64；以方便快速查询。我们可以定义函数：f(x)=x mod 10。即存储位置通过数据与10进行取余操作获取。所以，52应存放于2号空间，38存放于7号空间，依此类推。把所有数据存储好以后，若要查询85，只需要将85与10取余，得到5，再直接判断5号存储空间是否为85即可。若要查询99，则用99与10取余，得到9，然后判断9号空间是否为99，结果发现不是99而是79，所以查找失败，数列中没有我们需要的数据。

同时值得我们注意的是，并不是每个数据，我们都可以一次查找到，因为散列函数可能存在冲突。即两个不同的关键字，由于散列函数值相同，因而被映射到同一表位置上。例如，我们选取的散列函数为：X MOD 10。需要存储的数列为：1，8，11，4，5，9。此时，我们发现，1与11对10取余，余数均为1，此时，产生冲突。产生冲突时，通常有两种解决方案：

（1）线性探查法

该方法的思路很简单，当前空间已被占据，则选下一个空闲空间来存储。还是以上面的数据为例，当存储完数列中的：1、8，需要存储11时，发现冲突产生，此时将11存储于1号空间的下一个空闲位置，即2号空间。

（2）双散列函数法

双散列函数法的思想是采用多个散列函数，当产生冲突时，利用第2个函数再次进行地址计算，得到存储位置。

4. 插入排序

插入排序的基本思想是每步将一个待排序的记录按其排序码值的大小，插到前面已经排好的文件中的适当位置，直到全部插入完为止。插入排序方法主要有直接插入排序和希尔排序。

直接插入排序

直接插入排序的基本思想非常简单：每次从无序表中取出第一个元素，把它插入到有序表的合适位置，使有序表仍然有序。

第一趟比较前两个数，然后把这两个数按大小插入到有序表中；第二趟把第三个数据取出，然后依次与前两个数比较，把这个数按大小插入到有序表中；依次进行下去，进行了(n-1)趟以后就完成了整个排序过程。

希尔排序

希尔排序也称为缩小增量排序，它是在直接插入排序的基础上进行改进而得到的排序方法。其基本思想是：每一趟都按照确定的间隔（增量）将元素分组，在每一组内进行直接插入排序，使得小元素可以跳跃式地向前移动，以后逐步缩小增量值，直到增量值为1为止，这时序列中的元素也已经基本有序，再进行直接插入排序时就可以很快，如图14-1所示。

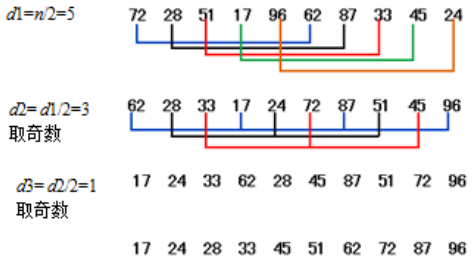


图14-1 希尔插入排序示意图

增量通常首先取d1=n/2（n为待排序的数值的总个数），di+1=di/2，如果值为偶数，则加1，以保证di为奇数。

5. 选择排序

选择排序的基本思想是每一步都从待排序的记录中选出排序码最小的记录，顺序存放在已排序的记录序列的后面。常见的选择排序有直接选择排序和堆排序两种。

直接选择排序

直接选择排序的过程是，首先在所有记录中选出排序码最小的记录，把它与第1个记录交换，然后在其余的记录内选出排序码最小的记录，与第2个记录交换……依次类推，直到所有记录排完为止。

无论文件初始状态如何，在第i趟排序中选出最小关键字的记录，需做n–i次比较，因此，总的比较次数为n(n–1)/2=O(n^2)。当初始文件为正序时，移动次数为0；文件初态为反序时，每趟排序均要执行交换操作，总的移动次数取最大值3(n–1)。直接选择排序的平均时间复杂度为O(n^2)。直接选择排序是不稳定的。

堆排序

堆排序是利用堆这一特殊的树形结构进行的选择排序，它有效地改进了直接选择排序，提高了算法的效率。堆排序的整个过程是：构造初始堆，将堆的根结点和最后一个结点交换，重新调整成堆，再交换，再调整，直到完成排序。

堆实际上就是一种特殊的完全二叉树，它采用顺序存储。如果从0开始对树的结点进行编号，编号的顺序按层进行，同层则按从左到右的次序；则编号为0～$\lfloor n/2 \rfloor -1$的结点为分支结点，编号大于$\lfloor n/2 \rfloor -1$的结点为叶结点，对于每个编号为i的分支结点，它的左子结点的编号为2i+1，右子结点的编号为2i+2。除编号为0的树根结点外，对于每个编号为i的结点，它的父结点的编号为$\lfloor (i-1)/2 \rfloor$。假定结点i中存放记录的排序码为S_i，则堆的各结点的排序码满足$S_i \geq S_{2i+1}$且$S_i \geq S_{2i+2}$（$0 \leq i \leq \lfloor n/2 \rfloor -1$）。这种堆称为大顶堆（大根堆），而如果相反（即满足$S_i \leq S_{2i+1}$且$S_i \leq S_{2i+2}$），则称为小顶堆（小根堆）。

构成堆就是把待排序的元素集合，根据堆的定义整成堆。这个过程需从对应的完全二叉树中编号最大的分支结点（编号为$\lfloor n/2 \rfloor -1$）起，至根结点（编号为0）止。依次对每个分支结点进行“渗透”运算，形成以该分支结点为根的堆，当最后对二叉树的根结点进行渗透运算后，整个二叉树就成为了堆。下面我们就以序列{42，13，24，91，23，16，05，88}为例，说明堆的构造过程（首先按层次遍历将序列生成对应的完全二叉树），如图14-2所示。

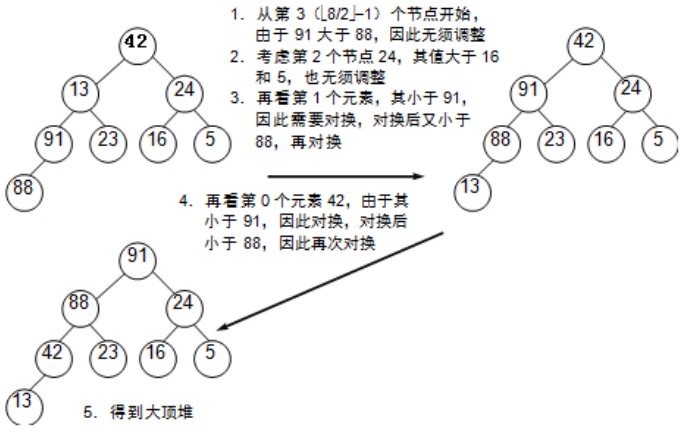


图14-2堆构造过程示意图

图14-2完成堆构造后，得到堆序列{91，88，24，42，23，16，5，13}。在历年的考试中，经常出现给出序列要求判断是否为堆。例如（10，50，80，30，60，20，15，18）是否为堆？我们应该从编号为$\lfloor n/2 \rfloor -1$的结点开始，逐个检查是否能够满足“$S_i \geq S_{2i+1}$且$S_i \geq S_{2i+2}$”或“$S_i \leq S_{2i+1}$且$S_i \leq S_{2i+2}$”。上例中，共有8个元素，因此从第4个开始。

序号	S_i	S_{2i+1}	比较结果	S_{2i+2}	比较结果
3	30	S_7，即 18	大于	无	无
2	80	S_5，即 20	大于	S_6，即 15	大于
1	50	S_3，即 30	大于	S_4，即 60	小于
0	10	S_1，即 50	小于	S_2，即 80	小于

显然，它并不满足定义，因此不是堆。

上面进行了堆构造的过程，接来下，我们来看如何来完成堆排序。

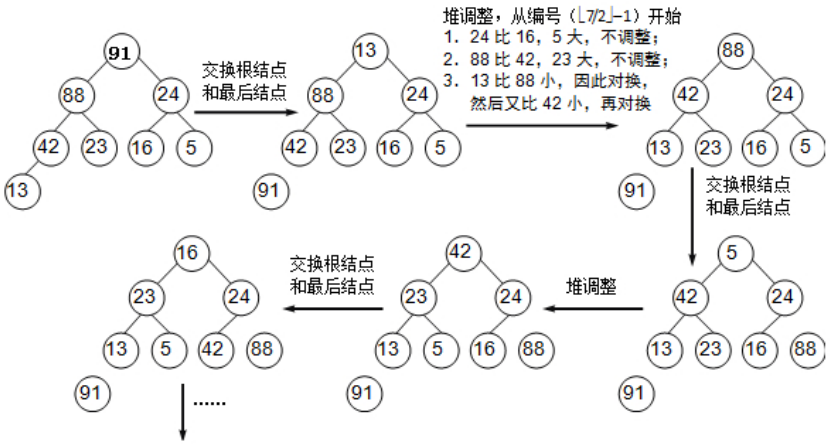


图14-3堆排序示意图

通过前面的论述，我们可以得知：构建初始堆需要花费比较长的时间，因此对于记录数较少的排序问题并不适合于应用堆排序。

6. 交换排序

交换排序的基本思想是：两两比较待排序记录的排序码，并交换不满足顺序要求的那些偶对，直到满足条件为止。交换排序的典型方法包括冒泡排序和快速排序。

冒泡排序

冒泡排序的基本思想是，通过相邻元素之间的比较和交换，将排序码较小的元素逐渐从底部移向顶部。由于整个排序的过程就像水底下的气泡一样逐渐向上冒，因此称为冒泡算法。

整个冒泡排序过程如下所述：首先将A[n-1]和A[n-2]元素进行比较，如果A[n-2]>A[n-1]，则交换位置，使小的元素上浮，大的元素下沉；当完成一趟排序后，A[0]就成为最小的元素；然后就从A[n-1]～A[1]之间进排序。下面就是一个实际的例子，如图14-4所示。

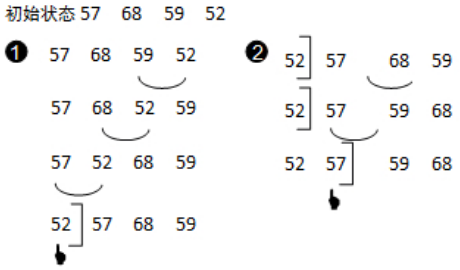


图14-4 冒泡排序示意图

6. 快速排序

快速排序采用的是分治法，其基本思想是将原问题分解成若干个规模更小但结构与原问题相似的子问题。通过递归地解决这些子问题，然后再将这些子问题的解组合成原问题的解。快速排序通常包括两个步骤：

第一步，在待排序的n个记录中任取一个记录，以该记录的排序码为准，将所有记录都分成两组，第1组都小于该数，第2组都大于该数，如图14-5所示。

第二步，采用相同的方法对左、右两组分别进行排序，直到所有记录都排到相应的位置为止。

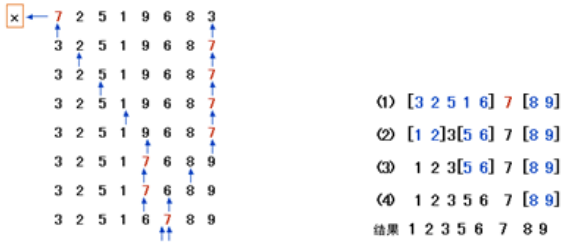


图14-5 快速排序示意图

7. 归并排序

归并也称为合并，是将两个或两个以上的有序子表合并成一个新的有序表。若将两个有序表合并成一个有序表，则称为二路合并。合并的过程是：比较A[i]和A[j]的排序码大小，若A[i]的排序码小于等于A[j]的排序码，则将第一个有序表中的元素A[i]复制到R[k]中，并令i和k分别加1；如此循环下去，直到其中一个有序表比较和复制完，然后再将另一个有序表的剩余元素复制到R中。

而归并排序就是使用合并操作完成排序的算法，如果利用二路合并操作，则称为二路合并排序，其过程如下：

首先把待排序区间中的每个元素都看做一个有序表（则有n个有序表），通过两两合并，生成$\lfloor n/2 \rfloor$个长度为2（最后一个表的长度可能小于2）的有序表，这也称为一趟合并。

然后再将这$\lfloor n/2 \rfloor$个有序表进行两两合并，生成$\lfloor \lfloor n/2 \rfloor 2 \rfloor$个长度为4的有序表。

如此循环直到得到一个长度为n的有序表，通常需要$\lfloor \log_2 n \rfloor$趟，如果该值为奇数，则为$\lfloor \log_2 n \rfloor + 1$。

基于磁盘进行的外排序经常使用归并排序的方法。其过程主要可以分为两个阶段。

建立外排序所有的内存缓冲区：根据它们的大小将输入的文件划分为若干段，用某种有效的内排序方法，对各段进行排序，这些经过排序的段称为初始归并段，生成后就写到外存中去。

使用归并树模式将第一个阶段生成的初始归并段加以归并，一趟趟地扩大归并段或减少归并段个数，直到最后归并成一个大的归并段为止。

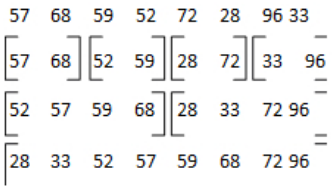


图14-6归并排序示意图

使用k路平衡归并时，如果有m个初始归并段，则相应的归并树就有$\lfloor \log_k m \rfloor + 1$层，需要归并$\lfloor \log_k m \rfloor$趟。例如，若对27个元素只进行三趟多路归并排序，则选取的归并路数是多少？这其实就是已知$\lfloor \log_k 27 \rfloor = 3$，求k值，很显然是3。

通常只需增加归并路数k，或减少初始归并段个数m，都能够减少归并趟数S，以减少读写磁盘的次数d，达到提高外排序速度的目的。

8. 基数排序

基数排序是一种借助多关键字排序思想对单逻辑关键字进行排序的方法。基数排序不是基于关键字比较的排序方法，它适合于元素很多而关键字较少的序列。基数的选择和关键字的分解是根据关键字的类型来决定的，例如关键字是十进制数，则按个位、十位来分解。

例如，我们需对{135，242，192，93，345，11，24，19}进行排序，因为数据的最高位是百位，所以要分三趟进行分配和收集，如图14-7所示。

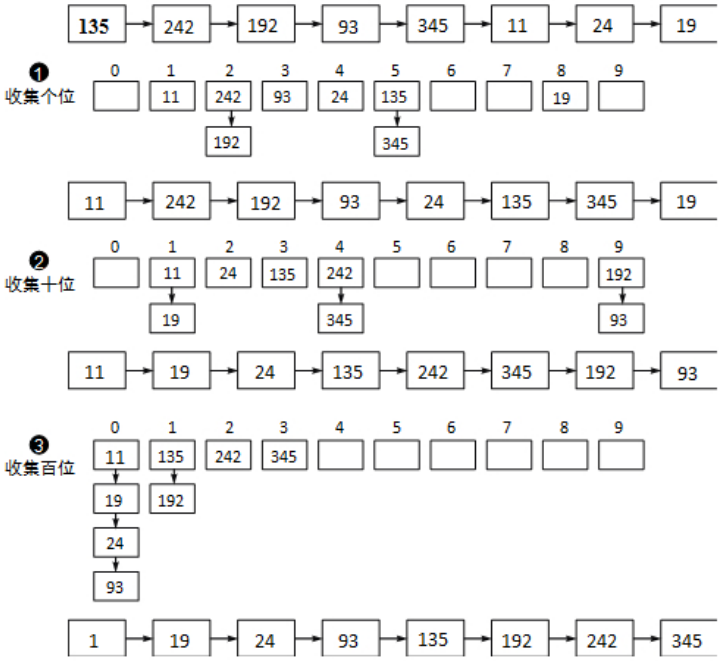


图14-7基数排序示意图

如图14-7所示，在匹配的过程中显然需要额外的辅助存储空间，通常采用链式存储分配的方式来存放中间结果。

9. 排序算法的稳定性和复杂度（如表14-2所示）

表14-2排序算法的稳定性和复杂度

类别	排序方法	时间复杂度		空间复杂度	稳定性
		平均情况	最坏情况	辅助存储	
插入排序	直接插入	$O(n^2)$	$O(n^2)$	O(1)	稳定
	Shell 排序	$O(n^{1.3})$	$O(n^2)$	O(1)	不稳定
选择排序	直接选择	$O(n^2)$	$O(n^2)$	O(1)	不稳定
	堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	O(1)	不稳定
交换排序	冒泡排序	$O(n^2)$	$O(n^2)$	O(1)	稳定
	快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	不稳定
归并排序		$O(n\log_2 n)$	$O(n\log_2 n)$	O(n)	稳定
基数排序		O(d(r+n))	O(d(r+n))	O(rd+n)	稳定

注：基数排序的复杂度中，r代表关键字的基数，d代表长度，n代表关键字的个数。

上一节　本书简介　下一节

作者：希赛教育软考学院　来源：希赛网　2014年05月06日

一点一练

试题1

对n个元素的有序表A[1．．n]进行顺序查找，其成功查找的平均查找长度（即在查找表中找到指定关键码的元素时，所进行比较的表中元素个数的期望值）为__(1)__。

（1）A．n　　B．（n+1）/2　　C．$\log_2 n$　　D．n^2

试题2

对n个元素值分别为-1、0或1的整型数组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n_1、n_2和n_3，然后将A中的前n_1个元素赋值为-1，第n_1+l到n_l+n_2个元素赋值为0，最后n_3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__(2)__。

（2）A．$\Theta(n)$和$\Theta(1)$ B． $\Theta(n)$和$\Theta(n)$

C．$\Theta(n^2)$和$\Theta(1)$ D． $\Theta(n^2)$和$\Theta(n)$

试题3

对于关键字序列（26，25，72，38，8，18，59），采用散列函数H（Key）=Key mod 13构造散列表（哈希表）。若采用线性探测的开放定址法解决冲突（顺序地探查可用存储单元），则关键字59所在散列表中的地址为__(3)__。

（3）A．6 B．7 C．8 D．9

试题4

用插入排序和归并排序算法对数组<3，1，4，1，5，9，6，5>进行从小到大排序，则分别需要进行__(4)__次数组元素之间的比较。

（4）A．12,14 B．10,14 C．12,16 D．10,16

试题5

某一维数组中依次存放了数据元素15,23,38,47,55,62,88,95,102,123,采用折半（二分）法查找元素95时，依次与__(5)__进行了比较。

（5）A．62,88,95 B．62,95 C．55,88,95 D．55,95

试题6

递增序列A（a_1，a_2，…，a_n）和B （b_1,b_2，…，b_n）的元素互不相同，若需将它们合并为一个长度为2n的递增序列，则当最终的排列结果为__(6)__时，归并过程中元素的比较次数最多。

（6）A．a_1，a_2，…，a_n，b_1，b_2，…，b_n

B．b_1，b_2，…，b_n，a_1，a_2，…，a_n

C．a_1，b_1，a_2，b_2，…，a_i，b_i，…，a_n，b_n

D．a_1，a_2，…，$a_{i/2}$，b_1，b_2，…，$b_{i/2}$，$a_{i/2+1}$，$a_{i/2+2}$，…，a_n，$b_{i/2+1}$，…，b_n

试题7

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更杂度分别为__(7)__。

```
    i=0;j=n-1;
    while i<j do
while A[i]<0 do
i= i+1;
while A[j]>0 do
j =j-l;
if i<j do
交换A[i]和A[j];
```

（7）A．$\Theta(n)$和$\Theta(n)$ B． $\Theta(1)$和$\Theta(n)$

C． $\Theta(n)$和$\Theta(1)$ D． $\Theta(1)$和$\Theta(1)$

　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

解析与答案

试题1分析

本题主要考查顺序查找。

使用顺序查找时，如果需要找的元素在第1个位置，则只需要比较1次；而元素在第2个位置，需要比较2次；依次类推，待查找元素在第n个位置时，需要查找n次。题目给出条件，待查找元素在每个位置出现概率相等，所以平均查找长度为：（1+2+3…+n）/n=（n+1）/2。

试题1答案

（1）B

试题2分析

时间复杂度是指程序运行从开始到结束所需要的时间。通常分析时间复杂度的方法是从算法中选取一种对于所研究的问题来说是基本运算的操作，以该操作重复执行的次数作为算法的时间度量。一般来说，算法中原操作重复执行的次数是规模n的某个函数T（n）。由于许多情况下要精确计算T（n）是困难的，因此引入了渐进时间复杂度在数量上估计一个算法的执行时间。其定义如下：

如果存在两个常数c和m，对于所有的n，当n≥m时有f（n）≤cg（n），则有f（n）=O（g（n））。也就是说，随着n的增大，f（n）渐进地不大于g（n）。例如，一个程序的实际执行时间为$T(n)=3n^3+2n^2+n$，则$T(n)=O(n^3)$。

在本题中，根据题目的描述，我们可以知道，遍历完整个数组中的元素，和修改数组中各元素的值都需要时间n，因此是2n，那么该算法的时间复杂度为O（n）。

空间复杂度是指程序运行从开始到结束所需的辅助存储量，在本题中，只需要辅助存储量来存储统计的元素个数，因此其空间复杂度为O（1）。

试题2答案

（2）A

试题3分析

根据题目给出的散列函数我们可以分别计算出关键字（26，25，72，38，8，18，59）对应的散列地址分别为（0，12，7，12，8，5，7）。

开放定址处理冲突的基本思路是为发生冲突的关键字在散列表中寻找另一个尚未占用的位置，其解决冲突能力的关键取决于探测序列，在本题中，题目告诉我们采用顺序探查法，即增量为1的线性探测法，在该线性探测法中，设Hi（1≤i<m）为第i次在散列表中探测的位置，其中增量序列为{1，2，3，4，5，…，m-1}则有：

H_i =（H（Key）+i）%m

其中H（Key）为散列函数，m为散列表长度，i为增量序列。而本题中m=13。因此本题的散列表构造过程如下：

（1）关键字26，25，72由散列函数H（key）得到没有冲突的散列地址而直接存入散列表中。

（2）计算关键38的散列地址为12，发生冲突（与关键字25冲突），其第一次线性探测地址为（12+1）%13=0，但仍然发生冲突（与关键字26冲突），因此需要进行第二次线性探测，其地址为（12+2）%13=1，这时没有发生冲突，即将38存入地址为1的空间。

（3）接着将关键字8，18计算其散列地址，由于没有冲突，即分别存入散列地址为8和5的空间中。

（4）计算关键59的散列地址为7，发生冲突（与关键字72冲突），其第一次线性探测地址（7+1）%13=8，但仍然发生冲突（与关键字8冲突），因此需要进行第二次线性探测，其地址为（7+2）%13=9，这时没有发生冲突，即将59存入地址为9的存储空间。

因此本题的答案选D。

试题3答案

（3）D

试题4分析

插入排序的基本思想是逐个将待排序元素插入到已排序的有序表中。假设n个待排序元素存储在数组R[n+1]中（R[0]预留），则：

（1）初始时数组R[1..1]中只包含元素R[1]，则数组R[1..1]必定有序；

（2）从i=2到n，执行步骤3；

（3）此时，数组R被划分成两个子区间，分别是有序区间R[1..i-1]和无序区间R[i..n]，将当前无序区间的第1个记录R[i]插入到有序区间R[1..i]中适当的位置上，使R[1..i]变为新的有序区间。

在实现的过程中，设置监视哨R[0]，并从R[i-1]到R[0]查找元素R[i]的插入位置

那么用插入排序对数组<3，1，4，1，5，9，6，5>进行排序的过程为：

名称	监视哨	序列	说明
原元素序列：		（3），1，4，1，5，9，6，5	
第一趟排序：	3	（1，3），4，1，5，9，6，5	1 插入时与 3 比较 1 次
第二趟排序：	4	（1，3，4），1，5，9，6，5	4 插入时与 3 比较 1 次
第三趟排序：	1	（1，1，3，4），5，9，6，5	1 插入时比较 3 次
第四趟排序：	5	（1，1，3，4，5），9，6，5	5 插入时与 4 比较 1 次
第五趟排序：	9	（1，1，3，4，5，9），6，5	9 插入时与 5 比较 1 次
第六趟排序：	6	（1，1，3，4，5，6，9），5	6 插入时与 9 和 5 分别比较 1 次
第七趟排序：	5	（1，1，3，4，5，5，6，9）	5 插入时与 9,6,5 分别比较 1 次

那么整个排序过程需要比较的次数为12次。

归并排序的思想是将两个相邻的有序子序列归并为一个有序序列，然后再将新产生的相邻序列进行归并，当只剩下一个有序序列时算法结束。其基本步骤如下：

（1）将n个元素的待排序序列中每个元素看成有序子序列，对相邻子序列两两合并，则将生成个子有序序列，这些子序列中除了最后一个子序列长度可能小于2外，其他的序列长度都等于2；

（2）对上述 个长度为2的子序列再进行相邻子序列的两两合并，则产生 个子有序序列，同理，只有最后一个子序列的长度可能小于4；

（3）第i趟归并排序为，对上述 个长度为i的子序列两两合并，产生 个长度为2i的子有序序列；

（4）重复执行此步骤，直到生成长度为n的序列为止。

名称	序列	说明
原元素序列：	3，1，4，1，5，9，6，5	
第一趟排序：	[1，3]，[1，4]，[5，9]，[5，6]	比较 4 次
第二趟排序：	[1，1，3，4]，[5，5，6，9]	前半部分比较 3 次，后半部分比较 3 次
第三趟排序：	[1，1，3，4，5，5，6，9]	5 分别与 1,1,3,4 比较一次

所以整个排序过程需要比较的次数为12次。

试题4答案

（4）A

试题5分析

本题主要考查折半（二分）法查找算法。这里首先就需要我们能清楚理解该查找算法。

在本题中，给出数据序列为15,23,38,47,55,62,88,95,102,123，其中有10个元素，那么首先进行比较的应该是第5个元素，即55，由于95大于55，那么应该在后半部分进行查找，这是应该与第8个元素进行比较，刚好是95，查找成功，然后结束。因此比较的元素有55和95。

试题5答案

（5）D

试题6分析

要将两个有序序列归并为一个有序序列时，当一个序列的最大值小于另一个序列的最小值时，这时需要比较的次数最小。当获得新序列后，两个序列的元素交替的情况（如选项C），这种情况下需比较的次数最多。

试题6答案

（6）C

试题7分析

根据程序不难看出，要将负实数位于正实数之前，其实就是对所有元素进行了一次遍历，正实数和负实数互换位置即可，因此其时间复杂度为O(n)，由于元素A[i]和A[j]互换时，需要一个临时存储空间来存放元素，因此其空间复杂度为O(1)。

试题7答案

（7）C

上一节　本书简介　下一节

考点精讲

1.迭代法

迭代法是用于解决数值计算问题中的非线性方程（组）求解或最优解（近似根）的一种算法设计方法。它的主要思想是：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；当两者之差接近到可接受的精度范围时，就认为找到了问题的解。由于它是不断进行这样的过程，因此称为迭代法，同时从中也可以看出使用迭代法必须保证其收敛性。

具体来说，迭代法包括简单迭代法、对分法、梯度法、牛顿法等。对分法是指在某个解空间采用二分搜索，它的优点是只要在该解空间内有根，就能够快速地搜索到；梯度法则又称为最速下降法，它常用于工程问题的解决。

在使用的迭代法的过程中，应该注意两种异常情况。

如果方程无解，那么近似根序列将不会收敛，迭代过程会成为“死循环”，因此在使用时应先

判断其是否有解，并应对迭代的次数进行限制；

方程虽然有解，但迭代公式选择不当，或迭代的初始近似根选择不合理，也会导致迭代失败。

迭代法总的来说是一种比较简单的求解方法，但是此类算法还存在两个不足：一是一次只能求方程的一个解，而且需要人工给出近似初值，如果初值选择不好就可能找不到解；二是不易保证程序的收敛性。

2. 穷举搜索法

穷举搜索法是穷举所有可能的情形，并从中找出符合要求的解，即对可能是解的众多候选解按某种顺序逐一枚举和检验，并从中找出那些符合要求的解作为问题的解。对于没有有效的解法的离散型问题，如果规模不大，穷举搜索法是很好的选择。

穷举搜索法通常需要使用多重循环来实现，对每个变量的每个值都进行测试，看其是否满足给定条件，如果满足则说明找到问题的一个解。

3. 递推法

递推法实际上首先需要抽象为一种递推关系，然后再按递推关系来求解。它通常表示为两种方式：

从简单推到一般，这常用于计算级数；

将一个复杂问题逐步推到一个具有已知解的简单问题，它常与“回归”配合为递归法。

递推法是一种简单有效的方法，通常可以编写出执行效率较高的程序。使用的关键是找出递推关系式，并确定初值。

任何用递推法可以解决的问题，都可以很方便地用递归法解决；但有很多可以使用递归法解决的问题，不一定可以使用递推法解决。但如果是既可以使用递归，又可以使用递推法来解决的问题，则应使用递推法，因为它的效率要高于递归法。

4. 递归法

递归是一种特别有用的工具，不仅在数学中广泛应用，还是设计和描述算法的一种有力工具。它经常用于分解复杂算法：将规模为N的问题分解成为规模较小的问题，然后从这些规模较小的问题的解中构造出大问题的解；而这些规模较小的问题采用同样的分解和综合的方法，分解成规模更小的问题；而特别的，当规模为1时，能够得到解。

从上面的描述中，我们可以看出递归算法包括“递推”和“回归”两个部分：递推是为了得到问题的解，将它推到比原问题简单的问题的求解；而回归则是当小问题得到解后，回归到原问题的解上来。

在使用递推时应该注意以下几点：

递推应该有终止点，终止条件便会使算法失效；

“简单问题”表示离递推终止条件更为接近的问题。也就是说简单问题与原问题解的算法是一致的，差别主要是参数。参数的变化将使问题递推到有明确解的问题。

在使用回归时应该注意：

递归到原问题的解时，算法中所涉及的处理对象应是关于当前问题的，即递归算法所涉及的参数与局部处理对象是有层次的。当解一个问题时，有它的一套参数与局部处理对象。当递推进入一“简单问题”时，这套参数与局部对象便隐蔽起来，在解“简单问题”时，又有自己一套。但当回归时，原问题的一套参数与局部处理对象又活跃起来了。

有时回归到原问题以得到问题解，回归并不引起其他动作。

采用递归方法定义的数据结构或问题最适合使用递归方法解答。

当然，回到实际的开发中，递归的表现形式有两种：

函数自己调用自己。

两个函数之间相互调用。

考试时以函数自己调用自己的方式居多。下面以两个程序实例说明该问题。

例：利用递归程序计算n的阶乘。

这是一个非常简单的计算问题，只要学过程序设计，都能用一个简单的循环来解决该问题。编写一个循环语句，实现：S=1*2*3*4*..*n即可。但在此，我们要求用递归来实现，这便要求我们找出阶乘中隐藏的推荐规则，通过总结可得出规律：

$$n! = \begin{cases} 1 & 当n = 0 \\ n * (n-1)! & 当n \neq 0 \end{cases}$$

也就是说：要求n的阶乘，需要分两种情况分析问题，当n=0时，阶乘的结果为1；当n不等于0时，n的阶乘等于n乘以（n-1）的阶乘。这样就产生了递推过程。下面是将这种思路进行程序实现：

```
int factorial(int n)
{
if(!n) return (1); /*对应当 n=0 时*/
      else    return (n*factorial(n-1));/*对应当 n<>0 时*/
}
```

接下来看一个更为复杂的例题：编写计算斐波那契（Fibonacci）数列的函数，数列大小为n。

无穷数列1，1，2，3，5，8，13，21，35，…，称为斐波那契数列。这种数列有一个规律，数列第1个与第2个元素的值均为1，从第3个值开始，每个数据是前两个数据之和。即，数列可以表示为：

1，1，（1+1），（1+（1+1）），（（1+1）+（1+（1+1））…

在此，我们可以把这种规律转成递推式：

$$F(n) = \begin{cases} 1 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$

有了递推式，再来写程序，也就很容易了，直接转化即可，该问题程序实现如下所示：

```
int   F(int n)
{     if(n==0)return   1;
if(n==1)return   1;
if(n>1)return   F(n-1)+F(n-2);
}
```

使用递归法写出的程序非常简洁，但其执行过程却并不好理解。在理解这种方法的过程中，建议大家使用手动运行程序的方式来进行分析，先从最简单的程序开始尝试，逐步到复杂程序。

递归法的用途非常广泛，图的深度优先搜索、二叉树的前序、中序和后序遍历等可采用递归实现。

5. 回溯法

回溯法（试探法）是一种选优搜索法，按选优条件向前搜索，以达到目标。但当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，这种走不通就退回再走的技术为回溯法，而满足回溯条件的某个状态的点称为“回溯点”。其工作机制如图14-8所示。

在使用回溯法时，必须知道以下几个关键的特性：

采用回溯法可以求得问题的一个解或全部解，为了不重复搜索已找过的解，通常会使用栈（也可以用位置指针、值的排列顺序等）来记录已经找到的解。

要注意的是，回溯法求问题的解时，找到的解不一定是最优解。

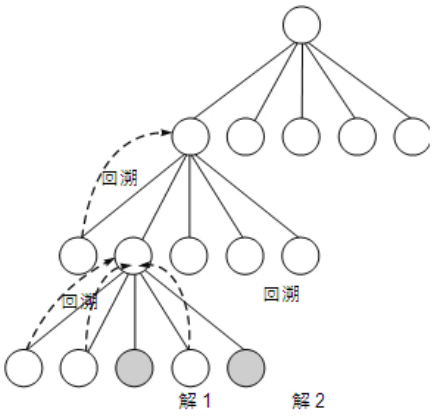


图14-8回溯法示意图

程序要注意记录中间每一个项的值，以便回溯；如果回溯到起始处，表示无解。

用回溯法求问题的全部解时，要注意在找到一组解时应及时输出并记录下来，然后马上改变当前项的值继续找下一组解，防止找到的解重复。

下面我们通过一个经典的问题来研究回溯法的应用。

例：使用回溯法解决迷宫问题，找到迷宫的出路。

基本思路分析：进入一个迷宫之所以难以找到出路，是因为迷宫会有多个岔路口，形成多条路径，而成千上万条路径中，仅有1条（或几条）路径可走出迷宫。若采用回溯法，则在尝试走一条路径时，会把这些分岔都记录好，当一条路走不通时，原路返回到最近的一个分岔口，从这个分岔口找下一路进行尝试，这个过程与14-8所示的情况完全一致。

接下来可以开始把这种思路转化为数据结构来表达了：

设迷宫为m行n列，利用数组maze[m][n]来表示一个迷宫。maze[i][j]=0或1。其中0表示通路，1表示不通。当从某点向下试探时，中间的点有8个方向可以试探，而4个角点只有3个方向，而其他边缘点有5个方向。为使问题简单化，我们用maze[m+2][n+2]来表示迷宫，而迷宫的四周的值全部为1。这样做使问题简单了，每个点的试探方向全部为8，不用再判断当前点的试探方向有几个，同时与迷宫周围是墙壁这一实际问题相一致。

如图14-9所示的迷宫是一个6×8的迷宫。入口坐标为（1,1），出口坐标为（6,8）。

入口(1,1)

	0	1	2	3	4	5	6	7	8	9
0	1	1	1	1	1	1	1	1	1	1
1	1	0	1	1	1	0	1	1	1	1
2	1	1	0	1	0	1	1	1	1	1
3	1	0	1	0	0	0	0	0	1	1
4	1	0	1	1	1	0	1	1	1	1
5	1	1	0	0	1	1	0	0	0	1
6	1	0	1	1	0	0	1	1	0	1
7	1	1	1	1	1	1	1	1	1	1

出口(6,8)

图 14-9 数组maze[m+2][n+2]表示的迷宫

迷宫的定义如下：

```
#define  m   6        /* 迷宫的实际行 */
#define  n   8  /* 迷宫的实际列 */
int maze [m+2][n+2] ;
```

在上述表示迷宫的情况下，每个点有8个方向可以试探。如当前点的坐标为（x,y），与其相邻的8个点的坐标都可根据与该点的相邻方位而得到。因为出口在（m,n），因此试探顺序规定为：从当前位置向前试探的方向为从正东沿顺时针方向进行。为了简化问题，方便地求出新点的坐标，将从正东开始沿顺时针方向进行的这8个方向的坐标增量放在一个结构数组move[8]中，在move 数组中，每个元素由两个域组成，x为横坐标增量，y为纵坐标增量。move数组如图14-10所示。

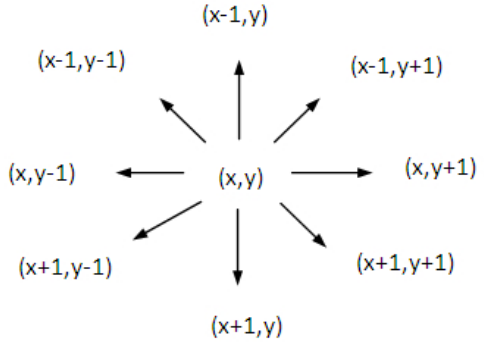


图14-10 点(x,y)相邻的8个点及坐标及增量数组move

move数组定义如下：

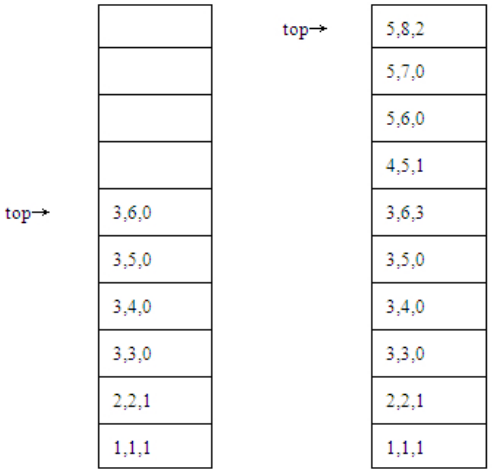
```
typedef  struct
{
   int x,y
} item ;
item move[8] ;
```

这样对move的设计会很方便地求出从某点（x,y）按某一方向v (0≤v≤7) 到达的新点（i,j）的坐标。

	x	Y
0	0	1
1	1	1
2	1	0
3	1	-1
4	0	-1
5	-1	-1
6	-1	0
7	-1	1

可知，试探点的坐标（i,j）可表示为i=x+move[v].x ;j=y+move[v].y。

到达了某点而无路可走时需返回前一点，再从前一点开始向下一个方向继续试探。因此，压入栈中的不仅是顺序到达的各点的坐标，而且还要有从前一点到达本点的方向。对于迷宫，依次入栈如下：



栈中每一组数据是所到达的每点的坐标及从该点沿哪个方向向下走的，对于如图14-9所示的迷宫，走的路线为：（1,1）$_1$→（2,2）$_1$→（3,3）$_0$→（3,4）$_0$→（3,5）$_0$→（3,6）$_0$（下脚标表示方向）；当从点（3,6）沿方向0到达点（3,7）之后，无路可走，则应回溯，即退回到点（3,6），对应的操作是出栈，沿下一个方向即方向1继续试探；方向1、2试探失败，在方向3上试探成功，因此将（3,6,3）压入栈中，即到达了（4,5）点。

栈中元素是一个由行、列、方向组成的三元组，栈元素的设计如下：

```
typedef struct
{int x , y , d ;            /* 横坐标和纵坐标及方向*/
}datatype ;
```

栈的设计如下：

```
#define MAXSIZE 1024   /*栈的最大深度*/
typedef  struct
  {datatype  data[MAXSIZE];
   int  top;/*栈顶指针*/
  }SeqStack
```

一种方法是另外设置一个标志数组mark[m][n]，它的所有元素都初始化为0，一旦到达了某一点（i,j）之后，使mark[i][j]置1，下次再试探这个位置时就不能再走了。另一种方法是当到达某点（i,j）后使maze[i][j]置-1，以便区别未到达过的点，同样也能起到防止走重复点的目的。本书采用

后者方法，算法结束前可恢复原迷宫。

算法简单描述如下：

栈初始化;

将入口点坐标及到达该点的方向（设为-1）入栈

while (栈不空) {

栈顶元素= >（x , y , d）

出栈;

求出下一个要试探的方向d++;

while（还有剩余试探方向时）{

if （d方向可走）

then {（x , y , d）入栈；

求新点坐标（i, j）;

将新点（i , j）切换为当前点（x , y）;

if （(x ,y)= =(m,n))结束；

else 重置d=0；

}

else d++；

}

}

该问题算法程序实现如下所示。

```
#include <stdio.h>
#define m 6                    /* 迷宫的实际行 */
#define n 8                    /* 迷宫的实际列 */
#define MAXSIZE 1024           /* 栈的最大深度 */
int maze[m+2][n+2];            /* 迷宫数组，初始时为0*/
typedef struct item{           /* 坐标增量数组 */
  int x,y;
}item;
item move[8];                  /* 方向数组 */
typedef struct datatype{       /* 栈结点数据结构 */
  int x,y,d;                   /* 横坐标和纵坐标及方向 */
  }datatype;
typedef struct SeqStack{ /* 栈结构 */
  datatype data[MAXSIZE];
int top;                       /* 栈顶指针 */
 }SeqStack;
  SeqStack *s;
  datatype temp;
int path(int maze[m+2][n+2],item move[8]){
```

```
    int x,y,d,i,j;
    temp.x=1;temp.y=1;temp.d=-1;
    Push_SeqStack(s,temp);   /* 辅助变量temp表示当前位置，将其入栈 */
    while(!Empty_SeqStack(s))
    {  Pop_SeqStack(s,&temp); /* 若栈非空，取栈顶元素送temp */
x=temp.x;y=temp.y;d=temp.d+1;
while(d<8)              /* 判断当前位置的8个方向是否为通路 */
{  i=x+move[d].x;j=y+move[d].y;
if(maze[i][j]= =0)
{  temp.x=x;temp.y=y;temp.d=d;
Push_SeqStack(s,temp);
x=i;y=j;maze[x][y]=-1;
if(x==m&&y==n)return 1; /* 迷宫有路 */
else d=0;
}
else d++;
}                          /*while (d<8)*/
}                           /*while */
return 0 ;                      /* 迷宫无路 */
 }
 int Empty_SeqStack(SeqStack *s)      /* 判断栈空函数 */
   {  if (s->top= =-1)return 1;
else return 0;
   }
 int Push_SeqStack(SeqStack *s, datatype x)   /* 入栈函数 */
   {  if(s->top= =MAXSIZE-1)return 0;     /* 栈满不能入栈 */
      else {s->top++;
s->data[s->top]=x;
return 1;
}
    }
int Pop_SeqStack(SeqStack *s,datatype *x)     /* 出栈函数 */
   {  if(Empty_SeqStack(s))return 0;        /* 栈空不能出栈 */
      else{*x=s->data[s->top];
  s->top--;return 1; }                /* 栈顶元素存入*x，返回 */
    }
void main(){
  int i,j,t;
  move[0].x=0;move[0].y=1;
```

```
    move[1].x=1;move[1].y=1;
    move[2].x=1;move[2].y=0;
    move[3].x=1;move[3].y=-1;
    move[4].x=0;move[4].y=-1;
    move[5].x=-1;move[5].y=-1;
    move[6].x=-1;move[6].y=0;
    move[7].x=-1;move[7].y=1;
    for(i=0;i<=n+1;i++){
      maze[0][i]=1;
      maze[m+1][i]=1;
     }
    for(i=1;i<=m;i++){
      maze[i][0]=1;
      maze[i][n+1]=1;
      }
    printf("please input maze\n");
    for(i=1;i<=m;i++)
      for(j=1;j<=n;j++)scanf("%d",&maze[i][j]);
    t=path(maze,move);
    if(t==1){
    printf("the track is :\n");
      while(!Empty_SeqStack(s))
      {  Pop_SeqStack(s,&temp);            /*若栈非空，则打印输出 */
         printf("%d,%d,%d\n",temp.x,temp.y,temp.d);
      }
     }
   }
```

6. 贪婪法

贪婪法，也叫贪心法，它是一种重要的算法设计技术，它总是做出当前来说最好的选择，而并不从整体上加以考虑，它所做的每步选择只是当前步骤的局部最优，而不一定是整体最优。由于它并不必为了寻找最优解而穷尽所有可能解，因此其耗费时间少，一般可以快速得到满意解（即“不追求最优，但求满意”）。

贪婪法只能够求问题的某个解，而不可能给出所有的解。经典的贪婪法算法包括背包问题、装箱问题、马踏棋盘问题、货郎担问题、哈夫曼编码问题。

如此解释很抽象，应用到一个实例中来说明问题。如一个超市的收银系统，要计算出最佳的找零方案，可以采用贪心法。假设找零金额为38元，贪心法的基本思路是：由于人民币面值为：100，50，20，10，5，1。为了找零时，找出的零钞张数最少，所以优先考虑面额大的，先用1张20块的，然后还需要找零18，此时再用1张10块的，再用1张5块的，接下来用3张一块的，这样就完成任务了。我们可以发现这个过程是按既定的顺序一步一步走，形成解决方案，这个过程中不涉及回

溯。通过这个实例，可以了解到贪心法的基本思想。

下面将通过装箱问题来分析贪婪法的具体应用。

例：有6种物品，它们的体积分别为：60、45、35、20、20和20单位体积，箱子的容积为100个单位体积。现在求需要几只箱子才能把这些物品都装起来。

使用贪婪法求解该问题的基本思想是：先将物品的体积按从大到小的顺序排列。然后依次将物品放到它第一个能放进去的箱子中，若当前箱子装不下当前物品，则启用一个新的箱子装该物品，直到所有的物品都装入了箱子。如采用这种方式，我们发现，得到解决方案为：1、3号物品放一个箱子，2、4、5放第二个箱子，6放在第3个箱子，一共需要3个箱子。但由于此问题很简单，我们可以很容易用人工计算的方式得知最优解只要2个箱子。即：1、4、5和2、3、6。所以从此可以看出贪婪法求的是可行解，而非最优解。下面是该问题的算法与程序实现过程。

算法简单描述：

{输入箱子的容积;

输入物品种数n;

按体积从大到小顺序，输入各物品的体积;

预置已用箱子链为空;

预置已用箱子计数器box_count为0;

for (i=0;i<n;i++)

{从已用的第一只箱子开始顺序寻找能放入物品i 的箱子j;

if （已用箱子都不能再放物品i）

{新启用一个箱子，并将物品i放入该箱子;

box_count++;

}

else

将物品i放入箱子j;

}

}

上述算法一次就能求出需要的箱子数box_count，并能求出各箱子所装物品，但该算法不一定能找到最优解。

该问题算法程序实现如下所示。

```
# include<stdio.h>
# include<stdlib.h>
typedef  struct  ele{/*物品结构的信息*/
int  vno;              /*物品号*/
struct  ele  *link;    /*指向下一物品的指针*/
}ELE;
typedef  struct  hnode{/*箱子结构信息*/
int  remainder;          /*箱子的剩余空间*/
ELE  *head;        /*箱子内物品链的首元指针*/
struct  hnode  *next;  /*箱子链的后继箱子指针*/
```

```
}HNODE;
void  main()
{int  n, i, box_count, box_volume, *a;
HNODE  *box_h,  *box_t,  *j;
ELE  *p,  *q;
printf("输入箱子容积\n"); scanf("%d",&box_volume);
printf("输入物品种数\n"); scanf("%d",&n);
a=(int *)malloc(sizeof(int)*n);
printf("请按体积从大到小顺序输入各物品的体积：");
for (i=0;i<n;i++)scanf("%d",a+i); /*数组a按从大到小顺序存放各物品的体积信息*/
box_h=box_t=NULL;          /*box_h为箱子链的首元指针，box_t为当前箱子的指针，初始为
空*/
box_count=0;              /*箱子计数器初始也为0*/
for (i=0;i<n;i++)           /*物品i按下面各步开始装箱*/
{p=(ELE *)malloc(sizeof(ELE));
p->vno=i;              /*指针p指向当前待装物品*/
/*从第一只箱子开始顺序寻找能放入物品i的箱子j*/
for (j=box_h;j!=NULL;j=j->next)
if (j->remainder>=a[i])break; /*找到可以装物品i的箱子，贪婪准则的体现*/
  if (j= =NULL) {  /*已使用的箱子都不能装下当前物品i*/
  j=(HNODE *)malloc(sizeof(HNODE)); /*启用新箱子*/
  j->remainder=box_volume-a[i];  /*将物品i放入新箱子j*/
  j->head=NULL;          /*新箱子内物品链首元指针初始为空*/
  if (box_h= =NULL) box_h=box_t=j; /*新箱子为第一个箱子*/
  elsebox_t=boix_t->next=j;  /*新箱子不是第一个箱子*/
  j->next=NULL;
  box_count++;
  }
  elsej->remainder - =a[i];        /*将物品i放入已用过的箱子j*/
  /*物品放入箱子后要修改物品指针链*/
  for (q=j->head;q!=NULL&&q->link!=NULL;q=q->link);
     if (q= =NULL) {             /*新启用的箱子插入物品*/
    p->link=j->head; j->head=p;  /*p为指向当前物品的指针*/
  }
     else{/*已使用过的箱子插入物品*/
       p->link=NULL; q->link=p;    /*q为指向箱子内物品链顶端的物品*/
     }
}
printf("共使用了%d只箱子", box_count);
```

```
printf("各箱子装物品情况如下：");
for (j=box_h,i=1;j!=NULL;j=j->next,i++)     /*输出i只箱子的情况*/
{printf("第%2d只箱子，还剩余容积%4d，所装物品有;\n",i,j->remainder);
for (p=j->head;p!=NULL;p=p->link)
printf("%4d",p->vno+1);
printf("\n");
}
}
```

7. 分治法

分治法可能算得上是使用最广泛的一种算法设计方法，其基本思想是将大问题分解成一些较小的问题，然后由小问题的解方便地构造出大问题的解。

采用分治法时，应该知道如果不能找到有效的将大问题分析为小问题的方法，那就可能无法得到问题的解；另外它也经常和递归法结合使用；在分治时要注意确保边界的清晰。分治法能够解决的问题通常具有以下特性：

问题缩小到一定程度将很容易解决——通常都能够满足。

问题可以分解为若干规模小的相同问题，这也称为最优子结构性质——前提条件。

利用该问题分解出的子问题的解可以合并为该问题的解——关键，如果只满足前两个，可以考虑使用贪婪法或动态规划法。

该问题所分解出的各个子问题是相互独立——和效率相关。

分治法最简单好理解的实例就是使用二分查找法进行查找。其程序实现如下所示。

```
function Binary_Search(L,a,b,x);
{  if ( a>b ) return(-1);
   else
    { m=(a+b)/2;
      if ( x= =L[m] ) return(m);
      else if(x>L[m])
        return(Binary_Search(L,m+1,b,x));       /*递归实现*/
      else return(Binary_Search(L,a,m-1,x));     /*递归实现*/
    }
}
```

在以上算法中，L为排好序的线性表，x为需要查找的元素，b、a分别为x的位置的上下界，即如果x在L中，则x在L[a..b]中。每次我们用L中间的元素L[m]与x比较，从而确定x的位置范围，然后递归地缩小x的范围，直到找到x。

8. 动态规划法

动态规划法的基本思想与分治法类似，也是将复杂的问题分解成子问题来解决。但只是此处的子问题通常是重叠的，它们是将复杂问题的某些阶段，所以处理方式也有所不同。在此方法中，引入一个数组，不管子问题是否对最终解有用，都会存于该数组中，利用对数组的分析得到最优解。

在求解问题中，对于每一步决策，列出各种可能的局部解，再依据某种判定条件，舍弃那些肯定不能得到最优解的局部解，在每一步都经过筛选，以每一步都是最优解来保证全局是最优解，这

种求解方法称为动态规划法。一般来说，适合于用动态规划法求解的问题具有以下特点：

可以划分成若干个阶段，问题的求解过程就是对若干个阶段的一系列决策过程。

每个阶段有若干个可能状态。

一个决策将你从一个阶段的一种状态带到下一个阶段的某种状态。

在任一个阶段，最佳的决策序列和该阶段以前的决策无关。

各阶段状态之间的转换有明确定义的费用，而且在选择最佳决策时有递推关系（即动态转移方程）。

使用动态规划法求解问题的基本思路如图14-11所示。

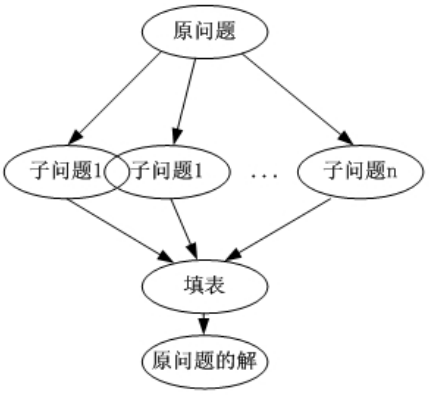


图14-11动态规划法

下面我们用一个通俗的例子，更进一步理解动态规划法：

假如我们要生产一批雪糕，在这个过程中要分好多环节：购买牛奶，对牛奶提纯处理，放入工厂加工，对加工后的商品进行包装，包装后就去销售……，这样每个环节就可以看做是一个阶段；产品在不同的时候有不同的状态，刚开始时只是白白的牛奶，进入生产后做成了各种造型，从冷冻库拿出来后就变成雪糕。每个形态就是一个状态，那从液态变成固态经过了冰冻这一操作，这个操作就是一个决策。一个状态经过一个决策变成了另外一个状态，这个过程就是状态转移。也就是说，在利用动态规划法解决问题时，我们会关注这些状态，以及状态的转移，对于中间结果，会予以保存，这样才能进行下一步的处理。

回到计算机方面的问题，同样是计算斐波那契（Fibonacci）数列，动态规划法的处理方式与递归法有一些差异，动态规划法一般要求填充一个表，如图14-12所示。

1	2	3	4	5	6	7	8
1	1	2	3	5	8	13	21

图14-12斐波那契数列示意图

这个表存储的内容，其实就是中间结果，每一个中间结果存储下来，对于后续要求的内容起到了直接的影响。如，数列的第一个元素是1，这被要求填入表中的第1项，第2个元素也为1，也将存储起来。当要求F（3）时，通过查表，得到F（1）与F（2）的值，将两者相加得到F（3），再将F（3）的值存储到表格第3项中。依此类推，当要求计算出F（9）的值时，可通过查表得到F（7）、F（8）之后相加得到F（9）。这便是动态规划法解决问题的方法。

通过上面的分析，相信大家对动态规划法的基本理念有一定的认知了，但现在要解决考试当中的问题，往往还很难。因为考试往往涉及到程序实现的问题，下面将通过实例描述动态规划法的实现过程。

例：使用动态规划法解决背包问题。有一个背包总容量为42，现有三个物品的价值/重量分别为：40/3，101/31，67/10，请求出背包应该装哪些物品。

使用动态规划法解决这种0-1背包问题的基本思路为：将原问题分解成一系列子问题，然后从这些子问题中求出原问题的解。对一个负重能力为m的背包，如果选择装入第i种物品，那么原背包问题就转化为一个子背包问题了。动态规划会利用空间换时间，将子问题和其结果记录下来，这样一步一步查询得到最终结果。本题的实现代码为：

```
#include<iostream>
int c[10][100];/*对应每种情况的最大价值*/
int knapsack(int m,int n)
{
    int i,j,w[10],p[10];
    for(i=1;i<n+1;i++)
scanf("%d,%d",& w[i],& p[i]);
    for(i=0;i<10;i++)
        for(j=0;j<100;j++)
            c[i][j]=0;/*初始化数组*/
    for(i=1;i<n+1;i++)
        for(j=1;j<m+1;j++)
        {
            if(w[i]<=j) /*如果当前物品的容量小于背包容量*/
            {
                if(p[i]+c[i-1][j-w[i]]>c[i-1][j])
                    /*如果本物品的价值加上背包剩下的空间能放的物品的价值*/
                    /*大于上一次选择的最佳方案则更新c[i][j]*/
                    c[i][j]=p[i]+c[i-1][j-w[i]];
                else
                    c[i][j]=c[i-1][j];
            }
            else c[i][j]=c[i-1][j];
        }
    printf("背包中放着重量如下的物品："); /*确定选取了哪几个物品*/
    i=n;j=m;
    while((i>=0)&&(j>=0))
    {
        if((p[i]+c[i-1][j-w[i]]>c[i-1][j])&&(i-1>=0)&&(j-w[i]>=0)){
            printf("%d  ",w[i]);
            j=j-w[i];
            i=i-1;
        }
        else
            i=i-1;
```

```
    }
    printf("\n");
    return(c[n][m]);
}
void main()
{
    int m,n,k;
    printf("输入总背包容量：");scanf("%d",&m);
    printf("\n");
    printf("输入背包最多可放的物品个数：");scanf("%d",&n);
    printf("\n");
    printf("输入每一组数据:");
    printf("\n");
    k=knapsack(m,n);
    printf("背包所能容纳的最大价值为：%d。",&k);
}
```


上一节　本书简介　下一节

一点一练

试题1

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C_{ij}。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，…，每次在需访问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。刚该算法采用了__(1)__算法设计策略，其时间复杂度为__(2)__。

（1）A．分治　B．动态规划 C．贪心 D．回溯

（2）A．$\Theta(n^2)$ B．$\Theta(n)$ C．$\Theta(n\lg n)$ D．$\Theta(1)$

试题2

迪杰斯特拉（Dijkstra）算法用于求解图上的单源点最短路径。该算法按路径长度递增次序产生最短路径，本质上说，该算法是一种基于__(3)__策略的算法。

（3）A．分治　B．动态规划　C．贪心　D．回溯

试题3

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__(4)__策略的算法。

（4）A．分治　B．动态规划　C．贪心　D．回溯

试题4

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__(5)__来实现。

（5）A．分治法　B．动态规划法　C．贪心法　D．回溯法

试题5

分治算法设计技术__(6)__。

（6）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

试题6

用动态规划策略求解矩阵连乘问题$M_1*M_2*M_3*M_4$，其中M_1(20*5)、M_2(5*35)、M_3(35*4)和M_4(4*25)，则最优的计算次序为__(7)__。

（7）A．$((M_1*M_2)*M_3)*M_4$ B．$(M_1*M_2)*(M_3*M_4)$

C．$(M_1*(M_2*M_3))*M_4$ D．$M_1*(M_2*(M_3*M_4))$

试题7

__(8)__不能保证求得0-1背包问题的最优解。

（8）A．分支限界法　B．贪心算法　C．回溯法　D．动态规划策略

试题8

阅读下列说明和C代码，回答问题1至问题3。

【说明】

用两台处理机A和B处理n个作业。设A和B处理第i个作业的时间分别为ai和bi。由于各个作业的特点和机器性能的关系，对某些作业，在A上处理时间长，而对某些作业在B上处理时间长。一台处理机在某个时刻只能处理一个作业，而且作业处理是不可中断的，每个作业只能被处理一次。现要找出一个最优调度方案，使得n个作业被这两台处理机处理完毕的时间（所有作业被处理的时间之和）最少。

算法步骤：

（1）确定候选解上界为最短的单台处理机处理所有作业的完成时间m，

$$m=\min\left(\sum_{i=1}^{n}a_i,\sum_{i=1}^{n}b_i\right)$$

（2）用p（x,y,k）=1表示前k个作业可以在A用时不超过x且在B用时不超过y时间内处理完成，则p（x,y,k）=p（$x-a_k$,y,k-1）||p（x,$y-b_k$,k-1）（||表示逻辑或操作）。

（3）得到最短处理时间为min（max（x，y））。

下面是该算法的C语言实现。

（1）常量和变量说明

n：作业数

m：候选解上界

a：数组，长度为n，记录n个作业在A上的处理时间，下标从0开始

b：数组，长度为n，记录n个作业在B上的处理时间，下标从0开始

k：循环变量

p：三维数组，长度为（m+1）*（m+1）*（n+1）

temp：临时变量

max：最短处理时间

（2）C代码

```
#include<stdio.h>
int n, m;
int a[60], b[60], p[100][100][60];
void read(){ /*输入n、a、b，求出m，代码略*/}
void schedule(){                /*求解过程*/
int x,y,k;
for（x=0;x<=m;x++）{
for(y=0;y<m;y++）{
（1）
for（k=1;k<n;k++）
p[x][y][k]=0;
}
}
for（k=1;k<n;k++）{
for（x=0;x<=m;x++）{
for（y=0;y<=m;y++）{
if（x - a[k-1]>=0）（2）；
if（（3））p[x][y][k]=(p[x][y][k] ||p[x][y-b[k-1]][k-1]);
}
}
    }
}
void write(){    /*确定最优解并输出*/
int x,y,temp,max=m;
for(x=0;x<=m;x++){
for(y=0;y<=m;y++){
if(（4）){
temp=（5）;
if（temp< max）max = temp;
}
```

```
}
    }
    printf( "\n%d\n" ,max);
}
void main(){
read();
schedule();
write();
}
```

【问题1】

根据以上说明和C代码，填充C代码中的空（1）～（5）。

【问题2】

根据以上C代码，算法的时间复杂度为（6）（用O符号表示）。

【问题3】

考虑6个作业的实例，各个作业在两台处理机上的处理时间如表14-3所示。该实例的最优解为（7），最优解的值（即最短处理时间）为（8）。最优解用（x1,x2,x3,x4,x5,x6）表示，其中若第i个作业在A上处理，则xi=l，否则xi=2。如（1,1,1,1,2,2）表示作业1，2，3和4在A上处理，作业5和6在B上处理。

表14-3

	作业 1	作业 2	作业 3	作业 4	作业 5	作业 6
处理机 A	2	5	7	10	5	2
处理机 B	3	8	4	11	3	4

上一节　本书简介　下一节

第 14 章：算法设计　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

解析与答案

试题1分析

根据题目描述“每次在需访问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”我们不难知道，每次都是选择当前看来最好的情况，因此这是一种贪心算法的设计策略。由于有n个目的地要选择，每选择一个目的地时，需要在所以的目的地中选择出一个最优情况，所以总的时间复杂度为 。

试题1答案

（1）C（2）A

试题2分析

分治法：对于一个规模为n的问题，若该问题可以容易地解决（比如说规模n较小）则直接解决；否则将其分解为k个规模较小的子问题，这些子问题互相独立且与原问题形式相同，递归地解这

些子问题，然后将各子问题的解合并得到原问题的解。

动态规划法：这种算法也用到了分治思想，它的做法是将问题实例分解为更小的、相似的子问题，并存储子问题的解而避免计算重复的子问题。

贪心算法：它是一种不追求最优解，只希望得到较为满意解的方法。贪心算法一般可以快速得到满意的解，因为它省去了为找到最优解而穷尽所有可能所必须耗费的大量时间。贪心算法常以当前情况为基础做最优选择，而不考虑各种可能的整体情况，所以贪心算法不要回溯。

回溯算法（试探法）：它是一种系统地搜索问题的解的方法。回溯算法的基本思想是：从一条路往前走，能进则进，不能进则退回来，换一条路再试。其实现一般要用到递归和堆栈。

针对单源最短路径问题，由Dijkstra提出了一种按路径长度递增的次序产生各顶点最短路径的算法。若按长度递增的次序生成从源点s到其他顶点的最短路径，则当前正在生成的最短路径上除终点以外，其余顶点的最短路径均已生成（将源点的最短路径看做是已生成的源点到其自身的长度为0的路径）。这是一种典型的贪心策略，就是每递增一次，经对所有可能的源点、目标点的路径都要计算，得出最优。

带权图的最短路径问题即求两个顶点间长度最短的路径。其中：路径长度不是指路径上边数的总和，而是指路径上各边的权值总和。

试题2答案

（3）C

试题3分析

在解答本题前请参看本节考点精讲中的常用算法描述，当了解完选项中的各种算法特点后，可以发现本题采用了分治法的策略思想。

试题3答案

（4）A

试题4分析

回溯法是一种选优搜索法，按选优条件向前搜索，以达到目标。但当探索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择，这种走不通就退回再走的技术为回溯法。回溯法求解的过程其实是搜索整个解空间，来找到最优的解。而“皇后”问题是一个典型的用回溯法求解的问题。

试题4答案

（5）D

试题5分析

分治的基本思想就是：对于一个规模为n的问题，若该问题可以容易地解决（比如说规模n较小）则直接解决，否则将其分解为k个规模较小的子问题，这些子问题互相独立且与原问题形式相同，递归地解这些子问题，然后将各子问题的解合并得到原问题的解。

所以分治算法设计技术主要包括三个步骤，分别是问题划分、递归求解、合并解。

试题5答案

（6）A

试题6分析

这个题目的关键是要求最优的计算次序，也就是要求计算过程中，乘法的次数最小。如果用选项A的次序来计算，需要计算的乘法次数为：20*5*35+20*35*4+20*4*25。同样我们可以求出其它

三种方法所需的乘法次数。其中最小的是选项C的5*35*4+20*5*4+20*4*25。

试题6答案

（7）C

试题7分析

分支限界法一般以广度优先或以最小耗费（最大效益）优先的方式搜索问题的解空间，那么肯定能找出最优解。

贪心算法的思想是：总是做出在当前来说是最好的选择，而并不从整体上加以考虑，它所做的每步选择只是当前步骤的局部最优选择，但从整体来说不一定是最优的选择。所以用该算法并不能保证求得0-1背包问题的最优解。

回溯法的思想是：按选优条件向前搜索，以达到目标。但当搜索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择。它其实是遍历了整个解空间，所以肯定能找到最优解。

动态规划法的思想是：在求解问题中，对于每一步决策，列出各种可能的局部解，再依据某种判定条件，舍弃那些肯定不能得到最优解的局部解，在每一步都经过筛选，以每一步都是最优解来保证全局是最优解。它能求得0-1背包问题的最优解。

试题7答案

（8）B

试题8分析

本题是一个求最优解的问题。

【问题1】

第（1）空所处的位置为schedule（）函数的for循环中，从题目的描述和程序不难看出该三重循环的作用是给三维数组p赋初值，而根据题目描述可知数组k=0时，其对应的数组元素值都为1（因为这个时候没有作业，那么肯定可以在A用时不超过x且在B用时不超过y时间内处理完成），因此第1空应该填p[x][y][0]=1。

第（2）空在函数schedule()中的第二个三重for循环中，而且是在if结构下，只有if条件的结果为真时，才执行第（2）空的程序，从题目和程序也不难看出，这个三重for循环的作用就是要实现题目算法描述中的第（2）步，即求出p数组中各元素的值。那么当x - a[k-1]>=0为真时，即说明前k个作业可以在A用时不超过x内处理完成，那么根据题目意思，应该p（x,y,k）=p（$x-a_k$,y,k-1），因此第（2）空的答案应该是p[x][y][k]=p[x-a[k-1]][y][k-1]。

第（3）空if判定的条件表达式，根据条件为真后面执行的语句可以判定出，这里的条件是要判定是否前k个作业可以在B用时不超过y内处理完成，因此第（3）空的答案是y- b[k -1]>=0，其实本题与第（2）空可以参照来完成。

第（4）空在函数write（）中，是双重循环下if判定的条件，从题目注释来看，该函数是要确定最优解并输出的，那么结合该函数我们不难知识，确定最优解就是用这个双重循环来实现的，从前面的程序中，我们知道，所有的解的情况保存在数组p当中，那么现在就是要找出那个是最优解，其中max是用来存放当前最优解的，而临时变量temp要与max的值做一个比较，将较小的（当前最优）存放在max中，因此求最优解其实就是将所有解做一个比较，然后取出最优解。综上所述，再结合程序，我们不难知道第（4）空的答案是p[x][y][n]==1或者类似的表达式，p[x][y][n]==1表示当前情况下有一个解，那么这个解是x还是y呢？这还需要接着判定x与y的值谁更小，将更小的赋值给临时变量temp，因此第5空答案为（x>=y）?x:y。

【问题2】

本题主要考查时间复杂度，相对于第一问来说，要简单很多。从给出的程序来看，最高的循环是三重循环，因此其时间复杂度为O（m^2n）或者O（n^3）。

【问题3】

在本题给出的实例中，如果我们用题目描述的方式来求解，其过程也是相当复杂，因为在题目描述的情况下，数组p的长度为（33+1）*（33+1）*（6+1），由于我们不是计算机，要计算出该数组中各元素，肯定也不容易。在这种情况下，因为题目给出的作用只有6个，因此可以采用观察法，不难发现，本题最优解的值为15，最优解为（1，1，2，2，1，1）或者（2，1，2，1，2，2）。

试题8答案

【问题1】

（1）p[X][y][0]=1

（2）p[X][y][k]=p[x-a[k-1]][y][k-1]

（3）y- b[k -1]>=0

（4）p[x][y][n]==1或 p[X][y][n] 或 p[x][y][n]!=0

（5）（x>=y）?x:y

【问题2】

（6）O（m^2n）

【问题3】

（7）（1,1,2,2,1,1）

（8）15

上一节　本书简介　下一节

作者：希赛教育软考学院　来源：希赛网　2014年05月06日

考前冲刺

试题1

阅读下列说明和c代码，将应填入（n）处的字句补充完整。

【说明】

设某一机器由n个部件组成，每一个部件都可以从m个不同的供应商处购得。供应商j供应的部件i具有重量Wij和价格Cij。设计一个算法，求解总价格不超过上限cc的最小重量的机器组成。

采用回溯法来求解该问题：

首先定义解空间。解空间由长度为n的向量组成，其中每个分量取值来自集合{l，2，…，m}，将解空间用树形结构表示。

接着从根结点开始，以深度优先的方式搜索整个解空间。从根结点开始，根结点成为活结点，同时也成为当前的扩展结点。向纵深方向考虑第一个部件从第一个供应商处购买，得到一个新结

点。判断当前的机器价格（C_{11}）是否超过上限（cc），重量（W_{11}）是否比当前已知的解（最小重量）大，若是，应回溯至最近的一个活结点；若否，则该新结点成为活结点，同时也成为当前的扩展结点，根结点不再是扩展结点。继续向纵深方向考虑第二个部件从第一个供应商处购买，得到一个新结点。同样判断当前的机器价格($C_{11}+C_{21}$)是否超过上限(cc)，重量（$W_{11}+W_{21}$）是否比当前已知的解（最小重量）大。若是，应回溯至最近的一个活结点；若否，则该新结点成为活结点，同时也成为当前的扩展结点，原来的结点不再是扩展结点。以这种方式递归地在解空间中搜索，直到找到所要求的解或者解空间中已无活结点为止。

下面是该算法的C语言实现。

（1）变量说明

n：机器的部件数

m：供应商数

cc：价格上限

w[][]：二维数组，w[i][j]表示第j个供应商供应的第i个部件的重量

c[][]：二维数组，c[i][j]表示第j个供应商供应的第i个部件的价格

bestW：满足价格上限约束条件的最小机器重量

bestC：最小重量机器的价格

bestX[]：最优解，一维数组，bestX[i]表示第i个部件来自哪个供应商

cw：搜索过程中机器的重量

cp：搜索过程中机器的价格

x[]：搜索过程中产生的解，x[i]表示第i个部件来自哪个供应商

i：当前考虑的部件，从0到n-l

j：循环变量

（2）函数backtrack

```
int n=3;
    int m=3;
    int cc=4;
    int w[3][3]={{1,2,3},{3,2,1},{2,2,2}};
    int c[3][3]={{1,2,3},{3,2,1},{2,2,2}};
    int bestW=8;
    int bestC=0;
    int bestX[3]={0,0,0}；
    int cw=0;
    int cp=0;
    int x[3]={0,0,0};
int backtrack(int i){
      int j=0;
      int found=0;
      if(i>n-1){/*得到问题解*/
 bestW= cw;
```

```
    bestC= cp;
        for(j=0 ; j<n ; j++){
 (1)_;
    }
        return 1;
}
if(cp<=cc){/*有解*/
        found=1;
}
for(j=0;  (2)_;j++){
/*第i个部件从第j个供应商购买*/
  (3) ;
cw=cw+w[i][j];
cp=cp+c[i][i][j];
if(cp<=cc &&  (4)  {/*深度搜索，扩展当前结点*/
 if(backtrack(i+1)){found=1;}
  }
        /*回溯*/
         cw= cw -w[i][j] ;
  (5) ;
    }
    return found;
 }
```

试题2

阅读下列说明和C代码，回答问题1至问题3。

【说明】

某应用中需要对100000个整数元素进行排序，每个元素的取值在0～5之间。排序算法的基本思想是：对每一个元素x，确定小于等于x的元素个数(记为m)，将x放在输出元素序列的第m个位置。对于元素值重复的情况，依次放入第m-l、m-2、…个位置。例如，如果元素值小于等于4的元素个数有10个，其中元素值等于4的元素个数有3个，则4应该在输出元素序列的第10个位置、第9个位置和第8个位置上。算法具体的步骤为：

步骤1：统计每个元素值的个数。

步骤2：统计小于等于每个元素值的个数。

步骤3：将输入元素序列中的每个元素放入有序的输出元素序列。

下面是该排序算法的C语言实现。

（1）常量和变量说明

R:常量，定义元素取值范围中的取值个数，如上述应用中R值应取6。

i：循环变量。

n：待排序元素个数。

a：输入数组，长度为n。

b：输出数组，长度为n。

c：辅助数组，长度为R，其中每个元素表示小于等于下标所对应的元素值的个数。

（2）函数sort

```
1    void sort(intn,int a[],intb[]){
2    intc[R],i;
3    for (i=0;i<   (1)  ;i++){
4          c[i]=0;
5    }
6    for(i=O;i<n;i++ ) {
7    c[a[i]] =   (2)  ;
8    }
9    for(i=1;i<R;i++ ) {
10       c[i]=   ( 3 ) ;
11    }
12   for(i=0;i<n;i++ ) {
13      b[c[a[i]]-1]=  (4)  ;
14    c[a[i]]=c[a[i]]-1;
15    }
16    }
```

【问题1】

根据说明和C代码，填充C代码中的空缺(1)～(4)。

【问题2】

根据C代码，函数的时间复杂度和空间复杂度分别为（5）和（6）（用O符号表示）。

【问题3】

根据以上C代码，分析该排序算法是否稳定。若稳定，请简要说明（不超过100字）；若不稳定，请修改其中代码使其稳定（给出要修改的行号和修改后的代码）。

试题3

阅读下列说明和C代码，回答问题1至问题3。

【说明】

堆数据结构定义如下：

对于n个元素的关键字序列 $\{a_1, a_2, \ldots, a_n\}$，当且仅当满足下列关系时称其为堆。

$$\begin{cases} a_i \le a_{2i} \\ a_i \le a_{2i+1} \end{cases} \text{或} \begin{cases} a_i \ge a_{2i} \\ a_i \ge a_{2i+1} \end{cases} \text{其中，} i = 1, 2, \ldots, \left\lfloor n/2 \right\rfloor$$

在一个堆中，若堆顶元素为最大元素，则称为大顶堆；若堆顶元素为最小元素，则称为小顶堆。堆常用完全二叉树表示，图14-13是一个大顶堆的例子。

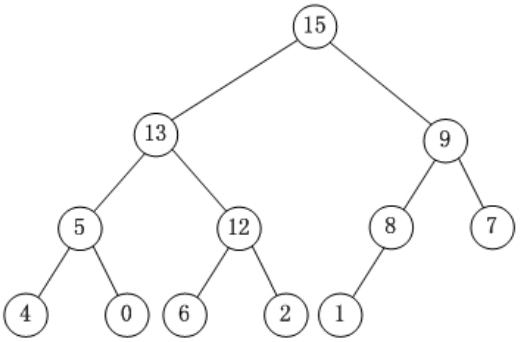


图14-13大顶堆示例

堆数据结构常用于优先队列中，以维护由一组元素构成的集合。对应于两类堆结构，优先队列也有最大优先队列和最小优先队列，其中最大优先队列采用大顶堆，最小优先队列采用小顶堆。以下考虑最大优先队列。

假设现已建好大顶堆A，且已经实现了调整堆的函数heapify(A,n,index)。

下面将C代码中需要完善的三个函数说明如下：

（1）heapMaximum（A）：返回大顶堆A中的最大元素。

（2）heapExtractMax（A）：去掉并返回大顶堆A的最大元素，将最后一个元素“提前”到堆顶位置，并将剩余元素调整成大顶堆。

（3）maxHeapInsert（A,key）：把元素key插入到大顶堆A的最后位置，再将A调整成大顶堆。

优先队列采用顺序存储方式，其存储结构定义如下：

```
#define PARENT（i） i/2
typedef struct array {
  int *intarray;//优先队列的存储空间首地址
  int arraysize;//优先队列的长度
  int capacity;//优先队列存储空间的容量
} ARRAY;
```

（1）函数heapMaximum

```
int heapMaximum（ARRAY *A）{ return  (1) ; }
```

（2）函数heapExtractMax

```
int heapExtractMax（ARRAY *A）{
int max;
    max = A->intarray[0];
（2）;
    A->array_size --;
heapify（A,A->array_size,0）;//将剩余元素调整成大顶堆
    return max;
  }
```

（3）函数maxHeapInsert

```
int maxHeapInsert（ARRAY *A, int key）{
    int i,*p;
    if (A->array-size=A->capacity）{//存储空间的容量不够时扩充空间
```

```
            p = (int*)realloc(A->int array, A->capacity *2*sizeof(int));
            if (!p)  return-1;
            A->intarray = p;
            A->capacity=2 * A->capacity;
    }
        A->array_size++;
        i =(3);
        while ( i>0&&  (4)  ) {
            A->int_array[i] = A->int_array[PARENT(i)];
            i = PARENT ( i ) ;
    }
    (5)  ;
        return 0;
    }
```

【问题1】

根据以上说明和c代码，填充c代码中的空（1）~（5）。

【问题2】

根据以上c代码，函数heapMaximum, heapExtractMax和maxHeapInsert的时间复杂度的紧致上界分别为（6）、（7）和（8）（用O符号表示）。

【问题3】

若将元素10插入到堆A = （15,13,9,5,12,8,7,4,0,6,2,1）中，调用maxHeapInsert

函数进行操作，则新插入的元素在堆A中第（9）个位置（从1开始）。

试题4

阅读下列说明，回答问题1和问题2，将解答填入答题纸的对应栏内。

【说明】

现需在某城市中选择一个社区建一个大型超市，使该城市的其它社区到该超市的距离总和最小。用图模型表示该城市的地图，其中顶点表示社区，边表示社区间的路线，边上的权重表示该路线的长度。现设计一个算法来找到该大型超市的最佳位置：即在给定图中选择一个顶点，使该顶点到其它各顶点的最短路径之和最小。算法首先需要求出每个顶点到其它任一顶点的最短路径，即需要计算任意两个顶点之间的最短路径；然后对每个顶点，计算其它各顶点到该顶点的最短路径之和；最后，选择最短路径之和最小的顶点作为建大型超市的最佳位置。

【问题1】

本题采用Floyd-Warshall算法求解任意两个顶点之间的最短路径。已知图G 的顶点集合为V={1,2,…,n }，W= { W_{ij} } $_{n*n}$为权重矩阵。设 $d^{(k)}_{ij}$=为从顶点i到顶点 j的一条最短路径的权重。当k = 0时，不存在中间顶点，因此$d^{(0)}_{ij}=w_{ij}$当k >0 时，该最短路径上所有的中间顶点均属于集合 {1,2, …, k } 若中间顶点包括顶点 k ，则$d^{(k)}_{ij}=d^{(k-1)}_{ik}+d^{(k-1)}_{kj}$若中间顶点不包括顶点则$d^{(k-1)}_{ij}=d^{(k-1)}_{i}$于是得到如下递归式

$$d_{ij}^{(k)} = \begin{cases} W_{ij} & k = 0 \\ \min(d_{ij}^{(k-i)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & k > 0 \end{cases}$$

因为对于任意路径，所有的中间顶点都在集合{1,2, ..., n } 内，因此矩阵$D^{(n)}=\{d^{(n)}_{ij}\}_{n*n}$给出了任意两个顶点之间的最短路径，即对所有i, j ∈V,$d^{(n)}_{ij}$表示顶点i到顶点 j的最短路径。

下面是求解该问题的伪代码，请填充其中空缺的 (1)至(6)处。伪代码中的主要变量说明如下：

W：权重矩阵

n：图的顶点个数

SP：最短路径权重之和数组，SP[i]表示顶点i到其它各顶点的最短路径权重之和，i从1到n

min_SP：最小的最短路径权重之和

min_v：具有最小的最短路径权重之和的顶点

i：循环控制变量

j：循环控制变量

k：循环控制变量

考前冲刺

```
3          for i = 1 to n
4                for j = 1 to n
5              if d_ij^(k-1) ≤ d_ik^(k-1) + d_kj^(k-1)
6         ( 2 )
7           else
8                      ( 3 )
9   for i = 1 to n
10       SP[i] = 0
11       for j = 1 to n
12          ( 4 )
13    min_SP = SP[1]
14    ( 5 )
15   for i = 2 to n
16         if min_SP > SP[i]
17               min_SP = SP[i]
18               min_v = i
19  return    ( 6 )
```

【问题2】

【问题1】中伪代码的时间复杂度为（ 7 ）（ 用O 符号表示 ）。

试题5

阅读以下说明和C程序，将应填入（ n ）处的字句写在答题纸的对应栏内。

【说明】

现有n（n<1000）节火车车厢，顺序编号为1,2,3,…,n，按编号连续依次从A方向的铁轨驶入，从B方向铁轨驶出，一旦车厢进入车站（Station）就不能再回到A方向的铁轨上；一旦车厢驶入B方向铁轨就不能再回到车站，如图14-14所示，其中Station为栈结构，初始为空且最多能停放1000节车厢。

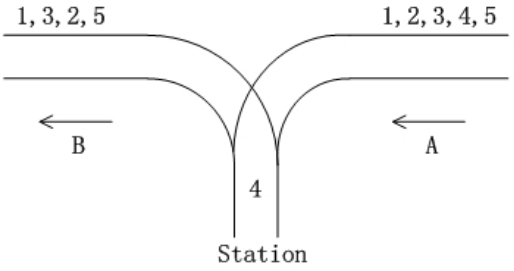


图14-14车站示意图

下面的C程序判断能否从B方向驶出预先指定的车厢序列，程序中使用了栈类型STACK，关于栈基本操作的函数原型说明如下：

void InitStack(STACK *s)：初始化栈。

void Push (STACK *s，int e）：将一个整数压栈，栈中元素数目增1。

void Pop (STACK *s)：栈顶元素出栈，栈中元素数目减1。

int Top (STACK s)：返回非空栈的栈顶元素值，栈中元素数目不变。

int IsEmpty (STACK s)：若是空栈则返回1，否则返回0。

```
#include<stdio.h>
/*此处为栈类型及其基本操作的定义，省略*/
 int main() {
    STACK station;
    int state[1000];
    int n;              /*车厢数*/
    int begin, i, j, maxNo; /*maxNo为A端正待入栈的车厢编号*/
    printf("请输入车厢数：");
    scanf("%d",&n);
    printf(“请输入需要判断的车厢编号序列（以空格分隔）：”);
    if（n<1）return-1;
    for (i=0; i<n; i++)  /*读入需要驶出的车厢编号序列，存入数组state[]*/
      scanf("%d",&state[i]);
（1）;          /*初始化栈*/
    maxNo=1;
    for(i=0; i<n; ){  /*检查输出序列中的每个车厢号state[i]是否能从栈中获取*/
      if（（2））{    /*当栈不为空时*/
        if (state[i]=Top(station)) {    /*栈顶车厢号等于被检查车厢号*/
          printf("%d",Top(station));
          Pop(&station);i++;
}
```

```
            else
                if ( ( 3 ) ) {
                    printf( “error\n” ) ;
                     return 1;
    }
                else {
                     begin= ( 4 ) ;
                     for(j=begin+l;j <=state [i];j++) {
                        Push(&station, j);
                      }
    }
    }
        else {    /*当栈为空时*/
            begin=maxNo ;
            for(j=begin; j<=state[i];j++) {
                Push(&station, j);
    }
              maxNo= ( 5 ) ;
    }
    }
      printf("OK");
      return 0;
    }
```


上一节　本书简介　下一节

习题解析

试题1分析

本题考查回溯法的应用。

在题目的描述中告诉了我们回溯法的基本思想。其实回溯法主要有两个过程，一个是向前探索，只要在当前满足设定的判定条件时，才向前探索，而另外一个就是回溯，在两种情况下，需要回溯，其分别是当不满足设定条件时和求的一个解的时候。

下面我们来具体分析本试题。根据题目给出的注释，我们知道第（1）空所处的位置是得到问题的一个解时，我们该怎么办，根据题目描述，应该是将这个解记录下来，存放到bestX数组当中，而求得的解是保存在x数组当中的，因此这里需要循环将x数组中的元素值赋给bestX数组，因此第（1）空答案为bestX[j] = x[j]。

第（2）空是for循环中的循环判定条件，根据题目注释我们知道该循环的作用是确定第i个部件从第j个供应商购买，那么在确定第i个部件到底是从哪个供应商购买时，需要比较从各供应商购买的情况，因此循环的次数为供应商数，因此第（2）空答案是j < m。结合这个循环体当中的语句和我们对回溯法的理解，我们可以发现循环下面的语句是要考虑将第i个部件从供应商j当中购买，也就是j是当前解的一部分，因此需要将j记录到解当中来，所以第（3）空应该是x[i] = j。

第（4）空是if语句中的一个条件，根据题目注释，我们可以知道如果该if语句表达式的计算结果为真，需要进行深度搜索，扩展当前结点，那么如果要继续向前探索，就需要满足设定的条件，也就是当前总重量要小于bestW,而当前总价格要小于等于cc，因此第（4）空的答案应该填cw < bestW。

根据题目注释，第（5）空是在回溯下面的语句，根据回溯的原则我们可以知道，回溯时，要将当前考虑的结点的重量和价格从总重量和总价格中减去，因此第（5）的答案是cp = cp - c[i][j]。

试题1答案

（1）bestX[j]=x[j]

（2）j<m

（3）x[i]=j

（4）cw<bestW

（5）cp=cp-c[i][j]

试题2分析

【问题1】

本题考查排序的相关内容。

题目告诉我们排序算法的基本思想是：对每一个元素x，确定小于等于x的元素个数（记为m），将x放在输出元素序列的第m个位置。对于元素值重复的情况，依次放入第m-l、m-2、…的位置。而且题目告诉我们算法的步骤。

下面我们来具体分析本试题。第（1）空所处的位置为函数sort()中第一个for循环中，从题目的描述和程序不难看出该循环的作用是给数组c赋初值，而根据题目描述可知数组c是一个辅助数组，长度为R，因此第一空应填R。

第（2）空在函数sort()中的第二个for循环中，很显然第（2）空是给数组c赋值，而且其下标为数组a的相应的元素值。再根据题目的描述“c数组中每个元素表示小于等于下标所对应的元素值的个数”，很显然，这个for循环的作用是统计每个元素值的个数，因此第（2）空的答案应该是c[a[i]]+1。

第（3）空在第三个for循环中，而且第（3）空是给出数组c赋值，根据题目提供的算法的步骤，我们可知，这个时候应该要统计小于等于每个元素值的个数，而等于的元素个数记录在c[i]中，小于的元素个数记录在c[i-1]中，因此第（3）空的答案是c[i]+c[i-1]。

第（4）空在最后一个for循环中，按题目要求，我们可以知道该for循环应该完成剩余的步骤3，即将输入元素序列中的每个元素放入有序的输出元素序列。而第（4）空是给数组b赋值，题目告诉我们b是输出数组，而a是输入数组，那么应该是将a中的值赋值值b中，因此第（4）空的答案应该为a[i]。

【问题2】

本题主要考查时间复杂度与空间复杂度的分析。

首先我们来看空间复杂度，空间复杂度是对一个算法在运行过程中临时占用存储空间大小的量度。在sort()函数中，声明了两个整型变量n和i（可忽略），两个整型数组b和c，而a不属于函数sort的临时空间，因此函数sort()的空间复杂度为O(n+R)，这里由于在本题中R的值为6，因此也可以忽略，所以答案也可以是O(n)。

接着我们来分析时间复杂度，时间复杂度是度量算法执行的时间长短，函数sort()中有四个循环，其中有两个循环n次，而另外两个分别循环R-1和R次，因此时间复杂度应该为O(n+R)，由于R的值为6，这里可以忽略，因此答案也可以是O(n)。

【问题3】

所谓稳定性是指两个关键字相等的元素在排序前后的相对位置不发生变化，一般来讲，只要排序过程中比较和移动操作发生在相邻的元素间，排序方法是稳定的。本题中的排序是不稳定的，修改第12行的for循环为：for(i=n-1;i>=0;i--)即可变得稳定。

试题2答案

【问题1】

（1）R （2）c[a[i]]+1

（3）c[i]+c[i-1]（4）a[i]

【问题2】

（5）O(n+R)或者O(n)或n或线性

（6）O(n+R)或者O(n)或n或线性

【问题3】

不稳定。修改第12行的for循环为for(i=n-1;i>=0;i--)即可。

试题3分析

本题考查排序算法中的堆排序相关内容。

【问题1】

题目告诉我们函数heapMaximum（A）的功能是返回大顶堆A中的最大元素；函数heapExtractMax（A）的功能是去掉并返回大顶堆A的最大元素，将最后一个元素“提前”到堆顶位置，并将剩余元素调整成大顶堆；而函数maxHeapInsert（A, key）的功能是把元素key插入到大顶堆A的最后位置，再将A调整成大顶堆。

第（1）空在函数heapMaximum（A）中，而且从程序中可以看出，是返回的结果，那么应该是大顶堆中最大元素，即A->int_array[0]。

第（2）空在函数heapExtractMax（A）中，根据该函数的功能描述，并结合程序可以看出，第（2）空是在将最大元素移出后，那么接下了来应该处理将最后一个元素“提前”到堆顶位置，那么就应该是A->int_array[0] = A->int_array[A->array_size -1]。

第（3）（4）（5）空都在函数maxHeapInsert（A, key）中。从程序和函数的功能我们可以知道，从程序第（3）空最后，其作用是找到元素key的插入位置并插入该元素。第（3）空是给变量i赋值，从后面的程序中我们可以看出i是作为数组下标的；而查找元素插入的位置应该从后往前的顺序，因此i的初值应该为A->array_size–1，从循环中也可以看出i的值在逐渐变小。

第（4）空是循环的一个条件，而循环的作用是找到合适的插入位置，由于大顶堆的特点是根节点的值大于左右子树节点上的值，那么找到比待插入元素大的父节点时，应该就找到了它插入的合适位置，而每次操作后i的值被赋值为PARENT（i），很显然这是找到其父节点的存储位置，因此循

环结束的一个条件就是找到一个比key值大的父节点，那么循环继续的条件就是父节点的值小于key的值，所以本空的答案为A->int_array[PARENT(i)] <key。

第（5）空就是插入元素，所以应该填A->int_array[i] = key。

【问题2】

根据题目描述，heapMaximum用来返回大顶堆A中的最大元素，而且大顶堆已经建成，只需要通过一步操作就能取到。因此时间复杂度是O（1）。

而对于heapExtractMax是用来去掉大顶堆A的根，然后重新建堆，当输出堆顶结点并将堆中最后一个结点设置为根结点之后，根结点将有可能不再满足堆的性质，所幸的是整个序列也只有根结点一处的堆结构可能被破坏，其余结点仍然满足堆性质，故可利用性质进行堆调整，算法的基本思想为：将新堆顶沿着其关键字较大的孩子结点向下移动，直到叶子结点或者满足堆性质为止。因此相对于有N个元素的堆，只需要lgn次比较即可完成，因此时间复杂度是O（lgn），这与书本说堆排序的算法时间复杂度是：O(nlgn)不冲突，因为书本上是对堆中所有元素进行操作，而这里其实相当于只将一个元素入堆，因此少了一个n。同样的道理可以得到maxHeapInsert的时间复杂度O（lgn）。

【问题3】

这个我们可以结合题目给出的那个大顶堆的图来看，首先将key插入在最后，应该是8这个节点的右子树，由于10比8大，所以应该互换，再与节点9比较，由于10任然大于9，所以也应该互换，这个时候再与其父节点15比较，由于小于15，所以不需要再调整，那么调整后的结果就是10这个元素应该作为根节点15的右子树。那么很显然10应该是在堆A中第3个位置。

试题3答案

【问题1】

（1）A->int_array[0]

（2）A->int_array[0] = A->int_array[A->array_size-1]

（3）A->array_size-1

（4）A->int_array[PARENT(i)]<key

（5）A->int_array[i]=key

【问题2】

（6）O（1）（7）O（lgn）（8）O（lgn）

【问题3】

（9）3

试题4分析

本题考查的是算法的设计和分析技术。

【问题1】

本问题考查算法流程。第（1）空表示主循环，k是循环控制变量，故第（1）空填k=1 to n。第（2）和（3）空根据题意和递归式，可分别得到答案为 $d_{ij}^{(k)}=d_{ij}^{(k-1)}$ 和 $d_{ij}^{(k)}=d_{ik}^{(k-1)}+d_{kj}^{(k-1)}$。计算了任意两个顶点之间的最短路径之后，对每个顶点，开始统计其到所有其他顶点的最短路径之和，因此第（4）空填SP[i]=SP[i]+$d_{ij}^{(n)}$。第13和第14行初始化，假设最小的到所有其他顶点的最短路径之和为第一个顶点的最小路径之和，大型超市的最佳位置为第一个顶点，故第（5）空填min_v = l。最后要求返回大型超市的最佳位置，即到所有其他项点的最短路径之和最小的顶点，故第

（6）空填min_v。

【问题2】

本问题考查【问题1】中的伪代码第2～8行，计算仟意两点之间的最短路径，有三重循环，故时间复杂度为O（n^3）。第9～12行，计算每个点到任意其他点的最短路径之和，有两重循环，故时间复杂度为O（n^2）。第15～18行，在所有点的最短路径之和中找到最小的最短路径之和，时间复杂度为O（n）。故算法总的时间复杂度为O（n^3）。

试题4参考答案

【问题1】

（1）k=1 to n（2）为$d_{ij}^{(k)} = d_{ij}^{(k-1)}$（3）$d_{ij}^{(k)} = d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$

（4）SP[i]=SP[i] +$d_{ij}^{(n)}$（5）min_v=1（6）min_v

【问题2】

（7）O(n^3）

试题5分析

本题考查栈数据结构的应用和C程序设计基本能力。

栈的运算特点是后进先出。在本题中，入栈序列为1、2、…、n-1、n，出栈序列保存在state[]数组中，state[0]记录出栈序列的第1个元素，state[1]记录出栈序列的第2个元素，依此类推。程序采用模拟元素入栈和出栈的操作过程来判断出栈序列是否恰当。需要注意的是，对于栈，应用时不一定是所有元素先入栈后，再逐个进行出栈操作，也不一定是进入一个元素紧接着就出来一个元素，而是栈不满且输入序列还有元素待进入就可以进栈，只要栈不空，栈顶元素就可以出栈，从而使得唯一的一个入栈序列可以得到多个出栈序列。当然，在栈中有多个元素时，只能让栈顶的元素先出栈，栈中其他的元素能从顶到底逐个出栈。本题中入栈序列和出栈序列的元素为车厢号。

空（1）处对栈进行初始化，根据题干中关于栈基本操作的说明，调用InitStack初始化栈，由于形参是指针参数，因此实参应为地址量，即应填入“Initstack(&station)”。

当栈不空时，就可以令栈顶车厢出栈，空（2）处应填入“!IsEmpty(station)”。

栈顶车厢号以Top(station)表示，若栈顶车厢号等于出栈序列的当前车厢号state[i]，说明截至到目前为止，出栈子序列state[0]～state[i]可经过栈运算获得。由于进栈时小编号的车厢先于大编号的车厢进入栈中，因此若栈顶车厢号大于出栈序列的当前车厢号state[i]，则对于state[i]记录的车厢，若它还在栈中，则此时无法出栈，因为它不在栈顶，所以出错，若它已先于当前的栈顶车厢出栈，则与目前的出栈序列不匹配，仍然出错，因此空（3）处应填入“state[i]<Top(station)”。

若栈顶车厢号小于出栈序列的当前车厢号state[i]，则说明state[i]记录的车厢还没有进入栈中，因此从入栈序列（A端）正待进入的车厢（即比栈顶车厢号正好大l）开始，直到state[i]记录的车厢号为止，这些车厢应依次进入栈中。程序中用以下代码实现此操作：

```
for(j=begin+1;j<=state[i];j++){
Push(&station,j);
}
```

显然，begin应获取栈顶车厢号的值，即空（4）处应填入“Top(station)”。

还有一种情况，就是待考查的出栈序列还没有结束而栈空了，则说明需要处理入栈序列，使其车厢入栈。程序中用maxNO表示A端正待入栈的车厢编号，相应的处理如下面代码所示：

begin=maxNO;

for(j=begin;j<=state[i];j++){

Push(&station,j);

}

接下来，A端正待入栈的车厢编号等于j或state[i]+1，即空（5）处应填入j或“state[i]+1”。

如果驶出的车厢编号序列是经由栈获得的，则程序运行时输出该序列及字符串“OK”否则输出“error”而结束。

试题5参考答案

（1）InitStack(&station)

（2）!IsEmpty(station)

（3）state[i]<Top(station)

（4）Top(station)

（5）j

考点突破

根据考试大纲，本章要求考生掌握以下几个方面的知识点。

（1）分析模式与设计模式知识

（2）面向对象程序设计知识

（3）用C++语言实现常见的设计模式及应用程序。

（4）用Java语言实现常见的设计模式及应用程序。

从历年的考试情况来看，本章的考点主要集中于：设计模式基本概念、设计模式的分类、设计模式的特点与应用场合以及面向对象程序语言与设计模式思想的综合案例。

考点精讲

1.什么是设计模式

设计模式是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式可以提高代码复用度、让代码更容易被他人理解、保证代码可靠性。毫无疑问，设计模式于己、于他人、于系统都是有利的，设计模式使代码编制真正工程化，设计模式是软件工程的基石，

如同大厦的一块块砖石一样。

2.设计模式的组成

一般来说，一个模式有4个基本成分，分别是模式名称、问题、解决方案和效果。

（1）模式名称

每个模式都有一个名字，帮助我们讨论模式和它所给出的信息。模式名称通常用来描述一个设计问题、它的解法和效果，由一到两个词组成。模式名称的产生使我们可以在更高的抽象层次上进行设计并交流设计思想。

（2）问题

问题告诉我们什么时候要使用设计模式、解释问题及其背景。例如，MVC（Model-View-Controler，模型-视图-控制器）模式关心用户界面经常变化的问题。它可能描述诸如如何将一个算法表示成一个对象这样的特殊设计问题。在应用这个模式之前，也许还要给出一些该模式的适用条件。

（3）解决方案

解决方案描述设计的基本要素，它们的关系、各自的任务以及相互之间的合作。解决方案并不是针对某一个特殊问题而给出的。设计模式提供有关设计问题的一个抽象描述以及如何安排这些基本要素以解决问题。一个模式就像一个可以在许多不同环境下使用的模板，抽象的描述使我们可以把该模式应用于解决许多不同的问题。

模式的解决方案部分给出了如何解决再现问题，或者更恰当地说是如何平衡与之相关的强制条件。在软件体系结构中，这样的解决方案包括两个方面。

第一，每个模式规定了一个特定的结构，即元素的一个空间配置。例如，MVC模式的描述包括以下语句：“把一个交互应用程序划分成三部分，分别是处理、输入和输出”。

第二，每个模式规定了运行期间的行为。例如，MVC模式的解决方案部分包括以下陈述：“控制器接收输入，而输入往往是鼠标移动、点击鼠标按键或键盘输入等事件。事件转换成服务请求，这些请求再发送给模型或视图”。

（4）效果

效果描述应用设计模式后的结果和权衡。比较与其他设计方法的异同，得到应用设计模式的代价和优点。对于软件设计来说，通常要考虑的是空间和时间的权衡。也会涉及到语言问题和实现问题。对于一个面向对象的设计而言，可重用性很重要，效果还包括对系统灵活性、可扩充性及可移植性的影响。明确看出这些效果有助于理解和评价设计模式。

3. 设计模式的分类

在设计模式概念提出以后，很多人把自己的一些成功设计规范成了设计模式，一时间提出了许许多多的模式，但这些模式并没有都为众人所接受，成为通用的模式。直到ErichGamma在他的博士论文中总结了一系列的设计模式，做出了开创性的工作。他用一种类似分类目录的形式将设计模式记载下来。我们称这些设计模式为设计模式目录。根据模式的目标（所做的事情），可以将它们分成创建性模式（creational）、结构性模式（structural）和行为性模式（behavioral）。创建性模式处理的是对象的创建过程，结构性模式处理的是对象/类的组合，行为性模式处理类和对象间的交互方式和任务分布。根据它们主要的应用对象，又可以分为主要应用于类的和主要应用于对象的。

表15-1是ErichGamma等人总结的23种设计模式，这些设计模式通常被称为GoF（Gang of

Four，四人帮）模式。因为这些模式是在《Design Patterns: Elements of Reusable Object-Oriented Software》中正式提出的，而该书的作者是Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides，这几位作者常被称为“四人帮”。

表15-1设计模式目录的分类

目的	设计模式	简要说明	可改变的方面
创建型	Abstract Factory 抽象工厂模式	提供一个接口，可以创建一系列相关或相互依赖的对象，而无需指定它们具体的类	产品对象族
	Builder 生成器模式	将一个复杂类的表示与其构造相分离，使得相同的构建过程能够得出不同的表示	如何建立一种组合对象
	Factory Method* 工厂方法模式	定义一个创建对象的接口，但由子类决定需要实例化哪一个类。 工厂方法使得子类实例化的过程推迟	实例化子类的对象
	Prototype 原型模式	用原型实例指定创建对象的类型，并且通过拷贝这个原型来创建新的对象	实例化类的对象
	Singleton 单子模式	保证一个类只有一个实例，并提供一个访问它的全局访问点	类的单个实例
结构型	Adapter* 适配器模式	将一个类的接口转换成用户希望得到的另一种接口。它使原本不相容的接口得以协同工作	与对象的接口
	Bridge 桥模式	将类的抽象部分和它的实现部分分离开来，使它们可以独立地变化	对象的实现
	Composite 组合模式	将对象组合成树型结构以表示“整体-部分”的层次结构，使得用户对单个对象和组合对象的使用具有一致性	对象的结构和组合
	Decorator 装饰模式	动态地给一个对象添加一些额外的职责。它提供了用子类扩展功能的一个灵活的替代，比派生一个子类更加灵活	无子类对象的责任
	Façade 外观模式	定义一个高层接口，为子系统中的一组接口提供一个一致的外观，从而简化了该子系统的使用	与子系统的接口
	Flyweight 享元模式	提供支持大量细粒度对象共享的有效方法	对象的存储代价
	Proxy 代理模式	为其他对象提供一种代理以控制这个对象的访问	如何访问对象,对象位置
行为型	Chain of Responsibility 职责链模式	通过给多个对象处理请求的机会，减少请求的发送者与接收者之间的耦合。将接收对象链接起来，在链中传递请求，直到有一个对象处理这个请求	可满足请求的对象
	Command 命令模式	将一个请求封装为一个对象，从而可用不同的请求对客户进行参数化，将请求排队或记录请求日志，支持可撤销的操作	何时及如何满足一个请求
	Interpreter* 解释器模式	给定一种语言，定义它的文法表示，并定义一个解释器，该解释器用来根据文法表示来解释语言中的句子	语言的语法和解释
	Iterator 迭代器模式	提供一种方法来顺序访问一个聚合对象中的各个元素，而不需要暴露该对象的内部表示	如何访问、遍历聚合的元素
	Mediator 中介者模式	用一个中介对象来封装一系列的对象交互。它使各对象不需要显式地相互调用，从而达到低耦合，还可以独立地改变对象间的交互	对象之间如何交互及哪些对象交互
	Memento 备忘录模式	在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，从而可以在以后将该对象恢复到原先保存的状态	何时及哪些私有信息存储在对象之外
	Observer 观察者模式	定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并自动更新	依赖于另一对象的数量
	State 状态模式	允许一个对象在其内部状态改变时改变它的行为	对象的状态
	Strategy 策略模式	定义一系列算法，把它们一个个封装起来，并且使它们之间可互相替换，从而让算法可以独立于使用它的用户而变化	算法

	Template Method* 模板模式	定义一个操作中的算法骨架，而将一些步骤延迟到子类中，使得子类可以不改变一个算法的结构即可重新定义算法的某些特定步骤	算法的步骤
	Visitor 访问者模式	表示一个作用于某对象结构中的各元素的操作，使得在不改变各元素的类的前提下定义作用于这些元素的新操作	无需改变其类而可应用于对象的操作

其中带*为关于类的，其他是关于对象的。

在后面的章节中，我们将详细论述23种模式的特点、应用场合及UML图。

上一节　本书简介　下一节

一点一练

试题1

在面向对象软件开发过程中，采用设计模式__(1)__。

（1）A．以复用成功的设计

B．以保证程序的运行速度达到最优值

C．以减少设计过程创建的类的个数

D．允许在非面向对象程序设计语言中使用面向对象的概念

试题2

设计模式根据目的进行分类，可以分为创建型、结构型和行为型三种。其中结构型模式用于处理类和对象的组合。__(2)__模式是一种结构型模式。

（2）A．适配器（Adapter）B．命令（Command）

C．生成器（Builder）D．状态（State）

试题3

在进行面向对象设计时，采用设计模式能够__(3)__。

（3）A．复用相似问题的相同解决方案 B．改善代码的平台可移植性

C．改善代码的可理解性 D．增强软件的易安装性

试题4

设计模式具有__(4)__的优点。

（3）A．适应需求变化 B．程序易于理解

C．减少开发过程中的代码开发工作量 D．简化软件系统的设计

上一节　本书简介　下一节

解析与答案