第3章 蛮力法

CHAPTER

教学重点	蛮力法的设计思想,各种经典问题的蛮力思想				
教学难点	串匹配问题,凸包问题				
教学内容和教学目标	知识点	教学要求			
		了解	理解	掌握	熟练掌握
	蛮力法的设计思想				√
	顺序查找				√
	串匹配问题			√	
	选择排序和起泡排序				√
	0/1 背包问题			√	
	任务分配问题		√		
	哈密顿回路问题		√		
	TSP 问题			√	
	最近对问题			√	
	凸包问题		√		

3.1 概 述

3.1.1 蛮力法的设计思想

蛮力法(brute force method,也称穷举法或枚举法)是一种简单直接地解决问题的方法,常常直接基于问题的描述,所以,蛮力法也是最容易应用的方法[①]。例如,对于给定的整数 a 和非负整数 n,计算 a^n 的值,最直接最简单的想法就是把 1 和 a 相乘 n 次,即:

① 想一想我们之前写过的算法,大多数是应用蛮力法设计的。

$$a^n=\underbrace{a\times a\times\cdots\times a}_{n\text{次}}$$

应 用 实 例

计算 a^n 的值是非对称加密算法 RSA 的主要组成部分。RSA 算法的加密和解密过程都要对一个整数求整数次幂再取模。例如,设公钥为(5,119),私钥为(77,119),明文 $m=19$,进行如下加密得到密文 c:

$$c=19^5 \bmod 119=66$$

进行如下解密得到明文 m:

$$m=66^{77} \bmod 119=19$$

因此,计算 a^n 算法的效率直接影响到 RSA 算法的性能。

蛮力法所依赖的基本技术是**遍历**(traversal,也称扫描),即采用一定的策略依次处理待求解问题的所有元素,从而找出问题的解。依次处理所有元素是蛮力法的关键,为了避免陷入重复试探,应保证处理过的元素不再被处理。用蛮力法设计的算法,一般来说,都可以对算法的第一个版本进行一定程度的改进,提高其时间性能,但只能减少系数,而数量级不会改变。

由于蛮力法需要依次穷举待处理的元素,因此,用蛮力法设计的算法其时间性能往往也是最低的,典型的指数时间算法一般都是采用蛮力穷举。但是,基于以下原因,蛮力法也是一种重要的算法设计技术:

(1) 理论上,蛮力法可以解决可计算领域的各种问题。对于一些非常基本的问题,例如,求一个序列中的最大元素,计算 n 个数的和等,蛮力法是一种常用的算法设计技术。

(2) 蛮力法经常用来解决一些较小规模的问题。如果要解决问题的输入规模不大,用蛮力法设计的算法其时间是可以接受的,此时,设计一个更高效算法的代价是不值得的。

(3) 对于一些重要的问题(例如排序、查找、串匹配),蛮力法可以设计一些合理的算法,这些算法具有实用价值,而且不受输入规模的限制。

(4) 蛮力法可以作为某类问题时间性能的下界,来衡量同样问题的其他算法是否具有更高的效率。

3.1.2 一个简单的例子——百元买百鸡问题

【问题】 已知公鸡5元一只,母鸡3元一只,小鸡1元三只,用100元钱买100只鸡,问公鸡、母鸡、小鸡各多少只?

【想法】 设公鸡、母鸡和小鸡的个数分别为 x、y 和 z,则有如下方程组成立:

$$\begin{cases}x+y+z=100\\5\times x+3\times y+z/3=100\end{cases}\quad\text{且}\quad\begin{cases}0\leqslant x\leqslant 20\\0\leqslant y\leqslant 33\\0\leqslant z\leqslant 100\end{cases}$$

则百元买百鸡问题转换为求方程组的解。应用蛮力法求方程组的解只能依次试探变量 x、y 和 z 的值，验证 x、y 和 z 的某个特定值是否能够使方程组成立。

【算法】 设变量 x 表示公鸡的个数，y 表示母鸡的个数，z 表示小鸡的个数，注意到方程组可能有多个解，需要输出所有满足条件的解，设变量 count 表示解的个数，算法用伪代码描述如下。

算法 3.1：百元买百鸡问题
输入：无
输出：公鸡、母鸡和小鸡的个数
1. 初始化解的个数 count＝0；
2. 循环变量 x 从 0～20 重复执行下述操作：
　2.1　循环变量 y 从 0～33 重复执行下述操作：
　　2.1.1　z＝100－x－y；
　　2.1.2　如果 5＊x＋3＊y＋z/3 等于 100，则 count＋＋；输出 x、y 和 z 的值；
　　2.1.3　y＋＋；
　2.2　x＋＋；
3. 如果 count 等于 0，则输出无解信息；

【算法分析】 算法由两层嵌套的循环组成，可以将判等操作(5＊x＋3＊y＋z/3 等于 100)作为基本语句，其执行次数为 21×34＝714。

【算法实现】 注意到小鸡 1 元三只，则小鸡的个数应该是 3 的倍数，因此，在判断总价是否满足方程时要先判断 z 是否是 3 的倍数。算法用 C++ 语言描述如下：

```
void Chicken()
{
    int x, y, z;                              //x、y 和 z 分别表示公鸡、母鸡和小鸡的个数
    int count=0;                              //解的个数初始化为 0
    for (x=0; x<=20; x++)                     //公鸡个数 x 的范围是 0 到 20
    {
        for (y=0; y<=33; y++)                 //母鸡个数 y 的范围是 0 到 33
        {
            z=100-x-y;                        //满足方程 x+y+z=100
            if ((z%3==0) && (5*x+3*y+z/3==100))       //满足总价是 100 元
            {
                count++;                      //解的个数加 1
                cout<<"公鸡："<<x<<"母鸡："<<y<<"小鸡："<<z<<endl;
            }
        }
    }
    if (count==0)
        cout<<"问题无解"<<endl;
}
```

3.2 查找问题中的蛮力法

应用蛮力法求解查找问题就是依次考察待查找集合,显然这是最笨拙的查找方法。但是,在某些情况下(如查找集合不大)也是一种合理的方法。

3.2.1 顺序查找

【问题】 **顺序查找**(sequential search)是指在查找集合中依次查找值为 k 的元素,若查找成功,则给出该元素在查找集合中的位置;若查找失败,则给出失败信息。

【想法1】 将查找集合存储在一维数组中,然后从数组的一端向另一端逐个将元素与待查值进行比较,若相等,则查找成功,给出该元素在查找集合中的序号;若整个数组检测完仍未找到与待查值相等的元素,则查找失败,给出失败的标志0。注意在查找过程中要检测比较位置是否越界。

【算法实现1】 将查找集合存储在数组r[1]~r[n]中,下标i初始化在数组的高端,这样查找结束时,若查找成功,i的值即为元素的序号,若查找失败,i的值即为失败标志0。顺序查找算法用C++语言描述如下:

```
int SeqSearch1(int r[ ], int n, int k)      //数组 r[1] ~r[n]存放查找集合
{
    int i=n;
    while (i>0 && r[i] !=k)                 //注意检测比较位置是否越界
        i--;
    return i;
}
```

【算法分析1】 算法SeqSearch1的基本语句是i>0和r[i]!=k[①],设 p_i 表示查找第 i 个元素的概率,等概率情况下,$p_i=1/n$,则基本语句的执行次数为:

$$\sum_{i=1}^{n} p_i b_i + \sum_{i=1}^{n} p_i c_i = \frac{1}{n}\sum_{i=1}^{n}(n-i+1) + \frac{1}{n}\sum_{i=1}^{n}(n-i+1) = n+1 = O(n)$$

【想法2】 为了避免在查找过程中每一次比较前都要判断查找位置是否越界,可以设置**观察哨**(sentinel),即将待查值放在查找方向的"尽头"处,则比较位置 i 至多移动到下标0处。改进的查找过程如图3.1所示。

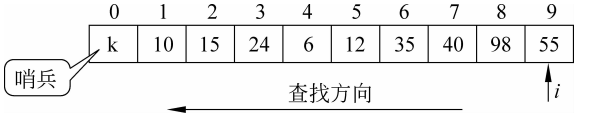

图3.1 改进的顺序查找示意图

① 基本语句也可以是i--,但此处需要细致分析所有关键语句的执行次数,所以,将i>0和r[i]!=k作为基本语句更合适。

【算法实现 2】 改进的顺序查找算法用 C++ 语言描述如下：

```
int SeqSearch2(int r[ ], int n, int k)      //数组 r[1] ~r[n]存放查找集合
{
    int i=n;                                //从数组高端开始查找
    r[0]=k;                                 //设置哨兵
    while (r[i] !=k)                        //不用检测比较位置是否越界
        i--;
    return i;
}
```

【算法分析 2】 算法 SeqSearch2 的基本语句是 r[i]!=k，设 p_i 表示查找第 i 个元素的概率，等概率情况下，基本语句的执行次数为：

$$\sum_{i=1}^{n} p_i c_i = \frac{1}{n}\sum_{i=1}^{n}(n-i+1) = \frac{n+1}{2} = O(n)$$

顺序查找算法和其改进算法的时间复杂性都是 $O(n)$，而系数相差一倍。实验表明，改进算法在表长大于 1000 时，进行一次顺序查找的时间几乎减少一半。

3.2.2　串匹配问题

【问题】 给定两个字符串 S 和 T，在主串 S 中查找子串 T 的过程称为**串匹配**(string matching，也称模式匹配)，T 称为模式。在文本处理系统、操作系统、编译系统、数据库系统以及 Internet 信息检索系统中，串匹配是使用最频繁的操作。

串匹配问题具有下面两个明显的特征：

(1) 问题的输入规模很大，常常需要在大量信息中进行匹配，因此，算法的一次执行时间不容忽视；

(2) 匹配操作经常被调用，执行频率高，因此，算法改进所取得的效益因积累往往比表面上看起来要大得多。

应 用 实 例

在 Word 等文本编辑器中有这样一个功能：在“查找”对话框中输入待查找内容(常见的是查找某个字或词)，编辑器会在整个文档中进行查找，并将与待查找内容相匹配的部分高亮显示。

【想法 1】 应用蛮力法解决串匹配问题的过程是：从主串 S 的第一个字符开始和模式 T 的第一个字符进行比较，若相等，则继续比较二者的后续字符；若不相等，则从主串 S 的第二个字符开始和模式 T 的第一个字符进行比较，重复上述过程，若 T 中的字符全部比较完毕，则说明本趟匹配成功；若 S 中的字符全部比较完毕，则匹配失败。这个算法称为朴素的模式匹配算法，简称 **BF 算法**，如图 3.2 所示。

设主串 S="abcabcacb"，模式 T="abcac"，BF 算法的匹配过程如图 3.3 所示。

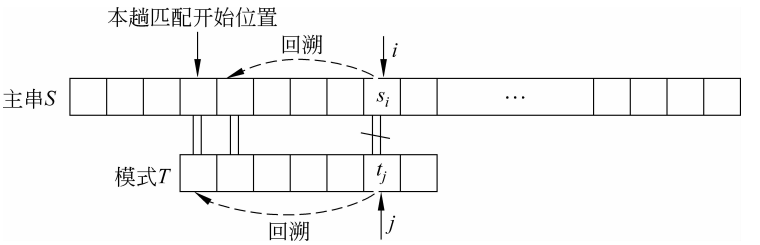

图 3.2 BF 算法的基本思想

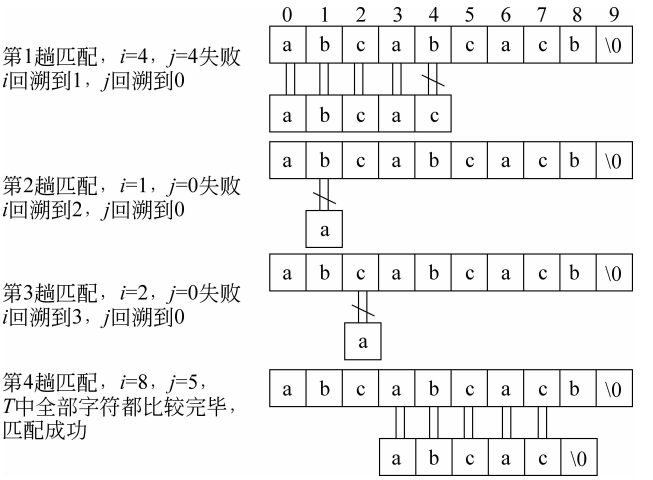

图 3.3 BF 算法的执行过程

【算法 1】 设字符数组 S 存放主串,字符数组 T 存放模式,BF 算法用伪代码描述如下。

算法 3.2：串匹配算法 BF
输入：主串 S,模式 T
输出：T 在 S 中的位置
1. 初始化主串比较的开始位置 index=0；
2. 在串 S 和串 T 中设置比较的起始下标 i=0，j=0；
3. 重复下述操作,直到 S 或 T 的所有字符均比较完毕：
   3.1 如果 S[i]等于 T[j],则继续比较 S 和 T 的下一对字符；
   3.2 否则,下一趟匹配的开始位置 index++,回溯下标 i=index,j=0；
4. 如果 T 中所有字符均比较完,则返回匹配的开始位置 index;否则返回 0；

【算法分析 1】 设主串 S 长度为 n,模式 T 长度为 m,在匹配成功的情况下,考虑最坏情况,即每趟不成功的匹配都发生在模式 T 的最后一个字符。

例如：S="aaaaaaaaaab"

T="aaab"

设匹配成功发生在 s_i 处,则在 $i-1$ 趟不成功的匹配中共比较了$(i-1)\times m$ 次,第 i 趟成功的匹配共比较了 m 次,所以总共比较了 $i\times m$ 次,因此平均比较次数是：

$$\sum_{i=1}^{n-m+1} p_i \times (i \times m) = \sum_{i=1}^{n-m+1} \frac{1}{n-m+1} \times (i \times m) = \frac{m(n-m+2)}{2}$$

一般情况下，$m \ll n$，因此最坏情况下的时间复杂性是 $O(n \times m)$。

【算法实现 1】 BF 算法用 C++ 语言描述如下：

```
int BF(char S[ ], char T[ ])
{
    int index=0;                              //主串从下标 0 开始第一趟匹配
    int i=0, j=0;                             //设置比较的起始下标
    while ((S[i] !='\0') && (T[j] !='\0'))
    {
        if (S[i]==T[j]) {i++; j++;}
        else {index++; i=index; j=0; }        //i 和 j 分别回溯
    }
    if (T[j]=='\0') return index+1;           //返回本趟匹配的开始位置(不是下标)
    else return 0;
}
```

【想法 2】 分析 BF 算法的执行过程，造成 BF 算法效率低的原因是回溯，即在某趟匹配失败后，对于主串 S 要回溯到本趟匹配开始字符的下一个字符，模式 T 要回溯到第一个字符，而这些回溯往往是不必要的。观察图 3.3 所示的匹配过程，在第 1 趟匹配过程中，S[0]～S[3]和 T[0]～ T[3]匹配成功，S[4]≠T[4]匹配失败，因此有了第 2 趟。因为在第 1 趟中有 S[1]＝T[1]，而 T[0]≠T[1]，因此有 T[0]≠S[1]，所以第 2 趟是不必要的，同理第 3 趟也是不必要的，可以直接到第 4 趟。进一步分析第 4 趟中的第一对字符 S[3]和 T[0]的比较是多余的，因为第 1 趟中已经比较了 S[3]和 T[3]，并且 S[3]＝T[3]，而 T[0]＝T[3]，因此必有 S[3]＝T[0]，因此第 4 趟比较可以从第二对字符 S[4]和 T[1]开始进行，这就是说，第 1 趟匹配失败后，下标 i 不回溯，而是将下标 j 回溯至第 2 个字符，从 T[1]和 S[4]开始进行比较。

综上所述，希望某趟在 S[i]和 T[j]匹配失败后，下标 i 不回溯，下标 j 回溯至某个位置 k，从 T[k]和 S[i]开始进行比较。显然，关键问题是如何确定位置 k？

观察部分匹配成功时的特征，某趟在 S[i]和 T[j]匹配失败后，下一趟比较从 S[i]和 T[k]开始，则有 T[0]～T[k－1]＝S[i－k]～S[i－1]成立，如图 3.4(a)所示；在部分匹配成功时，有 T[j－k]～T[j－1]＝S[i－k]～S[i－1]成立，如图 3.4(b)所示。

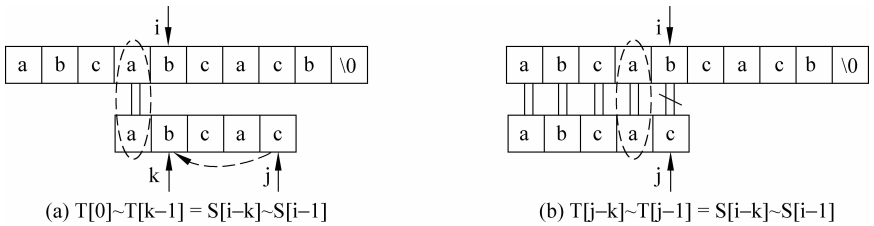

图 3.4　部分匹配时的特征

由 T[0]～T[k－1]＝S[i－k]～S[i－1]和 T[j－k]～T[j－1]＝S[i－k]～S[i－1]，可得：

$$T[0]\sim T[k-1]=T[j-k]\sim T[j-1]$$

上式说明，模式中的每一个字符 T[j]都对应一个 k 值，这个 k 值仅依赖于模式本身，与主串无关，且 T[0]～T[k－1]是 T[0]～T[j－1]的真前缀，T[j－k]～T[j－1]是 T[0]～T[j－1]的真后缀，k 是 T[0]～T[j－1]的真前缀和真后缀相等的最大子串的长度。用 next[j]表示 T[j]对应的 k 值(0≤j＜m)，其定义如下：

$$\text{next}[j]=\begin{cases}-1 & j=0\\ \max\{k\mid 1\leqslant k<j \text{ 且 } T[0]\cdots T[k-1]=T[j-k]\cdots T[j-1]\} & \text{集合非空}\\ 0 & \text{其他情况}\end{cases}$$

设模式 T＝"ababc"，根据 next[j]的定义，计算过程如下：

j＝0 时，next[0]＝－1

j＝1 时，next[1]＝0

j＝2 时，T[0]≠T[1]，则 next[2]＝0

j＝3 时，T[0]T[1]≠T[1]T[2]，T[0]＝T[2]，则 next[3]＝1

j＝4 时，T[0]T[1]T[2]≠T[1]T[2]T[3]，T[0]T[1]＝T[2]T[3]，则 next[4]＝2

设主串 S＝"ababaababcb"，模式 T＝"ababc"，模式 T 的 next 值为{－1，0，0，1，2}，改进的串匹配算法(称为 **KMP 算法**[①])的匹配过程如图 3.5 所示。

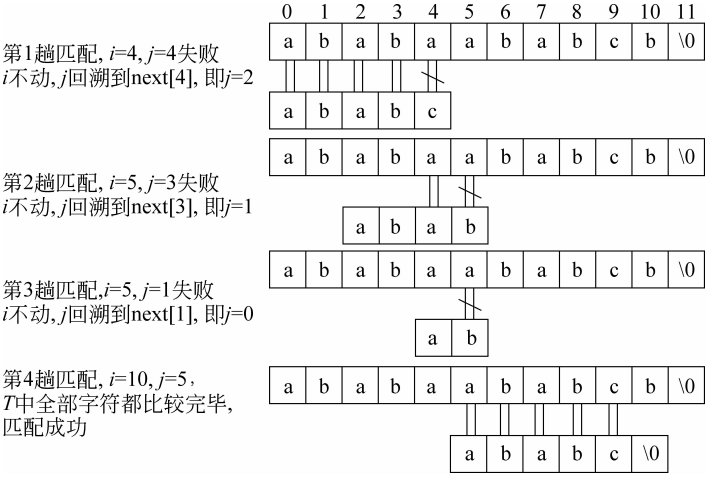

图 3.5 KMP 算法的执行过程

【算法 2】 在求得了模式 T 的 next 值后，KMP 算法用伪代码描述如下。

算法 3.3：串匹配算法 KMP

输入：主串 S，模式 T

输出：T 在 S 中的位置

① KMP 算法是由 Knuth、Morris 和 Pratt 同时设计的。

1. 在串 S 和串 T 中分别设置比较的起始下标 i=0，j=0；
2. 重复下述操作，直到 S 或 T 的所有字符均比较完毕：
   2.1　如果 S[i]等于 T[j]，则继续比较 S 和 T 的下一对字符；
   2.2　否则，将下标 j 回溯到 next[j]位置，即 j=next[j]；
   2.3　如果 j 等于－1，则将下标 i 和 j 分别加 1，准备下一趟比较；
3. 如果 T 中所有字符均比较完毕，则返回本趟匹配的开始位置；否则返回 0；

【算法分析 2】 在求得模式 T 的 next 值后，KMP 算法只需将主串扫描一遍，设主串的长度为 n，则 KMP 算法的时间复杂性是 $O(n)$。

【算法实现 2】 可以用蛮力法求得模式 T 的 next 值，KMP 算法用 C++ 语言描述如下：

```
void GetNext(char T[ ], int next[ ])  //求模式 T 的 next 值
{
    int i, j, len;
    next[0]=-1;
    for (j=1; T[j]!='\0'; j++)        //依次求 next[j]
    {
        for (len=j-1; len>=1; len--)//相等子串的最大长度为 j-1
        {
            for (i=0; i<len; i++)     //依次比较 T[0]~T[len-1]与 T[j-len]~T[j-1]
                if (T[i] !=T[j-len+i]) break;
            if (i==len)
            {
                next[j]=len; break;
            }
        }
        if (len<1) next[j]=0;         //其他情况，无相等子串
    }
}
int KMP(char S[ ], char T[ ])         //求 T 在 S 中的序号
{
    int i=0, j=0;
    int next[80];                     //假定模式最长为 80 个字符
    GetNext(T, next);
    while (S[i] !='\0' && T[j] !='\0')
    {
        if (S[i]==T[j])
        {
            i++; j++;
        }
        else {
            j=next[j];
            if (j==-1) {i++; j++;}
```

```
        }
    }
    if (T[j]=='\0') return (i-strlen(T)+1);     //返回本趟匹配的开始位置
    else return 0;
}
```

3.3 排序问题中的蛮力法

选择排序和起泡排序是运用蛮力法解决排序问题的最直观例子,它们都以一种清晰的方式运用了蛮力法。

3.3.1 选择排序

【问题】 应用选择排序方法对一个记录序列进行升序排列。**选择排序**(selection sort)的基本思想是:第 i 趟排序在无序序列 $r_i \sim r_n$ 中找到值最小的记录,并和第 i 个记录交换作为有序序列的第 i 个记录,如图 3.6 所示。

【想法】 图 3.7 给出了一个选择排序的例子(无序区用方括号括起来),具体的排序过程如下:

(1) 将整个记录序列划分为有序区和无序区,初始时有序区为空,无序区含有待排序的所有记录。

(2) 在无序区查找值最小的记录,将它与无序区的第一个记录交换,使得有序区扩展一个记录,同时无序区减少一个记录。

(3) 不断重复步骤(2),直到无序区只剩下一个记录为止。

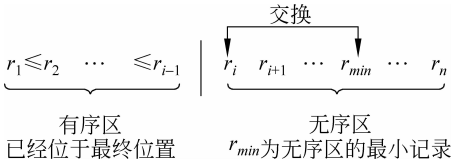

图 3.6 简单选择排序的基本思想图解

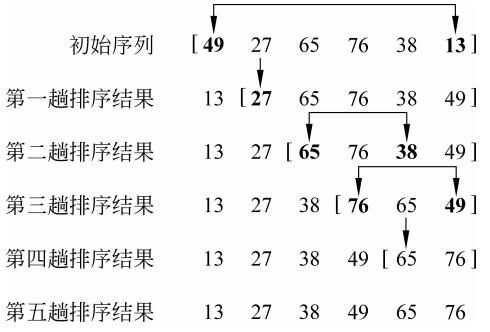

图 3.7 简单选择排序的过程示例

【算法实现】 设数组 r[n]存储待排序记录序列,注意到 C++ 语言的数组下标从 0 开始,则第 i 个记录存储在 r[i−1]中。选择排序算法用 C++ 语言描述如下:

```
void SelectSort(int r[ ], int n)
{
    int i, j, index, temp;
    for (i=0; i<n-1; i++)                    //对 n 个记录进行 n-1 趟选择排序
```

```
    {
        index=i;
        for (j=i+1; j<n; j++)                          //在无序区中查找最小记录
            if (r[j]<r[index]) index=j;
        if (index !=i)
        {
            temp=r[i]; r[i]=r[index]; r[index]=temp;    //交换记录
        }
    }
}
```

【算法分析】 该算法的基本语句是内层循环体中的比较语句(r[j]<r[index]),其执行次数为:

$$\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1}1=\sum_{i=0}^{n-2}(n-i-1)=\frac{n(n-1)}{2}=O(n^2)$$

因此,对于任何待排序记录序列,选择排序算法的时间性能都是 $O(n^2)$。但是,选择排序算法记录交换次数最多为 $n-1$ 次,因为外层循环每执行一次,交换记录的语句最多只执行一次,从移动元素的角度讲,选择排序算法优于许多其他排序算法。

3.3.2　起泡排序

【问题】 应用起泡排序方法对一个记录序列进行升序排列。**起泡排序**(bubble sort)的基本思想是:两两比较相邻记录,如果反序则交换,直到没有反序的记录为止,如图 3.8 所示。

【想法】 图 3.9 给出了一个起泡排序的例子(方括号括起来的为无序区)。具体的排序过程如下:

(1) 将整个待排序的记录序列划分成有序区和无序区,初始时有序区为空,无序区包括所有待排序的记录。

(2) 对无序区从前向后依次比较相邻记录,若反序则交换,从而使得值较小的记录向前移,值较大的记录向后移(像水中的气泡,体积大的先浮上来)。

(3) 重复执行(2),直到无序区中没有反序的记录。

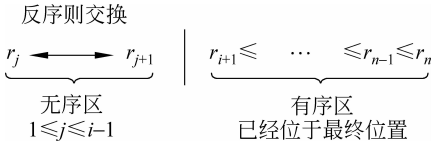

图 3.8　起泡排序的基本思想图解

初始键值序列	[50	13	55	97	27	38	49	65]
第一趟排序结果	[13	50	55	27	38	49	65]	**97**
第二趟排序结果	[13	50	27	38	49]	**55**	**65**	97
第三趟排序结果	[13	27	38	49]	**50**	55	65	97
第四趟排序结果	13	27	38	49	50	55	65	97

图 3.9　起泡排序过程示例

【算法实现】 注意到,在一趟起泡排序过程中,如果有多个记录交换到最终位置,则下一趟起泡排序将不处理这些记录;另外,在一趟起泡排序过程中,如果没有记录相交换,则表明序列已经有序,算法将终止。起泡排序算法请参见 2.1.1 节。

【算法分析】 参见 2.1.4 节。

3.4 组合问题中的蛮力法

对于组合问题来说,穷举试探是一种最简单的蛮力方法,也就是依次生成并考察解空间中的每一个组合对象,从中选出满足问题约束的解。对组合问题应用蛮力法,除非问题规模很小,否则会由于问题的解空间太大而产生组合爆炸现象。

3.4.1 0/1 背包问题

【问题】 给定 n 个重量为$\{w_1,w_2,\cdots,w_n\}$、价值为$\{v_1,v_2,\cdots,v_n\}$的物品和一个容量为 C 的背包,**0/1 背包问题**(0/1 knapsack problem)是求这些物品中的一个最有价值的子集,并且要能够装到背包中。

应 用 实 例

有 n 项可投资的项目,每个项目需要投入资金 s_i,可获利润为 v_i,现有可用资金总数为 M,应选择哪些项目来投资,才能获得最大利润。

【想法】 用蛮力法解决 0/1 背包问题,需要考虑给定 n 个物品集合的所有子集,找出所有总重量不超过背包容量的子集,计算每个可能子集的总价值,然后找到价值最大的子集。例如,给定 4 个物品的重量为{7,3,4,5},价值为{42,12,40,25},和一个容量为 10 的背包,表 3.1 给出了蛮力法求解 0/1 背包问题的过程。

表 3.1 蛮力法求解 0/1 背包问题的过程

序号	子集	总重量	总价值	序号	子集	总重量	总价值
1	ϕ	0	0	9	{2,3}	7	52
2	{1}	7	42	10	{2,4}	8	37
3	{2}	3	12	11	**{3,4}**	**9**	**65**
4	{3}	4	40	12	{1,2,3}	14	不可行
5	{4}	5	25	13	{1,2,4}	15	不可行
6	{1,2}	10	54	14	{1,3,4}	16	不可行
7	{1,3}	11	不可行	15	{2,3,4}	12	不可行
8	{1,4}	12	不可行	16	{1,2,3,4}	19	不可行

【算法】 蛮力法求解 0/1 背包问题的算法用伪代码描述如下。

算法 3.4:蛮力法求解 0/1 背包问题
输入:重量$\{w_1, w_2, \cdots, w_n\}$,价值$\{v_1, v_2, \cdots, v_n\}$,背包容量 C
输出:装入背包的物品编号

1. 初始化最大价值 maxValue=0;结果子集 S=Φ;
2. 对集合{1, 2, …, n}的每一个子集 T,执行下述操作:
   2.1　初始化背包的价值 value=0;背包的重量 weight=0;
   2.2　对子集 T 的每一个元素 j
      2.2.1　如果 weight+w_j<C,则 weight=weight+w_j;value=value+v_j;
      2.2.2　否则,转步骤 2 考察下一个子集;
   2.3　如果 maxValue<value,则 maxValue=value; S=T;
3. 输出子集 S 中的各元素;

【算法分析】 对于一个具有 n 个元素的集合,其子集数量是 2^n,所以,不论生成子集的算法效率有多高,蛮力法求解 0/1 背包问题都会导致一个 $\Omega(2^n)$的算法。

3.4.2　任务分配问题

【问题】 假设有 n 个任务需要分配给 n 个人执行,每个任务只分配给一个人,每个人只执行一个任务,且第 i 个人执行第 j 个任务的成本是 C_{ij} $(1\leqslant i,j\leqslant n)$,**任务分配问题**(task allocation problem)要求找出总成本最小的分配方案。

应 用 实 例

假设有 n 个建筑物要建在 n 个地点,c_{ij} 是在地点 j 建造建筑物 i 的成本,如何分配建筑任务,使得建造的总成本最少。

【想法】 可以用矩阵表示任务分配问题的成本,矩阵元素 C_{ij} $(1\leqslant i,j\leqslant n)$表示人员 i 执行任务 j 的成本。任务分配问题就是在成本矩阵中的每一行选取一个元素,这些元素分别属于不同的列,并且元素之和最小。可以用一个 n 元组$(j_1,j_2,\cdots,j_n)$来描述任务分配问题的一个可能解,其中第 i 个分量 j_i $(1\leqslant i\leqslant n)$表示在第 i 行中选择的列号,例如,(2,3,1)表示这样一种分配:任务 2 分配给人员 1、任务 3 分配给人员 2、任务 1 分配给人员 3。用蛮力法解决任务分配问题首先生成整数 1～n 的全排列,然后把成本矩阵中的相应元素相加求得每种分配方案的总成本,选出具有最小和的方案。

	任务1	任务2	任务3
人员1	9	2	7
人员2	6	4	3
人员3	5	8	1

图 3.10　一个成本矩阵

例如,图 3.10 是一个任务分配问题的成本矩阵,表 3.2 给出了蛮力法求解任务分配问题的过程。

表 3.2　蛮力法求解任务分配问题的过程

序号	分配方案	总成本	序号	分配方案	总成本
1	1, 2, 3	9+4+1=14	4	2, 3, 1	2+3+5=10
2	1, 3, 2	9+3+8=20	5	3, 1, 2	7+6+8=21
3	**2, 1, 3**	**2+6+1=9**	6	3, 2, 1	7+4+5=16

【算法】 蛮力法求解任务分配问题的算法用伪代码描述如下。

算法 3.5：蛮力法求解任务分配问题
输入：成本矩阵 C[n][n]
输出：最小总成本
1. 初始化最小总成本 minCost=MAX；
2. 对集合{1，2，…，n}的每一个全排列 $p_1 p_2 \cdots p_n$ 执行下述操作：
   2.1 初始化当前方案的总成本 cost = 0；
   2.2 对排列的每一个元素 p_i，执行 cost=cost+C[i][p_i]
   2.3 如果 cost<minCost，则 minCost=cost；
3. 输出 minCost；

【算法分析】 任务分配问题需要考虑的全排列数量是 $n!$，因此，其时间下界是 $\Omega(n!)$。所以，除了该问题的一些规模非常小的实例，蛮力法几乎是不实用的。

3.5 图问题中的蛮力法

3.5.1 哈密顿回路问题

【问题】 著名的爱尔兰数学家哈密顿(William Hamilton)提出的周游世界问题。假设一个正十二面体的 20 个顶点代表 20 个城市，**哈密顿回路问题**(Hamilton cycle problem)要求从一个城市出发，经过每个城市恰好一次，然后回到出发城市。图 3.11 所示是一个正十二面体的展开图，按照图中的顶点编号所构成的回路，就是哈密顿回路的一个解。

【想法】 蛮力法求解哈密顿回路的基本思想是，对于给定的无向图 $G=(V,E)$，依次考察图中所有顶点的全排列，满足以下两个条件的全排列(v_{i1}，v_{i2}，…，v_{in})构成的回路就是哈密顿回路：

(1) 相邻顶点之间存在边，即(v_{ij}，v_{ij+1})$\in E$($1 \leqslant j \leqslant n-1$)；

(2) 最后一个顶点和第一个顶点之间存在边，即(v_{in}，v_{i1})$\in E$。

例如，对于图 3.12 所示无向图，表 3.3 给出了蛮力法求解哈密顿回路的过程。

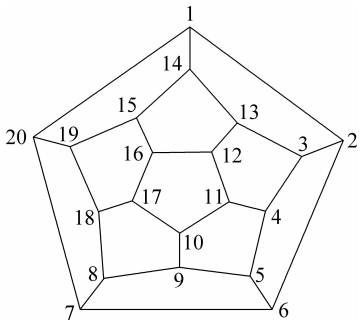

图 3.11 哈密顿回路问题示意图

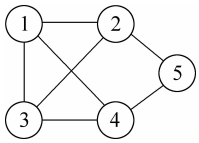

图 3.12 一个无向图

表 3.3　蛮力法求解哈密顿回路的过程

序号	路径	$(v_{ij}, v_{ij+1}) \in E$	$(v_{in}, v_{i1}) \in E$
1	12345	1→2→3→4→5(是)	否
2	12354	1→2→3　5→4(否)	是
3	12435	1→2　4→3　5(否)	否
4	12453	1→2　4→5　3(否)	是
5	12534	1→2→5　3→4(否)	是
6	**12543**	**1→2→5→4→3(是)**	**是**

【算法】 蛮力法求解哈密顿回路的算法用伪代码描述如下。

算法 3.6：蛮力法求解哈密顿回路
输入：无向图 $G=(V, E)$
输出：如果存在哈密顿回路，则输出该回路，否则，输出无解信息
1. 对顶点集合{1，2，…，n}的每一个排列 $v_{i1}v_{i2}\cdots v_{in}$ 执行下述操作：
　1.1　循环变量 j 从 1～$n-1$ 重复执行下述操作：
　　1.1.1　如果顶点 v_{ij} 和 v_{ij+1} 之间不存在边，则转步骤 1 考察下一个排列；
　　1.1.2　否则 j＋＋；
　1.2　如果 v_{in} 和 v_{i1} 之间存在边，则输出排列 $v_{i1}v_{i2}\cdots v_{in}$，算法结束；
2. 输出无解信息；

【算法分析】 算法 3.6 在找到某一条哈密顿回路后，即可结束算法，例如表 3.3 试探了 6 个全排列后找到了一条哈密顿回路。但是，最坏情况下需要考察顶点集合的所有全排列，其时间复杂性为 $O(n!)$。

3.5.2　TSP 问题

【问题】 TSP 问题是指旅行家要旅行 n 个城市然后回到出发城市，要求各个城市经历且仅经历一次，并要求所走的路程最短。

应 用 实 例

某工厂生产各种颜色的汽车，总共有 n 种颜色。随着给汽车所上颜色的转换，生产线上每台机器都需要从一种颜色切换到另一种颜色，其转换开销取决于转换的两种颜色以及它们的顺序。例如，从黄色切换到黑色需要 30 个单位的开销，从黑色切换到黄色需要 80 个单位的开销，从黄色切换到绿色需要 35 个单位的开销等。要解决的问题是找到一个最优生产调度，使得颜色转换的总开销最少。

【想法】 蛮力法求解 TSP 问题的基本思想是，找出所有可能的旅行路线，即依次考

察图中所有顶点的全排列,从中选取路径长度最短的哈密顿回路(也称为简单回路)。例如,对于图3.13所示无向带权图,表3.4给出了用蛮力法求解TSP问题的过程。

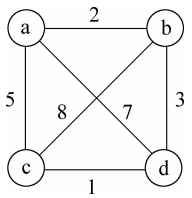

图3.13 一个无向带权图

表3.4 蛮力法求解TSP问题的过程

序号	路径	路径长度	是否最短
1	a→b→c→d→a	18	否
2	**a→b→d→c→a**	**11**	**是**
3	a→c→b→d→a	23	否
4	**a→c→d→b→a**	**11**	**是**
5	a→d→b→c→a	23	否
6	a→d→c→b→a	18	否

【算法】 蛮力法求解TSP问题与求解哈密顿回路问题类似,不同的仅是将回路经过的边上的权值相加得到相应的路径长度,具体算法请读者自行设计。

【算法分析】 蛮力法求解TSP问题必须依次考察顶点集合的所有全排列,从中找出路径长度最短的简单回路,因此,其时间下界是$\Omega(n!)$。

3.6 几何问题中的蛮力法

3.6.1 最近对问题

【问题】 最近对问题要求在一个包含n个点的集合中找出距离最近的两个点。严格地讲,距离最近的点对可能多于一对,简单起见,只找出其中的一对即可。

应 用 实 例

在空中交通控制问题中,若将飞机作为空间中移动的一个点来处理,则具有最大碰撞危险的两架飞机,就是这个空间中最接近的一对点。这类问题是计算几何中研究的基本问题之一。

【想法】 简单起见,只考虑二维的情况,并假设所讨论的点以标准笛卡儿坐标形式给出。因此,两个点$p_i(x_i,y_i)$和$p_j(x_j,y_j)$之间的距离是**欧几里得距离**(Euclidean distance):

$$d = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

蛮力法求解最近对问题的过程是显而易见的:分别计算每一对点之间的距离,然后找出距离最小的那一对。

【算法实现】 为了避免对同一对点计算两次距离,可以只考虑$i<j$的点对(p_i,p_j)。注意到在求欧几里得距离时,可以免去求平方根操作,因为如果被开方的数越小,则它的平方根也越小。算法用C++语言描述如下:

```
int ClosestPoints(int x[ ], int y[ ], int n)
{
    int index1, index2;                    //记载最近点对的下标
    int d, minDist=1000;                   //假设最大距离不超过 1000
    for (int i=0; i<n-1; i++)
        for (int j=i+1; j<n; j++)          //只考虑 i<j 的点对
        {
            d=(x[i]-x[j]) * (x[i]-x[j])+(y[i]-y[j]) * (y[i]-y[j]);
            if (d<minDist)
            {
                minDist=d;
                index1=i; index2=j;
            }
        }
    cout<<"最近的点对是："<<index1<<"和"<<index2<<endl;
    return minDist;
}
```

【算法分析】 算法 ClosestPoints 的基本语句是计算两个点的欧几里得距离，该语句的主要操作是求平方，其执行次数为：

$$T(n)=\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1}2=\sum_{i=0}^{n-2}(n-i-1)=n(n-1)=O(n^2)$$

3.6.2　凸包问题

先定义什么是凸包。

定义 3.1　对于平面上一个点的有限集合，如果以集合中任意两点 P 和 Q 为端点的线段上的点都属于该集合，则称该集合是**凸集合**(convex set)。

显然，任意凸多边形都是凸集合，图 3.14 给出了一些凸集合和非凸集合的例子。

定义 3.2　一个点集 S 的**凸包**(convex hull)是包含 S 的最小凸集合，其中，最小是指 S 的凸包一定是所有包含 S 的凸集合的子集。

对于平面上 n 个点的集合 S，它的凸包就是包含所有这些点(或者在内部，或者在边界上)的最小凸多边形，最小凸多边形上的点称为凸包的**极点**(end point)。图 3.15 给出了一个凸包的例子，其中，凸包的极点是 p_1、p_2、p_6、p_7 和 p_4。

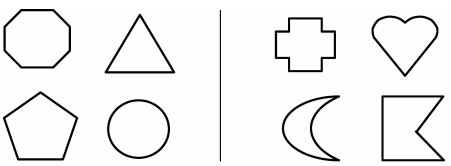

图 3.14　凸集合与非凸集合示例

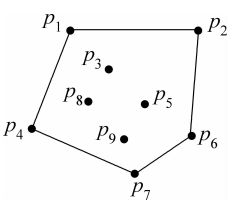

图 3.15　一个凸包的示例

应 用 实 例

在实际生活中,人脸检测技术有着广阔的应用空间。基于眼睛粗定位是将人脸区域送入分类器进行判别的人脸检测方法,该方法能够正确检测0°～360°旋转人脸图像,对于非监视环境下的人脸检测具有错误报警低、速度快的特点。

眼睛粗定位方法是一种基于彩色图像的定位方法。首先将皮肤部分提取出来,然后使用凸包填充算法,将大多数皮肤部分填充为凸包的形状,这样,就可以肯定在填充后的这些区域中,必定含有眼睛部分。

【问题】 凸包问题是为平面上具有 n 个点的集合 S 构造最小凸多边形。

【想法】 设集合 S 中经过两个点 p_i 和 p_j 的线段是 l_{ij},如果该集合中的其他点都位于线段 l_{ij} 的同一边(假定不存在三点同线的情况),则线段 l_{ij} 是该集合凸包边界的一部分。在平面上,经过两个点 $p_i(x_i,y_i)$ 和 $p_j(x_j,y_j)$ 的直线由下面的方程定义:

$$Ax+By+C=0 \quad (\text{其中},A=y_i-y_j,B=x_j-x_i,C=x_iy_j-x_jy_i)$$

这条直线把平面分成两个半平面:其中一个半平面中的点都满足 $Ax+By+C>0$,另一个半平面中的点都满足 $Ax+By+C<0$。可以基于这个原理来构造一个简单但缺乏效率的算法:对于点集 S 中的每一对顶点构成的线段,依次检验其余点是否位于这条线段的同一边,由于线段构成了凸包的边界,则满足条件的所有线段就构成了该凸包的边界。

【算法实现】 可以简单地把每个点代入表达式 $Ax+By+C$,检验这些表达式的符号是否相同。为了避免重复检验同一对点确定的直线表达式,可以只考虑 $i<j$ 的点对 (p_i,p_j)。算法用C++语言描述如下:

```
void ConvexHull(int x[ ], int y[ ], int n)
{
    int i, j, k, sign1, sign2;                    //sign1 和 sign2 表示两个半平面
    int A, B, C;
    for (i=0; i<n-1; i++)
        for (j=i+1; j<n; j++)
        {
            sign1=0; sign2=0;                     //初始化 sign1 和 sign2
            A=y[i]-y[j]; B=x[j]-x[i]; C=x[i]*y[j]-x[j]*y[i];
            for (k=0; k<n; k++)
            {
                if (k !=i && k !=j)
                {
                    if (A*x[k]+B*y[k]+C>0) sign1=1;
                    else sign2=1;
                    if (sign1==sign2) break;      //两个半平面均有点
                }
            }
```

```
            if (k==n)                                    //点 i 和 j 是极点
            cout<<"("<<i<<","<<j<<")"<<endl;
        }
    }
```

【算法分析】 所有不同的点组成了 $n(n-1)/2$ 条线段，对每条线段都要对其他 $n-2$ 个点求出在直线表达式 $Ax+By+C$ 中的符号，所以，算法的时间复杂性是 $O(n^3)$。

阅读材料——KMP 算法中 next 值的计算

【问题】 设模式的长度为 m，用蛮力法求解 KMP 算法中的 next 值时，$T[0]$可直接给出，一般情况下，计算 next$[j]$($1\leqslant j\leqslant m-1$)需要在 $T[0]\cdots T[j-1]$中分别取长度为 $j-1$、…、2、1 的真前缀和真后缀并比较是否相等，最坏情况下的时间代价是：

$$\sum_{j=1}^{m-1}\sum_{i=j-1}^{1} i=\sum_{j=1}^{m-1}\frac{j(j-1)}{2}=\frac{1}{2}\left(\sum_{j=1}^{m-1}j^2-\sum_{j=1}^{m-1}j\right)=O(m^3)$$

请改进蛮力法求解 KMP 算法中 next 值的算法，使其时间代价为 $O(m)$。

【想法】 模式 $T[0]T[1]\cdots T[m-1]$中的每一个字符 $T[j]$都对应一个 k 值，用 next$[j]$表示 $T[j]$对应的 k 值($0\leqslant j\leqslant m-1$)，其定义如下：

$$\text{next}[j]=\begin{cases}-1 & j=0\\ \max\{k\mid 1\leqslant k<j \text{ 且 } T[0]\cdots T[k-1]=T[j-k]\cdots T[j-1]\} & \text{集合非空}\\ 0 & \text{其他情况}\end{cases}$$

因为 $T[0]$既没有真前缀也没有真后缀，因此 next$[0]=-1$。假设已经计算出next$[0]$，next$[1]$，…，next$[j]$，如何计算 next$[j+1]$呢？设 $k=$next$[j]$，则 $T[0]\cdots T[k-1]=T[j-k]\cdots T[j-1]$，这意味着 $T[0]\cdots T[k-1]$是 $T[0]\cdots T[j-1]$的真前缀，同时 $T[j-k]\cdots T[j-1]$是 $T[0]\cdots T[j-1]$的真后缀。为了计算 next$[j+1]$，比较 $T[k]$和 $T[j]$，可能出现两种情况：

(1) $T[k]=T[j]$：说明 $T[0]\sim T[k-1]T[k]=T[j-k]\sim T[j-1]T[j]$，如图 3.16 所示。由 next 值的定义，next$[j+1]=k+1$。

图 3.16　$T[k]=T[j]$的情况

(2) $T[k]\neq T[j]$：此时要找出 $T[0]\cdots T[j-1]$的真前缀和真后缀相等的第 2 大子串，由于 $T[0]\cdots T[j-1]$的真前缀和真后缀相等的最大子串是 $T[0]\cdots T[k-1]$，而 next$[k]$的值为 $T[0]\cdots T[k-1]$的真前缀和真后缀相等的最大子串的长度，则 $T[0]\cdots T[\text{next}[k]-1]$即为 $T[0]\cdots T[j-1]$的真前缀和真后缀相等的第 2 大子串，如图 3.17 所示。令 $k=$next$[k]$，再比较 $T[k]$和 $T[j]$，此时仍会出现两种情况，当 $T[k]=T[j]$时，与情况(1)类似，此时，next$[j+1]=k+1$；当 $T[k]\neq T[j]$时，与情况(2)类似，再找出 $T[0]\cdots T[k-1]$的真前缀和真后缀相等的最大子串，重复(2)的过程，直到 $T[k]=T[j]$，

或 next[k]=−1,说明 $T[0]\cdots T[j-1]$不存在真前缀和真后缀相等的子串,此时,next[$j+1$]=0。

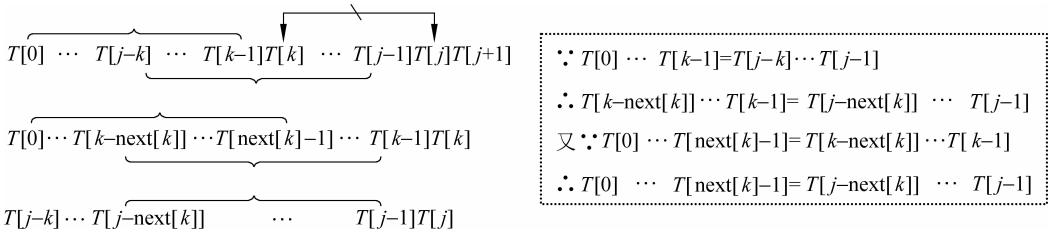

图 3.17 $T[k]\neq T[j]$的情况

例如,模式 T="abaababc"的 next 值计算如下:

j=0 时,next[0]=−1

j=1 时,k=next[0]=−1,next[1]=0

j=2 时,k=next[1]=0,$T[0]\neq T[1]$;k=next[k]=next[0]=−1,next[2]=0

j=3 时,k=next[2]=0,$T[0]=T[2]$,next[3]=k+1=0+1=1

j=4 时,k=next[3]=1,$T[1]\neq T[3]$;k=next[k]=next[1]=0,$T[0]=T[3]$,next[4]=k+1=1

j=5 时,k=next[4]=1,$T[1]=T[4]$,next[5]=k+1=1+1=2

j=6 时,k=next[5]=2,$T[2]=T[5]$,next[6]=k+1=2+1=3

j=7 时,k=next[6]=3,$T[3]\neq T[6]$;k=next[k]=next[3]=1,$T[1]=T[6]$,next[7]=k+1=2

【算法实现】 基于以上想法,求 next 值的算法用 C++ 语言描述如下:

```
void GetNext(char T[ ], int next[ ])
{
    int j=0, k=-1;
    next[0]=-1;
    while (T[j] !='\0')                    //直到字符串末尾
    {
        if (k==-1)                         //无相同子串
        {
            next[++j]=0; k=0;
        }
        else if (T[j]==T[k])               //确定 next[j+1]的值
            {
                k++;                       //相等子串长度加 1
                next[++j]=k;
            }
            else k=next[k];                //取 T[0]…T[j]的下一个相等子串的长度
    }
}
```