第 5 章 蛮 力 法

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。用蛮力法设计的算法其时间性能往往是最低的,典型的指数时间算法一般都是通过蛮力穷举得到的。

课件 5-1

5.1 概 述

5.1.1 蛮力法的设计思想

蛮力法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。通常来说,蛮力法是最容易应用的方法。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,再与 a 乘 $n-1$ 次。

依次处理所有元素是蛮力法的关键,应用蛮力法首先要确定穷举的范围,其次为了避免陷入重复试探,应保证处理过的元素不再被处理。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是最低的。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。对于一些基本的问题,例如求一个序列的最大元素,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(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,而且不受问题规模的限制。

(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。

5.1.2 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、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。

【算法实现】 设变量 x、y 和 z 分别表示公鸡、母鸡和小鸡的个数,由于方程组可能有多个解,设变量 count 表示解的个数,注意到小鸡1元三只,在判断总价是否满足方程时要先判断 z 是否是3的倍数。程序如下。

源代码 5-1

```
void Chicken( )
{
  int x, y, z, count = 0;
  for (x = 0; x <= 20; x++)
  {
    for (y = 0; y <= 33; y++)
    {
      z = 100 - x - y;                                    //满足共100只
      if ((z % 3 == 0) && (5 * x + 3 * y + z/3 == 100))   //满足总价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;
}
```

【算法分析】 算法由两层嵌套循环组成,基本语句是条件表达式((z%3 == 0) && (5 * x+3 * y+z/3 == 100)),执行次数为 $21\times 34=714$。一般地,假设共买 n 只鸡,基本语句的执行次数为 $\frac{n}{5}\times\frac{n}{3}=O(n^2)$。

5.2 查找问题中的蛮力法

课件 5-2

5.2.1 顺序查找

【问题】 顺序查找(sequential search)是在查找集合中依次查找值为 k 的元素。若

查找成功，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。

【算法实现 1】 将查找集合存储在数组元素 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，下标 i 的值即元素的序号，若查找失败，下标 i 的值即为失败标志 0。注意，在查找过程中要检测比较位置是否越界，程序如下：

源代码 5-2

```
int SeqSearch1(int r[ ], int n, int k)
{
  int i = n;
  while (i > 0 && r[i] != k)                    //检测比较位置是否越界
    i--;
  return i;
}
```

【算法分析 1】 算法 SeqSearch1 的时间主要耗费在条件表达式(i > 0 && r[i] != k)，设 p_i 表示查找第 i 个元素的概率，等概率情况下，执行次数为：

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

【想法 2】 为了避免在查找过程中每一次比较后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，也就是“哨兵”的位置。改进的查找过程如图 5-1 所示。

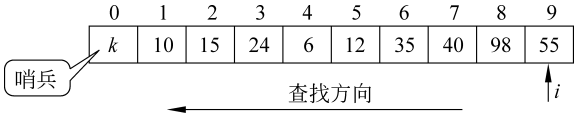

图 5-1　改进的顺序查找示意图

【算法实现 2】 设函数 SeqSearch2 实现改进的顺序查找算法，程序如下：

源代码 5-3

```
int SeqSearch2(int r[ ], int n, int k)
{
  int i = n;
  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的时间主要耗费在条件表达式(r[i] != k),设 p_i 表示查找第 i 个元素的概率,等概率情况下,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度都是 $O(n)$,但是算法SeqSearch1要执行两个判断,而算法SeqSearch2只执行一个判断。实验表明,改进算法在待查找集合的长度大于1000时,进行一次顺序查找的时间几乎减少一半。

5.2.2 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。串匹配问题具有两个明显的特征:①问题规模很大,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,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5-2所示。

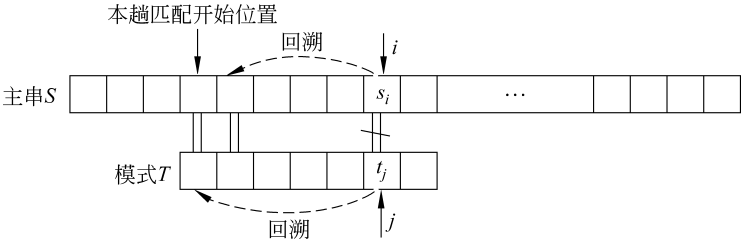

图5-2 BF算法的基本思想

设主串 S="abcabcacb",模式 T="abcac",BF算法的匹配过程如图5-3所示。

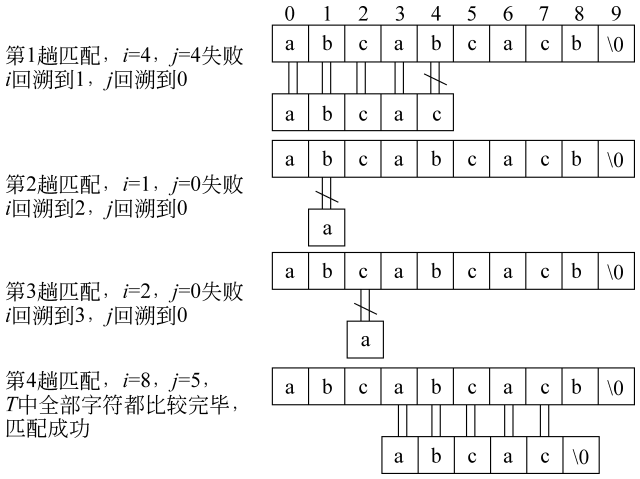

图 5-3　BF 算法的执行过程

【算法 1】　设字符数组 S 存放主串，字符数组 T 存放模式，BF 算法如下。

算法：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，则返回匹配开始位置的序号；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串 T 的最后一个字符。

例如：S="aaaaaaaaaaab"

　　T="aaab"

设匹配成功发生在 s_i 处，则在 $i-1$ 趟不成功的匹配共比较了 $(i-1)\times m$ 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，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度是 $O(n \times m)$。

【算法实现 1】　设字符数组 S 和 T 分别存储主串和模式，程序如下。

```
int BF(char S[ ], char T[ ])
{
```

源代码 5-4

```
    int index = 0, i = 0, j = 0;
    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;               //返回本趟匹配开始位置的序号
    else return 0;
  }
```

【想法2】 BF算法简单但效率较低,KMP算法[①]对于BF算法有了很大改进,基本思想是主串不进行回溯。

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5-3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5-4(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5-4(b)所示。

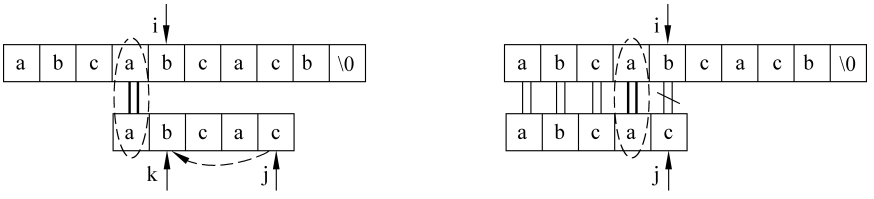

图5-4 部分匹配时的特征

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:

① KMP算法是克努思(Knuth)、莫里斯(Morris)和普拉特(Pratt)设计的。

$$T[0] \sim T[k-1] = T[j-k] \sim T[j-1] \tag{5-1}$$

式(5-1)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($0 \leqslant j < m$)，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＝"ababaabcbcb"，模式 T＝"ababc"，模式 T 的 next 值为{－1，0，0，1，2}，KMP 算法的匹配过程如图 5-5 所示。

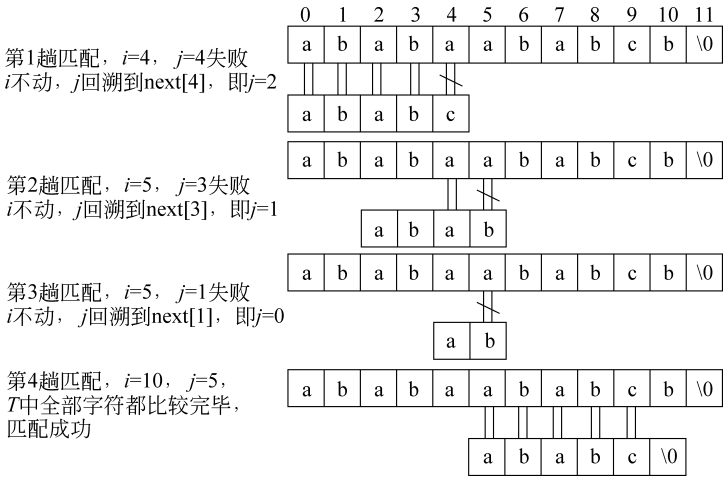

图 5-5　KMP 算法的执行过程

【算法 2】　在求得了模式 T 的 next 值后，KMP 算法如下。

算法：串匹配算法 KMP
输入：主串 S，模式 T
输出：T 在 S 中的位置
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,则返回本趟匹配开始位置的序号;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 算法的时间复杂度是 $O(n)$。

【算法实现 2】 设函数 GetNext 用蛮力法求得模式 T 的 next 值,函数 KMP 实现 KMP 算法,程序如下。

源代码 5-5

```
void GetNext(char T[ ], int next[ ])
{
  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++)         //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 - j + 1); //返回本趟匹配开始位置序号
  else return 0;
}
```

5.3　排序问题中的蛮力法

课件 5-3

5.3.1　选择排序

【问题】　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 5-6 所示。

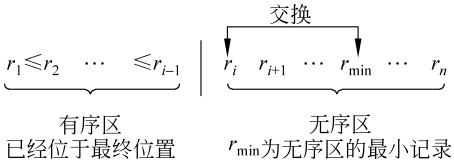

图 5-6　简单选择排序的基本思想图解

【想法】　图 5-7 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) 重复执行步骤(2)，直至无序区只剩下一个记录。

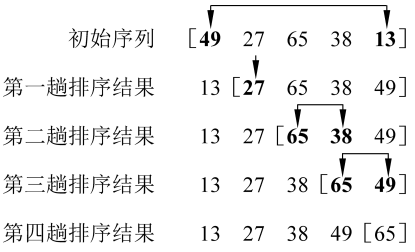

图 5-7　简单选择排序的过程示例

【算法实现】　设数组 r[n]存储待排序记录序列，注意，数组下标从 0 开始，则第 i 个记录存储在 r[i−1]中。程序如下。

源代码 5-6

```
void SelectSort(int r[ ], int n)
{
  int i, j, index, temp;
  for (i = 0; i < n - 1; i++)                    //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;
```

```
            temp = r[i]; r[i] = r[index]; r[index] = temp;     //交换记录
        }
    }
```

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

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

5.3.2 起泡排序

【问题】 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,直至没有反序的记录(图 5.8)。

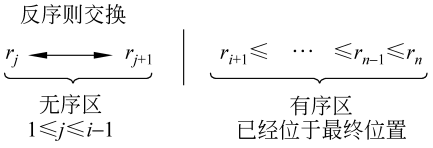

图 5-8 起泡排序的基本思想图解

【想法】 图 5-9 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,起泡排序因而得名)。

(3) 重复执行步骤(2),直至无序区没有反序的记录。

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 5-9 起泡排序过程示例

【算法实现】 注意,在一趟起泡排序过程中,如果有多个记录交换到最终位置,下一趟起泡排序将不再处理这些记录;另外,在一趟起泡排序过程中,如果没有交换记录操作,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。