北 京 林 业 大 学

2019 学年—2020 学年第 2 学期计算机算法设计与实践实验报告书

专业：计算机科学与技术(创新实验班)　班级：计创 18

姓　　名：连月菡　　学　　号：181002222

实验地点：家　　任课教师：王春玲

实验题目：实验 2 串匹配问题

实验环境：Visual Studio 2019 Community

一、 实验目的

（1）深刻理解并掌握蛮力法的设计思想；
（2）提高应用蛮力法设计算法的技能；
（3）理解这样一个观点：用蛮力法设计的算法，一般来说，经过适度的努力后，都可以对算法的第一个版本进行一定程度的改良，改进其时间性能。

二、 实验内容

给定一个文本，在该文本中查找并定位任意给定字符串。

三、 实验结果

主函数:
伪代码:

```
int main()
    输入主串 input ,输入模式 search
    调用函数,例如 KMP(),BF(),BM()
    输出位置 ans 和用时 t
```

C++代码:

```
int main()
{
    char input[1001];
    char search[1001];
    cout << "输入主串: ";
    cin >> input;
    cout << "输入模式: ";
    cin >> search;
```

```
    LARGE_INTEGER c1;//开始计时
    LARGE_INTEGER c2;//结束计时
    LARGE_INTEGER frequency;//计时器频率
    QueryPerformanceFrequency(&frequency);
    double quadpart = (double)frequency.QuadPart;
    QueryPerformanceCounter(&c1);
    int ans = BF(input, search);//此处可替换为 KMP,BM 函数
    QueryPerformanceCounter(&c2);
    cout << "位置: " << ans << endl;
    cout << "高精度计数器用时:
" << (double)((c2.QuadPart - c1.QuadPart) * 1.0 / quadpart * 1.0) * 10000000
00000 << endl;
    return 0;
}
```

（1） BF 算法；

算法思路：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。

伪代码：

```
int BF(char s[], char t[]) {

    while(字符串 s 和 t 的下标 i j 都指向字符串结束位置)
        if (s[i] ==t[j])
                下标 i<-i+1,j<-j+1
        else
            index<-index+1
             i = index; j = 0;
        endif
    endwhile
    if (主串 t[j] == '\0')
             return index + 1;
    else return 0;
    endif
}
```

C++代码：

```
int BF(char s[], char t[]) {
    int index = 0;//初始化变量
    int i = 0, j = 0;
    while ((s[i] != '\0') && (t[j] != '\0'))//当两个字符串下标都没有到字符串结束位置
    {
        if (s[i] == t[j]) { i++; j++; }//相等,下标都向后移动一位
        else {
            index++; i = index; j = 0;//不相等主串下标向后移动一位,模式串从头继续寻找
        }
    }
    if (t[j] == '\0') return index + 1;//如果模式串已经到末尾了,说明匹配成功
    else return 0;//没有到末尾,说明主串已经遍历完成,其中没有需要的模式串
}
```

（2） KMP 算法

算法思路：对于每一个模式串会事先通过 next()函数计算出模式串的内部匹配信息，在匹配失败时最大的移动模式串，利用匹配失败后的信息，尽量减少模式串与主串的匹配次数以达到快速匹配的目的。模式串尽量的右移和主串进行匹配。右移的距离是根据在已经匹配的模式串子串中，找出最长的相同的前缀和后缀，然后移动使它们重叠。

伪代码：

```
void GetNext(char T[], int next[])
{
    int j = 0, k = -1;
    next[0] = -1;
    while (T 匹配到结尾)
        if (k == -1)
            next 下标向后移动一位后的位置= 0;
            k = 0;
        else if (T[j]和 T[k]相等)
            k++;
            next 下标向后移动一位后的位置 = k;
        else k = next[k]; //取 T[0]…T[j]的下一个相等子串的长度
        endif
    endif
}
int nex[1001];
int KMP(char s[], char t[]) {
    int i = 0, j = 0;
    while 字符串 s 和 t 的下标 i j 都指向字符串结束位置
```

```
        if (s[i] == t[j])
            下标 i < -i + 1,j < -j + 1
        else j = nex[j];
        endif
        if (j == -1)
            下标 i < -i + 1, j < -j + 1
        endif
    endwhile
    if (t[j] == '\0') return 查找到的位置;
    else return 0;
    endif
}
```

C++代码:

```
(3)  void GetNext(char T[], int next[])
(4)  {
(5)      int j = 0, k = -1;
(6)      next[0] = -1;
(7)      while (T[j] != '\0')                //直到字符串末尾
(8)      {
(9)          if (k == -1) {                  //无相同子串
(10)             next[++j] = 0; k = 0;
(11)         }
(12)         else if (T[j] == T[k]) {            //确定 next[j+1]的值
(13)             k++;                    //相等子串长度加 1
(14)             next[++j] = k;
(15)         }
(16)         else k = next[k];           //取 T[0]…T[j]的下一个相等子串的长度
(17)     }
(18) }
(19) int nex[1001];
(20) int KMP(char s[], char t[]) {
(21)     int i = 0, j = 0;
(22)     while ((s[i] != '\0') && (t[j] != '\0'))
(23)     {
(24)         if (s[i] == t[j]) { i++; j++; }
(25)         else j = nex[j];//j 回退
(26)         if (j == -1) {
(27)             i++; j++;
(28)         }
(29)     }
(30)     if (t[j] == '\0') return i - strlen(t) + 1;//匹配成功，返回子串的位
     置
```

```
(31)       else return 0;//没找到
(32)   }
```

BM 算法；

算法思路：函数 dist 给出了正文中可能出现的任意字符在模式中的位置。将主串中自位置 i 起往左的一个子串与模式进行从右到左的匹配过程中，若发现不匹配，则下次应从主串的 i + dist (Si)位置开始重新进行新一轮的匹配，其效果相当于把模式和主串均向右滑过一段距离 dist (Si) ，即跳过 dist (Si)个字符而无需进行比较。

伪代码：

```
1.  int Dist(char t[], char c)//滑动距离函数
2.  {
3.      len = strlen(t)
4.       i = len - 1
5.      if c == t[i]
6.          return len
7.      endif
8.      i--
9.      while i >= 0
10.         if c == t[i]
11.             return len - 1 - i
12.         else
13.             i--
14.         endif
15.     endwhile
16.     return len
17. }
18.
19. int BM(char S[], char T[])
20. {
21.     n = strlen(S)
22.     m = strlen(T)
23.     i = m - 1
24.     j = m - 1
25.     while j >= 0 且 i < n
26.         if S[i] == T[j]
27.             i--
28.             j--
29.         else
30.             i += Dist(T, S[i])
31.             j = m - 1
32.         endif
33.     endwhile
```

```
    if j < 0
        return i + 1
    endif
    return -1
}
```

C++代码:

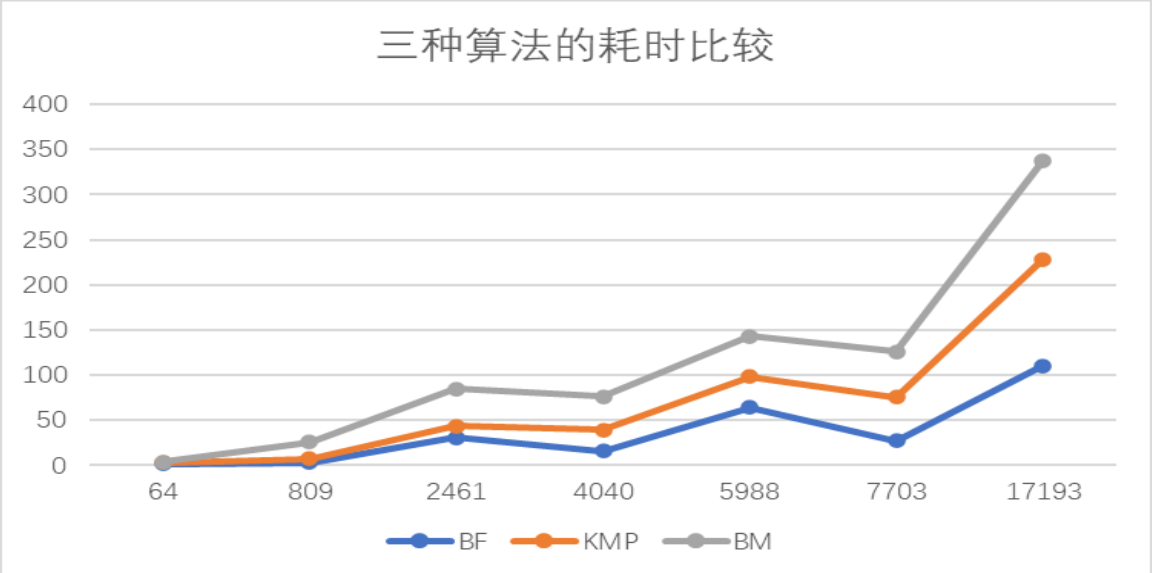
```
int Dist(char t[], char c)//滑动距离函数
{
    int len = strlen(t);
    int i = len - 1;
    if (c == t[i])
        return len;
    i--;
    while (i >= 0){
        if (c == t[i])
            return len - 1 - i;
        else
            i--;
    }
    return len;
}

int BM(char S[], char T[])
{
    int n = strlen(S);  int m = strlen(T);
    int i = m - 1;int j = m - 1;
    while (j >= 0 && i < n){//主串从 i 到左
        if (S[i] == T[j]){//匹配
            i--;j--;
        }
        else{//不匹配
            i += Dist(T, S[i]);//发现不匹配,开始新一轮匹配
            j = m - 1;
        }
    }
    if (j < 0)
        return i + 1;
    return -1;
}
```

（4）时间复杂性分析及分析结果的验证。

算法名称	时间复杂度
BF	$\sum_{i=1}^{n-m+1} p_i \times i \times m = \sum_{i=1}^{n-m+1} \frac{1}{n-m+1} \times i \times m = \frac{m(n-m+2)}{2}$ **T(n)=O(n *m)**
KMP	**运用摊还分析的聚合方法,得 T(n)=O(m)+O(n)=(m+n)**
BM	**最好情况 O(m/n),最坏情况 O(m • n)**

验证:利用搜索不同单词,更改它在同一个文本中出现的不同位置来比较耗时。

位置 / 算法	64	809	2461	4040	5988	7703	17193
BF	**2**	**3.1**	**30.7**	**16.2**	**64.3**	**27**	**110.4**
KMP	**0.8**	**4.6**	**13.3**	**23.3**	**33.8**	**48.7**	**117.8**
BM	**0.9**	**18.4**	**41.1**	**37.1**	**44.8**	**50.2**	**109.6**



*此折线图纵轴为时间的累积总和

由上述数据,可知:
在文本长度较小,数据不特殊的情况下,BF 算法耗时更少,KMP 算法发挥稳定,BM 算法耗时较长。在文本长度较长,数据复杂的情况下,根据计算出的时间复杂度,可知 KMP 算法较优,BF 较差,BM 算法有自身的特殊性。但总体来说,KMP 算法在时间复杂度上具有一定的优越性。

四、 实验中存在的问题及解决办法

Q1:一开始发现,无论如何改变查询位置,始终都是 BF 算法最快。
A1:用来测试的单词具有特殊性,不能体现另外两个算法的优势。于是修改了测试用例。
Q2:求 KMP 的时间复杂度
A2:查询算法导论电子书,学习如果计算 KMP 算法的时间复杂度。
Q3:有没有别的字符串查找算法?
A3:还有 Rabin-Karp 算法(时间复杂度为 O((n-m+1)*m+m),有限自动机算法(时间复杂度 O((m|Σ|+n))。