Lab4 串匹配算法

- **实验内容**

  > 使用朴素串匹配算法，Rabin-Karp算法，KMP算法，Boyer-Moore-Horspool算法对给定的匹配串和模式串进行串匹配

- **实验要求**

  > - 在5种不同规模下实现串匹配算法
  > - 每种串匹配算法分别输出在对应的output.txt中，输出的结果包括匹配串的规模，模式串的规模，第一次出现串匹配时首字符在匹配串中的位置，对应规模下程序的执行时间
  > - 对不同规模下程序的执行时间进行比较，得出每种算法的时间复杂度，并比较不同算法间时间复杂度的差异

- **实验设备和环境**

  > - 实验平台：MacOS上的Xcode集成IDLE
  > - 代码语言：C语言

- **实验方法**

  > - 在main函数中，采用了for循环，每一个循环中读入相应规模的数据，如下：
  >
  > ```
  > for( i = 1; i <= 5; i++ ) {
  >     n = pow(2,3*i+2);
  >     m = pow(2,i+1);
  >     fscanf(fp1,"%s",T);
  >     fscanf(fp1,"%s",P);
  > ```
  >
  > - 朴素匹配算法采用了两重for循环，对每个位置的字符串进行逐一比较，不匹配则用break跳出比较，当找到相应的匹配时，则输出匹配并结束匹配循环，如下：

```
void Naive_String_Match() {
    int flag;
    int s,j;
    flag = 0;
    for( s = 0; s <= n-m; s++ ){
        flag = 1;
        for( j = 0; j < m; j++ ){
            if( P[j] != T[s+j] ){
                flag = 0;
                break;
            }
        }
        if( flag ){
            fprintf(fp2,"文本串长度: %d\n",n);
            fprintf(fp2,"模式串长度: %d\n",m);
            fprintf(fp2,"首次匹配成功的起始位置: %d\n",s+1);
            break;
        }
    }
    if( !flag ){
        fprintf(fp2,"文本串长度: %d\n",n);
        fprintf(fp2,"模式串长度: %d\n",m);
        fprintf(fp2,"首次匹配成功的起始位置: -1\n");
    }
}
```

- Rabin-Karp算法在预处理的时候，对总共62个字符设置了一个新数组，用来存放对应的数字值，映射的建立是使用了ASCII码将需要的字符按顺序获取对应的值，按0～9，A～Z，a～z的顺序分别为0～61，q的值取了243，保证有q>m，预处理如下：

```
for( j = 0; j < n; j++ ) {
    if( T[j] >= 48 && T[j] <= 57 ) {
        t_num[j] = T[j]-48;
    }
    else if( T[j] >= 65 && T[j] <= 90 ) {
        t_num[j] = T[j]-55;
    }
    else {
        t_num[j] = T[j]-61;
    }
}
for( j = 0; j < m; j++ ) {
    if( P[j] >= 48 && P[j] <= 57 ) {
        p_num[j] = P[j]-48;
    }
    else if( P[j] >= 65 && P[j] <= 90 ) {
        p_num[j] = P[j]-55;
    }
    else {
        p_num[j] = P[j]-61;
    }
}
```

- 在Rabin-Karp算法中还有遇到一个问题：在循环中计算t的值时，t可能会出现负值的情况，此时不满足循环不变式。处理方法是在每次t值计算完成后，做如下操作：

```
if( i < n-m ) {
    t = (d*(t-t_num[i]*h)+t_num[i+m])%q;
    if( t < 0 ) t = t+q; //t可能会出现小于0的情况
}
```

- 在KMP算法中，需要预处理一个next数组，处理方法如下：

```
void Compute_Prefix_Function() {
    int q;
    int k;
    for( q = 0; q <= m; q++ ){
        next[q]=0;
    }
    next[0] = 0;
    k = 0;
    for( q = 1; q < m; q++ ) {
        while( k > 0 && P[k] != P[q] ){
            k = next[k];
        }
        if( P[k] == P[q] ) {
            k = k+1;
        }
        next[q] = k;
    }
}
```

- 在BMH算法中，在预处理Bc数组时，需要在字符的对应映射处存入模式串中从后往前第一次出现该字符的位置。在代码程序中，字符的位置信息被存储在Bc数组的该字符ASCII码值处，如下所示：

```
void Pre_Bc() {
    int i;
    for( i = 0; i < 130; i++ ) {
        Bc[i] = m;
    }
    for( i = 0; i < m-1; i++ ) {
        Bc[P[i]] = m-i-1; //模式串h中每一字母最后出现的下标，下标编号是从后往前
    }
}
```

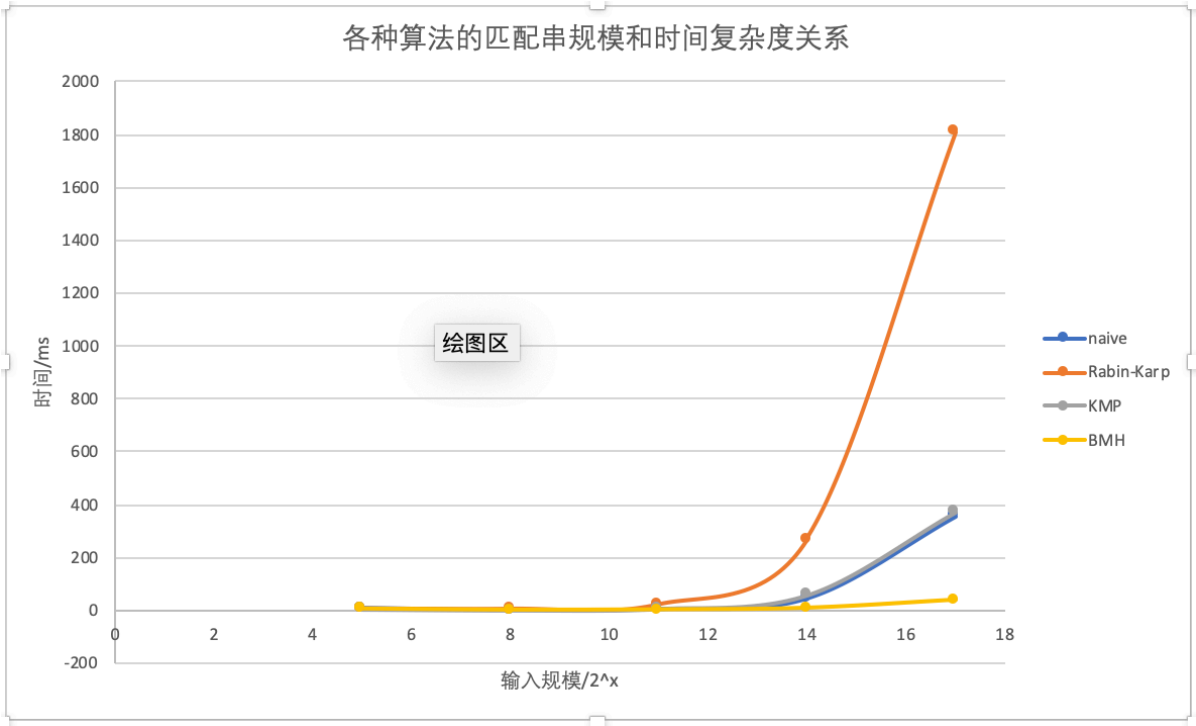
- 程序执行时间的统计采用了time.h库函数中的clock函数，使用begin和end两个double型变量，相应程序的执行时间为end-begin

- **实验步骤**

1. 创建对应文件夹：input，source，output
2. 从文件中读入数据
3. 完成代码部分
4. 调试程序，生成结果文件
5. 从结果中提取执行时间，作图分析时间复杂度

- **实验结果**

不同算法的时间复杂度图像如下：


各种算法的匹配串规模和时间复杂度关系
2000
1800
1600
1400
1200
1000
800
600
400
200
0
-200
时间/ms
绘图区
0
2
4
6
8
10
12
14
16
18
输入规模/2^x
naive
Rabin-Karp
KMP
BMH