

Project4 串匹配实验报告

徐煜森 PB16110173

一. 实验要求

实现字符串匹配算法，文本串 T 的长度为 n ，对应的模式串 P 的长度为 m ，字符串均是随机生成的字符 (A-F，共 6 种不同字符)。(n, m)共取五组数据: $(2^5, 2)$, $(2^8, 3)$, $(2^{11}, 4)$, $(2^{14}, 5)$, $(2^{17}, 6)$ 。

其中需要实现的算法有: Rabin-Karp 算法; KMP 算法; Boyer-Moore-Horspool 算法

二. 实验环境

1. Windows10 64 位 x86，机器内存 8G，时钟主频 2.59GHz
2. 软件环境: Visual Studio 2017

三. 实验过程

0. 编译选项

注意编译选项 `-std = c++11`

1. 生成数据

```

for(int i = 0; i < 5; i++){
    for(int j = 0; j < 1<<n[i]; j++){
        data<< char(rand()%6 + 'A');
    }
    data<<',';
    for(int j = 0; j < m[i]; j++){
        data<< char(rand()%6 + 'A');
    }
    data<<';';
}

```

2. Rabin Karp

- a. 按照书上方法实现算法，注意 C++ 中数组下标从 0 开始，在之后代码截图部分也将强调这一点。
- b. 书上考虑到模式串可能很长，对 ts 、 p 进行了 $\text{mod } q$ 计算，而本次实验中模式串最长为 6 位，故无需 $\text{mod } q$ ，因而也无需考虑伪命中情况，不过为了使算法可拓展、方便阅读，依然考虑了伪命中情况。
- c. 以下为运行结果截图

```

n=32 m=2
match_start_at -1

n=256 m=3
match_start_at -1

n=2048 m=4
pattern occurs with T+1245
match_start_at 1245

n=16384 m=5
pattern occurs with T+4116
match_start_at 4116

n=131072 m=6
pattern occurs with T+124709
pattern occurs with T+129260
match_start_at 124709

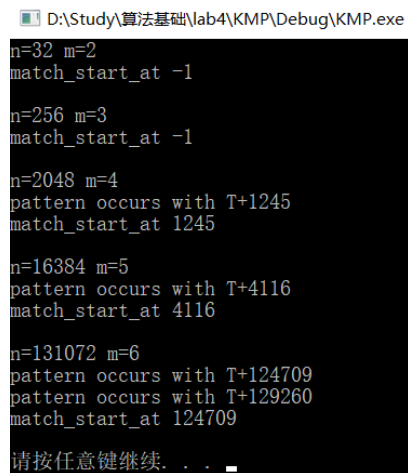
请按任意键继续. . .

```

3. KMP

a. 按照书上算法实现 KMP，同样需要注意数组下标从 0 开始

b. 以下为运行结果截图

A screenshot of a Windows command prompt window titled "D:\Study\算法基础\lab4\KMP\Debug\KMP.exe". The window displays the output of a KMP algorithm implementation. It shows several test cases with varying text lengths (n) and pattern lengths (m). For each case, it reports the match start index. The output is as follows:

```
n=32 m=2
match_start_at -1

n=256 m=3
match_start_at -1

n=2048 m=4
pattern occurs with T+1245
match_start_at 1245

n=16384 m=5
pattern occurs with T+4116
match_start_at 4116

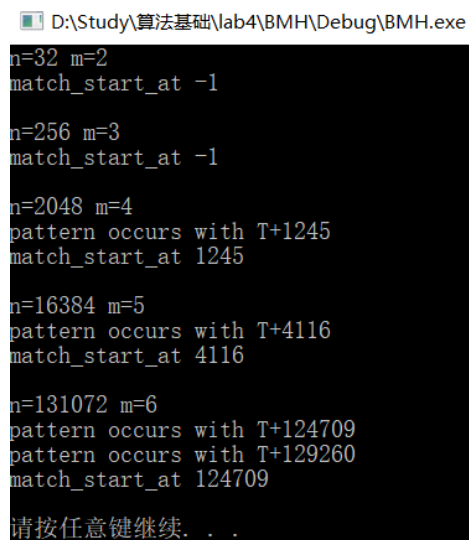
n=131072 m=6
pattern occurs with T+124709
pattern occurs with T+129260
match_start_at 124709

请按任意键继续. . .
```

4. BMH

a. 按照课件上算法实现 BMH，同样需要注意数组下标从 0 开始会导致偏移量不相同。

b. 以下为运行结果截图

A screenshot of a Windows command prompt window titled "D:\Study\算法基础\lab4\BMH\Debug\BMH.exe". The window displays the output of a BMH algorithm implementation. It shows several test cases with varying text lengths (n) and pattern lengths (m). For each case, it reports the match start index. The output is as follows:

```
n=32 m=2
match_start_at -1

n=256 m=3
match_start_at -1

n=2048 m=4
pattern occurs with T+1245
match_start_at 1245

n=16384 m=5
pattern occurs with T+4116
match_start_at 4116

n=131072 m=6
pattern occurs with T+124709
pattern occurs with T+129260
match_start_at 124709

请按任意键继续. . .
```

四. 关键代码截图

1. Rabin Karp

```
24      /* in this complement, s = s(in book) - 1 */
25
26      int n = strlen(T);
27      int m = strlen(P);
28      int d = 10;
29      int h = (int)pow((double)d, m - 1);
30      int p = 0;
31      int t = 0;
32
33      for (int i = 0; i < m; i++) {
34          p = d * p + P[i];
35          t = d * t + T[i];
36      }
37      for (int s = 0; s <= n - m; s++) {
38          if (p == t) {
39              if (memcmp(P, T + s, m) == 0) {
40                  cout << "pattern occurs with T+" << s << endl;
41                  if (match_start_at == -1) {
42                      match_start_at = s;
43                  }
44              }
45          }
46          if (s < n - m) {
47              t = d * (t - T[s] * h) + T[s + m];
48          }
49      }
```

从图中可以看到，计算 p 和 t 时没有像书上进行 $\text{mod } q$ 操作，因为实验中模式串最长为 6 位，可以使用一个 `int` 变量储存，但考虑到算法的可拓展性、可阅读性，依然加入伪命中判断。另外值得注意的是本代码中的 $s =$ 书上代码中的 $s - 1$ 。同时，记录下最早匹配到的字符串的起始下标。

2. KMP

- a. 以下为计算 KMP 辅助数组 pi 的函数，应注意这里数组中存的值表示已经匹配成功的字符的下标，-1 则表示没有字符成功匹配。

```

22 void ComputePrefixFunction() {
23     int m = strlen(P);
24     pi[0] = -1;
25     int k = -1;
26     for (int q = 1; q < m; q++) {
27         while (k > -1 && P[k + 1] != P[q]) {
28             k = pi[k];
29         }
30         if (P[k + 1] == P[q]) {
31             k = k + 1;
32         }
33         pi[q] = k;
34     }
35 }

```

b. 以下为 KMP 算法实现：

```

40 int n = strlen(T);
41 int m = strlen(P);
42 ComputePrefixFunction();
43 int q = -1; // index of characters matched
44 for (int i = 0; i < n; i++) {
45     while (q > -1 && P[q + 1] != T[i]) {
46         q = pi[q];
47     }
48     if (P[q + 1] == T[i]) {
49         q = q + 1;
50     }
51     if (q == m - 1) {
52         if (match_start_at == -1) {
53             match_start_at = i - m + 1;
54         }
55         cout << "pattern occurs with T+" << i - m + 1 << endl;
56     }
57     q = pi[q];
58 }
59 }

```

可以发现本代码实现的 KMP 算法中 q 的含义变化为已匹配成功的字符的下标，-1 则表示没有字符成功匹配。另外应注意下标的换算。

3. BMH

a. 以下为 BMH 辅助函数坏字符 Bc 数组的计算函数，其中 ASIZE 为字母表的大小，本实验中为 6。另外应注意偏移量的计算。

```

25 void PreBc(int i) {
26     for (int j = 0; j < ASIZE; j++) {
27         Bc[j] = m[i];
28     }
29     for (int j = 0; j < m[i] - 1; j++) {
30         Bc[P[j] - 'A'] = m[i] - j - 1;
31     }
32 }

```

b. 以下为 BMH 算法

```

37 int n = strlen(T);
38 int m = strlen(P);
39 int j;
40 char c;
41 PreBc(i);
42 j = 0;
43 while (j < n - m + 1) {
44     c = T[j + m - 1];
45     if (P[m - 1] == c && memcmp(P, T + j, m) == 0) {
46         cout << "pattern occurs with T+" << j << endl;
47         if (match_start_at == -1) {
48             match_start_at = j;
49         }
50     }
51     j += Bc[c - 'A'];
52 }

```

应注意的是下标计算与课件上有些区别。

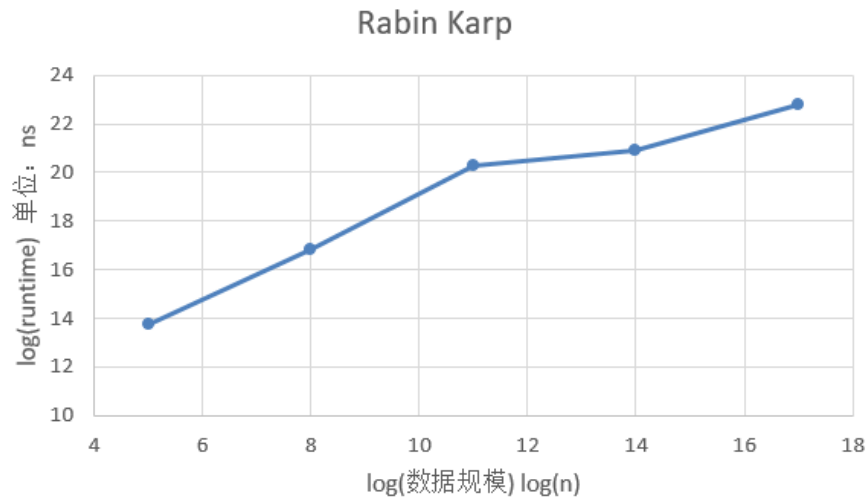
五. 实验结果及分析

1. Rabin Karp

a. 以下为 Rabin Karp 算法运行时间原数据，因为数据间差值太大，将对数据以 2 为底取 log 画图（KMP、BMH 同理）。

RabinKarp				
n	runtime(ns)		log(n)	log(runtime)
32	13827		5	13.7552
256	118504		8	16.85458
2048	1274468		11	20.28146
16384	1929086		14	20.87949
131072	7154565		17	22.77043

b. 以下为 $\log(n)-\log(\text{runtime})$ 图



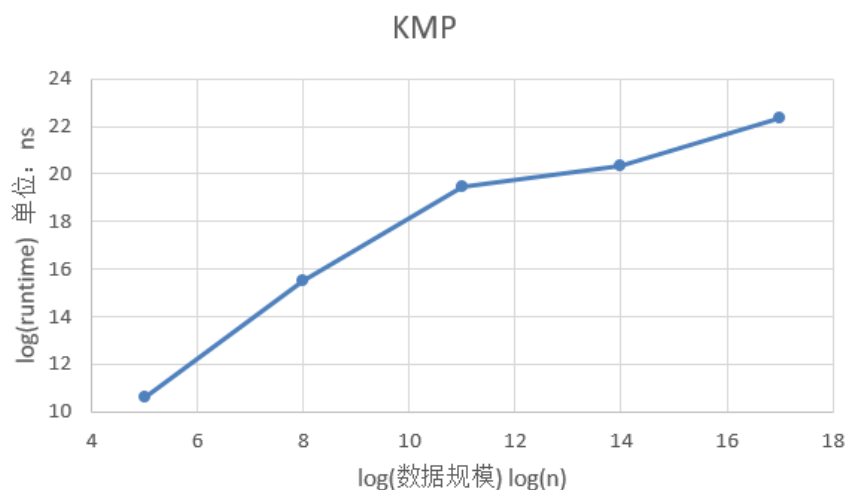
可以观察出，运行时间基本符合期望 $O(n)$ ，因为本次实验中数据量足够大，相较于之前实验更容易得到准确结果。曲线有些波动很有可能是受硬件 `cache` 和操作系统调换页的影响，几乎可以忽略不计。

2. KMP

a. 以下为 KMP 运行时间原数据

KMP			
n	runtime(ns)	$\log(n)$	$\log(\text{runtime})$
32	1580	5	10.62571
256	47411	8	15.53293
2048	713135	11	19.44382
16384	1320296	14	20.33243
131072	5462122	17	22.38103

b. 以下为 $\log(n)-\log(\text{runtime})$ 图



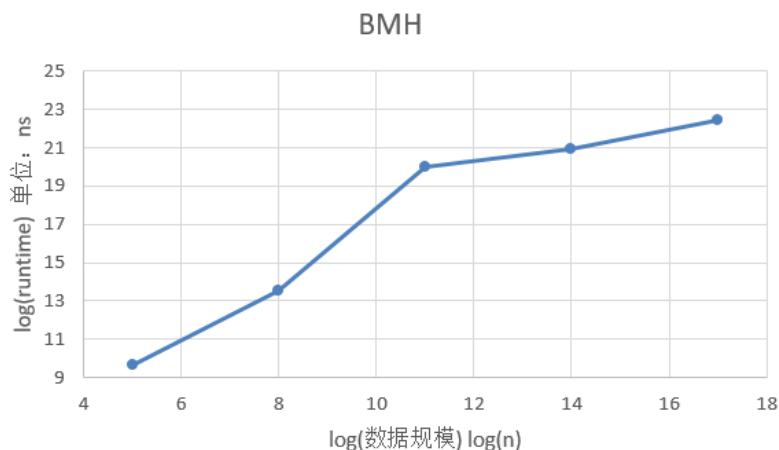
可以看出运行时间基本符合期望 $\Theta(n)$ ，这也是三个算法运行时间中曲线波动最小的，可以认为在本实验中 KMP 算法相较于 RK 和 BMH（见下面）算法表现更稳定。

3. BMH

a. 以下为 BMH 算法运行时间原数据

BMH				
n	runtime(ns)		log(n)	log(runtime)
32	791		5	9.627534
256	11875		8	13.53564
2048	1062715		11	20.01932
16384	1966617		14	20.90728
131072	5718121		17	22.44711

b. 以下为 $\log(n)-\log(\text{runtime})$ 图

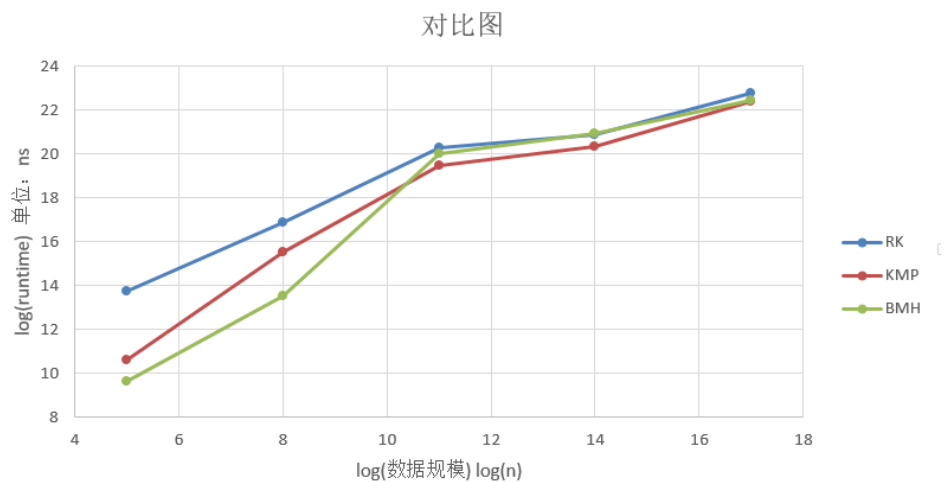


可以看出运行时间基本符合期望 $O(n)$ ，但曲线在中间有较小的向上波动，可能是受到硬件 **cache** 策略影响较大，如 **cache** 在运行第三组数据时，因其他程序也要占用一定存储空间的原因将 **BMH** 中使用的数组调换出一部分，在之后会引发较多的 **cache miss**。

我针对 **BMH** 算法查阅相关资料，发现 **BM** 和 **BMH** 算法在工业界应用比 **KMP** 算法要广泛的多。大家普遍认为 **BM** 算法比 **KMP** 算法要稳定，而且平均性能更好，尤其是在搜索引擎领域 **BM** 算法远好于 **KMP** 算法。

4. 对比图

将三个算法得到的三组数据画在同一张图上进行对比



可以看出三种算法在本次实验中的区分度并不大，渐进时间复杂度也与期望相同，均为 $O(n)$ 。

六. 实验总结

本次实验中让我对这三个算法有了更进一步的理解，同时也锻炼了我的对下标变换的理解能力。另外也通过查阅资料了解了更多关于字符串匹配算法的知识。