算法基础project4实验报告

PB19071535徐昊天

一.实验内容

1. KMP算法
2. Rabin-Karp算法

二.实验设备和环境

- windows操作系统，VMware虚拟机
- vscode、dev-c++
- Excel

三.实验方法和步骤

1.KMP算法

- 处理文件

  本次实验须通过相对路径读取 `input` 文件夹内的文件数据，经过算法处理后得到输出数据并通过相对路径写入 `output` 文件夹。涉及文件处理的代码如下：

```
FILE *fp1,*fp2,*fp3;
// 打开输入输出文件
if((fp1 = fopen("../input/4_1_input.txt","r"))==NULL)
{
    printf("cannot open input file\n");
    exit(0);
}
if((fp2 = fopen("../output/result.txt","w"))==NULL)
{
    printf("cannot open result.txt\n");
    exit(0);
}
if((fp3 = fopen("../output/time.txt","w"))==NULL)
{
    printf("cannot open time.txt\n");
    exit(0);
}
```

```
    ......
    ......
    ......
    //关闭输入输出文件
    fclose(fp1);
    fclose(fp2);
    fclose(fp3);
```

- 前缀函数

  模式的前缀函数π包含有模式与其自身的位移进行匹配的信息。这些信息可用于避免在朴素的字符串匹配算法中，对无用位移进行测试。模式P的前缀函数是函数满足如下公式：

$$\pi[q] = max(k : k < q且P_k \supset P_q)$$

  即π[q]是P[q]的真后缀P的最长前缀的长度。实现算法如下：

```
void compute_prefix_function()
{
    int m = strlen(P) - 1;
    int i, q;
    for (i = 0; i < m; i++)
        next[i] = 0;
    int k = 0;
    for (q = 2; q < m + 1; q++)
    {
        while (k > 0 && P[k + 1] != P[q])
        {
            k = next[k];
        }
        if (P[k + 1] == P[q])
        {
            k++;
        }
        next[q] = k;
    }
}
```

- KMP匹配算法

  调用辅助过程compute_prefix_function来计算π，并匹配字符串。实现算法如下：

```
void KMP_matcher(FILE *fp)
{
    int n = strlen(T) - 1;
    int m = strlen(P) - 1;
    compute_prefix_function();
    int q = 0;
    int i, num = 0;
    for (i = 1; i <= n; i++)
    {
        while (q > 0 && P[q + 1] != T[i])
            q = next[q];
        if (P[q + 1] == T[i])
```

```
            q++;
        if (q == m)
        {
            pos[num++] = i - m + 1;
            q = next[q];
        }
    }
}
```

- 输出信息

  根据以上KMP算法函数，利用 `num` 记录匹配次数， `pos` 数组记录所有匹配的T中开始位置， `next` 数组记录π的函数值，并按照要求输出到 `result.txt` 文件中。实现代码如下：

```
    fprintf(fp, "%d\n", num);
    for (i = 1; i < m + 1; i++)
        fprintf(fp, "%d ", next[i]);
    fprintf(fp, "\n");
    for (i = 0; i < num; i++)
        fprintf(fp, "%d ", pos[i]);
    fprintf(fp, "\n\n");
```

- 记录算法运行时间

  利用头文件 `#include<time.h>` 中 `clock()` 函数记录时间，结果以毫秒为单位，实现代码如下：

```
        clock_t start, end;
        double time;
        start = clock();
        KMP_matcher(fp2);
        end = clock();
        time = (double)(end - start) / (CLOCKS_PER_SEC);
        fprintf(fp3, "%f\n", time);
```

  得到时间单位为毫秒。

2.Rabin-Karp算法

- 处理文件

  本次实验须通过相对路径读取 `input` 文件夹内的文件数据，经过算法处理后得到输出数据并通过相对路径写入 `output` 文件夹。涉及文件处理的代码如下：

```
    FILE *fp1,*fp2,*fp3;
    // 打开输入输出文件
    if((fp1 = fopen("../input/4_2_input.txt","r"))==NULL)
    {
        printf("cannot open input file\n");
        exit(0);
    }
```

```
    if((fp2 = fopen("../output/result.txt","w"))==NULL)
    {
        printf("cannot open result.txt\n");
        exit(0);
    }
    if((fp3 = fopen("../output/time.txt","w"))==NULL)
    {
        printf("cannot open time.txt\n");
        exit(0);
    }
    ......
    ......
    ......
    //关闭输入输出文件
    fclose(fp1);
    fclose(fp2);
    fclose(fp3);
    return 0;
```

- 实现Rabin-Karp算法

  在该算法中，所有的字符都是d进制的数字。首先需将h初始化为m数位窗口中高位数字的值，其次计算出p的值为 `P[1..m] mod q` ，t的值为 `T[1..m] mod q` ，最后用for循环迭代所有可能的位移i，保持着如下的循环不变量：

$$t_i = T[i+1..i+m] mod\, q$$

  实现代码如下：

```
void rabin_karp_matcher(int d, int q)
{
    int n = strlen(T) - 1;
    int m = strlen(P) - 1;
    int i, j;
    int h = 1;
    num_r = num_f = 0;
    for (i = 0; i < m - 1; i++)
    {
        h = (h * d) % q;
    }
    int p = 0;
    for (i = 0; i < 4100; i++)
        t[i] = 0;
    for (i = 1; i <= m; i++)
    {
        p = ((d * p + P[i]) % q + q) % q;
        t[0] = ((d * t[0] + T[i]) % q + q) % q;
    }
    for (i = 0; i <= n - m; i++)
    {
        if ((p - t[i]) % q == 0)
        {
            for (j = 1; j <= m; j++)
            {
                if (P[j] != T[i + j])
                    break;
            }
```

```
            if (j == m + 1)
            {
                pos[num_r++] = i + 1;
            }
            else
            {
                num_f++;
            }
        }
        if (i < n - m)
        {
            t[i + 1] = (((d * (t[i] - T[i + 1] * h) + T[i + m + 1]) % q) + q) % q;
        }
    }
}

```

- 输出信息

  根据以上Rabin-Karp算法函数，利用 `num_r` 记录匹配次数， `pos` 数组记录所有匹配的T中开始位置， `num_f` 记录伪命中次数，并按照要求输出到 `result.txt` 文件中。实现代码如下：

```
    for (j = 0; j < 4; j++)
    {
        start = clock();
        rabin_karp_matcher(d[j], q[j]);
        end = clock();
        timer[j][i] = (double)(end - start) / (CLOCKS_PER_SEC);
        if (j == 0)
            fprintf(fp2, "%d\n", num_r);
        fprintf(fp2, "%d ", num_f);
    }
    fprintf(fp2, "\n");
    for (j = 0; j < num_r; j++)
        fprintf(fp2, "%d ", pos[j]);
    fprintf(fp2, "\n\n");
```

- 记录算法运行时间

  利用头文件 `#include<time.h>` 中 `clock()` 函数记录时间，结果以毫秒为单位，并利用二维数组 `timer` 记录不同(n,m)和(d,q)情况下的运行时间,实现代码如下：

```
    for (j = 0; j < 4; j++)
    {
        start = clock();
        rabin_karp_matcher(d[j], q[j]);
        end = clock();
        timer[j][i] = (double)(end - start) / (CLOCKS_PER_SEC);
    }
```

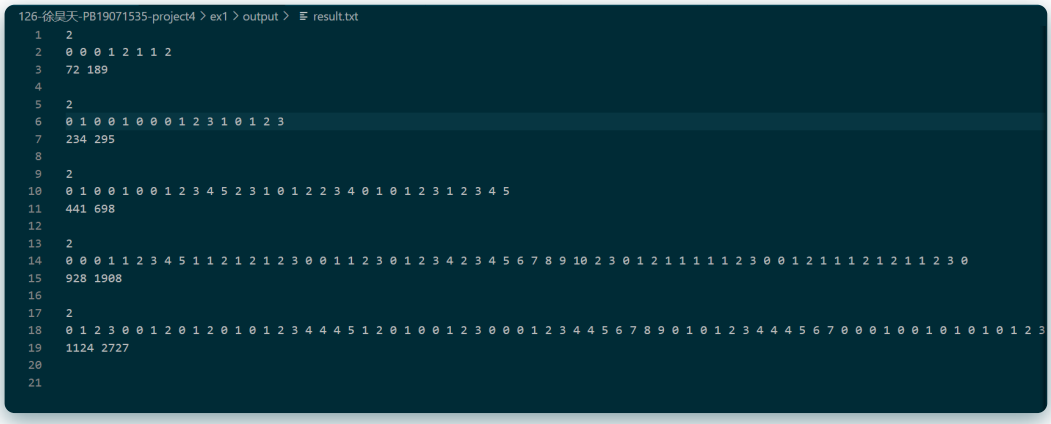
  得到时间单位为毫秒。

四.实验结果和分析

1.KMP算法

- result文件结果

  执行程序后 result.txt 文件打印的结果如下图所示：

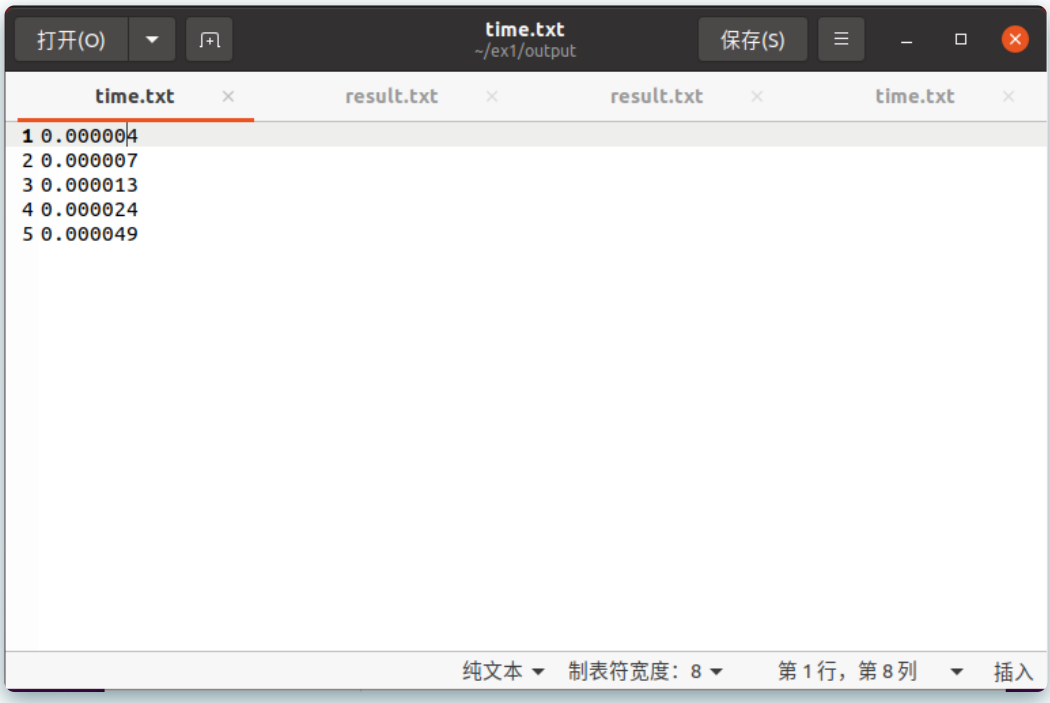


- 时间复杂度分析

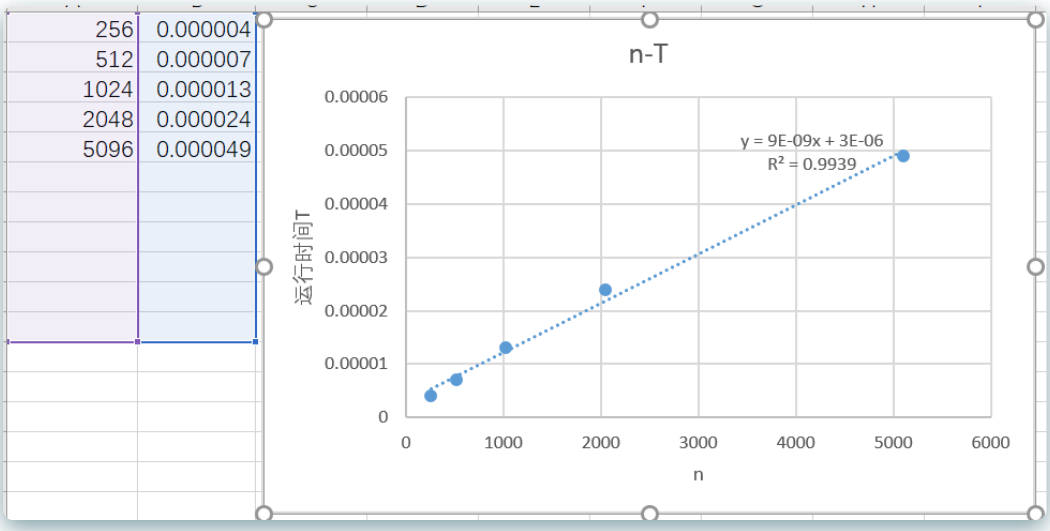
  将代码放入虚拟机中通过以下指令编译运行：

```
gcc main.c -o main
./main
```

  得到 time.txt 中时间如下图所示：



KMP算法计算前缀函数的复杂度是O(m)，实际匹配的复杂度是O(n).考虑到算法总复杂度是O(m+n)，n比m大得多，故以n为横坐标，直接画总运行时间的曲线。数据线性拟合如下图所示：

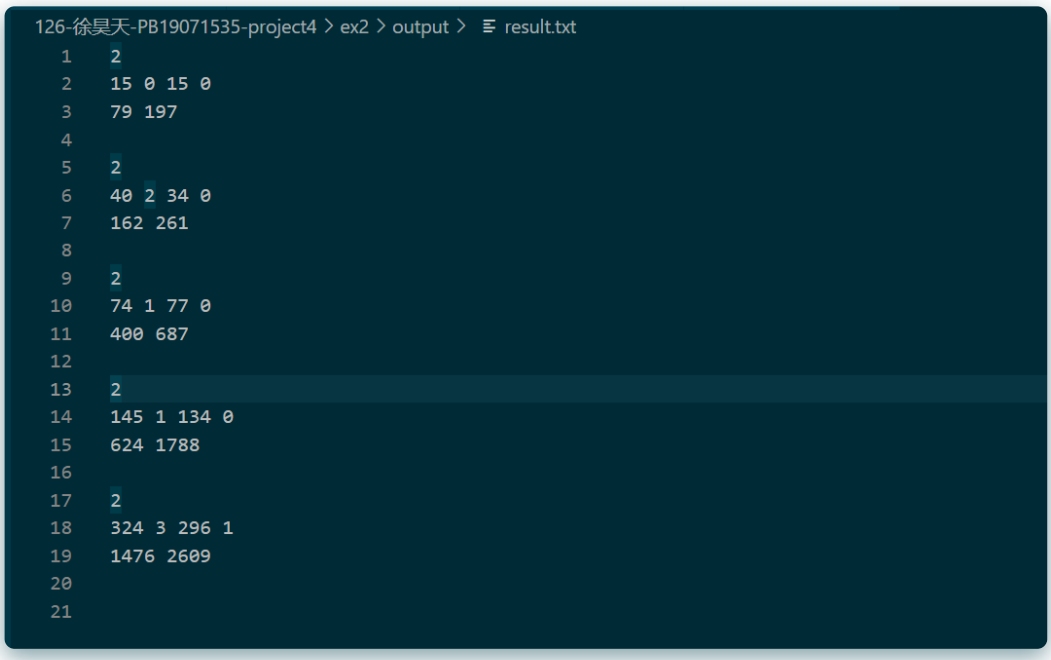


由上图可知，n和运行时间T呈线性关系，符合理论时间复杂度。

2.Rabin-Karp算法

- result文件结果

  执行程序后 `result.txt` 文件打印的结果如下图所示：

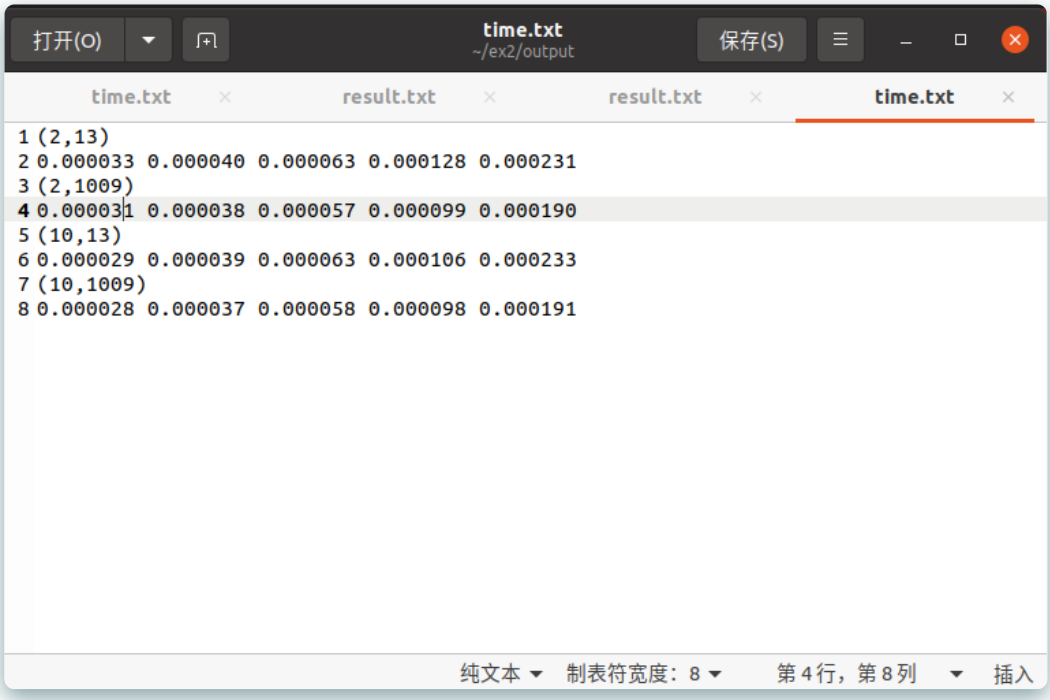


- 时间复杂度分析

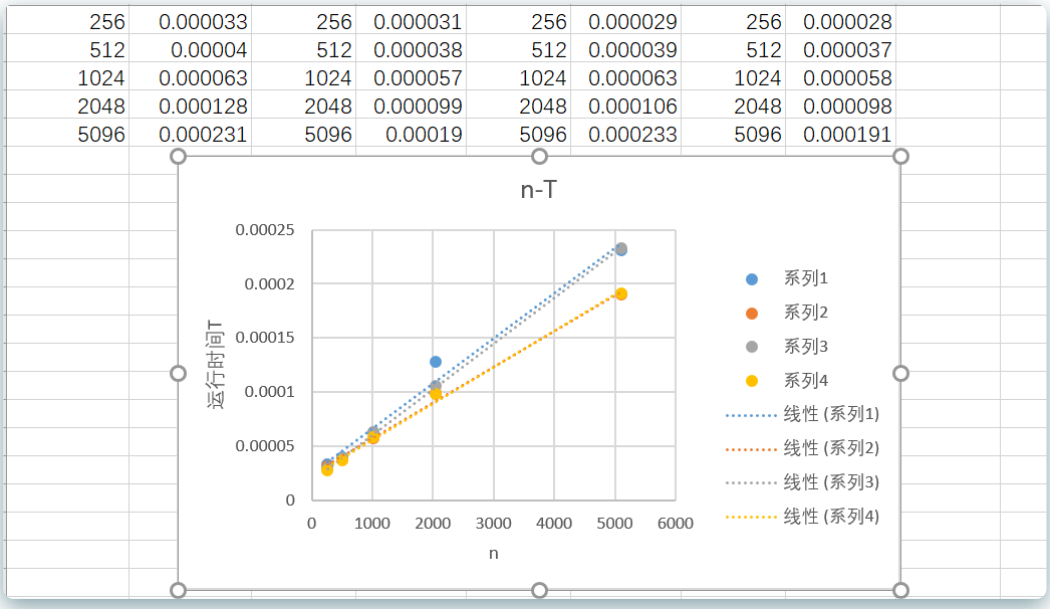
  将代码放入虚拟机中通过以下指令编译运行：

```
gcc main.c -o main
./main
```

得到 time.txt 中时间如下图所示：



RK算法最坏复杂度是O(mn)，考虑到n比m大得多，这里可以直接以n作横坐标，对应不同的(d,q)分别拟合不同曲线。数据线性拟合如下图所示：



由上图可知，在四种(d,q)情况下，n和运行时间T均呈线性关系，符合理论时间复杂度。

五.实验收获与感想

1. 通过实现经典算法实现串匹配算法，对以上算法有了更深刻的理解和更灵活的运用。
2. 通过处理较为复杂的字符串及其相关算法，提升了对C语言的运用和调试能力。