算法导论实验报告：串匹配算法

一、实验内容

字符串匹配问题，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（大、小写字母和数字，共 62 种不同字符）。(n, m)共取五组数据: (25, 4),(28, 8),(211,16),(214,32), (217,64).

算法： KMP 算法；Quick Search 算法； Karp-Rabin 算法

二、实验环境

编译环境：Microsoft Visual Studio 2015

机器内存：8G

时钟主频：2.5GHz

三、实验要求

a) 为实验建立一个文件夹，文件夹名称为：学号-project5，该文件夹包含实验报告和 3 个子文件夹：

input 文件夹： 输入数据

source 文件夹：源程序

output 文件夹：输出数据

b)input：

生成的所有文本串和模式串存入一个文件。存入格式：n1,m1,T1,P1; n2,m2,T2,P2; …… n5,m5,T5,P5 。 其中，每两组数据中间用换行符隔开。

c)output:

一组数据的运行结果存入一个文件，存入格式：n, m，首次成功匹配的起始位置（无成功匹配时输出-1），预处理所需时间，找到首次匹配所需运行时间.

example：第一组数据(25，4)，三个算法的结果存放路径：

学号-project5/output/output_1.txt

四、实验过程截图

```
#include <iostream>
#include<stdio.h>
#include<time.h>
#include<windows.h>
#include<fstream>
#include<sstream>
using namespace std;
char alphabet[64];
int* NEXT(char *P, int m) {
    int j = 0;
    int *next;
    next = new int[m];
    for (int i = 1; i <= m; i++) {
        next[i] = j;
        while (j > 0 && P[i] != P[j]) {
            j = next[j];
        }
        j = j + 1;
    }
    return next;
}

int KMP(char *T, char *P, int n,int m, int *next) {
    int j = 1;
    for (int i = 1; i <= n; i++) {
        while(j>0 && T[i]!=P[j]){
        j = next[j];
        }
        if (j == m) return i - m + 1;
        j++;
    }
    return -1;
}

int qs_bc[63], *p, *t;
```

五、实验过程

1、 生成随机字符串

设置一个初始数组 char alphabet[1~62]依次存放小写字母，大写字母和数字。

然后利用按时间生成的随机数在 alphabet 中取出字符。

```
for (int i = 0; i < 26; i++) {  //初始化alphabet  a~z
    alphabet[i+1] = i + 97;
}
for (int i = 26; i < 52; i++) {  //初始化alphabet A~Z
    alphabet[i+1] = i-26 + 65;
}
for (int i = 52; i < 62; i++) {  //初始化alphabet 0~9
    alphabet[i+1] = i -52+ 48;
}
//*生成随机字符串
        int k;
        for (int i = 1; i <= n; i++) {
             k = rand() % 62+1;
            T[i] = alphabet[k];
        }
        T[0] = ' '; T[n+1] = '\0';
        for (int i = 1; i <= m; i++) {
            k = rand() % 62+1;
            P[i] = alphabet[k];
        }
        P[0] = ' '; P[m+1] = '\0';
```

2、 KMP 算法

采用 next 数组的方案。定义: Next[j+1]= max{ k+1|P[1..k]是 P[1..j]的后缀，j>k≥0 }

kMP 算法分为两部分，第一部分为预处理过程，对模式 P 计算 next 数组，并返回该数组；第二部分为匹配过程。

```
int* NEXT(char *P, int m) {  //预处理函数
    int j = 0;
    int *next;
    next = new int[m];
    for (int i = 1; i <= m; i++) {
        next[i] = j;
        while (j > 0 && P[i] != P[j]) {
            j = next[j];
        }
        j = j + 1;
    }
    return next;
}

int KMP(char *T, char *P, int n,int m, int *next) {  //匹配函数
    int j = 1;
    for (int i = 1; i <= n; i++) {
        while(j>0 && T[i]!=P[j]){
        j = next[j];
        }
        if (j == m) return i - m + 1;
        j++;
    }
    return -1;
}
```

3、Quick-search 算法

Quick-search 算法分为两部分，第一部分为预处理过程，对模式 P 计算 qs-bc 数组，；第二部分为匹配过程。定义：Qs-Bc[c]=min{i : 1≤ i ≤ m and P[m+1-i]=c}

由于计算 qs_bc 需要利用的是字符在 alphabet 中的下标位置，所以要先将模式 P 和文本 T 映射到 int *p,*t。匹配过程中，P，T 方法字符的地方为[1~m(n)],但数组仍从 1 开始，所以调用的 memcmp 参数应为：memcmp(P+1，T + j ，m)

```
int qs_bc[63], *p, *t;
void prebc(char *P, int m,char *T,int n) {  //预处理过程

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= 62; j++) {
            if (P[i] == alphabet[j]) {
                p[i] = j; break;    //将char *P映射为int *p
            }
        }
    }
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= 62; j++) {
            if (T[i] == alphabet[j]) {
                t[i] = j; break; //将char *T映射为int *t
            }
        }
    }
    for (int i = 1; i <= 62; i++)  qs_bc[i] = m + 1;
    for (int i = 1; i <= m; i++)   qs_bc[p[i]] = m + 1-i;
}

int QS(char *P, int m, char *T, int n) {  //匹配过程
    int j = 1;
    while (j <= n - m + 1) {
        if (memcmp(P+1, T + j , m) == 0) return j;
        j = j + qs_bc[t[j + m]];
    }
    return -1;
}
```

4、Karp-Rabin 算法

Karp-Rabin 算法采用 d=2 的方案，分为两部分，第一部分为预处理过程，第二部分为匹配过程。

同 qs_bc 算法，由于计算 phash,tphash 需要利用的是字符在 alphabet 中的下标位置，所以再预处理过程中要先将模式 P 和文本 T 映射到 int *p,*t；再计算 phash 和第一个 thash.。匹配过程中，P，T 方法字符的地方为[1~m(n)],但数组仍从 1 开始，所以调用的 memcmp 参数应为：`memcmp(P+1, T + j , m)`

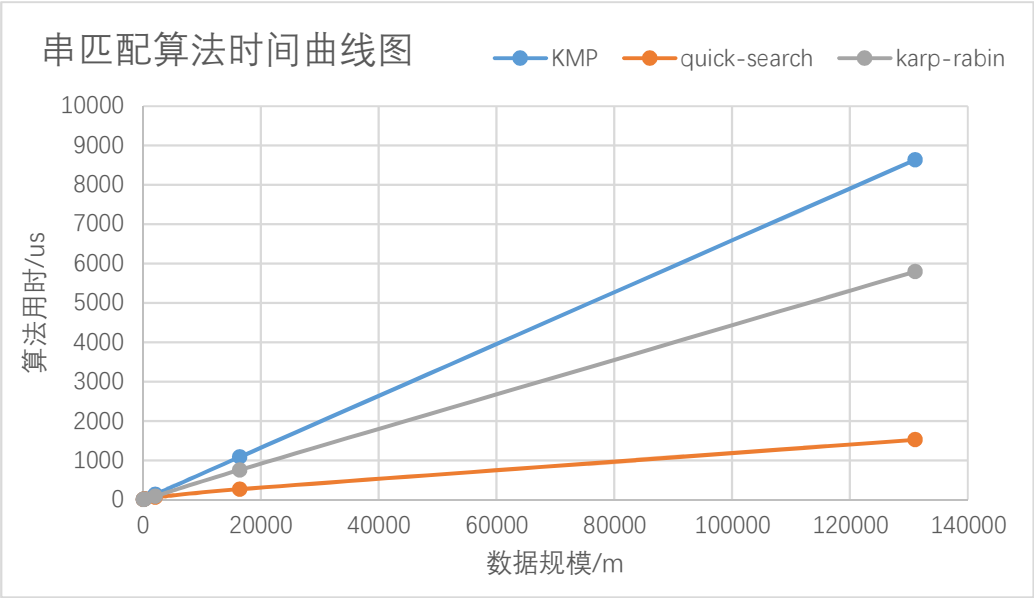
```
int d , phash, thash;
void prekr(char *P, int m, char *T, int n) { //预处理过程
    d = 1; phash = 0; thash = 0;
    /*映射过程同qs_bc*/
    for (int i = 1; i < m; i++) d = d << 1;
    for (int i = 1; i <= m; i++) {
        phash = (phash << 1) + p[i];
        thash = (thash << 1) + t[i];
    }
}

int KR(char *P, int m, char *T,int n) {  //匹配过程
    int j = 1;
    while (j <= n - m + 1) {
        if (phash == thash&&memcmp(P+1, T + j, m) == 0) return j;
        thash = ((thash - t[j] * d) << 1) + t[j + m];
        j++;
    }
    return -1;
}
```

六、实验结果与分析

各算法在不同规模下的预处理和匹配时间表。

数据规模	n	32	256	2048	16384	131072
	m	4	8	16	32	64
kmp	pretime	19	13	18	21	25
	matchtime	5	20	136	1084	8636
quick-search	pretime	42	274	2087	15796	123973
	matchtime	5	19	60	268	1523
karp-rabin	pretime	42	250	2023	15544	123645
	matchtime	3	12	87	758	5796

各算法的运行时间曲线图：



由上图可以看出，quick-search 算法的匹配时间最小，其次 karp-rabin 算法，kmp 算法的匹配时间最长。但 quick-search 和 karp-rabin 算法需要花费更长的预处理时间。

七、实验心得

检查各算法的实验结果，三个算法在不同规模下获得的首次成功匹配的起始位置都为-1，即没有找到成功匹配的位置。为检验算法的正确性,设置可以成功匹配的模式 P,和文本 T:

```
/*测试算法正确的例子
P = " Lczd";
    T = " 2izvB82Lczdgp4RQXBNam7hz8XVHNOvg";  */
```

得到的实验结果：

32,4

KMP:8,31,3

quick_search:8,39,5

Karp_Rabin:8,38,4

可知算法正确。

思考为什么随机生成的模式 P 和文中 T 中没有成功匹配的位置。原因是：

字符种类有 62 个，则模式 P 中可能出现的情形共有 62^m 个；

而文本 T 中可以有 n-m+1 个参与匹配的串；

则匹配成功的可能性为 p=$\frac{n-m+1}{62^m}$.

取 n=2^5，m=4，则 p=0.000002。结果非常之小。

且当 n,m 增大时，62^（2*m）的增长速率，远大于 8*n 的增长速率，所以 p 越来越小。

所以模式 P 在文本 T 中成功匹配的位置几乎没有。

为改进实验，可以减小字符的种类，增加文本的长度，缩短模式的长度等等使得匹配算法能成功匹配。