

数据结构实验报告

张恒硕 2212266 智科 2 班

实验目标：字符串匹配（暴力匹配和 KMP 算法）

实验原理：

暴力匹配算法：

也称朴素匹配算法，是一种简单而直观的字符串匹配方法。其基本思想是逐个字符地在主文本中与模式字符串进行比较，以查找模式字符串在主文本中出现的位置。其基本原理如下：

- 1、遍历主文本：从主文本的第一个字符开始，逐个字符地遍历主文本，直到遍历完所有字符。
- 2、比较字符：在每次遍历主文本时，将模式字符串的第一个字符与主文本当前位置的字符进行比较。
- 3、匹配检查：如果当前字符匹配，继续比较模式字符串的下一个字符与主文本中下一个字符，以此类推，直到整个模式字符串全部匹配。
- 4、发现匹配：如果模式字符串全部匹配，说明在当前主文本位置发现了一个匹配项。
- 5、继续搜索：无论是否找到匹配，都将主文本的指针向前移动一个字符，继续比较。
- 6、循环直到结束：重复上述过程，直到主文本中的所有位置都被检查过。

KMP（Knuth-Morris-Pratt）匹配算法：

一种高效的字符串匹配算法，通过利用部分匹配表来避免不必要的字符比较，从而提高匹配效率。其基本原理如下：

- 1、构建部分匹配表：根据模式字符串构建一个部分匹配表，其每个位置都存储了一个值，表示模式字符串在该位置发生不匹配时应回退到的位置，用于指导匹配过程中的跳跃操作，以便在不匹配时尽可能少地回退。部分匹配表的构建过程是一个预处理步骤，它从模式字符串的开头到结尾逐个字符地计算出每个位置对应的最长相同前缀后缀的长度。
- 2、匹配过程：从主文本开头开始，用两个指针分别指向主文本和模式字符串的当前字符位置。在每一步中，将模式字符串的指针与主文本的指针进行比较。如果当前字符匹配，那么继续比较下一个字符。如果当前字符不匹配，根据部分匹配表中的信息，将模式字符串的指针移动到适当的位置，以减少不必要的比较。重复上述过程，直到找到匹配或遍历完主文本。如果发现不匹配，不会回退到主文本的当前位置，而是根据部分匹配表中的信息来选择一个跳跃的位置。

代码：

```
#include <iostream>
using namespace std;
class String {
private:
    char* data;
    int size;
public:
    String(string str = "") { //构造函数
        const char* a = str.data();
        data = new char[strlen(a) + 1];
```

```

        strcpy(data, a);
        size = strlen(a);
    }
    ~String() { //析构函数
        delete[] data;
    }
    String operator+(const String& other) const { //重载运算符+
        String result;
        result.size = size + other.size;
        result.data = new char[result.size + 1];
        strcpy(result.data, data);
        strcat(result.data, other.data);
        return result;
    }
    bool operator==(const String& other) const { //重载运算符==
        if (size != other.size) {
            return false;
        }
        return strcmp(data, other.data) == 0;
    }
    bool operator!=(const String& other) const { //重载运算符!=
        return !(*this == other);
    }
    friend ostream& operator<<(ostream& os, const String& str) { //重载运算符<<
        os << str.data;
        return os;
    }
    int size_() const { //获取字符串长度
        return size;
    }
    const char* c_str() const { //获取字符串内容
        return data;
    }
    void Strong_match(const String& pattern) { //暴力匹配
        int num = 0; //用于计数
        int a = 0; //用于记录有无
        for (int i = 0; i <= size_() - pattern.size_(); ++i) {
            int j;
            for (j = 0; j < pattern.size_(); ++j) {
                if (data[i + j] != pattern.data[j]) {
                    break; //如果发现不匹配，退出内部循环
                }
            }
            if (j == pattern.size_()) { //找到匹配，输出匹配的位置

```

```

        num++;
        a = 1;
        cout << "第" << num << "个出现的对象在第" << i + 1 << "位开始向后" <<
pattern.size_() << "位" << endl;
    }
}

if (a == 0) {
    cout << "未找到目标字符串" << endl;
}

}

void LPS(String& pattern) { //部分匹配表
    int length = 0; //最长前缀后缀匹配的长度
    int i = 1;
    while (i < pattern.size_()) {
        if (pattern.data[i] == pattern.data[length]) {
            length++;
            data[i] = length;
            i++;
        }
        else {
            if (length != 0) {
                length = data[length - 1];
            }
            else {
                data[i] = 0;
                i++;
            }
        }
    }
}

void KMP_match(String& pattern) { //KMP匹配
    String str;
    for (int i = 0; i < pattern.size_(); i++) {
        str.data[i] = 0;
    }
    str.LPS(pattern);
    int i = 0; //用于遍历文本
    int j = 0; //用于遍历模式
    int num = 0; //用于计数
    int a = 0; //用于记录有无
    while (i < size_()) {
        if (pattern.data[j] == data[i]) {
            i++;
            j++;

```

```

    }
    if (j == pattern.size_()) {
        num++;
        a = 1;
        cout << "第" << num << "个出现的对象在第" << i - j + 1 << "位开始向后" <<
pattern.size_() << "位" << endl;
        j = str.data[j - 1]; //继续搜索下一个匹配
    }
    else if (i < size_() && pattern.data[j] != data[i]) {
        if (j != 0) {
            j = str.data[j - 1]; //部分匹配，回退到最长前缀后缀匹配的位置
        }
        else {
            i++;
        }
    }
}
if (a == 0) {
    cout << "未找到目标字符串" << endl;
}
};

int main() {
    int a;
    string str1, str2;
    cout << "输入主文本: " << endl;
    cin >> str1;
    cout << "输入模式字符串: " << endl;
    cin >> str2;
    String text(str1), pattern(str2);
    cout << "输入匹配方法 (1为暴力匹配, 2为KMP匹配): " << endl;
    cin >> a;
    if (a == 1) {
        cout << "通过暴力匹配得: " << endl;
        text.Strong_match(pattern);
    }
    if (a == 2) {
        cout << "通过KMP匹配得: " << endl;
        text.KMP_match(pattern);
    }
    return 0;
}

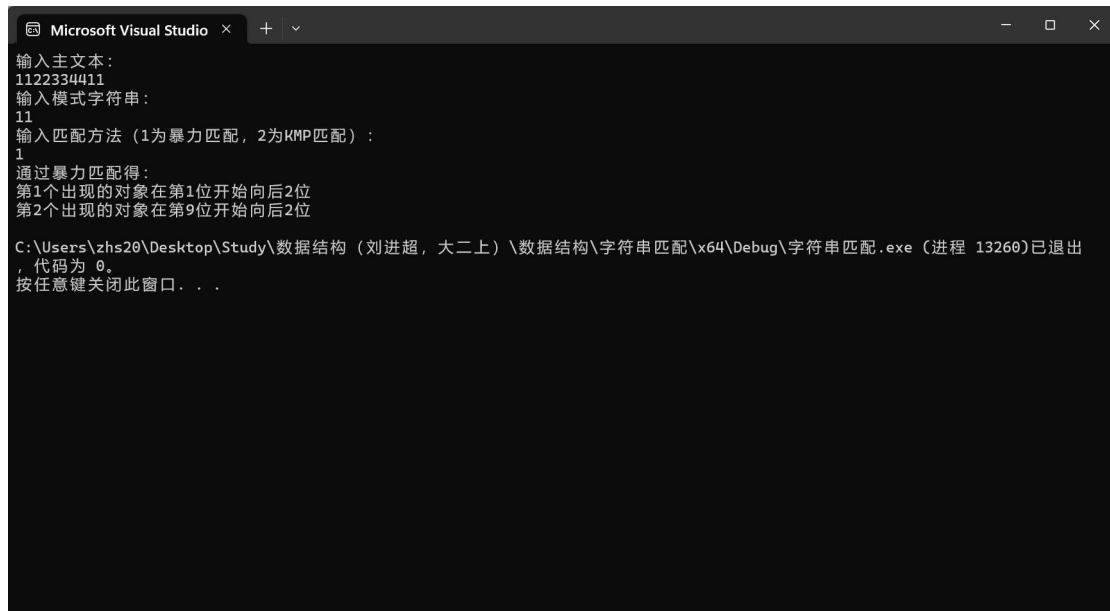
```

输入样例:

- 1、1122334411 11 1（暴力匹配）——测验暴力匹配及其多个输出
- 2、1122334411 11 1（KMP 匹配）——测验 KMP 匹配及其多个输出
- 3、sample 倒数第二个——测试未匹配
- 4、sample (wpwOhtp8H0QX4m*X wpwOhtp8H0QX4m*)
其他 sample 的测试略

运行结果：

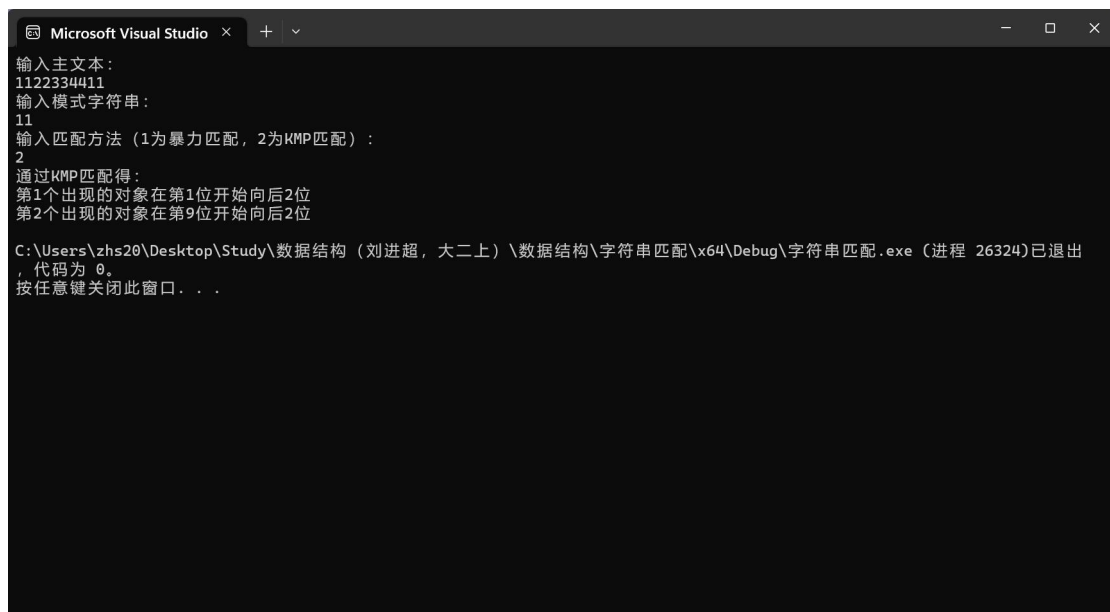
1、



```
Microsoft Visual Studio x + v
输入主文本：
1122334411
输入模式字符串：
11
输入匹配方法（1为暴力匹配，2为KMP匹配）：
1
通过暴力匹配得：
第1个出现的对象在第1位开始向后2位
第2个出现的对象在第9位开始向后2位

C:\Users\zhs20\Desktop\Study\数据结构（刘进超，大二上）\数据结构\字符串匹配\x64\Debug\字符串匹配.exe（进程 13260）已退出，
代码为 0。
按任意键关闭此窗口...
```

2、



```
Microsoft Visual Studio x + v
输入主文本：
1122334411
输入模式字符串：
11
输入匹配方法（1为暴力匹配，2为KMP匹配）：
2
通过KMP匹配得：
第1个出现的对象在第1位开始向后2位
第2个出现的对象在第9位开始向后2位

C:\Users\zhs20\Desktop\Study\数据结构（刘进超，大二上）\数据结构\字符串匹配\x64\Debug\字符串匹配.exe（进程 26324）已退出，
代码为 0。
按任意键关闭此窗口...
```

3、

分析：

暴力匹配算法：

该算法简单易懂，容易实现，但时间复杂度较高。在最坏情况下，当主文本和模式字符串都很长且没有匹配时，时间复杂度为 $O((n-m+1)*m)$ ，其中 n 是主文本的长度， m 是模式字符串的长度。因此，对于较大的文本和模式，它的性能可能不够高效。

KMP 匹配算法：

通过构建部分匹配表并使用它来指导匹配过程，该算法能够以线性时间复杂度 $O(n+m)$ 的速度找到模式字符串在主文本中的所有匹配，其中 n 是主文本的长度， m 是模式字符串的长度。

相比朴素的暴力匹配算法，KMP 算法在大型文本上的性能更加出色。这使得它成为实际应用中常用的字符串匹配算法之一。