第二次课后作业

作业要求

- **根据附件文件“test.txt”内容，该文件大小为 100M，全部为小写字母构成，没有符号等其它内容，回答以下问题：**

1. **文件中是否出现“zhongguo”，出现在哪，出现了多少次；文件中“deff”，出现了多少次，分别在哪些位置。**
2. **文件中出现最长的单个字母重复序列是哪个字母，长度多少，位置在哪？如“aaaaaaa”，“bbbbbbbbbbbbb”。**
3. **若任意取 3 个字母构成一个字符串“xxx”，哪个字符串出现次数最多，哪个字符串出现次数最少，分别多少次。**

- **编程实现 KMP 算法；**

可以发现，整个作业有多项功能需要分开实现，所以编程中最好采用多个函数封装调用的手段，这样可以避免多个孤立功能同时调试造成的问题。因此，我制定了一套编程方案如下：（只在所有代码之后展示总效果，不在每一步骤中展示分效果）

1.编写主函数，综合交互功能，同时编写所有的声明。

2.编写封装函数实现读取源目录中的文本文档，并且返回读入结果的字符串地址

3.编写封装函数实现第一个任务，即读入模式字符串和全局字符串地址，并识别模式字符串的位置并输出。

4.编写封装函数实现第二个任务，即读入模式字符串和全局字符串地址，并识别出现最长单个字母重复序列的长度和位置并输出。

5.编写封装函数实现第三个任务，即读入模式字符串和全局字符串地址，并识别三个字母字符串组合搭配出现次数最多和最少的情况以及分别对应次数并输出。

源代码说明:

代码和原来完整的 exe 文件将被存放在 homework0201 文件夹附件之中。

1. 编写主函数，综合交互功能，同时编写所有的声明。

编程实现：

（1）库文件：为了实现更多的功能，原有的 iostream 和 string 库文件已经不能满足我们的需求了，所以在头文件引用中，额外使用了 fstream、cstring、vector 库文件，这些库文件可以帮助我们更好地实现拓展功能，比如读取文件、容器存储等等。

（2）全局变量和结构体定义：有一些数据在后期会随时修改，而且要求不受函数限制，主要是存储最大字母和最小字母组合的变量，在这里分别定义了不同的全局变量和结构体来记录这些信息。

（3）函数声明：每一个任务都制定了自己的实现函数和过程函数，实现函数供主函数直接调用，过程函数供实现函数直接调用，另外，读取文件另外制定了自己的实现函数。

（4）主函数：按照不同函数的特点和交互需求进行编写

代码如下：

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <cstring>
#include <vector>

int total_max = -1;//定义一个全局变量，记录第三个任务中所有最大字母组合的数量
int total_min = -1;//定义一个全局变量，记录第三个任务中所有最小字母组合的数量

struct maxdata {
    int length_s_max;
    char function3_result_1_max ;
    char function3_result_2_max ;
    char function3_result_3_max ;
}maxdatas[999];//定义一个结构体和对应的数组存储最大字母组合的信息

struct mindata {
    int length_s_min;
    char function3_result_1_min;
    char function3_result_2_min;
    char function3_result_3_min;
}mindatas[999];//定义一个结构体和对应的数组存储最小字母组合的信息

// 函数声明
const char* read_txt();//读取文本文档，返回读取字符串指针。
void function1_searchforspecialtext(const char* pattern, const char* text);
//任务一实现函数：实现匹配字符串的完整功能，其中有调用到computeLPSArray()函数
void computeLPSArray(const char* pattern, size_t M, unsigned long long int* lps);
//任务一过程函数：计算部分匹配表（LPS数组），供实现函数所调用
void function2_searchforRS(const std::string& text);
//任务二实现函数：实现寻找最长重复字母序列的功能
void function3_searchforTL(const char* text);
//任务三实现函数：实现求出三字母搭配的最多和最少情形及它们出现数量
void function1_searchforspecialtext_s(const char* pattern, const char* text);
//任务三过程函数：是功能一实现函数的改装，实现匹配字符串的完整功能，供实现函数所调用
void computeLPSArray_s(const char* pattern, size_t M, unsigned long long int* lps);
//任务三过程函数：是功能一过程函数的改装，实现计算部分匹配表（LPS数组），供实现函数所调用

int main() {
    const char* textAddress = read_txt();// 调用函数读取文本文件
    if (textAddress != nullptr) {// 检查指针是否为空
        //输出读取成功的检查信息
        std::cout << "读取文本文档内容成功！" << std::endl;
```

```
    }
    else {
        //输出读取失败的检查信息
        std::cerr << "读取文本文档内容失败！" << std::endl;
    }
    const char* text = textAddress;//textAddress和text一样，成为两个独立存储文本内容的
变量
    std::string thestring;
    thestring = textAddress;//定义了一个string变量，并使其也存储文本内容
    std::cout<< "读取到了" << thestring.length() << "个字符" << std::endl;
    // 汇报读取内容的简要信息，即字数数量，让读者对于运行时间心里有数。
    std::cout<<"匹配字符串为zhongguo的时候，" << std::endl;
    function1_searchforspecialtext("zhongguo", text);
    //匹配字符串为zhongguo的时候，运行任务一的实现函数
    std::cout << "匹配字符串为deff的时候," << std::endl;
    function1_searchforspecialtext("deff",text);
    //匹配字符串为deff的时候，运行任务一的实现函数
    function2_searchforRS(thestring);
    //运行任务二的实现函数
    function3_searchforTL(textAddress);
    //运行任务三的实现函数
    delete[] textAddress;
    return 0;
    }
```

2.编写封装函数实现读取源目录中的文本文档，并且返回读入结果的字符串地址

编程思想：步骤可以制定为打开文件-从文件读取内容到字符串，如果出错返回空指针-关闭文件-检查文件内容是否为空，如果出错返回空指针-将字符串拷贝到动态分配的内存中，返回字符串的地址

重难点解释：

（1）发现新版本对于 cstring 库文件来说更兼容一些，然后文件的读取又不能脱离 fstream 的帮助，所以需要研究清楚这两个库函数中的具体函数的用法。

（2）由于读取文件过程中有很多步骤都有可能出现意外的错误情况，所以需要在每一步骤后都加入检查的代码，一旦有错误直接返回空指针而不是系统报错，这样方便调试。

（3）由于文本文档中的字数无法确定，尤其是在本次文本文档中出现大量字符串的情况，所以需要使用动态分配的方法建立内存，这与之前我们建立有限数组的方法有所不同，所以具有一定挑战性。

源代码说明：由于 2 只是编程的一个封装部分，所以总的代码放到 homework0201 的附件中了，只在 word 中展示单独的 2 的代码及其注释：

```
const char* read_txt() {
    // 文件路径，你可以根据需要修改
```

```
    std::string filePath = "test.txt";
    // 打开文件
    std::ifstream fileStream(filePath);
    if (!fileStream.is_open()) {
        std::cerr << "打开文件出错： " << filePath << std::endl;
        return nullptr;  // 返回空指针表示出错
    }
    // 从文件读取内容到字符串
    std::string text((std::istreambuf_iterator<char>(fileStream)),
        std::istreambuf_iterator<char>());
    // 关闭文件
    fileStream.close();
    // 检查文件内容是否为空
    if (text.empty()) {
        std::cerr << "错误：文件为空" << std::endl;
        return nullptr;  // 返回空指针表示出错
    }
    // 将字符串拷贝到动态分配的内存中
    char* textCopy = new char[text.size() + 1];
    // 使用 strcpy_s 替代 strcpy
    strcpy_s(textCopy, text.size() + 1, text.c_str());
    // 返回字符串的地址
    return textCopy;
}
```

3.编写封装函数实现第一个任务，即读入模式字符串和全局字符串地址，并识别模式字符串的位置并输出。

编程思想：利用了 KMP 算法，KMP 算法的关键步骤主要是构建一个部分匹配表，然后执行匹配，构建匹配表主要就是对于模式串，从左到右逐字符扫描，对每个字符记录当前字符之前的字串的最长相同前缀和后缀的长度，这个信息被记录在部分匹配表中，形成一个部分匹配值的数组，表的索引表示当前字符的位置，对应的值表示最长相同前缀和后缀的长度。执行匹配主要是在匹配过程中，将模式串与文本串逐字符比较，当发现不匹配时，利用部分匹配表中的信息进行跳跃，将模式串向右滑动，跳过已经匹配的部分。具体地，根据部分匹配表中的值，将模式串向右滑动不同的距离，以继续匹配。

重难点解释：与网上的 KMP 算法不同，大部分网上 KMP 算法都只能实现一次匹配识别，而不能实现在一次匹配之后记录并继续下一次匹配，一直到文本结束为止，所以这里需要充分理解 KMP 算法的思想才可以进行改进。

源代码说明：由于 3 只是编程的一个封装部分，所以总的代码放到 homework0201 的附件中了，只在 word 中展示单独的 3 的代码及其注释：

```
// 计算部分匹配表（LPS数组）
void computeLPSArray(const char* pattern, size_t M, unsigned long long int* lps) {
    unsigned long long int len = 0;// 初始化匹配长度为0
    lps[0] = 0;// 首个元素的LPS值为0
```

```
    unsigned long long int i = 1;
    while (i < M) {
        if (pattern[i] == pattern[len]) {// 当前字符匹配
            len++;// 增加匹配长度
            lps[i] = len;// 更新当前位置的LPS值
            i++;// 移动到下一个字符
        }
        else {
            if (len != 0) {
                len = lps[len - 1];// 回溯到前一个字符的LPS值
            }
            else {
                lps[i] = 0;// 当前字符无匹配，设置LPS值为0
                i++;// 移动到下一个字符
            }
        }
    }
}

// 使用KMP算法在文本中查找匹配的模式串
void function1_searchforspecialtext(const char* pattern, const char* text) {
    std::cout << "开始进行第一个任务程序，即找出模式串在文本中的匹配次数和对应的位置，
结果如下：" << std::endl;
    // 获取模式串和文本串的长度
    size_t M = strlen(pattern);
    size_t N = strlen(text);
    // 存储匹配位置的向量
    std::vector<unsigned long long int> positions;
    // 记录匹配的数量
    unsigned long long int count = 0;
    // 分配存储部分匹配表（LPS数组）的内存
    unsigned long long int* lps = new unsigned long long int[M];
    // 计算部分匹配表
    computeLPSArray(pattern, M, lps);
    // 初始化文本和模式串的索引
    unsigned long long int i = 0; // 用于遍历text[]
    unsigned long long int j = 0; // 用于遍历pattern[]
    // KMP算法主循环
    while (i < N) {
        // 如果字符匹配，增加索引
        if (pattern[j] == text[i]) {
            j++;
            i++;
```

```
        }
        // 如果整个模式串匹配，记录位置并调整索引
        if (j == M) {
            // 模式串在文本中的位置为 i - j
            positions.push_back(i - j);
            j = lps[j - 1];
            count++;
        }
        else if (i < N && pattern[j] != text[i]) {
            // 如果字符不匹配，根据部分匹配表调整索引
            if (j != 0) {
                j = lps[j - 1];
            }
            else {
                i++;
            }
        }
    }
    // 释放部分匹配表的内存
    delete[] lps;
    // 输出结果
    if(count!=0)
    {
        std::cout << "模式串在文本中出现 " << count << " 次，位置分别为：\n";
        for (unsigned long long int pos : positions) {
            std::cout << pos + 1 << " ";
        }//输出模式串结果
        std::cout << std::endl;
    }
    else
    {
        std::cout << "模式串在文本中未出现" << std::endl;
    }//在未找到模式串的情况下也同样输出模式串结果
}
```

4. 编写封装函数实现第二个任务，即读入模式字符串和全局字符串地址，并识别出现最长单个字母重复序列的长度和位置并输出。

编程思想：遍历字符串，遍历所有的文本字符串，使用循环向前查找相同字符，计算重复长度，通过比较当前重复长度与记录的最大长度，更新相应的变量，最后输出最长重复字母，本质是通过迭代和比较，动态地更新记录最长重复信息的变量，简单而高效

重难点解释：由于这个功能比较简单，认为不需要用 KMP 算法，所以直接使用了简单的迭代和比较算法就得到了结果

源代码说明：由于 4 只是编程的一个封装部分，所以总的代码放到 homework0201 的附件中了，只在 word 中展示单独的 4 的代码及其注释：

```
void function2_searchforRS(const std::string& text) {
    std::cout << "开始进行第二个任务，即找出最长重复字母、重复长度及长度，结果如下："
<< std::endl;
    int maxLen = 0;// 记录最长重复长度的变量
    int maxPos = -1;// 记录最长重复字母的起始位置的变量
    char maxChar = '\0';// 记录最长重复字母的变量
    // 遍历文本字符串
    for (size_t i = 1; i < text.length(); i++) {
        char currentChar = text[i];// 当前字符

        size_t j = i - 1;// 寻找与当前字符相同的最远位置
        while (j >= 0 && text[j] == currentChar) {
            j--;
        }

        int len = i - j - 1;// 计算重复长度
        if (len > maxLen) {// 如果当前重复长度超过记录的最大长度
            maxLen = len;// 更新最大重复长度
            maxPos = j + 1;// 更新最长重复字母的起始位置
            maxChar = currentChar; // 更新最长重复字母
        }
    }
    std::cout << "最长的重复字母为: " << maxChar << ", 重复长度: " << maxLen + 1 << ",
位置: " << maxPos + 1 << std::endl;
}
```

5. 编写封装函数实现第三个任务，即读入模式字符串和全局字符串地址，并识别三个字母字符串组合搭配出现次数最多和最少的情况以及分别对应次数并输出。

编程思想：遍历了所有的三个字母的组合，然后每到一个组合就把这个组合进行 KMP 匹配并算出其出现次数，利用“打擂台”的比较算法，如果有比最大值更大或者比最小值更小的情况直接替代，最后输出最大值和最小值即可，需要改造原来的 KMP 算法进行调用。

重难点解释：这里的“打擂台”不同于一般的情况，由于有最大值和最小值两种情况，而且其三个字符信息都需要记录下来，所以改造起来相当复杂，其中要利用大量的条件判断排除特殊情况，具有很大的挑战性。

源代码说明：由于 5 只是编程的一个封装部分，所以总的代码放到 homework0201 的附件中了，只在 word 中展示单独的 5 的代码及其注释：

```
void function3_searchforTL(const char* text) {
    std::cout<<"开始进行最后一个任务，将会已经计算出的字母组合及其数量，这虽然不是结
果，但可以看到运行进度" << std::endl;
    std::cout<<"在打印完所有的内容后，会输出结果，即最多和最少的字母组合及其对应的数量"
<< std::endl;
    std::cout << "现在，进程将开始进行，如果文本量较大，那么程序运行时间将会非常长，请
```

```
耐心等待" << std::endl;
    char pattern_s[]="aaa";
    for (char i = 'a'; i <= 'z'; ++i) {
        for (char j = 'a'; j <= 'z'; ++j) {
            for (char k = 'a'; k <= 'z'; ++k) {//遍历所有的三个字母组合
                pattern_s[0] = i;
                pattern_s[1] = j;
                pattern_s[2] = k;//把相应的字母进行赋值，形成完整的匹配字符串
                std::cout << pattern_s;//输出这个字符串，显示进度
                function1_searchforspecialtext_s(pattern_s,text);//调用函数进行KMP匹配
            }
        }
    }
    std::cout << "\n统计数量最大的字母组合是";
    for (int w=0;w<=total_max;w++)
    {
        std::cout << maxdatas[w].function3_result_1_max;
        std::cout << maxdatas[w].function3_result_2_max;
        std::cout << maxdatas[w].function3_result_3_max;//打出所有同时满足最大值的情况
        if (w!=total_max)
        {
            std::cout << "\t";
        }
        else
        {
            std::cout << "\n";
        }//这里需要分情况输出保证格式整洁
    }
    std::cout << "出现的次数是" <<maxdatas[0].length_s_max << std::endl;
    std::cout << "统计数量最小的字母组合是";
    for (int w = 0; w <= total_min; w++)
    {
        std::cout << mindatas[w].function3_result_1_min;
        std::cout << mindatas[w].function3_result_2_min;
        std::cout << mindatas[w].function3_result_3_min;//打出所有同时满足最小值的情况
        if (w != total_min)
        {
            std::cout << "\t";
        }
        else
        {
            std::cout << "\n";
        }//这里需要分情况输出保证格式整洁
    }
```

```
    std::cout << "出现的次数是" << mindatas[0].length_s_min << std::endl;

}

void function1_searchforspecialtext_s(const char* pattern, const char* text) {
    size_t M = strlen(pattern);
    size_t N = strlen(text);

    // 存储匹配位置的向量
    std::vector<unsigned long long int> positions;
    // 记录匹配的数量
    unsigned long long int count = 0;

    // 分配存储部分匹配表（LPS数组）的内存
    unsigned long long int* lps = new unsigned long long int[M];
    // 计算部分匹配表
    computeLPSArray_s(pattern, M, lps);

    // 初始化文本和模式串的索引
    unsigned long long int i = 0; // 用于遍历text[]
    unsigned long long int j = 0; // 用于遍历pattern[]

    // KMP算法主循环
    while (i < N) {
        // 如果字符匹配，增加索引
        if (pattern[j] == text[i]) {
            j++;
            i++;
        }

        // 如果整个模式串匹配，记录位置并调整索引
        if (j == M) {
            // 模式串在文本中的位置为 i - j
            positions.push_back(i - j);
            j = lps[j - 1];
            count++;
        }
        else if (i < N && pattern[j] != text[i]) {
            // 如果字符不匹配，根据部分匹配表调整索引
            if (j != 0) {
                j = lps[j - 1];
            }
            else {
                i++;
```

```
            }
        }
    }

    // 释放部分匹配表的内存
    delete[] lps;

    // 输出结果
    if (count != 0)
    {
        std::cout << count << "次";
        std::cout << "\t";//输出对应字符串的次数
        if ((maxdatas[total_max].length_s_max < count)&&(total_max!=-1))
        {
            total_max = 0;
            maxdatas[total_max].length_s_max = count;
            maxdatas[total_max].function3_result_1_max = pattern[0];
            maxdatas[total_max].function3_result_2_max = pattern[1];
            maxdatas[total_max].function3_result_3_max = pattern[2];//“打擂台”情况讨论，出现胜利者
        }
        else if (total_max == -1)
        {
            maxdatas[0].length_s_max = count;
            maxdatas[0].function3_result_1_max = pattern[0];
            maxdatas[0].function3_result_2_max = pattern[1];
            maxdatas[0].function3_result_3_max = pattern[2];//“打擂台”情况讨论，第一次特别处理
            total_max=0;
        }
        else if ((maxdatas[total_max].length_s_max == count) && (total_max != -1))
        {
            total_max++;
            maxdatas[total_max].length_s_max = count;
            maxdatas[total_max].function3_result_1_max = pattern[0];
            maxdatas[total_max].function3_result_2_max = pattern[1];
            maxdatas[total_max].function3_result_3_max = pattern[2];//“打擂台”情况讨论，出现平局
        }
        if ((mindatas[total_min].length_s_min > count) && (total_min != -1))
        {
            total_min = 0;
            mindatas[total_min].length_s_min = count;
            mindatas[total_min].function3_result_1_min = pattern[0];
```

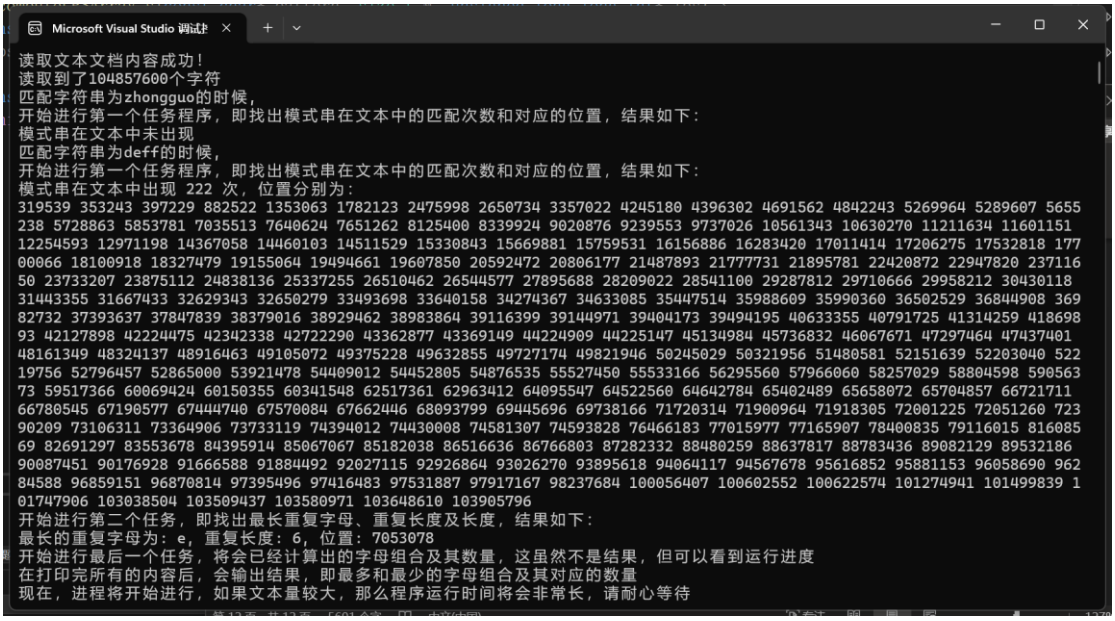
```
            mindatas[total_min].function3_result_2_min = pattern[1];
            mindatas[total_min].function3_result_3_min = pattern[2];//“打擂台”情况讨论，出现胜利者
        }
        else if (total_min == -1)
        {
            mindatas[0].length_s_min = count;
            mindatas[0].function3_result_1_min = pattern[0];
            mindatas[0].function3_result_2_min = pattern[1];
            mindatas[0].function3_result_3_min = pattern[2];//“打擂台”情况讨论，第一次特别处理
            total_min=0;
        }
        else if ((mindatas[total_min].length_s_min == count) && (total_min != -1))
        {
            total_min++;
            mindatas[total_min].length_s_min = count;
            mindatas[total_min].function3_result_1_min = pattern[0];
            mindatas[total_min].function3_result_2_min = pattern[1];
            mindatas[total_min].function3_result_3_min = pattern[2];//“打擂台”情况讨论，出现平局
        }
    }
    else
    {
        std::cout << "字母组合在文本中未出现" << std::endl;//根本没有出现出现的情况
    }
}


void computeLPSArray_s(const char* pattern, size_t M, unsigned long long int* lps) {
    unsigned long long int len = 0;//初始化匹配长度为0
    lps[0] = 0;// 首个元素的LPS值为0

    unsigned long long int i = 1;
    while (i < M) {
        if (pattern[i] == pattern[len]) {// 当前字符匹配
            len++;//增加匹配长度
            lps[i] = len;// 更新当前位置的LPS值
            i++;// 移动到下一个字符
        }
        else {
            if (len != 0) {
                len = lps[len - 1];// 回溯到前一个字符的LPS值
```

```
        }
        else {
            lps[i] = 0;// 当前字符无匹配，设置LPS值为0
            i++;// 移动到下一个字符
        }
    }
}
}
```

运行结果：



可以发现，实现了第一个任务和第二个任务，另外，也可以看到第三个任务的启动信息

可以发现，第三个任务也正在运行中，而且不断输出进度信息，让用户看到运行的位置

可以发现，在所有进度之后，程序成功输出了任务三的运行结果并且成功结束。

由于这份运行结果非常长，所以我另外打包了一份文本文档，名字是“运行结果.txt”

实现了所有的任务效果，创新点：

1.任务三中每次读取都显示进度，可以让用户更好地看到进度进行到哪里了

2.在刚开始读取文本文档的时候增加了报告读取字数的功能，可以让用户看到读取字数结果

3.增加了大量的文字以保证可读性

4.尽可能使用 KMP 算法以提高效率，对于不用使用 KMP 算法的部分则避免使用以免影响效率

5.对各个功能进行了函数的封装，保证了整体功能的可迁移性和可维护性

6.使用了多样化的库文件，提高编程效率