

字符串匹配实验报告

1.问题说明

用 $O(m)$ 时间复杂度找出一个长度为 m 的短字符串在一个长度为 n 的长字符串中的精确匹配 ($n \gg m$)，限制长字符串仅由 A、C、G、T 这四种字符组成。

输入：长短字符串

输出：短字符串在长字符串里的精确匹配

2.功能说明

在匹配前，输入长串文件，对长串进行分块编码的预处理。

在匹配时，输入短串文件，利用已经处理好的长串的分块编码和索引文件，找出短串在长串里的所有精确匹配。

3.设计原理

在匹配前先对长串进行编码和索引计算的预处理，该部分用到BWT编码的思想，并且利用后缀数组和倍增算法对编码过程进行优化。由于长串过长，在编码和存储时对其进行分治处理，并且在索引存储时进行优化，对索引进行间隔存储。在匹配时，用预处理得到的长串编码和索引进行匹配，用到的是FM索引-序列匹配。

BWT编码和索引计算

BWT的全称是The Burrows-Wheeler Transform，也就是所谓的轮转排序。在字符串匹配中首先用BWT对长字符串 S 进行预处理，得到编码结果 $BWT(S)$ 和相应索引，用于后续FM索引技术对字符串进行匹配。

BWT编码对应具体步骤如下：

- 1、设需要编码的长字符串为 S ，在 S 末尾加上标记字符\$（该字符的字典序值小于长串中所有字符），得到新字符串 S' ，记 S' 的长度为 len 。
- 2、对 S' 进行循环移位，每次将 S' 中的最后一个字符转移到整个字符串的最前面，一共转移 len 次，每次转移都得到一个长度为 len 的新字符串。
- 3、把上一步中得到的 len 个新字符串按照字典序排序，得到一个 $len \times len$ 的字符矩阵 M 。
- 4、 M 中每个字符串的第一个字符构成 F 列，最后一个字符构成 L 列。 L 列即为 $BWT(S)$ 。

输入 S	加标记得到 S'	循环转移	排序后得到 M	F 列	L 列
ACCGATG	ACCGATG\$	\$ACCGATG	\$ACCGATG	\$	G
		G\$ACCGAT	ACCGATG\$	A	\$
		TG\$ACCGA	ATG\$ACCG	A	G
		ATG\$ACCG	CCGATG\$A	C	A
		GATG\$ACC	CGATG\$AC	C	C
		CGATG\$AC	G\$ACCGAT	G	T
		CCGATG\$A	GATG\$ACC	G	C
		ACCGATG\$	TG\$ACCGA	T	A

BWT编码示例

通常为了节省空间，M 中各行字符串只保留 \$ 之前的部分，也就得到所谓的后缀字符串组 Suffix，Suffix 中每一行的字符串都是 S' 的一个后缀。

后缀字符串组	F 列	L 列
\$	\$	G
ACCGATG\$	A	\$
ATG\$	A	G
CCGATG\$	C	A
CGATG\$	C	C
G\$	G	T
GATG\$	G	C
TG\$	T	A

后缀字符串组示例

在求BWT(S)的过程中，需要同时建立索引，以节省后续字符串匹配时间。由于该问题中用到的长串和短串只包含 A、C、G、T 这四种字符，因此建立的索引主要这样三个部分：

- 1、num_a , num_c , num_g , num_t：分别记录A、C、G、T在S中出现的次数
- 2、后缀数组SA：SA[i]表示后缀字符串组Suffix里的第i个字符串的起始字符在S'中的下标，即F[i]在S'中的下标。
- 3、count_num[char][row]：count_num[0][i]表示到L[i - 1]为止，A出现过的次数；count_num[1][i]表示到L[i - 1]为止，C出现过的次数，以此类推。即若L[i]处的字符为A，则要用当前A的计数将count_num[0][i]的值赋完后才对A的计数+1，也就表示在L里的所有字符A中，L[i]处的字符A的排序（该排序从0开始）。

后缀字符串组	SA	count_num[0] (A)	count_num[1] (C)	count_num[2] (G)	count_num[3] (T)
\$	7	0	0	0	0
ACCGATG\$	0	0	0	1	0
ATG\$	4	0	0	1	0
CCGATG\$	1	0	0	2	0
CGATG\$	2	1	0	2	0
G\$	6	1	1	2	0
GATG\$	3	1	1	2	1
TG\$	5	1	2	2	1

索引计算示例

根据上述BWT算法的处理步骤，对 S 进行BWT编码和计算索引的过程对应下述代码操作：

```
//轮转和排序
vector<string>M(len);
for(int i = 0; i < len; i++) {
    string new_s(i + 1, ' ');
    for(int j = 0; j < i + 1; j++)
        new_s[j] = s[len - 1 - i + j];
    M[i] = new_s;
}
sort(M.begin(), M.end());

//得到BWT(S)和编码
string L(len, ' ');
for(int i = 0; i < M.size(); i++) {
    SA[i] = len - M[i].size();
    if(SA[i] == 0)
        L[i] = '$';
    else {
        L[i] = s[SA[i] - 1];
        count_num[0][i] = num_a;
        count_num[1][i] = num_c;
        count_num[2][i] = num_g;
        count_num[3][i] = num_t;

        if(L[i] == 'A') num_a++;
        else if(L[i] == 'C') num_c++;
        else if(L[i] == 'G') num_g++;
        else if(L[i] == 'T') num_t++;
    }
}
```

上述根据BWT的算法步骤对S进行编码和计算索引的复杂度分析：矩阵 M 所占空间为 $O(n^2)$ ，count_num 占空间 $O(n)$ ，SA 占空间 $O(n)$ ，其余占 $O(1)$ ，所以总的空间复杂度为 $O(n^2)$ 。轮转生成 M 的时间为 $O(n^2)$ ，而因为两个字符串比较大小的复杂度为 $O(n)$ ，所以对 M 进行快速排序的复杂度为 $O(n * n \log n)$ ，由 M 得到 L 和相应索引的复杂度为 $O(n)$ ，所以总的时间复杂度为 $O(n * n \log n)$ 。

所以可以看出，直接根据BWT的算法步骤对 S 进行编码和计算索引的时间复杂度和空间复杂度都很高，对于较大的数据来说难以接受。后续经过测试，当 S 的长度达到17万左右时，在执行BWT的过程中就会出现内存不够而导致程序终止的情况。

所以需要对该算法进行改进，即直接利用后缀数组得到BWT(S)和索引，并且对索引进行间隔存储以减小存储空间。

后缀数组和倍增算法

对编码和索引计算的过程进行优化，可以使用倍增算法，直接通过字符串 S' 计算出其后缀字符串的排序，即后缀数组 SA，替换用 S' 轮转得到 M 和对 M 排序的过程。使用倍增算法的时间复杂度为 $O(len * \log len)$ ，空间复杂度 $O(len)$ ，其中 len 是长串 S' 的长度。

在倍增算法中，用到名次数组 Rank。名次数组 Rank[i] 表示的是 S' 中从 i 位置开始的后缀字符串在 M 中排的位置，而后缀数组 SA[i] 表示的是 M 中排在 i 位置的后缀字符串的起始字符在 S' 中的下标。后缀数组 SA 和名次数组 Rank 之间为互逆运算，即 $SA[i] = j, Rank[j] = i$ 。

后缀字符串组	SA	Rank
\$	7	Rank[7]=0
ACCGATG\$	0	Rank[0]=1
ATG\$	4	Rank[4]=2
CCGATG\$	1	Rank[1]=3
CGATG\$	2	Rank[2]=4
G\$	6	Rank[6]=5
GATG\$	3	Rank[3]=6
TG\$	5	Rank[5]=7

排名 Rank 计算示例

倍增算法的主要思路是，对每轮排序的子串的长度进行倍增，即第 k 轮排序是对从每个字符开始的长度为 2^k 的子字符串进行排序，要在该轮中求出其 Rank 值。因为长度为 2^k 的字符串可以由两个长度为 2^{k-1} 的字符串组成，则可以使用基数排序，将前 2^{k-1} 长度的子串看作第一关键字，后 2^{k-1} 长度的子串为第二关键字。而长度为 2^{k-1} 的字符串在上一轮排序中已经有了排序结果，第一关键字和第二关键字在上一轮中都是被排序的对象，所以可以直接利用上一轮的排序结果，然后根据基数排序将第一关键字和第二关键字的排序合并，得到该轮的排序结果。直到 $2^k \geq n$ ，或者名次数组 Rank 中的元素不重复，已经是从 1 到 n ，就能得到最终的后缀数组 SA，而利用后缀数组 SA 和 S' 就可以计算出编码结果 L 和其余索引的值。

倍增算法对应具体处理过程如下：

在倍增算法的每一轮排序中，先用上一轮的排序结果得到该轮第一关键字的排序情况。用上一轮得到的 Rank 值，先统计上一轮名次相同的第一关键字的个数，记录在计数数组 cnt 的相应位置中，再将其转化为前缀和的形式。其中 sig 表示上一层得到的结果中有多少个名次不同的后缀字符串。

```
cnt.assign(sig + 2, 0);
for(int i = 1; i <= len; i++)
    ++cnt[Rank[i]];
for(int i = 1; i <= sig; i++)
    cnt[i] += cnt[i - 1];
```

然后用 Second 数组记录第二关键字排序的结果。对于后 $len - l$ 个后缀字符串，实际长度达不到 l ，后面的字符位上都是空的，显然是最小的，可以直接根据位置得到排序结果；对于其余后缀字符串的第二关键字，直接利用前一轮排序的结果。

```
for(int i = len - l + 1; i <= len; i++)
    Second[++pos] = i;
for(int i = 1; i <= len; i++)
    if(SA[i] > l) Second[ ++pos ] = SA[i] - l;
```

接着就要对第一关键字和第二关键字的排序结果进行合并，得到该轮的排序结果。根据基数排序，第一关键字相同的子串排在一起，然后根据第二关键字确定之间的顺序。

```
for(int i = len; i > 0; i--)
    SA[cnt[Rank[Second[i]]]--] = Second[i];
```

得到该轮的排序结果后，需要用该轮得到的 SA 更新 Rank，以便下一轮的计算。当两个合并后的前缀完全相同时，在 Rank 里的排名就相同。

```
pos = 0;
for(int i = 1; i <= len; i++)
    tmp[SA[i]] = (Rank[SA[i]] == Rank[SA[i - 1]] && Rank[SA[i] + 1] == Rank[SA[i - 1] + 1])? pos : ++pos;

for(int i = 1; i <= len; i++)
    Rank[i] = tmp[i];
```

当这整个倍增排序的循环结束后，就可以得到对应的后缀数组 SA。后续对 L 和索引值的计算与 BWT 算法中的处理相同。

FM索引-序列匹配

得到了BWT(S)和索引后，就可以用其实现对子串的的查询和精确匹配。序列匹配利用了 F 列和 L 列之间的关系和特点：

- 1、对 L 列进行排序就可以得到 F 列；
- 2、L 列的第一个字符是原字符串 S 中的最后一个字符；
- 3、对于某个相同的下标位置，F 列中的字符是 L 列中对应位置字符在 S' 中的下一个字符，也可以说 L 列中的字符是 F 列中对应字符在 S' 中的上一个字符。即对于某个下标 i，F[i] 是 L[i] 在 S' 中的下一个元素。例如，i = 5，则 L[i] = T，F[i] = G，G 是 T 在 S' (ACCGATG\$) 中的下一个元素。

S'	L 列		F 列
ACCGATG\$	G	→	\$
	\$	→	A
	G	→	A
	A	→	C
	C	→	C
	T	→	G
	C	→	G
	A	→	T

L 与 F 的关系

因为 F 列可以由 L 列经过排序得到，所以 L 列中的字符到 F 列中的字符和对应下标存在一个映射关系，即一个原本在 L 中排在 i 位置的字符，在 L 经过排序得到 F 后，在 F 中排在 j 位置，则 L 中的排在 i 位置的字符与 F 中排在 j 位置的字符存在映射关系，且从 L 中的 i 到 F 中的 j 也存在一个映射关系。

此外，该映射是满足稳定性的，即 L 列中相同字符间的相对位置在映射到 F 列后不会发生改变。就比如在 L 列中第3位字符是 A，这个 A 在 L 列中是第一次出现的A，即在 L 列的所有 A 中排第一个，所以要对应到 F 列中第一次出现的 A，也就是第1位字符。所以就是由 L 中的下标 3 映射到 F 中的下标 1。

F 列	L 列
\$	G1
1 A1	\$
A2	G2
C1	A1 3
C2	C1
G1	T1
G2	C2
T1	A2

L 到 F 的映射关系

在确定了字符从 L 列到 F 列的映射关系后，利用 L 列中的字符是 F 列中对应字符在 S' 中的上一个字符进行子串匹配。

将短串记为 sub，将其长度记为 n，将当前短串匹配到的字符记为 now。

首先在 F 列中找到 sub 中第 n 个字符对应的下标范围 [begin, end]，此时就已经匹配完了子串的最后一个字符。因为后缀数组 SA 记录的是 F[i] 在 S' 中的下标，所以此时 SA 中下标范围在 [begin, end] 内的值就对应着 S' 中 sub 的第 n 个字符出现的位置。

然后需要匹配 sub 中第 n - 1 个字符。因为 L 列中的字符是 F 列中对应字符在 S' 中的上一个字符，所以在 L 列中以 F 中的下标 [begin, end] 为对应范围，查找等于 sub 的第 n - 1 个字符的第一个位置和最后一个位置。

```
lbegin = L.find(now, begin);
lend = L.rfind(now, end);
```

再把这两个位置的下标映射到 F 列上，就可以得到新的 [begin', end'] 范围。映射的过程利用记录的 count_num[char][row]。取 count_num[now][lbegin]，即 L 中 lbegin 位置的 now 字符在所有的 now 字符中排第几位，将该值加上 A 在 F 中的起始位置后，得到的就是 L 中的 lbegin 映射到 F 中的下标，也就是新的 begin 位置。同理可以得到新的 end。此时 SA 中下标范围在 [begin', end'] 内的值就是 sub[n - 1, n] 在 S 中的匹配位置。

下述为 now 是 A 字符时的代码示例，其中 begin_a 中记录的是 A 字符的 F 列中的起始位置。

```
if(now == 'A') {
    begin = begin_a + count_num[0][lbegin];
    end = begin_a + count_num[0][end];
}
```

继续该过程，直到 L 的限定下标范围中无法找到此时 sub 中匹配到的字符，或者 sub 中所有字符都匹配时，短串的匹配完成。

子串匹配的过程示例如下，整个过程相当于是在 L 列中进行匹配，然后把可以匹配的字符映射到 F 列上，得到一个范围，该范围可以到 SA 中对应 S 中已匹配的部分子串的位置，也可以到 L 列中寻找下一个匹配的子串字符。

CGA		
S'	F 列	L 列
	\$	G
	A	\$
	A	G
ACCGATG\$	C	A
	C	C
	G	T
	G	C
	T	A

CGA		
S'	F 列	L 列
	\$	G
	A	\$
	A → G	
ACCGATG\$	C	A
	C	C
	G	T
	G	C
	T	A

CGA		
S'	F 列	L 列
	\$	G
	A	\$
	A	G
ACCGATG\$	C	A
	C	C
	G	T
	G	C
	T	A

CGA		
S'	F 列	L 列
	\$	G
	A	\$
	A	G
ACCGATG\$	C	A
	C	C
	G	T
	G → C	
	T	A

CGA		
S'	F 列	L 列
	\$	G
	A	\$
	A	G
ACCGATG\$	C	A
	C	C
	G	T
	G	C
	T	A

CGA			
S'	F 列	L 列	SA
	\$	G	7
	A	\$	0
	A	G	4
ACCGATG\$	C	A	1
	C → G → 2		
	G	T	6
	G	C	3
	T	A	5

字符串匹配过程示例

索引存储优化

因为长串 S 本身的长度 len 的值很大，L 本身就达到了原来长串的大小，且还有空间复杂度与 L 相同的几个索引，对 L 和索引的存储将占到很大的空间。当前测试中最大的长串的文件大小为894M，对应要进行存储的 L 和索引的大小达到6G左右，占到很大的空间，存储过程也相当耗时。所以需要索引的存储进行优化。

对索引中的后缀数组 SA 和记录各位置各字符已出现次数的 count_num 进行存储优化，即将其进行间隔存储。这里将 SA 和 count_num 值有被记录到的对应位置称为checkpoint。当需要某个位置的索引值时，如果该位置是 checkpoint，则直接取出对应记录的值。如果该位置不是 checkpoint，则需要找到离其最近的 checkpoint，利用 checkpoint 的索引值计算出该位置上的索引值。

例如下图中对 G 的 count_num 值计算。对于某个不是 checkpoint 的位置，利用对应位置前最近的 checkpoint 的值，再加上从 checkpoint 到该位置前 L 中对应字符的出现次数，即得到该位置的 count_num 值。

L 列	count_num[2] (G)
G	0
\$	0+1
G	1
A	1+1
C	2
T	2+0
C	2
A	2+0

G 对应的count_num 值计算示例

将 count_num 的间隔记为 COUNT_CKPT_LEN。下述即为计算对应 count_num 值的过程。

```

int pre_i = now_i / COUNT_CKPT_LEN;
int count = count_num[char_num][pre_i]; //前面最近的checkpoint位置上的值
pre_i = pre_i * COUNT_CKPT_LEN; //前面最近的checkpoint位置
for(int i = pre_i; i < now_i; i++) { //从checkpoint到该位前的对应字符计数
    if(L[i] == target)
        count++;
}

```

对应计算 SA 的过程同理。利用 L 列中的字符是 F 列中对应字符在 S' 中的上一个字符这一关系，仿照子串匹配的过程，在 S' 中重复往前查找前一个字符，直到找到的字符所在位置是 checkpoint 为止。利用找到的 checkpoint 在 SA 中的对应值，加上在 S' 中往前查找前一个字符的次数，就可以得到该位置的 SA 值。

sub = CGA

F	L	SA
\$ACCGATG		7
ACCGATG\$		
ATG\$ACCG		
CCGATG\$A		1
CGATG\$A		1+1=2
G\$ACCGAT		
GATG\$ACC		3
TG\$ACCGA		5

间隔存储的 SA 计算示例

下述是计算对应位置 SA 的过程：

```

if(i + 1 == 1 || ((i + 1) % SA_CKPT_LEN == 0)) { //要找的刚好是有记录的checkpoint上的值
    ans.push_back(SA[(i + 1) / SA_CKPT_LEN + 1] - 1 + file_num * CUT_LEN);
}
else {
    int pre = 0; //记录往前查找的次数
    int now_i = i;
    //利用 F 和 L 的关系，一直往前找，直到找到有记录的checkpoint的位置
    while(now_i + 1 != 1 && ((now_i + 1) % SA_CKPT_LEN != 0)) {
        char temp = L[now_i];
        if(temp == 'A')
            now_i = begin_a + Count_Num(temp, now_i);
        else if(temp == 'C')
            now_i = begin_c + Count_Num(temp, now_i);
        else if(temp == 'G')
            now_i = begin_g + Count_Num(temp, now_i);
        else if(temp == 'T')
            now_i = begin_t + Count_Num(temp, now_i);
        else if(temp == '$') {
            now_i = -1;
            break;
        }
    }
    pre++;
}

```



```

    }
    if(now_i != -1)
        ans.push_back(SA[(now_i + 1) / SA_CHKPT_LEN + 1] - 1 + pre + file_num *
CUT_LEN);
    else
        ans.push_back(0 + pre);
}

```

分治处理

由于长串过长，在对其进行预处理时需要对其进行拆分，对拆分后的各子串进行编码和索引计算。将各子串的编码结果和索引分别记录到文件中，即每个子串对应一个编码文件和一个索引文件。由于可能会将能够匹配短串的部分拆分到两个子串中，所以在拆分的过程中，需要给前后两个子串一定的重复部分，重复部分的大小即为短串的长度。

对应切割处理如下，其中 CUT_LEN 为长串的切割大小，SUB_LEN 为短串长度，i 为当前切割次数。

```
str = long_string.substr(i * CUT_LEN, CUT_LEN + SUB_LEN);
```

预处理结束后，后续进行匹配时，需要将长串拆分出的各子串对应的编码文件和索引文件——读出，代入上述匹配过程，将各子串中匹配出的结果记录在一起，得到最终匹配结果。

4.实验结果及分析

长串预处理

在对长串进行预处理时，将切割长度 CUT_LEN 设置为 1000000，再加上短串长度 SUB_LEN 为 200，实际从长串中截取的每个子串的长度为 CUT_LEN + SUB_LEN = 1000200。

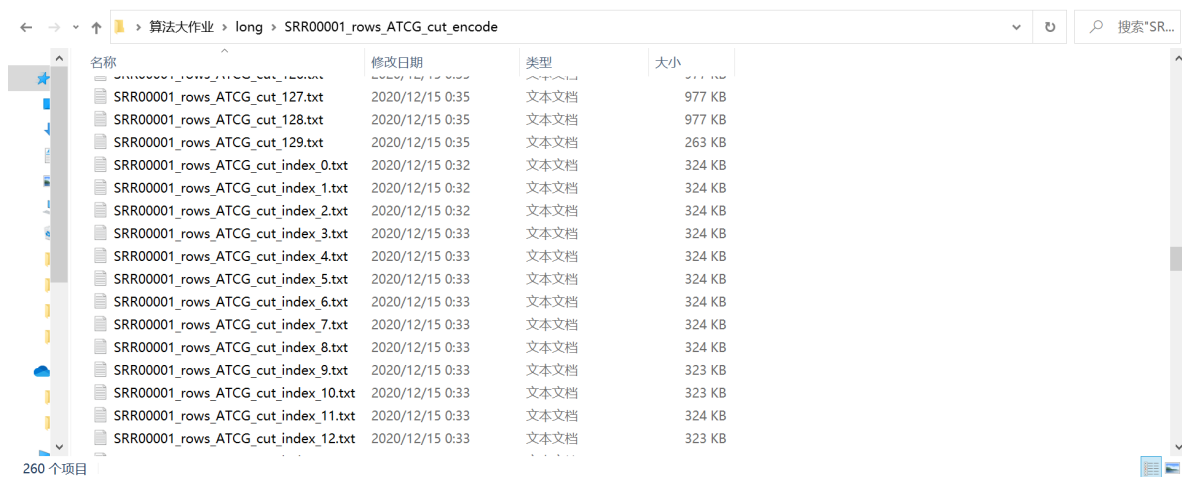
第一个长串的文件为SRR00001_rows_ATCG.txt，文件原始大小123M。将其进行预处理后得到260个文件，其中一半为编码文件，一半为索引文件，文件的总大小为164M。预处理运行结果和部分文件截图如下：

```

PS C:\Users\74285\Desktop\算法大作业> cd "c:\Users\74285\Desktop\算法大作业\" ; if ($?) { g++ BTW3.cpp -o BTW3 } ; if ($?) { .\BTW3 }
time = 181.421s
cut and encode SRR00001_rows_ATCG done

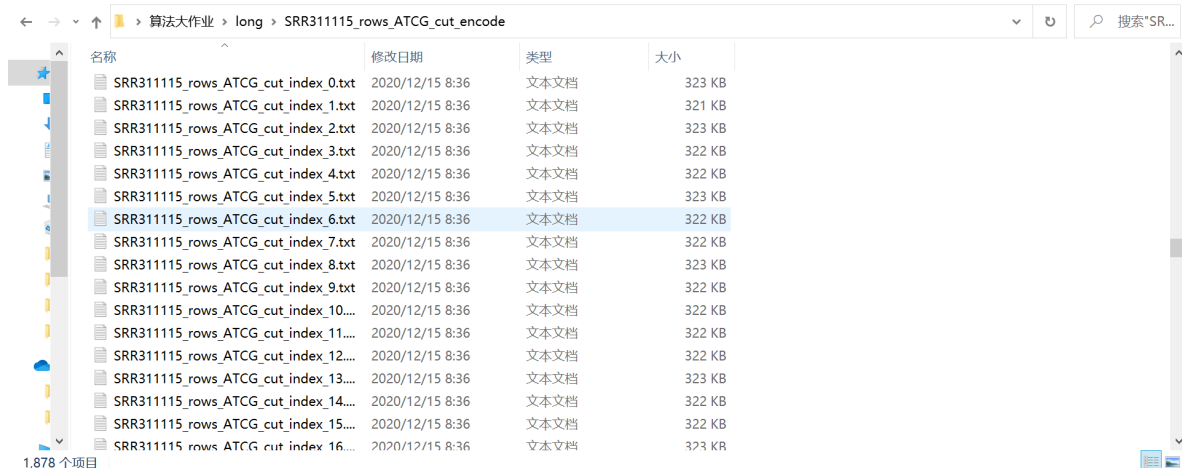
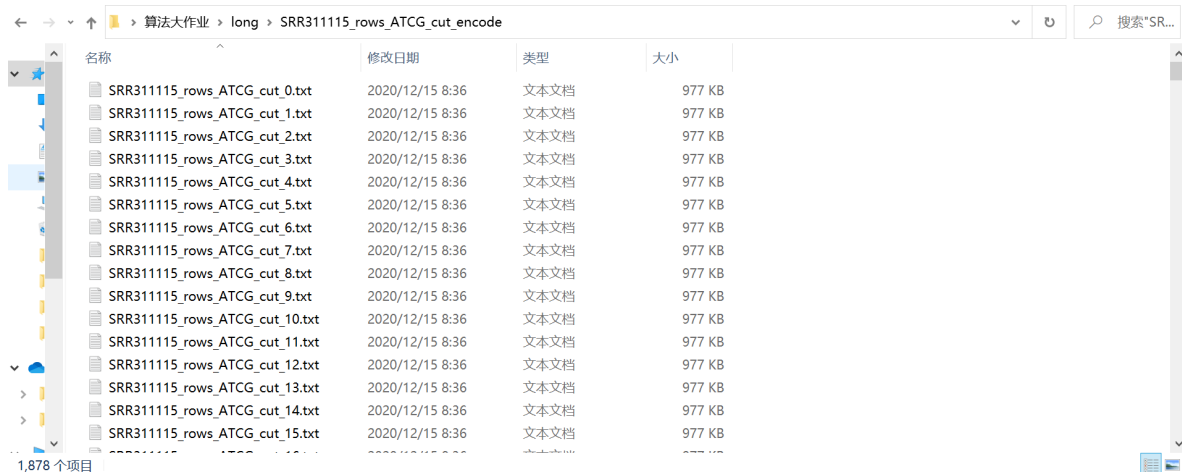
```

名称	修改日期	类型	大小
SRR00001_rows_ATCG_cut_0.txt	2020/12/15 0:32	文本文档	977 KB
SRR00001_rows_ATCG_cut_1.txt	2020/12/15 0:32	文本文档	977 KB
SRR00001_rows_ATCG_cut_2.txt	2020/12/15 0:32	文本文档	977 KB
SRR00001_rows_ATCG_cut_3.txt	2020/12/15 0:33	文本文档	977 KB
SRR00001_rows_ATCG_cut_4.txt	2020/12/15 0:33	文本文档	977 KB
SRR00001_rows_ATCG_cut_5.txt	2020/12/15 0:33	文本文档	977 KB
SRR00001_rows_ATCG_cut_6.txt	2020/12/15 0:33	文本文档	977 KB
SRR00001_rows_ATCG_cut_7.txt	2020/12/15 0:33	文本文档	977 KB
SRR00001_rows_ATCG_cut_8.txt	2020/12/15 0:33	文本文档	977 KB
SRR00001_rows_ATCG_cut_9.txt	2020/12/15 0:33	文本文档	977 KB
SRR00001_rows_ATCG_cut_10.txt	2020/12/15 0:33	文本文档	977 KB
SRR00001_rows_ATCG_cut_11.txt	2020/12/15 0:33	文本文档	977 KB
SRR00001_rows_ATCG_cut_12.txt	2020/12/15 0:33	文本文档	977 KB
SRR00001_rows_ATCG_cut_13.txt	2020/12/15 0:33	文本文档	977 KB
SRR00001_rows_ATCG_cut_14.txt	2020/12/15 0:33	文本文档	977 KB
SRR00001_rows_ATCG_cut_15.txt	2020/12/15 0:33	文本文档	977 KB



第二个长串文件为SRR311115_rows_ATCG.txt，文件原始大小894M。将其进行预处理后得到1878个文件，其中一半为编码文件，一半为索引文件，文件的总大小为1.16G。预处理运行结果和部分文件截图如下：

```
PS C:\Users\74285> cd "c:\Users\74285\Desktop\算法大作业\" ; if ($?) { g++ BTW3.cpp -o BTW3 } ; if ($?) { .\BTW3 }
time = 1438.623s
cut and encode SRR311115_rows_ATCG done
```



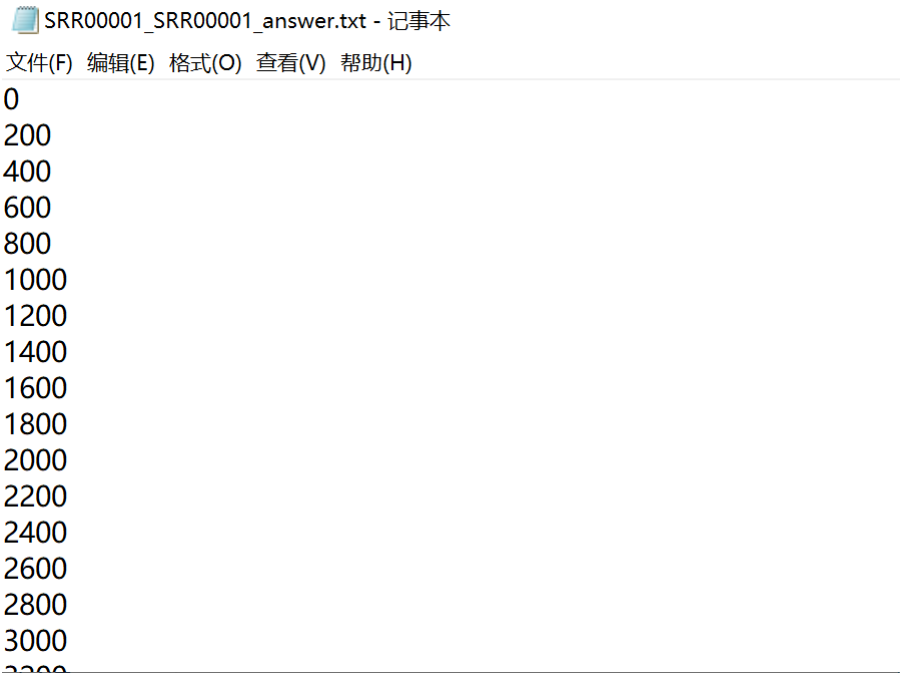
预处理的时间比预计的略长，分析是因为对长串进行拆分时，设置的长度不够长，导致分出了很多的子串。虽然对编码和索引计算的总时间没有影响，但是在对结果进行存储时需要读写的文件数变多，导致时间较长。

在匹配过程中，对于每个短串，如果要找到其在长串中所有匹配结果，则需要读取长串所有的子串编码和索引文件。因为每个短串文件中都有 1000 个短串，对应匹配结果数量较多，因此将匹配结果输出到文件中，每个短串的匹配结果占一行。

长串 SRR00001 匹配短串 SRR00001 的运行结果如下，答案存在文件 SRR00001_SRR00001_answer.txt 中。

```
PS C:\Users\74285> cd "c:\Users\74285\Desktop\算法大作业\" ; if ($?) { g++ BTW3.cpp -o BTW3 } ; if ($?) { .\BTW3 }
time = 3.078s
SRR00001 rows ATCG find done
```

答案的部分截图：



SRR00001_SRR00001_answer.txt - 记事本

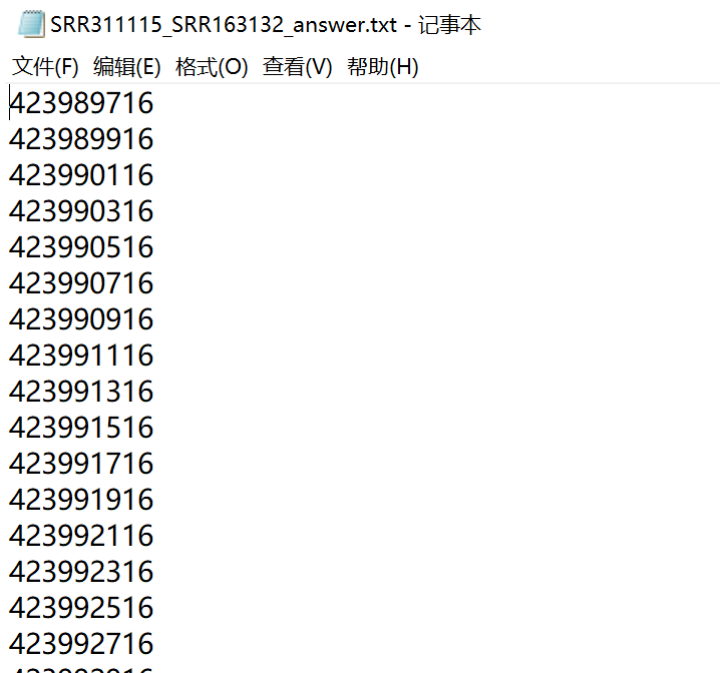
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

0
200
400
600
800
1000
1200
1400
1600
1800
2000
2200
2400
2600
2800
3000
3200

长串 SRR311115 匹配短串 SRR163132 的运行结果如下，答案存在文件 SRR311115_SRR163132_answer.txt 中。

```
PS C:\Users\74285\Desktop\算法大作业> cd "c:\Users\74285\Desktop\算法大作业\" ; if ($?) { g++ BTW3.cpp -o BTW3 } ; if ($?) { .\BTW3 }
time = 23.704s
SRR311115 rows ATCG find done
```

答案的部分截图：



SRR311115_SRR163132_answer.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

423989716
423989916
423990116
423990316
423990516
423990716
423990916
423991116
423991316
423991516
423991716
423991916
423992116
423992316
423992516
423992716
423992916

可以看出，整个匹配的时间是非常短的，在拆分出 939 个子串的长串中匹配 1000 个长度为 200 的短串的时间不到 30s。这是因为用到的 FM索引-序列匹配算法的时间复杂度为短串的长度，即满足一个长度为 m 的短字符串在一个长度为 n 的长字符串中匹配时，时间复杂度为 $O(m)$ 。

5.附加问题

允许 k 个字符失配条件下，短字符串在一个长字符串中的非精确匹配。

想法一

直接在短字符串中去掉 K 个字符。但是去掉 K 个字符的情况数目过多，而且还要考虑到小于 K 个字符失配的情况，所以只实现了对小于等于 K 个字符数目的连续缺失处理。即在进行匹配前，先在短串中去掉小于等于 K 个字符长度的子串，然后再把处理后的短串和长串进行匹配。

即使只处理了连续去掉字符的情况，要遍历所有小于等于 K 个连续字符失配的情况也需要很高的时间复杂度。

```
for(int j = 0; j <= K; j++){ //有几个失配的字符
    vector<int>ansk(0);
    for(int p = 0; p <= sub.size() - j; p++){ //在短串中的失配位置
        if(j == 0 && p != 0)
            break;
        //处理短串
        string subk = (j == 0)? sub : sub.substr(0, p) + sub.substr(p + j);
```

想法二

考虑在FM索引-序列匹配中进行处理。

用数组 `vector<pair<int, int>>temp_find` 记录当前符合匹配条件的下标位置和其失配次数。

```
//记录初始化，对于短串的最后一个字符的匹配情况
vector<pair<int, int>>temp_find(L.size(), pair<int, int>(0, 1));
for(int i = 0; i < L.size(); i++) {
    temp_find[i].first = i;
    if(now == 'A') {
        if(i >= begin_a && i < begin_c)
            temp_find[i].second = 0;
    } else if(now == 'C') {
        if(i >= begin_c && i < begin_g)
            temp_find[i].second = 0;
    } else if(now == 'G') {
        if(i >= begin_g && i < begin_t)
            temp_find[i].second = 0;
    } else if(now == 'T') {
        if(i >= begin_t)
            temp_find[i].second = 0;
    }
}
```

在匹配短串的每一位时，遍历该数组中记录的下标位置，判断 L 对应位上的字符是否与短串中的匹配，不匹配的话则将该下标位置的失配次数加一，如果失配次数超过 K 的话就将该下标位置从数组中删除，没有被删除的下标位置就像上述解释过的 FM 索引-序列匹配的那样，将该下标位置从 L 中映射到 F 中，然后进行短串的下一个匹配。

```
for(int i = sub.size() - 2; i >= 0; i--) {
    now = sub[i];
    int j = 0;
    //遍历每一个记录，判断能否匹配
    while(j < temp_find.size()) {
        char temp_now = L[temp_find[j].first];
        if(temp_now != now) //不能匹配的缺失次数+1
            temp_find[j].second++;
        if(temp_find[j].second > K) { //缺失次数超过限制的得被删除
            temp_find.erase(temp_find.begin() + j);
            continue;
        }
        //映射到F上
        if(temp_now == 'A') {
            temp_find[j].first = begin_a + Count_Num(temp_now,
temp_find[j].first);
        } else if(temp_now == 'C') {
            temp_find[j].first = begin_c + Count_Num(temp_now,
temp_find[j].first);
        } else if(temp_now == 'G') {
            temp_find[j].first = begin_g + Count_Num(temp_now,
temp_find[j].first);
        } else if(temp_now == 'T') {
            temp_find[j].first = begin_t + Count_Num(temp_now,
temp_find[j].first);
        }
        j++;
    }
}
```

当短串匹配完后，记录里剩下的就是满足最多失配 K 个字符的下标结果，直接跟上述解释过的 FM 索引-序列匹配里的一样，在 SA 中找到其在 S 中的对应下标值，最终还需要对结果进行去重。

由于提供的长串短串较大，在测试的时候无法判断结果是否正确，因此用较短的串进行了测试。测试长串为 ACGTTACGTAAGCTTA，短串为 ACGTTA，K 的值设为 2。得到的测试结果如下，可以判断出是正确的结果。

```
PS C:\Users\74285> cd "c:\Users\74285\Desktop\算法大作业\" ; if ($?) { g++ missingK.cpp -o missingK } ; if ($?) { .\missingK }
long : ACGTTACGTAAGCTTA
short : ACGTTA
K : 2
ans :
0 1 2 5 6 12
```

虽然该算法能找到最多失配 K 个字符的匹配结果，但是在处理时需要遍历所有可能的下标位置来进行记录和判断，还存在大量删除记录的操作，所以时间复杂度较高。