算法分析与设计大作业实验报告

孟妍廷 2015202009

一、 实验目的和要求

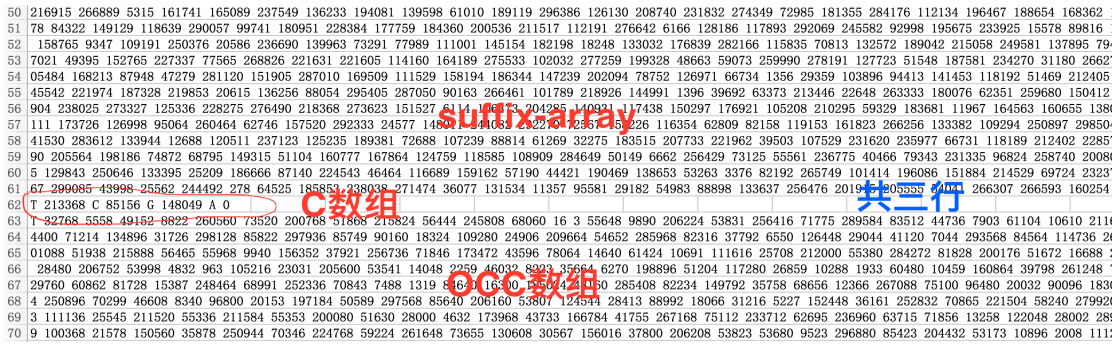
1.实验目的：用 O(m)时间复杂度找出一个长度为 m 的短字符串在一个长度为 n 的长字符串中的精确匹配，其中，m 远小于 n。

2.实验要求：要求基于 BWT 压缩和 FM 索引技术的序列匹配，用 Burrows-Wheeler transform 算法解决该问题。实验提供三个数量级为 900M 的长串和三个取自长串前 200000 个字符的短串。

二、 算法简述

前提：

由于长串的数量级是 900M，一次取出的计算量太大并且产生的索引文件占用的存储空间过大，因此每一次从长串中取 300000 个字符来生成索引，并保证两次取出的字符有 200000 个字符的重叠。同时，每 300000 个字符生成的索引存在一个文件中，也就是说由于每次有 100000 个字符是未被之前的字符串所覆盖的因此共生成 9000 个索引文件。索引文件的结构如下：



1.第一步：建立 BWT 结构——不生成矩阵创建索引：

若按照严格的 Burrows-Wheeler transform 算法建立 FM 索引，首先需要将长串取出，每次左移一位生成新的字符串。因此需要 O(n^2)的矩阵来存储，十分浪费空间。故我选择了不生成矩阵创建索引的方法：

首先将长度为 n 的长串 S 复制一遍生成长度为 2n 的字符串 SS，然后建立一个长为 n 的 int 型数组 A，使得 A[i]=i。接下来对 A 数组做快速排序，但是将 PARTITION 中的比较函数修改为：对于当前比较值 x,y，比较 S.substr(x, n)和 S.substr(y, n)的字典序。这样就不需要存储矩阵，缩小了空间复杂度，最后排序后的 A 数组就是 FM-index 需要的 suffix-array。

2.第二步：利用 suffix-array 建立 C 和 OCC 并写入本地

可知 suffix-array 的值 i 在原始长串中取到的字符就是就是左移 i 次后的字符串的最后一个字符，而 i-1 在原始长串中取到的字符就是就是左移 i 次后的字符串的第一个字符。因此利用 suffix-array 即可还原出左移过程中的第一列和最后一列数组，同时也可以计算出 C 和 OCC。

为了节约运算时间和存储空间，对 OCC 数组进行部分存储，每隔 16 个位置计算一次 OCC，即计 OCC[X,16]，OCC[X,32],OCC[X,48]....之后查询时先找离得最近的存储在计算真实值。

把 C 和 OCC 以及 suffix-array 数组写入索引文件，每个占一行，方便之后读取。

3.第三步：进行精确匹配

首先读取索引文件，还原出 suffix-array，C 和 OCC。

由于最后一列的第 i 个字符在第一列第 i 个字符前面，故对于待匹配的短串，从后往前进行匹配。先找最后一个字符，利用 C 确定这个字符在第一列中出现的范围，在这个范围中根据 OCC 找最后一列是倒数第二个字符 x 的位置，由于我的 OCC 此时是每隔 16 个位置存储一个，因此先找到最接近的位置，再以这个位置为上界计算真实值。得到真实值 count 之后，得到以倒数第一个字符和倒数第二个字符为开头，倒数第三个字符结尾的位置为 count+C[x]。匹配之后得到一个字符串，查看该字符串与待匹配的短串是否相同，若不同，则说明找到的倒数第二个字符的位置不正确（由于满足第一个字符是短串最后一个字符，最后一个字符是短串倒数第二个字符的位置并不唯一），此时回溯，找一个新的位置重新匹配，直到得到的字符串与待匹配短串相同，输出它

在长串中的位置，匹配成功。

三、 时间空间复杂度分析（具体到算法中每一步的复杂度）

设短串长度为 m，长串为 n

第一部分：生成索引

读入 → 时间复杂度 O(1),空间复杂度 O(300000)(相当于 O(m),3000000<<900M)

↓

快速排序 → 时间复杂度 O(mlgm),空间复杂度 O(2m)

↓

计算 C → 时间复杂度 O(m),空间复杂度 O(8)(四中字符+四个数量)

↓

计算 OCC → 时间复杂度 O(m),空间复杂度 O(4+300000/16*4) =O(75004)

↓

将索引存到本地 → 时间复杂度 O(m)，空间复杂度 O(1)

第二部分：精确匹配

读取索引 → 时间复杂度 O(1),空间复杂度 O(300000+8+75004)=O(m)

↓

还原 suffix-array → 时间复杂度 O(1),空间复杂度 O(m)

↓

还原 C → 时间复杂度 O(1),空间复杂度 O(8)

↓

还原 OCC → 时间复杂度 O(m),空间复杂度 O(75004)

↓

找短串最后一个字符的范围 → 时间复杂度 O(1),空间复杂度 O(m)

↓

确定满足倒数第二个字符在最后一列的位置 → 时间复杂度 O(m),空间复杂度 O(m)

↓

精确匹配 → 时间复杂度:单次 O(m),回溯 O(m^2),空间复杂度 O(m)

四、实验源代码

生成索引：

```
import os
#python 的 sorted 函数直接能够实现将字符串按照字典序排序
def Move_To_Right(S):
    SS=S#记录初始状态
    #result=[]
    #result.append(SS)
    result={}
    result[SS]=0
    #suffix_array.append(0)
    for j in range(len(SS)-1):
        temp = SS[0]
        SS = SS[1:] + temp
        #result.append(SS)
        #suffix_array.append(j+1)
        result[SS]=j+1
    return result

#同时也要注意空间复杂度
#一开始想的是根据 results 修改 suffix_array，时间复杂度较大
def BWT():
    suffix_array=[]
    results = []
    result = Move_To_Right(S)
    results = sorted(result.keys())#但是要根据 results 修改 suffix_array
    for i in results:
        suffix_array.append(result[i])
    #print(results,suffix_array)
    result.clear()
    results.clear()
    return suffix_array

#不生成矩阵创建索引
def exchange(A,a,b):
    temp = A[a]
    A[a]=A[b]
    A[b]=temp
def QUICKSORT(S,suffix_array,p,r):
    if p<r:
```

```
            q=PARTITION(S,suffix_array,p,r)
            QUICKSORT(S,suffix_array,p,q-1)
            QUICKSORT(S,suffix_array,q+1,r)

def PARTITION(S,suffix_array,p,r):
    n=len(suffix_array)
    x=suffix_array[r]
    i=p-1
    for j in range(p,r):
        if S[suffix_array[j]:suffix_array[j]+n-1]<S[x:x+n-1]:
            i = i+1
            exchange(suffix_array,i,j)
    exchange(suffix_array,i+1,r)
    return i+1

def BWT1(SS):
    suffix_array=[]
    for i in range(int(len(SS)/2)):
        suffix_array.append(i)
    QUICKSORT(SS,suffix_array,0,len(suffix_array)-1)
    print("FM-index:",suffix_array)
    return suffix_array



#建立 C 和 OCC 数组——当前想法有两层循环
#根据 suffix_array 就可以得到 F 和 L
#默认只有 AGCT 四种字符
def Build_Structure(S):
    SS = S+S
    F=[]
    L=[]
    suffix_array=[]
    suffix_array=BWT1(SS)
    for i in suffix_array:
        F.append(S[i])
        L.append(S[i-1])
    #print(F,L)
    C={}#C 表示的是在第一列的字符串,比字符 X 小的有多少个
    OCC={}#OCC 的的元素 OCC[X][i]表示的意思是在最后一列字符串中,前 i 个字符里有几个字符 X
    #首先求 C
    counta=0#记录 agct 个数
    countg=0
    countc=0
```

```
        #countt=0
        for i in F:
            if i == 'A':
                counta += 1
            elif i == 'C':
                countc += 1
            elif i == 'G':
                countg += 1
        C['A']=0
        C['C']=counta
        C['G']=counta+countc
        C['T']=counta+countc+countg
        #接下来求 OCC
        OCC['A']={}
        OCC['G']={}
        OCC['C']={}
        OCC['T']={}
        i=0
        counta=0#记录 agct 个数
        countg=0
        countc=0
        countt=0
        while i < len(L):
            #i=i+16
            for j in L[i:i+16]:
                if j=='A':
                    counta += 1
                elif j=='G':
                    countg += 1
                elif j=='C':
                    countc += 1
                elif j=='T':
                    countt += 1
            i=i+16
            OCC['A'][i]=counta
            OCC['G'][i]=countg
            OCC['C'][i]=countc
            OCC['T'][i]=countt
        F.clear()
        L.clear()
        return suffix_array,C,OCC
def main():
        file = open('s.txt','r')
        file.read(10)#前面 10 个是记录长度的和制表符
```

```
        begin=100010
        ff=1
        S=file.read(300000)
        while S:
                file.seek(begin,0)
                begin += 100000
                print(S)
                if str(0xFF) in S:
                        S.remove(str(0xFF))
                suffix_array,C,OCC=Build_Structure(S)
                filename = str(ff)+'.txt'
                f = open(filename,'w')
                for i in suffix_array:
                        f.write(str(i)+' ')
                f.write('\n')
                for i in C:
                        f.write(i+' '+str(C[i])+' ')
                f.write('\n')
                for i in OCC:
                        f.write(i+' ')
                        for j in OCC[i]:
                                f.write(str(j)+' '+str(OCC[i][j])+' ')
                f.write('\n')
                ff += 1
                f.close()
                S=file.read(300000)
        file.close()


if __name__ == "__main__":
        main()
```

精确匹配：

```
#根据存入本地的索引生成精确匹配
#注意 OCC 是每隔 16 个存一个，最后一个可能超出了字符串的长度，先找到最近的位置再精确匹配
import re
import math
import datetime

def FM_Reduction(line):#还原 FM-index
        suffix_array = re.split(r' ',line)
        #print(suffix_array[0])
        suffix_array.remove('\n')
        print(len(suffix_array))
```

```
        #suffix_array.remove('')
        return suffix_array

def C_Reduction(line):#还原 C
        C={}
        temp = re.split(r' ',line)
        i=0
        while i<len(temp)-1:
                C[temp[i]]=int(temp[i+1])
                i += 2
        temp.clear()
        return C
def OCC_Reduction(line):#还原 OCC
        OCC={}
        temp = re.split(r' ',line)
        length = 2*int(300000/16) #37500
        #print(length)
        i = 0
        while i<len(temp)-1:
                OCC[temp[i]]={}
                #print(temp[i])
                j=i
                i += 1
                while i<=j+length:
                        OCC[temp[j]][int(temp[i])]=int(temp[i+1])
                        i += 2
                i = j+length
                i += 1
        temp.clear()
        return OCC
def PRECISION(beg,sit,C,suffix_array,ff,aim):#根据最近的位置来计算精确的 OCC
        f = open(r"s.txt")
        f.read(10)
        f.seek((ff-1)*100000,1)
        S = f.read(300000)
        f.close()
        L=[]
        for i in suffix_array:#取出的 suffix 是 str 类的
                if i is not '\n':
                        L.append(S[int(i)-1])
        #print(F,L)
        res = -1
        print("PRECISION-beg",beg)
        for i in range(beg,sit):
```

```
            if L[i]==aim:
                res=i#找到第一个即可
                break
        print("PRECISION-res",res)
        count = 0
        for i in range(0,res):#求在他之前有几个他
            if L[i] == aim:
                count += 1
        count = C[aim]+count
        print("PRECISION",aim,count)
        return count,L[count]

def Match_Second(res,ff,cur,S,R,suffix_array,C):
        f = open(r"s.txt")
        f.read(10)
        f.seek((ff-1)*100000,1)
        SS = f.read(300000)
        f.close()
        aim = S[cur]
        L=[]
        R.append(aim)
        for i in suffix_array:#取出的 suffix 是 str 类的
            L.append(SS[int(i)-1])
        #print(F,L)
        count = 0
        for i in range(0,res):#求在他之前有几个他
            if L[i] == aim:
                count += 1
        print(res,aim,cur,count,C[aim])
        #print(aim,count)
        L.clear()
        #print(aim,res,cur)
        count = C[aim]+count
        print(count)
        print("当前位置",count,suffix_array[count])
        if cur == 0:
            print("该子串取自原长串的第",suffix_array[count],"位置")
            return
        Match_Second(count,ff,cur-1,S,R,suffix_array,C)


def Match_First(S,C,OCC,suffix_array,ff):#匹配
        R=[]
        cur = len(S)-1
```

```
#print(S[0])
rangea=[0,C['C']]
rangec=[C['C'],C['G']]
rangeg=[C['G'],C['T']]
ranget=[C['T'],300000]
r = []
if S[cur] == 'A':
    r = rangea
elif S[cur] == 'C':
    r = rangec
elif S[cur] == 'G':
    r = rangeg
elif S[cur] == 'T':
    r = ranget
flag = False #判断是否匹配成功的指示变量
i=0
while i < r[1]:
    if i == 0:
        sit = math.ceil((i+1)/16)*16
    else:
        sit = math.ceil(i/16)*16
    i=i+17
    if OCC[S[cur-1]][sit]>0:#要进一步精确
        print("sit",S[cur-1],sit,OCC[S[cur-1]][sit])
        beg = sit-16
        while beg < sit:
            print("beg",beg,"cur",cur)
            res,judge=PRECISION(beg,sit,C,suffix_array,ff,S[cur-1])
            if res == -1:
                continue
            if(judge==S[cur-2]):
                Match_Second(res,ff,cur-2,S,R,suffix_array,C)
                R.reverse()
                R.append(S[198])
                R.append(S[199])
                #print(R)
                R=''.join(R)
                if R == S:
                    print("匹配成功")
                    print(R)
                    return
                else:
                    beg += 1
                    print(R)
```

```
                                R=[]
                        else:
                                beg += 1
        return


def main():
        #S 一行 200 个
        S =
'TGCAATTTCTACTATAGTTACTCATTAAGTTATCTAGTAAATTACCATGTAATGTATAAGCTGTGGAATTAATTCTCAGTTACATA
GCATTCACCATGTAATTACTAAACAGTCCCCATCATTACAGCTTTCTAGCCTATTATGAAAAGACCAGAAACATAATTTCAATTA
CAAAGCCCCTGTTGGAACCGAAAGGGTTT'
        ff=1
        filename=str(ff)+".txt"
        f = open(filename,"r")
        line = f.readline()
        suffix_array = FM_Reduction(line)
        for i in range(len(suffix_array)):
                if int(suffix_array[i]) == 0:
                        print(i)
        line = f.readline()
        C = C_Reduction(line)
        line = f.readline()
        OCC=OCC_Reduction(line)
        f.close()
        Match_First(S,C,OCC,suffix_array,ff)


if __name__ == "__main__":
        start = datetime.datetime.now()
        main()
        end = datetime.datetime.now()
        print (end-start)
```

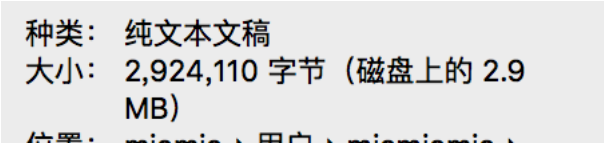
计算时间：

```
if __name__ == "__main__":
    start = datetime.datetime.now()
    main()
    end = datetime.datetime.now()
    print (end-start)
```

五、实验结果（存储空间、查询时间以及结果截图）

1.存储空间：

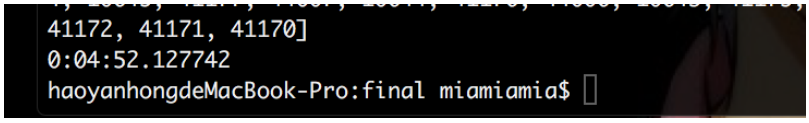
单个部分存储的索引：



共有 9000 个文件，因此总的空间约为 30G

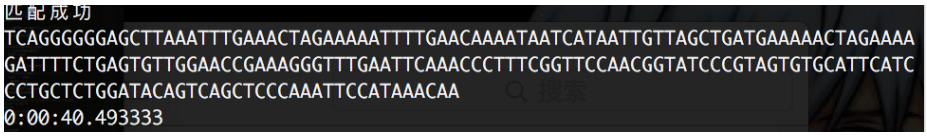
2.查询时间：

生成索引时间：



4 分 52 秒

精确匹配时间：



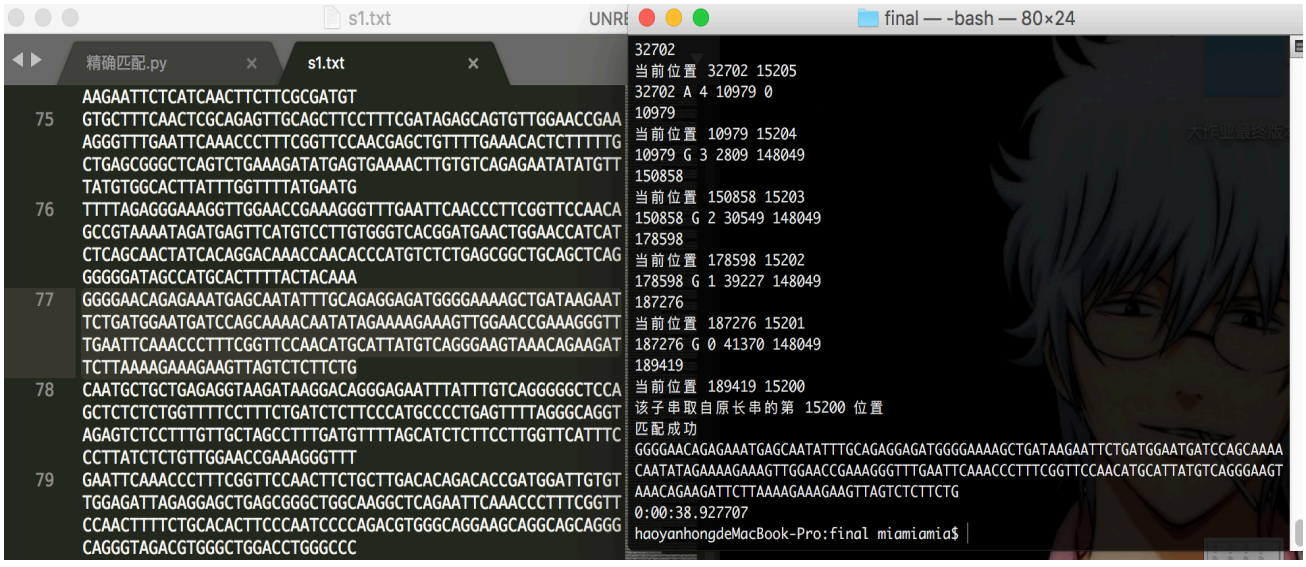
40 秒

3.结果截图

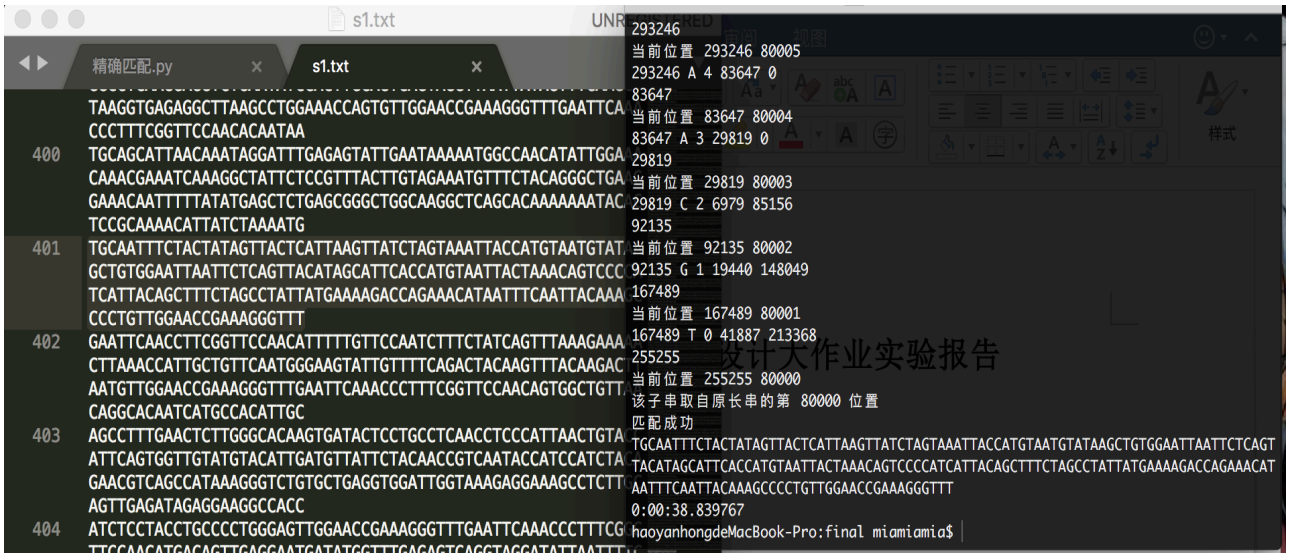
匹配短串第 1 行——结果为从位置 0 开始

匹配短串第 10 行——结果为从位置 200*(10-1)=1800 开始

匹配短串第 77 行——结果为从位置 200*(77-1)=15200 开始



匹配短串第 401 行——结果为从位置 200*(401-1)=8000 开始


s1.txt
精确匹配.py
s1.txt
TAAGGTGAGAGGCTTAAGCCTGGAAACCAGTGTTGGAACCGAAAGGGTTTGAATTCA
CCCTTTCGGTTCCAACACAATAA
400 TGCAGCATTAACAAATAGGATTTGAGAGTATTGAATAAAAATGGCCAACATATTGGA
CAAACGAAATCAAAGGCTATTCTCCGTTTACTTGTAGAAATGTTTCTACAGGGCTGA
GAAACAATTTTTATATGAGCTCTGAGCGGGCTGGCAAGGCTCAGCACAAAAAAATAC
TCCGCAAAACATTATCTAAAATG
401 TGCAATTTCTACTATAGTTACTCATTAAGTTATCTAGTAAATTACCATGTAATGTAT
GCTGTGGAATTAATTCTCAGTTACATAGCATTCACCATGTAATTACTAAACAGTCCC
TCATTACAGCTTTCTAGCCTATTATGAAAAGACCAGAAACATAATTTCAATTACAAA
CCCTGTTGGAACCGAAAGGGTTT
402 GAATTCAACCTTCGGTTCCAACATTTTTGTTCCAATCTTTCTATCAGTTTAAAGAAA
CTTAAACCATTGCTGTTCAATGGGAAGTATTGTTTTCAGACTACAAGTTTACAAGAC
AATGTTGGAACCGAAAGGGTTTGAATTCAAACCCTTTCGGTTCCAACAGTGGCTGTT
CAGGCACAATCATGCCACATTGC
403 AGCCTTTGAACTCTTGGGCACAAGTGATACTCCTGCCTCAACCTCCCATTAACTGTA
ATTCAGTGGTTGTATGTACATTGATGTTATTCTACAACCGTCAATACCATCCATCTA
GAACGTCAGCCATAAAGGGTCTGTGCTGAGGTGGATTGGTAAAGAGGAAAGCCTCTT
AGTTGAGATAGAGGAAGGCCACC
404 ATCTCCTACCTGCCCCTGGGAGTTGGAACCGAAAGGGTTTGAATTCAAACCCTTTCG
293246
当前位置 293246 80005
293246 A 4 83647 0
83647
当前位置 83647 80004
83647 A 3 29819 0
29819
当前位置 29819 80003
29819 C 2 6979 85156
92135
当前位置 92135 80002
92135 G 1 19440 148049
167489
当前位置 167489 80001
167489 T 0 41887 213368
255255
当前位置 255255 80000
该子串取自原长串的第 80000 位置
匹配成功
TGCAATTTCTACTATAGTTACTCATTAAGTTATCTAGTAAATTACCATGTAATGTATAAGCTGTGGAATTAATTCTCAGT
TACATAGCATTCACCATGTAATTACTAAACAGTCCCCATCATTACAGCTTTCTAGCCTATTATGAAAAGACCAGAAACAT
AATTTCAATTACAAAGCCCCTGTTGGAACCGAAAGGGTTT
0:00:38.839767
haoyanhongdeMacBook-Pro:final miamiamia$
样式