生物信息学中的字符串算法

崔皓玮

Github:https://github.com/misaka19260817/Survival-Analysis-and-String-Algorithms

背景

DNA、RNA、蛋白质的一级结构可以视为线性排列的字符串，许多生物学问题都可以转化为字符串问题，例如：寻找DNA上的重复序列能够帮助我们进一步了解生物进化历程，在预测蛋白质功能时删除匹配超过四分之一的氨基酸序列以避免冗余……在生物信息学中我们需要能够高效地处理字符串问题，因此需要使用各种字符串算法。

1. 字符串哈希

问题：给定一个DNA序列(s1)和一个DNA片段(s2)，求该片段在整个序列中出现的所有位置。

解决方法：使用字符串哈希。将字符串看作一个131进制的数（取其他质数也可以）对998244353取模（用其他质数作为除数也可以），如果s1的某个长度为len(s2)的子串哈希值与s2哈希值相同就认为这两个串相同，扫描一遍s1，维护长度为len(s2)的子串的哈希值即可。时间复杂度为O(n)。

```
M=998244353 #除数选择质数，尽量避免哈希冲突
s1=input()
s2=input()
h=H=0
for i in range(len(s2)):
    h=(h*131+ord(s1[i]))%M #进制也选取质数，尽量避免哈希冲突
    H=(H*131+ord(s2[i]))%M
pw=1
for i in range(len(s2)-1):
    pw=pw*131%M

i=0
while True:
    if h==H: #哈希值相等，认为s2在该处出现
        print(i,end=' ')
    if i+len(s2)==len(s1):
        break
    h=((h-ord(s1[i])*pw%M+M)*131+ord(s1[i+len(s2)]))%M #维护s1中长度为len(s2)的子串的哈希值
    i+=1
```

运行结果：

```
ACGTGTGAACGTGGACT
GTG
2 4 10
```

```
GCTCTTCTTGGAGGGCTTGGTACCAGAGCAAATTTTACCAGAGAGTACAATTGCATTGATGCAGTAGAATTGGGGCATGC
GCACATAACATAAGCCTCTGACAACGCCAGATAAGGCCAACCGCCCGGCGGCATGTCAGTGTATCCCTCATACAATTCGT
GCGACTCCCTTTCATGTGTTTAGCACCACAATGTGATTGCAAAAGGAAGGATTTGTCTGTCGACCTGTTGGTAGGATACC
CCCGCTGTTCCTAGTGGGATTACCAAGGGACGATGGGAAGCAGCGGCAGGTGCTTTCGCCCTTCTCGCTTAGCGTGTAGG
ACTGGGCTTGCTTGAAGACTAAGTTTGGTGACGTTCGTAGGCCACAACTTGGTCCGAGCATTCCACACCCGTAGTCTTCT
TTTGAAGCACTGGTTATATACTCTAAAGCCAGGGGGTTAGAATGTGAAAAATTTCCAGCAGTGTACCTGTGAGAGCCTTT
TGAGGCAGATTCCTGACAGTGTAATACGGCCGCCTATCTCCACTGCCCATAGTCGACGTCCCCTAGTCTGGGTACGGGAG
AGACCTATTTGTCGCTATTTTACCTACCGTTCGCCATCTAGTCCTCTTCGCCTTGATGGCTTGACATGCCCGCCTGCTAA
TGGGTAGCGCAGACCGGCTATTACCGTAACTGCGGACTCTCACGCTATTATCTCTTGTGTCAGGAACACTCGCGGAACCA
TGTTCTCTCTTCCACATCACCTGGCTCAATCCGTGTCTAGGAGAAAATAGTGTACGAGTTTGGCTTGCATCTCATCCCAT
GCCCCAAGCTAGTGTTTATACGAAGACAGGATGCCCTCGCTTCCCCACTTTAGCGGACAGCGTTTGGCTTATGACGGAAT
CAACAACGGGACCGGACATTCGCGTCAGTCATTGGCACGGCATTCCGCATCGGCTACCGGCCTCTCTACTATTCGACAGA
CTCAGCCGACGAACAAAGTTCTTGTTGACTGCACCTGGTG
TCAT
147 171 791 908
```

2. KMP算法

问题：给定一个DNA序列(s1)和一个DNA片段(s2)，求该片段在整个序列中出现的所有位置。

解决方法：先求出s2的Next数组。Next[i]表示s2[1..i]的最长公共前后缀（不包括s2[1..i]自身）的长度，例如ACGAC的最长公共前后缀为AC。求Next数组的过程实质上是s2串进行自身匹配的过程，遇到失配的情况就不断通过Next数组跳到更前面位置直到下一位匹配为止。将s1与s2匹配时，我们也可以采取类似的方法，找出s2在s1中出现的所有位置。每次i向后移动一位时，j最多向后移动1位，因此算法的时间复杂度是O(n)。

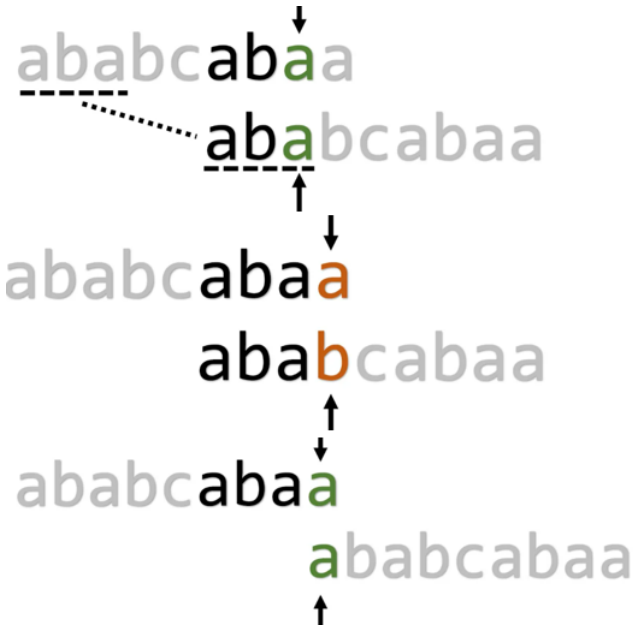

图1 求Next数组：s2串的自身匹配

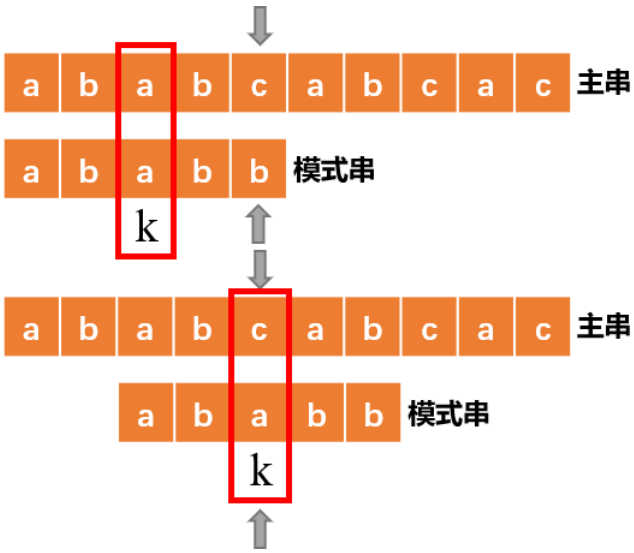

图2 主串(s1)与模式串(s2)的匹配

```
s1=input()
s2=input()
s1=" "+s1 #将字符串下标改为从1开始，方便后续处理
s2=" "+s2
Next=[0]*(len(s2)+1)

#求Next数组
Next[1]=0
j=0
for i in range(2,len(s2)):
    while j!=0 and s2[j+1]!=s2[i]: #s2[j+1]!=s2[i]，说明s2[1..j+1]不是s2[1..i]的公共前后缀，j通过Next数组跳到更前面的位置
        j=Next[j]
    if s2[j+1]==s2[i]: #s2[1..j+1]是s2[1..i]的最长公共前后缀
        j+=1
    Next[i]=j

j=0
for i in range(1,len(s1)):
    while j!=0 and (j+1>=len(s2) or s2[j+1]!=s1[i]): #失配，j跳回前面
        j=Next[j]
    if s2[j+1]==s1[i]: #s2[1..j+1]与s1[1..i]的后缀匹配上
        j+=1
    if j+1>=len(s2): #整个s2串都匹配上了，输出位置（输出的下标从0开始）
        print(i-(len(s2)-1),end=' ')
```

运行结果：

```
ACGTGTGAACGTGGACT
GTG
2 4 10
```

```
GCTCTTCTTGGAGGGCTTGGTACCAGAGCAAATTTTACCAGAGAGTACAATTGCATTGATGCAGTAGAATTGGGGCATGC
GCACATAACATAAGCCTCTGACAACGCCAGATAAGGCCAACCGCCCGGCGGCATGTCAGTGTATCCCTCATACAATTCGT
GCGACTCCCTTTCATGTGTTTAGCACCACAATGTGATTGCAAAAGGAAGGATTTGTCTGTCGACCTGTTGGTAGGATACC
CCCGCTGTTCCTAGTGGGATTACCAAGGGACGATGGGAAGCAGCGGCAGGTGCTTTCGCCCTTCTCGCTTAGCGTGTAGG
ACTGGGCTTGCTTGAAGACTAAGTTTGGTGACGTTCGTAGGCCACAACTTGGTCCGAGCATTCCACACCCGTAGTCTTCT
TTTGAAGCACTGGTTATATACTCTAAAGCCAGGGGGTTAGAATGTGAAAAATTTCCAGCAGTGTACCTGTGAGAGCCTTT
TGAGGCAGATTCCTGACAGTGTAATACGGCCGCCTATCTCCACTGCCCATAGTCGACGTCCCCTAGTCTGGGTACGGGAG
AGACCTATTTGTCGCTATTTTACCTACCGTTCGCCATCTAGTCCTCTTCGCCTTGATGGCTTGACATGCCCGCCTGCTAA
TGGGTAGCGCAGACCGGCTATTACCGTAACTGCGGACTCTCACGCTATTATCTCTTGTGTCAGGAACACTCGCGGAACCA
TGTTCTCTCTTCCACATCACCTGGCTCAATCCGTGTCTAGGAGAAAATAGTGTACGAGTTTGGCTTGCATCTCATCCCAT
GCCCCAAGCTAGTGTTTATACGAAGACAGGATGCCCTCGCTTCCCCACTTTAGCGGACAGCGTTTGGCTTATGACGGAAT
CAACAACGGGACCGGACATTCGCGTCAGTCATTGGCACGGCATTCCGCATCGGCTACCGGCCTCTCTACTATTCGACAGA
CTCAGCCGACGAACAAAGTTCTTGTTGACTGCACCTGGTG
TCAT
147 171 791 908
```

3. AC自动机（Aho-Corasick automaton）

问题1：给定n个DNA片段(s1,s2,...,sn)和一个DNA序列(t)，求哪些片段在序列中出现过。

解决方法：使用AC自动机，大致相当于trie树上的KMP算法。先根据模板串s1~sn建立一棵trie树，此时树上结点有一部分孩子为空，将会导致匹配失败。我们求出树上每个点的fail指针（类似于KMP中的Next数组），从根到fail[i]形成的字符串是从根到点i形成的字符串的后缀的最长公共前缀。当我们发现某个点的某个孩子不存在时（例如`ch[x][i]`），进行赋值操作`ch[x][i]=ch[fail[x]][i]`(fail[x]在x之前已经处理过了，所以`ch[fail[x]][i]`一定存在)。这样就避免了匹配失败，并且匹配过程中指针始终位于当前能够匹配的最长位置。时间复杂度为O(4*∑len(si))。

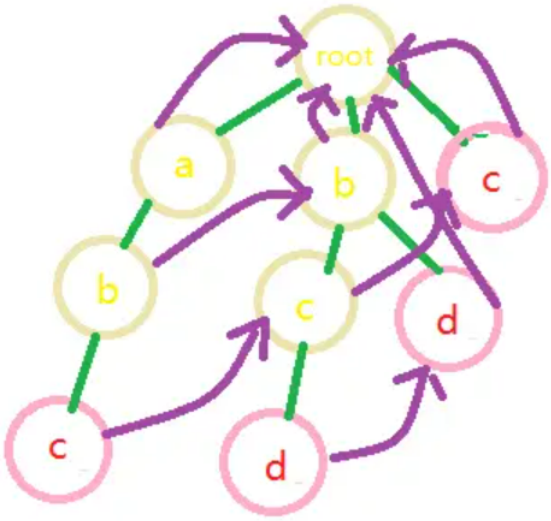

图3 trie树和fail指针

```
from queue import Queue

id={'A':0,'C':1,'G':2,'T':3}
n=int(input())
s=[0]*n
total_len=0
for i in range(n):
    s[i]=input()
    total_len+=len(s[i])
```

```
ch=[[0 for i in range(4)] for j in range(total_len+1)]
fail=[0 for i in range(total_len+1)]
v=[False for i in range(total_len+1)]
p=[[] for i in range(total_len+1)]

#建立trie树
l=0
for i in range(n):
    k=0
    for j in s[i]:
        if ch[k][id[j]]==0:
            l+=1
            ch[k][id[j]]=l
        k=ch[k][id[j]]
    p[k].append(i) #在trie树上s[i]串的终止位置结点存放编号i

#求出fail数组，补全每个结点的所有孩子
que=Queue()
for i in range(4):
    if ch[0][i]!=0:
        que.put(ch[0][i])
while not que.empty():
    Begin=que.get()
    for i in range(4):
        if ch[Begin][i]!=0:
            que.put(ch[Begin][i])
            fail[ch[Begin][i]]=ch[fail[Begin]][i]
        else:
            ch[Begin][i]=ch[fail[Begin]][i]

t=input()
j=0
print("在序列中出现过的片段：",end=' ')
#进行匹配
for i in t:
    j=ch[j][id[i]]
    k=j
    while k!=0 and v[k]==False: # v数组防止有结点重复访问，避免时间复杂度的退化
        v[k]=True
        for x in p[k]:
            print(s[x],end=' ')
        k=fail[k]
```

运行结果：

```
5
GTG
ACG
CCT
AA
TTT
ACGTGTGAACGTGGACT
在序列中出现过的片段：  ACG GTG AA
```

```
20
CTT
GGATC
CCTTT
CAA
CCA
CGAAA
CGT
CGAAG
CCGG
GGGT
CCG
ACGCC
AGACG
ACG
TAAGG
CATAG
ATA
TAGC
ATC
TCGGA
TGGTCACACGATAACTTTGTGCGCACTACGGTGTAGAGACAAGGGCACCAGACCCGGGTCTATTTACTCTTGGCCTACTG
CGGGCAAGGTGATCCTGCGGACATGGCACCCGTCATTGAAGTGTCCTCTCGCAGGTATATCAGCAGTCCACGCCGTCTTA
GGGCGGCGTTGAACAGGGAGTCCTCGTGACAAGACCGCGGTTAGGTCATGCCCCCTATCAACGGGCCTCTCATCACCTCA
CTCGCAGACAAACTATCAAGCAGCAGCCTCGATTCATTACAAAGGGAGAAGGGAGCACGCATGCCTCGCCCACGCAGTAA
ATGAAGTAGCGCCTTGCGGAAACATTCCTAGGAACGCGTCCTCATGGTAACTAAAAGTCTATCCCAAGAACAAGTTTCGC
CGACACTGAGGTACAAGGGCCATCTTGAGCTTCGAAAAAGTCAATTGCCAGAAAAAGTACTCTTAGTTGACGATGCTGTC
CCAGCGAAATAAAATCAATGGCCCTGTAACTATGTTCTACGAATCTGCTCAATACTTCAGTCCGTACGTAGAAATAAATA
CGACCTTTAGACAGTCCCGGTATACTTGCGGCAAGGTCATCTAACCAGGTCCTTGACAAGGTTGCATCTCTCCGGGAAGG
CTAGGACCAGTCGATATGCTTTCGCGCGTCAGTCGGGGGCTGTCGCCAGACGCCGCGTACAGAATGCCCAGTGAAAACGG
AAAGTATCAAAATTCACCTGTGCGACGCTGTTTTAAACGCTTGTGTTTGCCTTTGACCGTGCCCTCGATAGCAAACCAGG
CATCACCAGATCATCAGGTTCGGTGAAGGAGCGTACAGGGATGGGTAGGAACGATTAATTATTGCGTGCGTTGCTTTAAC
AGACTGATGAAAACCAGTCGTTGGTCGGTTGGCAGGAAGTGACCTAGATGACAAACCTAGCGAGGACCTTCTTGCACCCG
ACCACCACGCCGTTATCCCGGATGTGGTCATCCCGCTACA
在序列中出现过的片段： ACG ATA CTT CAA CCA CCG CCGG GGGT ATC CGT ACGCC TAGC CGAAA
CCTTT AGACG
```

问题2：给定n个DNA片段(s1,s2,...,sn)和一个DNA序列(t)，求出在序列中出现次数最多的片段（可能有多个）。

解决方法：同样是建立AC自动机，注意到将fail指针反转可以形成一棵树（fail树）。按照t串字符的顺序在AC自动机上走，假设在遍历到t[i]时走到结点x，则对x到树根的链上所有结点都产生1的贡献（因为这些点都是t[0..i]的后缀）。当然我们不需要每次都从x跳到根，只要在x结点标记一下（v[x]+=1），然后在fail树上进行动态规划即可。时间复杂度仍然是线性的。

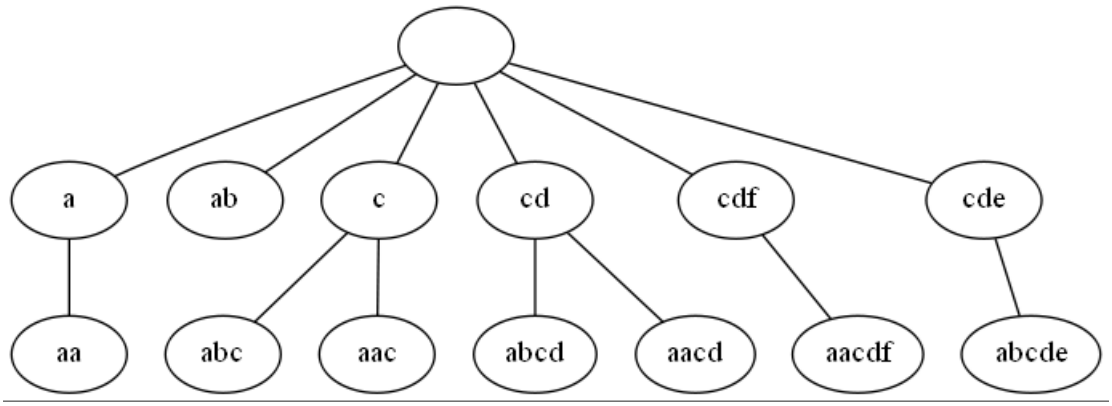

图4 fail树：每个结点的字符串都是子树中其他结点字符串的后缀

```
from queue import Queue

id={'A':0,'C':1,'G':2,'T':3}
n=int(input())
s=[0]*n
total_len=0
for i in range(n):
    s[i]=input()
    total_len+=len(s[i])

ch=[[0 for i in range(4)] for j in range(total_len+1)]
fail=[0 for i in range(total_len+1)]
p=[0 for i in range(n)]

#建立trie树
cnt=0
for i in range(n):
    k=0
    for j in s[i]:
        if ch[k][id[j]]==0:
            cnt+=1
            ch[k][id[j]]=cnt
        k=ch[k][id[j]]
    p[i]=k

head=[0 for i in range(cnt+1)]
adj=[0]
Next=[0]
l=0

def addedge(u,v,l): #连边，使用邻接表
    adj.append(v)
    Next.append(head[u])
    head[u]=l

#求出fail数组，补全每个结点的所有孩子，并得到fail树
que=Queue()
for i in range(4):
    if ch[0][i]!=0:
        que.put(ch[0][i])
        l+=1
        addedge(0,ch[0][i],l) #连边
while not que.empty():
    Begin=que.get()
```

```
    for i in range(4):
        if ch[Begin][i]!=0:
            que.put(ch[Begin][i])
            fail[ch[Begin][i]]=ch[fail[Begin]][i]
            l+=1
            addedge(ch[fail[Begin]][i],ch[Begin][i],l) #连边
        else:
            ch[Begin][i]=ch[fail[Begin]][i]

t=input()
v=[0 for i in range(cnt+1)]
x=0
for c in t:
    x=ch[x][id[c]]
    v[x]+=1 #当前走到x点，标记一下

def dfs(x): #使用dfs进行fail树上的动态规划
    y=head[x]
    while y!=0:
        dfs(adj[y])
        v[x]+=v[adj[y]]
        y=Next[y]

dfs(0)
k=0
for i in range(n):
    if v[p[i]]>k:
        k=v[p[i]]
print("在序列中出现最多的片段的出现次数：",k)
print("出现次数最多的片段有：")
for i in range(n):
    if v[p[i]]==k:
        print(s[i])
```

运行结果：

```
5
GTG
AGT
CCT
TC
AA
AGTGCATAGTGAAGTCCCGTGGGA
在序列中出现最多的片段的出现次数： 3
出现次数最多的片段有：
GTG
AGT
```

```
20
CTT
GGATC
CCTTT
CAA
```

```
CCA
CGAAA
CGT
CGAAG
CCGG
GGGT
CCG
ACGCC
AGACG
ACG
TAAGG
CATAG
ATA
TAGC
ATC
TCGGA
TGGTCACACGATAACTTTGTGCGCACTACGGTGTAGAGACAAGGGCACCAGACCCGGGTCTATTTACTCTTGGCCTACTG
CGGGCAAGGTGATCCTGCGGACATGGCACCCGTCATTGAAGTGTCCTCTCGCAGGTATATCAGCAGTCCACGCCGTCTTA
GGGCGGCGTTGAACAGGGAGTCCTCGTGACAAGACCGCGGTTAGGTCATGCCCCCTATCAACGGGCCTCTCATCACCTCA
CTCGCAGACAAACTATCAAGCAGCAGCCTCGATTCATTACAAAGGGAGAAGGGAGCACGCATGCCTCGCCCACGCAGTAA
ATGAAGTAGCGCCTTGCGGAAACATTCCTAGGAACGCGTCCTCATGGTAACTAAAAGTCTATCCCAAGAACAAGTTTCGC
CGACACTGAGGTACAAGGGCCATCTTGAGCTTCGAAAAAGTCAATTGCCAGAAAAAGTACTCTTAGTTGACGATGCTGTC
CCAGCGAAATAAAATCAATGGCCCTGTAACTATGTTCTACGAATCTGCTCAATACTTCAGTCCGTACGTAGAAATAAATA
CGACCTTTAGACAGTCCCGGTATACTTGCGGCAAGGTCATCTAACCAGGTCCTTGACAAGGTTGCATCTCTCCGGGAAGG
CTAGGACCAGTCGATATGCTTTCGCGCGTCAGTCGGGGGCTGTCGCCAGACGCCGCGTACAGAATGCCCAGTGAAAACGG
AAAGTATCAAAATTCACCTGTGCGACGCTGTTTTAAACGCTTGTGTTTGCCTTTGACCGTGCCCTCGATAGCAAACCAGG
CATCACCAGATCATCAGGTTCGGTGAAGGAGCGTACAGGGATGGGTAGGAACGATTAATTATTGCGTGCGTTGCTTTAAC
AGACTGATGAAAACCAGTCGTTGGTCGGTTGGCAGGAAGTGACCTAGATGACAAACCTAGCGAGGACCTTCTTGCACCCG
ACCACCACGCCGTTATCCCGGATGTGGTCATCCCGCTACA
在序列中出现最多的片段的出现次数： 18
出现次数最多的片段有：
CAA
```

4. 后缀数组（suffix array，sa）

问题：给定一个DNA序列s，将s的所有后缀按照字典序从小到大排序，按顺序输出每个后缀的起始位置。

解决方法：rank[i]表示以i为起始的后缀的排名，sa[i]表示排名为i的后缀的起始位置在哪里，这两个数组是互逆的关系。采用倍增的方法进行后缀排序：初始时rank[i]表示以i为起始、长度为1的串的排名，由于只有4种字符，所以有大量的名次是重复的。假设当前所有串长为k，而rank数组存放的是串长为k/2时的排名，则比较以i和j为起始两个串的排名时，首先看的是rank[i]和rank[j]，如果rank[i]和rank[j]不相同则名次已经可以比较，如果rank[i]和rank[j]相同则比较rank[i+k/2]和rank[j+k/2]。在倍增过程中维护rank和sa数组，最终可以确定每个后缀的唯一名次。时间复杂度为O(nlogn)。

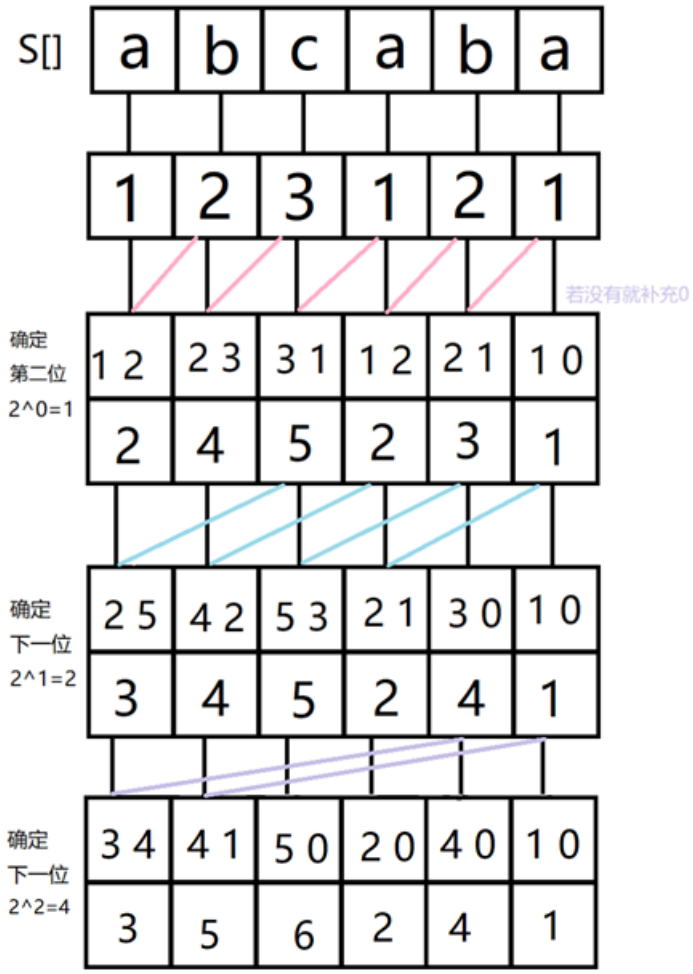

图5 后缀排序流程

```
s=input()
n=len(s)
s=" "+s
m=4
tag={'A':1,'C':2,'G':3,'T':4}
rank=[0]*(n+1)
sa=[0]*(n+1)
for i in range(1,n+1):
    rank[i]=tag[s[i]] #初始时的排名就是字符编号
c=[0]*(n+1)
for i in range(1,n+1):
    c[rank[i]]+=1
for i in range(2,m+1):
    c[i]+=c[i-1]
for i in range(1,n+1):
    sa[c[rank[i]]]=i #根据rank反推出sa数组
    c[rank[i]]-=1

def Y(x):
    if x<0 or x>=len(y):
        return 0
    return y[x]

k=1
while k<=n:
    #已经求得子串长度为k时的rank数组和sa数组，倍增求出子串长度为2k时的结果
```

```
    #基数排序的思想，y数组中是按照将要排序的子串的后半段字典序排列的，这样保证在两个子串前半段相同时可以直接确定顺序
    y=[0]
    for i in range(n-k+1,n+1): #后半段为空的子串，放在y数组的最前面
        y.append(i)
    for i in range(1,n+1):
        if sa[i]>k:
            y.append(sa[i]-k)
    c=[0]*(n+1)
    for i in range(1,n+1):
        c[rank[i]]+=1
    for i in range(2,m+1):
        c[i]+=c[i-1] #统计排名<=i的子串有多少个（因为存在排名重复的子串）
    for i in range(len(y)-1,0,-1): #从后往前遍历y数组，借助刚才统计的c数组，求出子串长度为2k时的sa数组
        sa[c[rank[y[i]]]]=y[i]
        c[rank[y[i]]]-=1
    y=rank.copy()

    #根据sa数组重新算出rank数组
    rank[sa[1]]=m=1
    for i in range(2,n+1):
        if not (Y(sa[i])==Y(sa[i-1]) and Y(sa[i]+k)==Y(sa[i-1]+k)):
            m+=1
        rank[sa[i]]=m
    k*=2
for i in range(1,n+1):
    print(sa[i]-1,end=' ')
```

运行结果：

```
ACGTGTGAACGTGGACT
7 8 0 14 9 1 15 6 13 12 4 10 2 16 5 11 3
```

```
TTGTCCTAGCCGACCCTTGTTTTGGTGTCCCATATATGAATAACTTGACTTCGCCTCCTACTCTAGTATGACATAATGGT
AGAGATCAAACACACTAGGTTTACACGTCACTGTTATCTGTTTAGCACACTCAGCGGCGTGGTTTTGTAAAGCAACACAT
ATTCTTATCTGGGAGTTAGTGAGGCATGTCGCATGGGTTCCCGACCAGGGGGCCAGTATTATGTCCTGTACTCAATGAGC
CCTAGGGTCCTCCTCCCTCCTGCGGTAAGGCCGCGCCAGACGGTTACTGCTTTTGGAATC
87 148 88 153 41 149 266 38 296 233 74 89 154 102 91 126 70 156 203 12 279 104 93
229 128 59 285 109 47 42 277 80 82 150 123 237 7 132 181 267 206 243 96 64 214
177 173 39 72 31 33 158 297 84 166 115 35 67 234 192 75 220 185 217 160 299 86
152 232 90 125 155 103 92 127 108 276 131 205 213 71 30 157 191 184 275 204 212
29 28 199 239 254 13 200 9 270 56 4 240 251 53 248 255 258 224 14 201 10 189 273
51 271 134 262 280 105 137 57 5 241 94 62 230 129 252 54 249 256 60 259 286 168
225 110 117 163 48 43 15 289 37 295 69 202 11 278 46 81 236 180 172 83 151 124
190 183 274 211 238 8 269 52 272 133 261 136 288 294 171 182 210 268 135 170 209
208 207 244 194 263 77 245 23 281 195 97 140 146 264 227 78 65 215 106 26 2 246
222 187 178 138 24 282 174 112 196 98 119 18 141 147 40 265 73 101 228 58 284 79
122 6 242 95 63 176 32 165 114 34 66 219 216 159 298 85 231 107 130 27 198 253 55
3 250 247 257 223 188 50 61 167 116 162 36 68 45 235 179 260 287 293 169 193 76
22 139 145 226 25 1 221 186 111 118 17 100 283 121 175 164 113 218 197 49 161 44
292 21 144 0 16 99 120 291 20 143 290 19 142
```

5. 后缀自动机（suffix automaton, SAM）

问题：在一个初始为空的DNA序列末尾不断添加碱基，每添加1个碱基之后都要给出当前序列中本质不同的DNA片段数量。

解决方法：构建一个后缀自动机，它可以表示出一个字符串的所有子串。后缀自动机上每个点都对应着一个“endpos集合”，包含了一些结束位置相同的子串。这些子串之间是后缀关系，长度是连续的。对于结点i上最短的那个子串，它的后缀只能是位于别的结点（设为j），并且可以证明是j结点上最长的子串。那么，我们将i结点的fa指针指向j结点。每个点记录的len代表该点所有子串的最大长度。

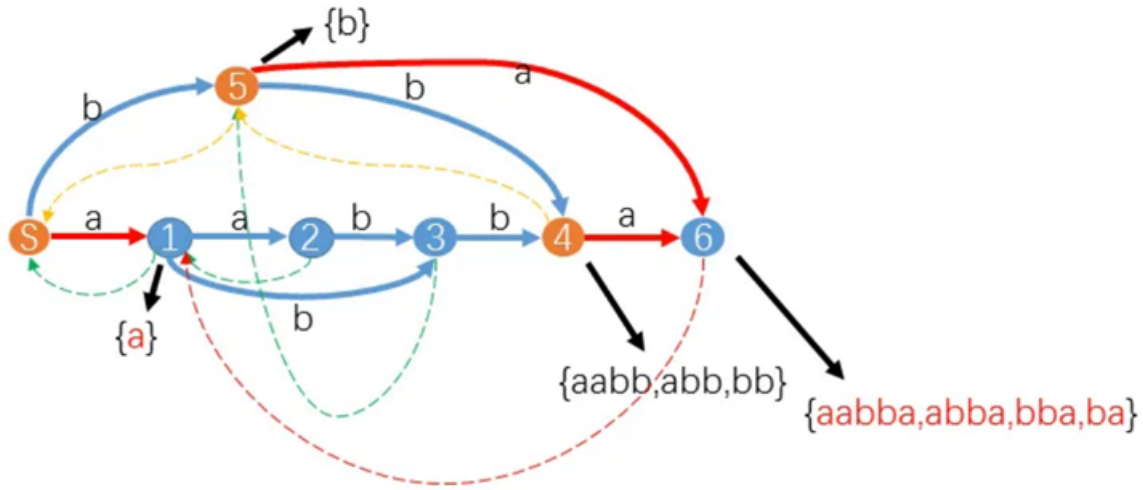

图6 后缀自动机

后缀自动机的构造：每次加入一个字符，在线构造，时间复杂度为O(n)。具体方法见代码：

```
class node:
    def __init__(self):
        self.fa=0
        self.len=0
        self.ch=[0]*4

n=int(input()) #n为添加字符个数
tag={'A':0,'C':1,'G':2,'T':3}
a=[node()]
lst=cnt=1 #lst表示之前最后一个位置的状态
a.append(node())
ans=0
for i in range(n):
    j=tag[input()]
    p=lst
    cnt+=1
    a.append(node()) #加入一个字符，SAM上必定会多出结点
    np=lst=cnt
    a[np].len=a[p].len+1
    while p!=0 and a[p].ch[j]==0:
        a[p].ch[j]=np #p结点上的子串加上字符j可以到达np结点
        p=a[p].fa
    if p==0:
        a[np].fa=1 #没有找到合适的p，np的fa指针指向1（空串）
    else:
        q=a[p].ch[j]
```

```
        if a[q].len==a[p].len+1: #如果q结点的子串只比p结点的最长子串多1个字符，np的fa可以指向它
            a[np].fa=q
        else:
            cnt+=1
            a.append(node())
            nq=cnt #否则需要新建结点nq，使a[nq].len的值为a[p].len+1
            a[nq].fa=a[q].fa
            a[nq].ch=a[q].ch.copy()
            a[nq].len=a[p].len+1
            a[np].fa=a[q].fa=nq #np的fa指向nq，由于a[q].len>a[nq].len，所以q的fa也指向nq
            while p!=0 and a[p].ch[j]==q:
                a[p].ch[j]=nq #把q替换成nq
                p=a[p].fa
    ans+=a[np].len-a[a[np].fa].len #加上np结点对答案的贡献，而额外添加的nq结点对答案没有贡献
    print(ans)
```

运行结果：

```
5
A
1
C
3
C
5
G
9
A
13
```

```
20
A
1
A
2
C
5
T
9
G
14
T
19
A
25
G
32
T
39
C
48
```

```
C
58
C
68
T
79
T
92
G
105
A
120
T
136
T
152
T
169
C
187
```

总结与拓展

运用哈希、KMP、AC自动机等算法处理字符串问题可以避免Brute Force方法造成的大量时间消耗，在可以令人接受的时间内完成任务。本文提到的算法基本上用于字符串的精确匹配，在生物信息学的实际应用中经常出现错配、间隔匹配等情况，这时可以用动态规划等方法计算字符串之间的编辑距离，或者采用一些模糊匹配算法。另外，回文序列有着独特的生物学意义，例如CRISPR中的回文序列可以形成发夹状结构以提高DNA的稳定性，还能识别和剪切入侵病毒的基因组。2014年Mikhail Rubinchik发明的回文自动机（回文树）可以高效解决一系列有关回文字符串的问题(https://codeforces.com/blog/entry/13959)，也许将来在生物信息学中可以得到应用。