字符串算法 part 2[1]

赖金霖[2]
清华大学　计算机科学与技术系

Aug 11, 2019

[1]本文件可在这里找到: https://github.com/lll6924/public_slides
[2]邮箱: laijl16@mails.tsinghua.edu.cn

单词统计问题

给定一个英语句子，统计给定单词的出现次数

样例输入

He will go with her, but he will not stay for long.
he

样例输出

3

规定

- 不区分大小写
- 单词可以出现在另一个单词里
- 单词可以相交（如 aa 在 aaa 里出现了两次）

这类问题通常被称为字符串匹配问题

调库

size_t find (const string& str, size_t pos = 0) const;

```
while((position=s.find(target,position))
                    !=string::npos){
    position++;
    cnt++;
}
```

这个做法的时间复杂度是多少呢?（假设 s 长度为 N，target 长度为 L）

调库

- 设找到的位置分别为 $pos_1, pos_2, ..., pos_M$
- 循环执行次数为 M
- 每次 find 函数的复杂度为 O(($pos_{i+1} - pos_i$)+L)
- 总复杂度为 O(ML+N)
- 由于 $M \leq N$
- 时间复杂度为 O(NL)

调库

- 设找到的位置分别为 $pos_1, pos_2, ..., pos_M$
- 循环执行次数为 M
- 每次 find 函数的复杂度为 O(($pos_{i+1} - pos_i$)+L)
- 总复杂度为 O(ML+N)
- 由于 $M \leq N$
- 时间复杂度为 O(NL)
- 当字符串随机生成时，期望复杂度如何?

调库

- 设找到的位置分别为 $pos_1, pos_2, ..., pos_M$
- 循环执行次数为 M
- 每次 find 函数的复杂度为 O($(pos_{i+1} - pos_i)$+L)
- 总复杂度为 O(ML+N)
- 由于 $M \leq N$
- 时间复杂度为 O(NL)
- 当字符串随机生成时，期望复杂度如何?
- 循环平均执行次数为 O($\frac{N}{26^L}$)
- 时间复杂度 O($\frac{NL}{26^L}$+N)
- 但是出题人显然不会这么善良

依次比较

find 函数是找到“下一个”匹配的单词，而我们的目标是找到“所有”匹配的单词，自己设计算法可能更加高效

一个直接的想法是，枚举匹配的起点，统计匹配的数量

```
a a a b a a a b a a a c
a a a c
  a a a c
    a a a c
      a a a c
        a a a c
          a a a c
            a a a c
              a a a c
                a a a c
```

依次比较

- 很显然，我们要进行 O(N) 次匹配
- 每次匹配的复杂度为 O(L)
- 总时间复杂度 O(NL)
- 最坏复杂度的一个例子是在 aaa....aaa 里找 aa...ab

依次比较

- 很显然，我们要进行 O(N) 次匹配
- 每次匹配的复杂度为 O(L)
- 总时间复杂度 O(NL)
- 最坏复杂度的一个例子是在 aaa....aaa 里找 aa...ab
- 当字符串随机生成时，期望复杂度如何?

依次比较

- 很显然，我们要进行 O(N) 次匹配
- 每次匹配的复杂度为 O(L)
- 总时间复杂度 O(NL)
- 最坏复杂度的一个例子是在 aaa....aaa 里找 aa...ab
- 当字符串随机生成时，期望复杂度如何?
- 我们仍然要进行 O(N) 次匹配
- 每次匹配的平均次数为 $\frac{26}{25}$
- 总时间复杂度 O(N)

字符串 Hash

一个类似的问题是：在一个十进制数字 x 中找到另一个十进制数字的出现次数 y

样例输入

56784321 843

样例输出

1

如果不考虑数据范围，我们很容易能设计如下的算法

- 1. 令 t=x，L 为 y 的位数
- 2. 若 $t\%10^L$==y，计数器加 1
- 3. t=t/10
- 4. 当 t!=0 时，跳到 2
- 5. 输出结果

时间复杂度 O(N)，N 为 x 的位数

字符串 Hash

当 x 和 y 都不能用 long long 存下时，模拟上述过程，复杂度退化成 O(NL)。但是，我们还可以使用 Hash 算法求解

Hash 算法原理：以 a%P 与 b%P 是否相等来代替 a 与 b 是否相等 (P 为一个任取的整数，P<<min(a,b))

- 如果 a==b，Hash 算法结果正确
- 如果 a!=b，Hash 算法只有 $\frac{1}{P}$ 的概率出错
- 如果我们执行 k 次比较，Hash 算法有 $(\frac{P-1}{P})^k$ 的概率正确
- 有 $1-(\frac{P-1}{P})^k$ 的概率出错
- 当 P 很大时，错误率约为 $\frac{k}{P}$

字符串 Hash

以在 56784321 内找 843 为例

假设数据类型只能存下两位整数，取 P=23

$56784321 = 1 * 10^0 + 2 * 10^1 + 3 * 10^2 + 4 * 10^3 + 8 * 10^4 + 7 * 10^5 + 6 * 10^6 + 5 * 10^7$

$843 = 3 * 10^0 + 4 * 10^1 + 8 * 10^2$

- 对于最低位，需要比较 843 和 321
- 对于次低位，需要比较 843 和 432

字符串 Hash

以在 56784321 内找 843 为例

假设数据类型只能存下两位整数，取 P=23

$56784321 = 1 * 10^0 + 2 * 10^1 + 3 * 10^2 + 4 * 10^3 + 8 * 10^4 + 7 * 10^5 + 6 * 10^6 + 5 * 10^7$

$843 = 3 * 10^0 + 4 * 10^1 + 8 * 10^2$

- 对于最低位，需要比较 843 和 321
- 对于次低位，需要比较 843 和 432
- $2 * 10^1 + 3 * 10^2$ 在最低位时已经算过了，我们可以重复利用
- 对于次低位，可以比较 8430 和 4320
- 接下来，我们比较 84300 和 84300

字符串 Hash

我们把 56784321 的各位模 P(P=23) 存在数组 A 中

i	0	1	2	3	4	5	6	7
x	1	2	3	4	8	7	6	5
A=$(x*10^i)$%P	1	20	1	21	6	18	13	6

计算数组 A 的时间复杂度是 O(N) 的

设 B[i]=$10^i\%P$，那么

- 对 i=0,B[i]=1
- 对 i>0,B[i]=(B[i-1]*10)%P
- A[i]=(B[i]*x[i])%P

字符串 Hash

计算 843%P 的时间复杂度是 O(L) 的

843%23=((8*10+4)*10+3)%23=((8*10+4)%21*10+3)%23=15

设 C[i]=(A[i]+A[i+1]+A[i+2])%23，D[i]=(843*10^{i})%23

我们只需要统计 C[i]=D[i] 的数量

i	0	1	2	3	4	5	6	7
A	1	20	1	21	6	18	13	6
C	22	19	5	22	14	14	-	-
D	15	12	5	4	17	9	-	-

字符串 Hash

```
B=1;A[0]=x[0]%P;
for(int i=1;i<N;i++){
      B=(B*10)%P;
      A[i]=(B*x[i])%P;
}
int C=0,D=0,cnt=0;
for(int i=0;i<L;i++){
      D=(D*10+y[L-1-i])%P;
      C+=A[i];
}
for(int i=0;i<=N-L;i++){
      if(C==D)cnt++;
      if(i==N-L)break;
      D=(D*10)%P;
      C=(C-A[i]+A[i+L]+P)%P;
}
```

字符串 Hash

- 这个算法的时间复杂度是 O(N+L) 的
- 错误率是多少呢?

字符串 Hash

- 这个算法的时间复杂度是 O(N+L) 的
- 错误率是多少呢?
- 我们一共进行了 N-L+1 次比较
- 每次比较的错误率为 $\frac{1}{P}$
- 每次都不出错的概率为 $(\frac{P-1}{P})^{N-L+1} \approx 1 - \frac{N-L+1}{P}$
- 在一个典型的 OI 比赛中，$N \le 10^6, L << N$
- 取 $P \approx 10^9$，正确率为 99.9%

字符串 Hash

- 如果嫌正确率太低，我们可以用两个模数 P_1, P_2
- 每次比较的错误率降为 $\frac{1}{P_1 P_2}$
- 总错误率为 $1 - (\frac{P_1 P_2 - 1}{P_1 P_2})^{N-L+1} \approx \frac{N-L+1}{P_1 P_2}$
- 如果采用 K 个模数，时间复杂度和错误率分别是多少?

字符串 Hash

- 如果嫌正确率太低，我们可以用两个模数 P_1,P_2
- 每次比较的错误率降为 $\frac{1}{P_1P_2}$
- 总错误率为 $1-(\frac{P_1P_2-1}{P_1P_2})^{N-L+1}\approx\frac{N-L+1}{P_1P_2}$
- 如果采用 K 个模数，时间复杂度和错误率分别是多少?
- 时间复杂度为 O(K(N+L))，错误率为 $\frac{N-L+1}{\prod_{i=1}^{K}P_i}$
- 在 noip2014 解方程中，K=4 时，错误率为 1%

KMP 算法

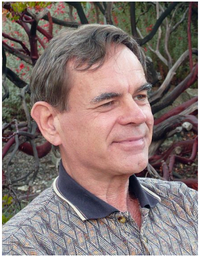
字符串 Hash 有两个缺点：

- 不能做到完全正确
- 大量乘法、取模运算导致速度慢

KMP 算法是匹配问题的通用算法，它的复杂度为 O(N+L)，稳定且常数小

Donald Knuth

James H. Morris



Vaughan Pratt

在下面的部分里，为了方便，下标从 1 开始

KMP 算法

a a a b a a a b a a a c

a a a c
  a a a c
    a a a c
      a a a c
        a a a c
          a a a c
            a a a c
              a a a c
                a a a c

- 绿色部分与原始字符串 x 比较，和与要找的字符串 y 比较是等价的
- 如果我们使用某种方法把绿色部分的比较次数降为线性，总的时间复杂度变为 O(N)

KMP 算法

假如我在 x=ababab 里找 y=abac：

a b a b a b
a b a c
  a b a c
    a b a c

- 在第一个位置没有找到，在第二个位置会找到吗?
- 我们只考虑要找的字符串 abac，沿用上一页的思路，在第二个位置，首先要比较'ab' 和'ba'，即 y[1..2] 和 y[2..3]
- 显然，这两个字符串不相等，所以第二个位置不会找到

KMP 算法

假如我在 x=ababab 里找 y=abac：

a b a b a b
a b a c
  a b a c
    a b a c

- 在第一个位置没有找到，在第二个位置会找到吗?
- 我们只考虑要找的字符串 abac，沿用上一页的思路，在第二个位置，首先要比较'ab' 和'ba'，即 y[1..2] 和 y[2..3]
- 显然，这两个字符串不相等，所以第二个位置不会找到
- 第三个位置会找到吗?
- 如果我们有“上帝视角”，能看出不会找到
- 但当程序处理到这里时，他会首先比较 y[1] 和 y[3]，发现 y[1] 和 y[3] 相等，再继续循环下去

KMP 算法

- 总结上述规律，我们可以得到一个思路
- 假设我们在 x 的第一个位置没有找到目标字符串 y，但是发现 x[1..k]=y[1..k]
- 在第二个位置，我们需要比较 x[2..k] 和 y[1..k-1]，等价于比较 y[2..k] 和 y[1..k-1]
- 在第三个位置，我们需要比较 y[1..k-2] 和 y[3..k]
- 在第 T 个位置，我们需要比较 y[1..k-T+1] 和 y[T..k]
- 如果我们能通过预处理找到最小的 T，使得 y[1..k-T+1]=y[T..k]，那么是否可以加速算法呢?
- 容易发现，在其他位置的算法和在第一个位置一样

KMP 算法

- 形式上理解，我们需要对每个 k 找到最大的 B，使得 y[1..k] 的后 B 个字符组成的字符串和前 B 个字符组成的字符串完全相同。不妨设 next[k] 存储这样的数字
- 匹配失败时，可以通过 next 数组寻找下一个匹配的位置
- 如果我们在 x[i+1] 处发现错误，且错误是 x[i+1]!=y[j+1]

KMP 算法

- 形式上理解，我们需要对每个 k 找到最大的 B，使得 y[1..k] 的后 B 个字符组成的字符串和前 B 个字符组成的字符串完全相同。不妨设 next[k] 存储这样的数字
- 匹配失败时，可以通过 next 数组寻找下一个匹配的位置
- 如果我们在 x[i+1] 处发现错误，且错误是 x[i+1]!=y[j+1]
- 我们知道 y[1..j] 已经匹配上了，我们需要移动 y 字符串使得 y 的前 next[j] 个和当前的后 next[j] 个相对
- 然后，我们比较 x[i+1] 和 y[next[j]+1]，如果不相等，要怎么做?

KMP 算法

- 形式上理解，我们需要对每个 k 找到最大的 B，使得 y[1..k] 的后 B 个字符组成的字符串和前 B 个字符组成的字符串完全相同。不妨设 next[k] 存储这样的数字
- 匹配失败时，可以通过 next 数组寻找下一个匹配的位置
- 如果我们在 $x[i+1]$ 处发现错误，且错误是 $x[i+1] != y[j+1]$
- 我们知道 y[1..j] 已经匹配上了，我们需要移动 y 字符串使得 y 的前 next[j] 个和当前的后 next[j] 个相对
- 然后，我们比较 $x[i+1]$ 和 $y[next[j]+1]$，如果不相等，要怎么做?
- 我们可以接着移动 y 字符串，比较 $x[i+1]$ 和 $y[next[next[j]]+1]$
- 直到匹配上，我们才可以让 i 加 1

KMP 算法

- 注意到，上述描述实际上是一个嵌套循环：
- j=0
- for i in 1..n:
-     while(j!=0&&y[j+1]!=x[i]):
-         j=next[j]
-     end while
-     if(y[j+1]==x[i]):
-         j=j+1
-     end if
-     if(j==l):
-         cnt=cnt+1
-         j=next[j]
-     end if
- end for

KMP 算法

- 上述算法的时间复杂度是多少?

KMP 算法

- 上述算法的时间复杂度是多少?
- 外层循环是 O(N) 的,while 循环总共会执行几次?

KMP 算法

- 上述算法的时间复杂度是多少?
- 外层循环是 O(N) 的,while 循环总共会执行几次?
- 注意到，每执行一次 while 循环，j 都会下降
- 每次外层循环里 j 最多加 1
- j 下降的次数不会超过 j 上升的次数
- while 循环也是 O(N) 的，总时间复杂度 O(N)
- 现在的关键是，如何在 y 上高效计算出 next 数组

KMP 算法

- 一个很直接的想法是枚举 + 字符串 Hash
- 枚举匹配长度 B，使用前缀和数组分别计算 y[1..B] 和 y[k-B+1..k] 的 Hash 值 C 和 D
- C 乘上 26^{k-B}，和 D 对比
- 总时间复杂度 O(L^2)
- 由于 next 数组的错误对答案的影响未知，故不分析错误概率
- 评价：这个算法没有利用 next[k] 和 next[k-1] 之间的联系

KMP 算法

- 如果 y[next[k-1]+1]==y[k]，那么 next[k]=next[k-1]+1
- 为什么此时 next[k] 不会大于 next[k-1]+1?
- 如果 y[next[k-1]+1]!=y[k]，要怎么算?

KMP 算法

- 如果 y[next[k-1]+1]==y[k]，那么 next[k]=next[k-1]+1
- 为什么此时 next[k] 不会大于 next[k-1]+1?
- 如果 y[next[k-1]+1]!=y[k]，要怎么算?
- 我们需要找到下一个 B'，满足 y[1..k] 的后 B' 个和前 B' 个相同
- 一个直接的想法是 next[next[k-1]]
- 为什么不会存在 next[next[k-1]]<next'<next[k-1] 的答案?

KMP 算法

- 如果 y[next[k-1]+1]==y[k]，那么 next[k]=next[k-1]+1
- 为什么此时 next[k] 不会大于 next[k-1]+1?
- 如果 y[next[k-1]+1]!=y[k]，要怎么算?
- 我们需要找到下一个 B'，满足 y[1..k] 的后 B' 个和前 B' 个相同
- 一个直接的想法是 next[next[k-1]]
- 为什么不会存在 next[next[k-1]]<next'<next[k-1] 的答案?
- 当 y[next[next[k-1]]+1]==y[k] 时，那么 next[k]=next[next[k-1]]+1
- 以此类推

KMP 算法

- 可以写出以下的伪代码
- j=next[1]=0
- for i in 2..l:
-     while(j!=0&&y[j+1]!=y[i])
-         j=next[j]
-     end while
-     if(y[j+1]==y[i])
-         j=j+1
-     end if
-     next[i]=j
- end for

KMP 算法

- j 实际上代表了 next[1]..next[l] 的“运动轨迹”
- 这段代码和匹配部分的代码很像!
- 如何分析时间复杂度?

KMP 算法

- j 实际上代表了 next[1]..next[l] 的“运动轨迹”
- 这段代码和匹配部分的代码很像!
- 如何分析时间复杂度?
- 通过分析 j 的变动，容易看出时间复杂度为 O(L)
- 所以 KMP 算法的时间复杂度为 O(N+L)

KMP 算法

- j 实际上代表了 next[1]..next[l] 的 “运动轨迹”
- 这段代码和匹配部分的代码很像!
- 如何分析时间复杂度?
- 通过分析 j 的变动，容易看出时间复杂度为 O(L)
- 所以 KMP 算法的时间复杂度为 O(N+L)
- 让我们来算一个例子
- 在 ababababac 里找 ababac

KMP 算法

我们首先计算 next 数组

i	1	2	3	4	5	6
y	a	b	a	b	a	c
next	0	0	1	2	3	0

next[3] 是怎么算的?

next[5] 是怎么算的?

next[6] 是怎么算的?

KMP 算法

我们接着寻找匹配

i	1	2	3	4	5	6
y	a	b	a	b	a	c
next	0	0	1	2	3	0

i	1	2	3	4	5	6	7	8	9	10
y	a	b	a	b	a	b	a	b	a	c
j	1	2	3	4	5	4	5	4	5	6

i=1,4,6,10 时，j 分别是如何计算的?

例题讲解-UVA1328 Period

- 对于一个字符串 S
- 求它的每个前缀是否是周期串
- 周期串的定义是某一字符串重复连接而成（至少重复 2 次）
- 对于是周期串的前缀，输出最多循环次数

样例输入

3
aaa
12
aabaabaabaab
0

样例输出

Test case #1
2 2
3 3
Test case #2
2 2
6 2
9 3
12 4

例题讲解-UVA1328 Period

- 我们可以枚举循环节长度 a，比较 x[1..a]、x[a+1..2a]、...
- 然后更新 a、2a、... 处的循环次数
- 最坏复杂度 $O(N^2)$，当字符串里都是同一个字符时最坏

例题讲解-UVA1328 Period

- 我们可以枚举循环节长度 a，比较 x[1..a]、x[a+1..2a]、...
- 然后更新 a、2a、... 处的循环次数
- 最坏复杂度 $O(N^2)$，当字符串里都是同一个字符时最坏
- 如果比较使用字符串 Hash，复杂度降为 O(NlogN)
- $\frac{1}{1} + \frac{1}{2} + ... + \frac{1}{X} \approx lnX$
- 平均复杂度是 O(N)，为什么？
- 评价：这个算法没有完全利用子串之间的关系

例题讲解-UVA1328 Period

- 考虑 S 的一个前缀，题目要比较 S 的一些相邻子串
- 很自然地，如果 S 是长度为 t 的字符串重复 k 次得到的，那么 S 的 next 数组会是什么样?
- next[tk]=t(k-1)
- 反过来想，如果 next[tk]=t(k-1)，那么 S 的前 tk 个字符串会是什么样?

例题讲解-UVA1328 Period

- 考虑 S 的一个前缀，题目要比较 S 的一些相邻子串
- 很自然地，如果 S 是长度为 t 的字符串重复 k 次得到的，那么 S 的 next 数组会是什么样?
- next[tk]=t(k-1)
- 反过来想，如果 next[tk]=t(k-1)，那么 S 的前 tk 个字符串会是什么样?
- 设 T=S[1..tk]
- T 的前 t(k-1) 个字符和后 t(k-1) 个字符一样

例题讲解-UVA1328 Period

- 考虑 S 的一个前缀，题目要比较 S 的一些相邻子串
- 很自然地，如果 S 是长度为 t 的字符串重复 k 次得到的，那么 S 的 next 数组会是什么样?
- next[tk]=t(k-1)
- 反过来想，如果 next[tk]=t(k-1)，那么 S 的前 tk 个字符串会是什么样?
- 设 T=S[1..tk]
- T 的前 t(k-1) 个字符和后 t(k-1) 个字符一样
- 所以 T[1..t]==T[t+1..2t]==...==T[t(k-1)+1..tk]
- 所以 T 是长度为 t 的字符串重复 k 次得到的

例题讲解-UVA1328 Period

- next[tk]=t(k-1) 隐含了什么条件?
- tk 是 tk-t(k-1) 的倍数!
- 对于所有 j，计算 t=j-next[j]
- 如果 t 能整除 j，那么说明 S[1..j] 是长度为 t 的字符串重复而成
- 重复次数为 $\frac{j}{t}$
- 我们只需要计算 S 的 next 数组，总时间复杂度 O(N)

例题讲解-[NOI2014] 动物园

- （如果你没有完全听懂 KMP，可以找这题的题面再理解一下）
- 给定字符串 S，设 num[i] 表示 S[1..i] 里，满足前 T 个字符和后 T 个字符相同且不相交的 T 的数量
- 对所有 i，求 num[i]+1 的积模 1,000,000,007 的值

样例输入

```
3
aaaaa
ab
abcababc
```

样例输出

```
36
1
32
```

例题讲解-[NOI2014] 动物园

- 注意到我们有两个条件：前 T 个字符和后 T 个字符相同；这两部分不相交
- 去掉第一个条件，题目就变简单了
- 去掉第二个条件，可以怎么做?

例题讲解-[NOI2014] 动物园

- 注意到我们有两个条件：前 T 个字符和后 T 个字符相同；这两部分不相交
- 去掉第一个条件，题目就变简单了
- 去掉第二个条件，可以怎么做?
- 在计算 next 数组时，我们可以维护另一个数组 num_0，并令 $num_0[i] = num_0[next[i]] + 1$
- num_0 就是我们去掉第二个条件的 num 数组，为什么？

例题讲解-[NOI2014] 动物园

- 注意到我们有两个条件：前 T 个字符和后 T 个字符相同；这两部分不相交
- 去掉第一个条件，题目就变简单了
- 去掉第二个条件，可以怎么做?
- 在计算 next 数组时，我们可以维护另一个数组 num_0，并令 $num_0[i] = num_0[next[i]] + 1$
- num_0 就是我们去掉第二个条件的 num 数组，为什么?
- 加上第二个条件，我们可以再像计算 next 数组时那样，执行一个循环
- 这时我们的 j 需要满足题目所给的条件，可以用另一个 while 循环降低 j 使其始终满足条件
- 递推仍然是合法的，在 i=t 时满足条件的 j 一定从 i=t-1 处满足条件的 j 递推而得
- 那么 $num[i] = num_0[j] + 1$

更多内容

- KMP 算法的内容略高于 NOIP 提高组要求，但算法思想十分重要
- KMP 是应用最广的 OI 算法之一
- 复杂度推导和错误率分析是设计算法的基础
- 几个提及的替代算法，如字符串 Hash，都是 NOIP 难度的
- 在实现 KMP 算法的过程中，思考边界情况和数值加减都对水平提升有益
- 感兴趣的同学可以进一步了解扩展 KMP 算法

Thanks for listening

Any Questions?