字串 Strings

by DarkBtf

I.概述 Introduction

顧名思義，字串就是把一堆無辜的字元，排成一列，然後做成的串燒。

字串是一種很特殊的資料結構，關於字串的處理，也衍生出許多資料結構以及演算法，在這裡先介紹一些定義：

1. 字元(Charater)

一個獨立的符號，例如：'A','B','C','1','2','3','金','坷','垃'...

2. 字元集(Alphabet)

由字元所組成的集合，習慣上用 Σ 表示。

3. 字串(String)

由字元集中的字元構成的序列。字串 S 的第 i 個字元用 Si 或 S[i]表示。例："tentacle"

4. 子字串(Substring)

字串中一段連續的字元組成的字串。例如："tent"是"tentacle"的子字串。

5. 前綴(Prefix)、後綴(Suffix)

字串的前 k 個字元組成的子字串稱為該子串的 k-前綴;反之，

由後 k 個字元組成的的子字串稱為 k-後綴。例如：”tent”是”tentacle”的前綴，而”acle”是”tentacle”的後綴。

一個顯然但值得一提的性質：「字串的每個子字串，都是它的某個前綴的後綴，也是後綴的前綴。」

5. 子序列(Subsequence)

任意刪去字串中一些字元產生的序列，與子字串不同的是，子字串一定是由原字串中連續的一段組成。例如：”tale”是”tentacle”的子序列，但不是子字串。

6. 字典序(Lexicographical Order)

用來定義字串間的大小順序，首先要定義出字元間的大小關係，例如： ‘a’ < ‘b’<‘c’< …… <‘z’。然後對於兩個字串比較時，從兩個字串開頭逐字元做比對，如有不同，那麼兩字串的大小關係就是該位字元的大小關係。若是還未比較出大小，其中一字串就到達末端了，則以短的那個字串為小。

II.匹配問題 String Matching

字串的匹配是字串的核心問題，定義上，就是判斷一個模版字串 T 是不是主字串的子字串並找出子字串的位置。關於字串匹配有很多很多的算法(多得令人髮指)，如果想要做深入研究可以參考這個網站：

http://www-igm.univ-mlv.fr/~lecroq/string/

1. 樸素算法(Naïve String-Matching Algorithm)

很直覺的，拿模版字串去從主字串的頭開始對齊比對，發現錯誤就往右移一位再試一次，如果出現了最糟糕的情形，像是：「"IIIIIZZ"匹配"JIIIIIIIIIIIIIIIIIIIIIIIIZZ"」，就會浪費掉很多寶貴的時間。最差複雜度是 O(m(n-m+1)){m 是模版字串長，n 是主字串長}。看似暴力，但是在隨機字串的情況下，這個算法是線性的！所以在測資沒有刻意刁難的情況上，不失為一種實用的演算法。

```
Function Naïve Matching(String S, String T)
N ← length(S)
M ← length(T)
for i ← 1 to N-M+1
    do if T[1...M] = S[i...i+M-1]
          then print "Found."
```

2. Rabin-Karp 演算法(RK Algorithm)

想一下，對於比較兩字串是否相同這個問題，如果把一個字串轉換成一個數字，那麼就可以直接 O(1)比較了！當然，事情沒有這麼簡單，字串有千百萬種組合，要把每個可能的字串都轉成一對一對應的數字，那數值的範圍可不是一般的變數負擔的起的。所以我們引入 hash 的想法，直接比較 mod 某個很大的數的值即可。

```
Function Rabin-Karp Matching(String S, String T)
N ← length(S)
M ← length(T)
p ← 0
q ← 0
k ← a big number
▷ pre-processing:
for i ← 1 to M
   do p ← ( p × |Σ| + T[i] ) mod k
for i ← 1 to M
   do q ← ( q × |Σ| + S[i] ) mod k
for i ← 1 to N-M+1
   do if p = q
         then print "Found."
      q ← ( (q − S[i] × |Σ|^(M-1)) × |Σ| + S[i+M] ) mod k
```

不過這樣的做法並不保證正確性(有可能不同字串取餘出一樣的值)，這時解決方法有二，第一種方法是當 key 值相等時，逐字比對確認，但會讓複雜度上升到跟樸素算法一樣的量級；第二種方法則是取多個數的餘數，來降低錯誤的機率。

3. Knuth-Morris-Pratt 演算法(KMP Algorithm)

看看這個例子，用"ABZABC"匹配"AABZABZABCZ"：

S： A [A B Z A B] Z A B C Z

T： [A B Z A B] C

↑在這位出現了錯誤

以樸素算法的做法，會往右位移一格，然後再從頭開始檢查，而底線部份會再被重覆檢查過，而既然我們已經檢查過這段了，那能否利用

已經檢查過的這段所得的資訊，來避免不必要的重覆檢查呢？

由[ ]包起來的是已經匹配好的段落 ABZAB，我們已經知道 S[2~6]=T[1~5]了，我們希望利用這五個已匹配字元的資訊來找到第一個可能產生的位移。

可以發現，第一個可能的位移是

S： A A B Z A B Z A B C Z

T： A B Z A B C

對於 T 的所有前綴，可以用預處理的方式求出次長的共同前後綴，當在匹配 S 的時候就可以在錯誤發生的時候，直接查詢表格來產生位移了！

定義一個函數 π[i]，當匹配模版的第 i 位跟主字串的第 j 位產生錯誤時，將模版的第 π[i]位對齊主字串的第 j 位繼續進行匹配。

至於預處理的方式可以用類似模版自己對自己做 KMP 的方式來求出位移值。可利用已知的 π[1]~π[i]之值，求出π[i+1]。

如果 T[π[i]+1]=T[i+1]：π[i+1]= π[i]+1

若不等，就繼續檢查

如果 T[π[π[i]]+1]=T[i+1]→π[i+1]=π[π[i]]+1

如果 T[π[π[π[i]]]+1]=T[i+1]→π[i+1]=π[π[π[i]]]+1

.....

不管是預處理還是匹配主字串部份，看似最壞情況下是在匹配每個字元時都檢查了 O(length(T)) 次，但事實上，由於當前 i 值每次最多增加 1，每次的迭代至少使 i 值減少 1，所以均攤來看，KMP 算法是線性的！O(length(S)+length(T))

這裡有做得很漂亮的動畫演示 KMP 算法：

http://www.csie.ntnu.edu.tw/~u91029/StringMatching.html#a3

附上 psuedo-code：

```
Function Pre-process(String T)
m ← length(T)
π[1] ← 0
k ← 0
for i ← 2 to m
    do while k > 0 and T[k + 1] ≠ T[i]
        do k ← π[k]
    if T[k + 1] = T[i]
    then k ← k + 1
    π[i] ← k
return π
```

```
Function KMP-MATCHER(String S, String T)
n ← length[S]
m ← length[T]
π ← COMPUTE-PREFIX-FUNCTION(T)
k ← 0
for i ← 1 to n
    do while k > 0 and T[k + 1] ≠ S[i]
        do k ← π[k]
    if T[k + 1] = S[i]
    then k ← k + 1
    if k = m
```

```
    then printf "found"
    k ← π[k]
```

4.結語

對自己人品有信心的人可以盡可能的使用不處理碰撞的 RK 算法來檢驗自己的人品值，也可以多玩玩食人魔法師或渾沌騎士這樣的角色。

關於 KMP 演算法，有很多有趣的應用，其中之一是求一字串所有後綴跟該字串的最長共同前綴，可以自己動動腦筋想想看。

另外還有處理多模版字串匹配的這類問題，可以利用後綴樹、後綴數組或是 AC(!)自動機這些資料結構或演算法來處理。

字串演算法勃大莖深，而匹配算法是處理複雜字串問題常用到的基本工具，總之多做題目，你會覺得這個世界真是奇妙！

III.資料結構 Data Structures

1.字典樹 Trie

字典樹是一棵|Σ|元有根樹，除了根結點之外每個點都代表一個字母，而每個結點至多有|Σ|個子結點，分別代表|Σ|個後繼字母。

最簡單的實做是對每個結點開一條長|Σ|的指標陣列，記錄所有後繼的子結點，還有一個布林變數記錄這個結點是否代表一個字。

查詢跟插入都可以自己想(被打)，反正就是 DFS 就行了！

關於字典樹的優化，由於大部份情況會有很多空指標造成記憶體的

浪費，所以可改用左子右兄弟法來建樹以節省記憶體。

2.後綴數組(後綴陣列) Suffix Array

先講一下後綴樹的定義，後綴樹是一棵由一個字串所有後綴構成的字典樹，顯然地如果要進行多模版匹配時，由於主字串的每個子字串都是其後綴的前綴，所以只要查詢該模板字串是否在後綴樹上即可。

後綴樹的構造可以做到時間複雜度 O(n)，空間複雜度 O(n)，但是由於編程太過麻煩，在競賽上通常使用後綴數組做代替。

後綴數組(SA[])是 0~N-1 的一個排列，滿足 SA[0]-後綴 < SA[1]-後綴 < …… < SA[N-1]-後綴。以 ababcad 為例，它的後綴數組就是 0,2,5,1,3,4,6。

利用後綴數組，就可以用基於二分搜尋的概念，在對數時間匹配一個模版字串，也可以進一步用來求出任兩後綴的 LCP(Longest Common Prefix, 最長共同前綴)。

關於構造的方法，最直覺的方法是把所有後綴拿去排序，複雜度是排序的複雜度乘上字串間比較的複雜度，很慢。

這裡介紹一個 O(nlgn)的倍增構造法，首先定義一個名次數組(rank[])用來表示每個後綴的大小次序，同時他也是 SA[]的反函數(rank[SA[k]]=k)。然後，再定義一種 k-前綴比較關係：

$u <_k v \quad \Leftrightarrow \quad u[1...k] < v[1...k]$

$u >_k v \quad \Leftrightarrow \quad u[1...k] > v[1...k]$

$u =_k v \quad \Leftrightarrow \quad u[1...k] = v[1...k]$

總之就是只比較兩字串的前 k 個字元。

同時再定義，k-SA[]以及 k-Rank[]，用來表示只把前 k 個字元拿來比較排序產生的 SA[]和 Rank[]。

最後，可以顯然地推出這些關係：

1.若 $k \geq n$，$suffix(i) >_k (=_k, <_k)\ suffix(j) \Leftrightarrow suffix(i) > (=, <)\ suffix(j)$

2.$suffix(i) =_{2k} suffix(j) \Leftrightarrow$

$suffix(i) =_k suffix(j)$ 且 $suffix(i+k) =_k suffix(j+k)$

3. $suffix(i) <_{2k} suffix(j) \Leftrightarrow (\ suffix(i) <_k suffix(j)\ )$ 或

$suffix(i) =_k suffix(j)$ 且 $suffix(i+k) <_k suffix(j+k)$

其實 $suffix(i) >_k (=_k, <_k)\ suffix(j)$，就代表 k-Rank[i]>(=,<)k-Rank[j]！

於是，在已知 k-Rank[]的情況下，要確認任意兩個後綴的 2k 前綴比較關係只需要至多兩次的比較，就可以在 O(1) x O(sorting)的複雜度來推出 2k-SA[]了。一般而言直接使用 quick sort 不會太慢，不過由於 Rank[]都是整數，所以可以利用 counting sort 把每次遞推的複雜度降到 O(n)！

所以利用這些性質，由排序字元求出 1-SA[]開始，可以從 k-SA[]

推出 k-Rank[]，再由 k-Rank[]來推出 2k-SA[]！當 2k ≥ n 時，我們就構造出了整個後綴數組了！總複雜度是 O(nlgn)(使用 counting sort)

關於後綴數組還有一個複雜度 O(n)的算法，DC3 算法，不過編程複雜度高，常數也頗大，在 n 小時不會比倍增法快多少，有興趣自虐的人可以咕狗一下。

字串題單：

POI 12th Template, 13th Periods of Words, Palindromes, 17th Hamsters

UVa 10298 11475 10679 475 11019 10526 10580 10829 11107 11512 902 10226 10391 10745 10945 10453

TIOJ 1306 1321 1497 1502 1515 1531