字串

Yihda Yol

2016 年 9 月 26 日

1 字串簡介

字串（string）是由有限個字元所構成的序列。而「字元」狹義來說，是指字母或文字的最小單位；廣義來說，可以是一個任意的非空有限集合的元素，這個集合稱為「字元集」或「字母表」。

例如，`algorithm` 是一個字串，它的字元集是英文字母（或以 ASCII 編碼來說是一個 8-bit 整數）。

看起來「字串」和「序列」似乎沒有兩樣，但一般談「字串」時，通常會建構在「字元前後連續」的基礎之上。因此若一個演算法或題目沒有用到「連續」的這個特性（例如最長公共子序列（LCS）問題），那麼我們就不將其歸類在「字串」的範疇內。

字串的術語不多，而且大多簡單易懂，但是有時在敘述或構造演算法的時候會全部混在一起，導致思考混亂。以下列舉常見的術語，最好是將它們全部弄熟：

1. 字元（character）：構成字串的單位。以下「字串 A 的第 i 個字元」將簡寫為 A_i。

2. 字元集（alphabet）：所有可能字元的集合，記為 Σ，其集合大小為 $|\Sigma|$。

3. 長度（length）：一個字串有幾個字元。以下「字串 A 的長度」將簡寫為 L_A。

4. 空字串（empty string）：長度為零的字串，記為 ϵ。

5. 子字串（substring）：一個字串取一段連續的字元所構成的字串稱為子字串。以下將「字串 A 的第 $[i,j)$ 個字元所構成的子字串」（注意是左閉右開）簡寫為 $A_{i\cdots j}$。可以注意到，$A_{i\cdots i}$ 是空字串。

6. 前綴、後綴（prefix, suffix）：一個字串只取最前面一些字元所構成的子字串是前綴，只取最後面的則是後綴。前綴可以寫成 $A_{0\cdots i}$、後綴則可寫成 $A_{i\cdots n}$。

7. 串接（concatenation）：把兩個字串或字元首尾相接變成新字串的操作，以下將簡記為「+」。例如 `b+bc=bbc`。

1.1 儲存

儲存一個字串最直觀的是用類似陣列的資料結構（如 char[]、std::string），用起來也都很方便。

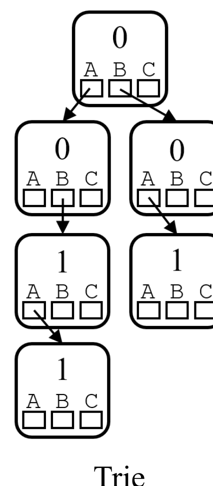
如果要儲存很多個字串（或稱為「字典」），除了弄一個字串的陣列或二元搜尋樹以外，還有一個稱為「字典樹」（trie）的方法。

字典樹，顧名思義，其結構是一棵樹，而使用的方法則和字典差不多。

字典樹每一個節點都包含一個標記，和 $|\Sigma|$ 個指標（分別代表每一種字元）。根節點代表的是空字串，其餘每個節點代表的是「父節點代表的字串」+「該指標所代表的字元」；標記則是指示字典中是否存在這個節點所代表的字串（或有幾個這樣的字串）。

例如右圖字典樹的字元集為 {A, B, C}，所儲存的字串為 AB、ABA 和 BA。

其空間複雜度為 $O(|\Sigma| \times \sum L)$，增加、刪除字串時間複雜度為 $O(L)$。



Trie

2 字串匹配

所謂字串匹配，就是給定兩個字串 A 和 B，求是否存在 A 的子字串 $A_{i\cdots j} = B$。

可以很簡單地得到 $O(L_A L_B)$ 的演算法，但是這個複雜度看起來亟需改進。因此，有以下數個演算法：

2.1 雜湊

所謂「雜湊」（hash），即是一種分類的法則：把所有有可能出現的字串分成 N 類。

不妨將其視作一個函數，稱作雜湊函數（hash function）$h(A)$，滿足 $\forall A \in \Sigma$，有 $h(A) \in \mathbb{Z}$ 且 $h(A) \in [0, M)$。也就是說，$h(A)$ 代表的是 A 這個字串屬於第幾類。

顯然對於兩字串 A, B，$h(A) \neq h(B) \Rightarrow A \neq B$。但是我們可以「相信」反過來的情況在通常時會是對的（一個拚人品的概念）。

至於要選用什麼樣的雜湊函數呢？既然要用作字串匹配，自然會希望除了原字串的雜湊函數以外，其子字串的雜湊值也要能快速計算。因此，這裡使用一個稱作 rolling hash 的技巧（有時稱作 Rabin fingerprint）。我們定義一個字串的雜湊函數：

$$\begin{aligned} f(A) &= \sum_{i=0}^{L_A-1} A_i p^{L_A-i-1} \mod M \\ &= A_0 p^{L_A-1} + A_1 p^{L_A-2} + \cdots + A_{L_A-1} \mod M \ ; p \geq |\Sigma|, p \in \mathbb{N} \end{aligned}$$

其意義就是字串 A 在 p 進位表示法之下模 M 的結果。可以發現，在計算 $f(A)$ 的過程中，也會順便得到 $f(A_{0\cdots i})$ 的值，也就是 A 所有前綴的雜湊函數值。另外還可以得到：

$$f(A_{i\cdots j}) \equiv f(A_{0\cdots j}) - p^{j-i} f(A_{0\cdots i}) \pmod M$$

因此，在 $O(L_A)$ 的預處理之後，可以用 $O(1)$ 時間得到一個子字串的雜湊函數值，也因此得以在 $O(L_A + L_B)$ 的時間內完成「近似字串匹配」。

至於兩個字串 $A \neq B$ 但 $f(A) = f(B)$（稱為「碰撞」）發生時會讓實際上沒有匹配的字串被判定為有匹配，其機率在隨機測資的情況下約為 $1/M$。但是如果 p, M 取得不好，可能會讓機率大幅上升，因此通常 p, M 都取兩相異質數。

另外，可以用不同的 p, M 重複做兩三次相同的事情，會大幅降低碰撞機率。

雜湊不只可以應用在字串匹配上，不少字串問題都可以用雜湊解決，效率也不一定比其它確定性演算法差。但有時需要視題目構造所需的雜湊函數。

2.2 Knuth-Morris-Pratt Algorithm (KMP)

然而模運算的常數本來就比較大。更何況，hash 終究只是個機率演算法，永遠有出錯的可能性。幸好有個確定的演算法可以達到相同的複雜度，我們稱為 KMP 演算法。

仔細觀察一下 $O(L_A L_B)$ 的演算法當中的流程：把兩個字串 A, B 的頭對齊，開始一個一個匹配，如果發生對應不相等的情況，則將 B 往後移動一格，從頭開始繼續匹配。如右圖舉例 AABAABAAC 和 AABAAC 的匹配過程：

```
  012345678
  AABAAAAAC
0 AABAAC
1  AABAAC
2   AABAAC
3    AABAAC
```

粗體字指的是匹配失敗發生的位置。我們發現，1、2 兩次匹配其實是多餘的，原因是在第 0 次匹配的時候，我們已經知道 $A_{0\cdots 5} = B_{0\cdots 5}$，如果善加利用這個性質，便可以預知 1、2 兩次必定會匹配失敗。也就是在匹配到 B_5 失敗的時候，應該要能得到「接下來可以直接把 B 往後挪三格」。

那要如何知道這件事呢？已完成的匹配結果是 $A_{i-j\cdots i} = B_{0\cdots j}$，如果可以判定 $A_{i-p\cdots i} (= B_{j-p\cdots j}) \neq B_{0\cdots p}$，那就可以跳過這一次的匹配。

也就是說，我們要對 B 的每個前綴找到一個**最大**且 $< j$ 的 p，使得 $B_{0\cdots p} = B_{j-p\cdots j}$，那麼當匹配到 $A_i \neq B_j$ 時（失敗），就可以直接將 B 往右挪動 $j - p$ 格，而且因為已經知道 $B_{j-p\cdots p} = A_{i-p\cdots i}$，不需要從頭開始匹配，而可以直接從 A_i 和 B_p 開始匹配。特別地，如果在 B_0 時就匹配失敗，那麼沒有任何可用的資訊，此時就只能將 B 右移一格從頭匹配。

由於對於每個 $j < L_B$，都有唯一的 p 值，我們不妨將其定義為一個函數 $F(j)$。由於這個函數在「匹配失敗」的時候會用到，因此將其稱之為「失敗函數」（failure function）。定義如下：

$$F_B(j) = \begin{cases} -1, & \text{if } j = 0 \\ \max\{p : B_{0\cdots p} = B_{j-p\cdots j} \text{ and } 0 \leq p < j\}, & \text{otherwise} \end{cases}$$

若已經知道所有 $F(j)$ 值，匹配複雜度是多少呢？在匹配過程中，只有兩種情形：

1. $A_i = B_j$ 或 $B_j = -1$，此時 i, j 皆加一。

2. $A_i \neq B_j$ 且 p，此時 i 不變，j 變成 $F(j)$。

顯然 1. 最多只會發生 $L_A - 1$ 次；由於 $F(j) < j$，且 $j \geq -1$，所以 2. 發生的次數不多於 1. 發生的次數，故這部分複雜度是 $O(L_A)$。

剩下的問題就是要如何計算 $F(j)$ 的值。我們可以使用類似 DP 的方法，利用現有的失敗函數值計算其它的值。

顯然地，$F(1) = 0$。而假設已知 $F(k)$ $(\forall k < i)$ 的值，而現在要求 $F(i)$。我們令 $j = F(i-1)$，如果 $B_j = B_i$，因為已經有 $B_{0 \cdots j} = B_{i-j-1 \cdots i-1}$，因此 $B_{0 \cdots j+1} = B_{i-(j+1) \cdots i}$。因為 $j = F(i-1)$ 的定義是取最大值，所以 $F(i) = j + 1$。

而如果 $B_j \neq B_i$，由失敗函數的定義告訴我們 $B_{0 \cdots F(j)} = B_{i-F(j)-1 \cdots i-1}$，而且 $F(j)$ 也是符合條件的最大值，因此我們可以把 j 換成 $F(j)$ 繼續重複上列動作。特例是如果 $j = -1$，代表 $F(i) = 0$。

仔細觀察可以發現這個流程和剛才字串匹配一模一樣，也就是其複雜度為 $O(L_B)$。綜上，字串匹配總共花上 $O(L_A + L_B)$ 時間，以及 $O(L_B)$ 額外空間。

另外還有一個小優化可以做，方法是稍微修改一下失敗函數的定義：算出 $F(i) = j$ 的時候，如果發現 $B_i = B_j$，那就令 $F(i) = F(j)$。其實就是事先預測「連續在 A 的同一個位置匹配失敗很多次」的狀況。這只是一個小常數優化，想不想寫基本上都無所謂。

雖然看似有很多 case 要判斷，但是其實很好簡化，也因此程式異常地簡潔，不需要甚麼特判。以下附上 C++ 程式碼：

Algorithm 1: Knuth-Morris-Pratt Algorithm in C++

```
1  int F[N];
2  int match(const std::string& A, const std::string& B)
3  {
4      F[0] = -1, F[1] = 0;
5      for (int i = 1, j = 0; i < B.size() - 1; F[++i] = ++j) { //計算失敗函數
6          if (B[i] == B[j]) F[i] = F[j];  //優化，略去此行依然有相同結果
7          while (j != -1 && B[i] != B[j]) j = F[j];
8      }
9      for (int i = 0, j = 0; i - j + B.size() <= A.size(); i++, j++) { //匹配
10         while (j != -1 && A[i] != B[j]) j = F[j];
11         if (j == B.size() - 1) return i - j; //成功匹配到 B 字串的結尾，回傳結果
12     }
13     return -1;
14 }
```

強烈建議可以練習在沒有任何參考資料的情況下自己動手寫一次，這樣會很容易對它運作的原理有更深的理解。

2.3 Gusfield's Algorithm

這個演算法俗稱「Z algorithm」，是一個想法比較簡單的演算法。它的時間複雜度與 KMP 皆相同，但是空間複雜度是 $O(L_A)$。

此演算法是對一個字串 A 的後綴計算一個「Z 函數」，定義如下：

$$Z_A(i) = \begin{cases} 0, & \text{if } i = 0 \\ \max\{p : A_{0\cdots p} = A_{i\cdots i+p}\}, & \text{otherwise} \end{cases}$$

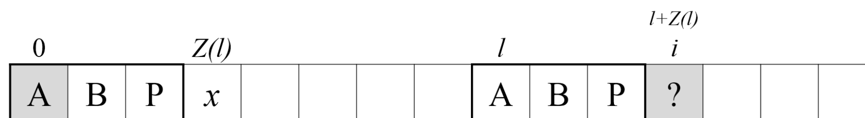
簡單來說，就是一個字串的某個後綴可以和字串本身匹配到多長，除了 $Z(0) = 0$。例如下列即是 `ABABABAABA` 的 Z 函數表：

```
   i|0 1 2 3 4 5 6 7 8 9
   S|A B A B A B A A B A
Z(i)|0 0 5 0 3 0 1 3 0 1
```

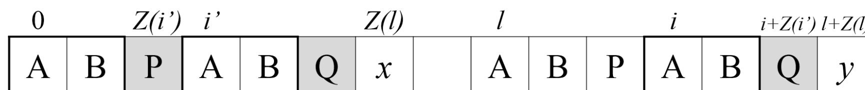
這個函數比 KMP 的失敗函數好懂很多，而且要進行字串匹配的話，只要令字串 $C = B + \phi + A$，其中 ϕ 是一個從未在 A, B 中出現過的字元，然後看是否存在 $0 \leq k < L_A$ 使得 $Z_C(L_B + 1 + k) = L_B$ 就可以了，十分簡單明瞭。

至於 Z 函數的計算想法和 KMP 一樣，都是妥善利用已算出的 Z 函數值。假設目前已經得到 $Z(k)$ $(\forall k < i)$ 的值，要計算 $Z(i)$ 的值。我們還需要知道 $l < i$ 使得 $l + Z(l)$ 最大。令 $i' = i - l$，有可能遇到四種情況：

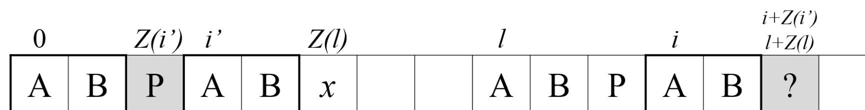
1. $l + Z(l) \leq i$，代表 A_i 以後都尚未被匹配過。直接從頭開始匹配。



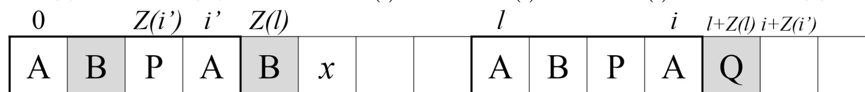
2. $l + Z(l) > i$ 且 $l + Z(l) > i + Z(i')$。可以注意到 $A_{0\cdots Z(i')} = A_{i'\cdots i'+Z(i')} = A_{i\cdots i+Z(i')}$，而且 $A_{Z(i')} \neq A_{i'+Z(i')} = A_{i+Z(i')}$。因此 $Z(i) = Z(i')$。



3. $l + Z(l) > i$ 且 $l + Z(l) = i + Z(i')$。2. 的第一個性質仍然符合，但不知道 $A_{Z(i')}$ 是否等於 $A_{i+Z(i')}$，故從該位置繼續開始匹配。



4. $l + Z(l) > i$ 且 $l + Z(l) < i + Z(i')$。因為 $A_{Z(l)-i'} = A_{Z(l)} \neq A_{l+Z(l)}$，因此 $Z(i) = Z(l) - i'$。



與 KMP 類似，我們發現在額外進行匹配（1. 與 3.）的時候，當前的 $l + Z(l)$ 會隨之增加，每一次最少會增加 1，且最多增加到 L_A，因此計算出字串 A 的所有 Z 函數值的時間複雜度為 $O(L_A)$。

這個演算法概念和實作都比較簡單，因此就不給 code 了。

2.4 習題

字串的概念不多，但是可以變化出很多的題目，因此做題的時候想法得靈活一點。

1. (ZJ d518) 依序給你一坨字串，對每個字串詢問之前有沒有出現過這個字串，以及最早在第幾個的時候出現。

2. (TIOJ 1306) 裸字串匹配。

3. (TIOJ 1321) 給一個長度 $\leq 10^6$ 的字串 A，問有幾種把該字串的某前綴搬到最後面的方法，使得最後的字串為回文。
（提示：可以用類似 Z 函數的概念計算出以某字元為中心的最長回文長度。）

4. (Codeforces 559B) 給兩個同樣長度 $\leq 2 \times 10^6$ 的字串 A, B。如果 L_A 是偶數，令 P_A 代表 $A_{0 \cdots L_A/2}$，S_A 代表 $A_{L_A/2 \cdots L_A}$。我們說 A, B 等價，代表下列三者至少一者成立：一、這兩個字串相等；二、P_A 等價於 P_B 且 S_A 等價於 S_B；三、P_A 等價於 S_B 且 S_A 等價於 P_B。求 A, B 是否等價。

5. (POI XII Stage II P4) 定義兩個字串 A, B 的「重疊」是選一個 A 後綴與 B 前綴的共同部分（可以是空字串），將刪去共同部分的 A 和 B 串接起來的結果。給一個長度 $\leq 10^6$ 的字串 A，求一個最長的字串 P 使得可以用很多個 P 重疊成 A。

6. (TIOJ 1725) 定義字串 A 和 B「k-幾乎相同」代表把字串 A 的前 k 字元搬到最後面時，與 B 恰有一個字元相異。給你兩個長度 $\leq 10^6$ 的字串 A 和 B，求所有使 A 和 B「k-幾乎相同」成立的 k 值。

3 後綴陣列

先插播一下「字典序」（lexicographical order）：先給定一個字元集裡面的每一個元素的排序，然後對於兩字串 A 和 B 找到第一個對應位置不一樣的字元比大小。特別地，如果找不到這樣一個字元，長度較小的字串字典序較小。顯然比較兩字串的字典序的複雜度是 $O(\min(L_A, L_B))$。

那所謂「後綴陣列」，即是把一個字串 A 的每一個後綴（總共 L_A 個）全部放到一個陣列 S_i 裡面，然後對其用字典序排序，所得到的陣列便稱為後綴陣列。實務上，陣列裡面會放這個後綴開頭的索引值，就不需要把整個字串複製一次了。

現在留下一個問題：要如何對所有後綴對字典序排序呢？直接開排序的複雜度是 $O({L_A}^2 \log L_A)$，有些緩慢。於是可以想到幾種優化：

3.1 基數排序

所謂基數排序（radix sort），即是對於正整數按照「不重要到重要」位（即小到大）的順序，每次對一個位數做計數排序（counting sort）。因為計數排序是穩定的，所以當所有位數被比較完後，即是排序好的結果。

此演算法也可以套用在字串上。即是對於一些字串，先把所有字串依照「最後一個字元」採用計數排序（如果長度不同需額外判斷是否超過長度），再依照倒數第二個字元、倒數第三個字元⋯⋯直到第一個字元進行排序。如此時間複雜度是 $O(L_A(L_A + |\Sigma|))$，需要額外 $O(|\Sigma|)$ 空間。

3.2 倍增法

字串演算法的中心思想就是「妥善利用已知資訊」，但上列兩種方法都沒有用到排序過程中所得到的資訊，也沒有用到「後綴」的性質。

換個方向思考，可以發現每次可以把排序的範圍擴大一倍：假如已知所有字串按照前 i 個字元排序的結果，就可以 $O(1)$ 用現有的結果得到任一對前 $2i$ 個字元的大小關係：直接先比對前半部分，如果相同再比對後半部分。因為所排序的是所有的後綴，所有後半部分的比較結果已經在排序結果裡面了。

至於要如何找到這個結果呢？可以在排序過程中維護一個排名陣列 R_i，也就是後綴 $A_{i \cdots L_A}$ 現在位在現有比較結果的第幾名，便可以快速得到大小關係。注意當前比較結果相同的要視為相同，不可以直接用比較結果的反函數。

因此，每次排序的範圍可以增加一倍，每次比較只要 $O(1)$ 時間。如果套用正常 sort，時間複雜度便是 $O(L_A \log^2 L_A)$；套用計數排序，時間複雜度 $O(L_A \log L_A + |\Sigma|)$，需要額外 $O(|\Sigma|)$ 空間。（想想為甚麼時間不會是 $O((L_A + |\Sigma|) \log L_A)$？）

這是複雜度第二好的演算法（有個叫做 DC3 的演算法可以達到 $O(L_A + |\Sigma|)$，但是過於複雜不實用），實作也不會太困難。比較需要注意的是，如果用計數排序的話，要好好判斷「超出字串範圍」的情況並放在最前面，也要小心不要打亂已經排好的順序。

另外可以發現在後綴數組上面二分搜就可以匹配字串，複雜度 $O(L_B \log L_A)$。

3.3 最長共同前綴（Longest Common Prefix, LCP）

兩字串的最長共同前綴（簡稱 LCP）可以照字面意思理解。對於兩字串的 LCP 可以輕鬆 $O(\min(L_A, L_B))$ 解決掉。但有時候在應用後綴數組時，會需要快速知道任兩個後綴的 LCP。此時有兩個性質可以使用：

1. 令 S_i 代表後綴數組上第 i 個位置所代表的後綴，$LCP(A,B)$ 代表 A 和 B 的 LCP 長度。則 $LCP(S_k,S_{k+1}) \geq LCP(S_{k-1},S_k)-1$。

2. $LCP(S_i,S_j) = \min_{i\leq k<j}(S_k,S_{k+1})$。

這兩個性質還算直觀，證明起來稍嫌複雜，可以參考 2012 年的講義。有了 1. 的性質，我們可以用 $O(L_A)$ 算出所有的 $LCP(S_k,S_{k+1})$；有了 2.，我們便可以將最長共同前綴轉換成區間最小值問題。至於區間最小值問題要如何解決，這就是下一堂課的主題了。

3.4 習題

字串很多問題都會有各種不同的解法，所以有些題目也可以嘗試看看非後綴數組的做法。

1. (No judge) 求 N 個字串 $A_1,A_2,\cdots,A_N$ 的「最長共同子字串」(longest common substring)。複雜度 $O(\sum_{i=1}^{N} L_{A_i})$。

2. (TIOJ 1497) 裸後綴數組。

3. (TIOJ 1515) 裸後綴數組 LCP。

4. (No judge) 給一個字串 A，求 A 一個最長的子字串，使得這個子字串在 A 中出現超過一次，且至少兩次出現的位置不重疊。假定 A 的後綴數組已經求出，複雜度 $O(L_A \log L_A)$。

5. (No judge) 給一個字串 A，求 A 一個最短的子字串，使得這個子字串在 A 中只出現一次。假定 A 的後綴數組已經求出，複雜度 $O(L_A)$。

6. (Codeforces 427D) 給兩個長度 ≤ 5000 的字串 A,B，求兩字串最短且滿足在兩個字串中只各出現一次的共同子字串的長度。(其實長度可以到 5×10^5。)

7. (Codeforces 653F) 給一個由左右括號組成、長度 $\leq 5\times10^5$ 的字串 A。定義一個「合法字串」為符合下列二者之一的字串：連續 k 個左括號串接連續 k 個右括號，或可以將其看成兩個合法字串的串接。求存在多少個相異的 A 的非空子字串是合法字串。
(提示：可以預習 sparse table，它可以在 $O(n\log n)$ 預處理後 $O(1)$ 查詢任意區間的最小值。)

4 後記

字串還有另一門學問，叫做「自動機」(automaton)，比較常見的有 AC 自動機和後綴自動機。可以把 AC 自動機想像成 KMP 的推廣，後綴自動機想像成後綴數組的推廣。但是因為它們稍嫌複雜，而且實際上遇到的機會不多，所以留待之後的課程再做說明。