字串

Ting.H

2017 年 9 月 27 日

1 字串簡介

字串 (string) 是由零個或多個字元所組成的有限序列。而「字元」是指字母或文字的最小單位；廣義來說是一個非空有限集合的元素。例如，$string$ 是一個字串，而它的字元集是英文字母。

「字串」和「序列」看起來差不多，但一般講到「字串」時，是要有連續性，若是題目或演算法與「連續」無關 (例如 Longest Common Subsequence)，那就不算在「字串」的範圍內。

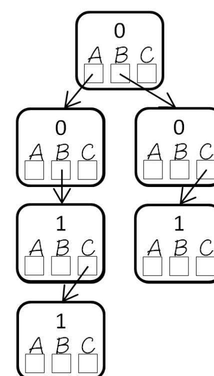
字串有一些基本的術語，以下就介紹一些常會用到的術語：

1. 字元 (character)：構成字串的單位。以下「字串 A 的第 i 個字元」簡寫為 A_i。
2. 字元集 (所有可能的字元的集合)，記為 Σ，其集合大小為 $|\Sigma|$。
3. 長度 (length)：一個字串的字元個數。以下「字串 A 的長度」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)：一個字串的一段連續的字元所構成的字串稱作子字串。以下將「字串 A 的第 $[i,j)$ 個字串所構成的子字串」簡寫為 $A_{i...j}$。(注意本篇講義的字串為左閉右開)
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。前綴可寫成 $A_{0...i}$、後綴可寫成 $A_{i...L_A}$。
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 $ti + oj = tioj$。

1.1 Trie

儲存一個字串一般是用類似陣列的資料結構 (如 char[]、std::string)。但如果是要儲存多個字串，開 string 的陣列並不一定是個好辦法。

有時我們會使用「字典樹」(trie) 來處理。字典樹的結構就是一棵樹，而使用方式和字典差不多。



字典樹的每一個節點都包含一個標記和 $|\Sigma|$ 個指標 (分別代表每種字元)。根節點代表的是空字串，其他的節點代表的是「其父節點代表的字串」+「該指標所代表的字元」。標記則是表示字典中是否存在這個節點所代表的字串 (或是該字串的數量)。

例如右圖字典樹的字元集是 {A,B,C}，所儲存的字串為 AB、ABC、BC。

字典樹的空間複雜度為 $O(|\Sigma| \times \Sigma L)$，增加、刪除、查詢字串的時間複雜度是 $O(L)$。

2 字串匹配

字串匹配，就是給定兩字串 A 和 B，求是否存在 A 的子字串 $A_{i...j} = B$。

最簡單的方式就是 $O(L_A L_B)$ 的演算法，但這個的複雜度有點悲劇。因此，以下有幾個改進的演算法。

2.1 Hash(雜湊)

Hash 雜湊，是利用函數將字串分成 M 類，而此函數稱作雜湊函數 (hash function) $h(A)$，滿足 $\forall A \in \Sigma$，有 $h(A) \in \mathbb{Z}$ 且 $h(A) \in [0, M)$，以下我們定義一個字串的雜湊函數：

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

Rabin fingerprint

顯然對於 A,B 兩字串，$h(A) \neq h(B) \Rightarrow A \neq B$。並且我們相信只要人品夠好，反過來的情況也會是對的。上方的式子的意思是字串 A 在 p 進位下模 M 的結果。仔細想會發現只要預處理 A 所有前綴的 hash 值，就可以 $O(1)$ 求 A 的任何子字串。

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

利用 hash，我們可以經過 $O(L)$ 的預處理之後，$O(1)$ 計算一個子字串的 hash 值，在 $O(L_A + L_B)$ 的時間完成「需人品的字串匹配」。

當字串 $A \neq B$ 但 $h(A) = h(B)$(碰撞) 時，字串匹配就出了問題，在隨機測資下發生碰撞的機率為 $\frac{1}{M}$。但如果 p,M 取得不好的話，機率就有可能大幅上升，所以我們一般會取兩相異質數。

當然也可以用多組 p,M 來使碰撞的機率變低。

2.2 Knuth-Morris-Pratt algorithm (KMP)

hash 終究只是機率演算法，一定有出錯的可能，而且模運算的常數本來就很大。以下介紹一個確定且有相同複雜度的演算法 KMP。

仔細觀察 $O(L_A L_B)$ 的演算法，A,B 的第一格對齊，從左到右一格一格匹配，當對應的字元不相等時，B 向右移一格。右圖舉例 AABAAAAAC 和 AABAAC 的匹配過程。

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

可以發現 1,2 兩次匹配是多餘的，因為在第 0 次匹配時，我們就知道 $A_{0...5} = B_{0...5}$。所以當 B_5 匹配失敗時，我們就可以將 B 直接向後移 3 格。

接下來就是要如何知道匹配失敗後，需向後挪的格數。若已經完成的匹配是 $A_{i-j...i} = B_{0...j}$，如果 $A_{i-p...i}(= B_{j-p...j}) \neq B_{0...p}$，那這一次的匹配就可以跳過。

由此可知，我們要對 B 的所有前綴找一個最大的 p ($p < j$ 且 $B_{0...p} = B_{j-p...j}$)，當匹配失敗時 ($A_i \neq B_j$) 就可以將 B 向右移 $j - p$ 格，並直接從 A_i 和 B_j 匹配。

以下介紹 KMP 所用到的失敗函數 (failure function):

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

假設我們已經知道所有的 $F(j)$ 值，那匹配的過程其實很簡單，只有兩種狀況：

1. $A_i = B_j$ 或 $j = -1$，此時 $i++, j++$。
2. $A_i \neq B_j$ 且 $j \neq -1$，$j = F(j)$。

上方的第 1 項顯然最多只會發生 $L_A - 1$ 次，並且因為 $F(j) < j$、$j \geq -1$，第 2 項發生的次數必定不會超過第 1 項發生的次數，所以此處的複雜度是 $O(L_A)$。

至於求 $F(j)$ 的方式和字串匹配類似，所以就不細講，而求所有位置的失敗函數值的複雜度是 $O(L_B)$。KMP 的時間複雜度是 $O(L_A + L_B)$，需額外的空間 $O(L_B)$。

除此之外還有一個小優化，當算出 $F(i) = j$ 時，如果 $B_i = B_j$，那令 $F(i) = F(j)$。這只是個常數優化，若不寫其實不會怎樣。

有優化是 KMP，沒有則是 MP。以下附 KMP 的程式碼：

Algorithm 1: Knuth-Morris-Pratt algorithm

```
int F[N];
int match(const std::string& A, const std::string& B){
    F[0]=-1,F[1]=0;
    for(int i=1, j=0;i<B.size()-1;F[++i]=++j){ //計算失敗函數
        if(B[i] == B[j])F[i]=F[j]; //優化，略去此行為MP
        while(j != -1 && B[i] != B[j]) j=F[j];
    }
    for(int i=0, j=0;i-j+B.size() <= A.size();i++, j++) { //匹配
        while(j != -1 && A[i] != B[j]) j=F[j];
        if(j == B.size()-1) return i-j; //成功匹配到 B 字串的結尾，回傳結果
    }
    return -1;
}
```

2.3 Gusfield's algorithm

這個演算法的俗稱是「Z algorithm」，概念比 KMP 簡單，時間複雜度與 KMP 相同。此演算法是對一個字串 A 的後綴計算一個函數，定義如下：

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

這個函式是在求一個字串的後綴能與字串本身匹配到的最大長度，以下有 ABABABAABA 的 Z 值：

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

在這個演算法中要儲存一個數 bst，使下方的式子成立。而明顯的 $Z(bst) + bst$ 是目前比對到的最遠距離，所以只要善用這件事，就可以利用來推 Z 值。

$$Z(bst) + bst = \max\{Z(x) + x\ , 0 < x < i\}$$

接下來就是要如何求 $Z(i)$，根據前面所提到的 bst，可以知道若 $Z(bst) + bst \leq i$，那就沒有其他的資訊，就只能從頭匹配。但若是 $Z(bst) + bst > i$，就可以善用這件事，而你會發現 $Z(i)$ 至少是 $\min(Z(bst) + bst - i, Z(i - bst))$，至於怎麼發現的就不重要了。

顯然 $A_{0...Z(bst)} = A_{bst...Z(bst)+bst}$

1. $Z(bst) + bst - i \geq Z(i - bst)$
   首先，我們知道 $A_{i-bst...Z(bst)} = A_{i...Z(bst)+bst}$、$A_{0...Z(i-bst)} = A_{i-bst...i-bst+Z(i-bst)}$，因為 $Z(bst) + bst - i \geq Z(i - bst)$，於是 $A_{0...Z(i-bst)} = A_{i-bst...i-bst+Z(i-bst)} = A_{i...i+Z(i-bst)}$，所以 Z(i) 至少是 Z(i-bst)。

2. $Z(bst) + bst - i < Z(i - bst)$
   跟第 1 項差不多但我們只能確定 $A_{0...Z(i-bst)} = A_{i-bst...i-bst+Z(i-bst)}$，而 $A_{i...i+Z(i-bst)}$ 已經超過 $Z(bst) + bst$ 了，所以我們只知道 $A_{0...Z(bst)+bst-i} = A_{i-bst...Z(bst)} = A_{i...Z(bst)+bst}$

Algorithm 2: Gusfield's algorithm

```
void z_build(const char* S,int *z){
    z[0]=0;
    int bst=0;
    for(int i=1;S[i];i++){
        if(z[bst]+bst<i) z[i]=0;
        else z[i]=std::min(z[bst]+bst-i,z[i-bst]);
        while(S[z[i]]==S[i+z[i]]) z[i]++;
        if(z[i]+i>z[bst]+bst) bst=i;
    }
}
```

講了老半天，好像跟字串匹配沒有關係，但仔細一想就會發現只要令一個字串 $S = B + \phi + A$，ϕ 是一個沒有在 A,B 兩字串出現的字元。然後看是否存在 $0 \leq k < L_A$，使得 $Z_S(L_B + 1 + k) = L_B$，就可以了。

2.4 習題

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 1276) 最長回文子字串。
4. (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 後綴數組

在講後綴數組前先來講一些前言，第一個要講的是後綴 Trie，假設現在有一個文本 BANANA，先在字串的最後面加 $(一個沒出現過的字元)，然後將它所有的後綴 (BANANA$, ANANA$, …, A$) 放到 Trie 上面，而字串的每一個後綴都和葉結點一一對應。接下來將沒有分支的鏈合併，就得到所謂的後綴樹 (suffix tree)，但因為後綴樹的構造較複雜，並且容易寫錯，所以一般競賽會用一個替代品：後綴數組。

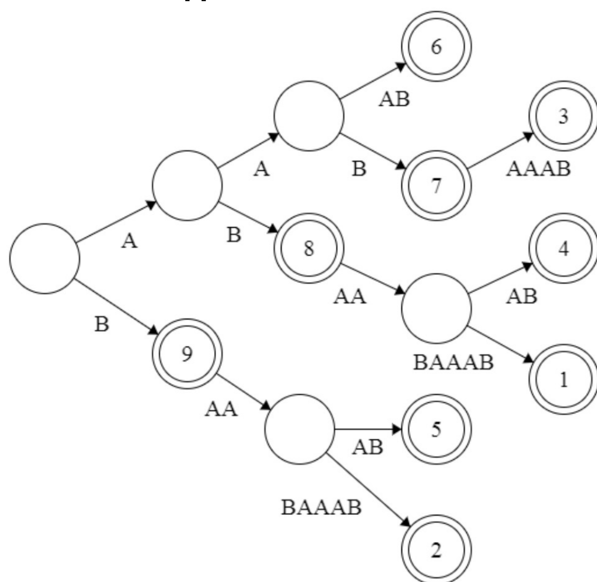
要開始講後綴數組了嗎? 當然還沒，要先講一個要用到的重要概念。

Sort

1. counting sort
   假設我們要排序的序列值是 1～100 的整數，我們就可以開一個 1～100 的陣列，記錄每一個數字出現的次數。複雜度是 $O(N+C)$，C 是數字的範圍。

2. radix sort
   將數字 (或字串) 補空白至一樣長，對每一個位數做 counting sort。複雜度是 $O(\log_B C \times (N+B))$，B 是進位制，C 是數字範圍。

下圖為一棵後綴樹 (省略 $)，表格中有三種數組「後綴數組」、「排名數組」和「hei 數組」，表格中的字串已照字典序排序。(後綴數組)$sa[i]$ 代表第 i 小的後綴是第幾個後綴，(排名數組)$ra[i]$ 代表第 i 個後綴是第幾小的後綴。($ra[sa[i]]=i$)。



i	sa	$S_A(sa_i)$	hei
1	6	AAAB	0
2	7	AAB	2
3	3	AABAAAB	3
4	8	AB	1
5	4	ABAAAB	2
6	1	ABAABAAAB	4
7	9	B	0
8	5	BAAAB	1
9	2	BAABAAAB	3

接下來的問題是要如何將後綴照字典序排序，最簡單的方式是 $O(L^2 \log L)$，但有點太慢了，所以就用前面講的 radix sort 來排序，於是複雜度就變成 $O(L(L+|\Sigma|))$。

倍增法

即使用的 radix sort，但複雜度還是不好，所以我們要用到在 LCA 提到的倍增法，假設已經將所有的後綴照前 i 個字元排序了，那就會發現可以 $O(1)$ 求任何一對後綴照前 $2i$ 個字元排序的結果，於是就 $O(L\log L+|\Sigma|)$。

順帶一題，在後綴數組上二分搜就可以匹配字串，複雜度 $O(L_B \log L_A)$。

Longest Common Prefix(LCP)

如果要算兩個後綴的 LCP 時，其實就是要求後綴樹的 LCA 深度，這時我們要用到 height 數組。hei 的定義：

$$\forall 1 < i \leq n,\ hei_A[i] = LCP(S_A(sa[i-1]), S_A(sa[i]))$$

也就是說，$hei_A[i]$ 就是後綴樹的 DFS 序中第 i 個點和第 $i-1$ 個點的 LCA 之深度。

以下是 hei 的性質：

$$\begin{aligned} hei_A[ra[j]] &= LCP(S_A(sa[ra[j]-1]), S_A(sa[ra[j]])) \\ &= LCP(S_A(sa[ra[j]-1]), S_A(j)) \\ &\geq LCP(S_A(sa[ra[j]-1]-1), S_A(j-1)) \\ &= hei_A[ra[j-1]]-1 \end{aligned}$$

想一下還會發現 $LCP(S_i, S_j) = \min_{i \leq k < j} LCP(S_k, S_{k+1})$，於是問題就變成了「區間最小值」，這會在之後的講義提到。

習題

1. (TIOJ 1497) 裸後綴數組。
2. (TIOJ 1515) 裸後綴數組 LCP。
3. (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。)
4. (Codeforces 653F) 給一個由左右括號組成、長度 $\leq 5 \times 10^5$ 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 自動機

字串還有另一門學問，「自動機」，常聽到的有 AC 自動機 (KMP 的推廣)、後綴自動機 (後綴數組的推廣)。因較為複雜，所以以後有機會再說明。有興趣的可以自行上網查詢。