# Lecture 1
# 流 | Flow

網路流與我們生活中的一些問題息息相關，比如我們要從一工作站 $s$ 傳輸一個資料到工作站 $t$ ，傳輸的過程有一些中繼站，並且某些中繼站間和某些中繼站到工作站間有傳輸率爲 $c$ 的網路線相通，現在要如何找出一個最佳的傳輸方案，使得 $s$ 到 $t$ 的資料傳輸速率最快?

以上就是網路流問題的原型，但隨後人們驚訝的發現網路流還可以用以解決一些看似無關的問題! 如匹配問題、歐拉回路問題等等。網路流的問題千變萬化，常常令人異想不到，也因此常成爲程式競賽的難題，如果能精熟這個領域，便如有了一把解題的利刃!

## 1.1 前言

首先我們定義何謂網路流。

**定義：** 一個 **s-t 網路流**是一張圖 (有向或無向) $G = (V, E)$ ，每條邊上有一個非負的權重 $c(u, v) \geq 0$ 代表邊的**流量上限** (如果 $(u, v) \notin E$ 我們定義 $c(u, v) = 0$)。並且有二個特別的點，源點 $s$ 與匯點 $t$ 。

一個 **s-t 可行流** 是一個函數 $f : V \times V \mapsto \mathbb{R}$ 滿足以下兩個條件

- $f(u, v) \leq c(u, v),\ \forall (u, v) \in V \times V$ (流量限制) (1.1)
- $f(u, v) = -f(v, u),\ \forall (u, v) \in V \times V$ (流量對稱) (1.2)
- 對於所有 $v \in V$ ，有 $\sum_{u \in V} f(v, u) = 0$ (流量守衡) (1.3)

而我們定義這個 s-t 流的**流量**爲

$$|f| = \sum_{v \in V} f(s, v)$$

換句話說，我們必須給出每一條邊應往哪個方向流、流多少。當然，不是隨隨便便流都可以的，必須滿足這三個限制!

(1.1)：每條邊的流量都沒有超過他的限制。
(1.2)：如果 $u$ 往 $v$ 流 $c$ 單位就相當於 $v$ 往 $u$ 流 $-c$ 單位。
(1.3)：流不能無中生有！每一點進去的流量要等於出去的流量，也就是淨流量必須是 0 。

舉個例子，如果 $\Sigma = \{\texttt{"A"}, \texttt{"C"}, \texttt{"G"}, \texttt{"T"}\}$，$A$ 可能表示你的 DNA 序列，或者如果 $\Sigma = \{\texttt{"0"}, \texttt{"1"}, \cdots \texttt{"9"}\}$，那麼 $A$ 可能代表客戶的電話號碼。這時候就會有許多有趣的問題了，比如:

- 假設另一段 DNA 字串 $B$ 代表一個可能的致癌基因，如何快速的判斷一個人是否帶有這種基因?
- 現在有幾種不同的生物的 DNA 序列，如何判斷這些字串間的相似成度，以了解演化順序?

對於這些問題，數據的大小可以是非常大的，我們勢必會需要一些好的演算法來解決它們! 不過在這之前我們必需先說清楚一些定義。

**定義：** 給定一個字串 $A = a_0a_1 \cdots a_{n-1}$

- 一個*子字串*是其連續的一段 $a_ia_{i+1}a_{i+2} \cdots a_j$ 記作 $A[i, j]$。
- 一個*子序列*是一個字串 $B = a_{q_1}a_{q_2}a_{q_3} \cdots a_{q_m}$，其中 $0 \leq q_1 < q_2 < q_3 < \cdots < q_{m-1} < q_m < n$ 。
- 一個 $A$ 的*前綴*是 $A$ 的一個子字串 $a_0a_1a_2 \cdots a_h$，其中 $0 \leq h < n$，記作 $P_A(h)$。
- 一個 $A$ 的*後綴*是 $A$ 的一個子字串 $a_ka_{k+1}a_{k+2} \cdots a_n$，其中 $0 \leq k < n$，我們特別記作 $S_A(k)$。並讓所有後綴的集合稱作 $\sigma(A)$。

舉例來說，如果 $A =$ "abcbbab"，那"bcb" 是他的子字串，"acb" 是他的子序列，而"bbab" 是他的一個後綴。

## 1.2 字串的儲存

通常最基本的儲存方式就是用一個陣列依序將字串的每一個字元存下來。不過當我們要同時儲存許多字串時，可能就要花點巧思了。而這邊要介紹一個可以同時儲存多個字串的資料結構—**字典樹 Trie**。
Trie 的道理非常簡單，其實就是用一棵樹來儲存字串。在這棵樹上，每個點 (除了根節點之外) 上都有一個字元，而從根節點一路走到某個節點，依序經過的字元串起來就是那個點代表的字串。最後我們再記錄哪些點是一個字串的最後一個字元即可! 如 Figure 1.1就是一個儲存 $\{A_1 =$ "abc", $A_2 =$ "abde", $A_3 =$ "bc", $A_4 =$ "bcd"$\}$ 的 trie

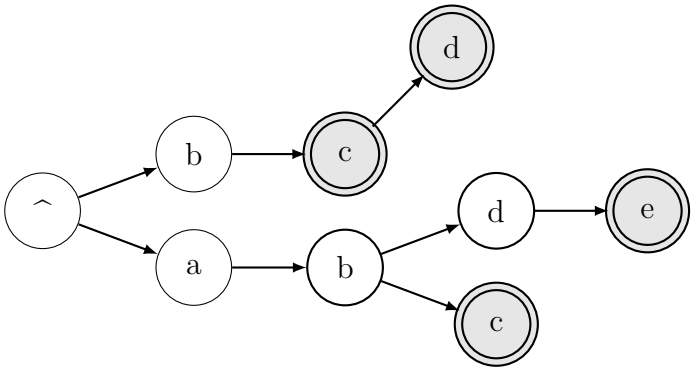


Figure 1.1: An example of trie

**定義：** 給定一個字典樹 Trie $T = \{V, E\}$，我們定義 $P_T(v)$ 爲從根節點走到 $v$ 所得出的字串。

而 Trie 的基本操作也都很簡單，如要新增一個字串 $A$，我們就從根節點開始，依照字串 $A$ 的第 $0, 1, 2, \cdots, n-1$ 個字元，如果此字元在當前節點的子節點中就繼續走下去，否則就新增一個節點。

## 1.3 字串匹配

字串匹配是一個生活中一定會遇到的問題。當你要從一篇滿滿文字的網頁或是 pdf 檔找關鍵字時，相信大家一定會本能的按下 ctrl-f，接著一瞬間，電腦就會自動算好所有蒐尋字串出現的位置。看似簡單，嘿嘿！其實裡面可是有大學問的呢。

例題 **1-1** String Matching — Everywhere

給你兩個字串 $A, B$ 找出所有 $B$ 出現在 $A$ 中的位置。

而一般大家最先想到的做法通常是...就把所有的位置試過一遍!

naive-matching **程式碼片段 1-1**

```
for (int st = 0; st + lenB <= lenA; st++) {
    int mat = 0;
    while (mat < lenB and A[st + mat] == B[mat]) mat++;
    if (mat == lenB) {
        // Match! A[st, st+lenB-1] = B
    }
}
```

這樣寫好不好呢？其實可以證明這樣的**期望複雜度**是 $\mathcal{O}(|A|)$。可惜的是，最差的情況可以

到 $\mathcal{O}(|A||B|)$! 更糟的是很容易構造出例子，如 $A =$ "AAAAA$\cdots$AA", $B =$ "AAA$\cdots$B"，這樣雖然 $B$ 從來沒有出現在 $A$ 中，但是每個位置我們都必需匹配到 $B$ 的最後一個字元才能確定匹配失敗! 那要怎麼改進這個演算法呢？著手點有兩個:

1. 事前處理：我們剛剛一拿到兩個字串便埋頭苦幹，但如果我們能先對這兩個字串做點觀察、處理，或許能得到一些有用的資訊。
2. 善用已知的資訊：在匹配的過程中，或許我們也可以得到一些訊息! 顯然這個做法並沒有妥善利用!

## 1.4 Hashing

想想看你要在茫茫人群中找你的同學，通常會怎麼找呢？不外乎是先看衣服顏色，髮形之類的。沒錯! 如果我們可以把東西加以分類，便可加速蒐尋速度! 但當然，分類也是有技巧的，就像你不會用" 正立/倒立走路" 或是" 指紋圖形" 來分類路人，因爲前者沒有多少人是用倒立走路的吧，而後者雖然極精確，但也太花時間了!
在字串上也是相同的道理，總而言之我們想做的事情是:

1. 找到一個分類函數，即是一個 $f : \mathbb{S}(\Sigma) \mapsto \mathbb{Z}$，把所有可能的字串打到有界的整數，不妨說 $f(s) \in [0, M-1]\ \forall s$ 吧!
2. 這個函數最好是均勻分部在 $[0, M-1]$ 上。
3. 計算這個函數最好不需花太多時間!

**定義：** 一個滿足以上條件的函數我們就稱作 *Hash function*

有了 Hash function 後有什麼用呢？有的! 雖然我們**無法保證** $A \neq B \ \Rightarrow f(A) \neq f(B)$，因爲 $f$ 把有無窮多個元素的字串集合打到有限個整數上，當然會有許多字串被打到同一個整數! 但至少我們會知道

$$f(A) \neq f(B) \quad \Rightarrow \quad A \neq B$$

也就是說如果兩個字串的 Hash value 不一樣，那我們連匹配都不需要了，他們鐵定不相等!
至於 Hash function 要怎麼找呢？一個常用的方法如下

**Rabin-Karp rolling hash function:** 給定 $p, q$, 令

$$\begin{aligned} f(A) &= a_0 p^{n-1} + a_1 p^{n-2} + \cdots + a_{n-2} p + a_{n-1} \mod q \\ &= \sum_{0}^{n-1} a_i p^{n-i-1} \mod q \end{aligned} \tag{1.4}$$

看起來很複雜，其實就是字串 $A$ 在 $p$ 進位制代表的值模 $q$ 而已! 那這個函數有符合我們的需求嗎？首先他把每個字串打到 $[0, q-1]$，可以想成他把所有字串分成 $q$ 類。另外數學家跟

我們說，如果 $p, q$ 取兩個不同的質數，通常結果會不錯，非常均勻[1]！另外計算這個函數只需要 $\mathcal{O}(|A|)$，並且他還有一個很好的性質!

**Sliding Window：** 假設 $f$ 由(1.4)給出，則

$$f(A) \equiv f(A[0, n-2])p + a_{n-1} \pmod{q} \tag{1.5}$$

並且

$$f(A[i, j]) \equiv f(A[0, j]) - p^{j-i+1} f(A[0, i-1]) \pmod{q} \tag{1.6}$$

這些都可以由(1.4)計算得出。(1.5)告訴了我們計算字串 $A$ 所有前綴的 hash value 可以在 $\mathcal{O}(|A|)$ 的時間用遞迴求出! 仔細看(1.6)可以發現如果知道了 $A$ 所有前綴的 hash value，任何 $A$ 的子字串的 hash value 都可以在 $\mathcal{O}(1)$ 的時間計算出來[2]!

回到我們字串匹配的問題，相信大家都看出來了，我們只需要事先算出 $A$ 所有前綴的 hash value 和 $f(B)$，再枚舉 $A$ 所有長度爲 $|B|$ 的子字串 (差不多 $\mathcal{O}(A)$ 個)，最後計算這些子字串的 hash value 是不是等於 $f(B)$，總共只需要 $\mathcal{O}(N)$...等等! 回想我們剛剛說的: 我們知道 $f(A) \neq f(B) \Rightarrow A \neq B$，但卻無法保證 $f(A) = f(B) \Rightarrow A = B$ 啊? 有人可能會想說: " 相等時重新檢查一次"，但如果 $A =$ "AAA$\cdots$AAA", $B =$ "AA$\cdots$AA"，完蛋了! 又退化成 $\mathcal{O}(|A||B|)$ 了!

那怎麼辦呢? 答案是: 把 $q$ 取大一點，然後就假設 $f(A) = f(B)$ 的機率很小，不會發生! 事實上如果 $f$ 是均勻的，那 $f(A) = k$ 的機率差不多是 $1/q$! 只要 $q$ 取夠大，比如一個 `long long` 的質數，差不多 $10^{15}$，那麼兩個不同的字串**碰撞**的機率是 $10^{-15}$，是一個人被閃電打到兩次的機率[3]，不太可能啦!

## 1.5 KMP

剛剛說的 Hashing method 畢竟還是一個機率演算法，而且模運算是很花時間的! 因此這邊要講一個很厲害的演算法，Knuth–Morris–Pratt algorithm!

我們先回到一開始 naive matching 失敗的例子，$A =$ "aabaabaac", $B =$ "aabaac"，我們第一次的匹配如下，在 5 的位置匹配失敗。

[1]可以參考習題 1-8

[2]眞的嗎? $p^{j-i+1}$ 怎麼辦?

[3]機率一次差不多是 $8 \times 10^{-7}$

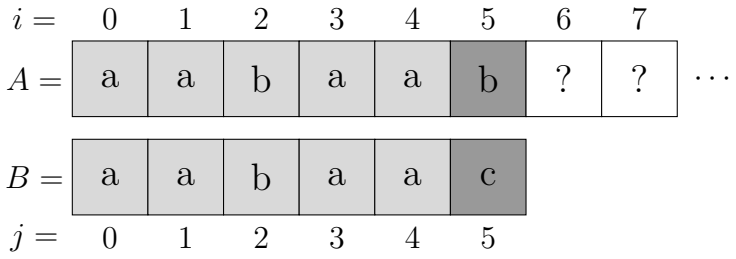


Figure 1.2

那些 $A$ 還沒被匹配過的地方我們用? 表示。現在如果我們用最原始的方法，我們必須將 $B$ 右移一格繼續匹配，重新來過。但仔細觀察就會發現

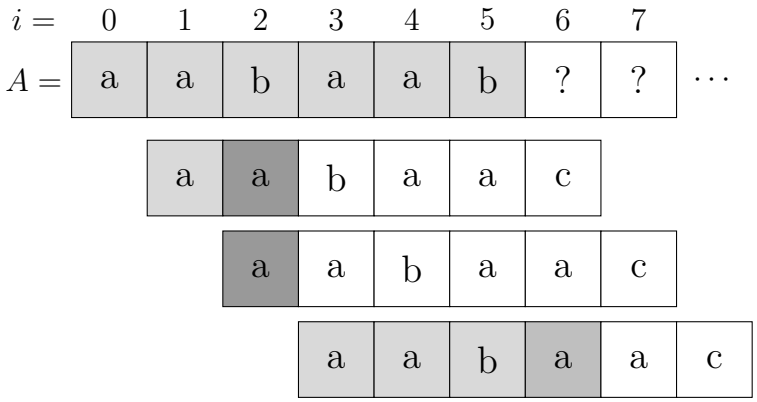


Figure 1.3

右移 $1, 2$ 格分別在 $1, 0$ 的位置就匹配失敗了。注意到這兩個位置都對應到 $A[2]$，而 $A[2]$ 是我們早就匹配過的! 也就是說這個" 失敗" 是我們應該要知道的。更甚者，這整件事情只跟 $B$ 本身有關! 因爲我們早以確認過 $A[0, 4] = B[0, 4]$。換句話說，對於一個字串 $B$，不論 $A$ 是多少，我們早就可以知道某些 $(j, s)$，代表" 如果我們在 $B[j + 1]$ 的位置匹配失敗後，往前 $s$ 格繼續匹配" 這件事情，是完全不必要的!

那到底在 $B[j + 1]$ 匹配失敗時，我們該往右移幾格呢? 根據我們上面的討論，如果我們已經匹配了 $A[i, i + j] = B[0, j]$，現在將 $B$ 右移 $s$ 格，那麼至少 $A[i + s, i + j] = B[0, j - s]$，也就是

$$B[0, j - s] = A[i + s, i + j] = B[s, j]$$

因此我們應該要找第一個 (最小的)$s$ 滿足上式，也就是最大的 $j - s$。這就是我們要定義的 KMP 裡一個非常重要的函數—失敗函數 $\mathcal{F}(j)$。

**定義 (Fail function):** 對於一個字串 $B = b_0 b_1 \cdots b_{m-1}$，我們定義

$$\mathcal{F}_B(i) = \begin{cases} \max\{k : P_B(k) = B[0, k] = B[i - k, i]\} & \text{if } i \neq 0 \text{ and at least a } k \text{ exists} \\ -1 & \text{else} \end{cases}$$

$\mathcal{F}(i)+1$ 也稱作在第 $i$ 個位置的共同前後綴長度

由上面的推論，我們總結 $\mathcal{F}(j)$ 的一個非常重要的性質: $\mathcal{F}_B(j)$ **告訴我們在拿** $B$ **去匹配** $A$ **的過程中，如果** $B[0,j]$ **已經匹配成功，但在第** $j+1$ **個位置匹配失敗了，應該要把** $B$ **的第** $\mathcal{F}(j)$ **個字元對齊原本** $B[j]$ **的位置繼續匹配**! 舉個例子，容易知道如果 $B=$ **"aabaabd"**，則 $\{\mathcal{F}(j)\}=\{-1,0,-1,0,1,2,-1\}$，假設我們已經匹配 $B[0,4]$，但在第 5 個字元出問題了，

```
01234567
A = aabaaa?????
    |||||*        Matching failed at position 5
B = aabaabd
       ||*        F(4) = 1, Matching failed at position 2
      aabaabd
        |*        F(1) = 0
         aabaabd
```

這樣我們一次可能往前一大步，而不用每次位移一格重新匹配了!
現在只剩一個問題了，要怎麼求失敗函數呢? 其實很簡單，假設我們已經求出了 $\mathcal{F}(i),\forall 0\le i\le n$，現在要求 $\mathcal{F}(n+1)$，根據定義相當於要求最大的 $k=k'+1$ 使 $B[0,k]=B[n+1-k,n+1]$。而

$$B[0,k]=B[n+1-k,n+1]\ \Leftrightarrow\ B[0,k']=B[n-k',n]\wedge B[k'+1]=B[n+1]$$

由失敗函數的定義我們知道 $k'$ 最大只能是 $\mathcal{F}(n)$,如果此時 $B[\mathcal{F}(n)+1]=B[n+1]$ 我們立刻便知道 $\mathcal{F}(n+1)=\mathcal{F}(n)+1$。但如果不是怎麼辦? 難道必需 $k'=\mathcal{F}(n)-1,\mathcal{F}(n)-2\cdots,0$ 一直試下去嗎? 不要忘記我們已經算出所有 $n$ 以下的 $\mathcal{F}(i)$ 了，而 $\mathcal{F}(j)$ 告訴我們在 $B$ 匹配 $A$ 的過程中，如果

$$A[i,i+j]=B[0,j]\ \wedge\ A[i+j+1]\neq B[j+1]$$

我們應該把 $B$ 的第 $\mathcal{F}(j)$ 個字元對齊原本 $B[j]$ 的位置繼續匹配，令 $A=B,i=n-\mathcal{F}(n),j=\mathcal{F}(n)\le n$，上式變成

$$B[n-\mathcal{F}(n),n]=B[0,\mathcal{F}(n)]\ \wedge\ B[n+1]\neq B[\mathcal{F}(n)+1]$$

這不恰恰是我們現在的情況嗎? 因此由 $\mathcal{F}(j)=\mathcal{F}(\mathcal{F}(n))$，我們便把當前位置 $(n)$ 對齊 $B[\mathcal{F}(\mathcal{F}(n))]$。也就是下一個要試的 $k'$ 是 $\mathcal{F}(\mathcal{F}(n))$! 如果又失敗，我們便再試 $\mathcal{F}^3(n),\mathcal{F}^4(n)\cdots$，直到終於成功或是確認沒有 $k$ 存在 $(\mathcal{F}(n+1)=-1)$。這正是失敗函數的精髓! 他告訴我們一旦失敗該何去何從。
這邊附上一個 KMP 的範例程式碼，並總結一下 KMP 的性質。

kmp 程式碼片段 1-2

```
void build_fail_function(string B, int *fail) {
    int len = B.length(), pos;
    pos = fail[0] = -1; //Specially fail[0] = -1
    for (int i = 1; i < len; i ++) {
        while (pos != -1 and B[pos + 1] != B[i])
            pos = fail[pos];

        if (B[pos + 1] == B[i]) pos ++;

        fail[i] = pos;
    }
}

void match(string A, string B, int *fail) {
    int lenA = A.length(), lenB = B.length();
    int pos = -1;
    for (int i = 0; i < lenA; i ++) {
        while (pos != -1 and B[pos + 1] != A[i])
            pos = fail[pos];

        if (B[pos + 1] == A[i]) pos ++;

        if (pos == lenB - 1) {
            // Match ! A[i - lenB + 1, i] = B
            pos = fail[pos];
        }
    }
}
```

**定理：** 對於一個字串 $B$，有

- $B[0, \mathcal{F}(i)]$ 是 $B$ 最長的一個前綴使得 $B[0, \mathcal{F}(i)] = B[i - \mathcal{F}(i), i]$ 但 $\mathcal{F}(i) \neq i$
- 令 $\mathcal{F}^k(i) = \overbrace{f \circ f \circ \cdots \circ f}^{k}(i)$，則:
  - $\exists n,\ \mathcal{F}^n(i) = -1$
  - $\mathcal{F}^{k+1}(i) < \mathcal{F}^k(i) \quad \text{if} \quad \mathcal{F}^k(i) \neq -1$ (1.7)
  - 令 $K = \{i, \mathcal{F}(i), \mathcal{F}^2(i), \cdots, \mathcal{F}^{n-1}(i), \mathcal{F}^n(i) = -1\}$，則
    $B[0, k] = B[i - k, i] \ \Leftrightarrow\ k \in K$ (1.8)
- $-1 \leq \mathcal{F}(i+1) \leq \mathcal{F}(i) + 1$ (1.9)

最後我們分析一下 KMP 的時間複雜度，參考範例程式碼，可以發現不管在計算 $\mathcal{F}$ 或是在匹配，對於每一次的匹配，當前 $B$ 的匹配位置 (current_pos) 會

(a) 被疊代入 $\mathcal{F}$ 若干次。

(b) 如果匹配成功，便加 1。

但因爲(1.7)，我們知道每次疊代 current_pos 至少會減 1，並且疊代到 $-1$ 時便會停止，因此 (a) 中疊代的次數不會超過 (b) 被執行的次數! 而 (b) 又不會超過字串的長度，所以 KMP 的時間複雜度是 $\mathcal{O}(|A|+|B|)$，線性!

## 1.6 Aho–Corasick Algorithm*

Aho-Corasick Algorithm 可以說是 KMP 的強化板。如果今天我們要在字串 $A$ 上搜尋很多字串 $B_1, B_2, \cdots B_n$ 要怎麼做? 當然我們可以做 $n$ 次 KMP 得到一個 $\mathcal{O}(n|A|+\sum|B|)$ 的方法，但信不信由你，其實我們可以在 $\mathcal{O}(|A|+\sum|B|)$ 的時間完成!
KMP 在進行字串匹配之前必需要計算失敗函數，與此類似，Aho-Corasick Algorithm 要先造出一個 AC 自動機，聽起來很炫，好像是可以自動幫我們 AC 題目的機器，其實應該只是 Aho-Corasick 的縮寫而已啦!

還記得 KMP 的 Fail function $\mathcal{F}$ 嗎? 其實我們可以想像 Fail function 是一堆指針，告訴我們匹配失敗時要回到哪裡。現在我們要推廣到多個匹配字串呢? 首先我們要把所有的匹配字串建成一棵**字典樹 Trie**，此時你會發現，匹配的過程其實就是在 Trie 上面移動! 假設我們已經在點 $v$ ，如果下一個字元 $A[i+1]$ 存在於 $v$ 的子節點中，那我們就繼續往下走。現在和單字串匹配其時一模一樣! 我們現在只需要克服那一個共同的問題即可，當匹配失敗時，也就是 $A[i+1]$ 不在 $v$ 的子節點中，如何避免回到根節點全部重新來過? 回憶一下 $\mathcal{F}(j)$ 的定義

$$B[0,\mathcal{F}(j)] = P_B(\mathcal{F}(j)) = B[j-\mathcal{F}(j), j], \quad \text{且}\,\mathcal{F}(j)\ \text{最大}$$

也就是說 $P_B(\mathcal{F}(j))$ 要是 $B[0,j]$ 的後綴 [4] 。那在字典樹上，我們理所當然應該修正爲以下的定義

**定義：** 對於一個字典樹 $T=\{V,E\}$，其中 $v_0$ 爲他的根節點，對 $v\in V$ 我們定義

$$\mathfrak{F}(v)=\begin{cases} u & \text{if } P_T(u)\ \text{是}P_T(v)\ \text{的一個後綴且}|S_T(u)|\text{最大} \\ v_0 & \text{else} \end{cases}$$

記得 $P_T(v)$ 我們定義成從根節點走到 $v$ 的字串。這樣與 KMP 時一樣，我們匹配失敗時就沿著 $\mathfrak{F}(v)$ 往下走，直到可以繼續匹配下去或是回到了根節點了。
現在唯一的問題只剩下如何建出 $\mathfrak{F}(v)$ 了。觀察後可發現

$$u=\mathfrak{F}(v) \quad \Rightarrow \quad |S_T(u)|<|S_T(v)|$$

因此我們可以以 BFS 的順序，逐一建出這個表!

[4] 回憶一下 $\mathcal{F}(i)+1$ 也是**最長共同前後綴**長度

AC automata 程式碼片段 1-3

```
root->fail = NULL;
queue<Node*> que;
que.push_back(root);
while (not que.empty()) {
    Node *fa = que.front(); que.pop_front();
    for (auto it = fa->child.begin();
         it != fa->child.end(); it++) {
        Node *cur = it->second, *ptr = fa->fail;
        while (ptr and not ptr->child.count(it->first))
            ptr = ptr->fail;
        cur->fail = ptr ? ptr->child[it->first] : root;
        que.push(cur);
    }
}
```

與 KMP 相似，匹配時若失敗便往 $\mathfrak{F}(v)$ 走，直到走到根節點或是終於匹配成功爲止。

## 1.7 Z Algorithm

在計算一個答案時，如果能妥善利用已知的資訊，便可以加速計算所需的時間。而 Z Algorithm 便是充分的利用這一點。現在我們就來介紹這個名字很帥氣的演算法。首先我們定義 Z function

**定義：** 對於一個字串 $A = a_0a_1\cdots a_{n-1}$，定義

$$Z_A(i) = \begin{cases} 0, & \text{if } i = 0 \text{ or } A[i] \neq A[0] \\ \max\{k : A[0, k-1] = A[i, i+k-1]\}, & \text{else} \end{cases}$$

看起來和失敗函數 $\mathcal{F}(i)$ 有點像，但不一樣的是 $Z(i)$ 表示 $A$ 的後綴 $S_A(i)$，也就是從 $A[i]$ 開始的字串，可以和 $A$ 自己匹配多長。舉例來說如果 $A =$ "abcabc"，則 $Z(3) = 3$。

現在我們需要一個快速求出所有 $Z(i)$ 的方法，假設我們已經知道了 $Z(i) = z$，也就是 $A[0, z-1] = A[i, i+z-1]$。那麼 $Z(i+1), Z(i+2), \cdots, Z(i+z-1)$ 是否會和 $Z(1), Z(2), \cdots, Z(z-1)$ 有關係呢？事實上 Z function 有一個很重要的性質是

**定理：** 對於一個字串 $A = a_0a_1\cdots a_{n-1}$，如果 $Z(i) = z$，則

- $A[k] = A[i+k]$, if $0 \leq k < z$.
- $A[z] \neq A[i+z]$. (1.10)
- 令 $L = i, R = i+z-1$，現在假設 $L \leq j \leq R, j' = j - L$，則:

- 如果 $j' + Z(j') < z$，則 $Z(j) = Z(j')$ (1.11)
- 如果 $j' + Z(j') > z$，則 $Z(j) = R - j + 1$ (1.12)
- 如果 $j' + Z(j') = z$，則 $Z(j) \geq R - j + 1 = Z(j')$ (1.13)

重點是(1.11), (1.12), (1.13) 這三個 case，我們來好好解釋

Case 1. $j' + Z(j') < z$

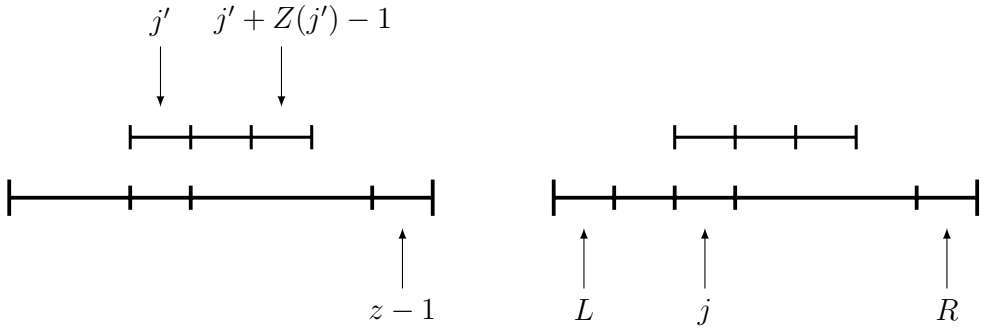


Figure 1.4

這種情況因爲沒有超出以前計算的範圍，所以我們可以直接用之前的結果，$Z(j) = Z(j')$。

Case 2. $j' + Z(j') > z$

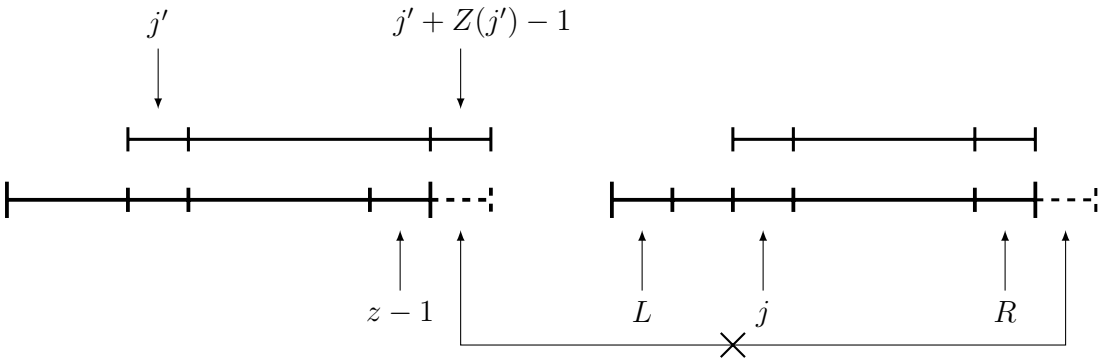


Figure 1.5

這個情況由 (1.10) 我們會知道 $A[j' + Z(j') - 1] \neq A[R + 1]$，所以從 $j$ 開始最長也只能匹配到 $R$ 了，因此 $Z(j) = R - j + 1$。

Case 3. $j' + Z(j') = z$ 最後一種情況雖然我們無法直接得出 $Z(j)$，但我們至少會知道 $Z(j) \geq Z(j')$，因此我們繼續從 $R$ 下去匹配就可以了!

仔細分析了之後我們可以知道，除非是第 3 種 Case 才需要計算。假設計算 $Z(j)$ 時用到了第 3 種 Case，我們就把 $(L, R)$ 更新爲 $(j, j + Z(j) - 1)$，可以發現如這樣做 $R$ 值只會一直增加，且增加次數不可能超過字串的長度，並且匹配的次數恰好就是 $R$ 增加的次數，因此求出所有位置的 $Z(i)$ 時間複雜度只需 $\mathcal{O}(N)$。

Z value 程式碼片段 1-4

```
int len = A.length();
Z[0] = 0;
int L, R;
L = R = 0;
for (int i = 1; i < len; i ++) {
    if (i > R) Z[i] = 0;
    else {
        int ip = i − L;
        if (ip + Z[ip] < Z[L]) Z[i] = Z[ip]; // Case 1
        else Z[i] = R − i + 1; // Case 2, 3
    }
    while (i + Z[i] < len and
           A[i + Z[i]] == A[Z[i]]) Z[i] ++;

    if (i + Z[i] − 1 > R) {
        L = i;
        R = i + Z[i] − 1;
    }
}
```

不過這和字串匹配有什麼關係呢? 假設我們要拿 $B$ 匹配 $A$ ，只要令 $C = B\phi A$，其中 $\phi$ 是從來沒有在 $A, B$ 間出現過的字元，這樣如果 $A[i, i+k-1] = B,\ k = |B|$ ，必有 $C[k+i+1, 2k+i] = C[0, k-1]$ ，也就是 $Z_C(k+i+1) = k$。

除此之外 Z algorithm 還可以解決諸如最長回文子字串等等的問題，方法和這裡類似，就留給讀者思考了。

## 1.8 Suffix Array

Suffix Array 是解決字串問題中的一把很神秘的武器，乍看之下不知道他的用處何在，其實一大半的問題都可以用他來解決。這麼好用的東西一定要把他學起來!

**定理：** 對於一個字串 $B$ 的兩個不同的後綴 $S_B(i), S_B(j)$，必有 $S_B(i) \neq S_B(j)$，即

$$S_B(i) < S_B(j) \quad \text{or} \quad S_B(i) > S_B(j)$$

這是理所當然的，因爲後綴兩兩不等長。而後綴數組要做的事就是把他們的順序找出來。後綴的順序在最後會給我們許多非常有用的資訊!

首先我們定義一些東西:

**定義：** 對於一個長度爲 $n$ 的字串 $B$，令 $S_B = \{S_B(0), S_B(1), \cdots S_B(n-1)\}$ 爲所有後綴的集合，定義

1. $\mathcal{R}(i)$ 表示後綴 $S_B(i)$ 在 $S_B$ 中是字典序第幾小的 (從 0 開始算)。
2. $\mathcal{SA}(i) = \mathcal{R}^{-1}(i)$，也就是說第 $i$ 小的後綴是哪一個。
3. $\mathcal{H}(i) = x$ 代表 $S_B(\mathcal{SA}(i))$ 跟 $S_B(\mathcal{SA}(i-1))$ 前 $k$ 個字元相同，也就是說如果 $S_B(\mathcal{SA}(i)) = S_1$, $S_B(\mathcal{SA}(i-1)) = S_2$，則 $S_1[0, k-1] = S_2[0, k-1]$。我們特別規定 $\mathcal{H}(0) = 0$。

這邊一定不能搞混，$\mathcal{R}(i)$ 表示 "$i$ 是第幾名"，而 $\mathcal{SA}(i)$ 表示 "誰是第 $i$ 名"，兩者互爲反函數。舉個列子，對於字串 "ABAABAAAB"：

| $i$ | $\mathcal{SA}$ | $\mathcal{H}$ | Suffix |
|---|---|---|---|
| 0 | 5 | 0 | AAAB |
| 1 | 6 | 2 | AAB |
| 2 | 2 | 3 | AABAAAB |
| 3 | 7 | 1 | AB |
| 4 | 3 | 2 | ABAAAB |
| 5 | 0 | 4 | ABAABAAAB |
| 6 | 8 | 0 | B |
| 7 | 4 | 1 | BAAAB |
| 8 | 1 | 3 | BAABAAAB |

在學會如何使用 suffix array 以前，我們得先學會如何建造 suffix array 。而建造 suffix array 的演算法也很多種，最佳的時間複雜度爲 $\mathcal{O}(N)$ 。但原理複雜，因此這裡我們先介紹一種最基本的倍增算法，複雜度是 $\mathcal{O}(N \log N)$，但在這之前我們需要一點先備知識。

**定理：** 如果 $A[0, k-1] = B[0, k-1]$ 我們便寫作 $A =_k B$，相同的道理我們可以定義 $A <_k B$，則

$$A < B \quad \Rightarrow \quad \begin{cases} A <_k B & \text{or} \\ A =_k B \ \wedge \ S_A(k) < S_B(k) \end{cases}$$

其實就是說假定今天 $A < B$，且 $A = A_1A_2, B = B_1B_2$, $|A_1| = |B_1|$，那麼不是 $A$ 在第一階段就比輸了 $A_1 < B_1$，就是在第二階段才比輸，$A_1 = B_1$ and $A_2 < B_2$。
藉由這個性質，我們可以觀察到一件很重要的事情: 如果我們已經知道所有後綴在只比較他們的前 $k$ 個字元時的大小關係，那麼我們其實可以快速的找出比較前 $2k$ 個字元時的大小關係，因爲任何後綴的最後 $k$ 個字元也是原本字串的某個後綴的前 $k$ 個字元! 因此倍增的做法便是

1. 使用某種方法讓所有後綴依照第一個字元排好
2. 基於上方結果，讓所有後綴依照前 2 個字元排好

3. 基於上方結果，讓所有後綴依照前 4 個字元排好
4. ....
5. 基於上方結果，讓所有後綴依照前 $2^{\lceil \log_2 |A| \rceil}$ 個字元排好

祥細的流程如下，定義 $\mathcal{R}_k(i)$ 表示依照前 $k$ 個字元排序時，比第 $i$ 個後綴小的有幾個 (也就是名次，只是相等的名次一樣)

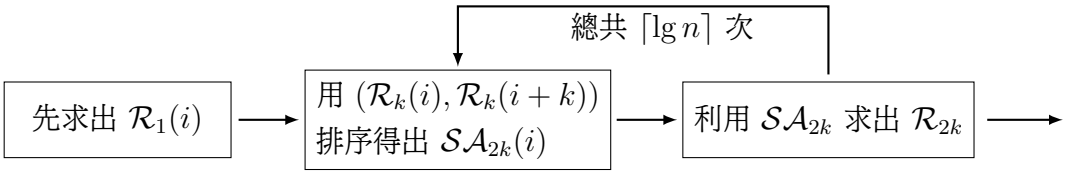


Figure 1.6

從 Figure 1.6 中我們知道瓶頸在排序 $(\mathcal{R}(i), \mathcal{R}(i+k))$，假設一次的時間複雜度是 $\mathcal{O}(F)$，那總體的複雜度就是 $\mathcal{O}(\log(n)F)$。如果我們用一般的排序法 $\mathcal{O}(F) = \mathcal{O}(n \log n)$，但因爲我們要排的東西都是兩個 $[0, n)$ 中的整數對，可以使用 Radix sort 加速到 $\mathcal{O}(n)$ ，因此總共的複雜度就是 $\mathcal{O}(n \log n)$ 了。

最後我們只剩 $\mathcal{H}$ 要求了。複習一下 $\mathcal{H}$ 的定義，$\mathcal{H}(i)$ 代表 $S_B(\mathcal{SA}(i))$ 跟 $S_B(\mathcal{SA}(i-1))$ 前幾個字元相同。我們先列出他的一些性質:

**定理：** 我們定義 $d(A_1, A_2)$ 表示 $A_1, A_2$ 的前幾個字元相同。對於一個長度爲 $n$ 的字串 $B$ 有:

- 假設 $\mathcal{R}(i) < \mathcal{R}(j)$，則 $d(S_B(i), S_B(j)) = \min_{\mathcal{R}(i) < k \le \mathcal{R}(j)} \mathcal{H}(k)$ (1.14)
- 如果 $\mathcal{H}(\mathcal{R}(i)) = k$，則字典序比 $S_B(i)$ 小的字串最多只會跟 $S_B(i)$ 前 $k$ 個字元一樣。 (1.15)
- $\mathcal{H}(\mathcal{R}(i)) \ge \mathcal{H}(\mathcal{R}(i-1)) - 1$ (1.16)

其實按照字典序排序的結果就是，長的像的後綴會被排在一起! 我們先證 (1.14) ，假設 $\mathcal{R}(i) = k_1, \mathcal{R}(j) = k_2$，注意到 $\mathcal{H}(k_2)$ 表示 $S_B(\mathcal{SA}(k_2)), S_B(\mathcal{SA}(k_2 - 1))$ 前幾個字元一樣，$\mathcal{H}(k_2 - 1)$ 表示 $S_B(\mathcal{SA}(k_2 - 1)), S_B(\mathcal{SA}(k_2 - 2))$ 前幾個字元一樣...，一直到 $\mathcal{H}(k_1 + 1)$ 表示 $S_B(\mathcal{SA}(k_1 + 1)), S_B(\mathcal{SA}(k_1))$ 前幾個字元一樣。但別忘了

$$S_B(\mathcal{SA}(k_1)) = S_B(\mathcal{SA}(\mathcal{R}(i))) = S_B(i), \quad S_B(\mathcal{SA}(k_2)) = S_B(\mathcal{SA}(\mathcal{R}(j))) = S_B(j)$$

因此 $S_B(i), S_B(j)$ 至少前 $h = \min_{k_1 < k \le k_2} \mathcal{H}(k)$ 個字元一樣! 但 $h = \min_{k_1 < k \le k_2} \mathcal{H}(k)$，表示一定有一次 $S_B(\mathcal{SA}(k'))$ 和 $S_B(\mathcal{SA}(k' - 1))$ 的第 $h$ 個字元不一樣。而之後這第 $h$ 個字元也不可能一樣，因爲我們是照字典序排序的，接下來只會越來越小! 因此 (1.14) 是對的! (1.15) 是 (1.14) 的一個直接的結果，而 (1.16) 我們用圖來說明

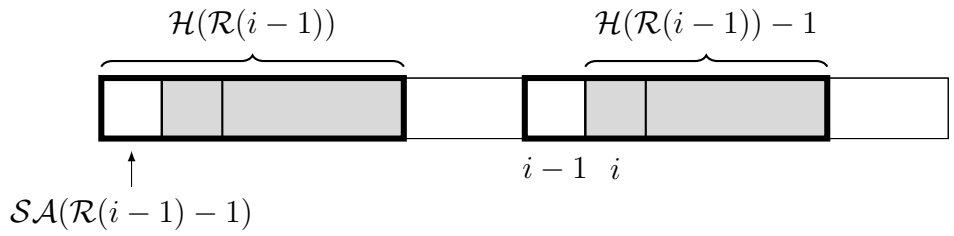


Figure 1.7

如 Figure 1.7，假設 $\mathcal{H}(\mathcal{R}(i-1))=k$，且不妨假設 $k>1$，也就是說 $S_B(i-1)$ 和他照字典序排序後的前一號 $j=\mathcal{SA}(\mathcal{R}(i-1)-1)$ 的前 $k$ 個字元一樣! 則 $S_B(j+1)<S_B(i)$ 因爲 $S[j]=S[i-1]$，且 $S_B(j+1)$ 至少會和 $S_B(i)$ 有 $k-1$ 個字元一樣。由(1.15)我們得知

$$\mathcal{H}(\mathcal{R}(i))\geq k-1=\mathcal{H}(\mathcal{R}(i-1))-1$$

### 1.8.1 Suffix Array 之應用

第一個 Suffix Array 主要的應用是處理匹配問題，而且他往往比前面介紹的演算法更有彈性!

我們用 $B$ 去匹配 $A$ 其實就是在問 $B$ 是不是 $A$ 的一個子字串。注意到一個子字串又一定是某個後綴的前綴，所以我們要做的事情就是找出 $A$ 和 $B$ 最像的後綴，也就是**問如果我們將 $A$ 的後綴排序好後，$B$ 在其中的哪個位置？**而排序的部分剛好就是 Suffix Array 做的事情。而找出 $B$ 在其中的位置，很顯然會想到二分搜。這樣的確只需要比較 $\mathcal{O}(n\log n)$ 次，不過因爲後綴的長度可以到 $\mathcal{O}(n)$，所以整體下來會是 $\mathcal{O}(n\log n)$。

一個優化是，假設我們已經將 $B$ 和一個 $A$ 的後綴 $S_A(i)$ 匹配了前 $k$ 個字元，現在我們要匹配另一個後綴 $S_A(i')$，注意到如果 $B$ 可以和 $S_A(i')$ 匹配超過 $k$ 個字元，那麼 $S_A(i),S_A(i')$ 至少要前 $k$ 個字元相同 (更甚者，要恰好前 $k$ 個字元相同)。也就是 $d(S_A(i),S_A(i'))\geq k$。於是 (1.14) 告訴了我們

$$d(S_A(i),S_A(i'))=\min_{\mathcal{R}_A(i)<k\leq\mathcal{R}(i')}\mathcal{H}(k)$$

這其是一個 RMQ 問題，如果我們用如**線段樹**等資料結構加速，每一次的查詢只需要 $\mathcal{O}(\log n)$ 甚至 $\mathcal{O}(1)$ [5]。而如果查詢的結果是 $d(S_A(i),S_A(i'))<k$，我們就不需要在重新匹配了，而如果 $d(S_A(i),S_A(i'))=k$ 時我們直接繼續從第 $k$ 個字元匹配。這樣成功匹配的數量 (也差不多會是匹配的次數) 永遠只會增加，因此匹配的次數不會超過 $\mathcal{O}(|B|)$，全部的複雜度會是 $\mathcal{O}(|A|\log|A|+|B|)$。

第二個應用可以處理字串中的計數問題，我們先考慮一個最基本的問題

[5]因爲我們不需修改操作，其實 RMQ 是可以做到 $\mathcal{O}(1)$ 的，當然，不是很好實做。

**例題 1-2** 不同的子字串數 經典問題

給你一個字串 $A$，問 $A$ 不同的子字串有幾個。比如 $A =$ "abab"，不同的子字串有 "a", "b", "ab", "ba", "aba", "bab", "abab"，共 7 種。

這邊還是用到一個子字串一定是一個後綴的前綴。如果一個子字串出現在多個後綴，我們在字典序最小的後綴計算，你會發現，$\mathcal{H}(i)$ 就會剛好是 $\mathcal{R}(i)$ 的後綴的前綴中，有多少已經在字典序比他小的後綴出現過。因此

$$\binom{n}{2} - \sum_{i=1}^{n-1} \mathcal{H}(i), \quad \text{where } n = |A|$$

恰好就是答案。

## 1.9 Suffix Automaton*

後綴字動機 (Suffix Automaton) 是現在很熱門的一個後綴結構。雖然大部分 Suffix Automaton 可以做到的事情 Suffix Array 也可以做到，但其程式碼精簡，不失爲另一個不錯的解題武器。

**定義：** 一個字串 $B$ 的後綴自動機是一個自動機，他接受一個字串 $s$ 若且唯若 $s$ 是 $B$ 的一個後綴。

如下圖 1.8 就是字串 "abcbc" 的後綴自動機。

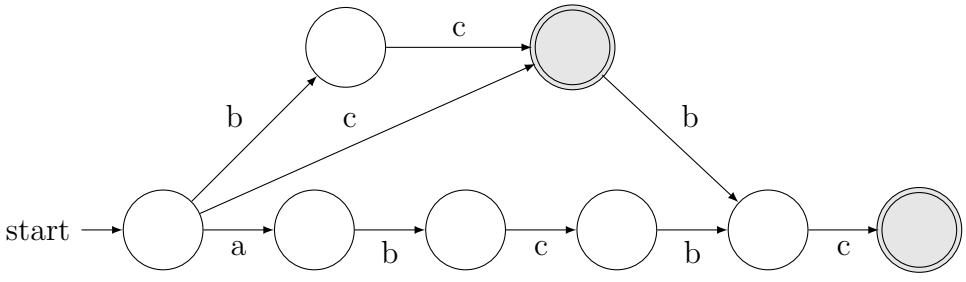


Figure 1.8

至於如何構造? 最基本的想法就是把所有 $B$ 的**子字串**都當作自動機上的一個 state，然後一個一個 state 慢慢建邊。這樣當然正確性是被保證的 (畢竟所有後綴的前綴一定是原字串的一個子字串)，但這樣 state 有 $\mathcal{O}(N^2)$ 個，需要再改進。

而減少 state 的唯一方法就是把等價的 state 合併。像 Figure 1.8 中 state 數量也只有 8 個而非 $\binom{6}{2} = 15$ 個那麼多，觀察後可以發現在圖中 "abcb", "bcb", "cb" 三個字串是屬於同一個 state 的!

仔細思考一陣後可以發現，只要結束位置相同的兩個子字串，他們的 state 就會是等價的，我們便可以將之合併! 我們先定義結束集合

**定義：** 給定兩個字串 $B, T$，我們定義 $T$ 在 $B$ 上的結束集合 $\mathcal{E}_B(T)$ 爲

$$\mathcal{E}_B(T) = \{i \ : B[i - |T| + 1, i] = T\}$$

比如說對 "abcbc" 而言，$\mathcal{E}(\text{"bc"}) = \{2, 4\}$，因爲 "bc" 在字串中出現兩次並且分別在位置 $2, 4$ 結束。

**定理：** 給定兩個字串 $T_1, T_2$，如果 $\mathcal{E}_B(T_1) = \mathcal{E}_B(T_2)$，記作 $T_1 \equiv_B T_2$，則

- 對任何字串 $A$，$\mathcal{E}_B(T_1A) = \mathcal{E}_B(T_2A)$，也就是 $T_1A \equiv_B T_2A$ (1.17)
- 對任何字串 $A$，如果 $T_1A$ 是 $B$ 的後綴，則 $T_2A$ 也是 $B$ 的後綴 (1.18)

(1.17) 稍爲推敲即可得出，而 (1.18) 是前者的一個直接推論，並且 (1.18) 也告訴我們，只要 $T_1 \equiv_B T_2$，他們的確可以看作是等價的 state ！ 如在前面的例子，$\mathcal{E}(\text{"bc"}) = \{2, 4\} = \mathcal{E}(\text{"c"})$，而他們在 Figure 1.8 上也的確會走到同一個 state 。
但這樣可以幫我們省下多少 state 呢? 我們還需要 $\mathcal{E}$ 的一個很重要的性質:

**定理：** 給定兩個字串 $T_1, T_2$，$|T_1| < |T_2|$，那麼不是 $\mathcal{E}(T_1) \cap \mathcal{E}(T_2) = \varnothing$，就是 $\mathcal{E}(T_1) \supset \mathcal{E}(T_2)$。並且如果是後者，$T_1$ 會是 $T_2$ 的一個後綴。 (1.19)

事實上從這個定理我們就可以知道 state 的數量最多只會有 $2n - 1$ 個。我們在這邊整理一下:

**定理：** 我們可以用 $\equiv_B$ 將 $B$ 的所有子字串分類，也就是 $T_1, T_2$ 被分在同一類若且唯若 $T_1 \equiv_B T_2$ 。假設我們將 $B$ 的所有子字串分成 $\alpha_1, \alpha_2, \cdots, \alpha_k$，則

- 如果 $t_1, t_2 \in \alpha_i$，則必有其中一個爲另一個的後綴。 (1.20)
- $k$ 恰好是後綴自動機上非起始節點的數量，每一個 $\alpha_i$ 可以看做是後綴自動機上的一個 state 。
- $k < 2|B| - 1$。

在後綴自動機還有一個很重要的指標，Fail link $\mathfrak{F}$。假設一個字串 $T_0$ 是 $B$ 的子字串，那我們如果不斷把 $T_0$ 的第一個字串去掉變成 $T_1, T_2, \cdots$，那一開始結束集合可能不變，但總會有一個 $T_k$ 結束集合開始改變了。比如說 $B =$ "ababa" ，"abab","bab" 的結束集合都是 $\{4\}$ ，但 "ab" 的結束集合變成 $\{2, 4\}$ 了。
假設 $T_0$ 所屬的集合稱作 $\alpha_1$ ，$T_k$ 所屬的集合稱作 $\alpha_2$ ，則令 $\mathfrak{F}(\alpha_1) = \alpha_2$。

**定理：** 我們定義 $\mathcal{E}(\alpha_i) = \mathcal{E}(t_1)$，對於任意一個 $t_i \in \alpha_i$。(反正任何一個的結束集合都一樣)。
定義 $\max(\alpha_i), \min(\alpha_i)$ 爲 $\alpha_i$ 裡最長/短的字串 (因爲 (1.20) 所以恰好只會有一個)。
我們用 $\alpha(t)$ 來表示字串 $t$ 所屬的集合，如果 $s_i = \min(\alpha_i)$，我們定義

$$\mathfrak{F}(\alpha_i) = \alpha(S_{s_i}(1))$$

也就是把 $s_i$ 的第一個字元去掉後所屬的類別。

現在我們要考慮怎麼樣把這一個後綴自動機建出來。一個做法是我們從空字串開始，一個一個字元慢慢加上去。因爲 $\alpha_i$ 就是後綴自動機上的點，所以我們要先了解 $\mathcal{E}(t)$ 會怎麼變動。

**定理：** 給定 $B$，現在如果在 $B$ 的後面加一個字元 $c$ 使得 $B' = Bc$，則如果 $T$ 是 $B$ 的一個子字串，有

- 如果 $T$ 不是 $B'$ 的一個後綴，那 $\mathcal{E}_{B'}(T) = \mathcal{E}_B(T)$。
- 如果 $T$ 是 $B'$ 的一個後綴，那 $\mathcal{E}_{B'}(T) = \mathcal{E}_B(T) \cap \{n\}$，其中 $n = |B| = |B'| - 1$。

這個定理告訴我們，在一個字串後面加一個字元，結束集合不會有太大的變動！會變的只有 $B'$ 的後綴，不超過 $n$ 個，而且分在同一類的可以一起處理！現在的問題就只有**怎麼找出所有後綴存在的集合了**。這時候就可以好好利用 $\mathfrak{F}$ 了！

**定理：** 給定一個字串 $B$，$B'$ 爲在 $B$ 後面加一個字元 $c$ 所成的字串，即 $B' = Bc$ 。

- 假設 $T'$ 是 $B'$ 的後綴，則 $B = Tc$，其中 $T$ 是 $B$ 的後綴。 (1.21)
- 令 $\beta_0 = \alpha(B)$ ，也就是整個字串 $B$ 所屬的集合，如果 $\mathfrak{F}(\beta_i) = \beta_{i+1}$ ，則所有 $B$ 的後綴都會在 $\{\beta_0, \beta_1, \cdots, \beta_h\}$ 的其中一個。 (1.22)

將 (1.21) 和 (1.22) 合起來用，我們就得出一個找到 $B'$ 所有後綴的方法了！我們只要檢查 $\{\beta_0, \beta_1, \cdots\}$ 有沒有連出 $c$ 的邊，有的話所有那些點就是 $B'$ 的後綴了。最後我們只要考慮這些點的結束位置會怎麼變就可以了。

**定理：** 給定一個字串 $B$:

- 現在如果在 $B$ 的後面加一個字元 $c$ 使得 $B' = Bc$，則如果 $T$ 是 $B'$ 的後綴，且 $T$ 是 $B$ 的子字串，那 $T$ 的所有後綴也是 $B$ 的子字串。
- 存在 $k$ 使得所有 $j \geq k$，$S_B(j)c$，也就是在 $S_B(j)$ 後面加一個字元 $c$，都是 $B$ 的子字串。而所有 $j < k$，$S_B(j)c$ 都不是 $B$ 的子字串。 (1.23)

也就是說會有一個臨界的後綴 $S = S_B(k)$，使得比他短的後綴加上 $c$ 後，會是原本 $B$ 的子字串。不妨設 $m = |S|$。令 $\beta_k$ 是第一個有連出 $c$ 的邊的點，由 (1.23) 我們知道 $\beta_i$ 有連出 $c$

的邊若且唯若 $i \geq k$。因此可令 $\gamma_i = \beta_i c$，也就是從 $\beta_i$ 連出 $c$ 的邊走到的點。我們現在只要考慮 $\mathcal{E}(\gamma_i)$，會怎麼變化就可以了！有兩種情形

Case 1. $|\max(\gamma_i)| = |\max(\beta_i)| + 1$，假設 $\max(\beta_i) = T, \max(\gamma_i) = U$，這代表 $Tc = U$ ，而 $\gamma_i$ 裡的字串都是 $U$ 的後綴，也就是**所有 $\gamma_i$ 中的字串都是 $B' = Bc$ 的後綴**！因此他們的結束位置都全部加一個 $n$，同時改變，不需要分家了。

Case 2. $|\max(\gamma_i)| > |\max(\beta_i)| + 1$，與前面的 Case 相反，這表示**並非所有 $\gamma_i$ 中的字串都是 $B' = Bc$ 的後綴**！這時我們勢必得把 $\gamma_i$ 拆成兩個新的點 $\gamma_{i1}, \gamma_{i2}$ 了。

因此我們必需維護兩個東西，$\mathfrak{F}(\alpha_i)$ 和 $\mathcal{M}(\alpha_i) = |\max(\alpha_i)|$。如果你仔細考慮每個 Case 這兩者該如何變化，可以得出以下構造後綴自動機的演算法:
我們從一個只有 initial state $\alpha_0$ 的自動機開始，每個點維護 $\mathfrak{F}(\alpha_i), \mathcal{M}(\alpha_i) = |\max(\alpha_i)|$，並且記好結束的 state $e$ 是哪一個 (一開始 $e = \alpha_0$)，逐一插入字元。假設這一次插入的字元爲 $c$：

1. 加入一個新點 $\alpha_{k+1}$
2. 從 $e$ 開始，令 $u = e$。看看 $u$ 有沒有 $c$ 的出去的邊，如果沒有，則加一條 $c$ 從 $u \to \alpha_{k+1}$ 的邊。令 $u \leftarrow \mathfrak{F}(u)$ 繼續嘗試，直到有或者 $u = \alpha_0$ 爲止。
3. 如果 $u$ 沒有 $c$ 連出去的邊，則 $u = \alpha_0$，加一條 $c$ 從 $\alpha_0 \to \alpha_{k+1}$ 並令 $\mathfrak{F}(\alpha_{k+1}) = \alpha_0$ 後結束。
4. 現在假設 $u$ 從 $c$ 連出去的邊到 $v$。
5. 如果 $\mathcal{M}(v) = \mathcal{M}(u) + 1$，(Case 1.) 則讓 $\mathfrak{F}(\alpha_{k+1}) = v$，結束。
6. 如果 $\mathcal{M}(v) \neq \mathcal{M}(u) + 1$，(Case 2.) 我們需要把 $v$ 拆成兩個點。因此我們複製一個點 $v'$ 使得
   - $v'$ 要保持 $v$ 的所有向外的邊。
   - $\mathfrak{F}(v') \leftarrow \mathfrak{F}(v), \quad \mathfrak{F}(v), \mathfrak{F}(\alpha_{k+1}) \leftarrow v'$。
   - 所有 $u$ 往 $\mathfrak{F}(u)$ 走到的節點如果有邊指向 $v$ 要改指到 $v'$。因此我們不段令 $u \leftarrow \mathfrak{F}(u)$ 去檢查，直到 $u = \alpha_0$ 爲止。

最後更新 $e = \alpha_{k+1}$，增加 $k$ 後重複步驟直到建完整個字串。
以下是一個 Suffix Automaton 的範例程式碼:

naive-matching **程式碼片段 1-5**

```
vector<State> vec;
int root, tail;
void extend(char c){
  int u = tail;
  vec.push_back( State(vec[u].minv + 1) );
    // State(a) : New state with minv = a
    np = vec.size() - 1; // np : new point
  for ( ; u && vec[u].child[c] == 0 ; u = vec[u].flk )
    vec[u].child[c] = np;  // Step 2.
  if (u == 0){  // Step 3.
    vec[np].flk = root;
```

```
  } else {                                                    ▲
    if (vec[ vec[u].child[c] ].minv == vec[u].minv + 1){
      vec[np].flk = vec[u].child[c]; // Step 4.
    } else {
      int v = vec[u].child[c];
            vec.PB(vec[v]);
            int vp = vec.size() - 1;
      vec[vp].minv = vec[u].minv+1;
      vec[v].flk = vec[np].flk = vp;
      for ( ; u && vec[u].child[c] == v; u=vec[u].flk)
        vec[u].child[c] = vp;
    }
  }
  tail = np;
}
```

Suffix Automata 的應用非常多，從匹配到計數等等。另外 Suffix Automata 有一個很重要的性質:

**定理：** 一個 $B$ 的後綴自動機上從起始點 $s$ 到任何一點 $v$ 的任何一條路徑都恰好對應到一個 $B$ 的子字串。

因此會有各種在 Suffix Automata 上 DP 的花招，就留給讀者自行探討了。

## 1.10 習題

### 1.10.1 String Matching

**習題 1-1** ABA — 經典問題

給定字串 $S$，求出最長的 $A$ 使得 $S = ABA$。$[\mathcal{O}(|S|)]$

**習題 1-2** Periods of Words — POI XIII

給定字串 $S$，求出最長的 $A$ 使得 $A$ 是 $S$ 的前綴且 $S$ 是 $AA$ 的前綴。$[|S| \leq 10^6]$

**習題 1-3** Template — POI XII

給定字串 $S$，求出最短的 $A$ 使得 $A$ 可以覆蓋 $S$ 。比如說 "abaabaab" 可以用兩個 "abaab"，分別開頭在 $0, 3$ 蓋住。$[|S| \leq 10^6]$

**習題 1-4** 近似匹配 — 經典問題

給定字串 $A, B$，以及一個整數 $k$，求出所有 $B$ 在 $A$ 中 $k$ 幾乎匹配的位置。我們稱

$A[i, i+n-1], B[j, j+n-1]$ 是 $k$ 幾乎匹配如果 $\{A[i+x] \neq B[j+x],\ 0 \leq x \leq n-1\}$ 的個數不超過 $k$ 個。[ $\mathcal{O}(k|A| + |B|)$ ]

**習題 1-5** k-口吃子字串 TIOJ 1735 / Kelvin

定義一個 $k$-口吃字串爲某一個長度爲 $k$ 的字串重複兩次的字串。如 "abcabc", "aaaaaa" 都是 3-口吃字串。給一個字串 $S$ 和 $k$，請問有多少 $(i, j)$ 滿足 $S[i, j]$ 是 $k$-口吃字串。[ $|S| \leq 10^5$ ]

**習題 1-6** KMP and Z value 經典問題

給你一個字串的 fail function $\mathcal{F}$ 和他的長度 $n$ (也就是說你並不知道原本的字串)，請求出他的 Z value $Z$。反之亦然，給你一個字串的 Z value $Z$，請求出他的 fail function $\mathcal{F}$。[ $\mathcal{O}(n)$ ]

**習題 1-7** Massacre at Camp Happy TIOJ 1725

定義兩個字串 $A, B$ 是 $k$-幾乎相同如果把 $A$ 的前 $k$ 個字元搬到最後面，那兩者恰相差一個字元，給你 $A, B$，求出所有的 $k$ 使得他們是 $k$-幾乎相同。[ $|A| = |B| \leq 10^6$ ]

**習題 1-8** 最長回文子字串 經典問題

給一個字串 $A$，求出他最長的一個回文子字串。[ $\mathcal{O}(|A|)$ ]

這題可以用類似 Z-algorithm 的方法。我們先考慮如何求最長的**奇數**回文子字串。令 $\mathcal{Z}(i)$ 表示以 $i$ 爲中心最長的回文字串爲 $A[i - \mathcal{Z}(i), i + \mathcal{Z}(i)]$。現在假設 $(L, R) = (i - \mathcal{Z}(m), i + \mathcal{Z}(m))$, 即 $A[L, R]$ 是一個以 $a[m]$ 爲中心的回文。如果 $j \in [m+1, R]$，考慮 $j' = 2m - j$，即 $j$ 對 $m$ 的反射點。現在與 Z-algorithm 類似，令 $L' = j - \mathcal{Z}(j')$，分別考慮 $L' > L, L' = L, L' < L$ 三種情況。

但此方法僅能求得奇數的回文長度。因此我們把原字串相鄰的兩個字元都插入一個不在字元集 $\Sigma$ 中的字元，如 $A =$ "abbab" $\rightarrow A' =$ "a$b$b$a$b"，這樣所有回文的長度都變成奇數了。

**習題 1-9** Anti-hash text Codeforces Zlobober's blog

在 Hashing 實作上，因爲 C++ 的模運算 % 是非常花時間的，因此有一種 Lazy hash 的方法是直接 $\bmod 2^{32}$ 或是 $\bmod 2^{64}$，這樣直接在 `int`, `long long` 下做運算就可以了。但這樣做有其風險在，以下說明其原因。

不妨假設此 Hash function 爲

$$f(A) = \sum_{i=0}^{n-1} a_i p^{n-1-i} \bmod q$$

其中 $p$ 爲一奇數 (否則此 Hash function 無意義)，$q = 2^m$，並且 "a", "b" 轉換成數字後

分別爲 $\alpha, \alpha+1$ 。此時令

$$A_0 = \text{"a"}, \quad A_j = \widetilde{A}_{j-1} A_{j-1}$$

其中 $\widetilde{A}$ 表示把字串 $A$ 中的 "a" 換成 "b"，"b" 換成 "a"。如:

$$A_1 = \text{"ba"}, \quad A_2 = \text{"abba"}, \quad A_3 = \text{"baababba"}, \quad \cdots$$

現在考慮 $A_k, \widetilde{A}_k$，注意這兩個字串恰好是 $A_{k+1}$ 的前後半且 $A_k \neq \widetilde{A}_k$。請證明

$$\begin{aligned} f(\widetilde{A}_k) - f(A_k) &= p^0 - p^1 - p^2 + p^3 - p^4 + p^5 + \cdots + (-1)^{k-1} p^{2^k-1} \\ &= \sum_{i=0}^{2^k-1} b_i p^i \quad = S \end{aligned}$$

其中 $b_i \in \{1, -1\}$，且如果 $i$ 的二進位表示有偶數個 1 ，則 $b_i = 1$，否則 $b_i = 0$。並且有

$$\sum_{i=0}^{2^k-1} b_i p^i = (p-1)(p^2-1)(p^4-1)\cdots(p^{2^{k-1}}-1)$$

注意到 $(p^{2^r}-1)$ 可以被 $2^{r+1}$ 整除 (Why?) ，所以 $S$ 可以被 $2^{1+2+\cdots+k} = 2^{k(k+1)/2}$ 整除。因此只要 $2^{k(k+1)/2}$ 被 $2^m$ 整除，也就是 $\frac{k(k+1)}{2} > m$，$A_k$ 和 $\widetilde{A}_k$ 的 Hash value 就會相同。因此可以用一個長度爲 $2^{\mathcal{O}(x)}$ 的字串構造出一個使 $q = 2^{\mathcal{O}(x^2)}$ 的 Hash function 失效的反例，並且與 $p$ 的選擇無關。

**習題 1-10** 二維匹配 經典問題

給你一個 $R_A \times C_A$ 的二維字串 $A$ ，問一個 $R_B \times C_B$ 的二維字串 $B$ 有沒有出現在其中? [ $\mathcal{O}(R_A C_A + R_B C_B)$ ]。

一個巧妙的做法可參考 Baker-Bird Algorithm，令外也有用 Hash 的方法，令 $c = C_B$, $a_i[j] = f(A[i][j, j+c-1])$，原題等價於問 $f(B[0]), f(B[1]), \cdots f(B[R_B-1])$ 是不是等於
$a_i[j], a_{i+1}[j], \cdots, a_{i+R_B-1}[j]$。這是一個一維的匹配問題。詳細可參考去年的講義。

**習題 1-11** sed ACM ICPC 2013-2014 NEERC Northern Subregional

給你 $A, B$，求出字串 $X, Y$ 使得將 $A$ 中的 $X$ 取代爲 $Y$ 後會變成 $B$，並且 $|X| + |Y|$ 最短。取代的定義爲找 $X$ 出現在 $A$ 的第一個位置，假設 $A[i, j] = X$ ，立刻將 $A[i, j]$ 變成 $Y$ ，並從 $j+1$ 繼續找下一個 $X$ 直到沒有爲止。[ $|A|, |B| \leq 5000$ ]。

### 1.10.2 Suffix structure

**習題 1-12** 最小的後綴 經典問題

給一個字串 $A$，求出他字典序最小的後綴。[ $\mathcal{O}(|A|)$ ]

有不需要建出 Suffix Array 的方法。[6]

這有一些應用，如最小循環表示法。有時候我們會把一個狀態或物體用一個字串表示，並且可能旋轉需視爲相同的，也就是把字串前 $k$ 個字元接到後面視爲相同。這時候如果要比較兩個字串，可以先求他們的最小循環表示在進行比較。

**習題 1-13** 重複子字串 經典問題

給一個字串 $A$，求出他最長的一個重複的子字串，重複的部分可以重疊。比如說 $A =$ "cababababc"，最長的重複子字串爲 "abab"。[ $\mathcal{O}(\mathcal{SA})$ ] [7]
如果改成不能重複呢？[ $\mathcal{O}(|A|\log|A|)$ ]

**習題 1-14** 連續重複的子字串 經典問題

給一個字串 $A$，求出他的連續重複次數最多的子字串。比如說 $A =$ "cababababc"，"ab" 在子字串 "abababab" 中連續重複了 4 次。[ $\mathcal{O}(|A|\log|A|)$ ]

**習題 1-15** Binary Suffix Array ASC 40

給定一個 $n$ 個整數的序列 $a_1, a_2, \cdots, a_n$，找出一個 01 字串 $A$ 使得 $A$ 的後綴數組 $\mathcal{SA}$ 恰好是給定的序列，注意到有可能無解。[ $|A| \leq 3 \cdot 10^5$ ]

**習題 1-16** Lexicographical Substring Search SPOJ 7258

給一個字串 $A$，現在把 $A$ 的所有子字串列出，將重複的刪去後排序。對於 $Q$ 比詢問，輸出字典序第 $k$ 小的子字串。[ $|A| \leq 90000, Q \leq 500$ ]

**習題 1-17** Beautiful Substring NTUOJ 2200

給一個字串 $A$，把他所有不同的子字串找出來，令這個集合爲 $S$，接著每次會給一個字串 $B_i$，假設他所有不同的子字串的集合爲 $T_i$，就令 $S \leftarrow S \setminus T_i$，並輸出此時的 $|S|$，也就是 $S$ 裡還剩多少字串。[ $|A| \leq 2 \times 10^5,\ \sum|B_i| \leq 10^6$ ]

**習題 1-18** Incomparable Suffixes ASC 42

給一個字串 $A$，我們說兩個字串 $s, t$ 是**可分辨的** 若且爲若存在一個字串 $z$ 使得 $sz, tz$ 中恰好只有一個是 $A$ 的後綴，說兩個字串 $s, t$ 是**無可比較的** 若且爲若對所有 $s$ 的前綴 $x$，$t$ 的前綴 $y$，$x, y$ 都是可分辨的。現在請找出 $a_1, a_2, \cdots, a_n$ 滿足：

1. 所有 $a_i$ 都是 $A$ 的後綴。
2. $n$ 越大越好。
3. 在 $n$ 是所有可能的最大值下，$\sum|a_i|$ 越大越好。

[ $|A| \leq 10^5$ ]

[6] Hint: 雙指針
[7] $\mathcal{O}(\mathcal{SA})$ 表示建出後綴數組所需的時間。

## 1.11 Special Thanks

此分講義部分參考自去年的講義，特別感謝去年的編輯者。