Chapter 8 字串演算法（String Algorithm）

●Section 0 字元與字串（Alphabets and Strings）

字串是一種很特別的結構，他不只是一個多個字元所組成的序列而已，因為我們會把字串看成一個整體，當我們要對這些字串進行操作時，就會有很多問題出現。因此這章要講的便是和字串相關的資料結構或演算法。以下就來說一下基本定義：

1. 字元（Character）：

字元代表的是一個象徵符號（Symbols），例如：A,B,C,a,b,c,1,2,3……。

2. 字元集（Alphabet）：

字元集是由有限個字元組成的非空集合，通常以符號∑表示。

3. 字串（String）：

(1) 一個由同個字元集中的字元所構成的序列，稱為一個字串或單字（Word），而通常以符號 S 表示一個字串。

(2) n 個字元所表示出的字串 S 我們通常寫成 $S=a_1a_2a_3…a_n$（其中 $a_i∈∑$, i=1…n），且我們說字串 **S 的長度**（表示成 **|S|**）為 n，第 i 個字元 a_i 也可以表示成 S_i 或 S[i]。

(3) 所有由∑中字元構成的字串，表示成**∑***，而任一∑*的子集我們都可以稱作一種**語言（Language）**。

(4) 字串 $S=a_1a_2a_3…a_n$ 中的一段 $a_ia_{i+1}…a_j$（1 ≦ i ≦ j ≦ n）我們稱為 **S 的子字串（Substring）**，且我們可以表示成 $S_{i…j}$，或 S[i…j]。

(5) 字串 S 刪掉一些元素之後所剩下來的序列我們稱為 S 的**子序列（Subsequence）**，其與子字串有點相近，但子字串一定是子序列，子序列卻不一定是子字串。

(6) $S_{1…k}$ 我們稱為 **S 的 k 前綴（k-prefix）**，且可表示成 $S_{1…k} \sqsubset S$。

(7) $S_{n-k+1…n}$ 我們稱為 **S 的 k 後綴（k-suffix）**，且可表示成 $S_{n-k+1…n} \sqsupset S$。

(8) **字串大小的比較**：假設兩個字串 S[1..n]與 T[1..m]，且 l 是使 S[1.. l]=T[1.. l]最大的數且 l ≤ n,m，則

1. 若 l = n 且 l = m，則 S = T。
2. 若 l = n 且 l < m，則 S < T。
3. 若 l < n 且 l < m，則 S > T。
4. 若 l < n 且 l < m，則 S[l+1]與 T[l+1]的大小比較即為 S 與 T 的大小比較。

一般我們所說的字典順序即是字串大小順序。

4. 其他專有名詞定義（Other）：

(1)最長公共前綴（Longest Common Prefix, LCP）：兩個字串 u,v 的 LCP 我們通常表示成 **LCP(u,v)**，意思就是 Max{ l | u[1..l] = v[1..l] 且 l ≤ u 與 v 的長度 }，也就是從前面開始比對，看最長可以比到哪裡才不同。

●Section 1 模式匹配（Pattern Matching）

模式匹配是字串相關問題中很常見的一個而基本問題，基本問題敘述如下：

給定一個**主字串（Text）**S[1…n]，和**模板字串（Pattern）**T[1…m]，問你說 T 是否為 S 的一個子字串。

§3-1 樸素算法（Naïve String-Matching algorithm）

對於這類問題，我們能夠有一個很明顯很直觀的想法就是說，對於所有可能匹配到的地方去匹配看看，簡單來說對於所有的 k（k=[1,n-m+1]）檢查 S[k…k+m-1]=T，當然這種作法複雜度是 O(m(n-m+1))，對於競賽這種很容易出現刁鑽字串比對的情形，很容易就會花費很多時間，因此較不實用。但是對於字串字元出現情形非常隨機散亂時卻也不失為一種好方法。

```
NAIVE-STRING-MATCHER(S, T)
n ← length[S]
m ← length[T]
for k ← 1 to n – m + 1
    do if T[1 ... m] = S[k ... k + m -1]
          then print "Pattern occurs with shift"
```

§3-2 RK 算法（Robin-Karp algorithm）

Robin-Karp 算法是對於樸素算法的一種改進，利用類似 rolling hash 的想法，但是卻不必真的去寫一個 hash table，而是利用 rolling hash 能夠滾動的特性，將 S 中長度為 m 的子字串轉成一個 key，並且與 T 的 key 比較，如果 key 相同才進行比較，這樣子其實能夠減少許多不必要比較的時間。

而其預處理複雜度是 Θ(m)，主程序的複雜度是 O((n-m+1)+cm)（其中 c 為 key 值相同的數量），或者更進一步分析 O(c) = O(v+n/p)（v 為匹配成功的次數，n/p 為錯誤匹配的次數），則我們可以得到複雜度為 O(n + m(v+n/p))，假設**匹配次數不多**且 **p>n** 時，幾乎會是線性的！！

```
RABIN-KARP-MATCHER(S, T, |Σ|, p)
n ← length[S]
m ← length[T]
h ← |Σ|^(m-1) mod p
q ← 0
t_1 ← 0
for i ← 1 to m                ▹ Preprocessing.
    do q ← ( q×|Σ| + T[i] ) mod p
       t_1 ← ( t_1×|Σ| + S[i] ) mod p
for k ← 1 to n – m + 1          ▹ Matching.
    do if q = t_k
          then if T[1 ... m] = S[k ... k + m - 1 ]
                  then print "Pattern occurs with shift" s
       if k < n – m + 1
          then t_(k+1) ← (( t_k – S[i]×h )|Σ|+ S[k + m]) mod p
```

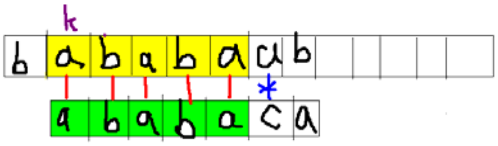
或許更進一步我們可以利用多個 p 值來降低 key 值相同的可能性，或許到 p 的個數夠多時，碰撞機率非常低，甚至不需要檢查是否匹配成功，利用所有 key 值相等即可判斷匹配成功。當然這樣做還是有某種程度的危險性就是。

§3-3 KMP 算法（Knuth-Morris-Paratt algorithm）

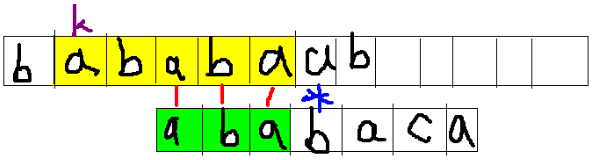
即便 RK 在理想狀況時能夠是線性的，但是最差複雜度仍然不能盡如人意，因此下面就要介紹一個有名的 KMP 算法，即使最差狀況預處理複雜度為 Θ(m)，主程序複雜度仍只有 Θ(n)。

我們可以觀察一下上面兩種算法，速度會慢有一個主要原因就是在於對於 S 之中的某些地方，T 對他去檢查次數太多次了，更白話的說法是：我們沒有將之前匹配的訊息「充分利

用」，我們沒有從匹配失敗的地方記取教訓，以致於某些地方不斷的與 T 進行比較，而造成時間緩慢。



(a)



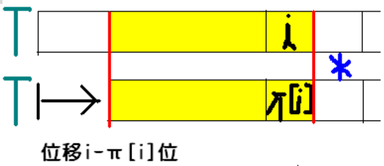
(b)

我們可以先觀察上面兩張圖，假設我們現在從 S 中的 k 開始與 T 做匹配，比對到 S[k+5]時發現匹配失敗了。根據樸素算法，我們應該會把 T 向前移動一格，並且重新從 k+1 開始做匹配。

然而我們其實可以發現到，因為在 k 的地方匹配了 5 個字，那接下來的四個字一定是 T[2..5]呀@@！假如我們有辦法先知道 T[2..5]根本沒辦法匹配 T，那我們幹嘛還傻傻的只位移一位繼續比對。又如果我們能知道哪個 l 能使得 T[l..5]能與 T 匹配成功的話，那直接位移到 l 的位置不就好了嗎（意思就是位移 l-1 位）？

因此我們可以把問題改成說，如果在某個地方 k，S 與 T 匹配到了 i 位，那下次我可以直接位移多少位，就不用再檢查 S[k..k+i-1]的任何一個地方了？

我們可以知道匹配了 i 位等價於知道接下來出現的字是 T[2..i]。我們現在要做的就是找一個**最小**的 l 使得 T[l..i]=T[1..i-l+1]且 2≤l。其實你可以發現到，這個問題就是問你說，**從 T 中的 i 位置往前延伸，最多可以往前幾位(< i)使得往前的這個位數是 T 的前綴**。



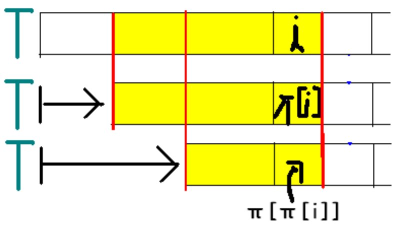
我們引入一個函數π，使上面所說的從 i 最多可以往前的位數為π[i]，則意思就是 T[i-π[i]+1..i] = T[1..π[i]]且π[i]<i。換句話說就是原本的位置 i，下次要匹配時，代表在 T 中的哪個位置？現在我們就是要求出所有的π[i]。

我們可以很明顯的知道π[1] = 0，假設我們現在知道π[1]...π[i]，那我們要如何從這些資訊去推出π[i+1]呢？（這個過程叫遞推，有點類似數學歸納法的過程）

很明顯的可以出現兩種狀況：

（1）T[i+1]=T[π[i]+1]：這時候很明顯可以多匹配一個，因此π[i+1]= π[i]+1。

（2）T[i+1]≠T[π[i]+1]：這時候就較麻煩一點，但你仔細觀察後會發現，既然 T[i+1]≠T[π[i]+1]，那下一個可能的匹配位置一定是 T[π[π[i]+1]，因此我們可以不斷迭代下去，直到π[π[..π[i]]]=0 或 T[i+1]=T[π[π[..π[i]]]+1]。




```
COMPUTE-PREFIX-FUNCTION(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
```

有了π函數，與 S 匹配就易如反掌了！

假設在某個時刻 S[i-k+1..i]=T[1..k]（即是從 S 的 i 往前與 T 匹配了 k 位），那我們要檢查 S[i+1]是否=T[k+1]：

（1）若 S[i+1] = T[k+1]，則 k=k+1 繼續匹配。

（2）若 S[i+1] ≠ T[k+1]，則 k 利用π[k]迭代下去，直到π[π[..π[k]]]=0 或 T[i+1]=T[π[π[..π[k]]]+1]。

一但在某個時候發現 k=m，即匹配成功（S[i-k+1..i]=T），如果要繼續進行匹配，之後可以先對 k 做一次迭代（k=π[k]）再繼續匹配下去。

```
KMP-MATCHER(S, T)
 1 n ← length[S]
 2 m ← length[T]
 3 π ← COMPUTE-PREFIX-FUNCTION(T)
 4 k ← 0                             ▷Number of characters matched.
 5 for i ← 1 to n                    ▷Scan the text from left to right.
 6     do while k > 0 and T[k + 1] ≠ S[i]
 7            do k ← π[k]            ▷Next character does not match.
 8        if T[k + 1] = S[i]
 9           then k ← k + 1           ▷Next character matches.
10        if k = m                    ▷Is all of T matched?
11           then print "Pattern occurs with shift" i - m
12                k ← π[k]            ▷Look for the next match.
```

會匹配之後我們來分析一下他的複雜度，很明顯的如果是考慮每個地方最多迭代幾次的話，那有可能迭代 O(m)次，想當然爾，這樣會變成 O(nm)。然而這是從部分來看，從整體來看的話，你會發現到每次 k 只可能+1，因此 k 最多只可能減少 n 次，第 7 行的 while 操作也只可能執行 n 次，因此最差就是 O(n)。預處理的估計也類似，所以只有 O(m)，因而 KMP 的整個複雜度為 O(n+m)。

§3-4 KMP 擴展算法（Knuth-Morris-Paratt Extend algorithm）

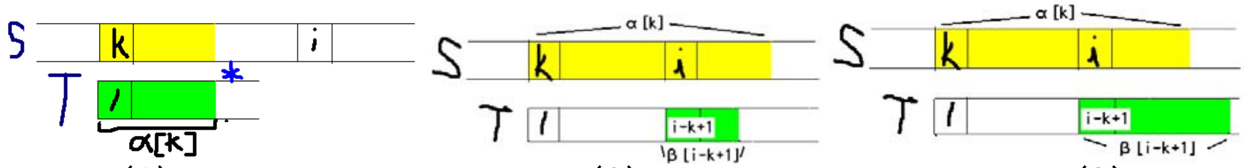
我們再看一下一般的 KMP 算法，發現到他是看 **S 中的每個位置往前最多可以與 T 匹配多少**，你會覺得這非常不符合人體工學呀(？)，因此我們決定要將 KMP 擴展成 KMPlayer(誤)。總之有人研究出了另外一個東西，我們把它稱為擴展 KMP（KMP-Extend）。

擴展 KMP 主要與 KMP 不同的就是他很符合人體工學，即是從 **S 的每個位置往後能匹配多少？**

我們定義一個 α 函數，為 $\alpha[i]$ = LCP(S[i..n],T) = $Max\{\ l \mid S[i...i+l-1] = T[1...l]\ \}$，意思就是 T 從 S[i]開始比較最大可以匹配到的長度。

如果我們能求出 α 函數，那就能求出所有 T 在 S 中匹配的位置了。

現在我們想嘗試利用與 KMP 類似的遞推法，藉由目前的 $\alpha[1]..\alpha[i-1]$，推出 $\alpha[i]$。要推出 $\alpha[i]$，其實可以使用之前的資訊來幫助求出之。首先我們可以分成幾種狀況（其中的 k 是介於 1 到 i-1 且能使 $k+\alpha[k]-1$ 最大的值。白話的說，就是能使我們對後面字串了解最多的位置）：



(1) (2) (3)

(1) $k+\alpha[k]-1 < i$：

如上圖(1)，很明顯的，因為先前的資訊不能提供我們任何關於 S[i]之後的訊息，因此要求出 $\alpha[i]$就必須從 S[i]開始慢慢向後比對直到求出與 T 的 LCP。

(2) $i \le k+\alpha[k]-1$：

這又可以分成兩種狀況，如上圖(2)(3)。可以很明顯看出，因為我們有的訊息已經超過 i，這代表說在 $k+\alpha[i]-1$ 之前的字串我們其實都已經知道會長什麼樣子了，即使要比對也不需要從 S[i]慢慢往後比。

但是我們這樣做有個前提，我們必須要知道 LCP(T[$i-k+1..m$],T)，有這個我們才能知道要比較直接從哪裡開始比較就好了。

因此我們在又要引入一個函數 β 函數，使 $\beta[i]$ = LCP(T[$i..m$],T)，可以發現到 β 的形式與 α 極像，因此我們可以先假設 β 已知，利用 β 求出 α。如果會從 β 推出 α，那從 β 推出 β 也不是難事。

以剛才的第二個條件下又能有兩種狀況：

1. $k+\alpha[k]-1 > i+\beta[i-k+1]-1$：

因為可能 LCP 長度在已知訊息範圍內，所以很明顯的，$\alpha[i]$只可能是 $\beta[i-k+1]$。（因為如果 $\alpha[i]$可能更大的話，$\beta[i-k+1]$也一定會更大，所以最大一定是 $\beta[i-k+1]$）

2. $k+\alpha[k]-1 \le i+\beta[i-k+1]-1$：

因為可能 LCP 長度超過目前已知訊息，但卻不能保證超過已知訊息後會與 LCP 相符合，所以必須要從 S[$k+\alpha[k]-1$]開始往後匹配，直到求出 LCP。

因此利用上述這樣遞推的過程就能推出 α，而上述的 β 雖然沒有說明，但其實與 α 一模一樣，因此就不再贅述。

```
KMP-MATCHER-EXTEND(S, T)
n ← length[S]
m ← length[T]
β ← COMPUTE-PREFIX-FUNCTION-EXTEND(T)
j ←0
while T[1 + j] = S[1 + j]
   do j ← j + 1
α[1] ← j
k ← 1
for i ← 2 to n
   do if i + β[i – k + 1] - 1 < k + α[k] – 1      ▹ Case 2-1
         then α[i] ←β[i – k + 1]
      else
         j ← Max( 0 , k + α[k] - i )              ▹ j = 0 -> Case 1
         while S[i + j] = T[1 + j]                ▹ j ≠ 0 -> Case 2-2
            do j ← j + 1
         α[i] ← j
         k ← i
```

```
COMPUTE-PREFIX-FUNCTION-EXTEND(T)
m ← length[T]
j ←0
while T[1 + j] = T[2 + j]
   do j ← j + 1
β[2] ← j
k ← 2
for i ← 3 to m
   do if i + β[i – k + 1] - 1 < k + β[k] – 1       ▷ Case 2-1
         then β[i] ←β[i – k + 1]
      else
         j ← Max( 0 , k + β[k] - i )               ▷ j = 0 -> Case 1
         while T[i + j] = T[1 + j]                 ▷ j ≠ 0 -> Case 2-2
            do j ← j + 1
         β[i] ← j
         k ← i
return β
```

同樣的如果只看單體，那有可能每次回圈都跑 O(m)次，不過整體來看的話，可以明顯發現，k 一定是不斷遞增的，要碼是在第一個 if 就判斷掉且 k 不變，要碼就是 k 越來越大，且每次要比對時的起點一定是不斷往後的，也就是對於每個 S 中的任一個位置頂多只會比對一次，因此 While 總共只會執行 n 次，是線性的。因此複雜度當然也就是 O(n+m)了。

§3-5 其他算法（Others algorithm）

其實匹配算法也有很多人研究過，也有著不同的結果，例如說 BM（Boyer-Moore algorithm）、z-value......等等，都是著名的算法，但是對於競賽來說，學一個 KMP 就已經很足夠，如果有興趣的話可以自己去專研。

這裡提供一個網址，說明許多字串匹配的算法，可以參考看看：

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

接下來將介紹一些特殊的字串問題或資料結構，對於處理許多字串的問題往往都能有高效的算法，當然不僅限於基本的模式匹配而已。

●Section 2 字典樹（Trie, Prefix Tree）

Trie 是一種特別的樹狀結構，就像我們平常查字典時一樣，對於查找一個字串的時候能夠依照前到後的順序查找。這種結構的好處是，因為他是依照字串的 prefix 來進行分支的，因此能夠迅速的查找與插入字串或支持一些對於 prefix 的操作（例如求兩個字串的 LCP）。

當然如果字符集 Σ 不是英文字母而是數字或其他序列也沒關係，只是為了方便以下都使用字母來說明。

§2-1 定義（Definition）

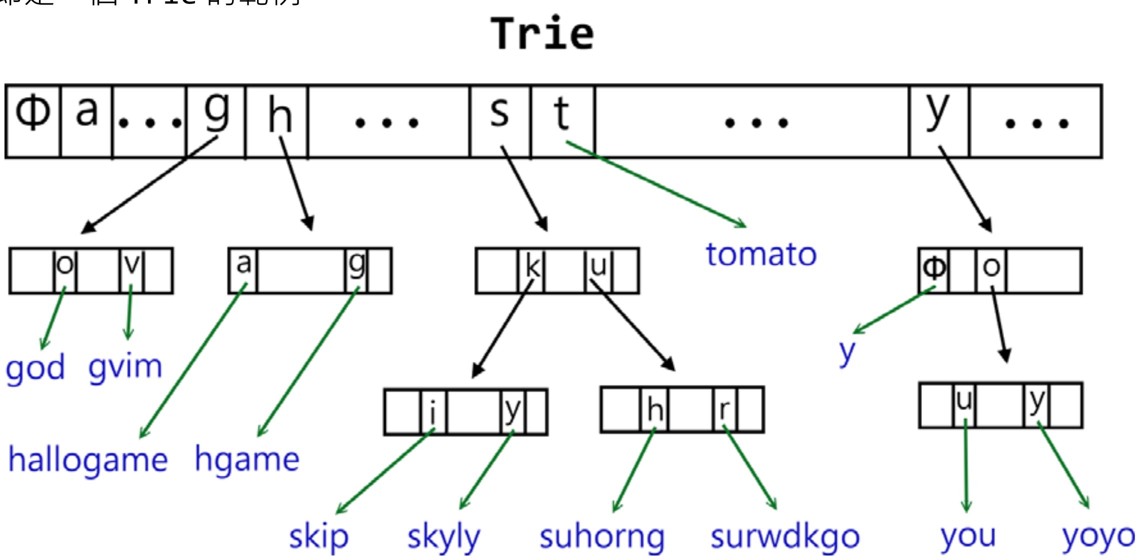
1. Trie 是一個|Σ|元樹，且有分成兩種節點：

（1）分支點（Branch Node）：全都是非葉節點，不儲存字串，但是有|Σ|個孩子，照順序是'A','B','C'...'Z'孩子，與一個 Φ 孩子（代表「空」）。

（2）資料點（Data Node）：全都是葉節點，且會儲存字串。

2. 對於一個第 i 層的節點，其α子樹中所有字串的第 i 個字都是 α (α = ’A’...’Z’,Φ)

下圖即是一個 Trie 的範例。



可以很明顯發現，樹最大的**高度 h** 只可能是**最大字串的長度+2**。

§2-2 實作 (Implementation)

在實作上我們會用一個結構來存取 Trie，並且每個 Trie 有**|Σ|+1 個兒子指標**與**一個 Count 值**和 **Data 指標**，兒子指標指向 Trie 或 NULL 或一個字串；Count 則是紀錄說這個節點以下有多少字串，如果 Count = 1 則其為資料節點，而這時候的 Data 指標就會指向一個字串。

a. 查找字串 (Search):

很明顯的，如果我們要查找字串 S，則只要在第 i 層的時候往子節點 S[i]走即可。

```
TRIE-SEARCH(S)
p ← root of Trie
for i ←1 to length[S]
   do if p->Count = 1                    ▹ p is a data node
         then return p->Data
      else if p->child[S[i]] = NULL      ▹ p is a branch node
            then return NULL
         else p ←p->child[S[i]]
```

不用說，因為只可能是往下走所以複雜度是 O(h)。

b. 插入字串 (Insertion):

同樣也是很明顯，如果我們要插入一個字串 S，我們一定是先查到他應該插入的位置。假如他可以插入的位置是空的，那當然就直接插入就好了。但若假如不是空的話，那就代表有一個字串 T 在那個位置上，因此這時候就必須往下擴張節點，直到找到這兩個節點的 LCP。

```
TRIE-INSERT(S)
p ← root of Trie
i ←1
while i ≤ length[S] and q = NULL
   do p->Count ← p->Count + 1
      if p->Count = 2                    ▹ p is a data node
        then T ← p->Data                 ▹ T is the collision string
```

```
            p->Data ← NULL
            while S[i] = T[i]
               do p->child[S[i]] ← new Node
                  p ← p->child[S[i]]
                  p->Count ← 2
                  i ← i + 1
            p->child[T[i]] ← new Node
            p->child[T[i]]->Data ← T
            p->child[T[i]]->Count ← 1
            p->child[S[i]] ← new Node
            q ← p->child[S[i]]
      else if p->child[S[i]] = NULL  ▹ p is a branch node
           then q ← p
           else p ←p->child[S[i]]
      i ← i + 1
q->Data ← S                          ▹ q is the inserted node
q->Count ← 1
```

明顯的複雜度是 O(h)。

c. 刪除字串（Deletion）:

刪除操作其實一般來說不太會用到，當然也是可以每次作標記之後再刪除，不過這裡要說的是動態刪除的方式。明顯的，首先我們會先找到要刪除的字串，而且在查找過程要順便紀錄所有會經過的結點（如果是遞回的話就不用紀錄了 XD）。

接下來沿著 Path 往上爬，每個結點的 Count 值都-1。因為一定存在至少一個 T 跟 S 的 LCP 最大，所以沿著原路回去的話，可能會發現離 S 最近的幾個分支點 Count=1，這樣會發生問題，因為照理來說分支點的 Count 一定大於 1，因此在回去的時候要一邊把下面結點 Release 掉，並把分支點變成資料點。

TRIE-DELETE(*S*)

```
p ← root of Trie
Stack ← EMPTY
for i ← 1 to Length[S]
   do push p onto Stack
      if p->Count = 1                ▹ p is a data node
         then break
      else if p->child[S[i]] = NULL  ▹ p is a branch node
           then break
           else p ←p->child[S[i]]
if Stack[top]->Count != 1 then EXIT  ▹ S does not find
i ← i - 1                            ▹ i is depth of the pointer
T ← NULL                             ▹ T is the string which has
p ← pop Stack                          maximal LCP with S
while Stack ≠ EMPTY
  do p ← pop Stack
     p->Count ← p->Count - 1
     if T = NULL
        then for each character j ∈ {Φ}∪Σ
                do if j ≠ S[i]
                      then T ← p->child[j]->Data
```

```
            RELEASE(p->child[S[i]])
        if p->Count = 1
            then p->Data ← T
                  RELEASE(p->child[T[i]])
        i ← i - 1
```

查找 T 的時間是 O(|Σ|)，而遞回上去刪點的複雜度是 O(h)，所以總的來說是 O(h+|Σ|)。

§2-3 應用與擴展（Application and Extension）

a. 排序（Sorting）:

因為他就像字典一樣，所以當我們在作 DFS 的時候其實就會照著字母的順序，所以當然可以輕易的做到字串排序了，複雜度則是 O(結點個數)=O(hn)=O(讀入字串的時間)。雖然複雜度還蠻理想的，不過一般來說我們不會為了排序就建一棵 Trie 吧 XD”。

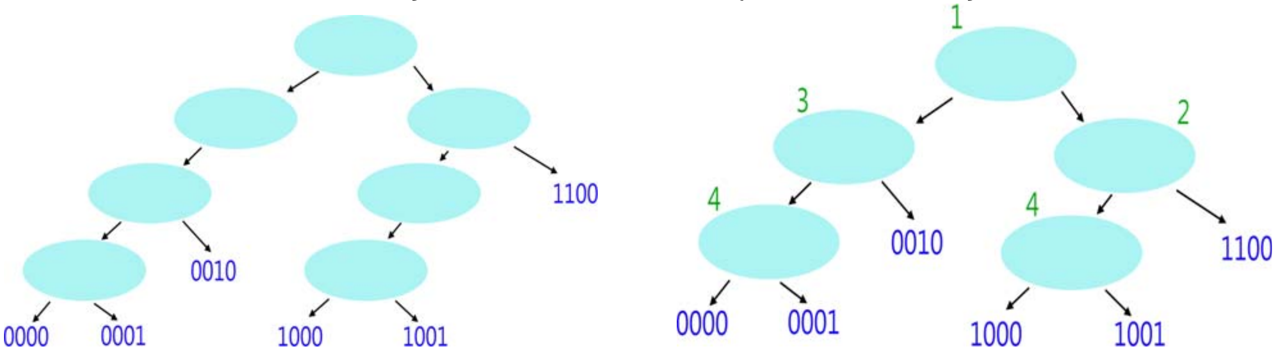
b. 求 LCP：

任兩個存在於 Trie 中的 LCP，很明顯的就是**最近共同祖先（Lowest Common Ancestor, LCA）**嘛，之後的課程將會講到，如果這棵樹是靜態的，那麼就利用 O(結點數量)的預處理的時間，回答每個 LCA 詢問只花 O(1)的時間。

c. 壓縮字典樹（Compressed Trie）:

因為我們會發現到，如果任兩個字串的 LCP 很長，那中間就必須會多出很多只有一個 child 的 branch node，因此我們可以把這些 branch node 壓縮起來，也就是給每個 branch node 標記一個 level 值，代表要往下查詢時直接看字串的第 level 個字就好了。當然這樣很多操作會更複雜些，因此就不在贅述，有興趣者就請自己參考相關書籍囉。

下左圖是一個 binary trie，右圖則是 Compressed binary trie。



是說還有一個叫做 **PATRICIA（Practical Algorithm To Retrieve Information Coded In Alphanumeric）**的東西，不過講師沒什麼研究，大概就是把 Trie 的空間與時間優化到極致的一種結構，有興趣者就自行參考相關書籍囉。

§2-4 小結（Conclusion）

Trie 的優點在於各項操作蠻快的，而且並不會說很難寫，如果對於 Prefix 相關的問題也能有一些好的複雜度解法，不過有一個很大的缺點是，因為結點上會有很多空指標，造成空間的負擔，因此使用時還是要謹慎估計與考慮才是。

●Section 3 後綴數組（Suffix Array）

了解了 Trie 之後我們可以知道，Trie 主要是對於不同字串之間尋求一種相關的結構，

然而對於只有單一個字串時，字串與其子字串的相關性問題卻不能有些很好的做法，而 KMP 所求的序列雖然對匹配很有用，但是卻不易擴展。因此我們現在就必須介紹一個叫做後綴數組的東西，他是一個關係到字串本身與其 Suffix 的序列，對於處理對特定字串相關的操作來說非常的有用，以下就來慢慢介紹：

§3-1 定義（Definition）

我們先假設對於一個特定的字串S[1..n]，其後綴S[i..n]定義為Suffix(i)。很明顯的因為兩個字串要相等一定長度一樣，所以對於一個S的所有Suffix都不可能相等。

由此我們可以定義後綴數組與名次數組。

（1）後綴數組：假設所有SA[i]是S的所有Suffix的一個排列，且使得Suffix(SA[1]) < Suffix(SA[2]) < ... < Suffix(SA[n])，則我們可以稱SA是S的後綴數組。其簡單的說，即是SA[i]為所有Suffix中第i大Suffix的位置。

（2）名次數組：名次數組Rank[i]其實就是SA的一個反元素，使得SA[Rand[i]] = i，也就是Suffix(i)在所有Suffix中的排名。

為了敘述方便我們再定義一些相關符號：

假設兩字串 u, v，則我們可以定義三種比較關係$<_k$、$>_k$、$=_k$：

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

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

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

簡單的說就是對於一個字串的前 k 個字做比較，特別要注意的是，如果沒有 k 個字元也沒關係，只要在那之前有出現不一樣就可以了。

§3-2 建立後綴數組（Implementation）

可以很簡單的想到一個方法，那就是直接把所有 suffix 列出來再排序，不過當然這樣很明顯的會是 O(n^2lgn)之類的，非常的沒有效率。因為我們沒有利用到 Suffix 之間的關係，因此下面就來介紹一個 Suffix array 的倍增算法：

● 倍增算法（Doubling Algorithm）：

首先我們可以先在 S 後面加一個字元Φ，並假設這個字元Φ < 字元集Σ中的所有字元。這樣做便可以使如果比較字串時可能出現未定義的情形變成有定義。

根據定義，我們可以得到下面三個性質：

性質 1　對 $k \ge n$，有 $\text{Suffix}(i) <_k \text{Suffix}(j) \Leftrightarrow \text{Suffix}(i) < \text{Suffix}(j)$

性質 2　$\text{Suffix}(i) =_{2k} \text{Suffix}(j)$
$\Leftrightarrow \text{Suffix}(i) =_k \text{Suffix}(j)$ 且 $\text{Suffix}(i+k) =_k \text{Suffix}(j+k)$

性質 3　$\text{Suffix}(i) <_{2k} \text{Suffix}(j) \Leftrightarrow \text{Suffix}(i) <_k \text{Suffix}(j)$ 或
（$\text{Suffix}(i) =_k \text{Suffix}(j)$ 且 $\text{Suffix}(i+k) <_k \text{Suffix}(j+k)$）

有了這三個性質，我們可以先定義 k-後綴數組 SA_k 與 k-名次數組 $Rank_k$：

k-後綴數組SA_k：假設所有SA_k[i]是S的所有Suffix的一個排列，且使得Suffix(SA[1]) $\le_k$ Suffix(SA[2]) $\le_k$ ... $\le_k$ Suffix(SA[n])，則我們可以稱SA_k是S的k-後綴數組。其簡單的說，即是SA[i]為所有Suffix中，以只比前k個為前提時，第i大Suffix的位置。要注意的一點是，k-後綴數組有可能是會出現相等的情形的，所以其實排列並不唯一，不過不影響我們繼續以下的操作。

k-名次數組 Rank$_k$：名次數組 Rank$_k$[i]其實就是在只比較前 k 個的前提下，Suffix(i)的排名。要注意的一點是，因為 Suffix(i)只基於比較前 k 個字元的關係，所以是有可能出現等於的情形的，如果對於 i, j 有 Suffix(i)=$_k$Suffix(i)，那便有 Rank$_k$[i]=Rank$_k$[j]。

其更好一點的說法是 Rank$_k$[i] = 1+ |{ j | Suffix(j) <$_k$ Suffix(i) }|。

而且很明顯的有了 SA$_k$ 我們很容易就能在 O(n)時間內求出 Rank$_k$。

有了上述的性質與定義，我們便可以嘗試使用類似遞推法的方法求出 SA 與 Rank。假設我們已經知道 SA$_k$ 與 Rank$_k$ 了，那我們便可以把所有的 Suffix$_{2k}$(i)想成有兩個數的有序數對（Rank$_k$[i], Rank$_k$[i+k]），那我們要求 SA$_{2k}$ 與 Rank$_{2k}$，其實就是要將 Suffix$_{2k}$(i)，做排序的動作，排完之後便可以在 O(n)的時間求出 SA$_{2k}$ 與 Rank$_{2k}$。但重點是要如何排序呢，如果是 Quicksort，則需要花 O(lgn)的時間，其實就一般情況來說算是不錯了，而且 Quicksort 也算好寫。不過通常必須要有 O(n)才比較足夠，因此這時候我們就會利用 Radix Sort，複雜度為 O(n)。而 SA$_1$ 跟 Rank$_1$ 要怎麼求呢？很簡單，就把字元排序就好了，因為這只會進行一次，所以 O(n)或 O(nlgn)都無妨。

會遞推之後我們可以知道，一但出現 SA$_m$ 與 Rank$_m$ 時（$m = 2^k \ge n$），我們便求出了 SA 與 Rank，因此總的複雜度是 O(nlgn)，如果是遞推過程是採用 Quicksort 則是 O(nlg^{2}n)。

DOUBLING-ALGORITHM(S)

```
sort suffix₁ by S
cnt ← 1
for i ← 2 to n
  do if S[ suffix₁[i-1] ] ≠ S[ suffix₁[i] ]
        then cnt ← cnt + 1
     rank₁[suffix₁[i]] ← cnt;
for k ← 1 to n step k
   do sort suffix2k by suffixk and rankk
      rank2k[ suffix2k[1] ] = 1;
      cnt ← 1
      for i ← 2 to n
        do if rankk[ suffix2k[i-1] ] ≠ rankk[ suffix2k[i] ] or
                rankk[ suffix2k[i-1]+k ] ≠ rankk[ suffix2k[i]+k ]
                then cnt ← cnt + 1
             rank2k[suffix2k[i]] ← cnt;
```

嘿嘿看起來很簡單對吧，其實真正寫的時候會遇到許多問題呢！

而且其實倍增算法有個特別的優化，在使用 Radix sort 時，我們會需要做兩次 Counting sort，而且 Counting Sort 一次大約要花 4*n 的時間（初始化、計算 Count 值、迭加 Count 值、計算 Suffix$_{2k}$），因此總共會需要 8*n 的時間。

但是其實不需要這麼麻煩，因為我們可以發現 Rank$_k$其實有一個特性，就是 Rank$_k$[SA$_k$[i]] ≤ i，且對於 i < j 一定會有 Rank$_k$[SA$_k$[i]] ≤ Rank$_k$[SA$_k$[j]]。

又 Count[p] = |{ i | Rank$_k$[i] ≤ p }|，所以對於所有可能的 i,i+1,...,j 能使 Rank$_k$[SA$_k$[i]] = ... = Rank$_k$[SA$_k$[j]] = p，可以發現 j = Count[p]。

求出 Count 值後，要依序填入 Suffix$_{2k}$。我們可以發現到，Rank$_k$[i]只要 i ≥ n-k+1，則 Rank$_{2k}$[i]中的相對順序不可能改變。因此 i ≥ n-k+1 的 Rank$_k$[i]是不可能重複的。

又會受到改變順序的影響的 $suffix_k[i]$，只可能是 $suffix_k[i] \leq n-k+1$ 的元素，因為這些元素要便成 $suffix_{2k}[i]$都需要看 $suffix_k[i+k]$的臉色。所以我們可以依 i 大到小的順序把 $suffix_k[i+k] \geq k$ 元素填入 $suffix_k[i]$的位置，因為 $suffix_k[i+k]$小的一定先填，所以一定會在後面。填完之後剩下的就是 $Rank_k[i]$中 $i \geq n-k+1$ 的元素了，只要照順序填上去就好，因為他們的值都不會重複的，而且因為不需要看 i+k 項的臉色，所以優先度一定是最高的。

此優化就是把上面的 code 的第八行改成以下幾行：

```
01 for i ← 1 to n
02    do Count[rank_k[suffix_k[i]]] ← i
03 for i ← n downto 1
04    do if suffix_k[i]-k ≥ 1
05          then suffix_2k[Count[Rank_k[Suffix_k[i]-k]]] ← Suffix_k[i]-k
06                Count[Rank_k[Suffix_k[i]-k]] ← Count[Rank_k[Suffix_k[i]-k]] - 1
07 for i ← n downto n-k+1
08    do if suffix_k[i]-k ≥ 1
09          then suffix_2k[Count[Rank_k[i]] ← i
```

因為每次只需要做 3*n 次，且第七行的回圈到最後總共只會做 n 次而不是 nlgn 次。因此每次能比原本做法快很多。而實際上這個優化當遇到可能需要用 O(n)建 SA 法，且 n 不大時，還能比 O(n)得更有效率許多。

● 三倍增算法（Skew Algorithm , Difference Cover modulo 3 , DC3）：

DC3 算法較倍增算法複雜很多，但他有 O(n)的複雜度，雖然當 n 不大時實行狀況不理想，或許是常數太大的原因。而且其不容易寫，對於競賽實用度來說不是很高。不過有興趣者還是可以自行觀看，在此就不說明。

下面有篇論文應該即是 DC3 的出處：

http://www.cs.helsinki.fi/u/tpkarkka/publications/jacm05-revised.pdf

§3-4 應用（Application）

如果想要充分發揮 Suffix array 的效果，我們還必須要有一個輔助工具，就是能在很快的時間內查出任兩個 Suffix 的 LCP。

我們先定義 LCP(i,j) = lcp(Suffix(SA[i]),Suffix(SA[j]))，意思就是第 i 大的 Suffix 跟第 j 大的 Suffix 的 LCP 長度。

而我們很明顯的會有兩個性質：

性質 1 LCP(i,j) = LCP(j,i)

性質 2 LCP(i,i) = Length(SA[i])

由這兩個性質我們可以得出一個 **LCP Lemma**：

對任意 1≤i<j<k≤n，LCP(i,j)=min{LCP(i,j)=LCP(j,k)}

且由這個 Lemma 我們可以得出 **LCP Theorem**：

對任意 1≤i<j≤n，LCP(i,j)=min{ LCP(k-1,k) | i<k≤j }

具體證明就不說明了，直觀來看其實是蠻顯然的，因為任兩個相鄰的 Suffix(SA[k])跟 Suffix(SA[k+1])是變異最少的，所以 i 與 j 的 LCP 就會是 i~j 之間最小的 LCP。

因此我們可以這裡定義一個 height 數列，另 height[i]=LCP(i-1,i)，1<i≤j。且令 height[1]=0。

且為了方便我們再設另一個 height 的反數列，h[i]=height[Rank[i]] ⇔

height[i]=h[SA[i]]。

我們可以明顯看出當要求 LCP(i,j)時就是要求 height[i+1]~height[j]之間的最小值，這我們可以利用 RMQ（Range Minimum Query）或線段樹來做，詳細做法之後課程才會說到。

然而現在最重要的就是如何求出 h 或 height 數列呢？（只要求出一個另一個都能容易求出）

研究發現我們能有第三個性質：

性質 3　對 i>1 且 Rank[i]>1 必有 h[i] ≥ h[i-1]-1。

這個性質其實還蠻不直觀，不過證明同樣不在此說明。我們可以根據性質三，並依照下列方法求出 h[i]：

1. **Rank[i] = 1**：h[i] = 0。
2. **i = 1 或 h[i-1] ≤ 1**：直接將 Suffix(i)跟 Suffix(Rank[i]-1)從第一個字開始比較直到有兩個字不同。
3. **i>1, Rank[i]>1, h[i-1]>1**：Suffix(i)跟 Suffix(Rank[i]-1)至少有前 h[i-1]-1 個字是相同的，於是比較從 h[i-1]開始，直到某個地方不相同就可以計算出 h[i]了。

看起來好像就只是暴力比對而已，但是這樣的複雜度卻只有 O(n)，是個很有效率的做法。因此便能在 O(n)時間內求出 h 跟 height 數列了。

§3-3 小結（Conclusion）

後綴數組在字串相關問題中是非常有利的應用，結合 height 數列的力量，發揮效力更加強大，例如說模式批配可以達到 O(mlgn)、多模式字串匹配可以達 O(m+lgn)、求最長回文子字串為 O(mlgm)...，總之其功能強大，在此就不說明。

事實上還有一種較做後綴樹的東西，對於所有後綴數組做得到的後綴樹都做得到，但是後綴樹實行難度較高，因此在此不做說明，如果有興趣者則可以自行研究。

另外推薦一篇對於後綴數組寫得很好的文章：

※ 2004 年 国家集训队论文许智磊 - 《后缀数组》

●Section 4 自動機（Automaton）

自動機在計算理論（Theory of Computation）中是很重要的一個應用，在此要說明的便是對字串問題很有利的工具—有限狀態自動機（Deterministic Finite Automata, DFA）。

§4-1 定義（Definition）

有限狀態自動機 $\mathcal{M}$ 是一個 5 元組（$Q, q_0, F, \Sigma, \delta$），其中：

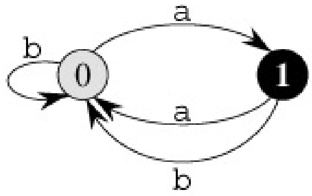
- Q 是狀態的一個有限集合
- $q_0 \in Q$ 是一個初始狀態
- $F \subseteq Q$ 是一個可接受的狀態集合
- Σ 是輸入的字元集
- δ 是一個 $Q \times \Sigma \rightarrow Q$ 的函數，稱為 $\mathcal{M}$ 的狀態轉移函數

下圖即顯示了一個簡單的自動機 $\mathcal{M}$（{0,1},0,{1},{a,b},δ）：

其中 δ 是轉移函數，δ(0,a)=1, δ(0,b)=0, δ(1,a)=0, δ(1,b)=0。

state	input a	input b
0	1	0
1	0	0

(a)



(b)

而我們還可以推導出一個函數 δ*，其為一個 Q×Σ*→Q 的函數，如果 S 是一個字串，q_0 是起始狀態，則 δ*有下列定義。

δ*(q_0,0) = q_0（0 代表一個空字串）

δ*(q_0,uc) = δ(δ*(q_0,u),c) （c 代表一個字元，uc 就是把 c 接在 u 末端）

特別的如果我們說 δ*(q_0,u) ∈ F，則 u 是對 $\mathcal{M}$ 來說可接受的（accepted），否則他就是被拒絕的（rejected）。

所有被 $\mathcal{M}$ 接受的字串所組成的集合我們寫做 L($\mathcal{M}$)，其為一個 Σ*的子集。

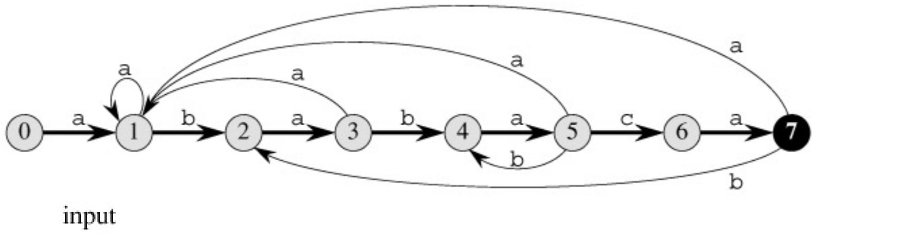
如果一個語言（Language）是規律（Regular）的，代表存在一個 $\mathcal{M}$ 能夠接受他的所有元素。

§4-2 實作與應用（Implementation and Application）

我們可以發現到，自動機在做的事其實跟 KMP 很像，每次往後匹配，匹配失敗就跳到另一個位置繼續匹配，一但出現狀態是被接受的，就代表字串匹配成功了。因此我們只要求出對模式字串 T 的自動機 $\mathcal{M}$，字串匹配就是小意思了。

```
FINITE-AUTOMATON-MATCHER(S, δ, m)
n ← length[S]
q ← 0
for i ← 1 to n
    do q ← δ(q, S[i])
       if q = m
          then print "Pattern occurs with shift" i - m
```

下面就是一個自動機的說明：



state	input a	input b	input c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
state $\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

但問題來了我們要怎麼求出對應模式字串 T 的自動機 $\mathcal{M}$ 呢？

假設我們的狀態指的就是已匹配成功的個數的話 $Q = \{0, 1, \ldots, m\}$。那麼當對某個狀態 q 與字元 a，若 $q = m$ 或者 $T[q+1] \neq a$，則 $\delta(q,a) = \delta(\pi[q],a)$（其中 π 是 KMP 算法中所求的前綴函數）。

因此很簡單我們可以得到下列算法：

```
COMPUTE-TRANSITION-FUNCTION(T, Σ)
m ← length[T]
for q ← 0 to m
    do for each character a ∈ Σ
           do if q = m or T[q+1] ≠ a
                 then δ(q, a) ← δ(π[q], a)
              else
                 δ(q, a) ← q + 1
return δ
```

複雜度很明顯是 $\Theta(m + m|\Sigma|)$，算很有效率了。

§4-3 小結（Conclusion）

當然自動機的應用沒那麼少，像多模式字串匹配也能夠用得上，它可以應用的地方還多著。大學還有專門位自動機開的一門**計算理論 Theory of Computation** 的課程，不過那不在我們今天討論的範圍，而且對競賽來說就也不是幫助那麼大了，有興趣者就自行參考相關書籍囉。