NCKU Programming Contest Training Course 2017/03/29

Jingfei Yang

https://nckuacm.github.io/slides/2017_stringMatching.pdf

Department of Computer Science and Information Engineering
National Cheng Kung University
Tainan, Taiwan

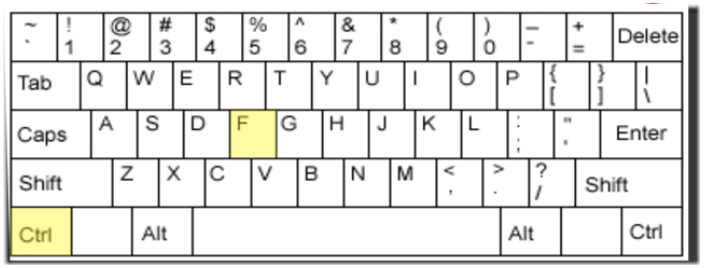
String Basic

- 字串 string
  - 字元的有序序列 $A = a_0 a_1 \ldots a_{n-1}$
  - $a_i \in$ 字元集，n 是字串的長度
- 子字串 substring
  - $A[i,j] = a_i a_{i+1} a_{i+2} \ldots a_j$ (A 連續的一段)
- 子序列 subsequence
  - $B = a_{q_1} a_{q_2} a_{q_3} \ldots a_{q_m}, 0 \le q_1 < q_2 < \cdots < q_m < n$ (不連續)
- 後綴 suffix
  - A 的一個子字串 $S_A(k) = a_k a_{k+1} a_{k+2} \ldots a_n,\ 0 \le k < n$
- 前綴 prefix
  - A 的一個子字串 $P_A(h) = a_0 a_1 a_2 \ldots a_h,\ 0 \le h < n$

String Basic

- S = “abcbbab”
  - 子字串： “bcb”，“bba”，…
  - 子序列： “acb”，“bbb”，…
  - 前綴： “abcb”，“ab”，…
  - 後綴： “bbab”，“ab”，…

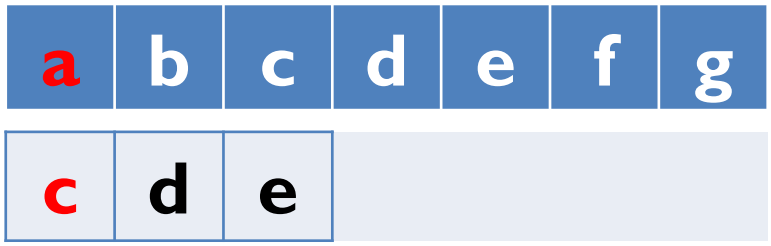
String Matching



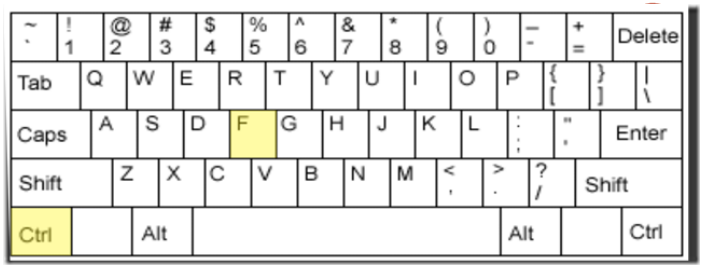
- 給兩個字串 A, B 找出所有 B 出現在 A 中的位置
-

```
1 for(int i=0; i+lenB<=lenA; ++i){
2       int mat=0;
3       while(mat<lenB && A[i+mat]==B[mat]) ++mat;
4       if(mat == lenB) print(i);
5 }
```

- 複雜度：$\mathcal{O}(|A|)$
- A = “abcdefg”
  B = “cde”



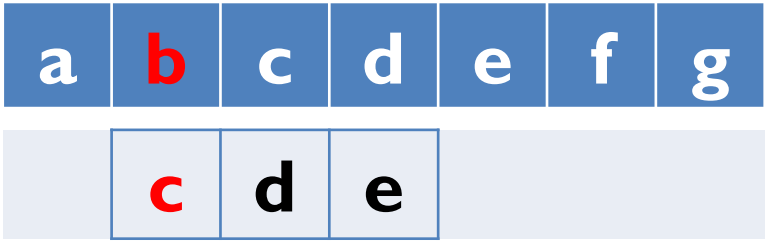
String Matching



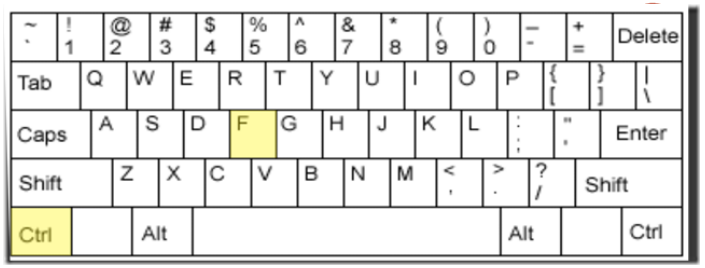
- 給兩個字串 A, B 找出所有 B 出現在 A 中的位置
-

```
1 for(int i=0; i+lenB<=lenA; ++i){
2       int mat=0;
3       while(mat<lenB && A[i+mat]==B[mat]) ++mat;
4       if(mat == lenB) print(i);
5 }
```

- 複雜度：$\mathcal{O}(|A|)$
- A = “abcdefg”
  B = “cde”



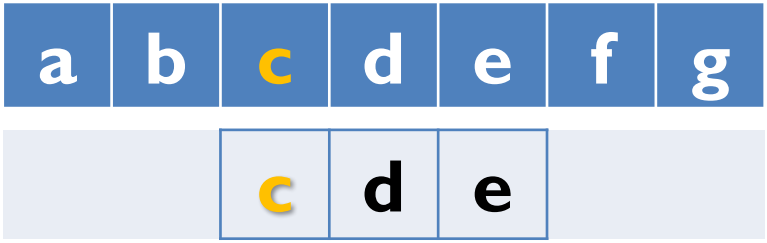
String Matching



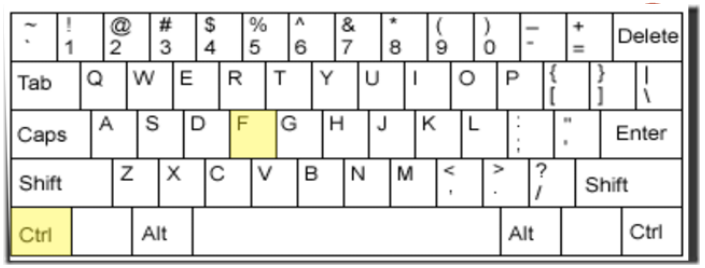
- 給兩個字串 A, B 找出所有 B 出現在 A 中的位置
-

```
1 for(int i=0; i+lenB<=lenA; ++i){
2       int mat=0;
3       while(mat<lenB && A[i+mat]==B[mat]) ++mat;
4       if(mat == lenB) print(i);
5 }
```

- 複雜度：$\mathcal{O}(|A|)$
- A = “abcdefg”
  B = “cde”



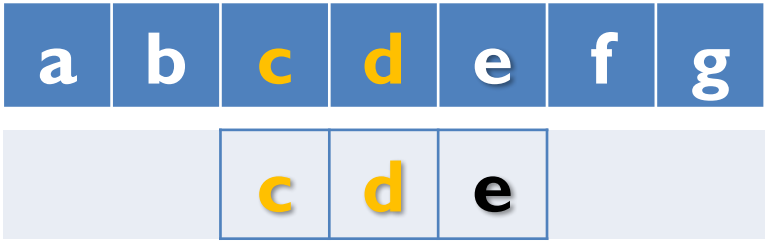
String Matching



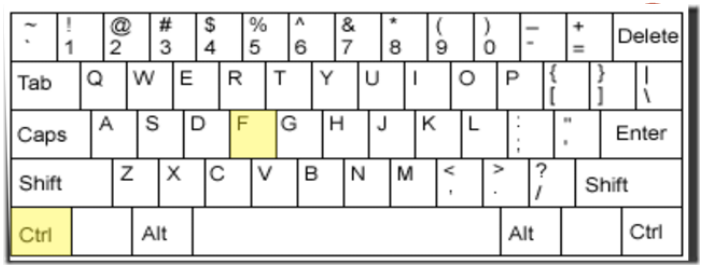
- 給兩個字串 A, B 找出所有 B 出現在 A 中的位置
-

```
1 for(int i=0; i+lenB<=lenA; ++i){
2       int mat=0;
3       while(mat<lenB && A[i+mat]==B[mat]) ++mat;
4       if(mat == lenB) print(i);
5 }
```

- 複雜度：$\mathcal{O}(|A|)$
- A = “abcdefg”
  B = “cde”



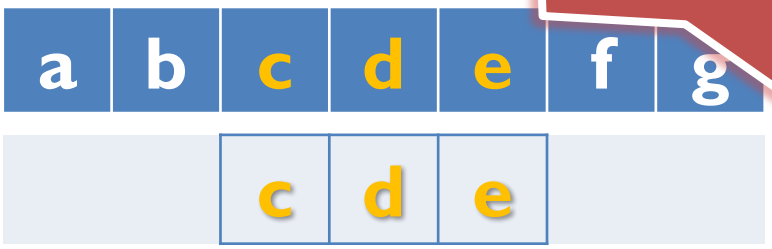
String Matching



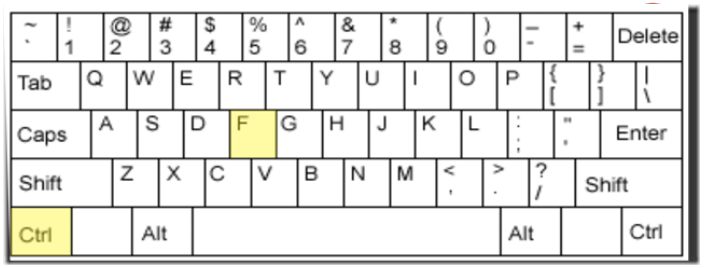
- 給兩個字串 A, B 找出所有 B 出現在 A 中的位置
-

```
1 for(int i=0; i+lenB<=lenA; ++i){
2       int mat=0;
3       while(mat<lenB && A[i+mat]==B[mat]) ++mat;
4       if(mat == lenB) print(i);
5 }
```

- 複雜度：$\mathcal{O}(|A|)$
- A = "abcdefg"
  B = "cde"



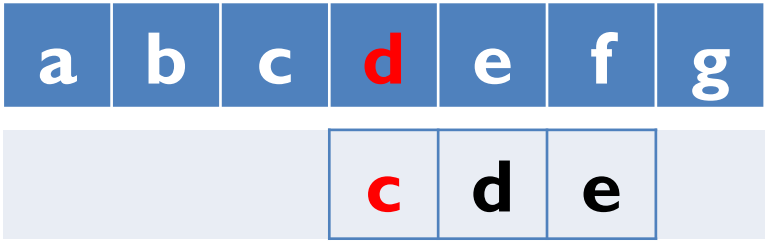
String Matching



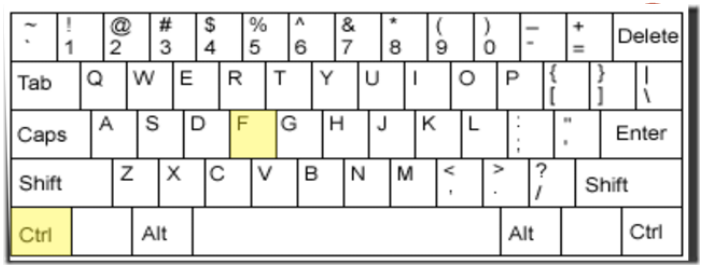
- 給兩個字串 A, B 找出所有 B 出現在 A 中的位置
-

```
1 for(int i=0; i+lenB<=lenA; ++i){
2       int mat=0;
3       while(mat<lenB && A[i+mat]==B[mat]) ++mat;
4       if(mat == lenB) print(i);
5 }
```

- 複雜度：$\mathcal{O}(|A|)$
- A = “abcdefg”
  B = “cde”



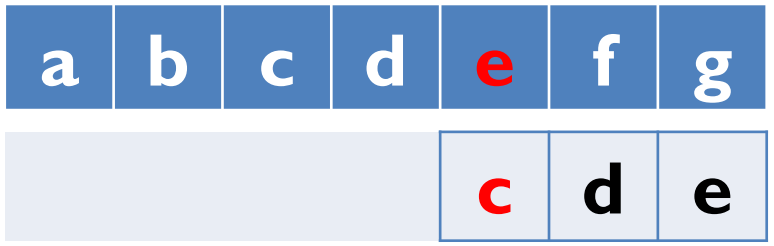
String Matching



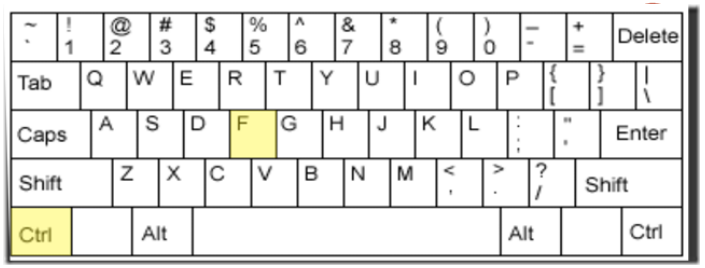
- 給兩個字串 A, B 找出所有 B 出現在 A 中的位置
-

```
1 for(int i=0; i+lenB<=lenA; ++i){
2       int mat=0;
3       while(mat<lenB && A[i+mat]==B[mat]) ++mat;
4       if(mat == lenB) print(i);
5 }
```

- 複雜度：$\mathcal{O}(|A|)$
- A = “abcdefg”
  B = “cde”



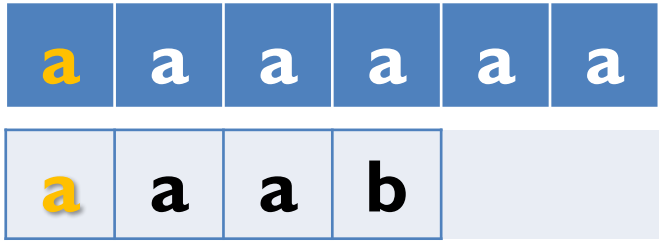
String Matching



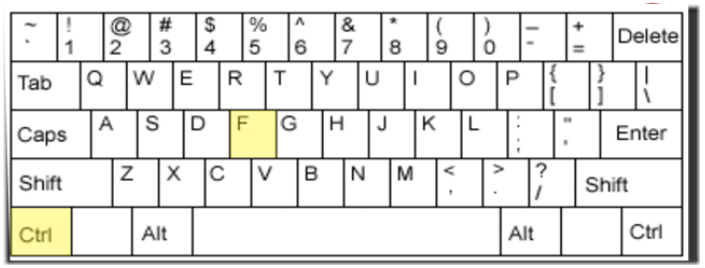
- 給兩個字串 A, B 找出所有 B 出現在 A 中的位置
-

```
1 for(int i=0; i+lenB<=lenA; ++i){
2       int mat=0;
3       while(mat<lenB && A[i+mat]==B[mat]) ++mat;
4       if(mat == lenB) print(i);
5 }
```

- 複雜度：$\mathcal{O}(|A|)$
- A = “aaaaaaa…aaa”
  B = “aaaaaaa…aab”



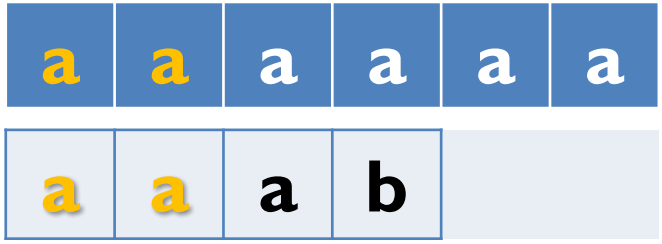
String Matching



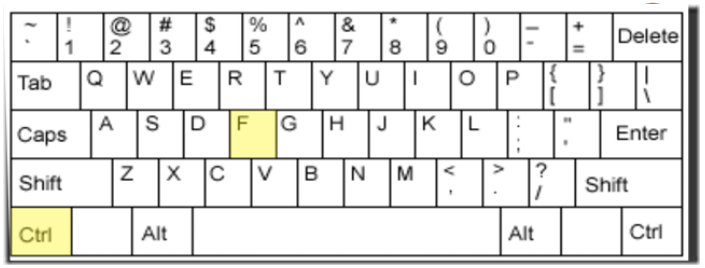
- 給兩個字串 A, B 找出所有 B 出現在 A 中的位置
-

```
1 for(int i=0; i+lenB<=lenA; ++i){
2       int mat=0;
3       while(mat<lenB && A[i+mat]==B[mat]) ++mat;
4       if(mat == lenB) print(i);
5 }
```

- 複雜度：$\mathcal{O}(|A|)$
- A = “aaaaaaa…aaa”
  B = “aaaaaaa…aab”



String Matching

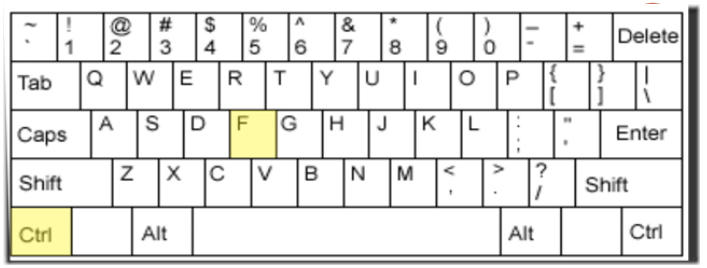


- 給兩個字串 A, B 找出所有 B 出現在 A 中的位置
-

```
1 for(int i=0; i+lenB<=lenA; ++i){
2       int mat=0;
3       while(mat<lenB && A[i+mat]==B[mat]) ++mat;
4       if(mat == lenB) print(i);
5 }
```

- 複雜度：$\mathcal{O}(|A|)$
- A = “aaaaaaa…aaa”
  B = “aaaaaaa…aab”

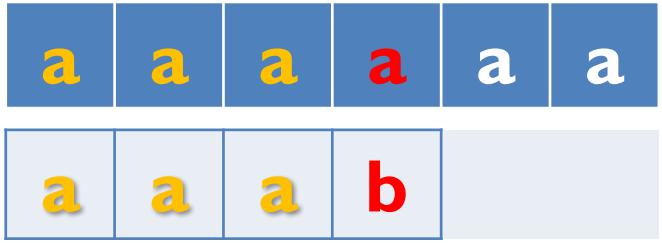
String Matching



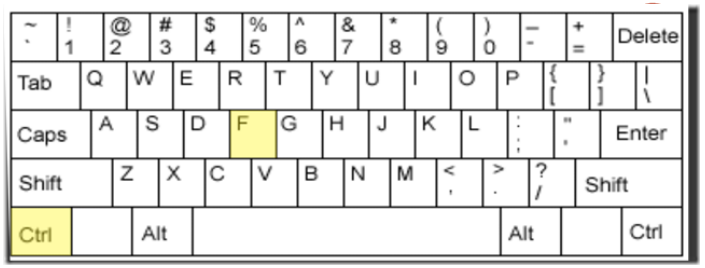
- 給兩個字串 A, B 找出所有 B 出現在 A 中的位置
-

```
1 for(int i=0; i+lenB<=lenA; ++i){
2       int mat=0;
3       while(mat<lenB && A[i+mat]==B[mat]) ++mat;
4       if(mat == lenB) print(i);
5 }
```

- 複雜度：$\mathcal{O}(|A|)$
- A = “aaaaaaa…aaa”
  B = “aaaaaaa…aab”



String Matching

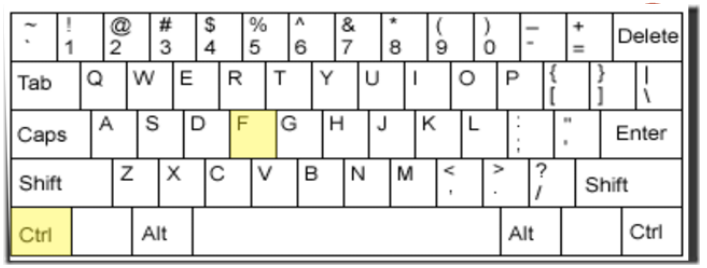


- 給兩個字串 A, B 找出所有 B 出現在 A 中的位置
-

```
1 for(int i=0; i+lenB<=lenA; ++i){
2       int mat=0;
3       while(mat<lenB && A[i+mat]==B[mat]) ++mat;
4       if(mat == lenB) print(i);
5 }
```

- 複雜度：$\mathcal{O}(|A|)$
- A = “aaaaaaa…aaa”
  B = “aaaaaaa…aab”

String Matching

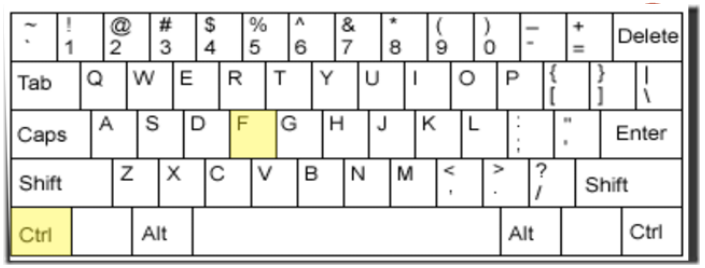


- 給兩個字串 A, B 找出所有 B 出現在 A 中的位置
-

```
1 for(int i=0; i+lenB<=lenA; ++i){
2       int mat=0;
3       while(mat<lenB && A[i+mat]==B[mat]) ++mat;
4       if(mat == lenB) print(i);
5 }
```

- 複雜度：$\mathcal{O}(|A|)$
- A = “aaaaaaa…aaa”
  B = “aaaaaaa…aab”

String Matching

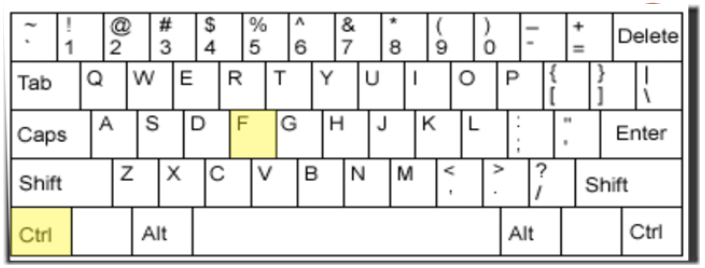


- 給兩個字串 A, B 找出所有 B 出現在 A 中的位置
-

```
1 for(int i=0; i+lenB<=lenA; ++i){
2       int mat=0;
3       while(mat<lenB && A[i+mat]==B[mat]) ++mat;
4       if(mat == lenB) print(i);
5 }
```

- 複雜度：$\mathcal{O}(|A|)$
- A = “aaaaaaa…aaa”
  B = “aaaaaaa…aab”

String Matching

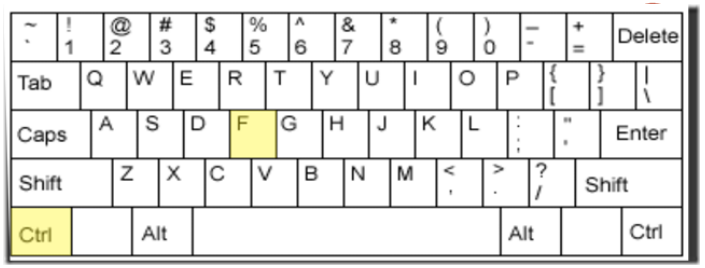


- 給兩個字串 A, B 找出所有 B 出現在 A 中的位置
-

```
1 for(int i=0; i+lenB<=lenA; ++i){
2       int mat=0;
3       while(mat<lenB && A[i+mat]==B[mat]) ++mat;
4       if(mat == lenB) print(i);
5 }
```

- 複雜度：$\mathcal{O}(|A|)$
- A = “aaaaaaa…aaa”
  B = “aaaaaaa…aab”

String Matching

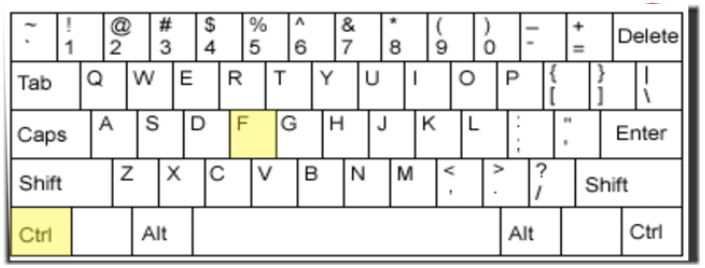


- 給兩個字串 A, B 找出所有 B 出現在 A 中的位置
-

```
1 for(int i=0; i+lenB<=lenA; ++i){
2       int mat=0;
3       while(mat<lenB && A[i+mat]==B[mat]) ++mat;
4       if(mat == lenB) print(i);
5 }
```

- 複雜度：$\mathcal{O}(|A|)$
- A = “aaaaaaa…aaa”
  B = “aaaaaaa…aab”

String Matching

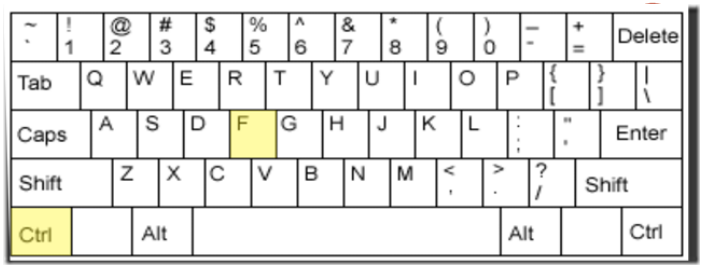


- 給兩個字串 A, B 找出所有 B 出現在 A 中的位置
-

```
1 for(int i=0; i+lenB<=lenA; ++i){
2       int mat=0;
3       while(mat<lenB && A[i+mat]==B[mat]) ++mat;
4       if(mat == lenB) print(i);
5 }
```

- 複雜度：$\mathcal{O}(|A|)$
- A = “aaaaaaa…aaa”
  B = “aaaaaaa…aab”

String Matching

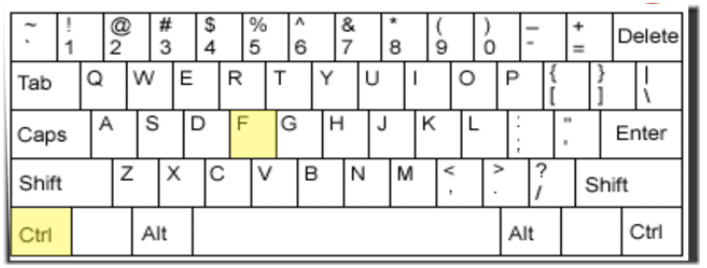


- 給兩個字串 A, B 找出所有 B 出現在 A 中的位置
-

```
1 for(int i=0; i+lenB<=lenA; ++i){
2       int mat=0;
3       while(mat<lenB && A[i+mat]==B[mat]) ++mat;
4       if(mat == lenB) print(i);
5 }
```

- 複雜度：$\mathcal{O}(|A|)$
- A = “aaaaaaa…aaa”
  B = “aaaaaaa…aab”

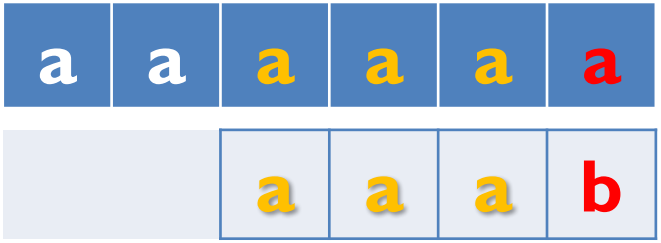
String Matching



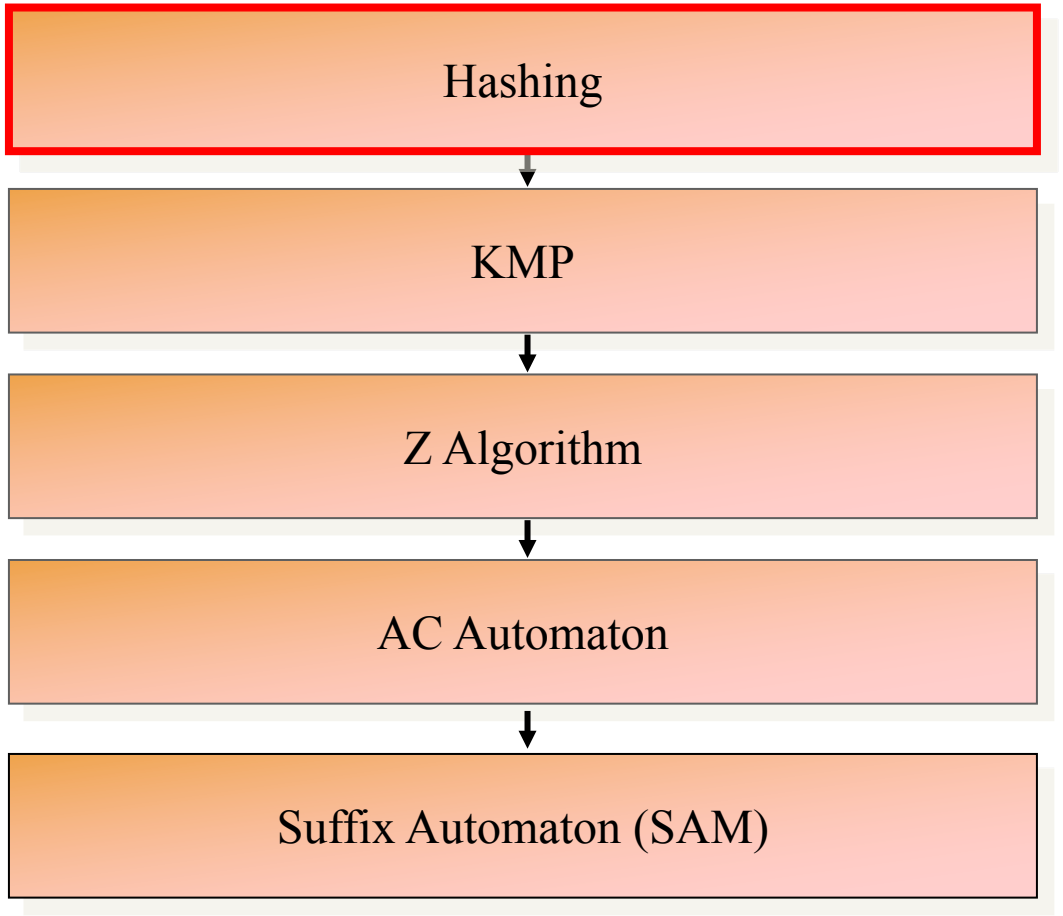
- 給兩個字串 A, B 找出所有 B 出現在 A 中的位置
-

```
1 for(int i=0; i+lenB<=lenA; ++i){
2       int mat=0;
3       while(mat<lenB && A[i+mat]==B[mat]) ++mat;
4       if(mat == lenB) print(i);
5 }
```

- 複雜度：~~$\mathcal{O}(|A|)$~~ $\mathcal{O}(|A||B|)$
- A = “aaaaaaa…aaa”
  B = “aaaaaaa…aab”



Outline



Hashing

- 分類
  - 將字串分到有限的整數裡
  - 函數 $f: string \longmapsto \{0, 1, \ldots, \mathbb{Q} - 1\}$
- 要求
  - f 容易取得
  - 均勻
- 思考
  1. $f(A) \neq f(B) \Rightarrow A \neq B$
  2. $A \neq B \Rightarrow f(A) \neq f(B)$ → 不一定
  3. 分 n 類，碰撞機率 1/n

Hashing

- Rabin-Karp rolling hash function 定義
  - $f(A) = a_0 p^{n-1} + a_1 p^{n-2} + \cdots + a_{n-2} p + a_{n-1} \bmod q$
  - 類似： p 進位制，分成 q 類
  - p, q 取不同**質數** → 均勻
- 滾動
  1. $f(A) \equiv f(A[0, n-2])p + a_{n-1} \bmod q$
     → 計算 A 所有前綴的 hash value，$\mathcal{O}(|A|)$
  2. $f(A[i, j]) \equiv f(A[0, j]) - p^{j-i+1} f(A[0, i-1]) \bmod q$
     → 任何 A 子字串的 hash value，$\mathcal{O}(1)$
  3. 枚舉 A 長度為 |B| 的子字串，比較 hash value
     → $\mathcal{O}(N)$

Hashing

- A = “abcdefg”

a	b	c	d	e	f	g
$0p^2 + 1p^1 + 2p^0$						

- B = “cde” = $2p^2 + 3p^1 + 4p^0$

Hashing

- A = “abcdefg”

a	b	c	d	e	f	g

$1p^2 + 2p^1 + 3p^0$

- B = “cde” = $2p^2 + 3p^1 + 4p^0$

Hashing

- A = "abcdefg"

a	b	c	d	e	f	g
		$2p^2 + 3p^1 + 4p^0$				

- B = "cde" = $2p^2 + 3p^1 + 4p^0$

made by Jingfei

Hashing

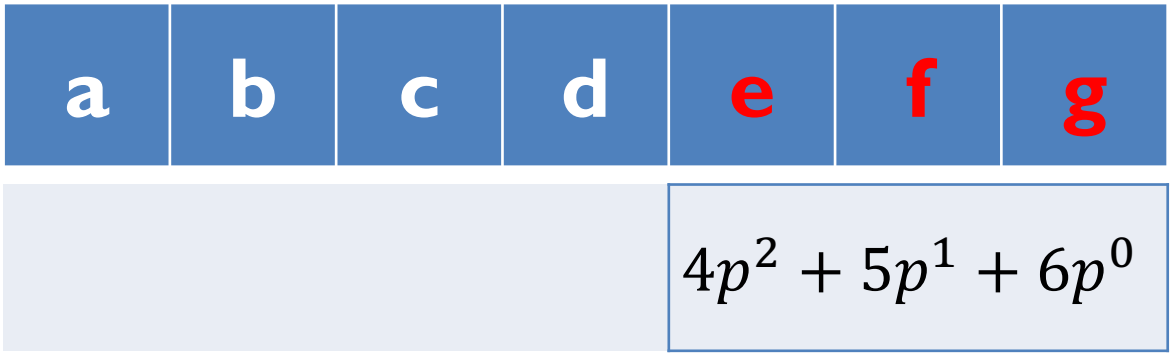
- A = “abcdefg”

a	b	c	d	e	f	g
			$3p^2 + 4p^1 + 5p^0$			

- B = “cde” = $2p^2 + 3p^1 + 4p^0$

Hashing

- A = “abcdefg”



- B = “cde” = $2p^2 + 3p^1 + 4p^0$

Hashing

- 回到剛剛
  - $A \neq B \Rightarrow f(A) \neq f(B)$ → 不一定
  - 相等時重新檢查一次？
  - A = "aaaaaaa...aaa"
    B = "aaaaaaa...aab"
- q 取大一點 (long long 質數)
  → 碰撞機率小
- ex. $q \in 10^{15} \Rightarrow probability: 10^{-15}$
- ex. 2147483647

Hashing – 參考

```
1   #define MAXN 1000000
2   #define prime_mod 1073676287
3   /*prime_mod 必須要是質數*/
4   typedef long long T;
5   char s[MAXN+5];
6   T h[MAXN+5];/*hash陣列*/
7   T h_base[MAXN+5];/*h_base[n]=(prime^n)%prime_mod*/
8   inline void hash_init(int len,T prime=0xdefaced){
9           h_base[0]=1;
10          for(int i=1;i<=len;++i){
11                  h[i]=(h[i-1]*prime+s[i-1])%prime_mod;
12                  h_base[i]=(h_base[i-1]*prime)%prime_mod;
13          }
14  }
15  inline T get_hash(int l,int r){/*閉區間寫法，設編號為0 ~ len-1*/
16          return (h[r+1]-(h[l]*h_base[r-l+1])%prime_mod+prime_mod)%prime_mod;
17  }
```

Source: 日月卦長的模板庫

→ [Rabin-Karp rolling hash] Rabin-Karp 字串hash演算法

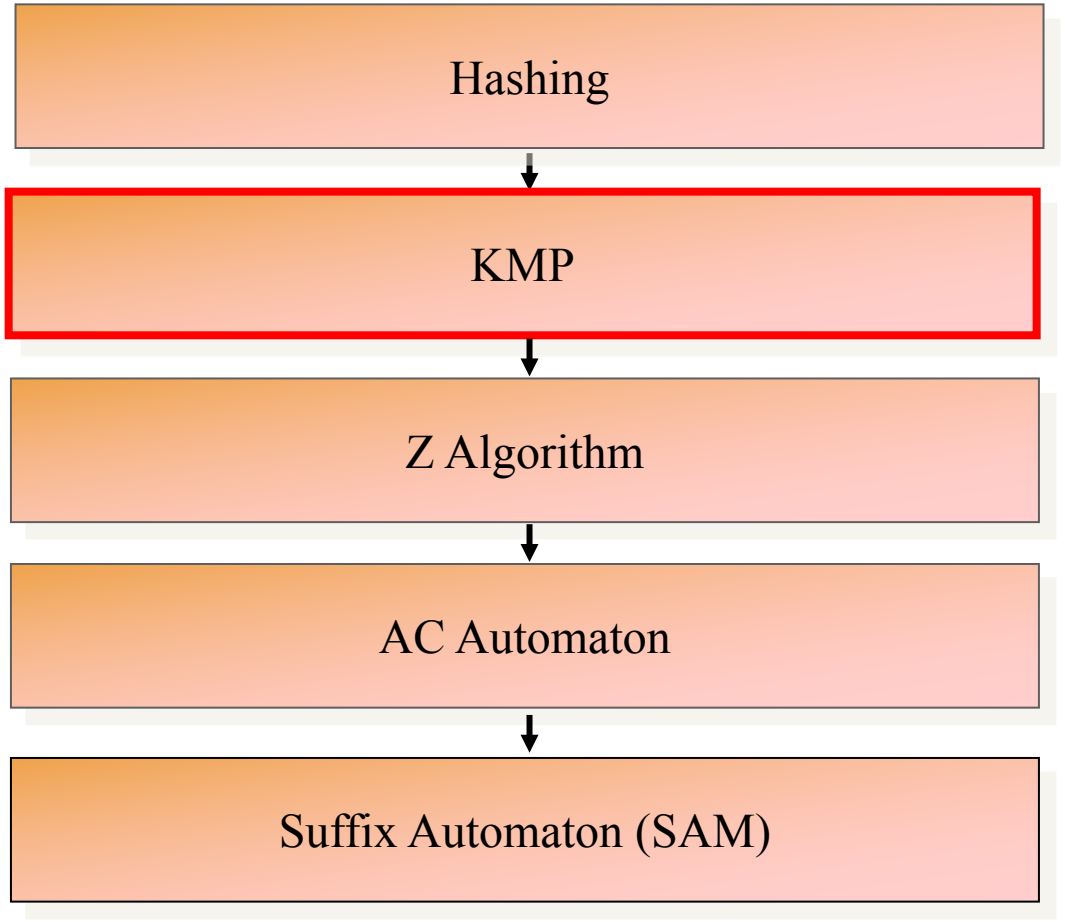
Hashing – 補充

- Double hashing → $h(k,i) = \big(h_1(k) + ih_2(k)\big)\%m$
- ex: $h_1(k) = k\%m$, $h_2(k) = 1 + \big(k\%(m-1)\big)$
  當 k=123456, m=701 → $h_1(k) = 80, h_2(k) = 257$
- 關鍵：即使 $h_1(k_1) == h_1(k_2)$,
  $h_2(k_1) == h_2(k_2)$ 應該不成立
- 總共會有 m^2 種！
- Double hashing 是最接近 uniform hashing 的方法，
  也就是 <0,1,2, … , $m-1$> 的任一種排列組合出現的機率是一樣的
- 實作： 用不同質數計算兩種不同的hash值，用pair拼起來即可

made by Jingfei

Source: https://www.csie.ntu.edu.tw/~hsinmu/courses/_media/dsa_12spring/hashing.pdf

Outline



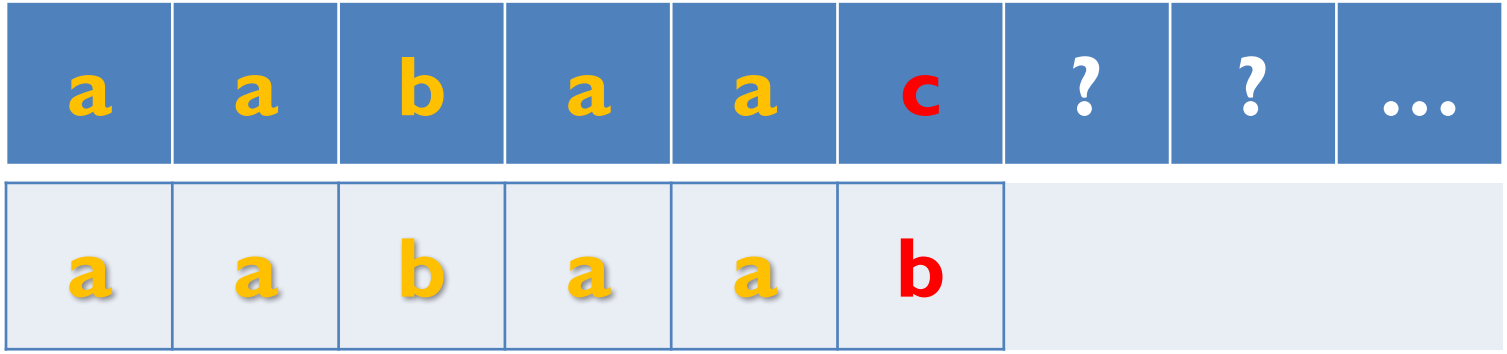
KMP

- Knuth-Morris-Pratt algorithm
- 再來看個例子
  a = “aabaac…”
  b = “aabaab”

a	a	b	a	a	c	?	?	…
a	a	b	a	a	b			

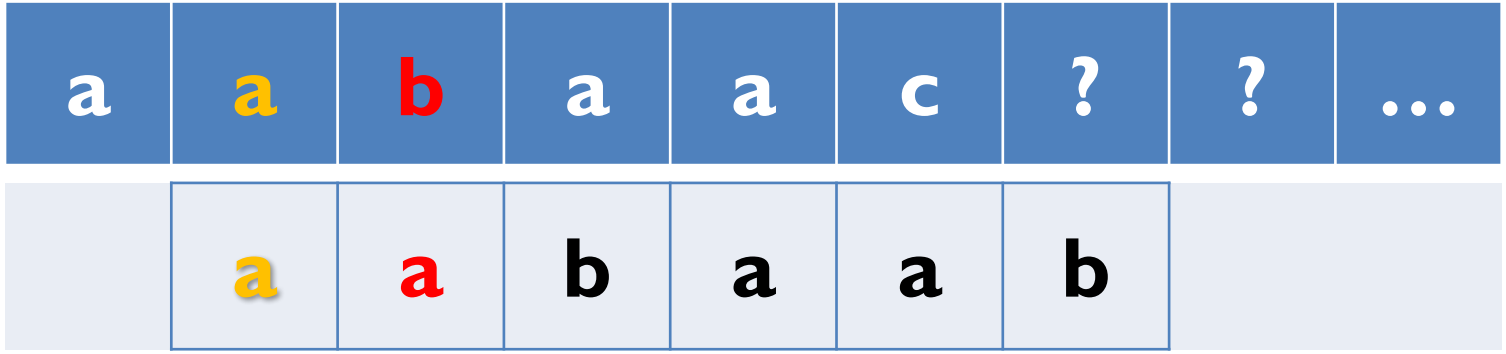
KMP

- Knuth-Morris-Pratt algorithm
- 再來看個例子
  a = “aabaac…”
  b = “aabaab”



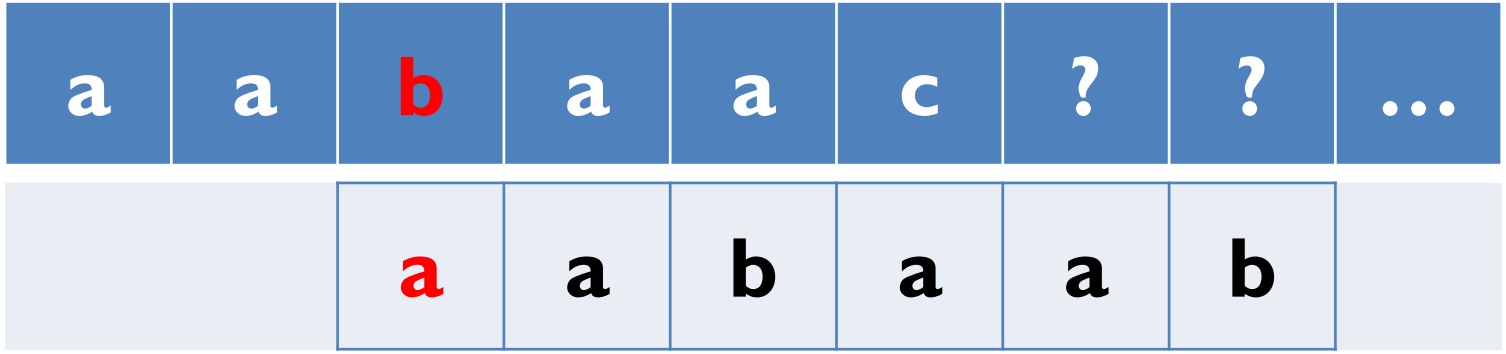
KMP

- Knuth-Morris-Pratt algorithm
- 再來看個例子
  a = “aabaac…”
  b = “aabaab”



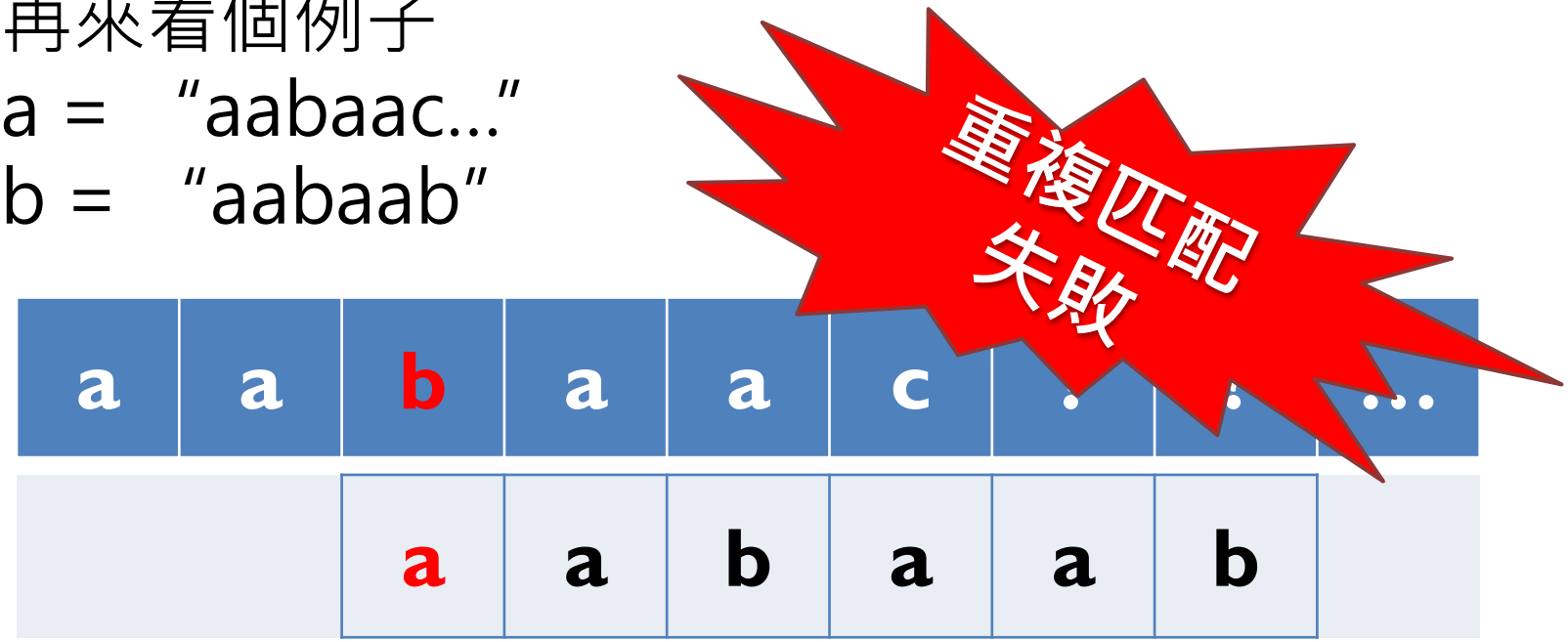
KMP

- Knuth-Morris-Pratt algorithm
- 再來看個例子
  a = “aabaac…”
  b = “aabaab”



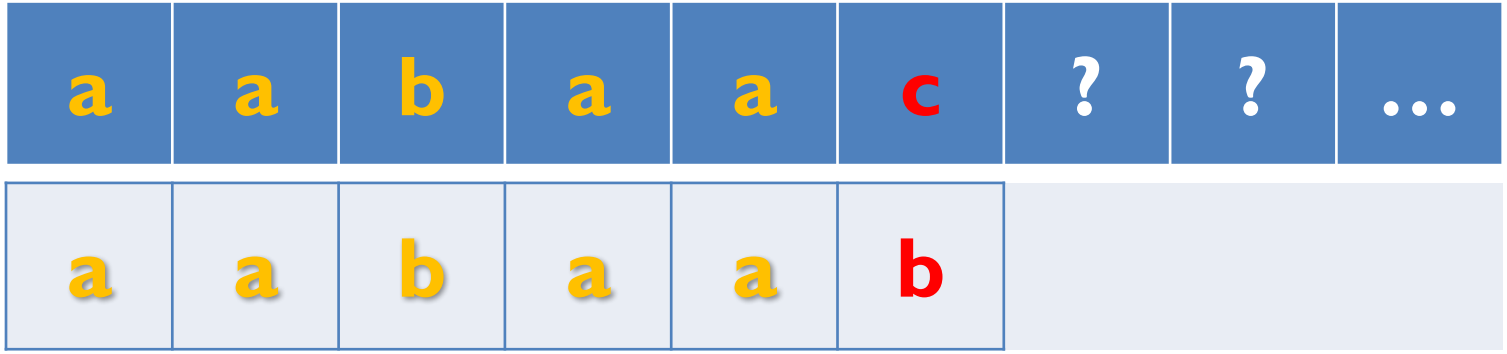
KMP

- Knuth-Morris-Pratt algorithm
- 再來看個例子
  a = “aabaac…”
  b = “aabaab”



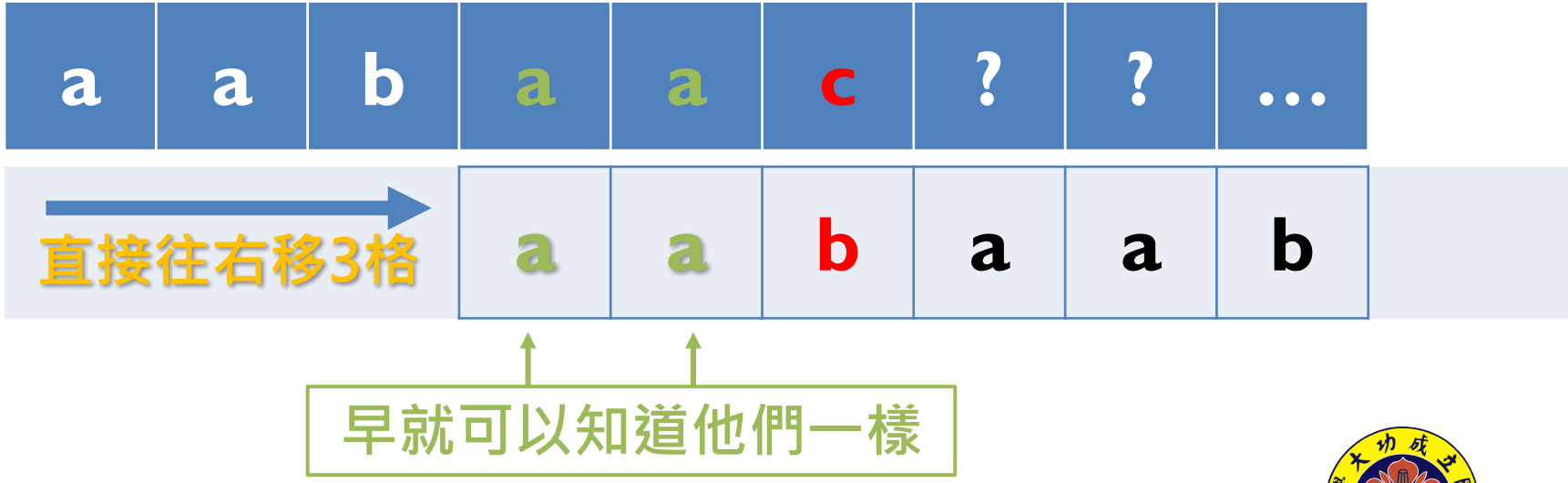
KMP

- Knuth-Morris-Pratt algorithm
- 再來看個例子
  a = “aabaac…”
  b = “aabaab”



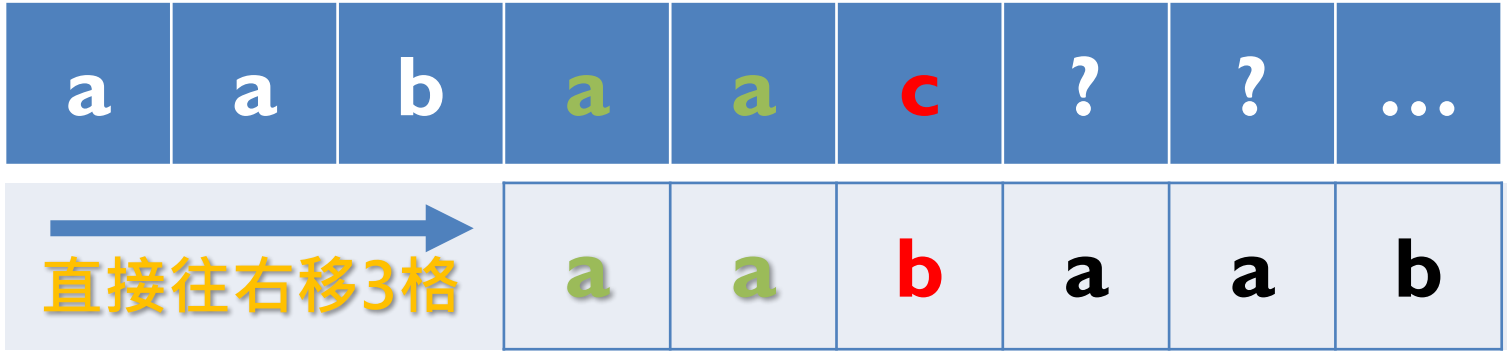
KMP

- Knuth-Morris-Pratt algorithm
- 再來看個例子
  a = “aabaac…”
  b = “aabaab”



KMP

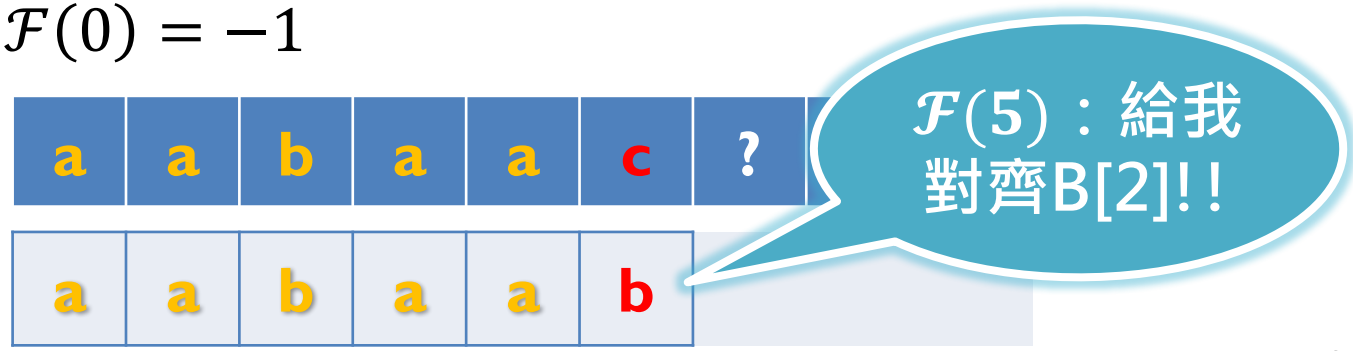
- Knuth-Morris-Pratt algorithm
- 再來看個例子
  a = “aabaac…”
  b = “aabaab”



問題出在 B 有重複子字串

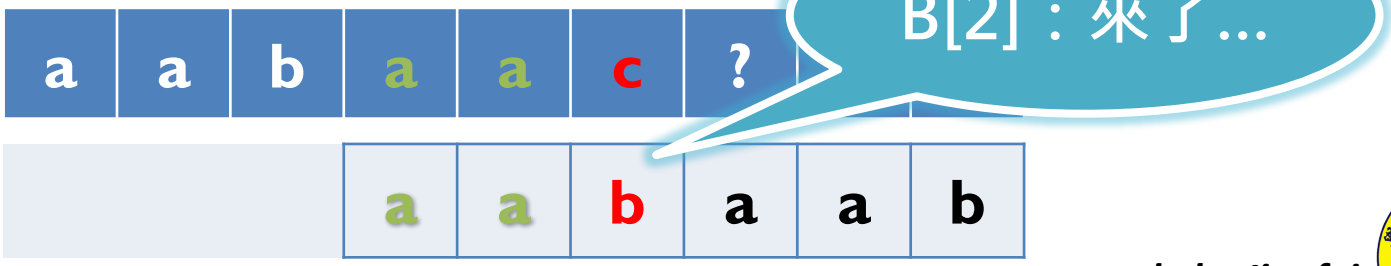
KMP

- Knuth-Morris-Pratt algorithm
- 怎麼處理 B ？
- 定義 **Fail function** (失敗函數)
  - 期望：$\mathcal{F}(i)$ 能知道匹配失敗時，B 要對齊哪裡繼續匹配
  - $\mathcal{F}_B(i) =$
    $$\begin{cases} \max\{k:\ P_B(k) = B[0,k] = B[i-k,i]\}, if\ i \neq 0\ and\ at\ least\ a\ \boldsymbol{k}\ exists \\ -1,\ else \end{cases}$$
  - $\mathcal{F}(0) = -1$
  -



KMP

- Knuth-Morris-Pratt algorithm
- 怎麼處理 B ？
- 定義 **Fail function** (失敗函數)
  - 期望：$\mathcal{F}(i)$ 能知道匹配失敗時，B 要對齊哪裡繼續匹配
  - $\mathcal{F}_B(i) =$
    $$\begin{cases} \max\{k:\ P_B(k) = B[0,k] = B[i-k,i]\}, if\ i \neq 0\ and\ at\ least\ a\ \boldsymbol{k}\ exists \\ -1,\ else \end{cases}$$
  - $\mathcal{F}(0) = -1$
  -



made by Jingfei

KMP

- Fail function example

i	0	1	2	3	4	5	6	7	8
B	a	b	z	a	b	z	a	b	c
pi	-1								

```
init:
    pi[0]=-1
    cur_pos=-1
```

KMP

- Fail function example

i	0	1	2	3	4	5	6	7	8
B	a	b	z	a	b	z	a	b	c
pi	-1	-1							

```
B[cur_pos+1]!=B[i]
pi[i]=cur_pos
```

KMP

- Fail function example

i	0	1	2	3	4	5	6	7	8
B	a	b	z	a	b	z	a	b	c
pi	-1	-1	-1						

```
B[cur_pos+1]!=B[i]
pi[i]=cur_pos
```

KMP

- Fail function example

i	0	1	2	3	4	5	6	7	8
B	a	b	z	a	b	z	a	b	c
pi	-1	-1	-1	0					

```
B[cur_pos+1]==B[i]
pi[i]=++cur_pos
```

KMP

- Fail function example

i	0	1	2	3	4	5	6	7	8
B	a	b	z	a	b	z	a	b	c
pi	-1	-1	-1	0	1				

```
B[cur_pos+1]==B[i]
pi[i]=++cur_pos
```

KMP

- Fail function example

i	0	1	2	3	4	5	6	7	8
B	a	b	z	a	b	z	a	b	c
pi	-1	-1	-1	0	1	2			

```
B[cur_pos+1]==B[i]
pi[i]=++cur_pos
```

KMP

- Fail function example

i	0	1	2	3	4	5	6	7	8
B	a	b	z	a	b	z	a	b	c
pi	-1	-1	-1	0	1	2	3		

```
B[cur_pos+1]==B[i]
pi[i]=++cur_pos
```

KMP

- Fail function example

i	0	1	2	3	4	5	6	7	8
B	a	b	z	a	b	z	a	b	c
pi	-1	-1	-1	0	1	2	3	4	

```
B[cur_pos+1]==B[i]
pi[i]=++cur_pos
```

KMP

- Fail function example

i	0	1	2	3	4	5	6	7	8
B	a	b	z	a	b	z	a	b	c
pi	-1	-1	-1	0	1	2	3	4	1

```
B[cur_pos+1]!=B[i]
pi[i]=pi[cur_pos]
```

KMP

- Fail function example

i	0	1	2	3	4	5	6	7	8
B	a	b	z	a	b	z	a	b	c
pi	-1	-1	-1	0	1	2	3	4	-1

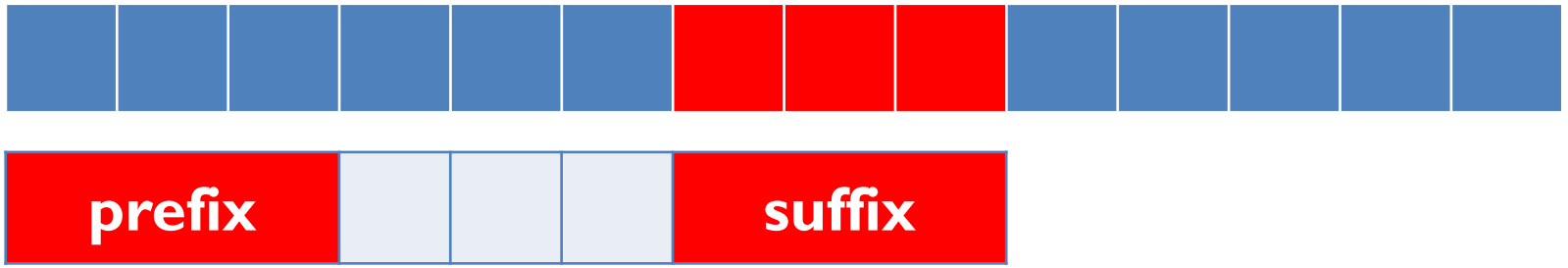
```
B[cur_pos+1]!=B[i]
pi[i]=pi[cur_pos]
```

KMP

- Fail function

```
inline void fail (char *B, int *pi) {
    int len = strlen(B);
    pi[0] = -1;
    for(int i=1, cur_pos=-1; i<len; ++i) {
        while(~cur_pos && B[i]!=B[cur_pos+1])
            cur_pos = pi[cur_pos];
        if(B[i]==B[cur_pos+1]) ++cur_pos;
        pi[i] = cur_pos;
    }
}
```

KMP

- Matching
- Fail function: 找出各後綴與前綴一樣的最大值
- 如果後綴 = 前綴 → 可直接位移
- 

KMP

- Matching
- Fail function: 找出各後綴與前綴一樣的最大值
- 如果後綴 = 前綴 → 可直接位移
- 

KMP

- Matching

A		x	a	b	z	a	b	z	a	b	z	a	b	c	d

cur_pos

	-1	0	1	2	3	4	5	6	7	8
B		a	b	z	a	b	z	a	b	c
pi		-1	-1	-1	0	1	2	3	4	-1

init:

cur_pos = -1

KMP

- Matching

i

A		x	a	b	z	a	b	z	a	b	z	a	b	c	d

A[i]!=B[cur_pos+1]

cur_pos

	-1	0	1	2	3	4	5	6	7	8
B		a	b	z	a	b	z	a	b	c
pi		-1	-1	-1	0	1	2	3	4	-1

KMP

- Matching

i

A		x	a	b	z	a	b	z	a	b	z	a	b	c	d

cur_pos

	-1	0	1	2	3	4	5	6	7	8
B		a	b	z	a	b	z	a	b	c
pi		-1	-1	-1	0	1	2	3	4	-1

A[i]==B[cur_pos+1]
++cur_pos

KMP

- Matching

i

A		x	a	b	z	a	b	z	a	b	z	a	b	c	d

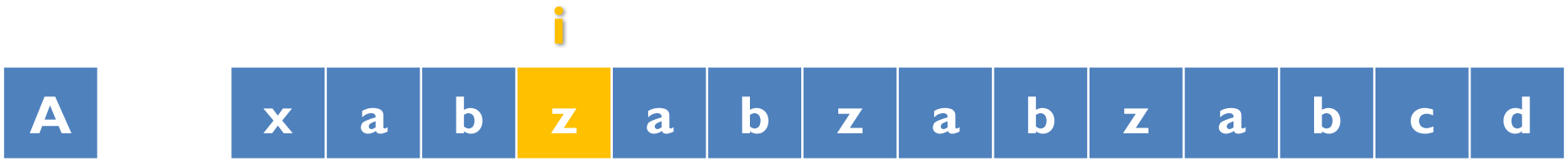
cur_pos

	-1	0	1	2	3	4	5	6	7	8
B		a	b	z	a	b	z	a	b	c
pi		-1	-1	-1	0	1	2	3	4	-1

A[i]==B[cur_pos+1]
++cur_pos

KMP

- Matching



A[i]==B[cur_pos+1]
++cur_pos

cur_pos

	-1	0	1	2	3	4	5	6	7	8
B		**a**	**b**	**z**	**a**	**b**	**z**	**a**	**b**	**c**
pi		-1	-1	-1	0	1	2	3	4	-1

KMP

- Matching

i

A		x	a	b	z	a	b	z	a	b	z	a	b	c	d

cur_pos

	-1	0	1	2	3	4	5	6	7	8
B		a	b	z	a	b	z	a	b	c
pi		-1	-1	-1	0	1	2	3	4	-1

A[i]==B[cur_pos+1]
++cur_pos

KMP

- Matching

i

A		x	a	b	z	a	b	z	a	b	z	a	b	c	d

cur_pos

	-1	0	1	2	3	4	5	6	7	8
B		a	b	z	a	b	z	a	b	c
pi		-1	-1	-1	0	1	2	3	4	-1

A[i]==B[cur_pos+1]
++cur_pos

KMP

- Matching

i

A		x	a	b	z	a	b	z	a	b	z	a	b	c	d

cur_pos

	-1	0	1	2	3	4	5	6	7	8
B		a	b	z	a	b	z	a	b	c
pi		-1	-1	-1	0	1	2	3	4	-1

A[i]==B[cur_pos+1]
++cur_pos

KMP

- Matching

i

A		x	a	b	z	a	b	z	a	b	z	a	b	c	d

cur_pos

	-1	0	1	2	3	4	5	6	7	8
B		a	b	z	a	b	z	a	b	c
pi		-1	-1	-1	0	1	2	3	4	-1

A[i]==B[cur_pos+1]
++cur_pos

KMP

- Matching

i

A		x	a	b	z	a	b	z	a	b	z	a	b	c	d

cur_pos

	-1	0	1	2	3	4	5	6	7	8
B		a	b	z	a	b	z	a	b	c
pi		-1	-1	-1	0	1	2	3	4	-1

A[i]==B[cur_pos+1]
++cur_pos

KMP

- Matching

i

A		x	a	b	z	a	b	z	a	b	z	a	b	c	d

cur_pos

	-1	0	1	2	3	4	5	6	7	8
B		a	b	z	a	b	z	a	b	c
pi		-1	-1	-1	0	1	2	3	4	-1

A[i]!=B[cur_pos+1]
cur_pos=pi[cur_pos]

KMP

- Matching

i

A		x	a	b	z	a	b	z	a	b	z	a	b	c	d

cur_pos

	-1	0	1	2	3	4	5	6	7	8
B		a	b	z	a	b	z	a	b	c
pi		-1	-1	-1	0	1	2	3	4	-1

A[i]==B[cur_pos+1]
++cur_pos

KMP

- Matching

i

A		x	a	b	z	a	b	z	a	b	z	a	b	c	d

cur_pos

	-1	0	1	2	3	4	5	6	7	8
B		a	b	z	a	b	z	a	b	c
pi		-1	-1	-1	0	1	2	3	4	-1

A[i]==B[cur_pos+1]
++cur_pos

KMP

- Matching

i

A		x	a	b	z	a	b	z	a	b	z	a	b	c	d

A[i]==B[cur_pos+1]
++cur_pos

cur_pos

	-1	0	1	2	3	4	5	6	7	8
B		a	b	z	a	b	z	a	b	c
pi		-1	-1	-1	0	1	2	3	4	-1

KMP

- Matching

i

A	x	a	b	z	a	b	z	a	b	z	a	b	c	d

cur_pos

A[i]==B[cur_pos+1]
++cur_pos

	-1	0	1	2	3	4	5	6	7	8
B		a	b	z	a	b	z	a	b	c
pi		-1	-1	-1	0	1	2	3	4	-1

KMP

- Matching

i

A		x	a	b	z	a	b	z	a	b	z	a	b	c	d

cur_pos

	-1	0	1	2	3	4	5	6	7	8
B		a	b	z	a	b	z	a	b	c
pi		-1	-1	-1	0	1	2	3	4	-1

A[i]==B[cur_pos+1]

++cur_pos

cur_pos+1==len(B)

Match!!!

KMP

- Matching

i

A		x	a	b	z	a	b	z	a	b	z	a	b	c	d

cur_pos

	-1	0	1	2	3	4	5	6	7	8
B		a	b	z	a	b	z	a	b	c
pi		-1	-1	-1	0	1	2	3	4	-1

A[i]==B[cur_pos+1]

++cur_pos

cur_pos+1==len(B)

cur_pos=pi[cur_pos]

KMP

- Matching

A		x	a	b	z	a	b	z	a	b	z	a	b	c	d

i

A[i]!=B[cur_pos+1]

cur_pos

	-1	0	1	2	3	4	5	6	7	8
B		a	b	z	a	b	z	a	b	c
pi		-1	-1	-1	0	1	2	3	4	-1

KMP

- Matching

```
inline void match(char *A, char *B, int *pi) {
    int lenA = strlen(A);
    int lenB = strlen(B);
    for(int i=1, cur_pos=-1; i<lenA; ++i) {
        while(~cur_pos && A[i]!=B[cur_pos+1])
            cur_pos = pi[cur_pos];
        if(A[i]==B[cur_pos+1]) ++cur_pos;
        if(cur_pos+1==lenB) {
          /* Match!! */
          cur_pos = pi[cur_pos];
        }
    }
}
```

made by Jingfei

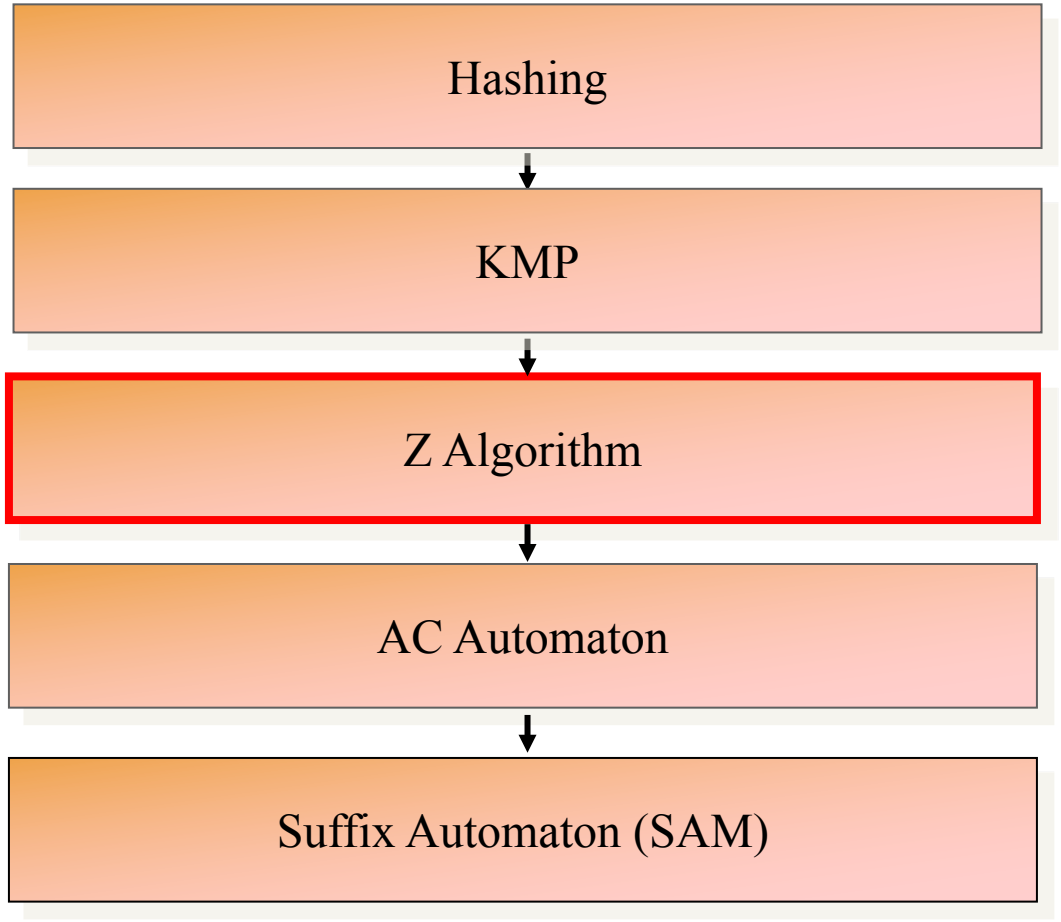
KMP

- Fail function + Matching
- Complexity
  - 關鍵: while-loop
  - cur_pos 每次只會 +1 或往前
  - 均攤後 $\mathcal{O}(|\mathrm{A}| + |\mathrm{B}|)$

Example

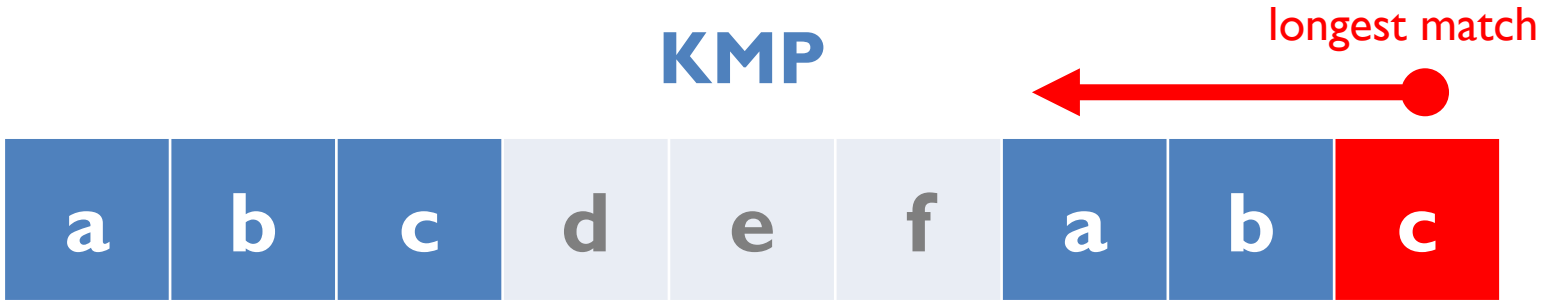
- POJ 3461
- UVA 455

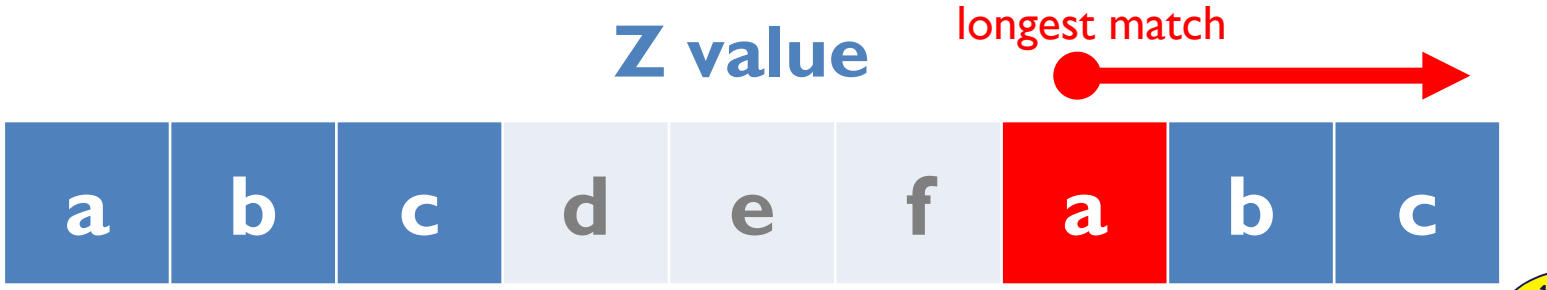
Outline



Z Algorithm

- vs KMP
  - S= "abcdefabc"





Z Algorithm

- 定義
  - $z_A(i) = \begin{cases} 0, if\ i = 0\ or\ A[i] \neq A[0] \\ \max\{k: A[0, k-1] = A[i, i+k-1]\}, else \end{cases}$
  - 由 $A[i]$ 開始的字串，可以和 A 自己匹配多長
  - ex.

i	0	1	2	3	4	5	6	7	8
A	a	b	z	a	a	b	z	a	b
z									

Z Algorithm

- 定義
  - $z_A(i) = \begin{cases} 0, if\ i = 0\ or\ A[i] \neq A[0] \\ \max\{k: A[0, k-1] = A[i, i+k-1]\}, else \end{cases}$
  - 由 $A[i]$ 開始的字串，可以和 A 自己匹配多長
  - ex.

i	0	1	2	3	4	5	6	7	8
A	a	b	z	a	a	b	z	a	b
z	9								

Z Algorithm

- 定義
  - $z_A(i) = \begin{cases} 0, if\ i = 0\ or\ A[i] \neq A[0] \\ \max\{k: A[0, k-1] = A[i, i+k-1]\}, else \end{cases}$
  - 由 $A[i]$ 開始的字串，可以和 A 自己匹配多長
  - ex.

i	0	1	2	3	4	5	6	7	8
A	a	b	z	a	a	b	z	a	b
z	9	0							

Z Algorithm

- 定義
  - $z_A(i) = \begin{cases} 0, if\ i = 0\ or\ A[i] \neq A[0] \\ \max\{k: A[0, k-1] = A[i, i+k-1]\}, else \end{cases}$
  - 由 $A[i]$ 開始的字串，可以和 A 自己匹配多長
  - ex.

i	0	1	2	3	4	5	6	7	8
A	a	b	z	a	a	b	z	a	b
z	9	0	0						

Z Algorithm

- 定義
  - $Z_A(i) = \begin{cases} 0, if\ i = 0\ or\ A[i] \neq A[0] \\ \max\{k: A[0, k-1] = A[i, i+k-1]\}, else \end{cases}$
  - 由 $A[i]$ 開始的字串，可以和 A 自己匹配多長
  - ex.

i	0	1	2	3	4	5	6	7	8
A	a	b	z	a	a	b	z	a	b
Z	9	0	0	1					

Z Algorithm

- 定義
  - $z_A(i) = \begin{cases} 0, if\ i = 0\ or\ A[i] \neq A[0] \\ \max\{k: A[0, k-1] = A[i, i+k-1]\}, else \end{cases}$
  - 由 $A[i]$ 開始的字串，可以和 A 自己匹配多長
  - ex.

i	0	1	2	3	4	5	6	7	8
A	a	b	z	a	a	b	z	a	b
z	9	0	0	1	4				

Z Algorithm

- 定義
  - $z_A(i) = \begin{cases} 0, if\ i = 0\ or\ A[i] \neq A[0] \\ \max\{k: A[0, k-1] = A[i, i+k-1]\}, else \end{cases}$
  - 由 $A[i]$ 開始的字串，可以和 A 自己匹配多長
  - ex.

i	0	1	2	3	4	5	6	7	8
A	a	b	z	a	a	b	z	a	b
z	9	0	0	1	4	0			

Z Algorithm

- 定義
  - $z_A(i) = \begin{cases} 0, if\ i = 0\ or\ A[i] \neq A[0] \\ \max\{k: A[0, k-1] = A[i, i+k-1]\}, else \end{cases}$
  - 由 $A[i]$ 開始的字串，可以和 A 自己匹配多長
  - ex.

i	0	1	2	3	4	5	6	7	8
A	**a**	**b**	**z**	**a**	**a**	**b**	**z**	**a**	**b**
z	9	0	0	1	4	0	0		

Z Algorithm

- 定義
  - $z_A(i) = \begin{cases} 0, if\ i = 0\ or\ A[i] \neq A[0] \\ \max\{k: A[0, k-1] = A[i, i+k-1]\}, else \end{cases}$
  - 由 $A[i]$ 開始的字串，可以和 A 自己匹配多長
  - ex.

i	0	1	2	3	4	5	6	7	8
A	a	b	z	a	a	b	z	a	b
z	9	0	0	1	4	0	0	2	

Z Algorithm

- 定義
  - $z_A(i) = \begin{cases} 0, if\ i = 0\ or\ A[i] \neq A[0] \\ \max\{k: A[0, k-1] = A[i, i+k-1]\}, else \end{cases}$
  - 由 $A[i]$ 開始的字串，可以和 A 自己匹配多長
  - ex.

i	0	1	2	3	4	5	6	7	8
A	**a**	**b**	**z**	**a**	**a**	**b**	**z**	**a**	**b**
z	9	0	0	1	4	0	0	2	0

Z Algorithm

- 假設一個字串 A 中，$\mathcal{Z}(i) = z$

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
A	**a**	**b**	**a**	**b**	**a**	**b**	**e**	**a**	**b**	**a**	**b**	**a**	**b**	**f**
$\mathcal{Z}$	14	0	4	0	2	0	0	**6**	0	4	0	2	0	0

Z Algorithm

- 假設一個字串 A 中，$\mathcal{Z}(i) = z$
  - $A[k] = A[i+k],\ if\ 0 \le k < z$

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
A	a	b	a	b	a	b	e	a	b	a	b	a	b	f
$\mathcal{Z}$	14	0	4	0	2	0	0	6	0	4	0	2	0	0

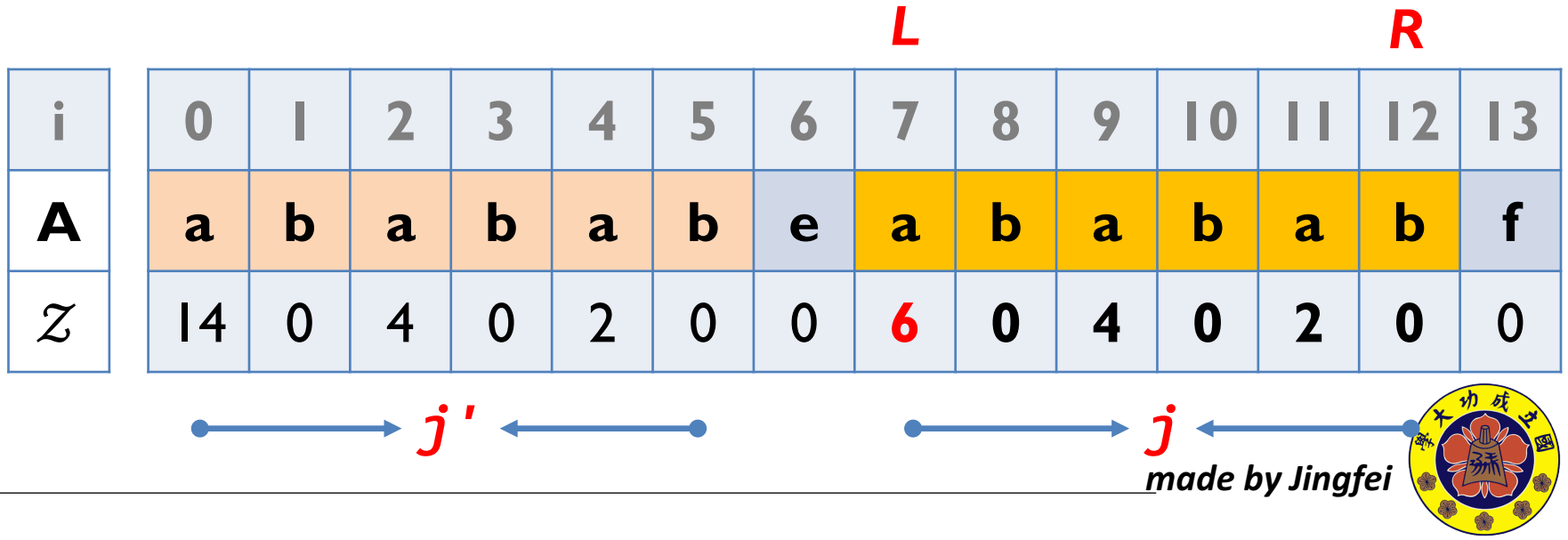
Z Algorithm

- 假設一個字串 A 中，$\mathcal{Z}(i) = z$
  - $A[k] = A[i+k],\ if\ 0 \le k < z$
  - $A[z] \neq A[i+z]$

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
A	a	b	a	b	a	b	e	a	b	a	b	a	b	f
$\mathcal{Z}$	14	0	4	0	2	0	0	6	0	4	0	2	0	0

Z Algorithm

- 假設一個字串 A 中，$\mathcal{Z}(i) = z$
  - $A[k] = A[i+k],\ if\ 0 \le k < z$
  - $A[z] \neq A[i+z]$
  - 令 $L = i,\ R = i+z-1,\ L \le j \le R,\ j' = j - L$



made by Jingfei

Z Algorithm

- 假設一個字串 A 中，$\mathcal{Z}(i) = z$
  - $A[k] = A[i+k],\ if\ 0 \le k < z$
  - $A[z] \neq A[i+z]$
  - 令 $L = i,\ R = i+z-1,\ L \le j \le R,\ j' = j - L$
  - $case \times 3$

								L					R	
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
A	a	b	a	b	a	b	e	a	b	a	b	a	b	f
$\mathcal{Z}$	14	0	4	0	2	0	0	6	0	4	0	2	0	0

made by Jingfei

Z Algorithm

- 假設一個字串 A 中，$\mathcal{Z}(i) = z$
  - $A[k] = A[i+k],\ if\ 0 \le k < z$
  - $A[z] \neq A[i+z]$
  - 令 $L = i,\ R = i+z-1,\ L \le j \le R,\ j' = j - L$
  - $\boldsymbol{case\ 1.}\quad \boldsymbol{j' + \mathcal{Z}(j') < z \implies \mathcal{Z}(j) = \mathcal{Z}(j')}$

								L					R	
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
A	a	b	a	b	a	b	e	a	b	a	b	a	b	f
$\mathcal{Z}$	14	0	4	0	2	0	0	6	0	4	0	2	0	0

made by Jingfei

Z Algorithm

- 假設一個字串 A 中，$\mathcal{Z}(i) = z$
  - $A[k] = A[i+k],\ if\ 0 \le k < z$
  - $A[z] \neq A[i+z]$
  - 令 $L = i,\ R = i+z-1,\ L \le j \le R,\ j' = j - L$
  - $\boldsymbol{case\ 2.}\quad \boldsymbol{j' + Z(j') > z} \implies \boldsymbol{Z(j) = R - j + 1}$ (j 到 R 的長度)

								L					R	
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
A	a	b	a	b	a	b	e	a	b	a	b	a	b	f
$\mathcal{Z}$	14	0	4	0	2	0	0	6	0	4	0	2	0	0

made by Jingfei

Z Algorithm

- 假設一個字串 A 中，$\mathcal{Z}(i) = z$
  - $A[k] = A[i+k],\ if\ 0 \le k < z$
  - $A[z] \neq A[i+z]$
  - 令 $L = i,\ R = i+z-1,\ L \le j \le R,\ j' = j - L$
  - $\boldsymbol{case\ 3.}\quad \boldsymbol{j' + Z(j') = z \implies Z(j) \ge Z(j')}$ (剛好在邊界)

								L					R	
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
A	a	b	a	b	a	b	e	a	b	a	b	a	b	f
$\mathcal{Z}$	14	0	4	0	2	0	0	6	0	4	0	2	0	0

made by Jingfei

Z Algorithm

- 假設一個字串 A 中，$\mathcal{Z}(i) = z$
  - $A[k] = A[i+k],\ if\ 0 \le k < z$
  - $A[z] \neq A[i+z]$
  - 更新 $L = j,\ R = j + Z(j) - 1,\ L \le j \le R,\ j' = j - L$
  - $\boldsymbol{case\ 3.}\quad \boldsymbol{j' + Z(j') = z \implies Z(j) \ge Z(j')}$

										L			R	
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
A	a	b	a	b	a	b	e	a	b	a	b	a	b	f
$\mathcal{Z}$	14	0	4	0	2	0	0	6	0	4	0	2	0	0

j' j

made by Jingfei

Z Algorithm

- 只會前進不會後退 → $\mathcal{O}(N)$

```
 1 L = R = 0;
 2 for(int i=1; i<len; i++){
 3     if(i>R) Z[i]=0; //Case 0
 4     else{
 5         int ip = i - L;
 6         if(ip+Z[ip] < Z[L]) Z[i]=Z[ip]; //Case 1
 7         else Z[i]=R-i+1; //Case 2, 3
 8     }
 9     while(i+Z[i] < len && A[i+Z[i]]==A[Z[i]])
10         Z[i]++;
11     if(i+Z[i]-1 > R){
12         L=i;
13         R=i+Z[i]-1;
14     }
15 }
```

Z Algorithm

- 只會前進不會後退 → $\mathcal{O}(N)$

```
 1 L = R = 0;
 2 for(int i=1; i<len; i++){
 3     if(i>R) Z[i]=0; //Case 0
 4     else{
 5         int ip = i - L;
 6         if(ip+Z[ip] < Z[L]) Z[i]=Z[ip]; //Case 1
 7         else Z[i]=R-i+1; //Case 2, 3
 8     }
 9     while(i+Z[i] < len && A[i+Z[i]]==A[Z[i]])
10         Z[i]++;
11     if(i+Z[i]-1 > R){
12         L=i;
13         R=i+Z[i]-1;
14     }
15 }
```

Case 0：
需要更新邊界，
跳到第11行

Z Algorithm

- 只會前進不會後退 → $\mathcal{O}(N)$

```
 1 L = R = 0;
 2 for(int i=1; i<len; i++){
 3     if(i>R) Z[i]=0; //Case 0
 4     else{
 5         int ip = i - L;
 6         if(ip+Z[ip] < Z[L]) Z[i]=Z[ip]; //Case 1
 7         else Z[i]=R-i+1; //Case 2, 3
 8     }
 9     while(i+Z[i] < len && A[i+Z[i]]==A[Z[i]])
10         Z[i]++;
11     if(i+Z[i]-1 > R){
12         L=i;
13         R=i+Z[i]-1;
14     }
15 }
```

Case 3 後續：
z index 剛好在邊界，
更新 z 值

Z Algorithm

- 只會前進不會後退 → $\mathcal{O}(N)$

```
 1 L = R = 0;
 2 for(int i=1; i<len; i++){
 3     if(i>R) Z[i]=0; //Case 0
 4     else{
 5         int ip = i - L;
 6         if(ip+Z[ip] < Z[L]) Z[i]=Z[ip]; //Case 1
 7         else Z[i]=R-i+1; //Case 2, 3
 8     }
 9     while(i+Z[i] < len && A[i+Z[i]]==A[Z[i]])
10         Z[i]++;
11     if(i+Z[i]-1 > R){
12         L=i;
13         R=i+Z[i]-1;
14     }
15 }
```

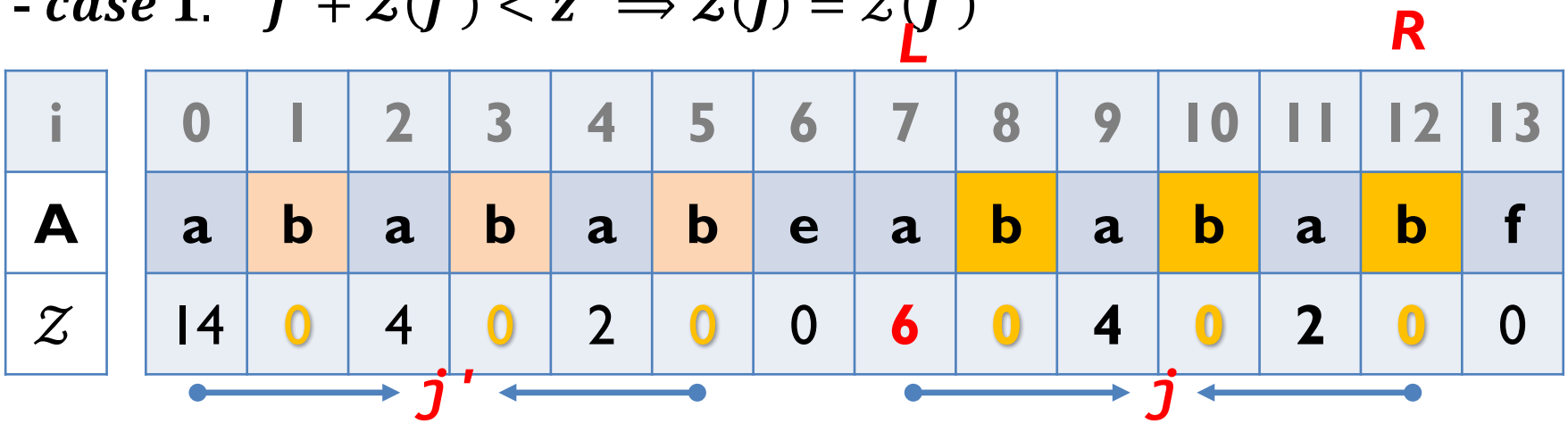
For Case 0, 3：更新邊界

Z Algorithm

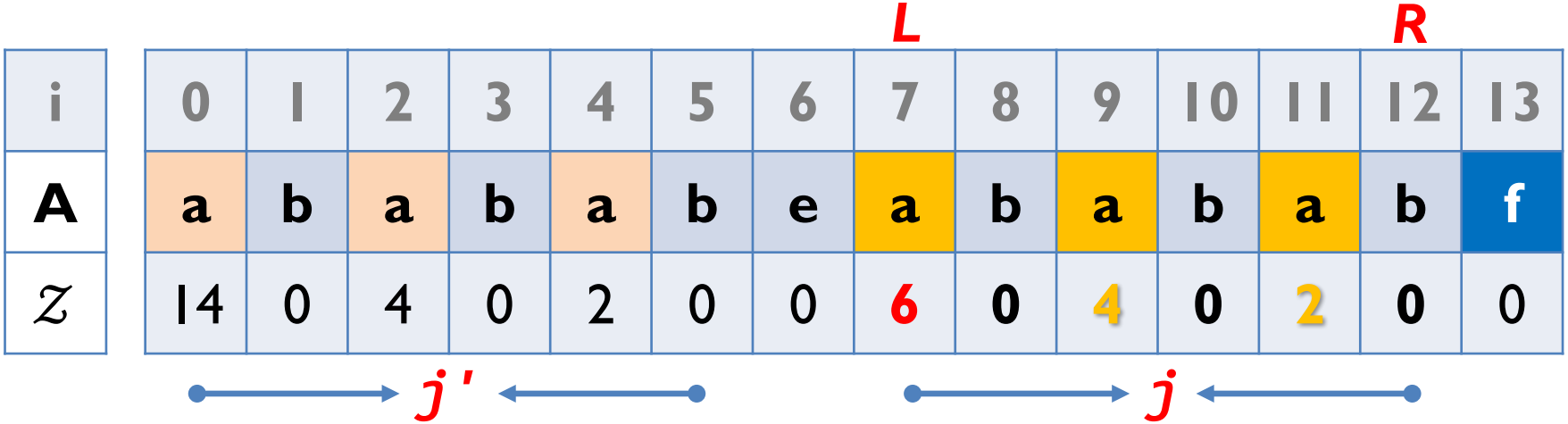
- Matching ?
- 把兩個字串接起來，B 在前，A 在後，
  中間用一個沒有出現過的字元連接
- 如 A = "abaab", B = "aab"
  另 S = "aab$abaab"
  → 發現 B 是否能在 A 的某個位置被匹配， 只要
  看那個位置的 Z value 是否等於 B 的長度
- 最長回文子字串

Z Algorithm – 補充

- 比較 case 1, 2, 3:
  - $case\ 1.\quad j' + Z(j') < z \implies Z(j) = Z(j')$



  - $case\ 2, 3.\quad j' + Z(j') \geq z \implies Z(j) = R - j + 1$ (j 到 R 的長度)



Z Algorithm – 補充

- 都是取 case 1, 2, 3 中最小值

```
 1 L = R = 0;
 2 for(int i=1; i<len; i++){
 3     if(i>R) Z[i]=0; //Case 0
 4     else{
 5         int ip = i-L;
 6         if(ip+Z[ip] < Z[L]) Z[i]=Z[ip]; //Case 1
 7         else Z[i]=R-i+1; //Case 2, 3
 8     }
 9     while(i+Z[i] < len && A[i+Z[i]]==A[Z[i]])
10         Z[i]++;
11     if(i+Z[i]-1 > R){
12         L=i;
13         R=i+Z[i]-1;
14     }
15 }
```

將其濃縮

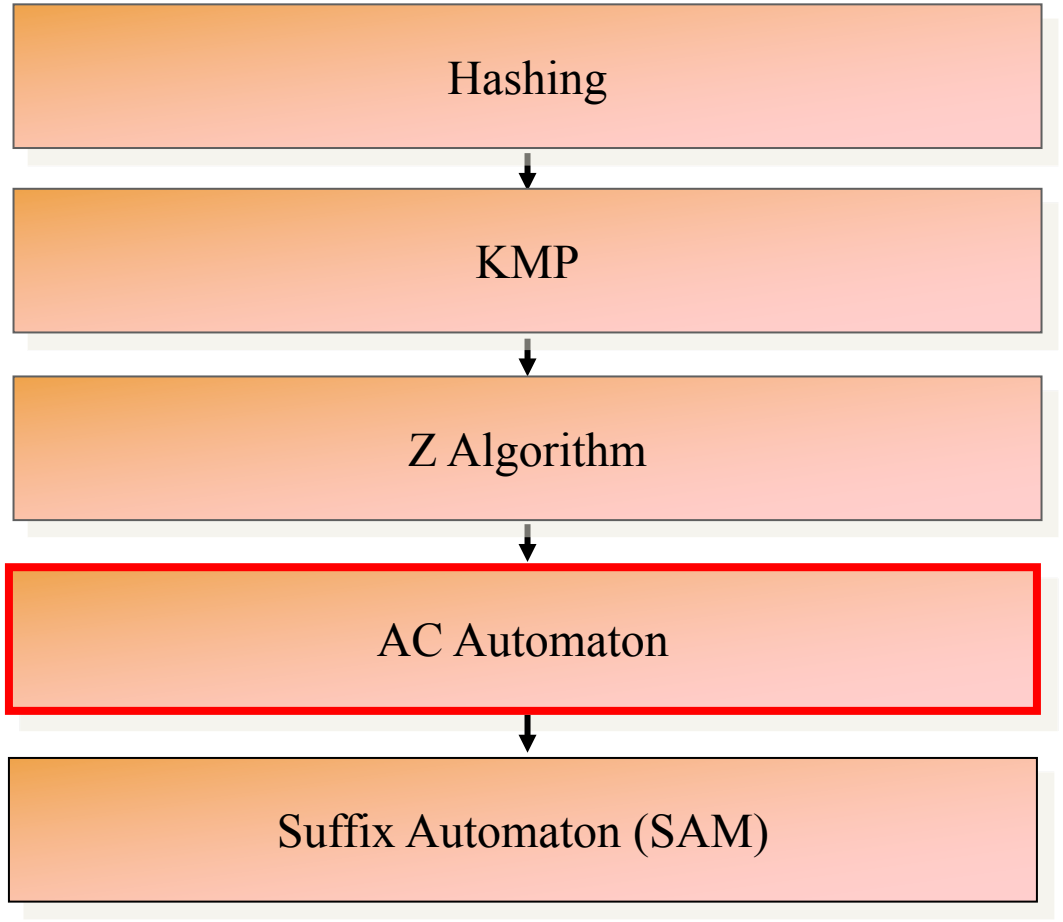
Z Algorithm – 補充

- 都是取 case 1, 2, 3 中最小值

```
 1 L = R = 0;
 2 for(int i=1; i<len; i++){
 3     if(i>R) Z[i]=0; //Case 0
 4     else{
 5         int ip = i - L;
 6         if(i+Z[ip] < R) Z[i]=Z[ip]; //Case 1
 7         else Z[i]=R-i+1; //Case 2, 3
 8     }
 9     while(i+Z[i] < len && A[i+Z[i]]==A[Z[i]])
10         Z[i]++;
11     if(i+Z[i]-1 > R){
12         L=i;
13         R=i+Z[i]-1;
14     }
15 }
```

```
Z[i] = R>i ? min(R-i+1, Z[Z[L]-(R-i+1)]) : 0;
或者
Z[i] = i>R ? 0 : (i-L+Z[i-L] < Z[L] ? Z[i-L] : R-i+1);
```

Outline



AC Automaton

- KMP 複雜度 $\mathcal{O}(|\mathrm{A}| + |\mathrm{B}|)$
- 多字串匹配
  1. 一個字串 B 匹配很多字串 A_i

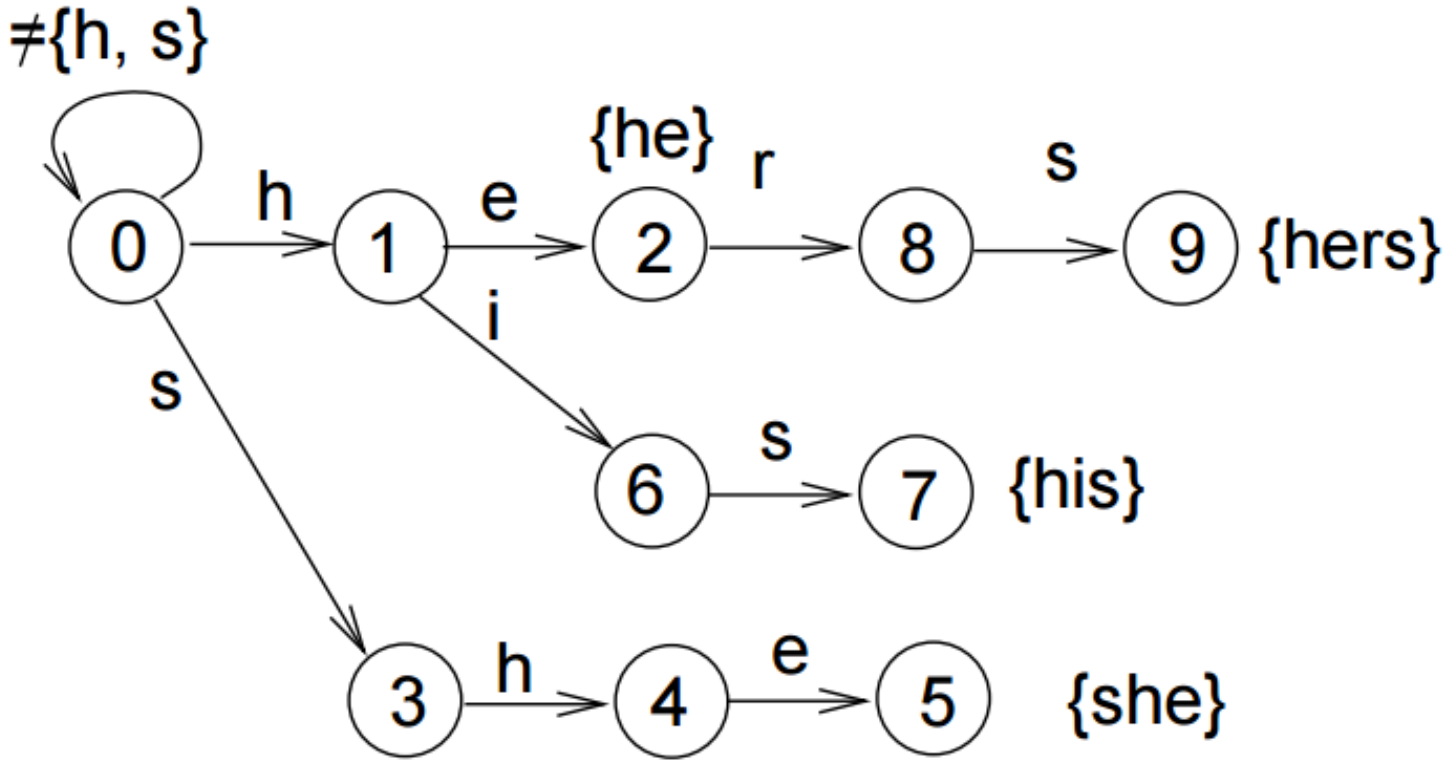
     $\rightarrow \mathcal{O}(\sum|\mathrm{A}_i| + |\mathrm{B}|)$

     → 線性
  2. 很多字串 B_i 匹配一個字串 A

     $\rightarrow \mathcal{O}(n|A| + \sum|B_i|)$

     → 弱弱的
- Trie：儲存多個字串
- AC 自動機 = KMP + Trie

AC Automaton



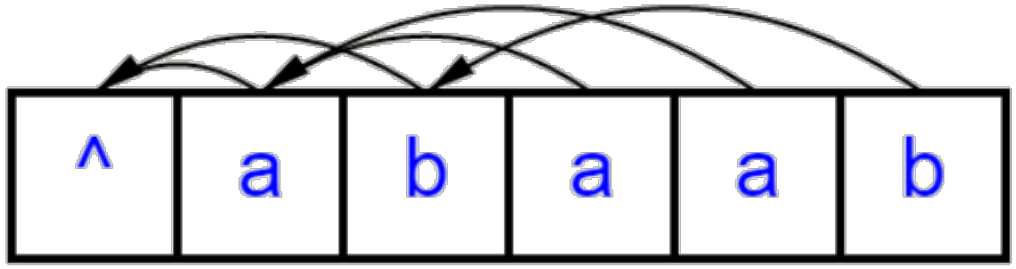
- Trie：儲存多個字串
- AC 自動機 = KMP + Trie

made by Jingfei

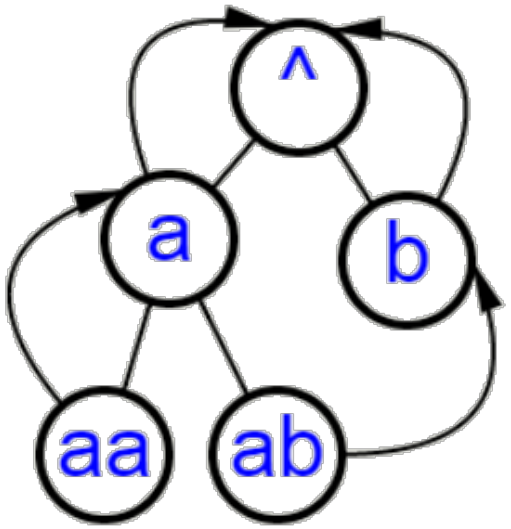
AC Automaton

- 比較 Fail function (圖)

KMP:



AC自動機:



AC Automaton

- 比較 Fail function (定義)
- KMP
  - $\mathcal{F}_B(i) =$
    $$\begin{cases} \max\{k:\ P_B(k) = B[0,k] = B[i-k,i]\}, if\ i \neq 0\ and\ at\ least\ a\ \boldsymbol{k}\ exists \\ -1,\ else \end{cases}$$
  - A[0,k] 是 A[0,i] 的前綴
- AC Automaton
  - $\mathfrak{F}_B(v) = \begin{cases} u,\ if\ \mathrm{B}_T(u) \text{ 是 } \mathrm{B}_T(v) \text{ 的前綴且 } |\mathrm{S}_T(u)| \text{ 最大} \\ v_0,\ else \end{cases}$
  - $\mathrm{B}_T(u)$ 是 $\mathrm{B}_T(v)$ 的前綴

AC Automaton

- 比較 Fail function (匹配失敗)
- KMP
  - 沿著 $\mathcal{F}(i),\ \mathcal{F}^2(i),\ \ldots$ 嘗試，直到 $\mathcal{F}^t(i) = -1$
- AC Automaton
  - 沿著 $\mathfrak{F}(v),\ \mathfrak{F}^2(v),\ \ldots$ 嘗試，直到 $\mathfrak{F}^t(v) = v_0\ (v_0\colon \text{root})$

AC Automaton

- 比較 Fail function (構造)
- KMP
  - 利用 $\mathcal{F}(i-1)$ 求出 $\mathcal{F}(i)$
- AC Automaton
  - 利用 $\mathfrak{F}(u)$ 求出 $\mathfrak{F}(v)$，u 為 v 的父節點
  - use BFS

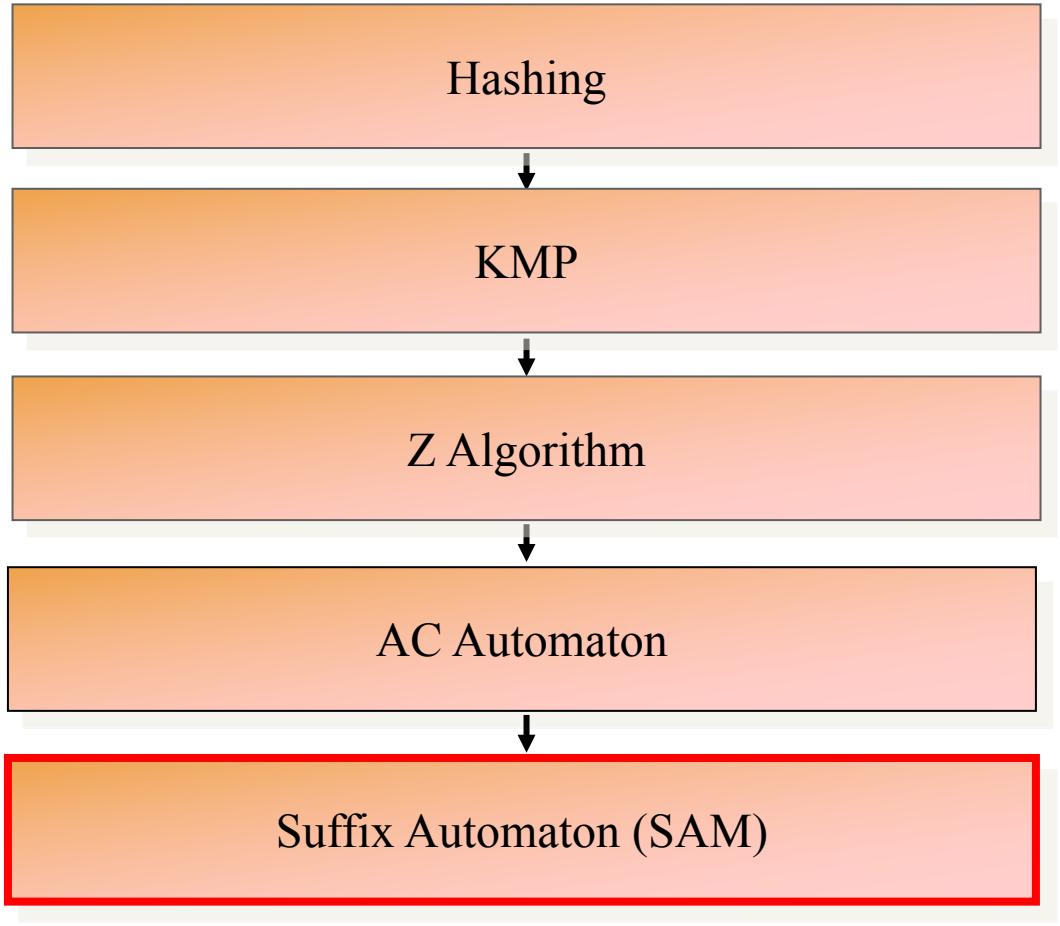
AC Automaton

- $\mathcal{O}(\,|A| + \sum|B|\,)$
-

```
 1 root->fail = NULL;
 2 queue< Node* > que;
 3 que.push_back(root);
 4 while ( !que.empty() ) {
 5     Node *fa = que.front(); que.pop_front();
 6
 7     for (auto it = fa->child.begin();
 8               it != fa->child.end(); it++) {
 9         Node *cur = it->second, *ptr = fa->fail;
10         while ( ptr && !ptr->child.count(it->first) )
11             ptr = ptr->fail;
12
13         cur->fail = ptr ? ptr->child[it->first] : root;
14         que.push(cur);
15     }
16 }
```

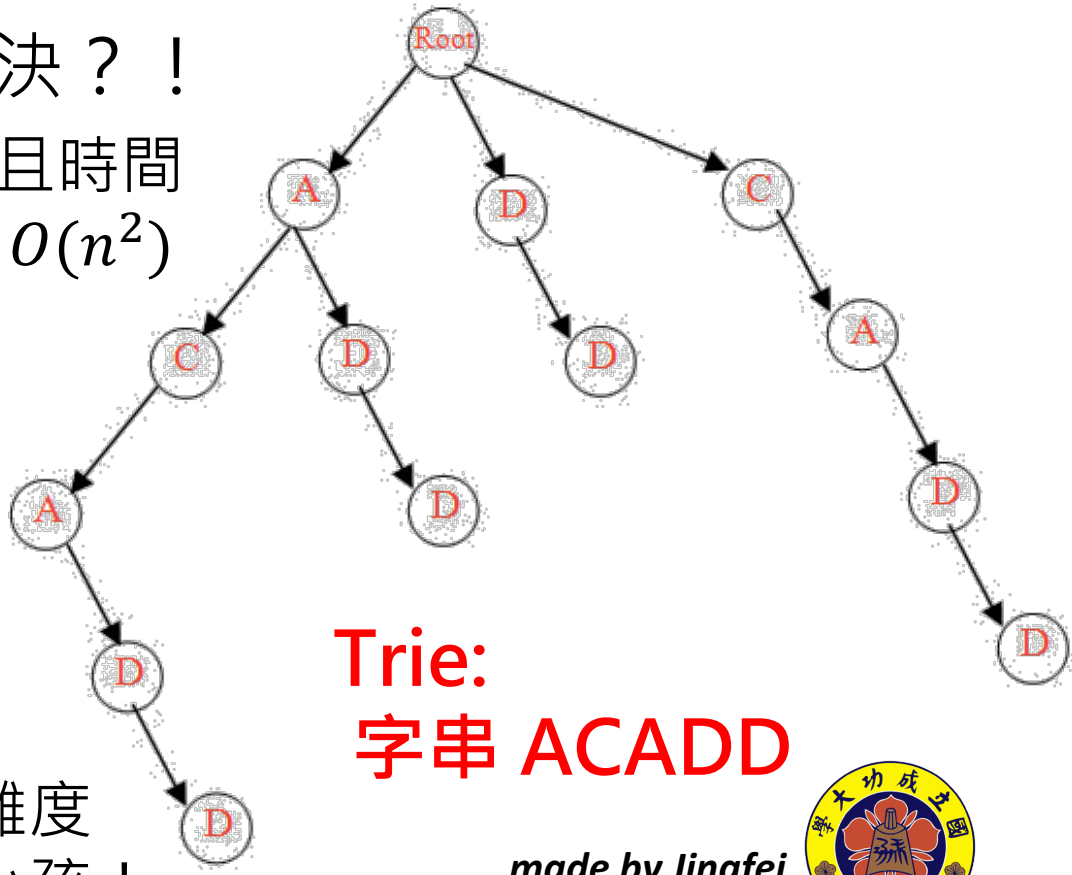
made by Jingfei

Outline



Suffix Automaton (SAM)

- 後綴自動機，可找出所有後綴的組合
  - 超強大，幾乎所有字串題目都可以用它解決
- 字典樹 (Trie) 就可解決？！
  - 正確，但空間用很多且時間複雜度也很高 $O(n^2)$
- 觀察：
  - 幾乎所有 node 都只有一個小孩
  - 很多一樣的小孩
- 解決方法：
  - 在可以降低時間複雜度的前提下利用共用小孩！



Trie:
字串 ACADD

made by Jingfei

Suffix Automaton (SAM)

- 每個節點存的資訊：
  - son[26]：
    原本的字串加上某個字母後生成的字串在後綴自動機的位置 (和字典樹一樣)
    如果 son[i] 不存在，代表此子字串是不存在的
  - pre：
    返回上一個可以接收後綴的節點 (如果該節點可以接收新的後綴，那麼 pre 的節點也一定可以接收後綴)
    需注意，pre 不是返回它的父節點 (可能有多個爸爸)
  - step：
    從 root 走到該節點，最多需要幾步

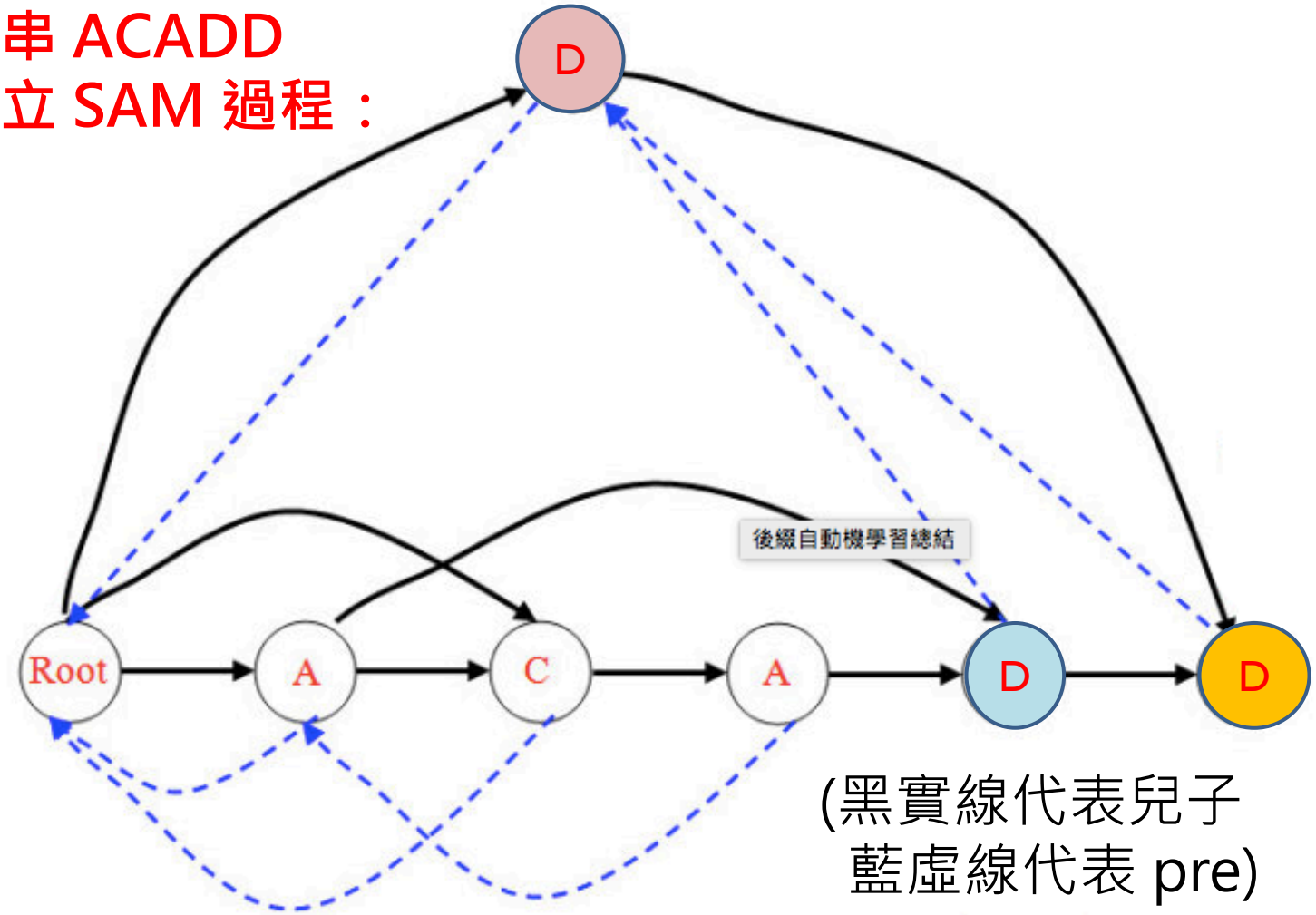
Suffix Automaton (SAM)

- 後綴自動機的三個特性：
  假設已建立字串 t 的後綴自動機
  1. 從 root 到任意節點 p 所有路徑組成的字串，都是 t 的子字串 (不一定是後綴)
  2. 如果節點 p 可以接收新後綴，那麼從 root 到 p 的所有路徑組成的字串，都是 t 的後綴
  3. 如果節點 p 可以接收新後綴，那麼 p 的 pre 節點也可以接收後綴，反過來不成立
- 可見下頁圖，假設黃色點為 p 節點，可以接收後綴；
  紅色點為 p 的 pre 節點 (p 的父節點之一)，可接收新後綴；
  藍色點為 p 節點另一個父節點，但不能接收新後綴

Suffix Automaton (SAM)

- 後綴自動機的三個特性：

字串 ACADD
建立 SAM 過程：



斤

可

(黑實線代表兒子
藍虛線代表 pre)

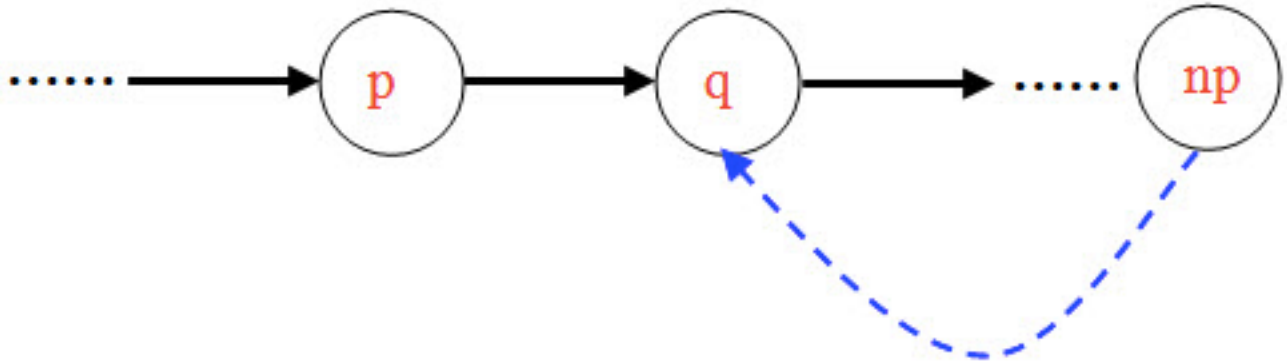
Suffix Automaton (SAM)

- 建立後綴自動機步驟：
  假設已建立字串 t 的後綴自動機，現在要插入字元 x，
  建立 tx 的後綴自動機
  1. 建立新的字元 x 的節點 np
  2. 找到之前最後建立的節點 r (一定滿足前頁特性2)，將 np 節點當作 r 的兒子 (此時 r 接收了後綴字元 x，目前不可以接收新的後綴字元，代表不能有 pre 接到 r)。
     並沿著 r 的 pre 往前找，直到找到有字元 x 兒子的節點
     2-1. 假設找到源頭都沒有找到節點有 x 兒子，直接將 np 的 pre 節點設為此源頭，並且此源頭多一個兒子 np
     2-2. 如果找到節點 p 有 x 兒子，假設目前 p 的 x 兒子是 q 節點，有 2 種情況：

Suffix Automaton (SAM)

2-2-1. step[q] = step[p] + 1

- 代表 q 一定是直接從 p 接過去，中間不會接其他節點，因此可以保證任何到達 p 的字串都是後綴
- 做法：將 np 的 pre 接到 q



Suffix Automaton (SAM)

2-2-2. step[q] > step[p] + 1

- 代表節點 p, q 之間有夾雜其他節點，不能直接用上一種做法
- 模仿前一種做法，想辦法讓新的節點 nq 同時能代表 q 又能保證 step[nq] = step[p] + 1

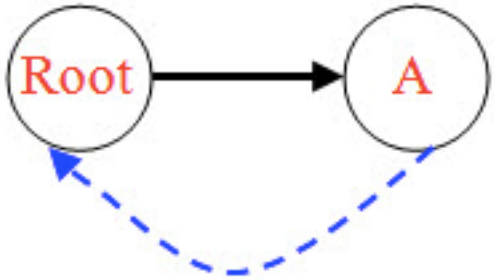
• 做法：

- 把 q 的 son 邊和 pre 邊都複製到 nq 上。
  把 nq 的 pre 改為 p，再把 q 和 np 的 pre 都改為 nq
  (因為現在 nq 代替了 q，所以 np 的 pre 是 nq。由特性 3 可知 nq 的 pre 只能是 p。同樣的，q 和 nq 也滿足特性 3，所以 q 的 pre 只能是 nq)
- 最後，更新 p 節點的 x 兒子，將 p 的 son[x]=q 改為 son[x]=nq (因為 nq 代替了 q)

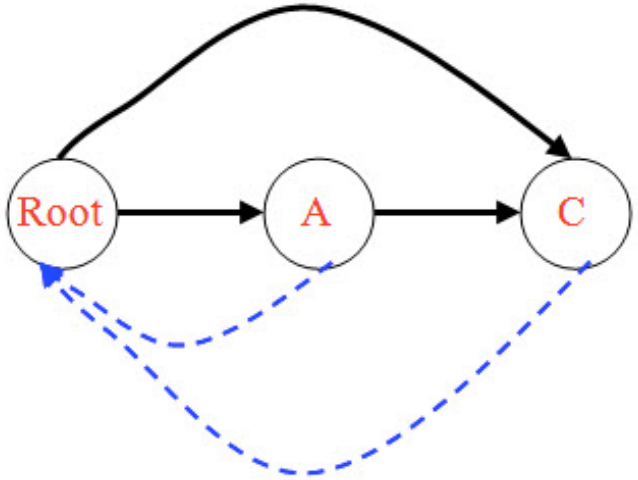
Suffix Automaton (SAM)

- ex. 建造 ACADD 後綴自動機
  (黑實線代表兒子，藍虛線代表 pre)

1. ACADD (步驟 1 → 2-1)



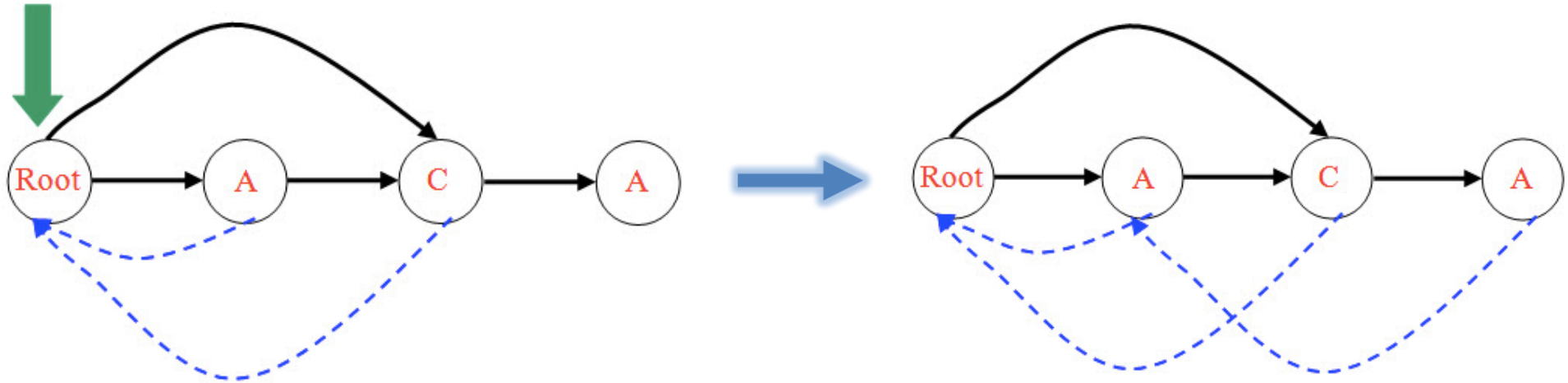
2. ACADD (步驟 1 → 2-1)



Suffix Automaton (SAM)

- ex. 建造 ACADD 後綴自動機
  (黑實線代表兒子，藍虛線代表 pre)

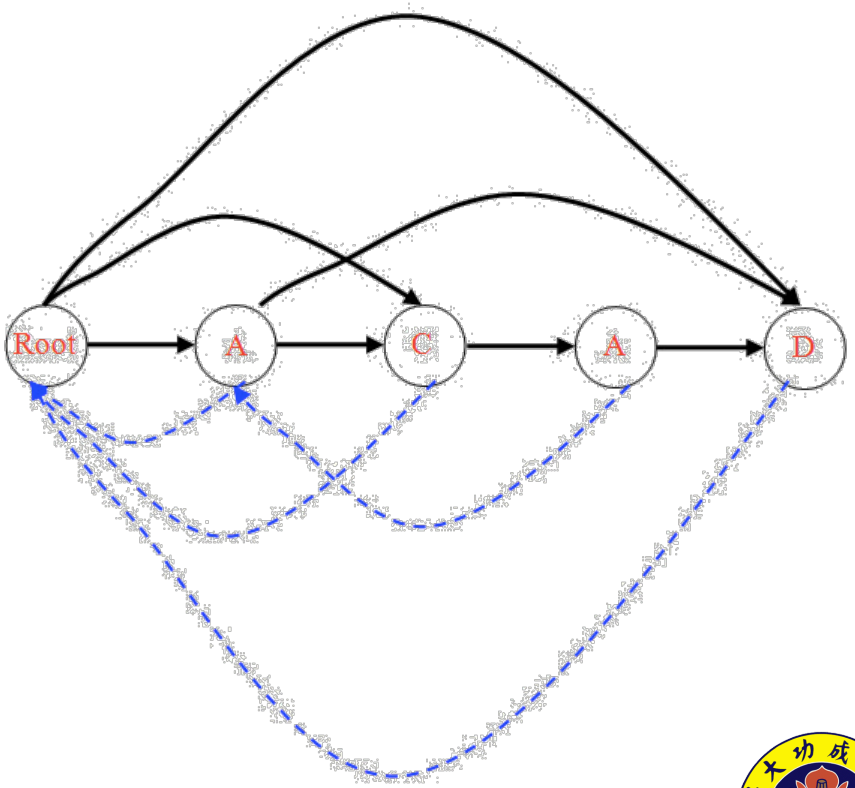
3. ACADD (步驟 1 → 2-2-1)
   - 檢查 C 的 pre 節點，發現 root 有 A 兒子 (進入步驟 2-2)
   - step[root] = 0, step[A] = 1 得知 step[A] = step[root] + 1
   - 因此，root 的 A 兒子現在有雙重身份：後綴 A 的最後一個字元，和後綴 ACA 的最後一個字元
   - 做法：將新建立的點 A 的 pre 連到 root 的 A 兒子



Suffix Automaton (SAM)

- ex. 建造 ACADD 後綴自動機
  (黑實線代表兒子，藍虛線代表 pre)

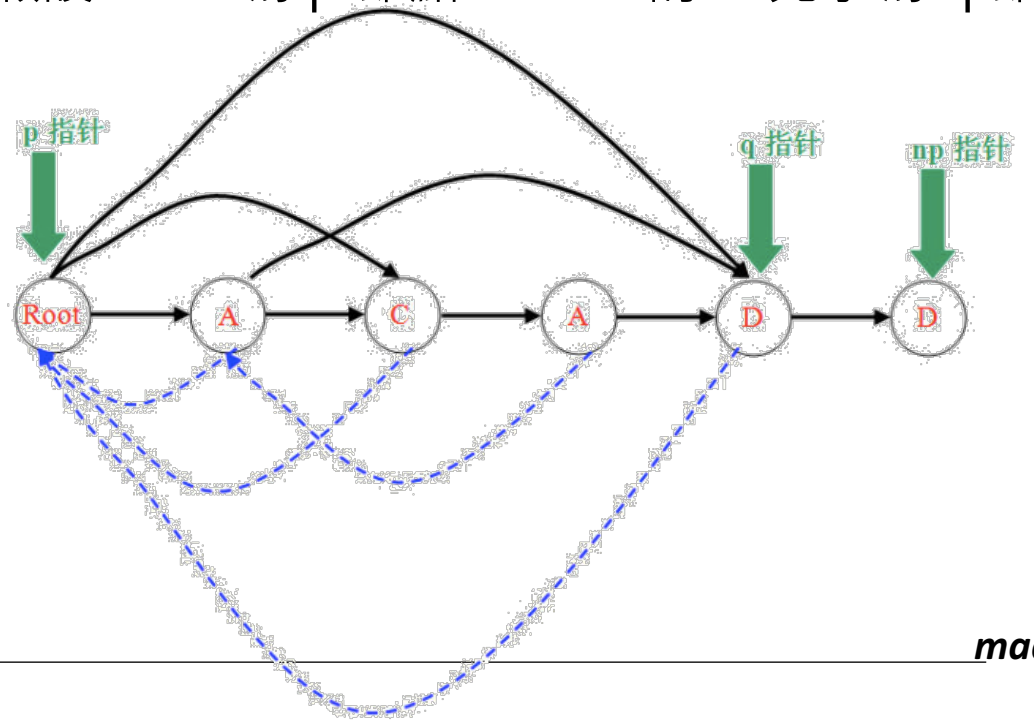
4. ACADD (步驟 1 → 2-1)



Suffix Automaton (SAM)

- ex. 建造 ACADD 後綴自動機
  (黑實線代表兒子，藍虛線代表 pre)

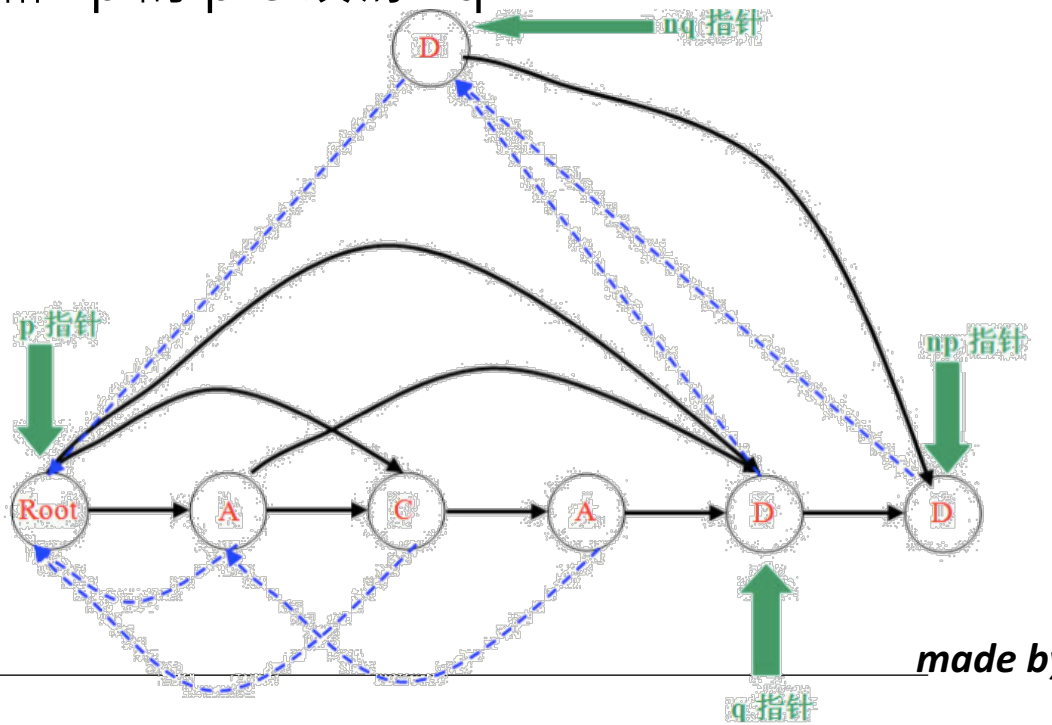
5. ACADD (步驟 1 → 2-2-2)
   - 檢查前一個 D 的 pre 節點，發現 Root 節點有 D 兒子 (步驟 2-2)
     因此假設 Root 為 p 節點，Root 的 D 兒子為 q 節點 (如下圖)



Suffix Automaton (SAM)

- ex. 建造 ACADD 後綴自動機

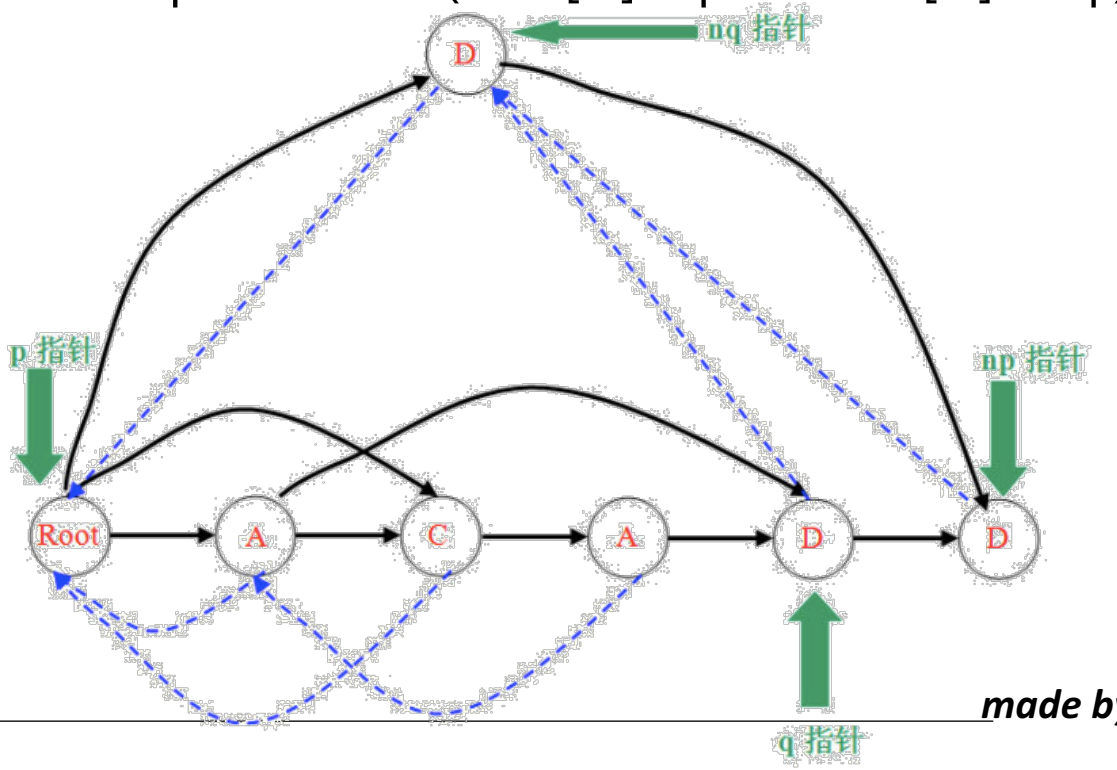
5. ACADD (步驟 1 → 2-2-2)
   - step[p] = 0, step[q] = 4 得知 step[p] > step[q] + 1
   - 做法：建立新節點 nq，把節點 q 複製到 nq，nq 的 pre 改為 p，再把 q 和 np 的 pre 改為 nq



Suffix Automaton (SAM)

- ex. 建造 ACADD 後綴自動機
  (黑實線代表兒子，藍虛線代表 pre)

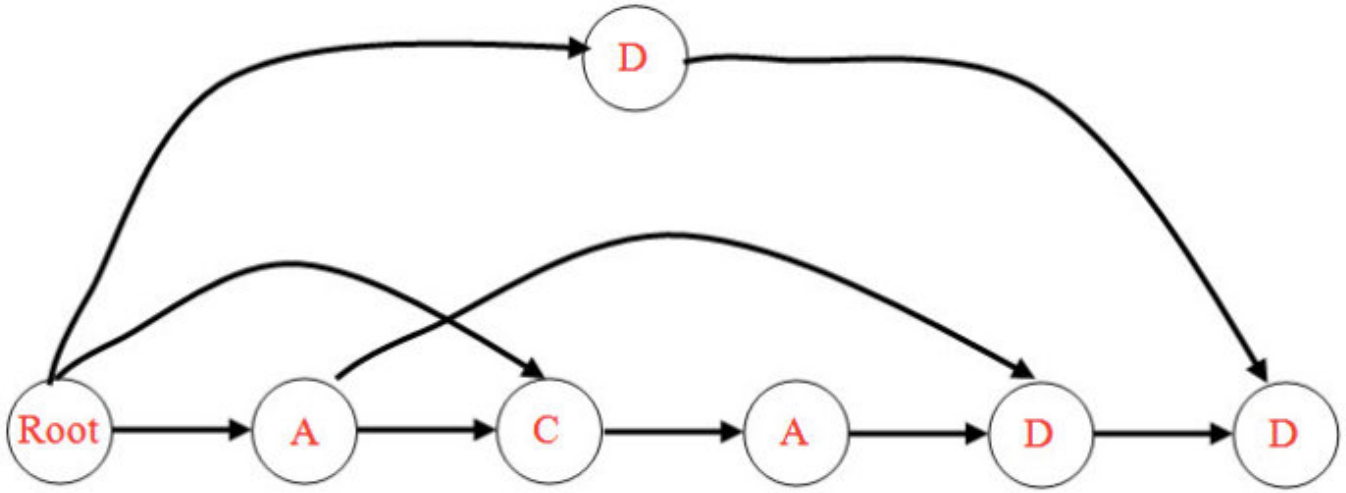
5. ACADD (步驟 1 → 2-2-2)
   - 最後，更新 p 的兒子 D (son[D]=q 改為son[D]=nq)



Suffix Automaton (SAM)

- ex. 建造 ACADD 後綴自動機
  (黑實線代表兒子，藍虛線代表 pre)

6. ACADD 完成圖
   - 把 pre 虛線拿掉，完成
   - 可把所有的後綴印出、所有子字串印出、尋找第 k 小 (利用拓撲) ...



   - 以上資料整理自：
     http://blog.sina.com.cn/s/blog_70811e1a01014dkz.html

Automaton – 範例 code

- 日月卦長的模板庫
  → [Aho-Corasick Automaton] AC自動機

- 日月卦長的模板庫
  → [Suffix Automaton] 後綴自動機

Automaton – 例題

- AC Automaton
  - SPOJ NSUBSTR 題解
  - SPOJ SUBLEX 題解
  - Codeforces 235C 題解
  - 這些都是作法很多 (Suffix Array, Suffix Tree...)，非常經典的問題，可以從中理解 SAM 的精妙之處
- Suffix Automaton
  - Uva 1449 題解 (模板題)
  - Uva 1502 題解
  - Uva 11019 題解

Reference

- 歷屆PPT..... (electron, free999, louis6340, ...)
- 2015 IOI camp 字串處理
  http://ioicamp.csie.org/content
  http://bobogei81123.github.io/ioi-lecture