Algorithm Design and Analysis

วิชาบังคับก่อน: 204251 และ 206281
ผู้สอน: ตอน 1 ผศ. เบญจมาศ ปัญญางาม เรียน ห้อง 207
ตอน 2 ผศ. ดร. จักริน ชวชาติ เรียน ห้อง 209

บทที่ 7
String Problem

String Problem

Contents

- Brute-force algorithm
- Rabin-Karp algorithm
- Knuth-Morris-Pratt's algorithm
- Boyer-Moore's algorithm

String Matching Problem

- กำหนดสตริง T = $t_1t_2t_3...t_n$ และ pattern P = $p_1p_2p_3...p_m$, $n \geq m$
- การจับคู่สตริงเป็นปัญหาการหาว่ามี pattern P ปรากฏอยู่ในตำแหน่งใดในสตริง T
  - ต้องการหา substring T(j) ของสตริง T ที่มีความยาว m และเริ่มต้นที่ตำแหน่ง j ใน T นั่นคือ T(j) = $t_jt_{j+1}t_{j+2}...t_{j+m-1}$ โดยที่ T(j) = P

Brute-force algorithm

การหาว่ามี pattern P ปรากฏอยู่ในตำแหน่งใดในสตริง T โดยการเปรียบเทียบ substring T(j) กับ pattern P สำหรับทุกตำแหน่ง j = 1 ... n-m-1

```
Input:  String T with n characters  and pattern P with m characters.
Output:  Starting index of the first substring of T matching P,
         or an indication that P is not a substring of T.

n  = length[ T ] , m =  length[ P ]
for ( i=0; i <= (n-m); i++) {
   j = 0;
   while ((j < m) && (T[ i+j ] == P[ j ]))
      j++;
   if (j == m)
      return i;   // match at i
 }
 return -1;   // no match
} // end
```

Brute-force algorithm

- การหาว่ามี pattern P ปรากฏอยู่ในตำแหน่งใดในสตริง T โดยการลองเปรียบเทียบ substring T(j) กับ pattern P สำหรับ j = 1... n-m-1
- กรณีพบ pattern P ปรากฏในตำแหน่งแรก (j =1) ของสตริง T
- กำหนด T= aaaaaaaaaaaaaaaaaaaB และ pattern P = aaaa , m=4

```
aaaaaaaaaaaaaaaaaaaB      กรณี j = 1  เปรียบเทียบ 4  ครั้ง
aaaa
```

จำนวนครั้งในการเปรียบเทียบทั้งหมด คือ (m)
Best case time complexity is O(m)

Brute-force algorithm

- การหาว่ามี pattern P ปรากฏอยู่ในตำแหน่งใดในสตริง T โดยการลองเปรียบเทียบ substring T(j) กับ pattern P สำหรับ j = 1 ... n-m-1
- กรณีไม่พบ pattern P= bbbB , m=4 ในสตริง T

```
aaaaaaaaaaaaaaaaaaaB      กรณี j = 1      เปรียบเทียบ 1 ครั้ง
bbbB
aaaaaaaaaaaaaaaaaaaB      กรณี j = 2      เปรียบเทียบ 1 ครั้ง
 bbbB
  . . .
aaaaaaaaaaaaaaaaaaaB      กรณี j = n-m-1  เปรียบเทียบ 1 ครั้ง
                  bbbB
```

จำนวนครั้งในการเปรียบเทียบทั้งหมด คือ (n-m+1)
Best case time complexity is O(n)

Brute-force algorithm

- การหาว่ามีสตริง P ปรากฏอยู่ในตำแหน่งใดในสตริง T โดยการลองเปรียบเทียบ substring T(j) กับ pattern P สำหรับ j = 1 ... n-m-1
- กรณีพบ pattern P = aaaB, m=4 ในสตริง T ที่ตำแหน่งสุดท้าย

```
aaaaaaaaaaaaaaaaaaaB  กรณี j = 1          เปรียบเทียบ 4 ครั้ง
aaaB
aaaaaaaaaaaaaaaaaaaB  กรณี j = 2          เปรียบเทียบ 4 ครั้ง
 aaaB
...
aaaaaaaaaaaaaaaaaaaB  กรณี j = n-m-1      เปรียบเทียบ 4 ครั้ง
                  aaaB
```

จำนวนครั้งในการเปรียบเทียบทั้งหมด คือ m (n-m+1)
Worst case time complexity is O(mn)

Rabin-Karp algorithm

- กำหนดให้เรามี hash function (H) สำหรับหาค่า hash value สำหรับสตริง s ใดๆ
- กำหนดสตริง T = aaaaaaaaaaaaaaaaaaaB, pattern P = aaaB, m=4

สมมติค่า hash value สำหรับสตริง “aaaa” และ “aaaB” ดังนี้

H(“aaaa”) = 80 และ H(“aaaB”) = 87

```
aaaaaaaaaaaaaaaaaaaB  กรณี j = 1          เปรียบเทียบ 1 ครั้ง
aaaB
```

พบว่า 80 ≠ 87 จึงต้องเปรียบเทียบในตำแหน่งถัดไป (j = 2)

Rabin-Karp algorithm

- กำหนดสตริง T = aaaaaaaaaaaaaaaaaaaB, pattern P = aaaB, m=4
- สมมติค่า hash value สำหรับสตริง “aaaa” และ “aaaB” ดังนี้

H(“aaaa”) = 80 และ H(“aaaB”) = 87

กรณีพบ pattern P = aaaB, m=4 ในสตริง T ที่ตำแหน่งสุดท้าย

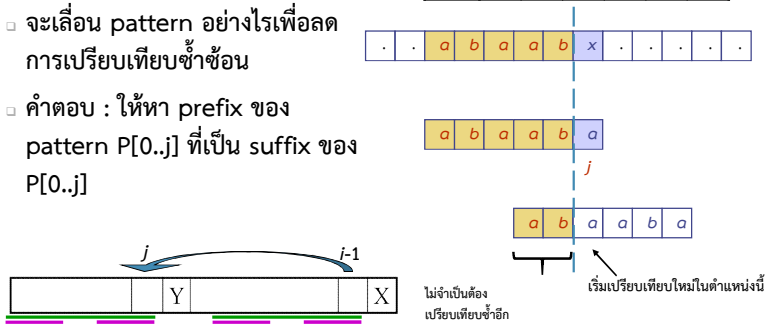
aaaaaaaaaaaaaaaaaaaB aaaB	กรณี j = 1	เปรียบเทียบ 1 ครั้ง (พบว่า 80 ≠ 87)
aaaaaaaaaaaaaaaaaaaB aaaB	กรณี j = 2	เปรียบเทียบ 1 ครั้ง (พบว่า 80 ≠ 87)
...		
aaaaaaaaaaaaaaaaaaaB aaaB	กรณี j = n-m-1	เปรียบเทียบ 1 ครั้ง (พบว่า 80 = 87)

Worst case time complexity is O(mn) // hash function H ใช้ O(m)

Knuth-Morris-Pratt’s algorithm

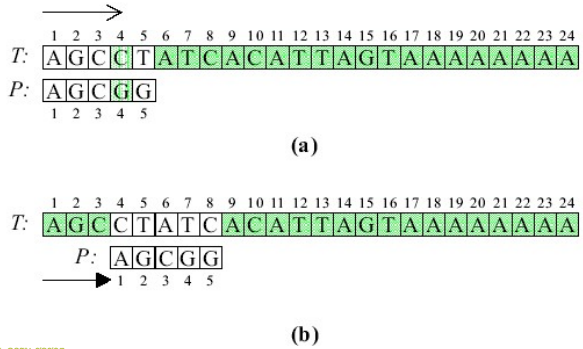
- เมื่อพบการเปรียบเทียบอักขระแล้วพบว่า p[j] ≠T[i]
- จะเลื่อน pattern อย่างไรเพื่อลดการเปรียบเทียบซ้ำซ้อน
- คำตอบ : ให้หา prefix ของ pattern P[0..j] ที่เป็น suffix ของ P[0..j]

j	0	1	2	3	4	5
P[j]	a	b	a	a	b	a
F(j)	0	0	1	1	2	3



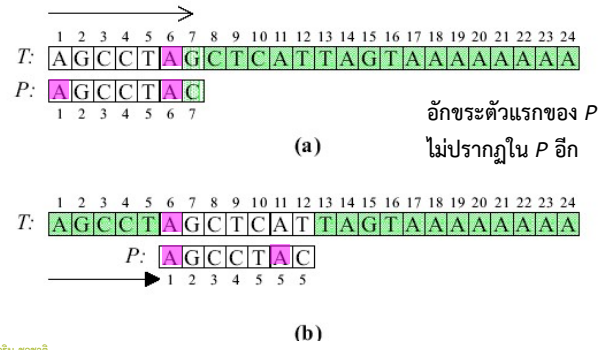
First Case for KMP Algorithm

- อักขระตัวแรกของ P ไม่ปรากฏใน P อีก
- เราจะเลื่อนไปยัง T_4 เนื่องจาก $T_4 \neq P_4$ in (a).



Second Case for KMP Algorithm

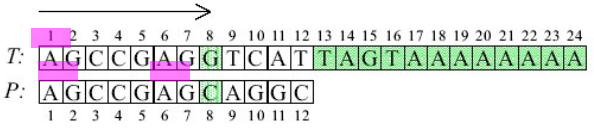
- อักขระตัวแรกของ P ปรากฏใน P อีก
- $T_7 \neq P_7$ ใน (a) เราจะเลื่อนไปยัง T_6 เนื่องจาก $P_6=P_1=T_6$.



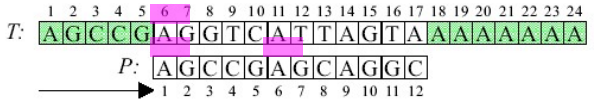
อักขระตัวแรกของ P ไม่ปรากฏใน P อีก

Third Case for KMP Algorithm

- prefix ของ P ปรากฏใน P อีก
- $T_8 \neq P_8$ ใน (a) เราจะเลื่อนไปยัง T_6 เนื่องจาก $P_{6,7}=P_{1,2}=T_{6,7}$.



(a)



(b)

Knuth-Morris-Pratt's algorithm : หา Prefix

- เริ่มต้น i =0, กำหนดค่า Prefix(F) ของตำแหน่ง P[0] เป็น 0

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
P	b	c	b	a	b	c	b	a	e	b	c	b	a	b	c	b	a
Prefix	0																

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
P	b	c	b	a	b	c	b	a	e	b	c	b	a	b	c	b	a
Prefix	0	0															

F[i] คือ ตำแหน่ง prefix ของ P[i]

i = 1, จะหา F[1] ที่เป็น Prefix (F) ตำแหน่ง P[1] = c

t = F[i-1] = F[0] = 0 พบว่า P[t] = P[0]= b

เนื่องจาก P[i] ≠ P[t] (c ≠ b) และ t = 0 ดังนั้น f [1] = 0

Knuth-Morris-Pratt's algorithm : หา Prefix

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
P	b	c	b	a	b	c	b	a	e	b	c	b	a	b	c	b	a
Prefix	0	0	1														

i=2, จะหา F[2] ที่เป็น Prefix (F) ตำแหน่ง P[2] = b

t = F[i-1] = F[1] = 0 ดังนั้น P[t] = P[0] = b

เนื่องจาก P[i]=P[t] = b ดังนั้น F[2] = t+1 = 1

Knuth-Morris-Pratt's algorithm : หา Prefix

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
P	b	c	b	a	b	c	b	a	e	b	c	b	a	b	c	b	a
Prefix	0	0	1	0													

i=3, จะหา F[3] ที่เป็น Prefix (F) ตำแหน่ง P[3] = a

t = F[i-1] = F[2] = 1 ดังนั้น P[t] = P[1] = c

เนื่องจาก P[i] ≠ P[t] (a ≠ c) แต่ t ≠ 0 ดังนั้นพิจารณา t ค่าใหม่

t = F[t-1] = F[0] = 0 พบว่า พบว่า P[t] = P[0] = b

เนื่องจาก P[i] ≠ P[t] (a ≠ b) และ t = 0 ดังนั้น F(3) = 0

Knuth-Morris-Pratt's algorithm : หา Prefix

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
P	b	c	b	a	b	c	b	a	e	b	c	b	a	b	c	b	a
Prefix	0	0	1	0	1												

i = 4 , หา Prefix (F) ตำแหน่ง P[4] = b

t = F[i-1] = F[3] = 0 พบว่า P[t] = P[0] = b

เนื่องจาก P[i]=P[t] = b ดังนั้น F [4] = t+1 = 0+1= 1

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
P	b	c	b	a	b	c	b	a	e	b	c	b	a	b	c	b	a
Prefix	0	0	1	0	1	2											

i = 5 , หา Prefix (F) ตำแหน่ง P[5] = c

t = F[i-1] = F[4] = 1 พบว่า P[t] = P[1] = c

เนื่องจาก P[i]=P[t] = c ดังนั้น F [5] = t+1 = 2

Knuth-Morris-Pratt's algorithm : หา Prefix

i = 8 , หา Prefix (F) ตำแหน่ง P[8] = e

t = F[i-1] = F[7] = 4 พบว่า P[t] = P[4] = b

เนื่องจาก P[i] ≠ P[t] (e ≠ b) แต่ t ≠ 0 ดังนั้นพิจารณา t ค่าใหม่

t = F[t-1] = F[3] = 0 พบว่า พบว่า P[t] = P[0] = b

เนื่องจาก P[i] ≠ P[t] (e ≠ b) และ t = 0 ดังนั้น F(8) = 0

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
P	b	c	b	a	b	c	b	a	e	b	c	b	a	b	c	b	a
Prefix	0	0	1	0	1	2	3	4	0								

Knuth-Morris-Pratt's algorithm : หา Prefix

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
P	b	c	b	a	b	c	b	a	e	b	c	b	a	b	c	b	a
Prefix	0	0	1	0	1	2	3	4	0	1	2	3	4	5	6	7	

i = 15 , หา Prefix (F) ตำแหน่ง P[15] = b

t = F[i-1] = F[14] = 6 พบว่า P[t] = P[6] = b

เนื่องจาก P[i]=P[t] = b ดังนั้น F [15] = t+1 = 0+1= 7

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
P	b	c	b	a	b	c	b	a	e	b	c	b	a	b	c	b	a
Prefix	0	0	1	0	1	2	3	4	0	1	2	3	4	5	6	7	8

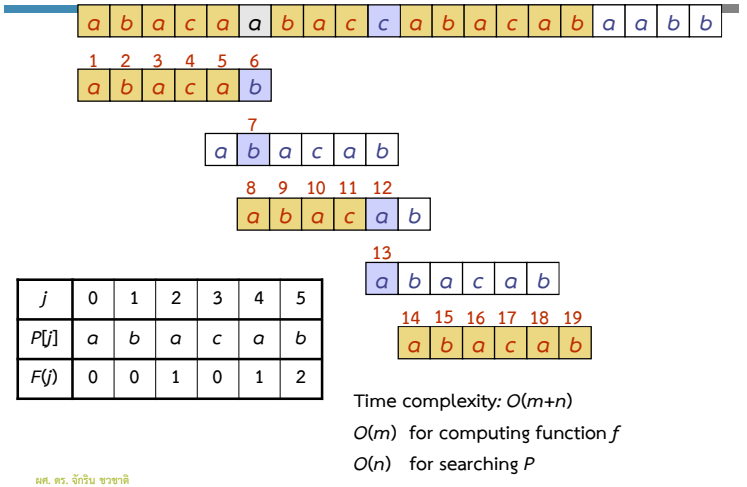
Knuth-Morris-Pratt's algorithm : หา Prefix

```
F[] KMTFail (P[])
        F[0] ← 0
        i ← 1
        t ← 0 // t = F(i-1)
        while i < m
          if P[i] = P[t]  {we have matched t + 1 characters}
                F[i] ←  t + 1
                i ← i + 1
                t ← t + 1 // t = F(i-1)
           else if  t > 0 then {use failure function to shift P}
                t ← F[t - 1]
           else
                F[i] ← 0 { no match }
                i ← i + 1
                t ← 0
Return(F)
```

Knuth-Morris-Pratt's algorithm

```
F ← KMPFail(P) {build failure function}
  i ← 0
  j ← 0
  while i < n do
    if P[j] = T[i] then
    if j = m - 1 then
          return i - m - 1 {a match}
    i ← i + 1
    j ← j + 1
    else if j > 0 then {no match, but we have advanced}
        j ← F(j-1) {j indexes just after matching prefix in P}
        else
        i ← i + 1
  return "There is no substring of T matching P"
```


Knuth-Morris-Pratt's algorithm : Example



j	0	1	2	3	4	5
P[j]	*a*	*b*	*a*	*c*	*a*	*b*
F(j)	0	0	1	0	1	2

Time complexity: $O(m+n)$

$O(m)$ for computing function f

$O(n)$ for searching P

Boyer-Moore's algorithm

- จะเปรียบเทียบอักขระใน pattern จากขวามาซ้าย
- จะนำความรู้ที่ได้จากการเทียบอักขระมาเพื่อขยับการเทียบใช้ heuristic 2 แบบได้แก่
  - Bad character
  - Good suffix

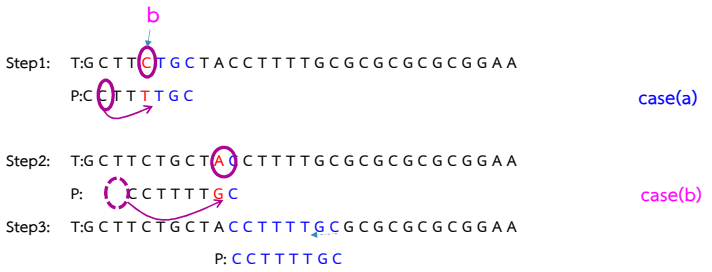
Bad Character rule

เมื่ออักขระที่เทียบไม่ตรงกัน ให้ b แทนอักขระที่ไม่ตรงกันใน T

เราจะเลื่อนการเทียบจนกระทั่ง

(a) b ตรงกับอักขระใน P

(b) เลื่อนทั้ง Pattern P ผ่าน b



Good Suffix Rule

ให้ t เป็น substring ของ T ที่ match กับ suffix ของ P จะเลื่อนการเปรียบเทียบจนกระทั่ง (เลือกกรณีที่เกิดขึ้นก่อนจาก 3 กรณีนี้)

(a) t ตรงกับกลุ่มของอักขระใน P

(b) prefix ของ P ตรงกับ suffix ของ t

(c) ย้าย P ผ่าน t

Let *t* be the substring of *T* that matched a suffix of *P*.

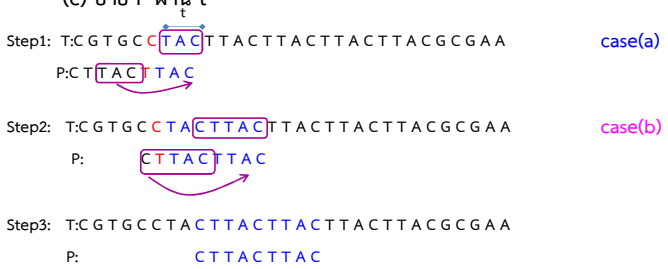
Skip alignments until (a) *t* matches opposite characters in *P*, or (b) a prefix of *P* matches a suffix of *t*, or (c) *P* moves past *t*, whichever happens first

เลือกกรณีที่เกิดขึ้นก่อนจาก 3 กรณีนี้

(a) t ตรงกับกลุ่มของอักขระใน P

(b) prefix ของ P ตรงกับ suffix ของ t

(c) ย้าย P ผ่าน t



Boyer-Moore

หลังจากเทียบแต่ละครั้งจะเลือกใช้ bad character หรือ good suffix ที่ให้ผลลัพธ์การเลื่อนข้ามมากที่สุด