String Matching

String Matching

- String Matching (หรือ string searching) เป็นปัญหาในการหา index เริ่มต้นของ substring หรือ pattern P ใน string ที่เรียกว่า text T

- ตัวอย่างเช่น สมมติว่าเรามี T='STEVEN EVENT'
- และถ้าเรามี P='EVE' คำตอบคือ index 2 และ 7 (เมื่อเริ่มที่ index 0)
- แล้วถ้า P='EVENT' แล้วคำตอบจะเป็น 7 อย่างเดียว
- แต่ถ้า P='EVENING' แล้วไม่มีคำตอบ (หากไม่ match โดยทั่วไปแล้วเราจะคืนค่า –1 หรือ NULL)

Library Solution

- หากทำปัญหา string matching ตรงๆ ส่วนใหญ่กับข้อความสั้นๆ เราสามารถใช้ string library ในการทำงานได้

- นั่นคือ strstr in C <string.h>, find in C++<string>, indexOf in Java String class.

Knuth–Morris–Pratt's (KMP) Algorithm

จากที่เราได้ทดลองเขียนโปรแกรมไปแล้วในการหาทุก substring P(ยาว m) ที่เกิดขึ้นใน string(ยาว n) T ต่อไปเป็นการ implement naive *ของ* String Matching algorithm.

```
void naiveMatching() {
  for (int i = 0; i < n; i++) { //try all potential starting indices
    bool found = true;
    for (int j = 0; j < m && found; j++) // use boolean flag 'found'
      if (i + j >= n || P[j] != T[i + j]) // if mismatch found
        found = false;//abort this, shift the starting index i by +1
    if (found) // if P[0..m-1] == T[i..i+m-1]
      printf("P is found at index %d in T\n", i);
} }
```

- naive algorithm นั้นทำงานโดยเฉลี่ยใน $O(n)$ ถ้าใช้กับข้อความธรรมดา เช่น paragraph ต่างๆในหนังสือ แต่มันจะทำงานใน $O(nm)$ ในกรณี worst case ในการแข่งขันเช่น
- T = ‘AAAAAAAAAAB’ (‘A’ สิบตัวหลังจากนั้นมี ‘B’ หนึ่งตัว) และ
- P = ‘AAAAB’.
- โดย naive algorithm จะผิดที่ตัวอักขระตัวสุดท้ายใน pattern P และหลังจากนั้นจึงจะลอง index ถัดไป
- การทำเช่นนี้จึงไม่ค่อยมีประสิทธิภาพ

- ในปี 1977, Knuth, Morris, and Pratt (อัลกอริทึมนี้เลยชื่อ KMP) ได้คิดอัลกอริทึมที่ดีขึ้นสำหรับ String Matching ที่ใช้ข้อมูลที่ได้รับจากการเปรียบเทียบอักขระก่อนหน้า
- KMP algorithm จะไม่เปรียบเทียบอักขระใน T อีกเมื่อมัน match กับอักขระใน P
- อย่างไรก็ตามมันทำงานคล้ายกับ naive algorithm ถ้าอักขระตัวแรกของ pattern P และอักขระปัจจุบันนั้น T mismatch
- ในตัวอย่างต่อไป การเปรียบเทียบ P[j] และ T[i] จาก i = 0 ถึง 13 เมื่อ

j = 0 (อักขระตัวแรกของ P) ไม่แตกต่างจาก naive algorithm.

```
                1         2         3         4         5
    012345678901234567890123456789012345678901234567890
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P = SEVENTY SEVEN
    0123456789012
```

- อักขระตัวแรกของ P mismatch กับ T[i] ตั้งแต่ index i = 0 ถึง 13 KMP จะเลื่อน index เริ่มต้น I ไปทีละ +1 ซึ่งเหมือนกับ naive matching
- ... เมื่อ i = 14 และ j = 0 ...

```
                1         2         3         4         5
    012345678901234567890123456789012345678901234567890
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P =               SEVENTY SEVEN
                  0123456789012
                            1
                             ^ then mismatch at index i = 25 and j = 11
```

```
                1         2         3         4         5
    012345678901234567890123456789012345678901234567890
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P =               SEVENTY SEVEN
                  0123456789012
                            1
                             ^ then mismatch at index i = 25 and j = 11
```

- ตรงกัน 11 ตัวจาก index i = 14 ถึง 24, แต่ไม่ตรงกันที่ i = 25 (j = 11).
- naive algorithm จะเริ่มแบบไม่มีประสิทธิภาพจาก index i = 15
- แต่ KMP สามารถทำงานต่อที่ i = 25 นั่นเป็นเพราะอักขระที่ตรงกันก่อนหน้าที่จะ mismatch นั้นคือ ‘SEVENTY SEV’ ซึ่งมี ‘SEV’ (of length 3) ปรากฏทั้ง suffix และ prefix ของ ‘SEVENTY SEV’.

- substring ‘SEV’ ถูกเรียกว่าขอบของ ‘SEVENTY SEV’. เราสามารถกระโดดข้ามจาก i = 14 มายัง 21 เนื่องจาก ‘SEVENTY ’ ใน ‘SEVENTY SEV’ จะไม่ match อีก แต่มันอาจจะเป็นไปได้ที่การเหมือนกันครั้งต่อไปเริ่มต้นที่ ‘SEV’ ตัวที่สอง ดังนั้น KMP จะ reset j ลง 3 ทำให้เราข้ามไปได้ 11 – 3 = 8 อักขระของ ‘SEVENTY ’ (สังเกตว่ารวมเคาะส่วนท้ายด้วย), ในขณะที่ i ยังคงอยู่ที่ index 25.
- นี่เป็นจุดต่างระหว่าง KMP และ naive matching algorithm.

```
... at i = 25 and j = 3 (This makes KMP efficient) ...
              1         2         3         4         5
    01234567890123456789012345678901234567890123456789
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P =                       SEVENTY SEVEN
                          0123456789012
                                    1
                             ^ then immediate mismatch at index i = 25, j = 3
```

- ในคราวนี้ prefix ของ P ก่อนที่จะ mismatch คือ ‘SEV’ แต่ไม่มี border (ไม่มี suffic ที่ซ้ำ) ดังนั้น KMP จะ resets j ให้กลับเป็น 0 (หรืออีกความหมายคือ reset matching pattern P)

```
... mismatches from i = 25 to i = 29... then matches from i = 30 to i = 42 ...
                1         2         3         4         5
    012345678901234567890123456789012345678901234567890
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P =                               SEVENTY SEVEN
                                  0123456789012
                                            1
```

- ซึ่ง match ดังนั้น P = ‘SEVENTY SEVEN’ ถูกพบที่ index i = 30. หลังจากนี้ KMP รู้ว่า ‘SEVENTY SEVEN’ มี ‘SEVEN’ (ความยาว 5) เป็น border ดังนั้น KMP ปรับ j ให้กลับไปเป็น 5, โดยกระโดดข้ามไป 13 – 5 = 8 อักขระของ ‘SEVENTY ’ ทำให้กลับไปค้นหาต่อที่ i = 43, และหลังจากนั้นพบอีกครั้ง

```
... at i = 43 and j = 5, we have matches from i = 43 to i = 50 ...
So P = 'SEVENTY SEVEN' is found again at index i = 38.
              1         2         3         4         5
    012345678901234567890123456789012345678901234567890
T = I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN
P =                                       SEVENTY SEVEN
                                          0123456789012
                                                    1
```

- ในการเพิ่มความเร็วนั้น KMP จะต้องทำการประมวลผลล่วงหน้า (preprocess) รูปแบบของ string และดูแล 'reset table' b (back). ถ้ากำหนด pattern string P = 'SEVENTY SEVEN' ตาราง b จะหน้าตาประมาณนี้

```
     0 1 2 3 4 5 6 7 8 9 0 1 2 3
P =  S E V E N T Y   S E V E N
b = -1 0 0 0 0 0 0 0 0 1 2 3 4 5
```

- นั่นหมายความว่า ถ้า mismatch เกิดขึ้นที่ j = 11 หรือหลังจากเทียบ ‘SEVENTY SEV’, แล้วเรารู้ว่าเราจะต้องเทียบ P ใหม่จาก index j = b[11] = 3, นั่นคือ KMP สมมติว่ามันได้เทียบ 3 อักขระแรกของ ‘SEVENTY SEV’ ไปแล้วซึ่งคือ ‘SEV’ ทำให้เราเริ่มเทียบต่อจาก ‘SEV’ ได้เลย

- ต่อไปจะเป็น code ในการ implement KMP

- เวลาในการทำงานเป็น $O(n + m)$.

```
#define MAX_N 100010
char T[MAX_N], P[MAX_N]; // T = text, P = pattern
int b[MAX_N], n, m; // b = back table, n = length of T, m = length of P
void kmpPreprocess() { // call this before calling kmpSearch()
  int i = 0, j = -1; b[0] = -1; // starting values
  while (i < m) { // pre-process the pattern string P
    while (j >= 0 && P[i] != P[j]) j = b[j]; // different, reset j using b
    i++; j++; // if same, advance both pointers
    b[i] = j; // observe i = 8, 9, 10, 11, 12, 13 with j = 0, 1, 2, 3, 4, 5
} } // in the example of P = "SEVENTY SEVEN" above

void kmpSearch() { // this is similar as kmpPreprocess(), but on string T
  int i = 0, j = 0; // starting values
  while (i < n) { // search through string T
    while (j >= 0 && T[i] != P[j]) j = b[j]; // different, reset j using b
    i++; j++; // if same, advance both pointers
    if (j == m) { // a match found when j == m
      printf("P is found at index %d in T\n", i - j);
      j = b[j]; // prepare j for the next possible match
} } }
```

```
int main() {
  strcpy(T, "I DO NOT LIKE SEVENTY SEV BUT SEVENTY
SEVENTY SEVEN");
  strcpy(P, "SEVENTY SEVEN");
  n = (int)strlen(T);
  m = (int)strlen(P);
  kmpPreprocess();
  kmpSearch();
  return 0;
}
```

หากรัน kmpPreprocess() บน P = ‘ABABA’ ตารางเป็นอย่างไร

หากรัน kmpSearch() บน P = ‘ABABA’ และ T = ‘ACABAABABDABABA’. บายการค้นหาที่ได้

String Matching in a 2D Grid

- ปัญหา string matching สามารถอยู่ในรูปของ 2D ได้
- กำหนด ตาราง 2D ของอักขระมาให้ จงหาการเกิดขึ้นของ pattern P ในตาราง
- ขึ้นกับปัญหาในข้อนั้นแล้วว่าต้องการให้เราหาใน 4 ทิศทางหรือ 8 ทิศทาง หรือ pattern ต้องเรียงต่อกันเป็นเส้นตรงหรือหักงอได้
- อย่างเช่น ตัวอย่างนี้ให้หา WALDORF

```
abcdefghigg
hebkWaldork
ftyawAldorm
ftsimrLqsrc
byoarbeDeyv
klcbqwikOmk
strebgadhRb
yuiqlxcnbjF
```

- วิธีการแก้ปัญหาของ string matching ใน 2D grid โดยทั่วไปจะใช้ *recursive backtracking* เพราะว่ามันไม่เหมือนกับใน 1 มิติที่เราจะเดินต่อไปทางขวาเสมอ
- ที่ตำแหน่ง (row, col) ใน 2D grid เรามีทางเลือกมากกว่าหนึ่งทางในการสำรวจ ทั้งนี้ในการเพิ่มความเร็วในการทำ backtracking โดยทั่วไปเราจะใช้วิธีการ pruning strategy: นั่นคือเมื่อความลึกของ recursive เกินความยาวของ pattern เราจะ prun การเรียก recursive นั้นทิ้ง วิธีนี้เรียกว่า *depth–limited search*