Arrays and Structures

Pattern Matching

Joseph Chuang-Chieh Lin (林莊傑)

Department of Computer Science & Engineering,
National Taiwan Ocean University

Fall 2025

Outline

String ADT

- A string is:
  - objects: a finite set of zero or more characters.
- functions: for all strings s, t, and $i, j, m \in \mathbb{Z}^+$:
  - String Null(m) ::= return a string whose maximum length is m characters, but is initially set to NULL We write NULL as "".
  - Integer Compare(s, t) ::= if s equals t return 0 else if s precedes t return -1 else return $+1$
  - Boolean IsNull(s) ::= if (Compare(s, NULL)) return FALSE else return TRUE
  - Integer Length(s) ::= if (Compare(s, NULL)) return the number of characters in s else return 0.

String ADT (functions contd.)

- String Concat(s, t) ::= if (Compare(t, NULL)) return a string whose elements are those of s followed by those of t else return s.
- String Substr(s, i, j) ::= if $((j > 0)$ && $(i + j - 1) <$ Length(s)) return the string containing the characters of s at positions i, $i + 1, \ldots,$ $i + j - 1$. else return NULL.
- void StrInsert(s, t, i)::= insert string t at position i of s.

String Insertion (code)

```
void strInsert(char *s, char *t, int i) {
    /* insert string t into string s at position i */
    char string[MAX_SIZE], *temp = string;

    if (i < 0 && i > strlen(s)) {
        fprintf(stderr, "Position is out of bounds \n");
        exit(1);
    }
    if (!strlen(s))
        strcpy(s, t);
    else if (strlen(t)) {
        strncpy(temp, s, i); // Copy the first i characters of s to temp
        temp[i] = '\0'; // complete a string with '\0'
        strcat(temp, t);
        strcat(temp, (s + i)); // concatenate the rest characters of s to temp
        strcpy(s, temp);
    }
}
```

String Matching

String Matching

- **Input:** Two strings s and p of sizes n and m respectively.
- **Output:** Determine if p is a substring of s. Return the starting position of s if it is and -1 otherwise.

String Matching

String Matching

- **Input:** Two strings s and p of sizes n and m respectively.
- **Output:** Determine if p is a substring of s. Return the starting position of s if it is and -1 otherwise.

- Substring: a contiguous sequence of characters within a string.
- For example,

String Matching

String Matching

- **Input:** Two strings s and p of sizes n and m respectively.
- **Output:** Determine if p is a substring of s. Return the starting position of s if it is and -1 otherwise.

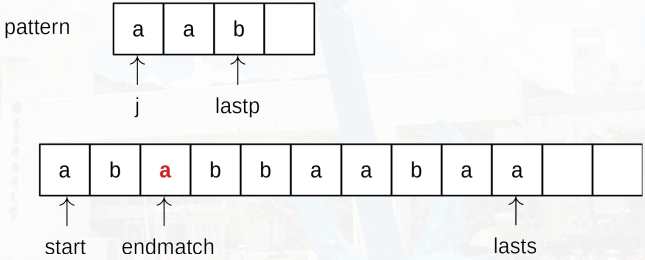
- Substring: a contiguous sequence of characters within a string.
- For example,

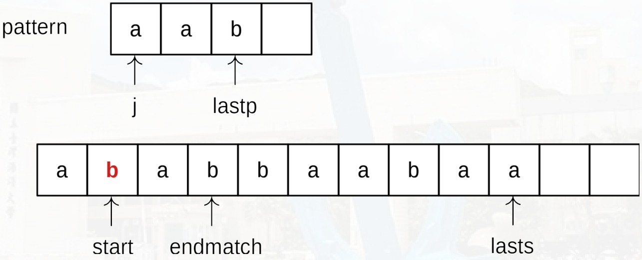
  `cgta` is a substring of `acgtacct`.

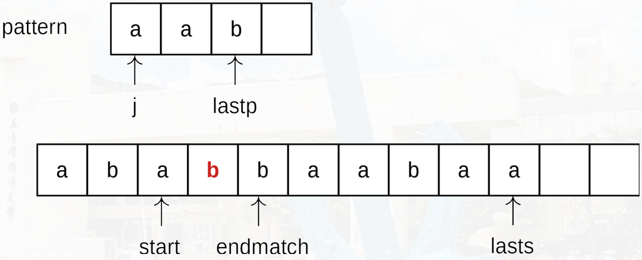
  `acgg` is NOT a substring of `acgtacct`.

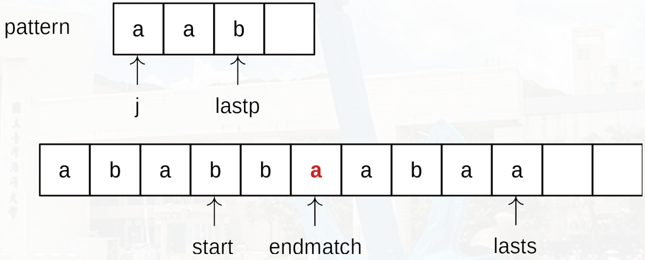
Outline

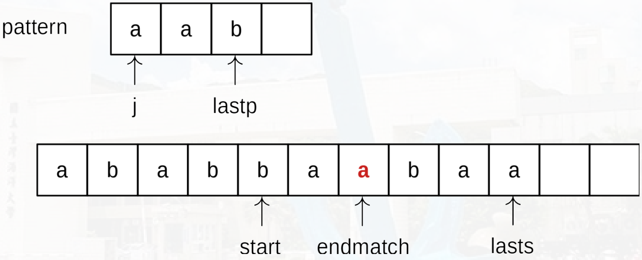
```c
int naiveFind(char *string, char *pat) {
    /* match the last character of pattern first,
       and then match from the beginning */
    int i, j, start = 0;
    int lasts  = strlen(string) - 1;
    int lastp  = strlen(pat)    - 1;
    int endmatch = lastp;

    for (i = 0; endmatch <= lasts; endmatch++, start++) {
        if (string[endmatch] == pat[lastp])
            for (j = 0, i = start; j < lastp && string[i] == pat[j]; i++, j++)
                ;  // empty statement: advance while matching
        if (j == lastp)
            return start; /* successful */
    }
    return -1;
}
```

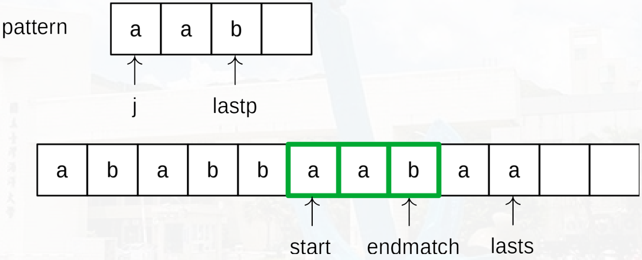

pattern
a
a
b
j
lastp
a
b
a
b
b
a
a
b
a
a
start
endmatch
lasts


pattern
a
a
b
j
lastp
a
b
a
b
b
a
a
b
a
a
start
endmatch
lasts


pattern
a
a
b
j
lastp
a
b
a
b
b
a
a
b
a
a
start
endmatch
lasts


pattern
a
a
b
j
lastp
a
b
a
b
b
a
a
b
a
a
start
endmatch
lasts


pattern
a
a
b
j
lastp
a
b
a
b
b
a
a
b
a
a
start
endmatch
lasts


pattern
a
a
b
j
lastp
a
b
a
b
b
a
a
b
a
a
start
endmatch
lasts

Time Complexity

- Worst case: $O(n \cdot m)$.

Time Complexity

- Worst case: $O(n \cdot m)$.
- Actually, we can do much better for the pattern matching.

Outline

Knuth, Morris, & Pratt (1977)


Home → SIAM Journal on Computing → Vol. 6, Iss. 2 (1977) → 10.1137/0206024

Previous Article Next Article

Fast Pattern Matching in Strings

Authors: Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt | AUTHORS INFO & AFFILIATIONS

https://doi.org/10.1137/0206024

GET ACCESS BibTeX Tools

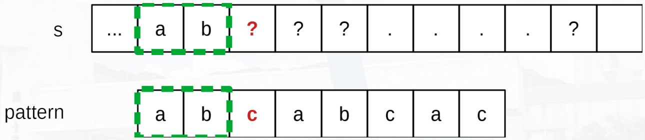
Abstract

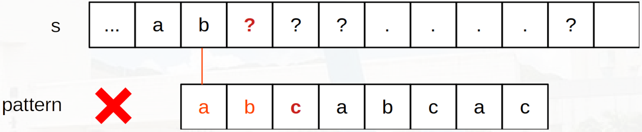
An algorithm is presented which finds all occurrences of one given string within another, in running time proportional to the sum of the lengths of the strings. The constant of proportionality is low enough to make this algorithm of practical use, and the procedure can also be extended to deal with some more general pattern-matching problems. A theoretical application of the algorithm shows that the set of concatenations of even palindromes, i.e., the language $\{\alpha\alpha^R\}^*$, can be recognized in linear time. Other algorithms which run even faster on the average are also considered.

Keywords

pattern string text-editing pattern-matching trie memory searching

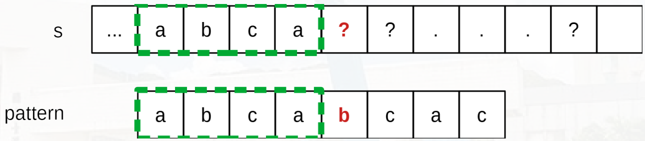
- *SIAM Journal on Computing*, Vol. 6(2) (1977), pp. 323–350.

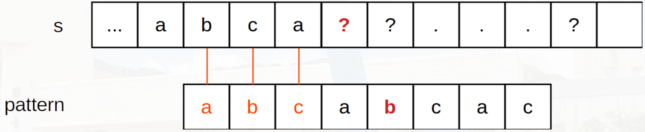

s
...
a
b
?
?
?
.
.
.
.
?
pattern
a
b
c
a
b
c
a
c

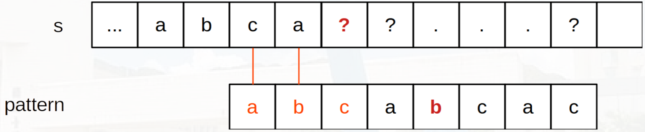

s
...
a
b
?
?
?
.
.
.
.
?
pattern
a
b
c
a
b
c
a
c

s	...	a	b	?	?	?	.	.	.	.	?	

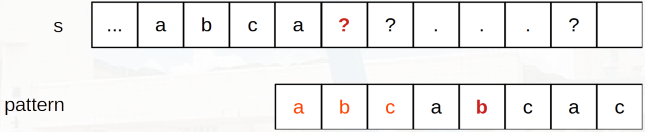
pattern	O		a	b	c	a	b	c	a	c


s
... a b c a ? ? . . . ?
pattern
a b c a b c a c


s
... a b c a ? ? . . . ?
pattern
a b c a b c a c


s
...
a
b
c
a
?
?
.
.
.
?
pattern
a
b
c
a
b
c
a
c

s	...	a	b	c	a	**?**	?	.	.	.	?	
pattern					a	b	c	a	**b**	c	a	c



- We can "learn some information" from the pattern and the mismatches.

Key Observation

Observation

Suppose that

- $\mathrm{Substr}(p, 0, k) = \mathrm{Substr}(s, x, k)$ but
- $p[k] \neq s[x + k]$.

Then, for any $0 < r < k$, if $\mathrm{Substr}(s, x + r, k - r)$ is not a *prefix* of p, we know p cannot occur as a substring of s at position $x + r$.

Key Observation

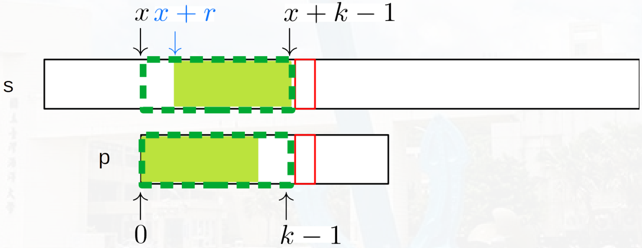
Observation

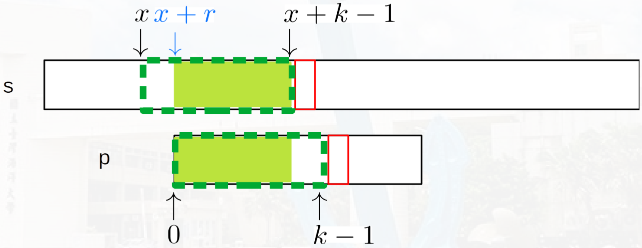
Suppose that

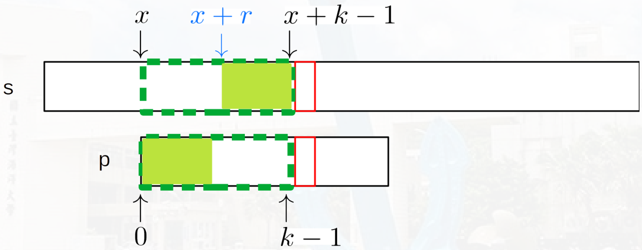
- $\mathrm{Substr}(p, 0, k) = \mathrm{Substr}(s, x, k)$ but
- $p[k] \neq s[x + k]$.

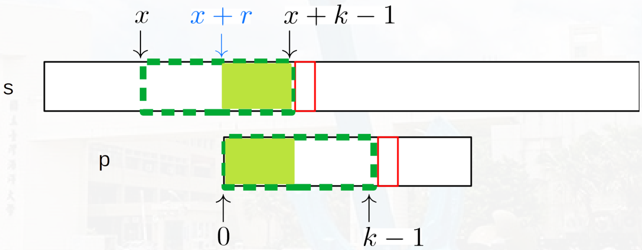
Then, for any $0 < r < k$, if $\mathrm{Substr}(s, x + r, k - r)$ is not a *prefix* of p, we know p cannot occur as a substring of s at position $x + r$.

- We can bypass the impossibilities!


x $x + r$
$x + k - 1$
s
p
0
$k - 1$


x
x + r
x + k − 1
s
p
0
k − 1


x
$x + r$
$x + k - 1$
s
p
0
$k - 1$


x
$x + r$
$x + k - 1$
s
p
0
$k - 1$

The failure function

If $p = p_0 p_1 \cdots p_{m-1}$ is a pattern, then the failure function f is defined as $f(j) =$

- largest $k < j$ such that

$$p_0 p_1 \cdots p_k = p_{j-k} p_{j-k+1} \cdots p_j$$

  if such $k \geq 0$ exists
- -1 otherwise.

The failure function

If $p = p_0 p_1 \cdots p_{m-1}$ is a pattern, then the failure function f is defined as $f(j) =$

- largest $k < j$ such that

$$p_0 p_1 \cdots p_k = p_{j-k} p_{j-k+1} \cdots p_j$$

  if such $k \geq 0$ exists
- -1 otherwise.

- For example, suppose $p =$ abcabcacab,

j	0	1	2	3	4	5	6	7	8	9
p	a	b	c	a	b	c	a	c	a	b
failure	-1	-1	-1	0	1	2	3	-1	0	1

The rule of using the failure function

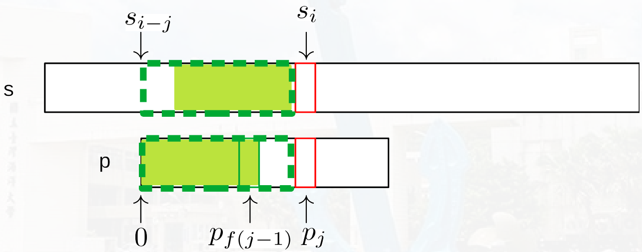
- If a partial match is found and $j \neq 0$ such that

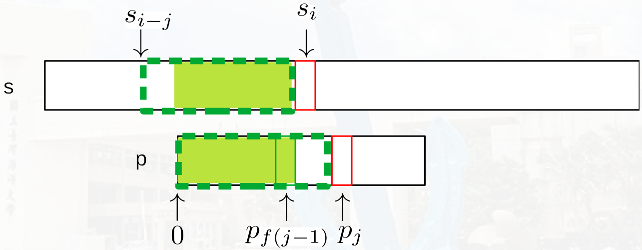
$$s_{i-j} \cdots s_{i-1} = p_0 p_1 \cdots p_{j-1}$$

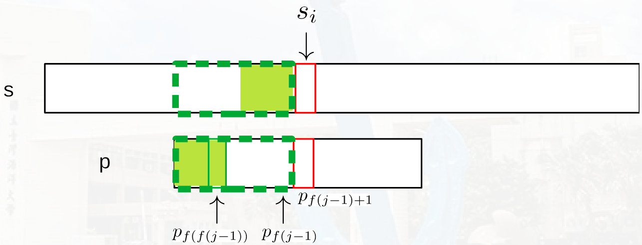
and

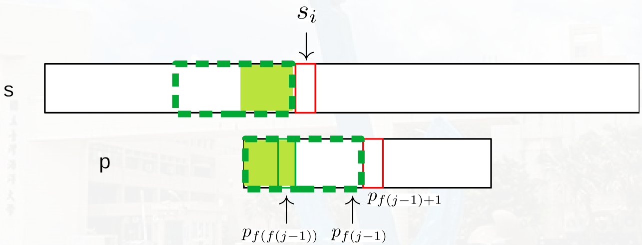
$$s_i \neq p_j,$$

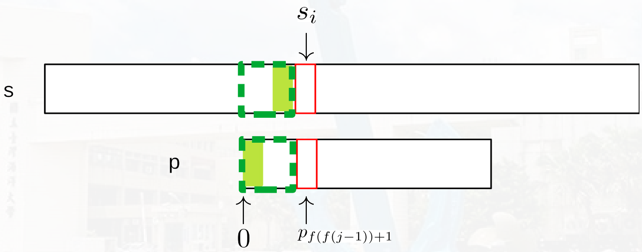
then matching may be resumed by comparing s_i and $p_{f(j-1)+1}$.
- If $j = 0$, then we may continue by comparing s_{i+1} and p_0.


s_{i-j}
s_i
s
p
0
$p_{f(j-1)}$
p_j


s_{i-j}
s_i
s
p
0
$p_{f(j-1)}$
p_j


s_i
s
p
$p_{f(j-1)+1}$
$p_{f(f(j-1))}$
$p_{f(j-1)}$


s_i
s
p
$p_{f(j-1)+1}$
$p_{f(f(j-1))}$
$p_{f(j-1)}$


s_i
s
p
0
$p_{f(f(j-1))+1}$

A fast way to compute the failure function

$$f(j) = \begin{cases} -1, & j = 0, \\ f^m(j-1)+1, & \text{if } m \geq 0 \text{ is the least integer such that } p_{f^m(j-1)+1} = p_j, \\ -1, & \text{otherwise.} \end{cases}$$

- **Note:** $f^1(j) = f(j)$ and $f^m(j) = f(f^{m-1}(j))$.

Failure Function Code ($O(m)$)

```
void fail(const char* p, int* f) { // p: pattern; f: failure
    /* compute the pattern's failure function */
    int m = strlen(p);
    f[0] = -1;
    for (int j = 1; j < m; j++) {
        int i = f[j-1]; // i: prefix length
        while ((i >= 0) && (p[j] != p[i+1])) i = f[i];
        if (p[j] == p[i+1])
            f[j] = i + 1;
        else
            f[j] = -1;
    }
}
```

j	0	1	2	3	4	5	6	7	8	9
p	a	b	c	a	b	c	a	c	a	b
f	-1	-1	-1	0	1	2	3	-1	0	1

The time complexity of fail

- In each iteration of the while loop, the value of i decreases ($\because f$).
- i is reset at the beginning of each iteration of the for loop.
  - It is either reset to -1 or to a value 1 greater than its terminal value on the previous iteration (i.e., $f[j] = i + 1$).
  - Since the for loop is iterated only $m - 1$ times, the value of i is incremented for at most $m - 1$ times.
    - Hence, it cannot be decremented more than $m - 1$ times.
- The while loop is iterated $\leq m - 1$ times overall.

Declarations

```c
#include <stdio.h>
#include <string.h>

#define MAX_STRING_SIZE  100
#define MAX_PATTERN_SIZE 100

int KMPmatch(void);
void fail(void);

int  f[MAX_PATTERN_SIZE]; // computed by the failure function
char s[MAX_STRING_SIZE];
char p[MAX_PATTERN_SIZE];
```

The KMP Algorithm

```
int KMPmatch(char *s, char *p) {
    int i = 0, j = 0; // i: for string s, and j: for pattern p
    int n = strlen(s); // n: length of string s
    int m = strlen(p); // m: length of pattern p

    while (i < n && j < m) {
        if (s[i] == p[j]) {
            i++; j++;
        } else if (j == 0) {
            i++;
        } else {
            j = f[j-1] + 1;
        }
    }
    return (j == m) ? (i - m) : -1;
}
```

- the pattern is not found $\Rightarrow$ the pattern index $j \neq m$ $\Rightarrow$ return -1.
- the pattern is found $\Rightarrow$ the starting position is $i - m$.

Analysis of KMPmatch

- The while loop is iterated until the end of either the string or the pattern is reached.
- i is never decreased, so the lines that increase i cannot be executed $> n$ times.
- The resetting of j to $f[j-1]+1$ decreases the value of j.
  - This cannot be done more times than j is incremented by the statement j++ (otherwise j falls off the pattern...).
- Each time j++ is executed, i is also incremented.
  - So, j cannot be incremented $> n$ times.
- No statement in the program is executed $> n$ times, therefore the time complexity of KMTmatch is $O(n+m)$ ($O(m)$ for fail()).

Discussions