Chap 2. Arrays and Structures (2)

Contents

2.5 Sparse Matrix

2.5.1 The Abstract Data Type

- Standard representation of a matrix
  - A[MAX_ROWS][MAX_COLS]

$$
\begin{array}{c} \\ \text{row } 0 \\ \text{row } 1 \\ \text{row } 2 \\ \text{row } 3 \\ \text{row } 4 \end{array}
\begin{array}{c} \begin{array}{ccc} \text{col } 0 & \text{col } 1 & \text{col } 2 \end{array} \\
\begin{bmatrix} -27 & 3 & 4 \\ 6 & 82 & -2 \\ 109 & -64 & 11 \\ 12 & 8 & 9 \\ 48 & 27 & 47 \end{bmatrix} \end{array}
$$

(a)

$$
\begin{array}{c} \\ \text{row } 0 \\ \text{row } 1 \\ \text{row } 2 \\ \text{row } 3 \\ \text{row } 4 \\ \text{row } 5 \end{array}
\begin{array}{c} \begin{array}{cccccc} \text{col } 0 & \text{col } 1 & \text{col } 2 & \text{col } 3 & \text{col } 4 & \text{col } 5 \end{array} \\
\begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix} \end{array}
$$

(b)

Figure 2.4: Two matrices

- Sparse matrix
  - $m \times n$ matrix A : $\dfrac{\text{no. of nonzero elements}}{m \times n} << 1$

2.5.2 Sparse Matrix Representation

- *An array of triples*
  - <row, column, value> : 3-tuples (triples)

SparseMatrix Create(*maxRow*, *maxCol*) ::=

```
#define MAX_TERMS 101 /* maximum number of terms +1*/
typedef struct {
        int col;
        int row;
        int value;
        } term;
term a[MAX_TERMS];
```

$$
\begin{array}{c c}
 & \begin{array}{cccccc} \text{col 0} & \text{col 1} & \text{col 2} & \text{col 3} & \text{col 4} & \text{col 5} \end{array} \\
\begin{array}{c} \text{row 0} \\ \text{row 1} \\ \text{row 2} \\ \text{row 3} \\ \text{row 4} \\ \text{row 5} \end{array} &
\begin{bmatrix}
15 & 0 & 0 & 22 & 0 & -15 \\
0 & 11 & 3 & 0 & 0 & 0 \\
0 & 0 & 0 & -6 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
91 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 28 & 0 & 0 & 0
\end{bmatrix}
\end{array}
$$

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	−15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	−6
[7]	4	0	91
[8]	5	2	28

Use *an array of triples* (cont')

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	−15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	−6
[7]	4	0	91
[8]	5	2	28

a[0].row : the number of rows
a[0].col : the number of columns
a[0].value : the total number of nonzero entries

– The triples are ordered by row and within rows by columns. (*row major ordering*)

ADT *SparseMatrix* is

objects: a set of triples, <*row*, *column*, *value*>, where *row* and *column* are integers and form a unique combination, and *value* comes from the set *item*.

functions:

for all $a, b \in SparseMatrix, x \in item, i, j, maxCol, maxRow \in index$

SparseMatrix Create(*maxRow*, *maxCol*) ::=
: **return** a *SparseMatrix* that can hold up to $maxItems = maxRow \times maxCol$ and whose maximum row size is *maxRow* and whose maximum column size is *maxCol*.

SparseMatrix Transpose(*a*) ::=
: **return** the matrix produced by interchanging the row and column value of every triple.

SparseMatrix Add(*a*, *b*) ::=
: **if** the dimensions of *a* and *b* are the same
: **return** the matrix produced by adding corresponding items, namely those with identical *row* and *column* values.
: **else return** error

SparseMatrix Multiply(*a*, *b*) ::=
: **if** number of columns in *a* equals number of rows in *b*
: **return** the matrix *d* produced by multiplying *a* by *b* according to the formula: $d[i][j] = \sum(a[i][k] \cdot b[k][j])$ where $d(i, j)$ is the (i, j)th element
: **else return** error.

ADT 2.3: Abstract data type *SparseMatrix*

2.5.3 Transposing a Matrix

for $j \leftarrow 1$ to n do
  for $i \leftarrow 1$ to m do
    b(j, i) ← a(i, j)
  end
end

a[0]	6	6	8
a[1]	0	0	15
a[2]	3	0	15
a[3]	5	0	22

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	–15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	–6
[7]	4	0	91
[8]	5	2	28

(a)

	row	col	value
b[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	–6
[8]	5	0	–15

(b)

Figure 2.5: Sparse matrix and its transpose stored as triples

- Is this a good algorithm for transposing a matrix?

`for each row i` of original matrix
```
  take element <i, j, value> and store it
  as element <j, i, value> of the transpose;
```

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	−15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	−6
[7]	4	0	91
[8]	5	2	28

a		***b***
(0, 0, 15)	→	(0, 0, 15)
(0, 3, 22)	→	(3,0, 22)
(0, 5, -15)	→	(5, 0, -15)
(1, 1, 11)	→	(1, 1, 11)
		data movement
(1, 2, 3)	→	(2, 1, 3)
		data movement
	...	

We must move elements to maintain the correct order!

- Using column indices

`for all elements in column j` of original matrix
`place element <i, j, value> in`
`element <j, i, value>` of the transpose

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	−15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	−6
[7]	4	0	91
[8]	5	2	28

a		***b***
(0, 0, 15)	→	(0, 0, 15)
(4, 0, 91)	→	(0, 4, 91)
(1, 1, 11)	→	(1, 1, 11)
(2, 1, 3)	→	(2, 1, 3)
(5, 2, 28)	→	(2, 5, 28)
	…	

We can avoid data movement!

```c
typedef struct {
        int col;
        int row;
        int value;
        } term;
term a[MAX_TERMS];
```

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	−15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	−6
[7]	4	0	91
[8]	5	2	28

(a)

	row	col	value
b[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	−6
[8]	5	0	−15

(b)

Figure 2.5: Sparse matrix and its transpose stored as triples

```c
void transpose(term a[], term b[])
{/* b is set to the transpose of a */
  int n,i,j, currentb;
  n = a[0].value;          /* total number of elements */
  b[0].row = a[0].col; /* rows in b = columns in a */
  b[0].col = a[0].row; /* columns in b = rows in a */
  b[0].value = n;
  if (n > 0 )  { /* non zero matrix */
    currentb = 1;
    for (i = 0; i < a[0].col; i++) //a의 열 인덱스
    /* transpose by the columns in a */
      for (j = 1; j <= n; j++)  //원소의 개수 만큼, a의 행 인덱스
      /* find elements from the current column */
        if (a[j].col == i) {
        /* element is in current column, add it to b */
          b[currentb].row = a[j].col;
          b[currentb].col = a[j].row;
          b[currentb].value = a[j].value;
          currentb++;
        }
  }
}
```

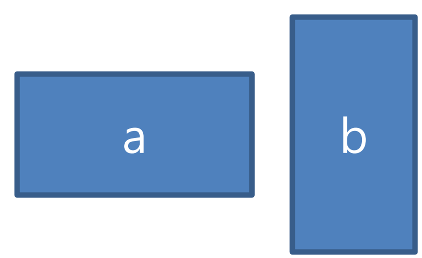
Program 2.8: Transpose of a sparse matrix

Time complexity : O(*columns* · *elements*)

- Analysis of *transpose*
  - Nested for loops are the decisive factor.
  - The remaining part requires only constant time.
  - Time complexity : **$O(\textit{columns} \cdot \textit{elements})$**
  - If $\textit{elements} = \textit{rows} \cdot \textit{columns}$, $O(\textit{columns}^2 \cdot \textit{rows})$
    - To conserve space, we have traded away too much time.

cf) If the matrices are represented as 2D arrays,

```
for ( j = 0; j < columns; j++)
    for ( i = 0; i < rows; i++)
        b[j][i] = a[i][j];
```



- $O(\textit{columns} \cdot \textit{rows})$

- Fast transpose of a sparse matrix

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	−15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	−6
[7]	4	0	91
[8]	5	2	28

① calculation of rowTerms

	row	col	value
b[0]	6	6	8
→ [1]			
[2]			
→ [3]			
→ [4]			
[5]			
→ [6]			
[7]			
→ [8]			

	[0]	[1]	[2]	[3]	[4]	[5]
rowTerms =						
startingPos =	1	3	4	6	8	8

a 에서 같은 col 수

② calculation of startingPos

- Fast transpose of a sparse matrix(cont')

③ b(j,i)← a(i,j)

	row	col	value
a[0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	–15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	–6
[7]	4	0	91
[8]	5	2	28

	row	col	value
b[0]	6	6	8
[1]	0	0	15
[2]	0	4	91
[3]	1	1	11
[4]	2	1	3
[5]	2	5	28
[6]	3	0	22
[7]	3	2	-6
[8]	5	0	-15

현재 처리시 0 행의 시작 주소는 2, 처리 후 1증가

	[0]	[1]	[2]	[3]	[4]	[5]
rowTerms =	2	1	2	2	0	1
startingPos =	3	4	6	8	8	9

완료 후 StartingPos 값

Fast transpose of a sparse matrix(cont')

	행	열	값
a[0]	6	6	8
a[1]	0	0	15
a[2]	0	3	22
a[3]	0	5	-15
a[4]	1	1	11
a[5]	1	2	3
a[6]	2	3	-6
a[7]	4	0	91
a[8]	5	2	28

rowTerms[MAX_COL]

[0]	2
[1]	1
[2]	2
[3]	2
[4]	0
[5]	1

[0]	1
[1]	3
[2]	4
[3]	6
[4]	8
[5]	8

startingPos[MAX_COL]

	행	열	값
b[0]	6	6	8
b[1]	0	0	15
b[2]	0	4	91
b[3]	1	1	11
b[4]	2	1	3
b[5]	2	5	28
b[6]	3	0	22
b[7]	3	2	-6
b[8]	5	0	-15

```
void fastTranspose(term a[], term b[])
{/* the transpose of a is placed in b */
   int rowTerms[MAX_COL], startingPos[MAX_COL];
   int i,j, numCols = a[0].col, numTerms = a[0].value;
   b[0].row = numCols;  b[0].col = a[0].row;
   b[0].value = numTerms;
   if (numTerms > 0) { /* nonzero matrix */
     for (i = 0; i < numCols; i++)
       rowTerms[i] = 0;
     for (i = 1; i <= numTerms; i++)
       rowTerms[a[i].col]++;
     startingPos[0] = 1;
     for (i = 1; i < numCols; i++)
       startingPos[i] =
                 startingPos[i-1] + rowTerms[i-1];
     for (i = 1; i <= numTerms; i++) {
       j = startingPos[a[i].col]++;
       b[j].row = a[i].col;   b[j].col = a[i].row;
       b[j].value = a[i].value;
     }
   }
}
```

calculation of rowTerms

calculation of startingPos

b(j,i) ← a(i,j)

	[0]	[1]	[2]	[3]	[4]	[5]
rowTerms =	2	1	2	2	0	1
startingPos =	1	3	4	6	8	8

Program 2.9: Fast transpose of a sparse matrix O(*columns* + *elements*)

- Analysis of *fastTranspose*
  - The number of iterations of the four loops
    - *numCols*, *numTerms*, *numCols*-1, *numTerms*, respectively
  - The statements within the loops require constant time.
  - Time complexity : **O(*columns*+ *elements*)**

    (columns · (rows+1))
  - If *elements* = *columns* · *rows*, O(*columns* · *rows*)
    - equals that of the 2D array representation
  - However, if *elements* << *columns* · *rows*,
    - much faster than 2D array representation
  - Thus, in this representation *we save both time and space*.

2.6 Representation of Multidimensional Arrays

- *Array of arrays* in C (Section 2.2.2)
  - store it in consecutive memory like1D array
  - $a[upper_0][upper_1]\ldots[upper_{n-1}]$

The number of elements = $\prod_{i=0}^{n-1} upper_i$

Row Major Order

- Declaration: A[2][3][2][2]
  - the range of index values
    - *0..1, 0..2, 0..1, 0..1*
  - order to store

A[0][0][0][0], A[0][0][0][1], A[0][0][1][0], A[0][0][1][1]
A[0][1][0][0], A[0][1][0][1], A[0][1][1][0], A[0][1][1][1]
…
A[1][2][0][0], A[1][2][0][1], A[1][2][1][0], A[1][2][1][1]

A synonym for row major order is ***lexicographic order***!!

Row Major Order(cont')

- $a[upper_0][upper_1]$

	address
$a[0][0]$	α
$a[i][0]$	$\alpha + i \cdot upper_1$
$a[i][j]$	$\alpha + i \cdot upper_1 + j$

Assuming that each array element requires only one word of memory

- $a[upper_0][upper_1][upper_2]$

$a[0][0][0]$	α
$a[i][0][0]$	$\alpha + i \cdot upper_1 \cdot upper_2$
$a[i][j][k]$	$\alpha + i \cdot upper_1 \cdot upper_2 + j \cdot upper_2 + k$

Row Major Order(cont')

- $a[upper_0][upper_1]\ldots[upper_{n-1}]$

$a[0]\,[0]\ \ldots\ [0]$	α
$a[i_0][0][0]\ldots[0]$	$\alpha + i_0\, upper_1\, upper_2 \ldots upper_{n-1}$
$a[i_0][i_1][0]\ldots[0]$	$\alpha + i_0\, upper_1\, upper_2 \ldots upper_{n-1} + i_1\, upper_2\, upper_3 \ldots upper_{n-1}$
$a[i_0][i_1]\ \ldots\ [i_{n-1}]$	$\alpha + i_0 upper_1 upper_2 \ldots upper_{n-1} + i_1 upper_2 upper_3 \ldots upper_{n-1} + i_2 upper_3 upper_4 \ldots upper_{n-1} + \cdots + i_{n-2} upper_{n-1} + i_{n-1}$

$$= \alpha + \sum_{j=0}^{n-1} i_j a_j \text{ where: } \begin{cases} a_j = \prod_{k=j+1}^{n-1} upper_k & 0 \le j < n-1 \\ a_{n-1} = 1 \end{cases}$$

Pattern matching

- More complex string applications

  Assume pat is a pattern to search for a string

  Easiest way to determine if pat is in string or not

    Using the built-in function strstr

      Statement identifying whether pat is in string

```
if (t=strstr(string, pat))
        printf("The string from strstr is : %s\n", t);
else
        printf("The pattern was not found with strstr\n");
```

  – Although strstr appears to be well-suited for pattern matching, let's develop our own pattern matching function

```c
#include <stdio.h>
#include <string.h>
void main()
{
char *string = "This is a test program for strsrt
function ";
char *pat = "test";
char *t;
if (t = strstr(string, pat))
printf("The string from strstr is : %s₩n", t);
else
printf("The pattern was not found with strstr₩n");
}
```

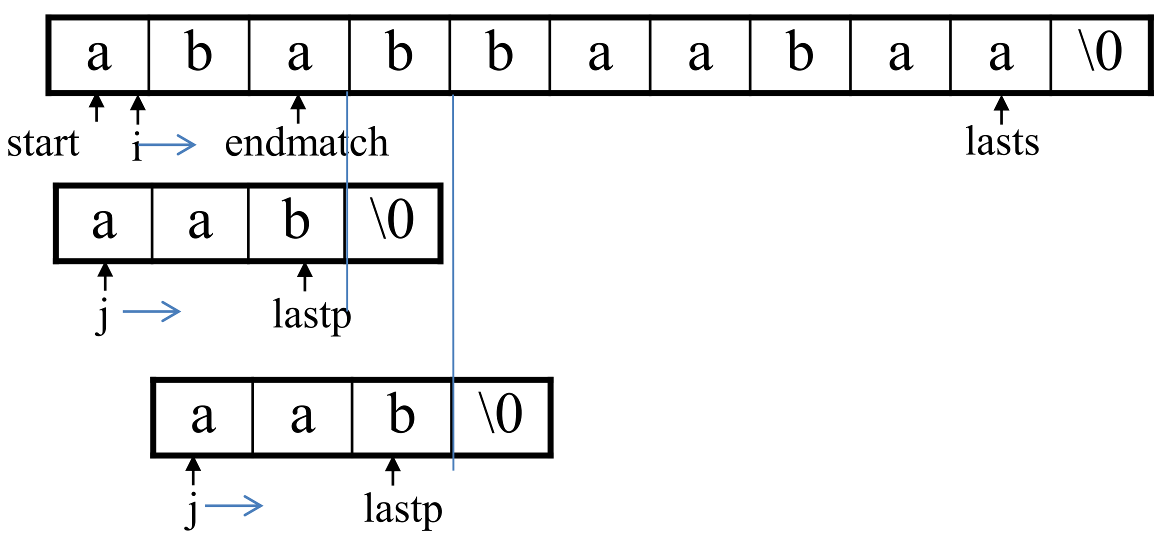
C:₩WINDOWS₩system32₩cmd.exe

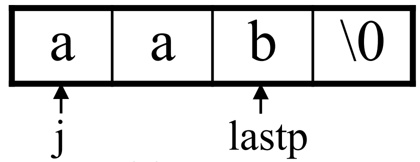
```
The string from strstr is : test program for strsrt function
계속하려면 아무 키나 누르십시오 . . .
```

Pattern matching

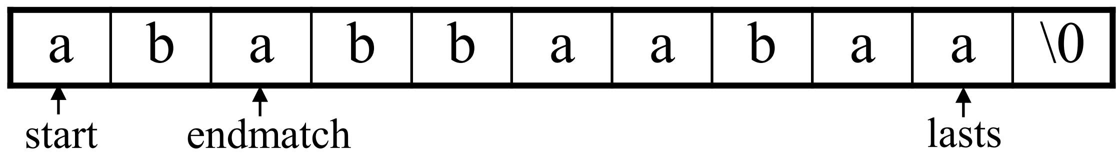
- **nfind simulation**

  Assume pat = "aab" and string = "ababbaabaa"
  Point end of string to lasts, end of pat array to lastp
  Compare string [endmatch] with pat [lastp]
  If they match, use i, j to move two strings until pat is matched.
  The variables start and endmatch are incremented.

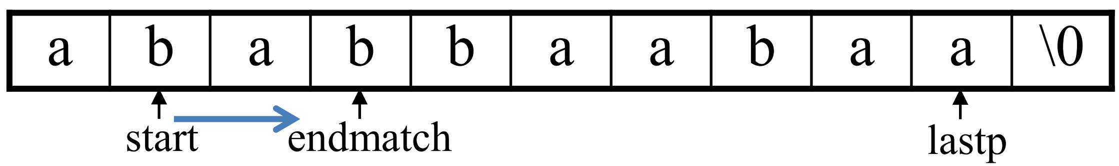




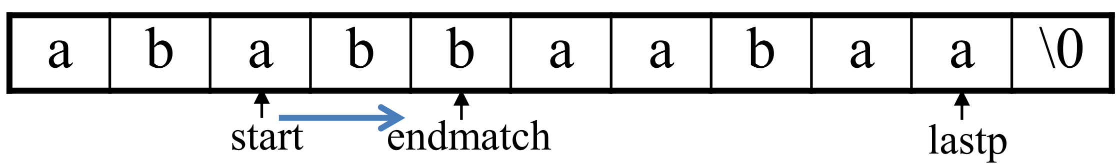
(a) pattern



(b) no match

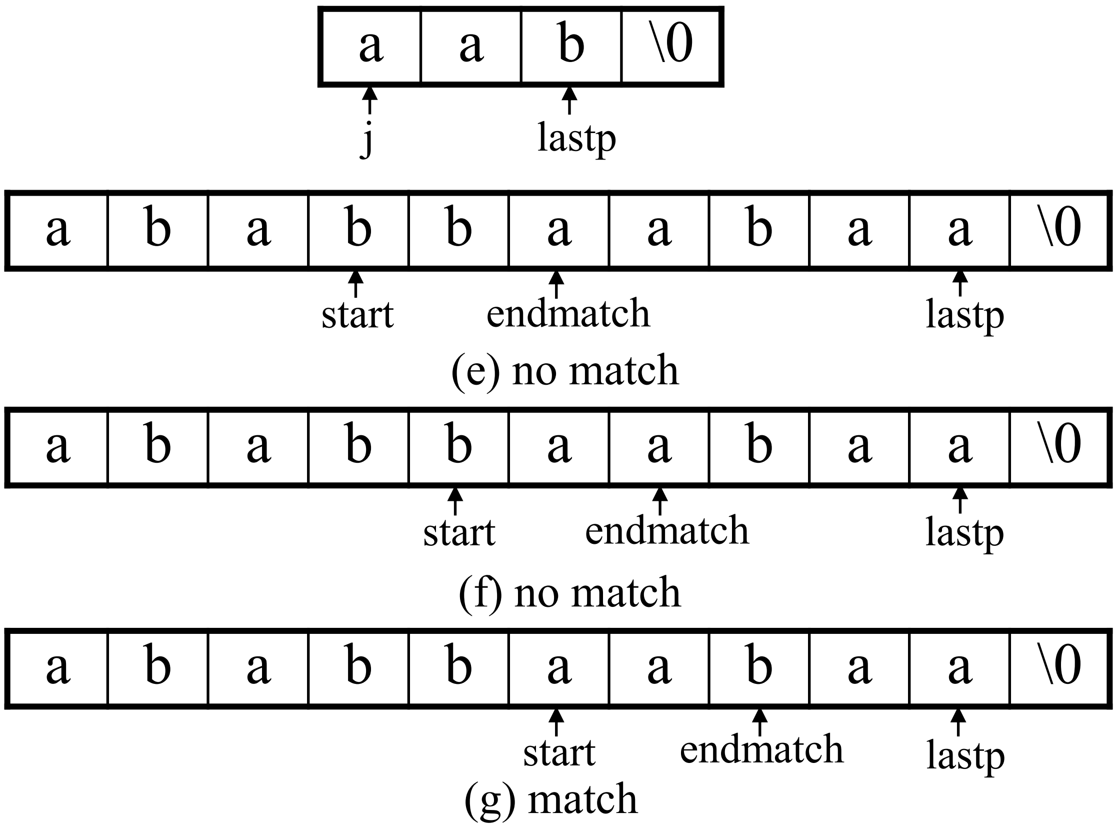


(c) no match



(d) no match

Pattern matching



Pattern matching

- Pattern matching that checks the last character of the pattern first

```
int nfind(char *string, char *pat)
{ /* Match the last character in the pattern first, then match it from the beginning. */
    int i=0, j=0, 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++);
        if (j = = lastp)
            return start;    /* success */
    }
    return -1;
}
```

< nfind function >

Time complexity : O(lasts*lastp)

aaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaba

Pattern matching

- **KMP (Knuth, Morris, Pratt)** Pattern matching
- **failure function**
  - String : a b c a b c a b c a b c a b d a b c c
  - Pattern : a b c a b c a b d a

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

실패함수 : 패턴에 대한 정보를 제공,
현재의 비교가 실패했을때,
패턴의 몇 번째문자와 비교해야 할까에 대한 정보를 제공

Pattern matching

- **KMP (Knuth, Morris, Pratt)** Pattern matching
- **failure function**
  - String : a b c a b c a b c a b c a b d a b c c
  - Pattern : a b c a b c a b d a

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

j : pattern의 index
f : pattern에 있는 문자들이 패턴의 시작 위치에서 부터 일치하는 문자 수

위의 strin에서 8번째 문자 ‘c’와 pattern ‘d’가 다름
pattern의 7번째 문자는 pattern의 처음부터 4 번째까지 일치함으로,
string에서 8번째 문자 ‘c’와 pattern의 5번째 문자 ‘c’와 비교를 수행

패턴매칭

- **실패함수(failure function)**
  - 정의 : 임의의 패턴 $p=p_0p_1 \ldots p_{n-1}$이 있을 때
    이 패턴의 실패함수(f)는 다음과 같이 정의한다.

$$f(j) = \begin{cases} \text{제일 큰 } i < j : \text{여기서 } p_0p_1\ldots p_i = p_{j-i}p_{j-i+1}\ldots p_j \text{인 } i \geq 0 \text{이 존재시} \\ -1 \quad : \text{그 이외의 경우} \end{cases}$$

ex) 패턴 pat = abcabcaab에 대해 f는 다음과 같다.

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

Pattern matching(6/7)

- **pmatch 함수 : 패턴 매칭의 규칙을 함수화한 것**

```
#include <stdio.h>
#include <string.h>
#define max_string_size 100
#define max_pattern_size 100
int pmatch();
void fail();
int failure[max_pattern_size];
char string[max_string_size];
char pat[max_pattern_size];

int pmatch(char *string, char *pat)
{
    /* Knuth, Morris, Pratt의 스트링 매칭 알고리즘 */
    int i = 0, j = 0;
    int lens = strlen(string);
    int lenp = strlen(pat);
```

Pattern matching(7/7)

```
    while (i<lens && j < lenp)  {
          if (string[i] = = pat[j])  {
                i++; j++;
          }
          else if (j = = 0) i++;  //패턴의 처음부터 다름
          else j = failure[j−1]+1;
    }
    return (( j = = lenp) ? (i − lenp) : −1);
}
```

복잡도 : $O(m) = O(\text{strlen(string)})$

- String : a b c a b c a b c a b c a b d a b c c
- Pattern : a b c a b c a b d a

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

```
void fail(char *pat)
{/* compute the pattern's failure function */
    int n = strlen(pat);
    failure[O] = -1;
    for (j=l; j < n; j++) {
        i = failure[j-1];
        while ((pat[j] !=pat[i+l]) && (i>=O))
            i = failure[i];
        if (pat[j] == pat[i+l])
            failure[j] = i+l;
        else failure[j] = -1;
    }
}
```

패턴인덱스

failure 값

시작부터 반복되는 패턴이 없을 경우 빠져나옴

현재의 문자와 반복 패턴 문자가 다를 경우, 앞부분에서 비교할 위치

Program 2.15: Computing the failure function

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
pat	a	b	c	a	b	a	b	c	a	b	c	a	b	c	a	b	e	a
f	-1	-1	-1	0	1	0	1	2	3	4	2	3	4	2	3	4	-1	0