Chap 2. Arrays and Structures (2)

Contents

2.5 Sparse Matrix

- **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)

- **Sparse matrix**
  - $m \times n$ matrix A : $\dfrac{no.of\ nonzero\ elemnets}{m \times n} \ll 1$

2.5.2 Sparse Matrix Representation

- *An array of triples*
  - <row, column, value> : 3-tuples (triples)
- #define MAX-TERMS 101 /* maximum number of terms +1*/

```
typedef struct {
        int col;
        int row;
        int value;
} term;
term a[MAX-TERMS];
```

	col 0	col 1	col 2	col 3	col 4	col 5
row 0	15	0	0	22	0	−15
row 1	0	11	3	0	0	0
row 2	0	0	0	−6	0	0
row 3	0	0	0	0	0	0
row 4	91	0	0	0	0	0
row 5	0	0	28	0	0	0

	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')

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*)

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

	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

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**
    $b(j, i) \leftarrow a(i, j)$
  end
end

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

15	0	0	0	91	0
0	11	0	0	0	0
0	3	0	0	0	28
22	0	-6	0	0	0
0	0	0	0	0	0
-15	0	0	0	0	0

	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

2.5.3 Transposing a Matrix

❖ **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!

2.5.3 Transposing a Matrix

❖ Using column indices

for all elements in column j

place element <i, j, value> in

element <j, i, value>

	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!

```
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

```
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 *///원소의 개수 만큼, a의 행 인덱스
         for (j = 1; j <= n; j++)
         /* 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

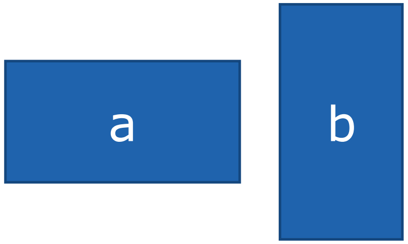
Time complexity : O(*columns* · *elements*)

❖ Analysis of *transpose*

- Nested for loops are the decisive factor.
- The remaining part requires only constant time.
- Time complexity : **O(*columns* · *elements*)**
- If *elements* = *rows* · *columns*, O($columns^2$ · *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(*columns* · *rows*)

2.5.3 Transposing a Matrix

❖ 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

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

① calculation of rowTerms

	[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 =	1	3	4	6	8	8
완료 후 StartingPos 값 →	3	4	6	8	8	9

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

calculation of rowTerms

calculation of startingPos

b(j,i) ← a(i,j)

```
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;
    }
  }
}
```

	[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*)

Fast transpose of a sparse matrix(cont')

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

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

$a[0][0][0]$	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
$a[i_0][0][0]\ldots[0]$	$\alpha + i_0\, upper_1\, upper_2 \ldots upper_{n-1}$
$a[i_0][i_1][0]\ldots[0]$	$\alpha + i_0\, upper_1\, upper_2 \ldots upper_{n-1}$ $+ i_1\, upper_2\, upper_3 \ldots upper_{n-1}$

$a[i_0][i_1]\ \ldots\ [i_{n-1}]$

$$
\begin{aligned}
&\alpha + i_0 upper_1 upper_2 \ldots upper_{n-1} \\
&\quad + i_1 upper_2 upper_3 \ldots upper_{n-1} \\
&\quad + i_2 upper_3 upper_4 \ldots upper_{n-1} \\
&\quad \cdot \\
&\quad \cdot \\
&\quad \cdot \\
&\quad + i_{n-2} upper_{n-1} \\
&\quad + i_{n-1}
\end{aligned}
$$

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

2.7 Strings

ADT *String* is
  objects: a finite set of zero or more characters.
  functions:
    for all $s, t \in$ *String*, $i, j, m \in$ non-negative integers

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 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 $i, i+1, \cdots, i+j-1$. **else return** *NULL*.

ADT 2.4: Abstract data type *String*

2.7.3 Pattern matching

- **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))  // t is start address of pat in string
        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

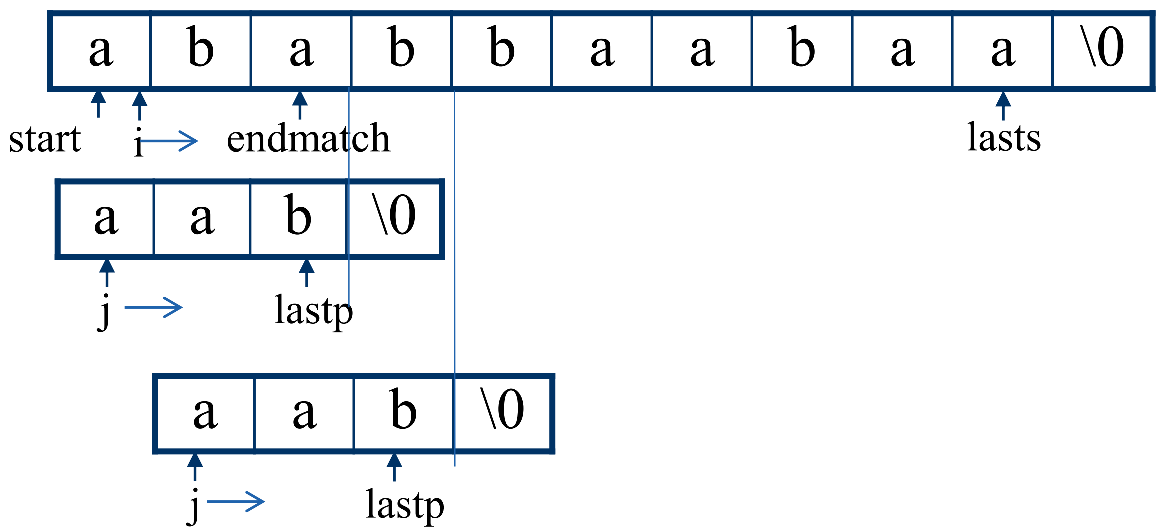
```c
#include <stdio.h>
#include <string.h>
void main()
{
char *string = "This is a test program for strsrt function ";
char *pat = "test";
char *t;
if (t = strstr(string, pat))
        printf("The string from strstr is : %s₩n", t);
else
        printf("The pattern was not found with strstr₩n");
}
```

C:₩WINDOWS₩system32₩cmd.exe

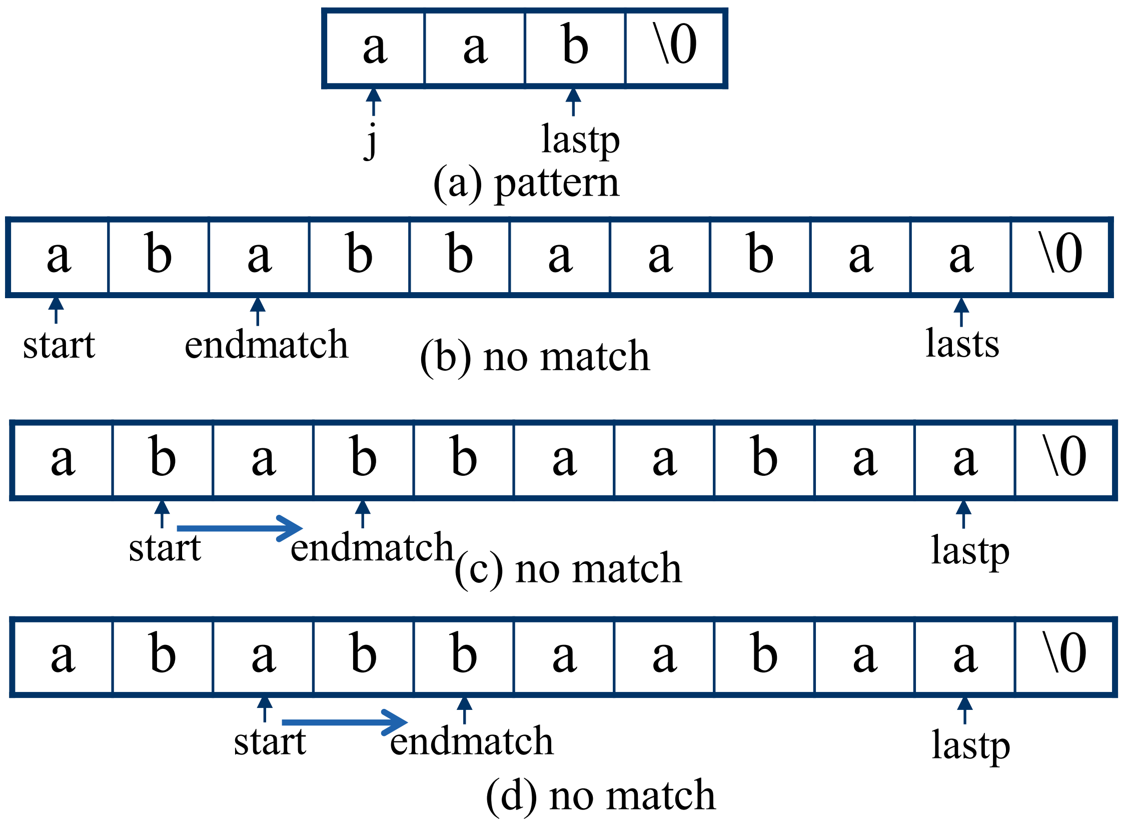
```
The string from strstr is : test program for strsrt function
계속하려면 아무 키나 누르십시오 . . .
```

Pattern matching

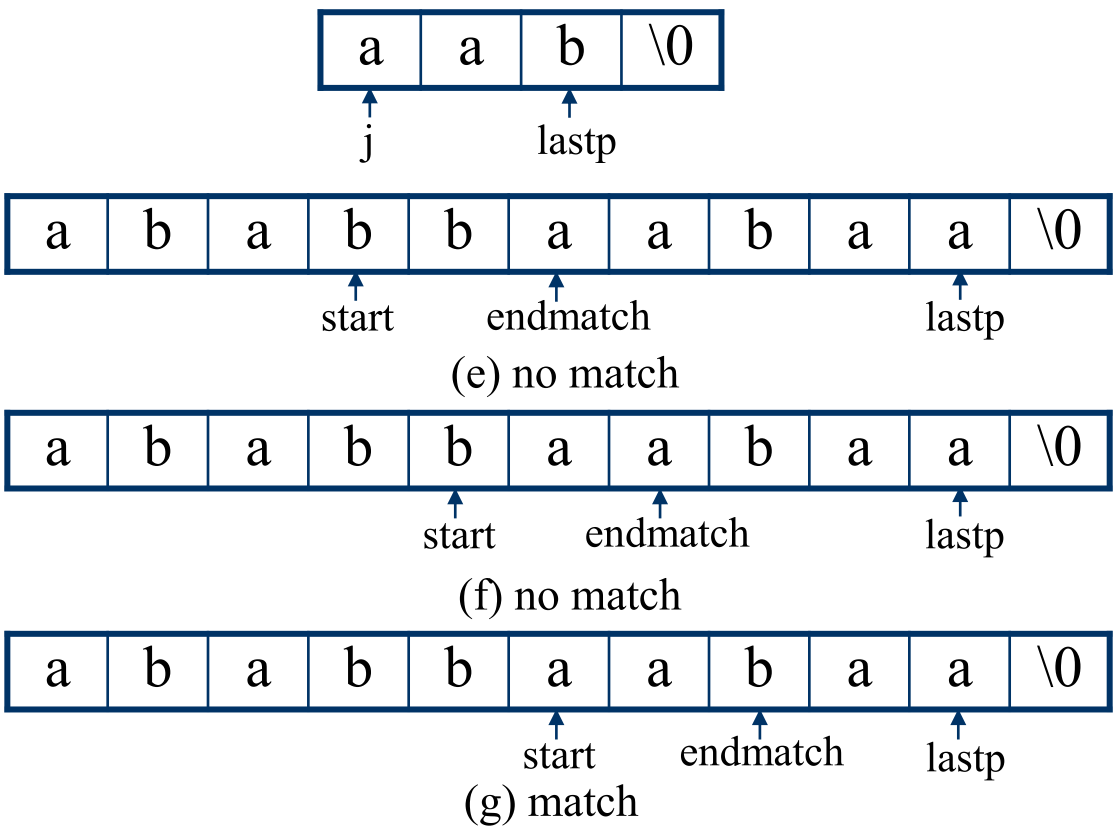
- **nfind simulation**



Pattern matching



Pattern matching



(e) no match

(f) no match

(g) match

Pattern matching

- **nfind simulation**
  - Assume pat = "aab" and string = "ababbaabaa“
    - Point end of string to lastp,
    - Point end of pat array to lasts
  - 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.

Pattern matching

- **Pattern matching that checks the last character of the pattern first**
- **Time complexity : O(lasts*lastp)**

```
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;
}
```

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

Pattern matching

- 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.

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

  - 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