Chapter 2
Arrays and structures

2024 Spring

Ri Yu
Ajou University

Contents

Arrays
Dynamically Allocated Arrays
Structures and Unions
Polynomials
Sparse Matrices
Strings

Arrays

- A consecutive set of memory (Implementation perspective)
- A collection of data of the same type
- A set of pairs <index, value>
  - For each index, there is a value associated with it

- Operations on an array
  - Creating a new array
  - Retrieving a value
  - Storing a value

Array: ADT

❖ **Structure** *Array* is

Objects: A set of pairs <*index, value*> where for each value of *index* there is a value from the set *item*. *Index* is a finite ordered set of one or more dimensions, for example, {0, … , *n*-1} for one dimension, {(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)} for two dimensions, etc.

Functions:

for all *A*∈*Array*, *i* ∈*index*, *x*∈*item, j, size*∈integer

Array Create(*j, list*) ::= **return** an array of *j* dimensions where list is a *j*-tuple whose *i*th element is the size of the *i*th dimension. *Items* are undefined.

Item Retrieve(*A, i*) ::= **if** (*i*∈*index*) **return** the item associated with index value *i* in array *A*
else return error

Array Store(*A, i, x*) ::= **if** (*i* in *index*)
return an array that is identical to array *A* except the new pair <*i, x*> has been inserted

else return error

end array

Array in C

- One-dimensional array
  - Array of 5 integers: int list[5];
  - Array of 5 pointers to integers: int *plist[5];

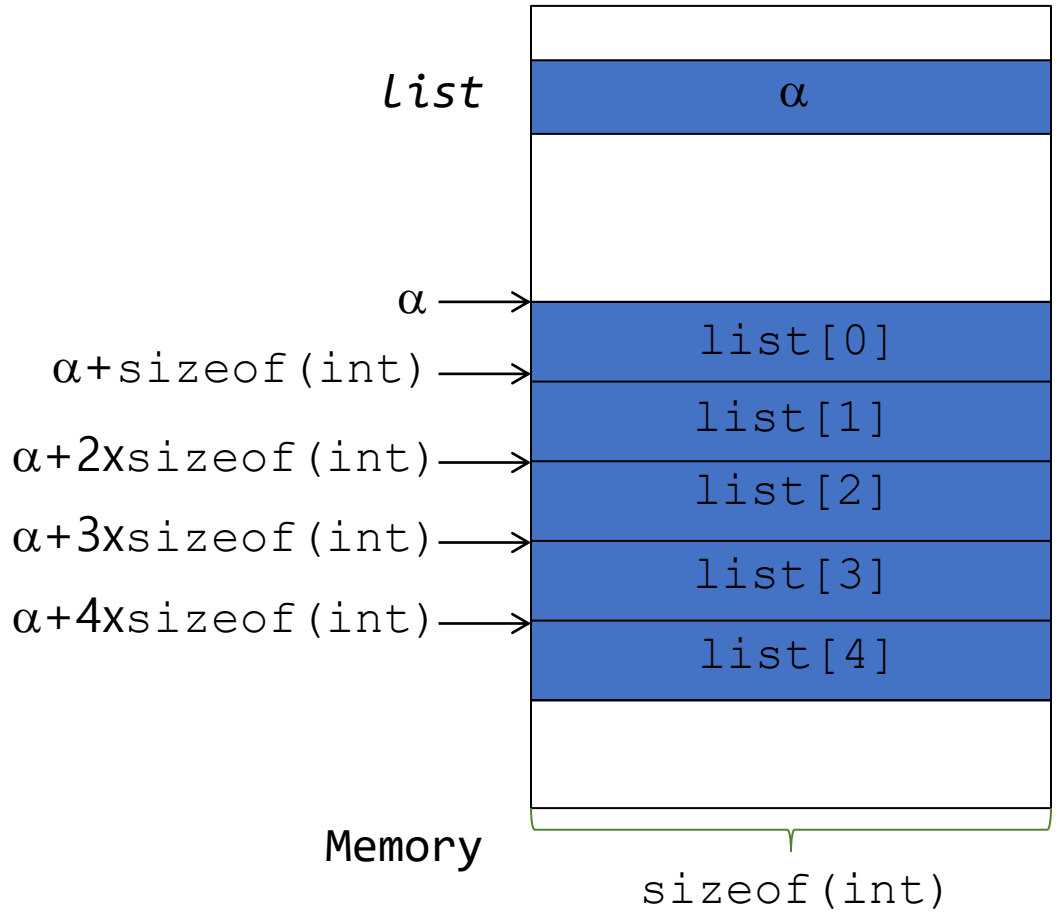
Five array elements, each of which contains a pointer to an integer

Array in C

int list[5];

Variable	Memory Address
list[0]	base address = α
list[1]	α + *sizeof*(*int*)
list[2]	α + 2 x *sizeof*(*int*)
list[3]	α + 3 x *sizeof*(*int*)
list[4]	α + 4 x *sizeof*(*int*)

list[*i*] : α + *i* x *sizeof*(*int*)



Example

```
#define MAX_SIZE 100
float sum(float [], int);
float input[MAX_SIZE], answer;
void main(void)
{
        int i;
        for (i = 0; i < MAX_SIZE; i++)
                input[i] = i;
        answer = sum(input, MAX_SIZE);        /* input = &input[0] */
        print("The sum is: %d\n", answer);
}

/* input is copied into a temporary location and associated with the formal parameter list */
float sum(float list[], int n)
{
        int i;
        float tempsum = 0;
        for (i = 0; i < n; i++)
                tempsum += list[i];
         return tempsum;
}
```

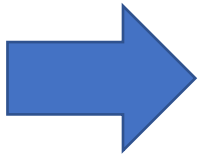
Example

❖ One-dimensional array addressing

Write a function that prints out both the address of the i-th element of the array and the value found at this address:

```
int one[] = {0, 1, 2, 3, 4};
```

```
void print1 (int *ptr, int rows )          /* invoked as print1( &one[0], 5 ) */
{
          /* print out a one-dimensional array using a pointer */
          int i;
          printf( "Address Contents\n" );
          for ( i=0; i < rows; i++ )
                    printf( "%8u%5d\n", ptr+i, *(ptr+i) );
          printf( "\n" );
}
```



Address	Contents
12344868	0
12344872	1
12344876	2
12344880	3
12344884	4

Dynamically Allocated Arrays

- One-dimensional arrays

```
#include <math.h>
#include <stdio.h>
#define MAX_SIZE 101

main()
{
 int i, n, list[MAX_SIZE];

 printf("Enter the number of number to generate: ");
 scanf("%d", &n);

 for (i=0;i<n;i++)
 {
        list[i]=rand()%1000;
        printf ("%d\n", list[i]);
 }
}
```

What if
the size of an array were unknown?

Dynamically Allocated Arrays

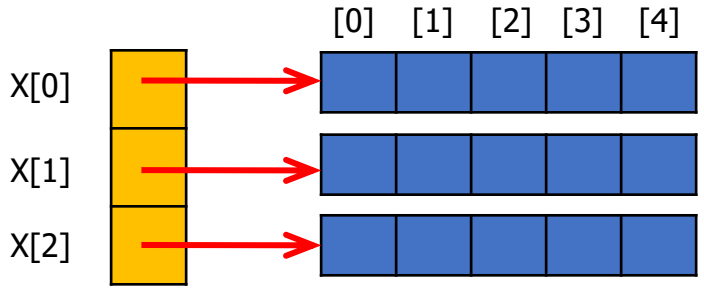
```
int i, n, *list;

printf("Enter the number of number to generate: ");
scanf("%d", &n);

if (n < 1) {
        fprintf(stderr, "Improper value of n \n");
        exit (EXIT_FAILURE);
}

list=malloc(n * sizeof(int));
if (list==NULL) {
        fprintf(stderr, "lack of memory\n");
        exit (EXIT_FAILURE);
}
```

To defer this decision to run time

It fails only when n<1 or we do not have sufficient memory to hold the list of numbers

Dynamically Allocated Arrays

- Two-dimensional arrays
  - One-dimensional array in which each element is, itself, a one-dimensional array
    - int x[3][5];
      - x's length is 3 and
      - each element of x is a one-dimensional array whose length is 5



Array-of-arrays representation

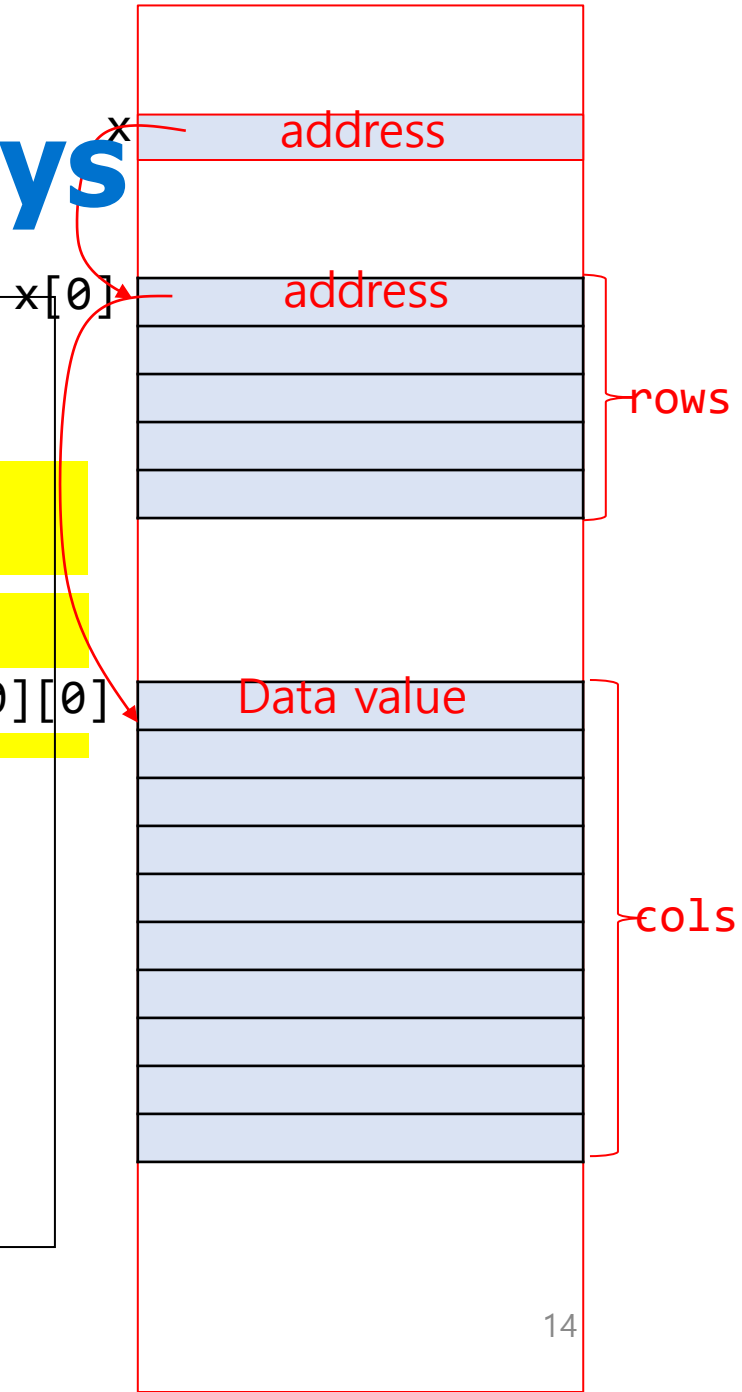
Dynamically Allocated Arrays

```
int** make2dArray(int rows, int cols)
{       /* create a two dimensional rows × cols array */
        int** x, i;
        /* get memory for row pointers */
        MALLOC(x, rows * sizeof (*x));

        /* get memory for each row */
        for (i = 0; i < rows; i++)
                MALLOC(x[i], cols * sizeof(**x));
        return x;
}

void main()
{
        int** myArray;
        myArray = make2dArray(5,10);
        myArray[2][4]=6;
}
```

```
#define MALLOC(p,s) \
  if (!((p)=malloc(s))) {\
     fprintf(stderr, “Insufficient memory”);\
     exit(EXIT_FAILURE);\
}  (Chapter 1, p.7)
```

Dynamically Allocated Arrays

```
int** make2dArray(int rows, int cols)
{       /* create a two dimensional rows × cols array */
        int** x, i;
        /* get memory for row pointers */
        MALLOC(x, rows * sizeof (*x));

        /* get memory for each row */
        for (i = 0; i < rows; i++)
                MALLOC(x[i], cols * sizeof(**x));
        return x;
}

void main()
{
        int** myArray;
        myArray = make2dArray(5,10);
        myArray[2][4]=6;
}
```



Structures

- Arrays : Collections of data of the same type

- Structure : an alternate way to group data
  - Permits the data to vary in type
  - A collection of data items
    - Each item is identified as to its type and name

- Accessing structures
  - Structure member operator: "."

```
struct {
        char    name[10];
        int     age;
        float   salary;
} person;

strcpy(person.name, "james");
person.age=10;
person.salary=35000;
```

Structures

- Creating our own structure data types: **typedef**

```
typedef struct humanBeing {
          char    name[10];
          int       age;
          float    salary;
};
```

or

```
typedef struct {
          char     name[10];
          int        age;
          float     salary
} humanBeing;
```

- Declarations of variables
  - humanBeing person1, person2;

If person1's birthday is May 28, 2021,
person1.dob.month = 5;
person1.dob.day = 28;
person1.dob.year = 2021;

- Embedding a structure within a structure

```
typedef struct {
          int month;
          int day;
          int year;
} date;
```

```
typedef struct humanBeing {
          char name[10];
          int age;
          float salary;
          date dob;
};
```

Unions

- Similar to a structure
- But the fields of a union must share their memory space
  - Only one field is active at any given time
- Example : adding fields for male and female

```
#define   TRUE      1
#define   FALSE     0
typedef struct sexType {
          enum tagField {female, male} sex;
          union {
                    int children;
                    int beard;
          } u;
};
```

```
typedef struct humanBeing {
          char          name[10];
          int           age;
          float         salary;
          date          dob;
          sexType       sexInfo;
};
```

Unions

```
#define   TRUE     1
#define   FALSE    0
typedef struct sexType {
          enum tagField {female, male} sex;
          union {
                    int children;
                    int beard;
          } u;
};
```

```
typedef struct humanBeing {
          char          name[10];
          int           age;
          float         salary;
          date          dob;
          sexType       sexInfo;
};
```

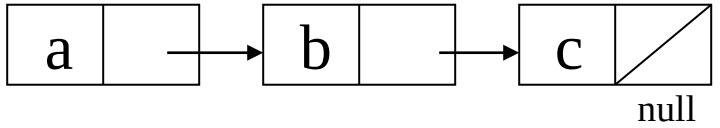
```
humanBeing person1, person2;

person1.sexInfo.sex=male;
person1.sexInfo.u.beard=FALSE;

person2.sexInfo.sex = female;
person2.sexInfo.u.children = 4;
```

First place a value in the tag field to determine which field in the union is active

Self-Referential Structures 자기 참조 구조

- One or more of its components is a pointer to itself

```
typedef struct list* listPtr;
 typedef struct list {
        char data;
        listPtr link;
};

list item1, item2, item3;

item1.data = 'a';
item2.data = 'b';
item3.data = 'c';
item1.link = item2.link = item3.link = NULL;
 item1.link = &item2;
 item2.link = &item3;
```



Ordered List 순서 리스트

- Ordered (linear) list: ($item_0$, $item_1$, $item_2$, ..., $item_{n-1}$)
- Examples:
  - Days of the week
    - (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday)
  - Values in a deck of cards
    - (Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King)
  - Years the United States fought in World War II
    - (1941, 1942, 1943, 1944, 1945)
  - Years Switzerland fought in World War II
    - (): an empty list

Operations on Ordered List

❖ Examples :

① Find the length, *n* , of the list.
② Read the items from left to right (or right to left).
③ Retrieve the *i* th element
④ Store a new value into the *i* th position
⑤ Insert a new element at the position *i* , causing elements numbered *i*, *i*+1, …, *n* to become numbered *i*+1, *i*+2, …, *n*+1
⑥ Delete the element at position *i* , causing elements numbered *i*+1, …, *n* to become numbered *i*, *i*+1, …, *n*-1

- Array: sequential mapping
  - Associate the list element, $item_i$, with the array index *i*
  - Works well for the operations of ①~④
  - But not for the operations of ⑤~⑥
    - Leads us to consider non-sequential mappings of ordered lists in Chapter 4

Polynomials ADT 다항식

❖ Polynomials

- $A(x) = 3x^2 + 2x + 4$ $B(x) = x^4 + 10x^3 + 3x^2 + 1$

❖ **ADT** *Polynomial*

Objects: $p(x) = a_1 x^{e_1} + \ldots + a_n x^{e_n}$; a set of ordered pairs of $<e_i, a_i>$ where a_i in *Coefficients* and e_i in *Exponents*, e_i are integers >= 0

Functions:

for all *poly, poly1, poly2* ∈ *Polynomial*, *coef* ∈ *Coefficients*, *expon* ∈ *Exponents*

Polynomial	Zero( )	::= **return** the polynomial, $p(x) = 0$
Boolean	IsZero(*poly*)	::= **if** (*poly*) **return** *FALSE* **else return** *TRUE*
Coefficient	Coef(*poly, expon*)	::= **if** (*expon* ∈ *poly*) **return** its coefficient **else return** zero
Exponent	LeadExp(*poly*)	::= **return** the largest exponent in *poly*

Polynomials ADT (Cont.)

Polynomial Attach(*poly, coef, expon*) ::= **if** (*expon* $\in$ *poly*) **return** error
else return the polynomial *poly*
with the term <*coef, expon*> inserted

Polynomial Remove(*poly, expon*) ::= **if** (*expon* $\in$ *poly*)
return the polynomial *poly* with the
term whose exponent is *expon* deleted
else return error

Polynomial SingleMult(*poly, coef, expon*) ::= **return** the polynomial
$poly \cdot coef \cdot x^{expon}$

Polynomial Add(*poly1, poly2*) ::= **return** the polynomial
poly1 + *poly2*

Polynomial Mult(*poly1, poly2*) ::= **return** the polynomial
poly1 $\cdot$ *poly2*

End *Polynomial*

Polynomial Representation 1

❖ Store the coefficients in order of decreasing exponents

```
#define MAX_DEGREE 101
typedef struct {
        int degree;
        float coef[MAX_DEGREE];
} polynomial;
```

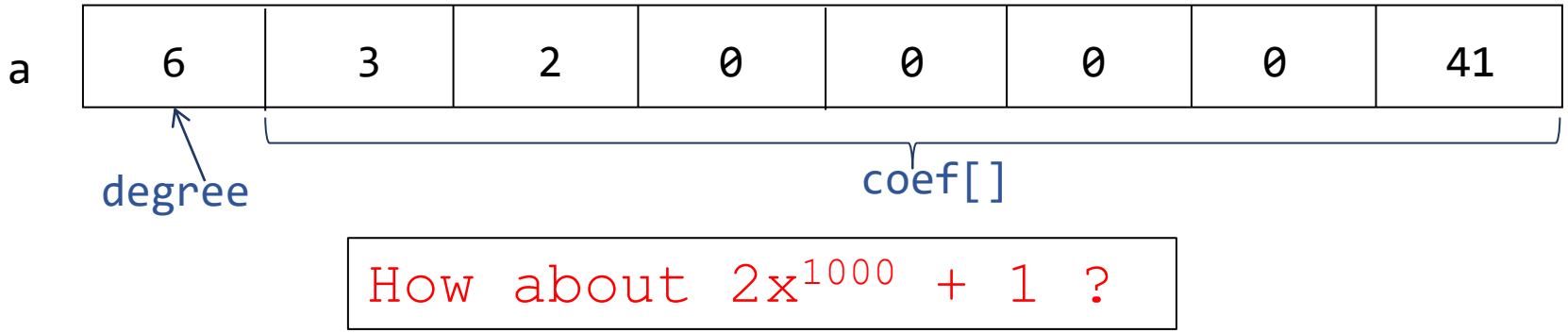
Polynomial Representation 1

❖Example :

$A(x) = 3x^6 + 2x^5 + 41$

polynomial a;

$a.degree = n$, $a.coef[i] = a_{n-i}$, $0 \le i \le n$



How about $2x^{1000} + 1$?

- Leads to very simple algorithms for most of the operations
- Wastes a lot of space when sparse

Polynomial Representation 2

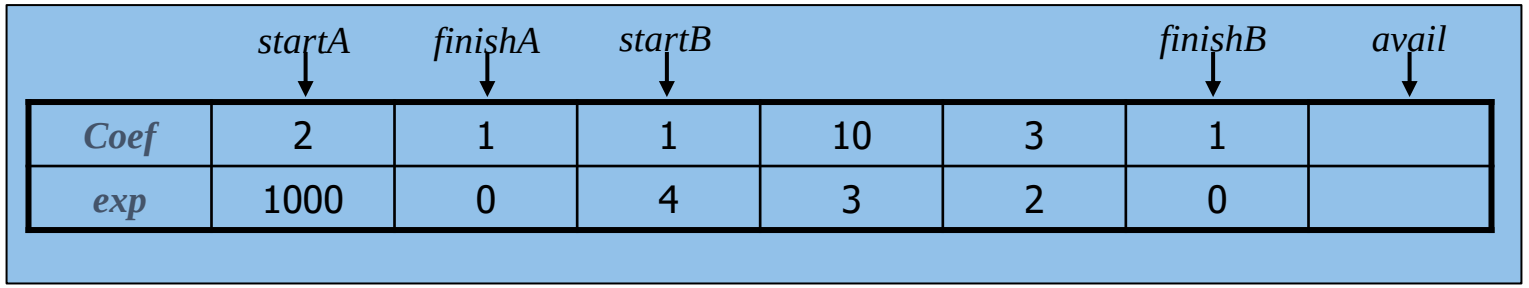
❖Use one global array to store all polynomials

```
#define MAX_TERMS 100
    Typedef struct {
      float coef;
      int expon;
    } polynomial;
    polynomial terms[MAX_TERMS];
```

Polynomial Representation 2

❖ Example:

$A(x)=2x^{1000}+1$ and $B(x)=x^4+10x^3+3x^2+1$



- startA , startB : the index of the first term of A and B
- finishA, finishB : the index of the last term of A and B
- avail : the index of the next free location in the array

- Storage requirements: start, finish, 2*(finish-start+1)
- When all terms are non zero : twice as much space as the first one

Polynomial Addition

```
void padd (int startA, int finishA, int startB, int finishB, int *startD, int *finishD)
{       /* add A(x) and B(x) to obtain D(x) */
        float coefficient;
        *startD = avail;
        while ( startA <= finishA && startB <= finishB )
                switch ( COMPARE(terms[startA].expon, terms[startB].expon) )
                        {
                                case -1:          /* a expon < b expon */
                                attach( terms[startB].coef, terms[startB].expon );
                                startB++;
                                break;
                                case  0:          /* equal exponents */
                                coefficient = terms[startA].coef + terms[startB].coef;
                                if ( coefficient )
                                        attach ( coefficient, terms[startA].expon );
                                startA++;
                                startB++;
                                break;
                                case 1:            /* a expon > b expon */
                                attach( terms[startA].coef, terms[startA].expon );
                                startA++;
                        }   /* end of switch */
```

Polynomial Addition

```
        /* add in remaining terms of  A(x) */
        for( ; startA <= finishA; startA++ )
        {
                attach( terms[startA].coef, terms[startA].expon );
        }
        /* add in remaining terms of B(x) */
        for( ; startB <= finishB; startB++ )
        {
                attach( terms[startB].coef, terms[startB].expon );
        }
        *finishD =avail -1;
}
```

- Analysis of padd
  - The number of non-zero terms in A and in B are the most important factors in the time complexity
  - O(n+m)
    - m: the number of non-zero terms in A
    - n: the number of non-zero terms in B

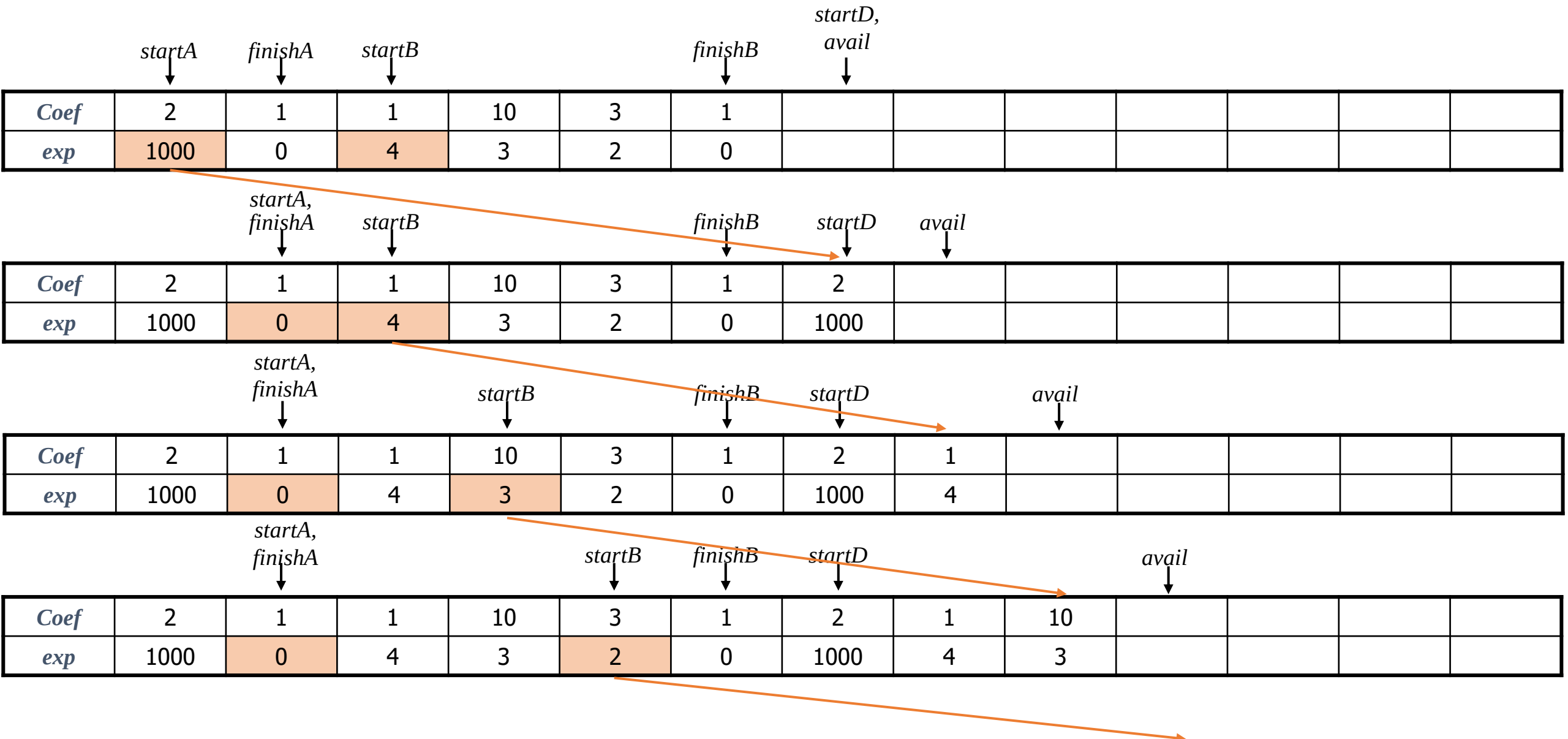
Polynomial Addition

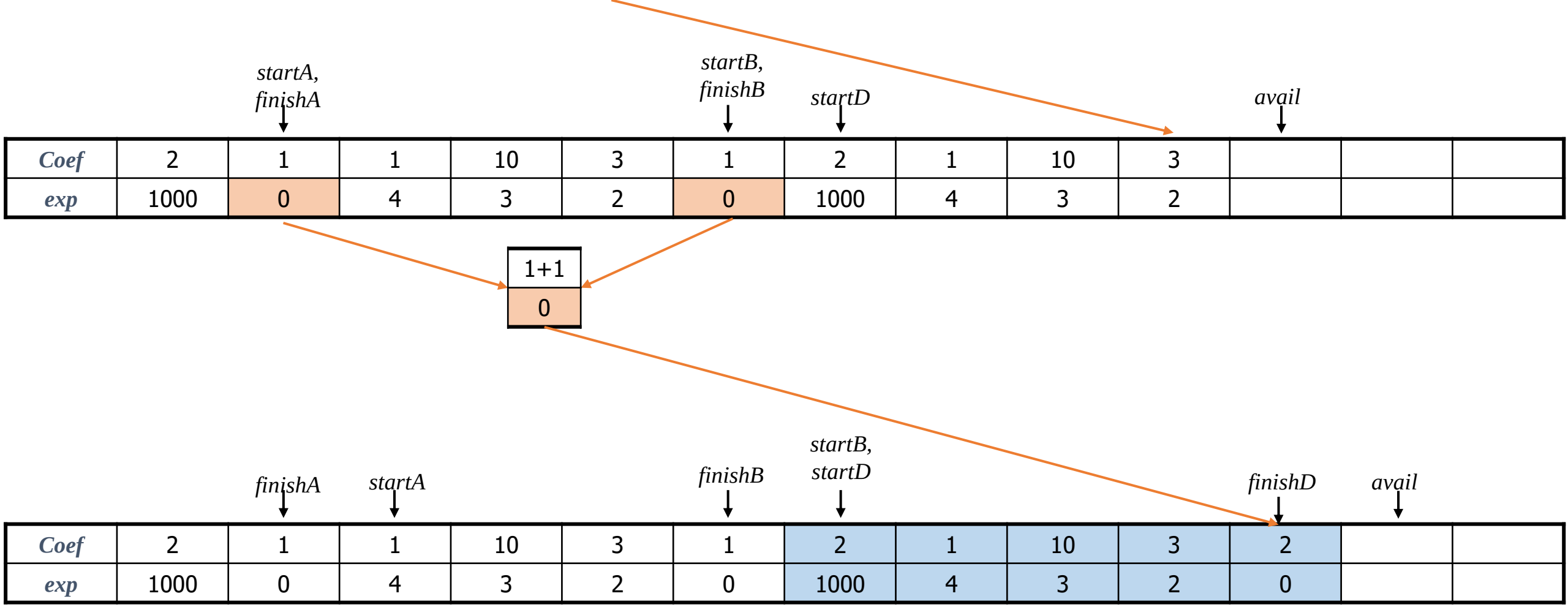
```
#define MAX_TERMS 100
    Typedef struct {
        float coef;
        int expon;
    } polynomial;
    polynomial terms[MAX_TERMS];
```

```
void attach(float coefficient, int exponent)
{
        /* add a new term to the polynomial */
        if (avail >= MAX_TERMS)
        {
                fprintf(stderr, “Too many terms in the polynomial\n”);
                exit(EXIT_FAILURE);
        }
        terms[avail].coef  = coefficient;
        terms[avail++].expon = exponent;
}
```

- Problem
  - Compaction is required when polynomials that are no longer needed
  - Data movement takes time

$A(x) = 2x^{1000} + 1\,, \qquad B(x) = x^4 + 10x^3 + 3x^2 + 1$





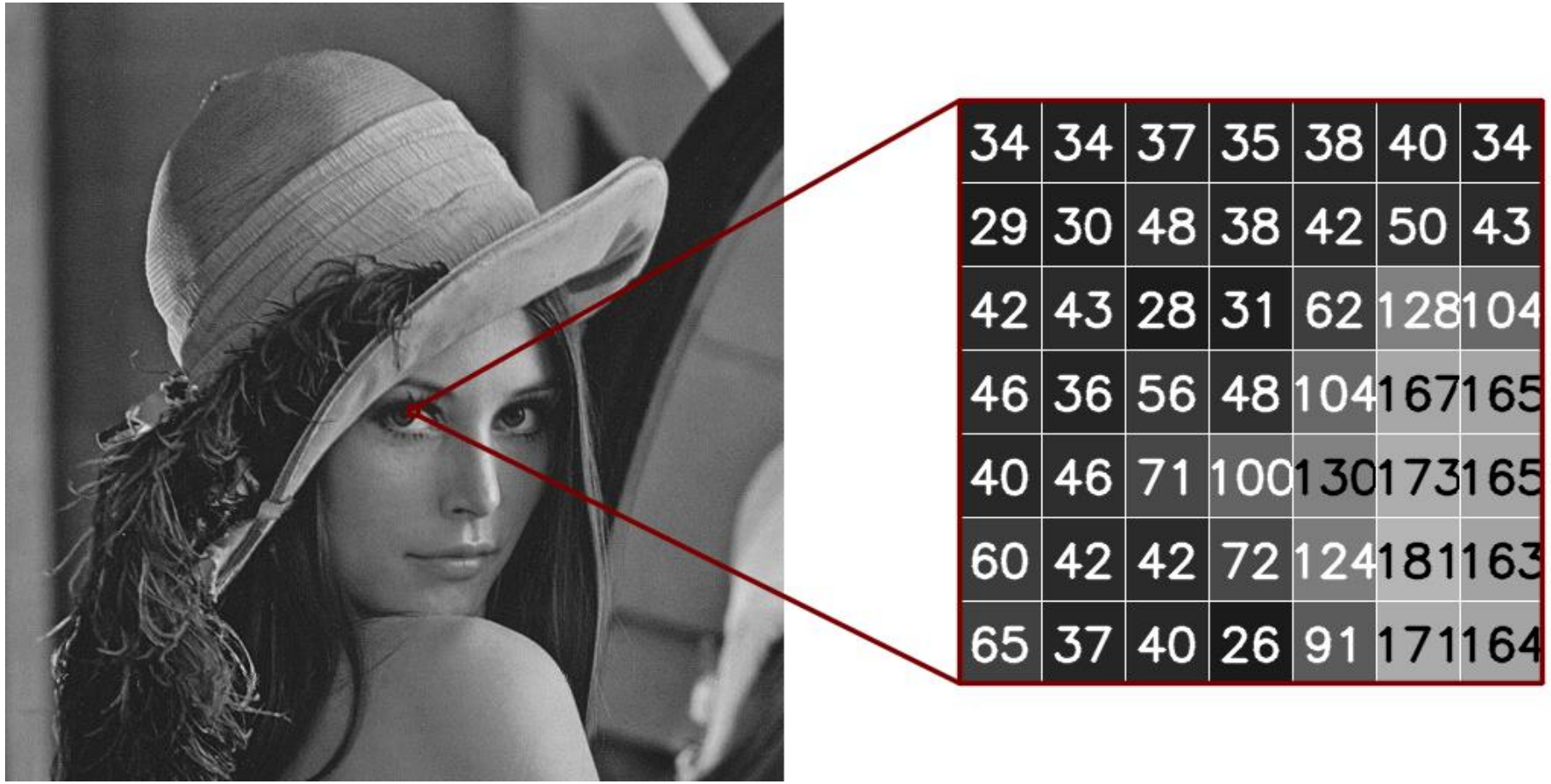
$$D(x) = 2x^{1000} + x^4 + 10x^3 + 3x^2 + 2$$

Matrix 행렬

Matrices are used in most areas of mathematics and most scientific fields.

Ex) Digital image representation

Matrix


34 34 37 35 38 40 34
29 30 48 38 42 50 43
42 43 28 31 62 128 104
46 36 56 48 104 167 165
40 46 71 100 130 173 165
60 42 42 72 124 181 163
65 37 40 26 91 171 164

Sparse Matrices 희소행렬

❖ A matrix contains *m* rows and *n* columns of elements as illustrated

	Col0	Col1	Col2
Row0	-27	3	4
Row1	6	82	-2
Row2	109	-64	11
Row3	12	8	9
Row4	48	27	47

	Col0	Col1	Col2	Col3	Col4	Col5
Row0	15	0	0	22	0	-15
Row1	0	11	3	0	0	0
Row2	0	0	0	-6	0	0
Row3	0	0	0	0	0	0
Row4	91	0	0	0	0	0
Row5	0	0	28	0	0	0

Sparse Matrices

- A sparse matrix
  - A matrix containing many zero entries
  - Representing a sparse matrix as a two-dimensional array
    - Wastes space

	Col0	Col1	Col2
Row0	-27	3	4
Row1	6	82	-2
Row2	109	-64	11
Row3	12	8	9
Row4	48	27	47

	Col0	Col1	Col2	Col3	Col4	Col5
Row0	15	0	0	22	0	-15
Row1	0	11	3	0	0	0
Row2	0	0	0	-6	0	0
Row3	0	0	0	0	0	0
Row4	91	0	0	0	0	0
Row5	0	0	28	0	0	0

Sparse Matrix: ADT

❖ Structure *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* ∈ *SparseMatrix*, *x* ∈ *item*, *i*, *j*, *maxCol*, *maxRow* ∈ *index*

SparseMatrix Create(*maxRow* , *maxCol*) ::=
return a *SparseMatrix* that can hold up to
maxItems = *maxRow* x *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

Sparse Matrix: ADT

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*] = Σ (*a*[*i*][*k*]·*b*[*k*][*j*]) where *d*(*i*, *j*) is the (*i*, *j*)th element
else return error

Sparse Matrix

```
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];
```

❖ Sparse matrix representation

- Each element is characterized by using the triple <row, col, value>

	Col0	Col1	Col2	Col3	Col4	Col5
Row0	15	0	0	22	0	-15
Row1	0	11	3	0	0	0
Row2	0	0	0	-6	0	0
Row3	0	0	0	0	0	0
Row4	91	0	0	0	0	0
Row5	0	0	28	0	0	0

	row	col	value
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

number of rows, columns, and values of a respectively

Ordered by row and within rows by columns

Example

	Col0	Col1	Col2
Row0	0	3	0
Row1	0	0	-2
Row2	0	0	0
Row3	0	8	0
Row4	48	0	0

Number of elements

Number of columns

Number of rows

	row	col	value
a[0]	5	3	4
a[1]	0	1	3
a[2]	1	2	-2
a[3]	3	1	8
a[4]	4	0	48

Approximate Memory Requirements

- 500 x 500 matrix with 1994 nonzero elements, 4 bytes per element

  2D array: 500 x 500 x 4 = 1M bytes = 1000K bytes

  1D array of triples: 3 x 1995 x 4 ≈ 23K bytes

Transposing a Matrix 행렬의 전치

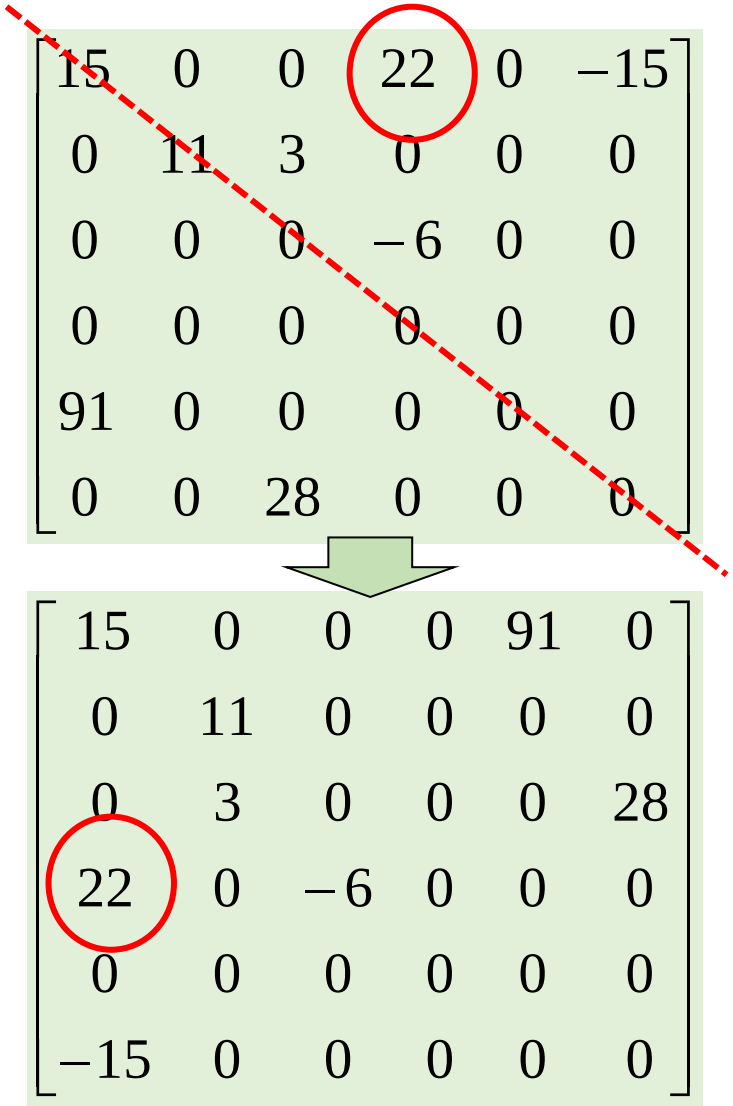
- Transpose: interchange the rows and columns

  1) for each row *i*
     - Take element <*i*, *j*, *value*> and store it as element <*j*, *i*, *value*> of the transpose
     - Unfortunately don't know exactly where to place element <*j*, *i*, *value*>
       - Until we have processed all the elements that precede it
     - Example

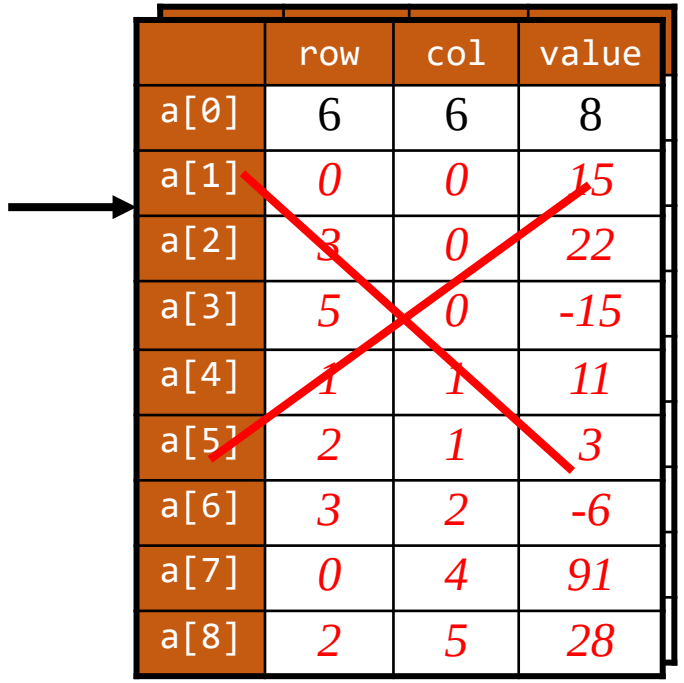
       (0, 0, 15) ====> (0, 0, 15)
       (0, 3, 22) ====> (3, 0, 22)
       (0, 5, -15) ====> (5, 0, -15)

  2) for all elements in column *j*,
     - Place element <*i*, *j*, *value*> in element <*j*, *i*, *value*>
     - O(*columns* x *elements*)
       - Scan the array "column" times and the array has "elements" elements

Transposing a Matrix



	row	col	value
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



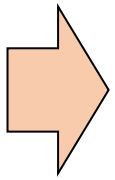
	row	col	value
a[0]	6	6	8
a[1]	*0*	*0*	*15*
a[2]	*3*	*0*	*22*
a[3]	*5*	*0*	*-15*
a[4]	*1*	*1*	*11*
a[5]	*2*	*1*	*3*
a[6]	*3*	*2*	*-6*
a[7]	*0*	*4*	*91*
a[8]	*2*	*5*	*28*

Transposing a Matrix

for each row i

take element <i,j,value> and store it as element <j,i,value>

	row	col	value
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



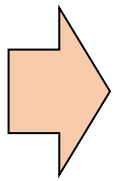
	row	col	value
b[0]			
b[1]			
b[2]			
b[3]	a[??]	3	0	22
b[4]			
b[5]			
b[6]			
b[7]			
b[8]			

Transposing a Matrix

for all elements in column j

place element <i,j,value> in element <j,i,value>

	row	col	value
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



	row	col	value
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

Transposing a Matrix

```
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++){ /* transpose by the column */
              for (j = 1; j <= n; j++){ /* scan all the elements */
                    if (a[j].col == i){
                          b[currentb].row = a[j].col;
                          b[currentb].col = a[j].row;
                          b[currentb].value = a[j].value;
                          currentb++;
                    } /* end of if */
              } /* end of for j */
        } /* end of for i */
  } /* end of if */
}
```

→ $O(columns \cdot elements)$

Fast Transposing a Matrix

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

3	0	22

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

```
2 elements in row 0
1 element in row 1
2 elements in row 2
```

Fast Transposing a Matrix

- Step 1: #non-zero in each row of transpose = #non-zero in each column of original matrix

- Step2: Starting position of each row of transpose

  = sum of size of preceding rows of transpose

- Step 3: Move elements, left to right, from original matrix to transpose matrix

Number of non-zero elements in each row of transpose matrix

Starting position of each row of transpose matrix

Original matrix

	row	col	value
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

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

startingPos

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

```
startingPos[0] = 1;
for(i = 1; i < num_cols; i++){
  startingPos[i]=
     startingPos[i-1]+
     rowTerms[i-1];
}
```

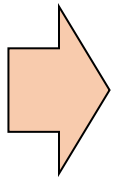
a[2]: <0,3,22> → b[startingPos[3]]: <3,0,22>

```
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]
startingPos	3	4	6	8	8	9

Original matrix

	row	col	value
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



Transpose matrix

	row	col	value
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

→ O (*columns + elements*)

Fast Transposing a Matrix

```
void fastTranspose(term a[], term 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) {/* non-zero 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;
                }
        }
}
```

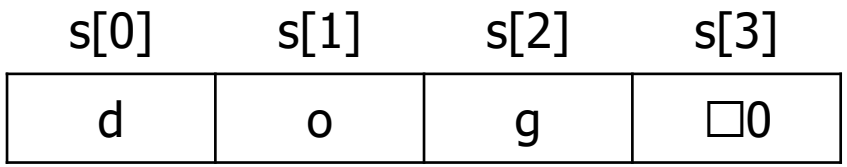
→ O (columns + elements)

Strings ADT

- *String* is a finite set of zero or more characters

- String representation in C

```
#define MAX_SIZE 100
char s[MAX_SIZE] = {“dog”};
```



Strings ADT

C언어 표준 라이브러리
#include <string.h>

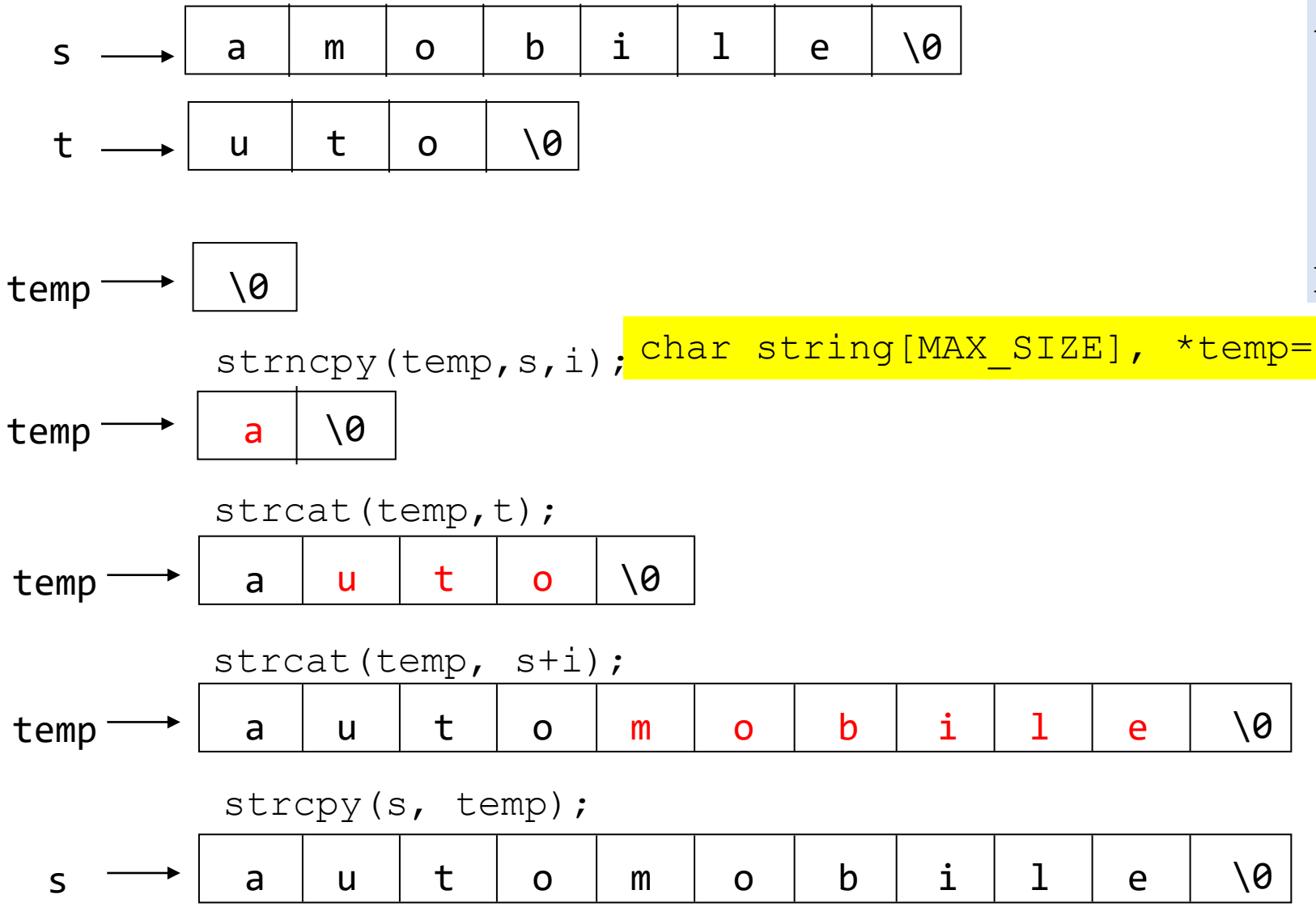
❖ C String functions

- `char *strcat(char *dest, char *src)`
- `char *strncat(char *dest, char *src, int n)`
- `int strcmp(char *str1, char *str2)`
- `int strncmp(char *str1, char *str2, int n)`
- `char *strcpy(char *dest, char *src)`
- `char *strncpy(char *dest, char *src, int n)`
- `size_t strlen(char *s)`
- `char *strchr(char *s, int c)`
- `char *strtok(char *s, char *delimiters )`
- `char *strstr(char *s, char *pat)`
- `size_t strspn(char *s, char *spanset)`
- `size_t strcspn(char *s, char *spanset)`
- `char *strpbrk(char *s, char *spanset)`

String Insertion

- Insert string *t* into string *s* starting at the *i*-th position of string *s*

```
void strnins( char *s, char *t, int 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 );                 /* copy t into s */
        else if ( strlen(t) ) {
                strncpy( temp, s, i ); /* copy i characters from s to temp */
                strcat( temp, t );
                strcat( temp, (s+i) );
                strcpy( s, temp );
        }
}
```




```
void main()
{
  char s[MAX_SIZE]="amobile";
  char t[MAX_SIZE]="uto";
  strnins(s,t,1);
  printf("%s\n", s);
  return;
}
```

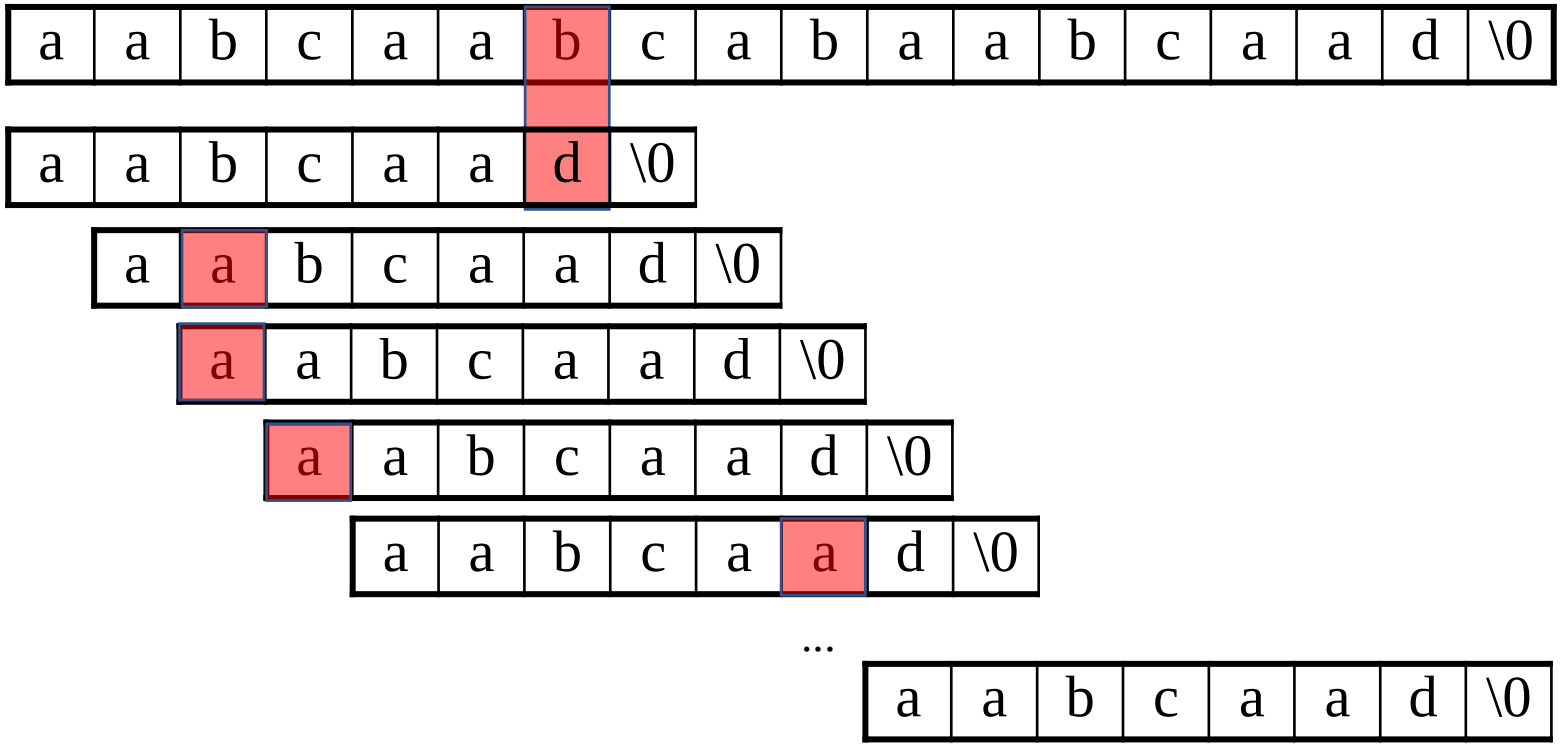
Pattern Matchir

```
#include <string.h>
…
if (t=strstr(string, pat))
   printf("The string from strstr is : %\n", t);
else
   printf("The pattern was not found with strstr\n");
```

- Assume that we have two strings, *string* and *pat*, where *pat* is a pattern to be searched for in *string*
- The easiest way
  - use the built-in function strstr(char *s, char *pat)
    - returns a pointer to start of *pat* in *string*
    - Returns a null pointer if *pat* is not in *string*
- Developing our own pattern matching function
  - The easiest but least efficient method sequentially examines each character of string until it finds the pattern or reaches the end of the string
    - If *pat* is not in *string*, the time complexity is O(*nm*),
      - where *n* denotes the length of *pat* and *m* the length of *string*

Pattern Matching

❖ Naïve approach

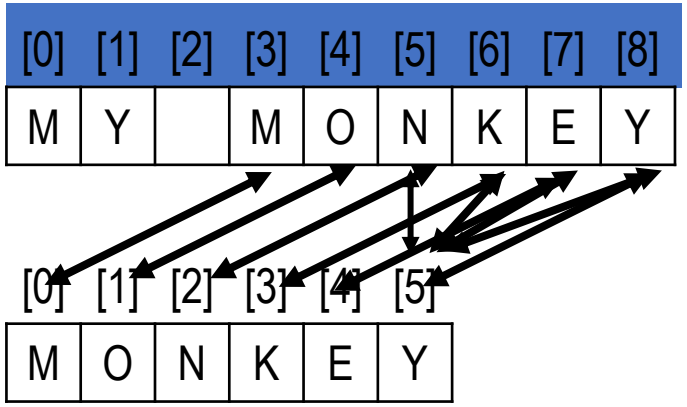


nfind

- Improvements
  - Quitting when strlen(*pat*) is greater than the number of remaining characters in *string*
  - Checking the first and last characters of *pat* and *string* before we check the remaining characters

```
int nfind(char *s, char *pat)
{
    int i, j, start = 0, lasts = strlen(s)-1, lastp = strlen(pat)-1;
    int endmatch = lastp;
    for (i = 0; endmatch <= lasts; endmatch++, start++) {
        if (s[endmatch] == pat[lastp])
            for (j = 0, i = start; j < lastp && s[i] == pat[j]; i++,j++) ;
        if (j == lastp)
            return start;
        }
    return -1;
}
```

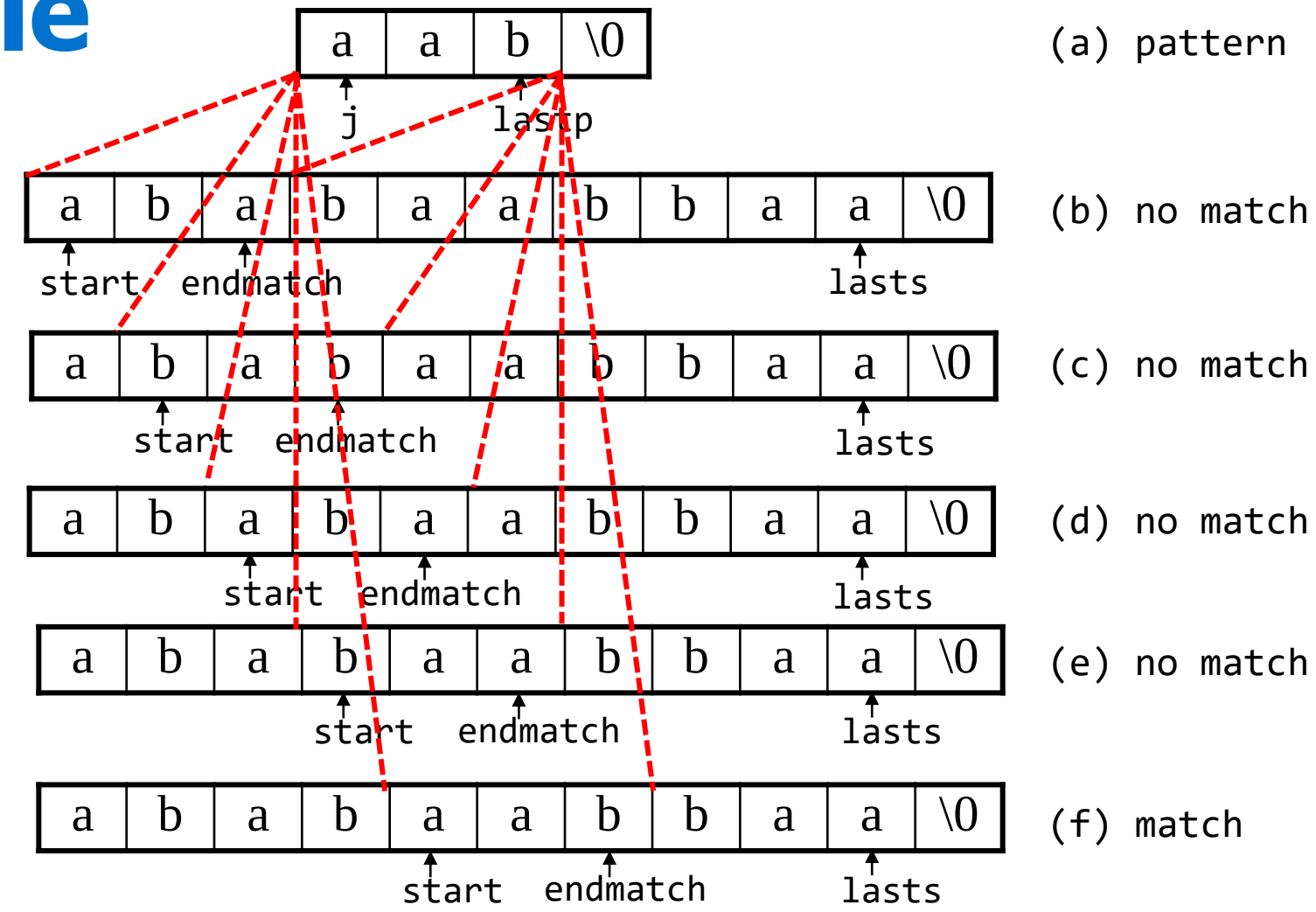
- The worst case of computing time of nfind is O(*nm*), where *n* is the length of *pat* and *m* is the length of *string*



lasts=8
lastp=5
endmatch=
start=
i=
j=

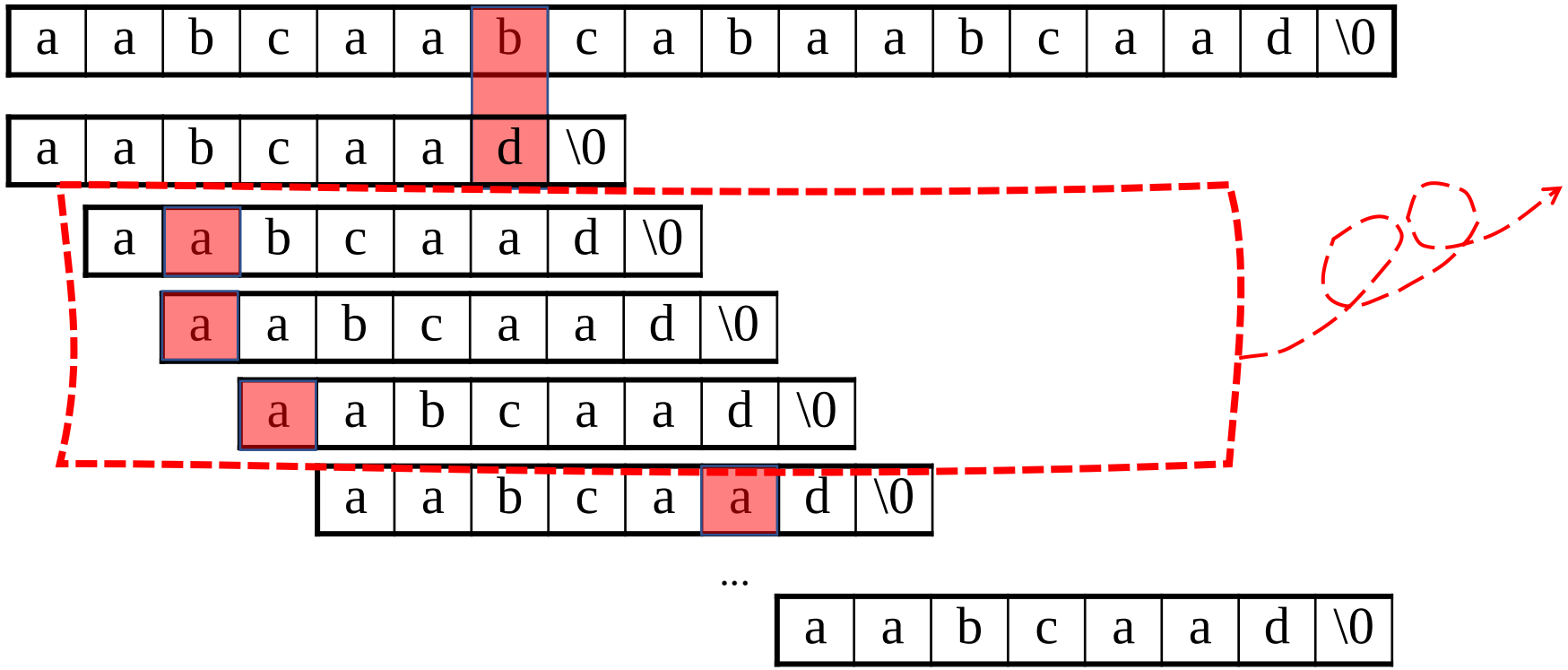
```
int i, j, start = 0
int lasts = strlen(string)-1, lastp = strlen(pat)-1;
int endmatch = lastp;

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;
        }
   }
```

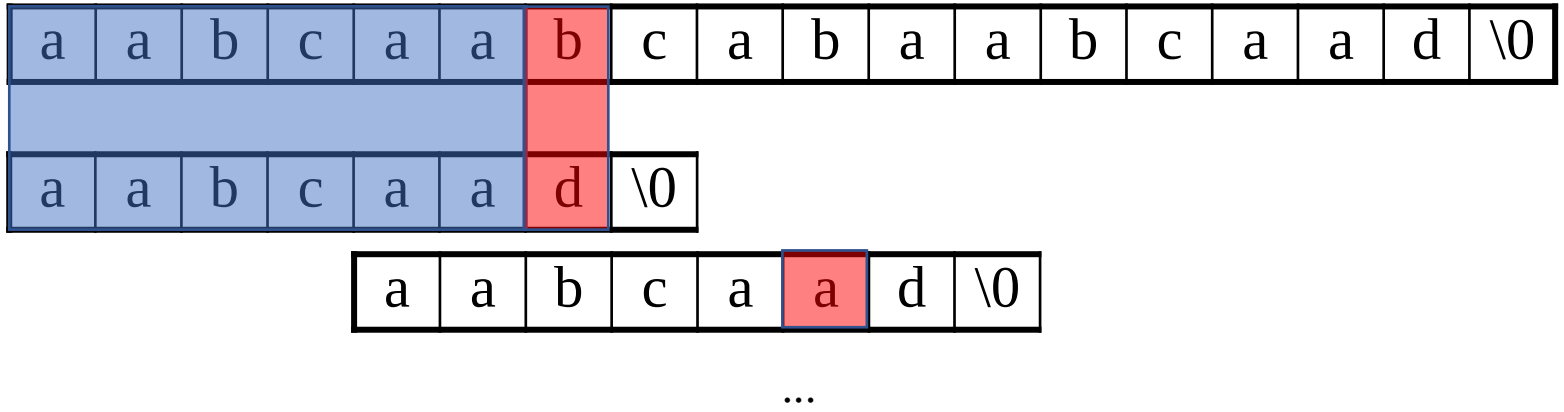
Example



→ O(length of string · length of pattern))

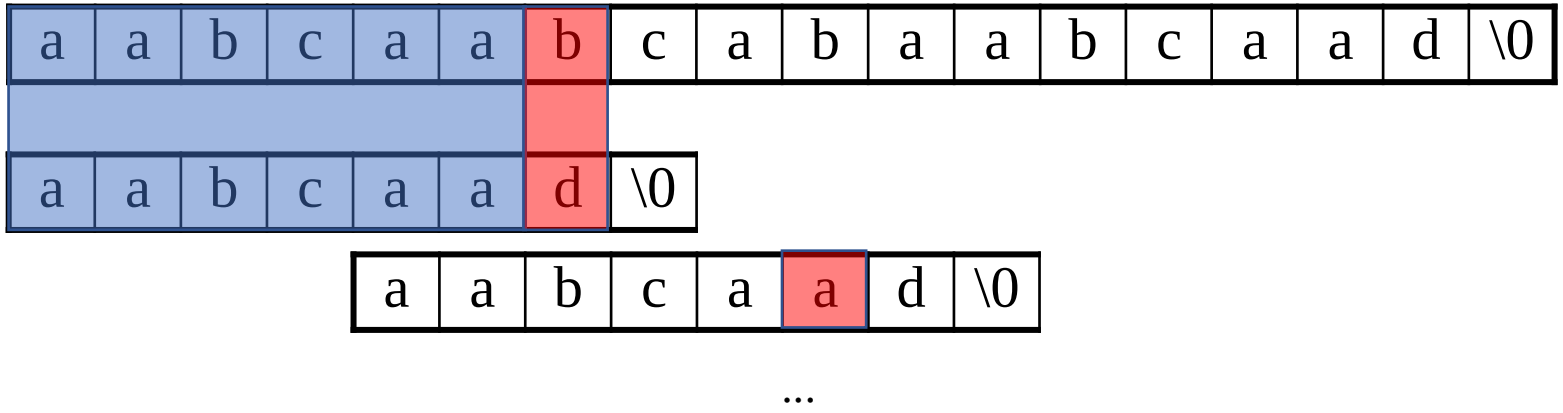
Naïve approach





Knuth, Morris, and Pratt algorithm

- Key idea: Make use of previous match information!



Knuth, Morris, and Pratt algorithm

- Example:
  - pat = 'a b c a b c a c a b'
  - $s = s_0 s_1 \ldots s_{m-1}$
  - determining whether or not there is a match beginning at s_i
- If $s_i \neq$ a then, we may proceed by comparing s_{i+1} and a
- If s_i = a, and $s_{i+1} \neq$ b then we may proceed by comparing s_{i+1} and a
- If $s_i s_{i+1}$ = ab and $s_{i+2} \neq$ c then we may continue the search for a match by comparing the first character in pat with s_{i+2}
- If we have the situation where $s_i s_{i+1} s_{i+2} s_{i+3}$ = abca and $s_{i+4} \neq$ b:

```
        s = '~ a b c a ? ? . . . ?'
pat =     'a b c a b c a c a b'
```

  - the search for a match can proceed by comparing s_{i+4} and the second character in pat, b

Knuth, Morris, and Pratt algorithm

❖정의

- 임의의 패턴 $p=p_0p_1...p_{n-1}$ 이 있을 때 이 패턴의 실패함수 (*failure function*) f는 다음과 같이 정의된다.

$$f(j) = \begin{cases} \text{제일 큰 } i < j, \text{ 여기서 } p_0p_1 \dots p_i = p_{j-i}p_{j-i+1} \dots p_j \text{ 인 } i \geq 0 \text{가 존재 시} \\ -1, \text{ 그 외의 경우} \end{cases}$$

- Example

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

❖ 규칙

- *만일 $s_{i-j}...s_{i-1}=p_0p_1...p_{j-1}$ 이고 $s_i \neq p_j$ 인 부분 매치가 일어난다면, $j \neq 0$ 일 때 s_i 와 $p_{f(j-1)+1}$ 을 비교하는 것으로 매칭을 재개할 수 있다.*
- *$j=0$인 경우에는 s_{i+1} 와 p_0*을 비교하는 것으로 매칭을 재개할 수 있다.

Failure function - Example

- Pattern : abcabcacab (n = 10) $p=p_0p_1 \ldots p_9$, p_0=a, p_9 =b

•$f(j) = \begin{cases} \text{제일 큰 } i < j, \text{ 여기서 } p_0p_1 \ldots p_i = p_{j-i}p_{j-i+1} \ldots p_j \text{ 인 } i \geq 0\text{가 존재 시} \\ -1, \text{ 그 외의 경우} \end{cases}$

j	$f(j)$	Partial string	$f(j)$ *results*
0		a	-1
1	$p_0=p_1\ (i=0)$	ab	-1
2	$p_0 = p_2\ (i=0)$ $p_0p_1 = p_1p_2\ (i=1)$	abc	-1
3	$p_0 = p_3\ (i=0)$ $p_0p_1 = p_2p_3\ (i=1)$ $p_0p_1p_2 = p_1p_2p_3\ (i=2)$	abca	0
4	$p_0 = p_4\ (i=0)$ $p_0p_1 = p_3p_4\ (i=1)$ $p_0p_1p_2 = p_2p_3p_4\ (i=2)$ $p_0p_1p_2p_3 = p_1p_2p_3p_4\ (i=3)$	abcab	1

Failure function - Example

- Pattern : abcabcacab (n = 10) $p=p_0p_1 \ldots p_9$, p_0=a, p_9 =b

•$f(j) = \begin{cases} \text{제일 큰 } i < j, \text{ 여기서 } p_0p_1 \ldots p_i = p_{j-i}p_{j-i+1} \ldots p_j \text{ 인 } i \geq 0\text{가 존재 시} \\ -1, \text{ 그 외의 경우} \end{cases}$

j	$f(j)$	Partial string	$f(j)$ *results*
5	$p_0 = p_5$ $(i = 0)$ $p_0p_1 = p_4p_5$ $(i = 1)$ $p_0p_1p_2 = p_3p_4p_5$ $(i = 2)$ $p_0p_1p_2p_3 = p_2p_3p_4p_5$ $(i = 3)$ $p_0p_1p_2p_3p_4 = p_1p_2p_3p_4p_5$ $(i = 4)$	abcabc	2
6	$p_0 = p_6$ $(i = 0)$ $p_0p_1 = p_5p_6$ $(i = 1)$ $p_0p_1p_2 = p_4p_5p_6$ $(i = 2)$ $p_0p_1p_2p_3 = p_3p_4p_5p_6$ $(i = 3)$ $p_0p_1p_2p_3p_4 = p_2p_3p_4p_5p_6$ $(i = 4)$ $p_0p_1p_2p_3p_4p_5 = p_1p_2p_3p_4p_5p_6$ $(i = 5)$	abcabca	3

Failure function - Example

- Pattern : abcabcacab (n = 10) $p=p_0p_1 \ldots p_9$, p_0=a, p_9 =b

•$f(j) = \begin{cases} \text{제일 큰 } i < j, \text{ 여기서 } p_0p_1 \ldots p_i = p_{j-i}p_{j-i+1} \ldots p_j \text{ 인 } i \geq 0\text{가 존재 시} \\ -1, \text{ 그 외의 경우} \end{cases}$

j	$f(j)$	Partial string	$f(j)$ *results*
7	$p_0 = p_7 (i = 0)$ $p_0p_1 = p_6p_7 (i = 1)$ $p_0p_1p_2 = p_5p_6p_7 (i = 2)$ $p_0p_1p_2p_3 = p_4p_5p_6p_7 (i = 3)$ $p_0p_1p_2p_3p_4 = p_3p_4p_5p_6p_7 (i = 4)$ $p_0p_1p_2p_3p_4p_5 = p_2p_3p_4p_5p_6p_7 (i = 5)$ $p_0p_1p_2p_3p_4p_5p_6 = p_1p_2p_3p_4p_5p_6p_7 (i = 6)$	abcabcac	-1
8	$p_0 = p_8 (i = 0)$ $p_0p_1 = p_7p_8 (i = 1)$ $p_0p_1p_2 = p_6p_7p_8 (i = 2)$ $p_0p_1p_2p_3 = p_5p_6p_7p_8 (i = 3)$ $p_0p_1p_2p_3p_4 = p_4p_5p_6p_7p_8 (i = 4)$ $p_0p_1p_2p_3p_4p_5 = p_3p_4p_5p_6p_7p_8 (i = 5)$ $p_0p_1p_2p_3p_4p_5p_6 = p_2p_3p_4p_5p_6p_7p_8 (i = 6)$ $p_0p_1p_2p_3p_4p_5p_6p_7 = p_1p_2p_3p_4p_5p_6p_7p_8 (i = 7)$	abcabcaca	0

Failure function - Example

- Pattern : abcabcacab (n = 10) $p=p_0p_1 \ldots p_9$, p_0=a, p_9 =b

•$f(j) = \begin{cases} \text{제일 큰 } i < j, \text{ 여기서 } p_0p_1 \ldots p_i = p_{j-i}p_{j-i+1} \ldots p_j \text{ 인 } i \geq 0\text{가 존재 시} \\ -1, \text{ 그 외의 경우} \end{cases}$

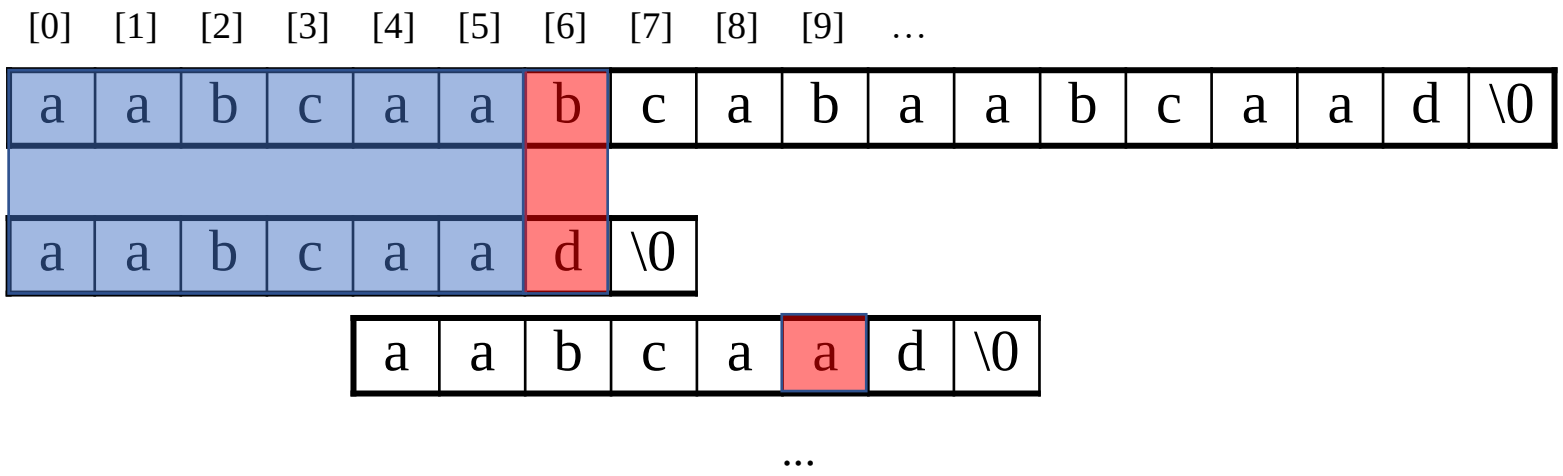
j	$f(j)$	Partial string	$f(j)$ *results*
9	$p_0 = p_9$ $(i = 0)$ $p_0p_1 = p_8p_9$ $(i = 1)$ $p_0p_1p_2 = p_7p_8p_9$ $(i = 2)$ $p_0p_1p_2p_3 = p_6p_7p_8p_9$ $(i = 3)$ $p_0p_1p_2p_3p_4 = p_5p_6p_7p_8p_9$ $(i = 4)$ $p_0p_1p_2p_3p_4p_5 = p_4p_5p_6p_7p_8p_9$ $(i = 5)$ $p_0p_1p_2p_3p_4p_5p_6 = p_3p_4p_5p_6p_7p_8p_9$ $(i = 6)$ $p_0p_1p_2p_3p_4p_5p_6p_7 = p_2p_3p_4p_5p_6p_7p_8p_9$ $(i = 7)$ $p_0p_1p_2p_3p_4p_5p_6p_7p_8 = p_1p_2p_3p_4p_5p_6p_7p_8p_9$ $(i = 8)$	abcabcacab	1

Knuth, Morris, and Pratt algorithm

```
int pmatch( char *string, char *pat )
{       /* Knuth, Morris, Pratt string matching algorithm */
        int i=0, j=0;
        int lens = strlen( string );
        int lenp = strlen( pat );
        while( i<lens && j<lenp ) {
                if( string[i] == pat[j] ) {
                        i++;
                        j++;
                }
                else if( j==0 ) i++;
                else    j=failure[j-1]+1;
        }
        return ( (j==lenp) ? (i-lenp) : -1 );
}
```

```
void fail( char *pat )
{       /* compute the pattern's failure function */
        int n=strlen( pat );
        failure[0] = -1;
        for( j=1; j<n; j++ ) {
                i=failure[j-1];
                while( (pat[j] != pat[i+1]) && (i >= 0) )
                        i = failure[i];
                if( pat[j] == pat[i+1] )
                        failure[j] = i+1;
                else
                        failure[j] = -1;
        }
}
```

KMP algorithm



Failure function

[0]	[1]	[2]	[3]	[4]	[5]	[6]
a	a	b	c	a	a	d
-1	0	-1	-1	0	1	-1

```
while( i<lens && j<lenp ) {
                if( string[i] == pat[j] ) {
                        i++;
                        j++;
                }
                else if( j==0 ) i++;
                else    j=failure[j-1]+1;
        }
```

When i=6 and j=6,
string[6] ! = pat[6] (b!=d)
j=f[6-1]+1;
=f[5]+1 = 1+1 = 2;

i=6,j=2
Compare string[6] and pat[2] (b =b) and continue

Next Topic

Stacks and Queues