Chap 2. Arrays and Structures

2.1 Arrays

2.2 Dynamically Allocated Arrays

2.3 Structures and Unions

2.4 Polynomials

2.5 Sparse Matrices

2.6 Representation of Multidimensional Arrays

2.7 Strings

2.1 Arrays

Arrays are collections of data of the same type.

ADT *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 $A \in Array$, $i \in index$, $x \in item$, j, $size \in$ 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** ($i \in index$) **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

ADT 2.1: Abstract Data Type Array

2.1.2 Arrays in C

- A one-dimensional array in C.

```
int list[5];          // five integers
int *plist[5];        // five pointers to integers
```

- In C all arrays start at index 0.
  list[0], list[1], list[2], list[3], and list[4] (abbreviated list[0:4])
  plist[0:4].

- the compiler allocates five **consecutive memory locations**. Each memory location is large enough to hold a single integer.

- The address of the first element list[0], is called the **base address**.

- the memory address of *list*[*i*] is ***a + i*sizeof(int)*, where *a* is the base address**.

Observe that there is a difference between a declaration such as

```
int *listl;
```

and

```
int list2[5];
```

The variables *listl* and *list2* are both pointers to an **int**, but in the second case five memory locations for holding integers have been reserved. list2 is a pointer to list2[0] and list2+i is a pointer to list2[i].

Notice that in C, we do not multiply the offset *i* with the size of the type to get to the appropriate element of the array. Thus, regardless of the type of the array *list2*, it is always the case that (*list2* + *i*) equals &list2[i]. So, *(*list2* + *i*) equals list2[i].

```
#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);
  printf("The sum is: %f\n", answer);
}

float sum(float list[], int n)
{
  int i;
  float tempsum = 0;
  for (i = 0; i < n; i++)
            tempsum += list[i];
  return tempsum;
}
```

Program 2.1: Example array program

- Upon Invocation:

  Copy &input[0] into the parameter *list*.

- *Dereference*:
  - list[i] is on the right of "="

  return the value to which (list + i) points.
  - list[i] is on the left of "="

  Store the value into the location to which (list + i) points.

- **Example 2.1 [One-dimensional Addressing]**

int one[] = {0, 1, 2, 3, 4};
print1(&one[0], 5);

```
void printl(int *ptr, int rows)
{
  /* 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");
}
```

Program 2.2: One-dimensional array accessed by address

Address	Contents
12244868	0
12344872	1
12344876	2
12344880	3
12344884	4

%d : 10진수로 출력 정수형
%f : 실수형
%e : 지수형
%o : 8진수로 출력
%x : 16진수로 출력
%u : 부호없는 10진수로 출력
%g : 실수형으로 자동 출력
%p : 포인터의 주소를 출력
%c : 하나의 문자로 출력 문자형
%s : 문자열을 출력

Assumption: sizeof(int) == 4

2.2 Dynamically Allocated Arrays

C provides three memory allocation functions – ***malloc***, ***calloc*** and ***realloc*** - that are useful in the context of dynamically allocated arrays.

(1) malloc

Since *malloc* may be invoked from several places in your program, it is often convenient to define a macro that invokes *malloc* and exits when *malloc* fails. A possible macro definition is:

```
#define MALLOC(p,s) \
        if (! ( (p) = malloc(s) ) ) { \
                fprintf(stderr, "Insufficient memory"); \
                exit(EXIT_FAILURE);\
        }
```

Now,

```
pf = (float*) malloc(sizeof(float));
```

may be replaced by the code

```
MALLOC(pf, sizeof(float));
```

(2) calloc

```
int *x;
x = calloc(n, sizeof(int)); /* allocated bits are set to 0 */
```

could be used to define a one-dimensional array of integers; the capacity of this array is *n*, and **x[0:n-1]** are initially 0. As was the case with *malloc*, it is useful to define the macro CALLOC as below and use this macro to write clean robust programs.

```
#define CALLOC(p,n,s)\
        if (! ( (p) = calloc(n,s))) {\
                fprintf(stderr, "Insufficient memory"); \
                exit(EXIT_FAILURE);\
        }
```

(3) realloc

The function *realloc* resizes memory previously allocated by either malloc or calloc. For example, the statement

```
realloc(p, s);
```

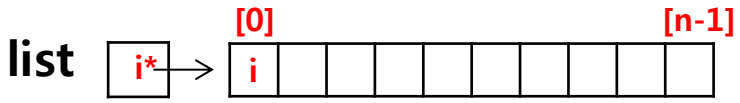
changes the size of the memory block pointed at by *p* to *s*. The contents of the first min{*s*, oldSize} bytes of the block are unchanged as a result of this resizing. When *s* > oldSize the additional *s* - oldSize have an unspecified value and when *s* < oldSize, the rightmost olsSize - *s* bytes of the old block are freed. When realloc is able to do the resizing, it returns a pointer to the start of the new block and when it is unable to do the resizing, the old block is unchanged and the function returns the value NULL.

As with *malloc* and *calloc*, it is useful to define a macro REALLOC as below.

```
#define REALLOC(p,s)\
        if (! ( (p) = realloc (p, s))) {\
                fprintf(stderr, "Insufficient memory");\
                exit(EXIT_FAILURE);\
        }
```

2.2.1 One-dimensional Arrays

```
int i, n,*list;
printf("Enter the number of numbers to generate: ");
scanf ( "%d", &n);
if( n < 1 ) {
        fprintf(stderr, "Improper value of n\n");
        exit(EXIT_FAILURE);
}
MALLOC(list, n * sizeof(int));
```



※ i stands for int

```
list = (int*) malloc(n*sizeof(int));
```

2.2.2 Two-Dimensional Arrays

C uses the so-called **array-of-arrays representation** to represent a multidimensional array. In this representation, a two-dimensional array is represented as a one-dimensional array in which each element is, itself, a one-dimensional array.

To represent the two-dimensional array

```
int x[3][5];
```

we actually create a one-dimensional array *x* whose length is 3; each element of *x* is a one-dimensional array whose length is 5.

Figure 2.2 shows the memory structure. Four separate memory blocks are used. One block (the lightly shaded block) is large enough for three pointers and each of the remaining blocks is large enough for 5 **int**s.

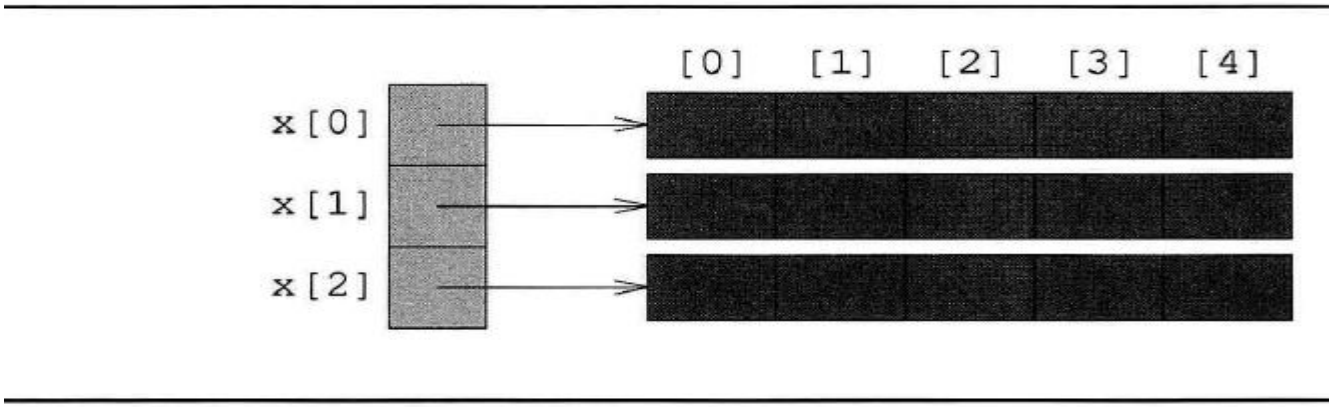


Figure 2.2: Array-of-arrays representation

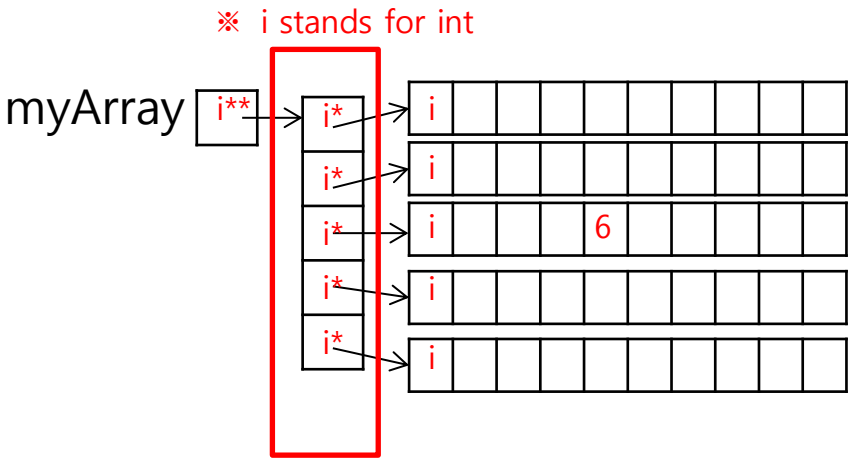
C finds the element *x*[*i*][*j*] by first accessing the pointer in *x*[*i*]. This pointer gives us the address, in memory, of the zeroth element of row *i* of the array. Then by adding *j**sizeof(int) to this pointer, the address of the [*j*]th element of row *i* (i.e., element x[i][j]) is determined. Program 2.3 gives a function that creates a two-dimensional array at run time.

A three-dimensional array is represented as a one-dimensional array, each of whose elements is a two-dimensional array. Each of these two-dimensional arrays is represented as shown in Figure 2.2.

```
int** make2dArray(int rows, int cols)
{
        /* create a two dimensional rows x 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;
}
```

Program 2.3: Dynamically create a two-dimensional array

```
void main()
{
   int **myArray;
   myArray = make2dArray(5,10);
   myArray[2][4] = 6;
   printf("%d\n", myArray[2][4]);
}
```



2.3 Structures and Union

2.3.1 Structures

- A **structure** (called a ***record*** in many other languages) is a collection of data items, where each **item** is identified as to its type and name.

```
struct person {
        char name[10];
        int age;
        float salary;
};
struct person person1;
```

```
struct person {
        char name[10];
        int age;
        float salary;
};
typedef struct person humanBeing;
humanBeing  person1;
```

```
struct {
    char name[10];
    int age;
    float salary;
} person1;
```

– Structure member operator : dot(.)

← **person1 is a variable.**

```
strcpy(person1.name,"james");
person1.age = 10;
person1.salary = 35000;
```

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

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

← humanBeing is a data type.

```
humanBeing person1, person2;

if (strcmp(personl.name, person2.name))
        printf("The two people do not have the same name\n");
else
        printf("The two people have the same name\n");
```

It would be nice if we could write if (person1 == person2) and have the entire structure checked for equality, or if we could write person1 = person2 and have that mean that the value of every field of the structure of *person2* is assigned as the value of the corresponding field of *personl*.

- ANSI C permits **structure assignment**, but most earlier versions of C do not. For older versions of C, we are forced to write the more detailed form:

```
strcpy(person1.name, person2.name);
person1.age = person2.age;
person1.salary = person2.salary;
```

- While **structures cannot be directly checked for equality or inequality**, we can write a function (Program 2.4) to do this.

A typical function call might be:

```
if (humansEqual(person1,person2))
        printf("The two human beings are the same\n");
else
        printf("The two human beings are not the same\n");
```

```
#define FALSE 0
#define TRUE 1
int humansEqual(humanBeing person1, humanBeing person2)
{
        /* return TRUE if person1 and person2 are the same human being
                otherwise return FALSE */
        if (strcmp(person1.name, person2.name)) return FALSE;
        if (person1.age != person2.age) return FALSE;
        if (person1.salary != person2.salary) return FALSE;
        return TRUE;
}
```

Program 2.4: Function to check equality of structures

- A structure within a structure

```
typedef struct {
        int month;
        int day;
        int year;
} date;
typedef struct {
        char name [10];
        int age;
        float salary;
        date dob;              // day of birth
} humanBeing;

humanBeing person1;
person1.dob.month = 2;
person1.dob.day = 11;
person1.dob.year = 1944;
```

2.3.2 Unions

- A union declaration is similar to a structure, but the fields of a union must share their memory space.
- Only one filed is “active” at any given time.

```
typedef struct {
    enum tagField {female, male} sex;
    union {
            int children;
            int beard
    } u;
} sexType;
typedef struct {
    char name[10];
    int age;
    float salary;
    date dob;
    sexType sexInfo;
} humanBeing;

humanBeing personl, person2;

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

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

2.3.3 Internal Implementation Of Structures

In most cases you need not be concerned with exactly how the C compiler will store the fields of a structure in memory. Generally, if you have a structure definition such as:

struct {int i, j; float a, b;};

or

struct {int i; int j; float a; float b; };

these values will be stored in the same way using increasing address locations in the order specified in the structure definition. However, it is important to realize that **holes** or **padding** may actually occur within a structure to permit two consecutive components to be properly **aligned** within memory.

The size of an object of a **struct** or **union** type is the amount of storage necessary to represent the largest component, including any padding that may be required. Structures must begin and end on the same type of memory boundary, for example, **an even byte boundary** or an address that is **a multiple of 4, 8, or 16.**

```
#include <stdio.h>

struct test1 {int i, j; float a, b;};
struct test2 {int i; int j; float a; float b; };
struct test3 {int i; char j; float a, b;};
struct test4 {int i; short j; float a; float b; };
struct test5 {int i; char j; char k; float a, b;};
struct test6 {int i; short j; short k; float a; float b; };

void main(void)
{
  printf(" %d %d\n",sizeof(struct test1), sizeof(struct test2));
  printf(" %d %d\n",sizeof(struct test3), sizeof(struct test4));
  printf(" %d %d\n",sizeof(struct test5), sizeof(struct test6));
}
```

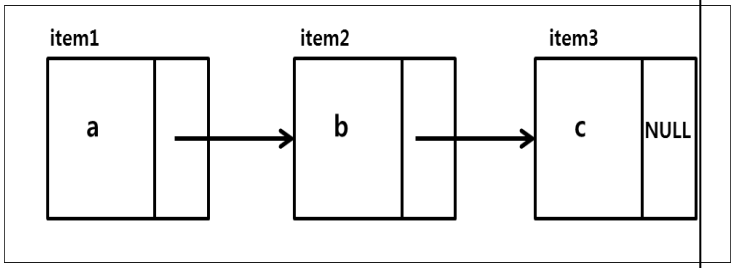
```
16 16
16 16
16 16
```

2.3.4 Self-Referential Structures

A ***self-referential structure*** is one in which one or more of its components is a pointer to itself.

```
typedef struct list {
        char data;
        struct list *link ;
} node;
node iteml, item2, item3;
iteml.data =  'a'; iteml.link = NULL;
item2.data = 'b'; item2.link = NULL;
item3.data = 'c'; item3.link = NULL;
iteml.link = &item2;
item2.link = &item3;
```

```
In <stdio.h> of Microsoft visual studio 10,
/* Define NULL pointer value */
#ifndef NULL
#ifdef __cplusplus
#define NULL    0
#else
#define NULL    ((void *)0)
#endif
#endif
```



2.4 Polynomials

the ***ordered*** or ***linear list***.

many examples of this data structure:

- **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)**
- **Floors of a building: (basement, lobby, mezzanine, first, second)**
- **Years the United States fought in World War II: (1941, 1942, 1943, 1944, 1945)**
- **Years Switzerland fought in World War II: ( ) ← an empty list**

Some operations on lists

- Find the length, *n*, of a list.
- Read the items in a list from left to right (or right to left).
- Retrieve the ith item from a list, $0 \leq i < n$.
- Replace the item in the ith position of a list, $0 \leq i < n$.
- Insert a new item in the ith position of a list, $0 \leq i \leq n$. The items previously numbered i, i+l, $\cdots$,n-1 become items numbered i+l,i+2, $\cdots$,n.
- Delete an item from the ith position of a list, $0 \leq i < n$. The items numbered i+1, $\cdots$,n-1 become items numbered i, i+l, $\cdots$, n-2.

a sequential mapping of ordered lists: an array

assuming the standard implementation of an array

- the list element, $item_i$, with the array index i.
- store $item_i$, $item_{i+1}$ into consecutive slots i and i +1 of the array.
- we can retrieve an item, replace an item, or find the length of a list, in constant time.
- we can read the items in the list, from either direction.
- insertion and deletion pose problems since the sequential allocation forces us to move items so that the sequential mapping is preserved.
- We consider **nonsequential mappings of ordered lists** in Chapter 4.

Manipulation of **symbolic polynomials**

$$A(x) = 3x^{20} + 2x^5 + 4\ ,\ \ B(x) = x^4 + 10x^3 + 3x^2 + 1$$

The largest (or leading) exponent of a polynomial is called its ***degree***. There are standard mathematical definitions for the sum and product of polynomials. Assume that we have two polynomials, $A(x) = \Sigma\, a_i x^i$ and $B(x) = \Sigma\, b_i x^i$ *then :*

$$A(x) + B(x) = \Sigma\,(a_i + b_i)x^i$$
$$A(x)\,B(x) = \Sigma\,(a_i\, x^i\ \Sigma\,(b_j\, x^j))$$

ADT *Polynomial* is

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

functions:

for all *poly*, *poly*1, *poly*2 ∈ *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*
Polynomial Attach(*poly*, *coef*, *expon*)	::=	**if** (*expon* ∈ *poly*) **return** error **else return** the polynomial *poly* with the term <*coef*, *expon*> inserted
Polynomial Remove(*poly*, *expon*)	::=	**if** (*expon* ∈ *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(*poly*1, *poly*2)	::=	**return** the polynomial *poly*1 + *poly*2
Polynomial Mult(*poly*1, *poly*2)	::=	**return** the polynomial *poly*1 · *poly*2

end *Polynomial*

ADT 2.2: Abstract data type *Polynomial*

2.4.2 Polynomial Representation

Representation of polynomials in C

1) One way to represent polynomials in C

```
#define MAX_DEGREE 101 /*Max degree of polynomial+1*/
typedef struct {
        int degree;
        float coef[MAX_DEGREE];
        } polynomial;
```

Now if *a* is of type *polynomial* and $n < MAX\_DEGREE$, the polynomial $A(x) = \Sigma_{i=0}^{n} a_i x^i$ would be represented as :

```
a.degree = n
a.coef[i] = a_i, 0 ≤ i ≤ n
```

Although this representation leads to very simple algorithms for most of the operations, it wastes a lot of space.

representation of polynomials in C(cont')

2) an alternate representation.

```
#define MAX_TERMS 100 /*size of terms array*/
typedef struct {
        float coef;
        int expon;
        } polynomial;
polynomial terms[MAX_TERMS];
int avail = 0;
```

Consider the two polynomials

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

	startA ↓	*finishA* ↓	*startB* ↓			*finishB* ↓	*avail* ↓
coef	2	1	1	10	3	1	
exp	1000	0	4	3	2	0	
	0	1	2	3	4	5	6

2.4.3 Polynomial Addition

Write a C function that adds two polynomials, *A* and *B*, represented as above to obtain *D =A + B*.

To produce *D(x)*, *padd* (Program 2.6) adds *A(x)* and *B(x)* term by term. Starting at position *avail*, *attach* (Program 2.7) places the terms of *D* into the array, *terms*. If there is not enough space in *terms* to accommodate *D*, an error message is printed to the standard error device and we exit the program with an error condition.

```
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++;
        }
    }
    /* 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;
}
```

Program 2.6: Function to add two polynomials

```
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;
}
```

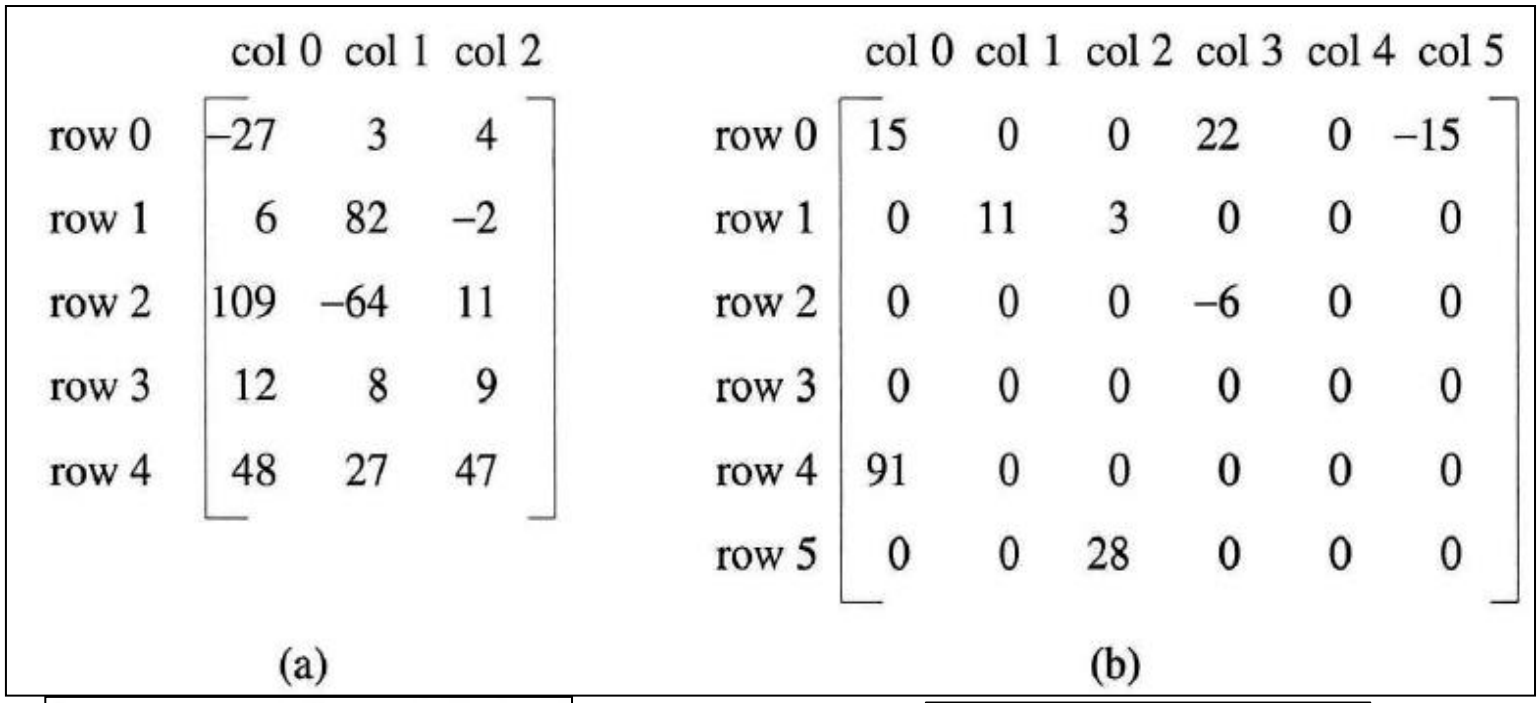
Program 2.7: Function to add a new term

Analysis of the algorithm *padd*:
Let m and n be the number of nonzero terms in A and B, respectively.
The asymptotic computing time of this algorithm is O(n +m).

2.5 SPARSE MATRICES

In general, we write **m x n (read "m by n")** to designate a matrix with *m* rows and *n* columns. If *m* equals *n*, the matrix is **square**.

A two-dimensional array defined as *a*[MAX_ROWS][MAX_COLS].



	col 0	col 1	col 2
row 0	−27	3	4
row 1	6	82	−2
row 2	109	−64	11
row 3	12	8	9
row 4	48	27	47

(a)

	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

(b)

Figure 2.4: Two matrices

a ***sparse matrix***

희소 행렬[sparse matrix] 행렬의 원소에 비교적 0이 많은 행렬.

Sparse matrix : $m \times n$ matrix A

such that $\frac{\text{no. of nonzero elements}}{m \times n} << 1$

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* × *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*

2.5.2 Sparse Matrix Representation

- Use *an array of triples :* <row, column, value>

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

```
#define MAX_TERMS 101 /* maximum number of terms +1*/
typedef struct{
    int row;
    int col;
    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

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***).

2.5.3 Transposing a Matrix

Figure 2.5(b) shows the transpose of the sample matrix. To transpose a matrix we must interchange the rows and columns. This means that each element a[i][j] in the original matrix becomes element b[j][i] in the transposed 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

(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

transposed matrix((수학) **전치행렬(轉置行列)**(자리바꿈행렬))

Since we have organized the original matrix by rows, we might think that the following is a good algorithm for transposing a matrix:

for each row i

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
   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)
(1, 2, 3)	→	(2, 1, 3)
(5, 2, 28)	→	(2, 5, 28)
	...	

We can avoid data movement!

This algorithm is incorporated in *transpose* (Program 2.8). The first array, *a*, is the original array, while the second array, *b*, holds the transpose.

```
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 columns in a */
            for (j = 1; j <= n; j++) {
                /* find elements from the current column */
                if (a[j].col == i) {
                    /* the element is in the 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

Nested for loops are the decisive factor.
The remaining part requires only constant time.
Time complexity : **O(*columns* · *elements*)**

만약 elements = rows · columns 이라면,
O(*columns* · *elements*) = O(*columns*^{2} · *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

calculation of rowTerms

	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

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

calculation of startingPos

```
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;
        }
    }
}
```

Program 2.9: Fast transpose of a sparse matrix

Time complexity : O(*columns + elements*)

만약, ***elements = columns · rows*** 라면, ***O(columns + elements) = O(columns · rows)***가 됨

2.5.4 Matrix Multiplication

Definition: Given A and B where A is m x n and B is n x p, the product matrix D has dimension m x p. Its $<i, j>$ element is :

$$d_{ij} = \sum_{k=0}^{n-1} a_{ik}\, b_{kj}$$

for $0 \le i < m$ and $0 \le j < p$. □

We can multiply matrices using **the standard array representation**. The classic multiplication algorithm is:

```
for (i = 0; i < rowsA; i++) {
        for (j = 0; j < colsB; j++) {
                sum = 0;
                for (k = 0; k < colsA; k++)
                        sum+= (a[i][k] * b[k][j]);
                d[i][j] = sum;
        }
}
```

This algorithm takes **O(rowsA · colsA · colsB)** time.

The product of two sparse matrices may no longer be sparse, as Figure 2.6 shows.

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Figure 2.6: Multiplication of two sparse matrices

multiply two sparse matrices represented as an ordered list (Figure 2.5).

- We need to compute the elements of *D* by rows so that we can store them in their proper place without moving previously computed elements.
- To do this we pick a row of *A* and find all elements in column *j* of *B* for j = 0, 1, · · · , *colsB* - 1.
- Normally, we would have to scan all of *B* to find all the elements in column *j*.
- However, we can avoid this by first computing the **transpose of *B***. This puts all column elements in consecutive order.
- Once we have located the elements of row *i* of *A* and column *j* of *B* we just do a merge operation similar to that used in the polynomial addition of Section 2.2.

	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

multiply

A	B	newB = B^T
a b c d	x y z	x m v u
e f g h	m n o	y n w k
p q r s	v w t	z o t j
	u k j	

1	0 0 a	0 0 x	1 0 0 x
2	0 1 b	0 1 y	2 0 1 m
3	0 2 c	0 2 z	3 0 2 v
4	0 3 d	1 0 m	4 0 3 u
5	1 0 e	…	5 1 0 y
6	1 1 f		6 1 1 n
7	1 2 g		7 1 2 w
8	1 3 h		8 1 3 k
9	2 0 p	3 0 u	9 2 0 z
10	2 1 q	3 1 k	10 2 1 o
11	2 2 r	3 2 j	11 2 2 t
12	2 3 s		12 2 3 j

D

A[1:4] ↔ newB[1:4]
A[5:8] ↔ newB[1:4]
A[9:12] ↔ newB[1:4]

ax + bm + cv + du
ex + fn + gw + hk
px + qn + rw + sk

A[1:4] ↔ newB[5:8]
A[5:8] ↔ newB[5:8]
A[9:12] ↔ newB[5:8]

ay + bn + cw + dk
ey + fn + gw + hk
py + qn + rw + sk

A[1:4] ↔ newB[9:12]
A[5:8] ↔ newB[9:12]
A[9:12] ↔ newB[9:12]

az + bo + ct + sj
ez + fo + gt + hi
pz + qo + rt + sj

- To obtain the product matrix *D*, *mmult* (Program 2.10) multiplies matrices *A* and *B* using the strategy outlined above.
- We store the matrices *A*, *B*, and *D* in the arrays *a*, *b*, and *d*, respectively.
- To place a triple in *d* and to reset *sum* to 0, *mmult* uses *storeSum* (Program 2.11).
- In addition, *mmult* uses several local variables that we will describe briefly.
- The variable *row* is the row of *A* that we are currently multiplying with the columns in *B*.
- The variable *rowBegin* is the position in *a* of the first element of the current row.
- The variable *column* is the column of *B* that we are currently multiplying with a row in *A*.
- The variable *totalD* is the current number of elements in the product matrix *D*.
- The variables *i* and *j* are used to examine successively elements from a row of *A* and a column of *B*.
- Finally, the variable *newB* is the sparse matrix that is the transpose of *b*.
- Notice that we have introduced an additional term into both *a* (*a*[*totalA*+l].row = *rowsA*;) and *newB* (*newB*[*totalB*+l].row = *colsB*;). These dummy terms serve as sentinels that enable us to obtain an elegant algorithm.

```
void mmult(term a[], term b[], term d[])
{/* multiply two sparse matrices */
    int i, j, d_column, totalB = b[0].value, totalD = 0;
    int rowsA = a[0].row, colsA = a[0].col, totalA = a[0].value;
    int colsB = b[0].col;
    int rowBegin = 1, d_row= a[1].row, sum= 0;
    term newB[MAX_TERMS];
    if (colsA != b[0].row) {
        fprintf(stderr,"Incompatible matrices\n");
        exit(EXIT_FAILURE);
    }
    fastTranspose(b,newB);
    /* set boundary condition */
    a[totalA+1].row= rowsA;
    newB[totalB+1].row= colsB;
    newB[totalB+1].col = 0;
    for (i = 1; i <= totalA; ) {
        d_column= newB[1].row;
        for (j = 1; j <= totalB+1;) {
            /* multiply row of a by column of b */
            if (a[i].row != d_row) {
                storeSum(d,&totalD,d_row,d_column,&sum);
                i = rowBegin;
                for (; newB[j].row == d_column; j++) ;
                d_column = newB[j].row;
            } else if (newB[j].row != d_column) {
                storeSum(d, &totalD, d_row,d_column, &sum);
                i = rowBegin;
                d_column = newB[j].row;
            } else  {
                switch (compare(a[i].col, newB[j].col)) {
                    case -1: /* go to next term in a */
                        i++; break;
                    case 0: /* add terms, go to next term in a and b*/
                        sum+= (a[i++].value* newB[j++].value);
                        break;
                    case 1 : /* advance to next term in b */
                        j++;
                }
            }
        }
        for (; a[i].row == d_row; i++) ;
        rowBegin = i; d_row = a[i].row;
    } /* end of for i<=totalA */
    d[0].row = rowsA; d[0].col = colsB; d[0].value = totalD;
}
```

Program 2.10: Sparse matrix multiplication

totalD는 배열 d에 마지막으로 삽입된 배열 d의 엔트리의 인덱스를 나타낸다. 따라서, 그 값은 0으로 초기화되고 삽입시에 그 값을 1 만큼 증가시킨 후 사용한다.

D[row, column] = SUM (A[row, x] * B[y, column]) if x == y, where 0<= x and y <= n-1.
D[row, column] = SUM (A[row, x] * newB[column, y]) if x == y, where 0<= x and y <= n-1.
행열 A의 현재 행 row,
    행열 a에서,
        rowBegin : 행열 A의 현재 행 row에 해당하는, 엔트리들의 시작 index.
        i : 행열 A의 현재 행 row에 해당하는, 엔트리들의 현재 index.
행열 B의 현재 열 column,
    행열 b에서,
        j : 행열 B의 현재 열 column에 해당하는, 엔트리들의 현재 index.
    행열 newB에서,
        j : 행열 B의 현재 열 column에 해당하는, 엔트리들의 현재 index.

/* 배열 d의 엔트리 d[++*totalD]에 <row, column, *sum>을 삽입하시오 */

```
void storeSum(term d[], int *totalD, int row, int column, int *sum)
{/* if *sum != 0, then it along with its row and column
     position is stored as the *totalD+l entry in d */
    if (*sum) {
        if (*totalD < MAX_TERMS) {
            d[++*totalD].row= row;
            d[*totalD].col = column;
            d[*totalD].value = *sum;
            *sum = 0;
        } else {
            fprintf(stderr,"Numbers of terms in product exceeds %d \n" ,
                                MAX_TERMS) ;
            exit(EXIT_FAILURE);
        }
    }
}
```

Program 2.11: storeSum functionThis algorithm takes O(rowsA · colsA · colsB) time.

```
int compare(int x, int y)
{
   if (x < y) return -1;
   else if (x == y) return 0;
   else return 1;
}
```

Program 1.6: Comparison of two integers

- the time needed to transpose B. $O(colsB + totalB)$.

- The outer for loop is executed *totalA* times.
- At each iteration either i or j or both increase by 1, or i and column are reset. The maximum total increment in j over the entire loop is totalB + 1.
- If *termsRow* is the total number of terms in the current row of A, then i can increase at most termsRow times before i moves to the next row of A. When this happens, we reset i to rowBegin, and, at the same time, advance column to the next column. Thus, this resetting takes place at most colsB time, and the total maximum increment in i is *colsB*termsRow*. Therefore, the maximum number of iterations of the outer for loop is colsB + colsB*termsRow + totalB.
- The time for the inner loop during the multiplication of the current row is $O(colsB*termsRow + totalB)$, and the time to advance to the next row is $O(termsRow)$. Thus, the time for one iteration of the outer for loop is $O(colsB*termsRow + totalB)$.
- The overall time for this loop is:

$$O(\sum_{row} (colsB \cdot termsRow + totalB)) = \mathbf{O(colsB \cdot totalA + rowsA \cdot totalB).}$$

Symbolic polynomial (기호 다항식) 문제 [4 점]

우리는 “2.4 POLYNOMIALS”에서 두 개의 기호 다항식을 읽어 그를 Figure 2.3과 같이 <계수(coefficient), 지수(exponent)> 쌍을 엔트리로 가지는 배열로 표현하고, 그 표현을 기반으로 Program 2.6과 Program 2.7에서 그 두 다항식의 합을 구하는 연산을 공부하였다. 이제 우리는 아래 두 다항식에 대해 연산을 수행하고자 한다.

$$A(x) = 3x^{20} + 2x^5 + 6x^3 + x + 4 ,$$
$$B(x) = 7x^5 + x^4 + 10x^3 + 3x^2 + 1$$

위 두 기호 다항식을 대상으로 다음을 수행하시오.

1. 키보드로 부터 읽어 들인다. 즉, 각 다항식을 readPoly로 읽는다.
2. A(x)와 B(x)를 출력한다. 즉, 각 다항식을 printPoly로 출력한다.
3. A(x) + B(x)를 padd를 호출하여 수행한 후, 결과를 printPoly로 출력한다.
4. A(x) * B(x)를 pmult를 호출하여 수행한 후, 결과를 printPoly로 출력한다.

Sparse Matrix (희소 행렬) 문제 [4 점]

우리는 “2.5 SPARSE MATRIX”에서 두 개의 희소 행렬을 읽어 그를 Figure 2.5 (a)와 같이 <row, col, value>의 엔트리를 가지는 배열로 표현하고, 그 표현을 기반으로 Program 2.10에서 그 두 행렬의 곱을 구하는 연산을 공부하였다. 이제 우리는 아래 6 X 6의 두 희소 행렬 A와 B에 대해 연산을 수행하고자 한다.

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

matrix A

$$\begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}$$

matrix B

```
6 6 12
0 0 15
0 2 28
0 3 22
0 5 -15
1 0 -858993460
1 2 3
1 3 -6
4 0 15
4 3 22
4 5 -15
5 0 -858993460
5 3 -6
```

```
6 6 10
0 0 15
0 2 28
0 3 22
0 5 -15
1 2 3
1 3 -6
4 0 15
4 3 22
4 5 -15
5 3 -6
```

위 두 희소 행렬 A와 B를 대상으로 다음을 수행하시오.

1. 키보드로 부터 읽어 들인다. 즉, 각 행렬을 readMatrix로 읽는다.
2. A와 B를 출력한다. 즉, 각 행렬을 printMatrix로 출력한다.
3. A + B를 madd를 호출하여 수행한 후, 결과를 printMatrix로 출력한다.
4. A * B를 mmult를 호출하여 수행한 후, 결과를 printMatrix로 출력한다.

2.6 Representation of Multidimensional Arrays

In C, multidimensional arrays are represented using **the array-of-arrays representation** (Section 2.2.2).

An alternative to the array-of-arrays representation is to **map all elements of a multidimensional array into an ordered or linear list**. **The linear list is then stored in consecutive memory just as we store a one-dimensional array.** This mapping of a multidimensional array to memory requires a more complex addressing formula that required by the mapping of a one-dimensional array to memory. If an array is declared a [$upper_0$][$upper_1$] · · · [$upper_{n-1}$], then it is easy to see that the number of elements in the array is:

$$\prod_{i=0}^{n-1} upper_i$$

There are two common ways to represent multidimensional arrays: ***row major order*** and ***column major order***. We consider only row major order here, leaving column major order for the exercises.

Row major order example

- A[2][3][2][2] ← A[$upper_0$][$upper_1$] ... [$upper_{n-1}$]
- 2*3*2*2 = 24 elements
- stored as A[0][0][0][0], A[0][0][0][1], ..., A[2][3][2][0], A[2][3][2][1]
- indices are increasing : lexicographic order
- translate to locations in the one-dim array

  we assume that α is the address of A[0][0][0][0]

  · A[0][0][0][0] → position α

  · A[0][0][0][1] → position α +1

  · A[2][3][2][1] → position α +23

  · A[i][j][k][m] → position: α + i*$upper_1$*$upper_2$* $upper_3$
  + j*$upper_2$*$upper_3$
  + k * $upper_3$
  + m

  · A[i_0][0][0]…[0] → α + i_0 $upper_1$ $upper_2$ … $upper_{n-1}$

  · A[i_0][i_1][0]…[0] → α + i_0 $upper_1$ $upper_2$ … $upper_{n-1}$
  + i_1 $upper_2$ $upper_3$ … $upper_{n-1}$

Translation for an n-dim array

- assume $p_i=0$ and $q_i=u_i-1$
- one-dim array $A[u_1]$

array element:	A[0]	A[1]	A[2]	A[i]	$A[u_1-1]$
address:	α	$\alpha+1$	$\alpha+2$	$\alpha+i$	$\alpha+u_1-1$

- two-dim array $A[u_1][u_2]$

  let α be the address of A[0][0]

  $A[i][0] : \alpha + i * u_2$

  $A[i][j] : \alpha + i * u_2 + j$

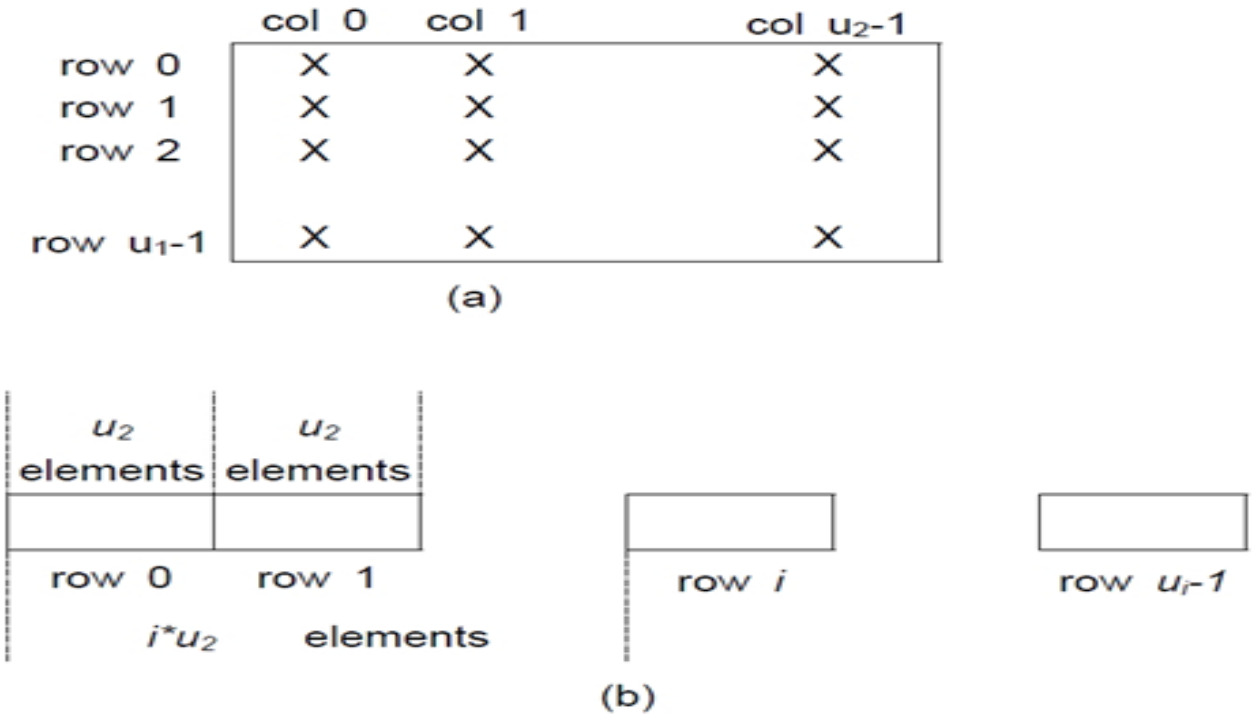
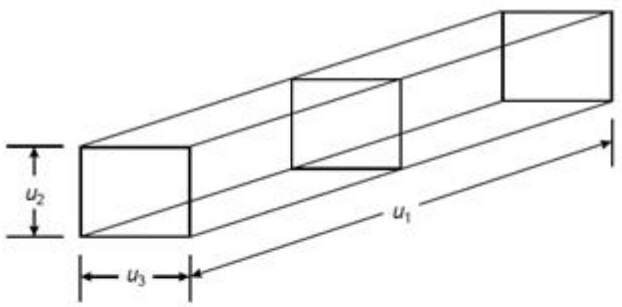
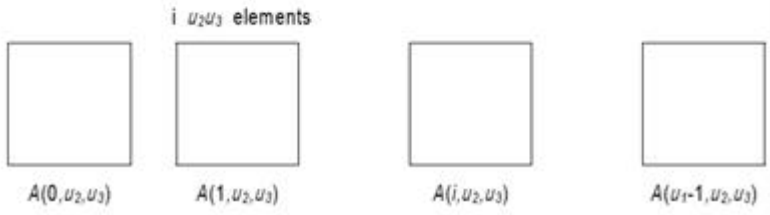


Figure: Sequential representation of one-dimensional array

three-dim array A[u1][u2][u3]
the address of A[0][0][0] : α
A[i][0][0] : α+iu2u3
A[i][j][k] : α+iu2u3+ju3+k



(a) 3-dimensional array A[u_1][u_2][u_3] regarded as u_1 2-dimensional array



(b) Sequential row major representation of a 3-dimensional array. Each 2-dimensional array is represented as in previous figure

Repeating in this way the address for $A[i_0][i_1]\ldots[i_{n-1}]$ is:

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

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

2.7 Strings

2.7.1 the Abstract Data type

As an ADT, we define a string to have the form, $S = s_0, \ldots, s_{n-1}$, where s_i are characters taken from the character set of the programming language. If $n = 0$, then S is an *empty* or *null string* (“”).

There are several useful operations we could specify for strings. We have listed the essential operations in ADT 2.4, which contains our specification of the string ADT. Actually there are many more operations on strings, as we shall see when we look at part of C's string library in Figure 2.8.

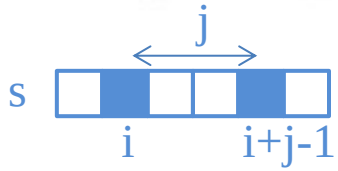
ADT *String* is

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*

2.7.2 Strings in C

In C, we represent **strings** as character arrays terminated with the null character **\0**. For instance, suppose we had the strings:

```
#define MAX_SIZE 100 /*maximum size of string */
char s[MAX_SIZE] = {"dog"};        // = "dog"
char t[MAX_SIZE] = {"house"};      // = "house"
```

Figure 2.9 shows how these strings would be represented internally in memory. Notice that we have included array bounds for the two strings.

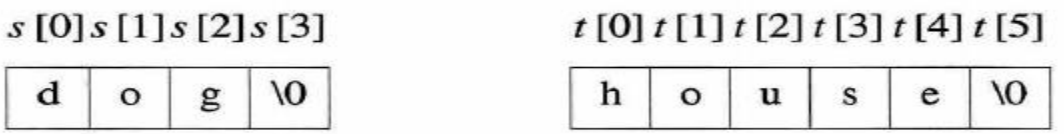


Figure 2.9: String representation in C

〈아래 형태 모두 가능〉

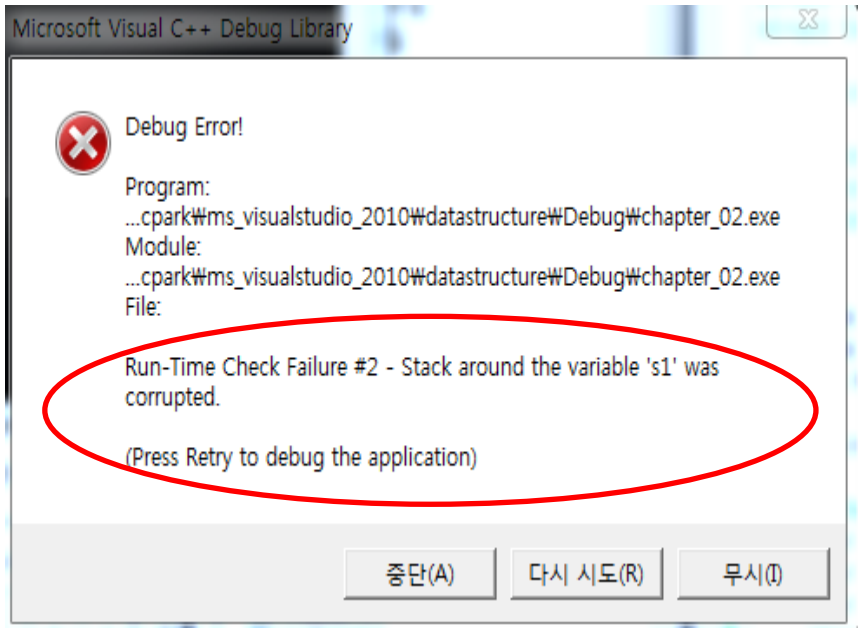
```
char s[MAX_SIZE] = {"dog"};     char s[MAX_SIZE] = "dog";   ← 배열 s의 크기 = MAX_SIZE로 고정됨.
char s[ ] = {"dog"};            char s[ ] = "dog";          ← 배열 s의 크기
char *s = "dog";                                            ←
```

Technically, we could have declared the arrays with the statements:

```
char s [] = {"dog"};
char t [] = {"house"}
```

Using these declarations, the C compiler would have allocated just enough space to hold each word including the null character. Now suppose we want to concatenate these strings together to produce the new string, "doghouse." To do this we use the C function ***strcat*** (See Figure 2.8). Two strings are joined together by ***strcat(s, t)***, which stores the result in *s*. Although *s* has increased in length by five, we have no additional space in *s* to store the extra five characters. Our compiler handled this problem inelegantly: it simply overwrote the memory to fit in the extra five characters. Since we declared *t* immediately after s, this meant that part of the word "house" disappeared.

char s1 [] = {"dog"};
char t1 [] = {"house"}
이 상태에서 strcat (s1, t1)하면 실행시에 오류발생한다.



Function	Description
*char *strcat(char *dest, char *src)*	concatenate *dest* and *src* strings; return result in *dest*
*char *strncat(char *dest, char *src, int n)*	concatenate *dest* and *n* characters from *src*; return result in *dest*
*char *strcmp(char *str1, char *str2)*	compare two strings; return < 0 if *str1* < *str2*; 0 if *str1* = *str2*; > 0 if *str1* > *str2*
*char *strncmp(char *str1, char *str2, int n)*	compare first *n* characters return < 0 if *str1* < *str2*; 0 if *str1* = *str2*; > 1 if *str1* > *str2*
*char *strcpy(char *dest, char *src)*	copy *src* into *dest*; return *dest*
*char *strncpy(char *dest, char *src, int n)*	copy *n* characters from *src* string into *dest*; return *dest*;
*size_t strlen(char *s)*	return the length of a *s*
*char *strchr(char *s, int c)*	return pointer to the first occurrence of *c* in *s*; return *NULL* if not present
*char *strrchr(char *s, int c)*	return pointer to last occurrence of *c* in *s*; return *NULL* if not present
*char *strtok(char *s, char *delimiters)*	return a token from *s*; token is surrounded by *delimiters*
*char *strstr(char *s, char *pat)*	return pointer to start of *pat* in *s*
*size_t strspn(char *s, char *spanset)*	scan *s* for characters in *spanset*; return length of span
*size_t strcspn(char *s, char *spanset)*	scan *s* for characters not in *spanset*; return length of span
*char *strpbrk(char *s, char *spanset)*	scan *s* for characters in *spanset*; return pointer to first occurrence of a character from *spanset*

Figure 2.8: C string functions

Example 2.2 [String insertion]: Assume that we have two strings, say *string1* and *string2*, and that we want to insert *string2* into *string1* starting at the *i*th position of *string1*. We begin with the declarations:

```
#include <string.h>
#define MAX_SIZE 100 /*size of largest string*/
char stringl[MAX_SIZE], *s = stringl;
char string2[MAX_SIZE], *t = string2;
```

The call **strnins(s, t, 0)** is equivalent to **strcat(t, s).**

Program 2.12 is presented as an example of manipulating strings. It should never be used in practice as it is wasteful in its use of time and space. Try to revise it so the string *temp* is not required.

char *strcpy(char *, const char *); 앞 문자열의 처음에 뒤 문자열을 복사하여 그 복사된 문자열을 반환한다.
char *strncpy(char *, const char *, size_t n); 앞 문자열의 처음에 뒤 문자열을 NULL 문자까지, 최대 n개의 문자를 복사하여 그 복사된 문자열을 반환한다.

```
void strnins(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(EXIT_FAILURE);
        }
        if (!strlen(s))
                strcpy(s, t);
        else if (strlen(t)) {
                strncpy(temp, s, i);
                strcat(temp, t);
                strcat (temp, (s+i));
                strcpy (s, temp);
        }
}
```

Program 2.12: String insertion function

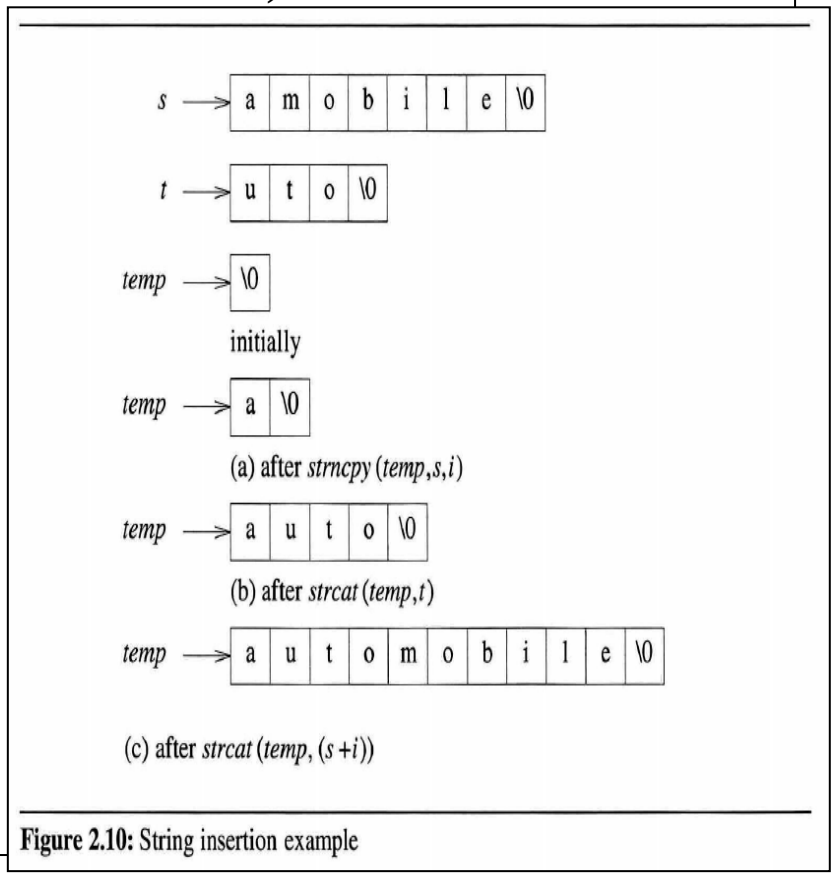


Figure 2.10: String insertion example

2.7.3 Pattern Matching

Assume that we have two strings, *string* and *pat*, where *pat* is a pattern to be searched for in *string*. The easiest way to determine if *pat* is in *string* is to use the built-in function *strstr*.

```
char pat[MAX_SIZE], string[MAX_SIZE], *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");
```

char * strstr(const char *string, const char * pat);

If *pat* is in *string*, it returns a pointer to the start of *pat* in *string*.
If *pat* is not in *string*, it returns a null pointer.

Although *strstr* seems ideally suited to pattern matching, we may want to develop our own pattern matching function because there are several different methods for implementing a pattern matching function.

Brute-force algorithm

```
int pattern_find(char *string, char *pat)
{ /* match from the beginning of pattern */
        int i, j, start = 0;
        int lasts = strlen(string)-1;
        int lastp = strlen(pat)-1;

        for (i = 0; i <= lasts - lastp; i++) {
                for (j = 0; j <= lastp; j++)
                        if (string[i+j] != pat[j]) break;
                if ( j == lastp+1) return i; /* pattern start index in string */
        }
        return -1; /* not found */
}
```

Program : Brute-force pattern matching

참고 : 가장 쉽지만 비효율적인 알고리즘

< 직선적 패턴 매칭 알고리즘 >

string : a b a c a b a c a b a b …

pat : a b a b

a b a b

a b a b

a b a b

시간복잡도 : 최악의 경우 O(*mn*)

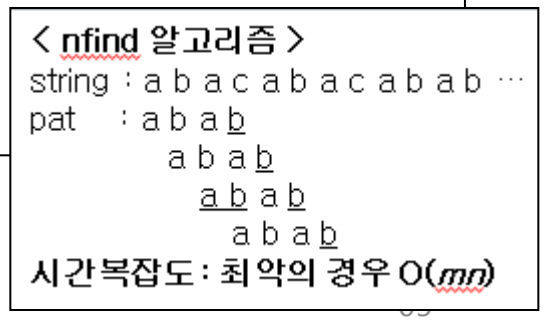
The worst case is rare in typical applications. Hence, the **`indexOf()`** method in Java's **`String`** class uses brute-force.

Simple algorithm

```
int nfind(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++);
                        if ( j == lastp) return start; /* successful */
                }
        }
        return -1;
}
```

Program 2.13: Pattern matching by checking end indices first



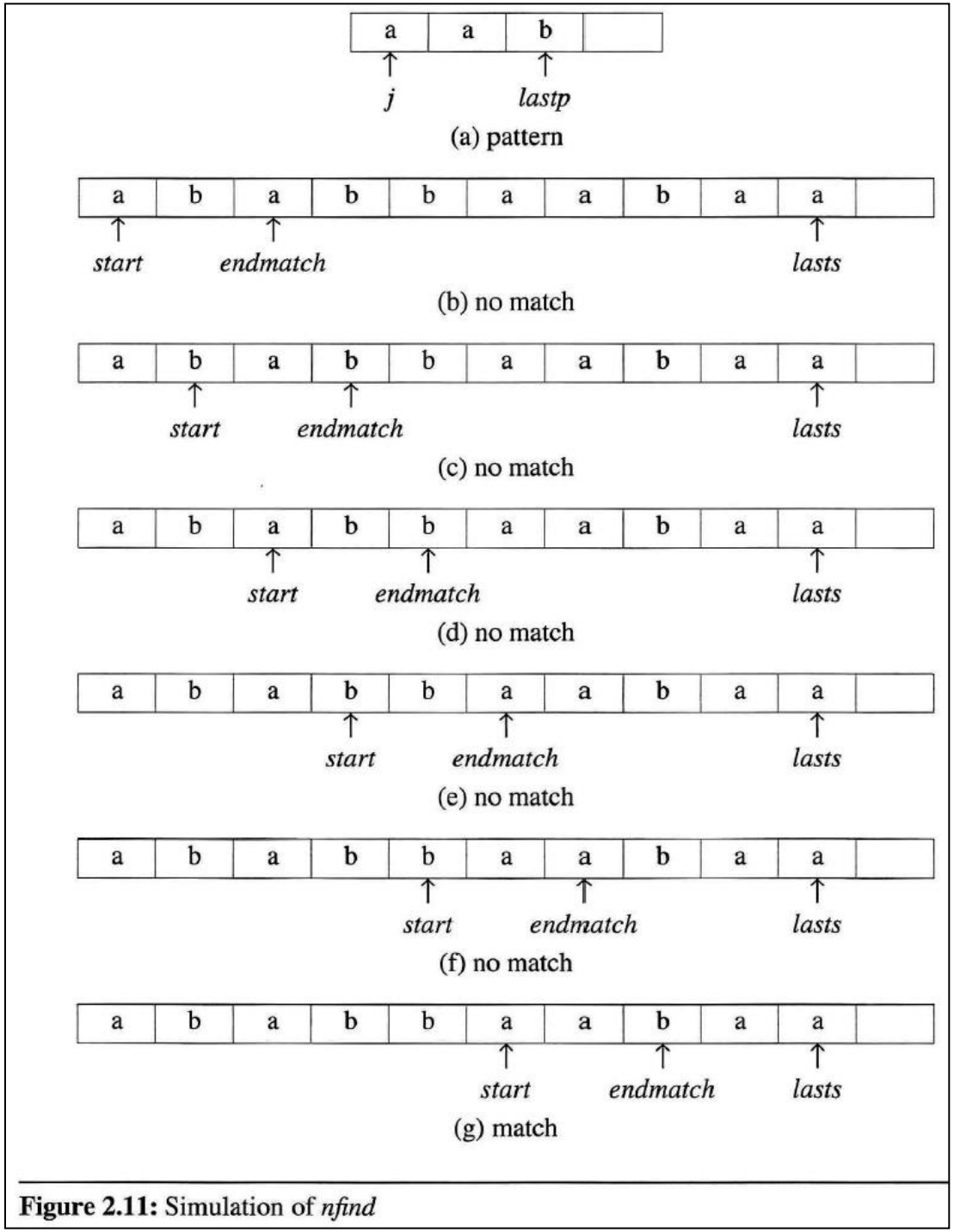


Figure 2.11: Simulation of *nfind*

The length of *pat* is ***n*** and the length of *string* is ***m***

Time complexity:

O(m) – linear, for the best case

- string : aa…a
- pattern: a…ab

O(*m*·*n*), for the worst case

- String : bbbbbbbbbb…bbbbb
- pattern: bb...bab

Brute-force is not good enough for all applications.
Theoretical challenge: Linear-time guarantee.

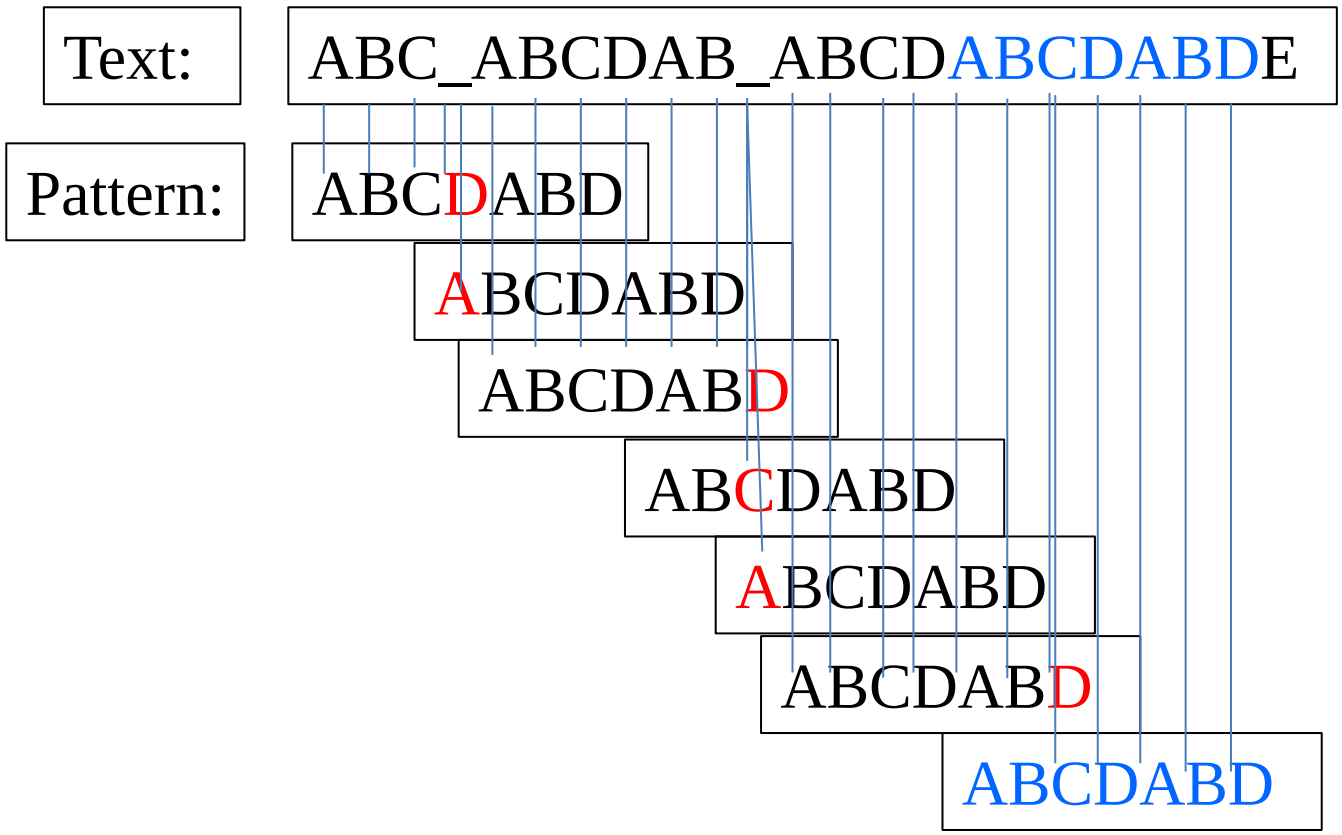
Ideally, we would like an algorithm that works in **O(strlen(string)+ strlen(pat))** time. This is optimal for this problem as in the worst case it is necessary to look at all characters in the pattern and string at least once.

We want to search the string for the pattern without moving backwards in the string. That is, if a mismatch occurs we want to use our knowledge of the characters in the pattern and the position in the pattern where the mismatch occurred to determine where we should continue the search. Knuth, Morris, and Pratt have developed a pattern matching algorithm that works in this way and has **linear complexity**.

- **Fast algorithm by Knuth, Morris, Pratt**

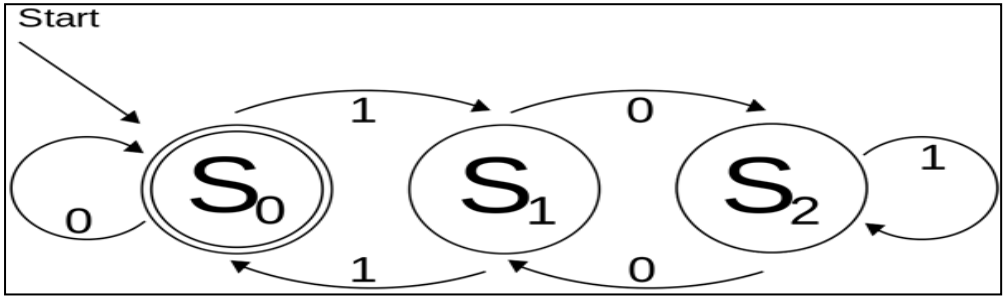
https://www.cs.princeton.edu/~rs/AlgsDS07/21PatternMatching.pdf

Basic Idea


Text:
ABC_ABCDAB_ABCDABCDABDE
Pattern:
ABCDABD
ABCDABD
ABCDABD
ABCDABD
ABCDABD
ABCDABD
ABCDABD

Deterministic finite automaton (결정 유한 오토마턴)

In theory of computation, a branch of theoretical computer science, a deterministic finite automaton (DFA)—also known as deterministic finite state machine—is a finite state machine that accepts/rejects finite strings of symbols and only produces a unique computation (or run) of the automaton for each input string. **Deterministic** refers to the uniqueness of the computation. In search of simplest models to capture the finite state machines, McCulloch and Pitts were among the first researchers to introduce a concept similar to finite automaton in 1943.



The figure illustrates a deterministic finite automaton using a state diagram. In the automaton, there are three states: S0, S1, and S2 (denoted graphically by circles). The automaton takes a finite sequence of 0s and 1s as input. For each state, there is a transition arrow leading out to a next state for both 0 and 1. Upon reading a symbol, a DFA jumps deterministically from a state to another by following the transition arrow. For example, if the automaton is currently in state S0 and current input symbol is 1 then it deterministically jumps to state S1. A DFA has a start state (denoted graphically by an arrow coming in from nowhere) where computations begin, and a set of accept states (denoted graphically by a double circle) which help define when a computation is successful.

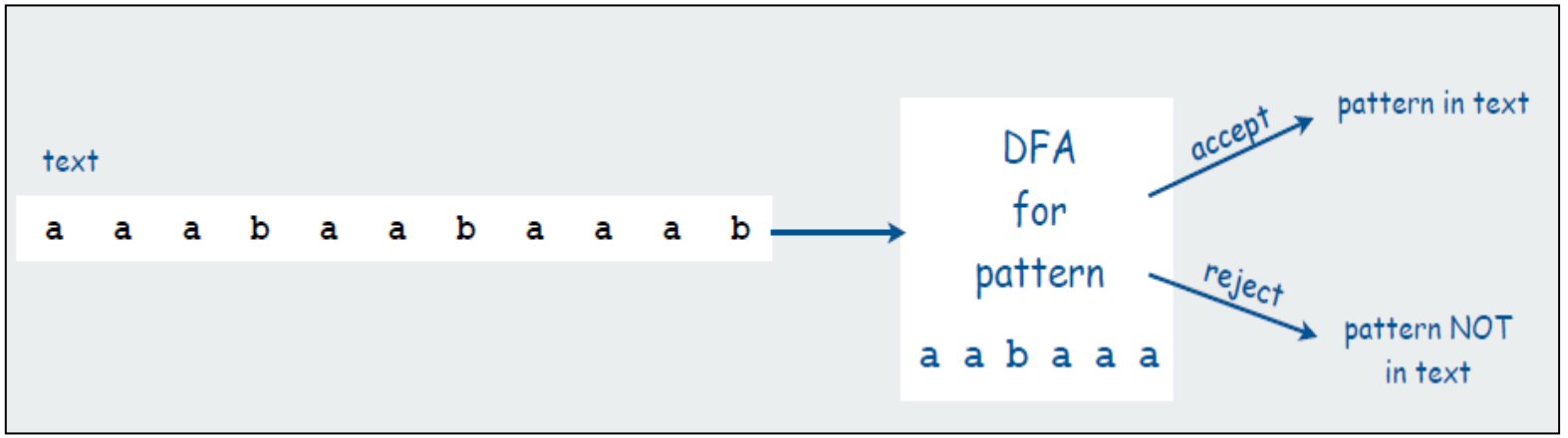
Knuth-Morris-Pratt (KMP) exact pattern-matching algorithm

Classic algorithm that meets both challenges

- linear-time guarantee
- no backup in text stream

Basic plan (for binary alphabet)

- build DFA from pattern
- simulate DFA with text as input

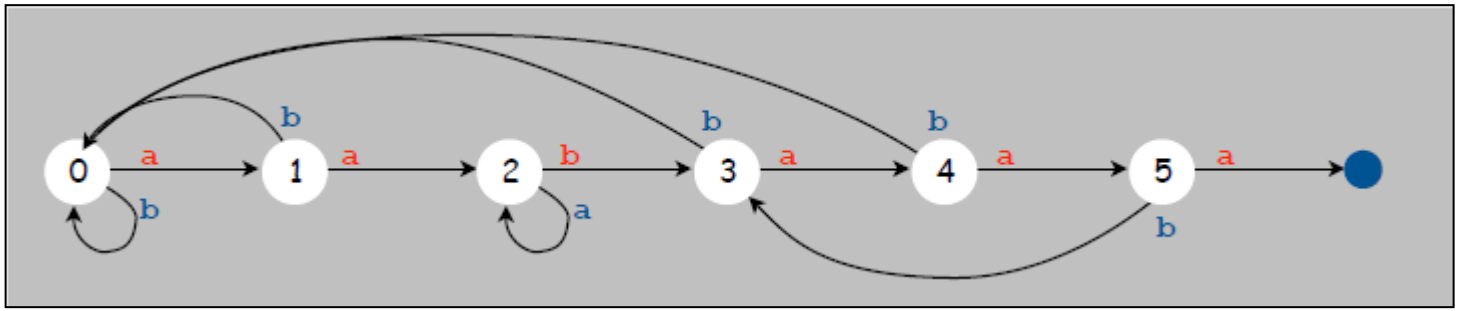


Knuth-Morris-Pratt DFA example

One state for each pattern character

- Match input character: move from *i* to *i*+1
- Mismatch: move to previous state

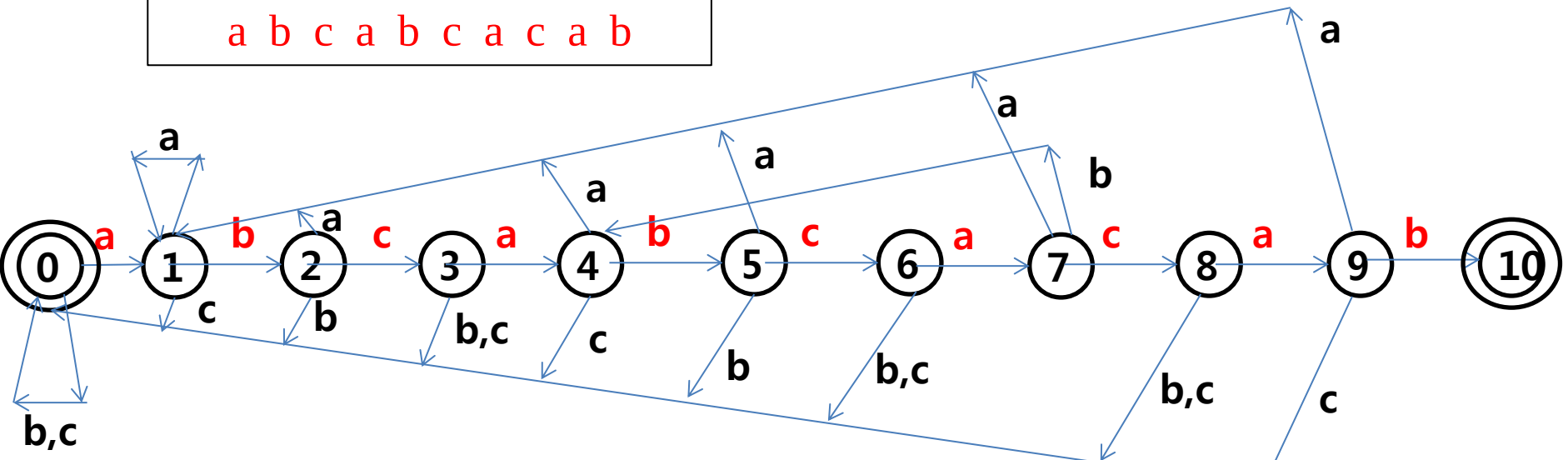
DFA
for
pattern
a a b a a a



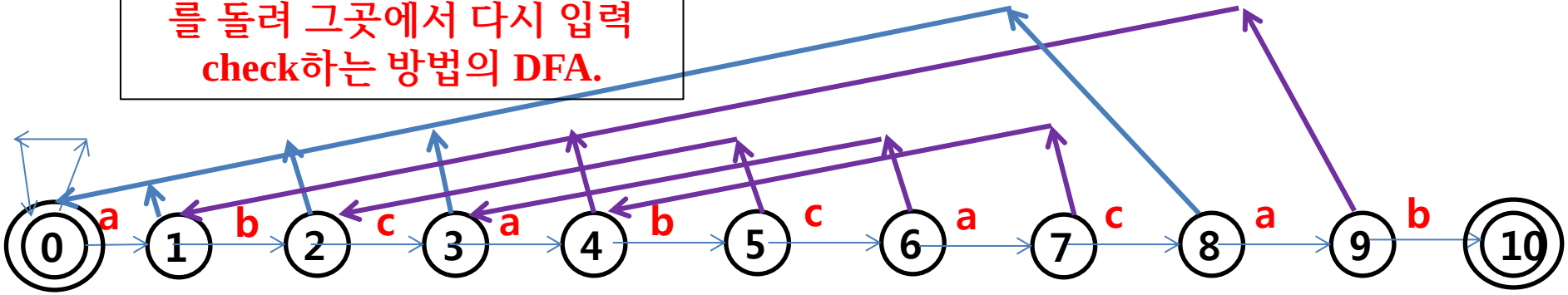
DFA for pattern

a b c a b c a c a b

단순 DFA



DFA: 만약 그 값이 아니면 상태를 돌려 그곳에서 다시 입력 check하는 방법의 DFA.



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

Using their example, suppose pat= 'abcabcacab'.

Let $s = s_0\ s_1\ s_2\ \cdots\ s_{m-1}$ be the string and assume that we are currently determining whether or not there is **a match beginning at s_i**.

- If $s_i \neq$ a then, clearly, we may proceed by comparing s_{i+1} and *a*.
- Similarly if s_i = a and $\boldsymbol{s_{i+1}} \neq$ b then we may proceed by comparing $\boldsymbol{s_{i+1}}$ and a.
- If $s_i\ \boldsymbol{s_{i+1}}$ = ab and $\boldsymbol{s_{i+2}} \neq$ c then we have the situation:

  s= '- *a b* ? ? ? ?'

  pat= '*a b c a b c a c a b*'

  The ? implies that we do not know what the character in *s* is. The first ? in *s* represents $\boldsymbol{s_{i+2}}$ and $\boldsymbol{s_{i+2}} \neq$ c. At this point we know that we may continue the search for a match by comparing the first character in *pat* with $\boldsymbol{s_{i+2}}$. There is no need to compare this character of *pat* with $\boldsymbol{s_{i+1}}$ as we already know that $\boldsymbol{s_{i+1}}$ is the same as the second character of *pat*, *b*, and so $\boldsymbol{s_{i+1}} \neq$ a.

- Let us try this again assuming a match of the first four characters in *pat* followed by a nonmatch, i.e., $s_{i+4} \neq b$. We now have the situation:

  s= '- a b c a ? ? ?'

  pat= 'a b c a b c a c a b'

  We observe that the search for a match can proceed by comparing s_{i+4} and the second character in pat, *b*. This is the first place a partial match can occur by sliding the pattern *pat* towards the right.

- Thus, by knowing the characters in the pattern and the position in the pattern where a mismatch occurs with a character in *s* we can determine where in the pattern to continue the search for a match without moving backwards in *s*. To formalize this, we define **a failure function for a pattern**.

Fast algorithm by Knuth, Morris, Pratt(cont')

Definition : If $p=p_0p_1\ldots p_{n-1}$ is a pattern, then its ***failure function***, ***f***, is defined as:

$$f(j) = \begin{cases} \text{largest } i < j \text{ such that } p_0p_1 \cdots p_i = p_{j-i}p_{j-i+2} \cdots p_j \text{ if such an } i \geq 0 \text{ exists} \\ -1 \qquad \text{otherwise} \end{cases}$$

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

$$f(j) = \begin{cases} -1 & \text{if } j = 0 \\ f^m(j-1)+1 & \text{where } m \text{ is the least integer } k \text{ for which } P_{f^k(j-1)+1} = P_j \\ -1 & \text{if there is no k satisfying the above} \end{cases}$$

* note that $f^1(j) = f(j)$ and $f^m(j) = f(f^{m-1}(j))$.

Failure[j]의 의미
S[i+1] != P[j+1]이면
S[i+1]을 P[0] 부터 비교하지 말고
S[i+1]을 P[failure[j]+1]과 비교하라는 의미이다.

From the definition of the failure function, we arrive at the following rule for pattern matching:

If a partial match is found 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 $j \neq 0$. If $j = 0$, then we may continue by comparing s_{i+1} and p_0.

This pattern matching rule translates into function *pmatch* (Program 2.14). The following declarations are assumed:

Assuming that s[i-j]=p[0], …, s[i-1]=p[j-1],

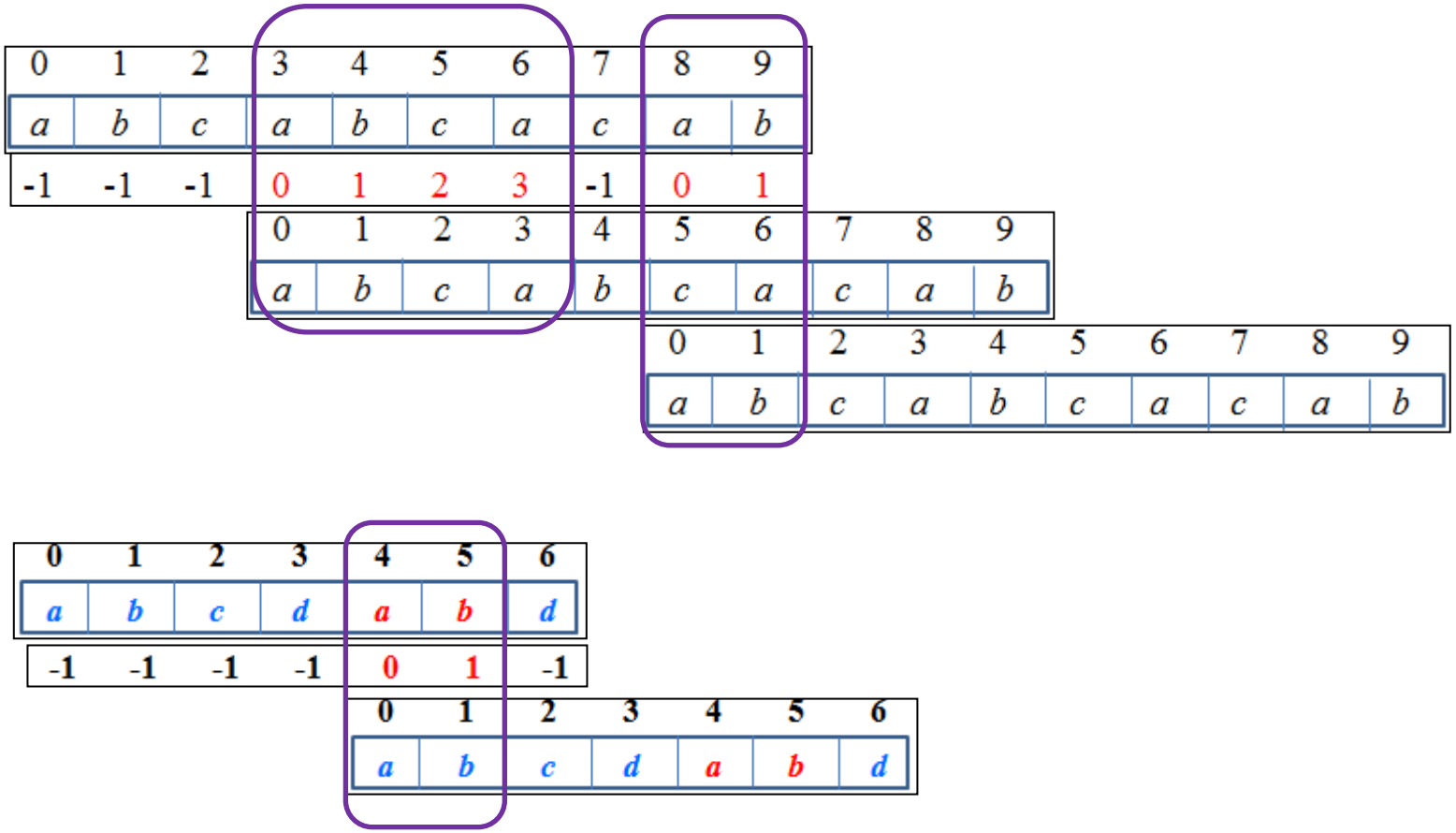
Upon s[i] != p[j],

Matching may be resumed by comparing s[i] and p[f[j–1]+1] if j != 0.

Matching may be resumed by comparing s[i+1] and p[0] if j = 0.

$$\begin{array}{cccccccc} \cdots & s_{i-j} & s_{i-j+1} & \cdots & s_{i-1} & s_i & s_{i+1} & \cdots \\ & p_0 & p_1 & \cdots & p_{j-1} & p_j & p_{j+1} & \end{array}$$

- Fast algorithm by Knuth, Morris, Pratt
  - Can we do it in **linear time**? → yes!



Got from Wikipedia

```
          1         2
i: 012345678901234567890 12
m: 012345678901234567890 12
S: ABC ABCDAB ABCDABCDABDE
P: ABCDABD
j: 0123456
       ABCDABD
       0123456
```

- S[i=3] != P[j=3].

j != 0, so, resume match S[i=3] and P[j=f[j-1]+1]=P[j=0].

- S[i=3] != P[j=0].

Advance i (now 4) but j is still 0.

0	1	2	3	4	5	6
A	B	C	D	A	B	D
-1	-1	-1	-1	0	1	-1

```
          1         2
i:  012345678901234567890 12
m:  012345678901234567890 12
S:  ABC ABCDAB ABCDABCDABDE
P:      ABCDABD
j:      0123456
```

- S[i=10] != P[j=6].

j != 0, so, resume match S[i=10] and P[j=f[j-1]+1]=P[j=2].

```
          1         2
i:  012345678901234567890 12
m:  012345678901234567890 12
S:  ABC ABCDAB ABCDABCDABDE
P:          ABCDABD
j:          0123456
              ABCDABD
              0123456
```

- S[i=10] != P[j=2].

j != 0, so, resume match S[i=10] and P[j=f[j-1]+1]=P[j=0].

- S[i=10] != P[j=0].

Advance i (now 11) but j is still 0.

```
          1         2
i:  012345678901234567890 12
m:  012345678901234567890 12
S:  ABC ABCDAB ABCDABCDABDE
P:              ABCDABD
j:              0123456
```

- S[i=17] != P[j=6].

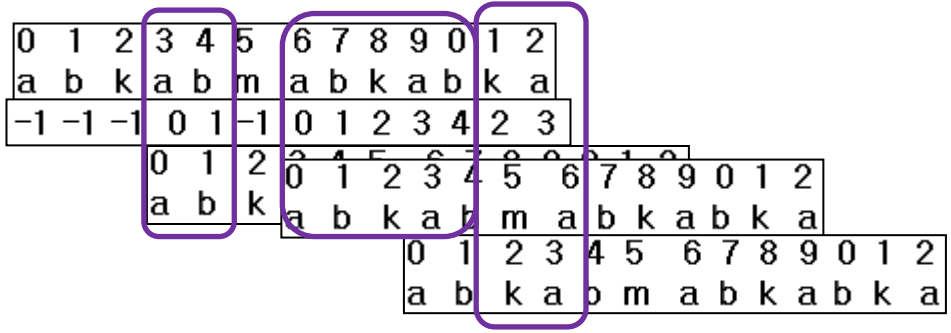
j != 0, so, resume match S[i=17] and P[j=f[j-1]+1]=P[j=2].

```
          1         2
i:  012345678901234567890 12
m:  012345678901234567890 12
S:  ABC ABCDAB ABCDABCDABDE
P:                  ABCDABD
j:                  0123456
```

This time we are able to complete the match, whose first character is S[15].

j	0	1	2	3	4	5	6
pat	*a*	*b*	*c*	*d*	*a*	*b*	*d*
f	-1	-1	-1	-1	0	1	-1

```
#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];
```




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

Program 2.15: Computing the failure function

Time complexity : O(n), if pattern length is n

```
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);
}
```

Program 2.14: Knuth, Morris, Pratt pattern matching algorithm

Time complexity : O(*strlen(string)*)

Time complexity of fail and pmatch: O(*strlen(string)* + *strlen(pattern)*)

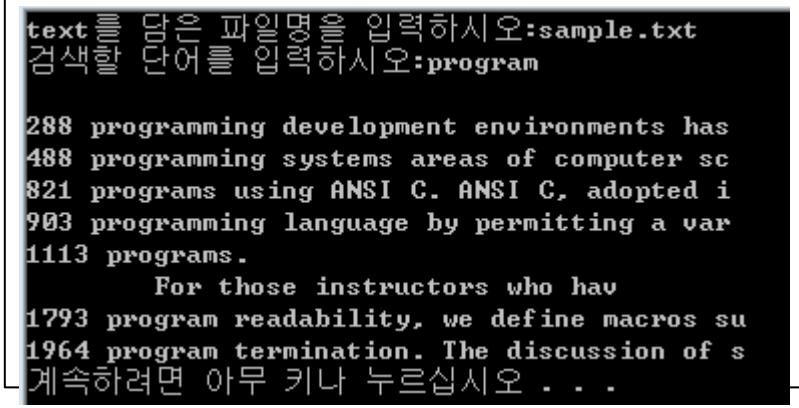
Pattern Matching 문제 [4 점]

다음 페이지의 내용은 우리 교재의 PREFACE의 첫 세 paragraph를 나타낸다. 우리는 2.7.3 Pattern Matching에서 공부한 내용을 기반으로 그 text에서 단어를 찾고자 한다. 반드시, 그 text를 파일로 저장한 후 그 파일을 읽어서 수행하여야 한다.

다음을 수행하시오.

1. 그 text를 읽어 배열에 저장하시오.
2. 그 text에서 탐색할 단어를 받아 들이시오. 예를 들어 “program”을 입력한다.
3. Simple algorithm으로 그 단어를 그 text에서 찾으시오.
4. Fast algorithm by Knuth, Morris, Pratt 으로 그 단어를 그 text에서 찾으시오.

3과 4의 수행에서 그 단어가 나오는 모든 곳을 찾아야 하며 각 발견마다 그 단어를 포함하여 모두 40 자를 출력하여야 한다. 아래 결과는 내가 수행한 결과이다.



Why Fundamentals of Data Structures in C? There are several answers. The first, and most important, is that C has become the main development language both on personal computers (PCs and Macs) as well as on UNIX-based workstations. Another reason is that the quality of C compilers and C programming development environments has improved to the point where it makes sense to provide instruction to beginners in a C environment. Finally, many of the concepts that need to be taught in the programming systems areas of computer science, such as virtual memory, file systems, automatic parser generators, lexical analyzers, networking, etc. are implemented in C. Thus, instructors are now teaching students C early in their academic life so that these concepts can be fully explored later on.

We have chosen to present our programs using ANSI C. ANSI C, adopted in 1983, has attempted to strengthen the C programming language by permitting a variety of features not allowed in earlier versions. Some of these features, such as typing information in the function header, improve readability as well as reliablity of programs.

For those instructors who have used other versions of the book Fundamentals of Data Structures, you will find that this book retains the in-depth discussion of the algorithms and computing time analyses. In addition we have attempted to preserve the chapter organization and the presentation style of the earlier book whenever it was desirable. But this has not kept us from making improvements. For example, pointers and dynamic memory allocation are introduced in Chapter 1 as these concepts are quite common in C. Error messages are written to stderr. Programs that use system function calls, such as malloc, check that they return successfully. However, to enhance program readability, we define macros such as MALLOC that both invoke malloc and do the checking. We use exit(EXIT_FAILURE) and exit(EXIT_SUCCESS) for normal and abnormal program termination. The discussion of strings is now found in the chapter on arrays.

Sparse Matrix (희소 행렬) 문제 [4 점] 🡸 정답만 인정

우리는 “2.5 SPARSE MATRIX”에서 두 개의 희소 행렬을 읽어 그를 Figure 2.5 (a)와 같이 <row, col, value>의 엔트리를 가지는 배열로 표현하고, 그 표현을 기반으로 Program 2.10에서 그 두 행렬의 곱을 구하는 연산을 공부하였다. 이제 우리는 아래 6 X 6의 두 희소 행렬 A와 B에 대해 연산을 수행하고자 한다.

$$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}$$

```
6 6 12
0 0 15
0 2 28
0 3 22
0 5 -15
1 0 -858993460
1 2 3
1 3 -6
4 0 15
4 3 22
4 5 -15
5 0 -858993460
5 3 -6
```

```
6 6 10
0 0 15
0 2 28
0 3 22
0 5 -15
1 2 3
1 3 -6
4 0 15
4 3 22
4 5 -15
5 3 -6
```

matrix A　　matrix B

위 두 희소 행렬 A와 B를 대상으로 다음을 수행하시오.

1. 키보드로 부터 읽어 들인 후 A * B를 mmult를 호출하여 수행한 후, 결과를 printMatrix로 출력한다.