

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 $\langle index, value \rangle$ 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, $\{0, \dots, n-1\}$ 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 $A \in \text{Array}$, $i \in \text{index}$, $x \in \text{item}$, j , $size \in \text{integer}$

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** ($i \in \text{index}$) **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 $\langle i, x \rangle$
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 $a + i * \text{sizeof}(\text{int})$, where *a* is the **base address**.

Observe that there is a difference between a declaration such as

```
int *list1;
```

and

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

The variables *list1* 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 print1(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("%08u%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

C 언어 review

- 포인터 상수
- 메인 메모리의 네 영역(Segment): 스택, 힙, 데이터, 코드 영역
- Stack에 지역변수의 할당
- 숫자의 binary encoding schema: Little Endian, Big Endian.

포인터 상수

```
const int *pi = &i;
```


*pi를 사용해 포인터 pi가 가리키는 변수인 i를 수정할 수 없도록 하는 상수 선언 방법이다.
간접 연산식 *pi를 상수로 만드는 방법으로, 선언 이후 *pi를 l-value로 사용할 수 없다.

```
int const *pi = &i;
```

위와 동일한 문장으로 간접 연산식 *pi를 상수로 만드는 방법이다.

```
int* const pi = &i;
```

포인터 pi에 저장되는 초기 주소값을 더 이상 수정할 수 없도록 하는 상수 선언 방법이다.
포인터 변수 pi를 상수로 만드는 방법으로, 선언 이후 pi를 l-value로 사용할 수 없다.



메인 메모리의 네 영역(Segment): 스택, 힙, 데이터, 코드

High address



Stack Segment

(매개 변수, 지역 변수)

스택 영역에서는 먼저 선언되는 변수가 큰 주소값의 메모리 공간에 할당된다.



Heap Segment

(동적 할당 변수)

힙 영역에서는 먼저 선언되는 변수가 작은 주소값의 메모리 공간에 할당된다.

Data Segment

(전역 변수, 정적 변수)

데이터 영역에서는 먼저 선언되는 변수가 작은 주소값의 메모리 공간에 할당된다.

Code Segment

(instruction code(실행코드))

Low address

Stack에 지역변수의 할당

시작 주소값: 어떤 변수에 공간이 할당된 경우, **그 공간의 가장 작은 주소값**을 그 변수의 시작 주소값이라 한다.

```
char ch = 'a';  
int num = 100;  
double real[2] = {3.14, 1.87};
```

- (1) **stack**에 변수 **ch**의 공간이 할당된다.
- (2) **stack**에 변수 **num**의 공간이 할당된다.
- (3) **stack**에 배열 **real**의 공간이 할당된다.

- **Stack은 큰 주소에서 작은 주소의 방향으로 차례로 공간을 할당한다.** 따라서, 세 개의 공간 할당에서, 변수 **ch**가 가장 큰 주소에, 변수 **num**이 그 보다 작은 주소에, 그리고 배열 **real**이 가장 작은 주소에 할당된다.
- **배열 **real**의 경우, 그 배열의 시작 주소 즉, 가장 작은 주소로부터 8 bytes가 **real[0]**, 그 다음 8 bytes가 **real[1]**으로 사용된다.**

편의상 아래와 같이 생각하자.

High address

0025F883	ch	<'a'>
0025F874	num	<00 00 00 64>
0025F864	real[1]	<yy yy yy yy yy yy yy yy>
0025F85C	real[0]	<xx xx xx xx xx xx xx xx>

Low address

여러 바이트를 이용하여 표기되는 숫자의 **binary encoding scheme**(바이너리 표기방법)

➔ **리틀엔디안(Little Endian):** 데이터의 최하위 바이트(**LSB**)를 메모리의 낮은 주소에 저장한다.

16진수	00	00	00	64
2진수	0000 0000	0000 0000	0000 0000	0110 0100
주소값	0012FF7B	0012FF7A	0012FF79	0012FF78

표 14.1 주소값 0012FF78일 때 정수 int형의 저장값 100(00000064)을 리틀엔디안 방식으로 저장한 메모리

빅엔디안(Big Endian): 데이터의 최하위 바이트(**LSB**)를 메모리의 높은 주소에 저장한다.

16진수	64	00	00	00
2진수	0110 0100	0000 0000	0000 0000	0000 0000
주소값	0012FF7B	0012FF7A	0012FF79	0012FF78

표 14.2 주소값 0012FF78일 때 정수 int형의 저장값 100(00000064)을 빅엔디안 방식으로 저장한 메모리

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

```
int main(void)
```

```
{
```

```
    int i = 10;
```

```
    int *pi = &i;
```

```
    printf("변수 i의 주소는 %p이며\n", &i);
```

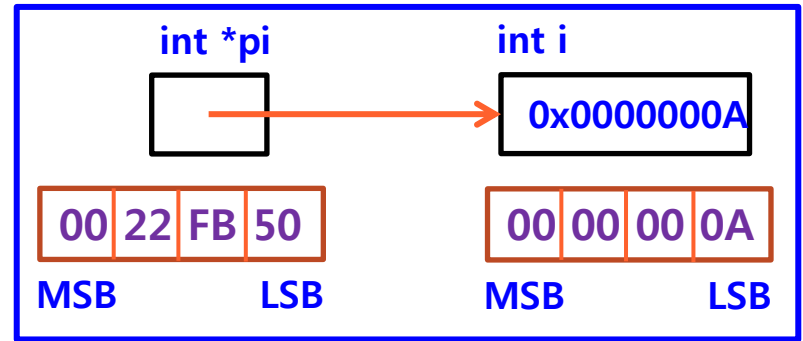
```
    printf("변수 i에 저장된 값은 %d이다.\n", i);
```

```
    printf("변수 pi의 주소는 %p이며\n", &pi);
```

```
    printf("변수 pi의 내부값은 주소로 %p이다.\n", pi);
```

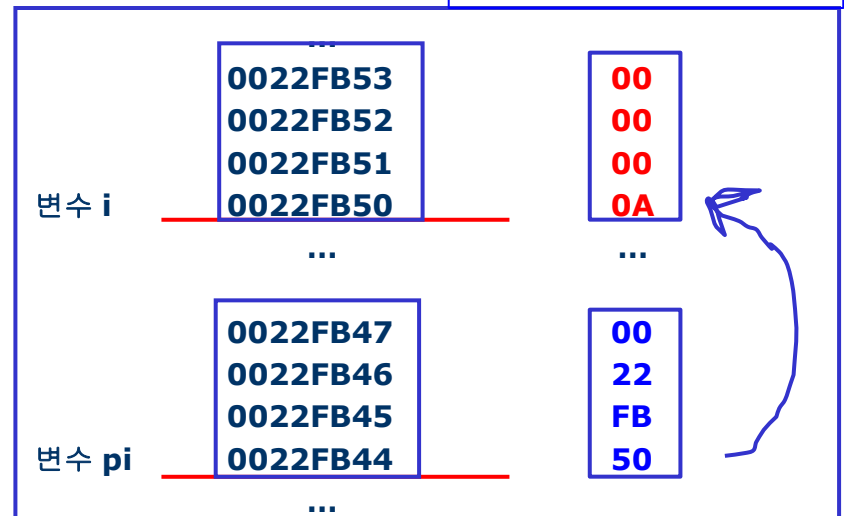
```
    return 0;
```

```
}
```



Little Endian 방식

```
변수 i의 주소는 0022FB50이며
변수 i에 저장된 값은 10이다.
변수 pi의 주소는 0022FB44이며
변수 pi의 내부값은 주소로 0022FB50이다.
```



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));
```

나는
위의 `MALLOC(pf, sizeof(float));`
의 `macro`를 사용하는 것 보다
우측의 코드를 더 좋아한다.



```
pf = (float*) malloc(sizeof(float));
if (pf == NULL) {
    fprintf(stderr, "Insufficient memory");
    exit(EXIT_FAILURE);
}
```

(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);
```

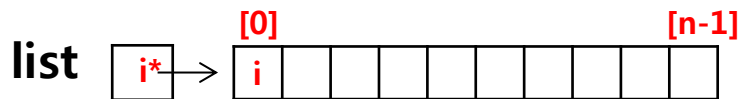
changes the size of the memory block pointed at by *p* to *s*. The contents of the first $\min\{s, \text{oldSize}\}$ bytes of the block are unchanged as a result of this resizing. When $s > \text{oldSize}$ the additional $s - \text{oldSize}$ have an unspecified value and **when $s < \text{oldSize}$, the rightmost $\text{oldSize} - s$ 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));
if (list == NULL) {
    fprintf(stderr, "Insufficient memory");
    exit(EXIT_FAILURE);
}
```

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 ints.

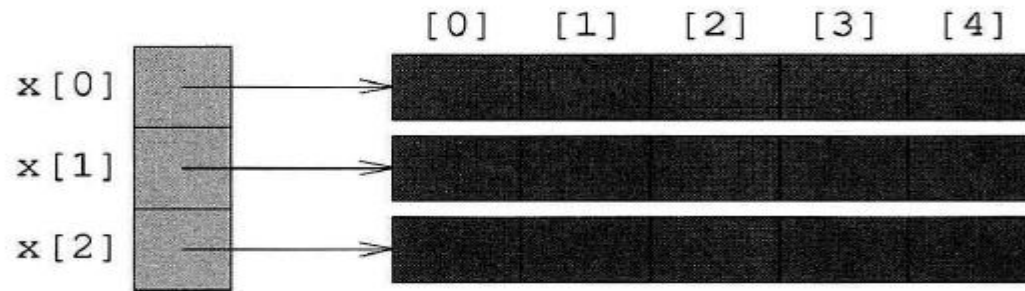


Figure 2.2: Array-of-arrays representation

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 $j * \text{sizeof}(\text{int})$ 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;
}

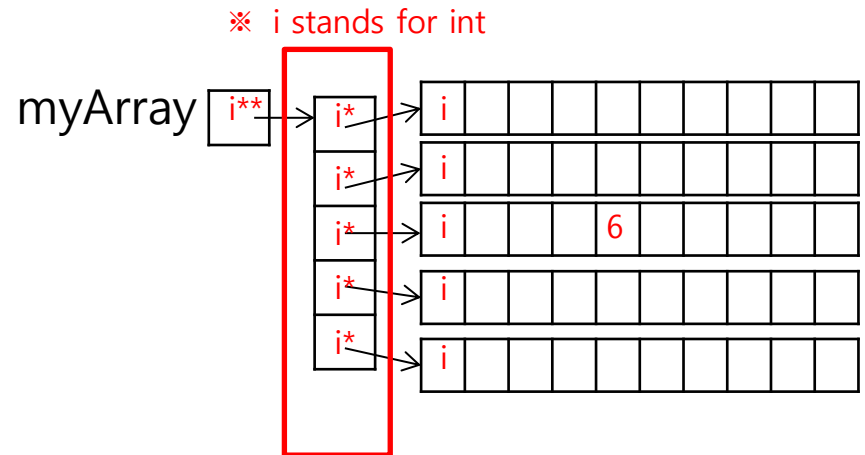
```

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]);
}

```



```

#include <stdio.h>
int main(void)
{
    int a1[3];
    int a2[2][3];

    a1[2] = 4;
    a2[1][2] = 7;
    printf("배열 a1에서 \na1 = %p, \n", a1);
    printf("&a1[0] = %p, &a1[1] = %p, &a1[2] = %p, \n", &a1[0], &a1[1], &a1[2]);
    printf("배열 a1[2] = %d.\n\n", a1[2]);
    printf("배열 a2에서 \na2 = %p, \n", a2);
    printf("&a2[0] = %p, \n", &a2[0]);
    printf("&a2[0][0] = %p, &a2[0][1] = %p, &a2[0][2] = %p,\n",
           &a2[0][0], &a2[0][1], &a2[0][2]);
    printf("&a2[1] = %p, \n", &a2[1]);
    printf("&a2[1][0] = %p, &a2[1][1] = %p, &a2[1][2] = %p,\n",
           &a2[1][0], &a2[1][1], &a2[1][2]);
    printf("배열 a2[1][2] = %d이다.\n", a2[1][2]);
    return 0;
}

```

...		
a1	003FFAE3	00
	003FFAE2	00
	003FFAE1	00
	003FFAE0	04
	003FFADF	00
	003FFADE	00
	003FFADD	00
	003FFADC	00
	003FFADB	00
	003FFADA	00
	003FFAD9	00
	003FFAD8	00

```

배열 a1에서
a1 = 003FFAD8,
&a1[0] = 003FFAD8, &a1[1] = 003FFADC, &a1[2] = 003FFAE0,
배열 a1[2] = 4.

배열 a2에서
a2 = 003FFAB8,
&a2[0] = 003FFAB8,
&a2[0][0] = 003FFAB8, &a2[0][1] = 003FFABC, &a2[0][2] = 003FFAC0,
&a2[1] = 003FFAC4,
&a2[1][0] = 003FFAC4, &a2[1][1] = 003FFAC8, &a2[1][2] = 003FFACC,
배열 a2[1][2] = 70이다.

```

...		
a2	003FFACF	00
	003FFACE	00
	003FFACD	00
	003FFACC	07
	003FFACB	00
	003FFACA	00
	003FFAC9	00
	003FFAC8	00
	003FFAC7	00
	003FFAC6	00
	003FFAC5	00
	003FFAC4	00
	003FFAC3	00
	003FFAC2	00
	003FFAC1	00
	003FFAC0	00
	003FFABF	00
	003FFABE	00
	003FFABD	00
	003FFABC	00
	003FFABB	00
	003FFABA	00
	003FFAB9	00
	003FFAB8	00

```

#include <stdio.h>
int gvz = 5;
char sga[100] = "global";
const char *sgp1 = "dog1";
const char *sgp2 = "dog2";

int main()
{
    int lvz = 7;
    char sla[] = { "dog" };
    const char *slp1 = "dog1";
    const char *slp2 = "dog2";

    printf("address of gvz = %p, contents of gvz = %d\n", &gvz, gvz);
    printf("address of sga = %p, string of sga = %s\n", sga, sga);
    printf("address of sgp1 = %p, contents of sgp1 = %p, string of sgp1 = %s\n", &sgp1, sgp1, sgp1);
    printf("address of sgp2 = %p, contents of sgp2 = %p, string of sgp2 = %s\n", &sgp2, sgp2, sgp2);

    printf("address of lvz = %p, contents of lvz = %d\n", &lvz, lvz);
    printf("address of sla = %p, string of sla = %s\n", sla, sla);
    printf("address of slp1 = %p, contents of slp1 = %p, string of slp1 = %s\n", &slp1, slp1, slp1);
    printf("address of slp2 = %p, contents of slp2 = %p, string of slp2 = %s\n", &slp2, slp2, slp2);

    return 0;
}

```

lvx
sla
slp1
slp2

7
"dog"

High address

Stack Segment

(매개 변수, 지역 변수)

```
address of gv_x = 0121A000, contents of gv_x = 5  
address of sga = 0121A008, string of sga = global  
address of sgp1 = 0121A004, contents of sgp1 = 01217B30, string of sgp1 = dog1  
address of sgp2 = 0121A06C, contents of sgp2 = 01217B38, string of sgp2 = dog2  
address of lv_x = 0038F750, contents of lv_x = 7  
address of sla = 0038F744, string of sla = dog  
address of slp1 = 0038F738, contents of slp1 = 01217B30, string of slp1 = dog1  
address of slp2 = 0038F72C, contents of slp2 = 01217B38, string of slp2 = dog2
```

Heap Segment

(동적 할당 변수)

Data Segment

(전역 변수, 정적 변수)

Code Segment

(instruction code(실행코드))

sgp2
sga
sgp1
gv_x

"global"

5

"dog2"
"dog1"

← 리터럴(literal)은
Data Segment의
Low Address 영역에
저장됨을 알 수 있다.

동일 리터럴(literal)은 오직 한번만
저장됨을 알 수 있다.

Low address

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 p1;
```

```
struct person {  
    char name[10];  
    int age;  
    float salary;  
};  
typedef struct person humanBeing;  
humanBeing p1;
```

```
struct {  
    char name[10];  
    int age;  
    float salary;  
} p1;
```

– **Structure member operator** : dot(.)

← p1 is a variable.

```
strcpy(p1.name, "james");
```

```
p1.age = 10;
```

```
p1.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 p1, p2;
```

```
if (strcmp(p1.name, p2.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 (p1 == p2)` and have the entire structure checked for equality, or if we could write `p1 = p2` and have that mean that the value of every field of the structure of *p2* is assigned as the value of the corresponding field of *p1*.

- 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(p1.name, p2.name);  
p1.age = p2.age;  
p1.salary = p2.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(p1,p2))
    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 p1, humanBeing p2)
```

```
{
```

```
    /* return TRUE if p1 and p2 are the same human being
       otherwise return FALSE */
```

```
    if (strcmp(p1.name, p2.name)) return FALSE;
```

```
    if (p1.age != p2.age) return FALSE;
```

```
    if (p1.salary != p2.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 p1;  
p1.dob.month = 2;  
p1.dob.day = 11;  
p1.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 field 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 pl, p2;  
  
pl.sexInfo.sex = male;  
pl.sexInfo.u.beard = FALSE;  
  
p2.sexInfo.sex = female;  
p2.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));
```

```
}
```

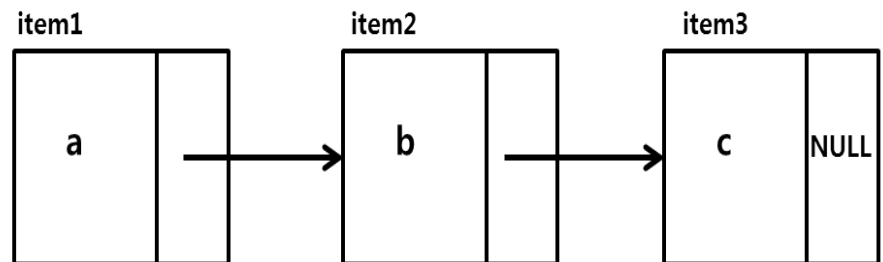
16	16
16	16
16	16

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 item1, item2, item3;  
item1.data = 'a'; item1.link = NULL;  
item2.data = 'b'; item2.link = NULL;  
item3.data = 'c'; item3.link = NULL;  
item1.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 i th item from a list, $0 \leq i < n$.
- Replace the item in the i th position of a list, $0 \leq i < n$.
- Insert a new item in the i th position of a list, $0 \leq i \leq n$. The items previously numbered $i, i+1, \dots, n-1$ become items numbered $i+1, i+2, \dots, n$.
- Delete an item from the i th position of a list, $0 \leq i < n$. The items numbered $i+1, \dots, n-1$ become items numbered $i, i+1, \dots, n-2$.

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**

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

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, $A(x) = \sum a_i x^i$ and $B(x) = \sum b_i x^i$ then :

$$A(x) + B(x) = \sum (a_i + b_i) x^i$$

$$A(x) B(x) = \sum (a_i x^i \sum (b_j x^j))$$

ADT Polynomial is

objects: $p(x) = a_1x^{e_1} + \dots + a_nx^{e_n}$; 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 $poly, poly1, poly2 \in \text{Polynomial}$, $coef \in \text{Coefficients}$, $expon \in \text{Exponents}$

<i>Polynomial</i> Zero()	::=	return the polynomial, $p(x) = 0$
<i>Boolean</i> IsZero(<i>poly</i>)	::=	if (<i>poly</i>) return FALSE else return TRUE
<i>Coefficient</i> Coef(<i>poly</i> , <i>expon</i>)	::=	if (<i>expon</i> \in <i>poly</i>) return its coefficient else return zero
<i>Exponent</i> LeadExp(<i>poly</i>)	::=	return the largest exponent in <i>poly</i>
<i>Polynomial</i> Attach(<i>poly</i> , <i>coef</i> , <i>expon</i>)	::=	if (<i>expon</i> \in <i>poly</i>) return error else return the polynomial <i>poly</i> with the term $\langle coef, expon \rangle$ inserted
<i>Polynomial</i> Remove(<i>poly</i> , <i>expon</i>)	::=	if (<i>expon</i> \in <i>poly</i>) return the polynomial <i>poly</i> with the term whose exponent is <i>expon</i> deleted else return error
<i>Polynomial</i> SingleMult(<i>poly</i> , <i>coef</i> , <i>expon</i>)	::=	return the polynomial $poly \cdot coef \cdot x^{expon}$
<i>Polynomial</i> Add(<i>poly1</i> , <i>poly2</i>)	::=	return the polynomial $poly1 + poly2$
<i>Polynomial</i> Mult(<i>poly1</i> , <i>poly2</i>)	::=	return the polynomial $poly1 \cdot poly2$

end 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 $A(x) = \sum_{i=0}^n a_i x^i$ would be represented as :

```
a.degree = n  
a.coef[i] =  $a_i$ ,  $0 \leq i \leq n$ 
```

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

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

	<i>startA</i>	<i>finishA</i>	<i>startB</i>		<i>finishB</i>	<i>avail</i>
	↓	↓	↓		↓	↓
<i>coef</i>	2	1	1	10	3	1
<i>exp</i>	1000	0	4	3	2	0
	0	1	2	3	4	5

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 $a[\text{MAX_ROWS}][\text{MAX_COLS}]$.

	col 0	col 1	col 2		col 0	col 1	col 2	col 3	col 4	col 5
row 0	-27	3	4	row 0	15	0	0	22	0	-15
row 1	6	82	-2	row 1	0	11	3	0	0	0
row 2	109	-64	11	row 2	0	0	0	-6	0	0
row 3	12	8	9	row 3	0	0	0	0	0	0
row 4	48	27	47	row 4	91	0	0	0	0	0
				row 5	0	0	28	0	0	0
(a)				(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 $\frac{\text{no. of nonzero elements}}{m \times n} \ll 1$

ADT *SparseMatrix* is

objects: a set of triples, $\langle \text{row}, \text{column}, \text{value} \rangle$, where *row* and *column* are integers and form a unique combination, and *value* comes from the set *item*.

functions:

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

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

return a *SparseMatrix* that can hold up to $\text{maxItems} = \text{maxRow} \times \text{maxCol}$ 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.

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
<i>a</i> [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))

행렬 전치 방법 1. 각 원소에 대해 차례로 전치 원소를 계산하여 전치 행렬에 첨가한 후, 그 전치 원소를 자신의 전치 위치로 이동시키는 방법.

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 $\langle i, j, \text{value} \rangle$ and store it

as element $\langle j, i, \text{value} \rangle$ of the transpose;

	row	col	value	a		b
$a[0]$	6	6	8	$(0, 0, 15)$	\rightarrow	$(0, 0, 15)$
[1]	0	0	15	$(0, 3, 22)$	\rightarrow	$(3, 0, 22)$
[2]	0	3	22	$(0, 5, -15)$	\rightarrow	$(5, 0, -15)$
[3]	0	5	-15	$(1, 1, 11)$	\rightarrow	$(1, 1, 11)$
[4]	1	1	11			data movement
[5]	1	2	3	$(1, 2, 3)$	\rightarrow	$(2, 1, 3)$
[6]	2	3	-6			data movement
[7]	4	0	91			
[8]	5	2	28			
					...	

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.

방법 2를 구현한 프로그램

```
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 : $O(\text{columns} \cdot \text{elements})$

만약 $\text{elements} = \text{rows} \cdot \text{columns}$ 이라면,
 $O(\text{columns} \cdot \text{elements}) = O(\text{columns}^2 \cdot \text{rows})$

방법 3. 각 열에 대해, 그 열의 원소 수를 계산한 후, 각 열의 전치 시작위치를 구함으로써, 각 원소에 대해 차례로 그 원소의 전치위치에 배치한다.

Fast transpose of a sparse matrix

	row	col	value		row	col	value	
<i>a</i> [0]	6	6	8		<i>b</i> [0]	6	6	8
[1]	0	0	15	→	[1]	0	0	15
[2]	0	3	22		[2]	0	4	91
[3]	0	5	-15	→	[3]	1	1	11
[4]	1	1	11	→	[4]	2	1	3
[5]	1	2	3		[5]	2	5	28
[6]	2	3	-6	→	[6]	3	0	22
[7]	4	0	91		[7]	3	2	-6
[8]	5	2	28	→	[8]	5	0	-15

calculation of
rowTerms

	[0]	[1]	[2]	[3]	[4]	[5]
<i>rowTerms</i> =	2	1	2	2	0	1
<i>startingPos</i> =	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;
        }
    }
}

```

Time complexity : $O(\text{columns} + \text{elements})$

만약, $\text{elements} = \text{columns} \cdot \text{rows}$ 라면, $O(\text{columns} + \text{elements}) = O(\text{columns} \cdot \text{rows})$ 가 됨

Program 2.9: Fast transpose of a sparse matrix

2.5.4 Matrix Multiplication

Definition: Given A and B where A is $m \times n$ and B is $n \times p$, the product matrix D has dimension $m \times p$. Its $\langle i, j \rangle$ element is :

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

for $0 \leq i < m$ and $0 \leq j < p$. \square

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;  
    }  
}
```

행렬 곱. A의 각 행 x에 대해, B의 각 열 y에 대해, $A[x][z] * B[z][y]$ 를 구하여 $D[x][y]$ 에 대입한다.

This algorithm takes **$O(\text{rowsA} \cdot \text{colsA} \cdot \text{colsB})$** 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 $j = 0, 1, \dots, colsB - 1$.
- 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
a b c d
e f g h
p q r s

B
x y z
m n o
v w t
u k j

newB = B^T
x m v u
y n w k
z o t j

행렬 곱. A의 각 행 x에 대해, B의 각 열 y에 대해, $A[x][z] * B[z][y]$ 를 구하여 $D[x][z]$ 에 대입한다.

↓
1 0 0 a
2 0 1 b
3 0 2 c
4 0 3 d
5 1 0 e
6 1 1 f
7 1 2 g
8 1 3 h
9 2 0 p
10 2 1 q
11 2 2 r
12 2 3 s

0 0 x
0 1 y
0 2 z
1 0 m
...

↓
1 0 0 x
2 0 1 m
3 0 2 v
4 0 3 u
5 1 0 y
6 1 1 n
7 1 2 w
8 1 3 k
9 2 0 z
10 2 1 o
11 2 2 t
12 2 3 j

3 0 u
3 1 k
3 2 j

D

$A[1:4] \leftrightarrow \text{newB}[1:4]$
 $A[5:8] \leftrightarrow \text{newB}[1:4]$
 $A[9:12] \leftrightarrow \text{newB}[1:4]$

$A[1:4] \leftrightarrow \text{newB}[5:8]$
 $A[5:8] \leftrightarrow \text{newB}[5:8]$
 $A[9:12] \leftrightarrow \text{newB}[5:8]$

$A[1:4] \leftrightarrow \text{newB}[9:12]$
 $A[5:8] \leftrightarrow \text{newB}[9:12]$
 $A[9:12] \leftrightarrow \text{newB}[9:12]$

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

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

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 ($a[totalA+1].row = rowsA;$) and *newB* ($newB[totalB+1].row = colsB;$). 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; // 새 행의 시작 index
    int d_row = a[1].row; // 새 행의 값 설정
    int 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;
}

```

Program 2.10: Sparse matrix multiplication

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

$D[\text{row}, \text{column}] = \text{SUM} (A[\text{row}, x] * B[y, \text{column}])$ if $x == y$, where $0 \leq x$ and $y \leq n-1$.
 $D[\text{row}, \text{column}] = \text{SUM} (A[\text{row}, x] * \text{newB}[\text{column}, y])$ if $x == y$, where $0 \leq x$ and $y \leq n-1$.

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.

```

for (i = 1; i <= totalA; ) {
    d_column = newB[1].row; // 새 행에 대한 첫 열의 값 설정
    for (j = 1; j <= totalB+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != d_row) {
            // 현행 현열 계산 끝, 행 넘침
            storeSum(d, &totalD, d_row, d_column, &sum);
            i = rowBegin; // 현 행의 시작 index부터 다시
            for (; newB[j].row == d_column; j++) ;
            d_column = newB[j].row; // 다음 열을 새로운 현 열의 값으로
        } else if (newB[j].row != d_column) {
            // 현행 현열 계산 끝, 열 넘침
            storeSum(d, &totalD, d_row, d_column, &sum);
            i = rowBegin; // 현 행의 시작 index부터 다시
            d_column = newB[j].row; // 다음 열을 새로운 현 열의 값으로
        } else {
            // 현행 현열 계산 중
            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++;
            }
        }
    }
}
// 현행 계산 끝났음.
for (; a[i].row == d_row; i++) ;
rowBegin = i; // 새 행의 시작 index
d_row = a[i].row; // 새 행의 값 설정
} /* end of for i<=totalA */
d[0].row = rowsA; d[0].col = colsB; d[0].value = totalD;

```

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

```
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+1 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 function This algorithm takes $O(\text{rowsA} \cdot \text{colsA} \cdot \text{colsB})$ 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. $O(\text{cols}B + \text{total}B)$.
- The outer for loop is executed $\text{total}A$ 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 $\text{total}B + 1$.
- 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 $\text{cols}B + \text{cols}B \cdot \text{termsRow} + \text{total}B$.
- The time for the inner loop during the multiplication of the current row is $O(\text{cols}B \cdot \text{termsRow} + \text{total}B)$, and the time to advance to the next row is $O(\text{termsRow})$. Thus, the time for one iteration of the outer for loop is $O(\text{cols}B \cdot \text{termsRow} + \text{total}B)$.
- The overall time for this loop is:

$$O(\sum_{\text{row}} (\text{cols}B \cdot \text{termsRow} + \text{total}B)) = O(\text{cols}B \cdot \text{total}A + \text{rows}A \cdot \text{total}B).$$

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 $[upper_0][upper_1] \cdot \cdot \cdot [upper_{n-1}]$, 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

- $A[2][3][2][2] \quad \leftarrow A[\text{upper}_0][\text{upper}_1] \dots [\text{upper}_{n-1}]$
- $2*3*2*2 = 24$ elements
- stored as $A[0][0][0][0], A[0][0][0][1], \dots, A[2][3][2][0], A[2][3][2][1]$
- indices are increasing : lexicographic order
- translate to locations in the one-dim array

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

- $A[0][0][0][0] \rightarrow \text{position } \alpha$
- $A[0][0][0][1] \rightarrow \text{position } \alpha + 1$
- $A[1][2][1][1] \rightarrow \text{position } \alpha + 23$
- $A[i][j][k][m] \rightarrow \text{position: } \alpha + i*\text{upper}_1*\text{upper}_2*\text{upper}_3$
 $\quad \quad \quad + j*\text{upper}_2*\text{upper}_3$
 $\quad \quad \quad + k * \text{upper}_3$
 $\quad \quad \quad + m$
- $A[i_0][0][0]\dots[0] \rightarrow \alpha + i_0 \text{ upper}_1 \text{ upper}_2 \dots \text{upper}_{n-1}$
- $A[i_0][i_1][0]\dots[0] \rightarrow \alpha + i_0 \text{ upper}_1 \text{ upper}_2 \dots \text{upper}_{n-1}$
 $\quad \quad \quad + i_1 \text{ upper}_2 \text{ upper}_3 \dots \text{upper}_{n-1}$

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 $A[u_1][u_2]$

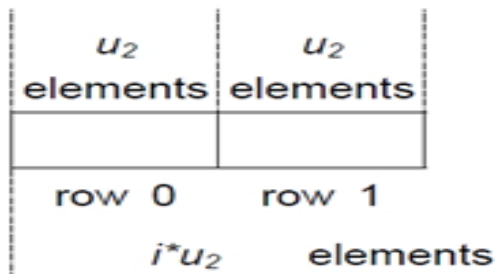
let a be the address of $A[0][0]$

$A[i][0] : a + i * u_2$

$A[i][j] : a + i * u_2 + j$

	col 0	col 1	col u_2-1
row 0	X	X	X
row 1	X	X	X
row 2	X	X	X
row u_1-1	X	X	X

(a)



(b)

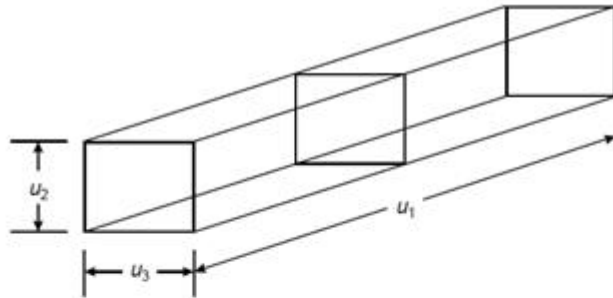
Figure: Sequential representation of one-dimensional array

three-dim array $A[u_1][u_2][u_3]$

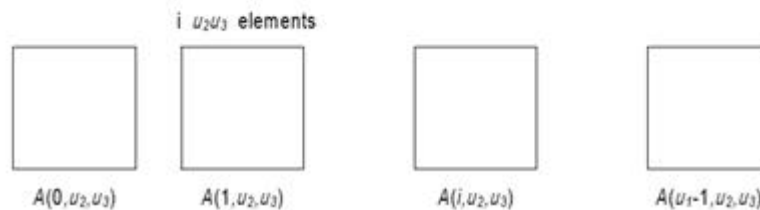
the address of $A[0][0][0] : \alpha$

$A[i][0][0] : \alpha + iu_2u_3$

$A[i][j][k] : \alpha + iu_2u_3 + ju_3 + k$



(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 $A[i_0][i_1] \dots [i_{n-1}]$ is:

$$\begin{aligned}
 & \alpha + i_0 \text{upper}_1 \text{upper}_2 \dots \text{upper}_{n-1} \\
 & \quad + i_1 \text{upper}_2 \text{upper}_3 \dots \text{upper}_{n-1} \\
 & \quad + i_2 \text{upper}_3 \text{upper}_4 \dots \text{upper}_{n-1} \\
 & \quad \cdot \\
 & \quad \cdot \\
 & \quad \cdot \\
 & \quad + i_{n-2} \text{upper}_{n-1} \\
 & \quad + i_{n-1} \\
 & = \alpha + \sum_{j=0}^{n-1} i_j a_j \text{ where: } \begin{cases} a_j = \prod_{k=j+1}^{n-1} \text{upper}_k & 0 \leq j < n-1 \\ a_{n-1} = 1 \end{cases}
 \end{aligned}$$

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.

Matrix (행렬) 문제 [4 점]

아래 6 X 6의 두 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

위 두 행렬 A와 B를 대상으로 다음을 수행하시오. 단, 앞의 희소행렬과 같은 방법이 아닌 일반 행렬로 다루어야 한다. 즉, readMatrix는 모든 엔트리들의 입력을 읽어야 하며 2차원 배열 A와 B에 그 값들을 저장하여야 하며, printMatrix는 모든 엔트리들을 출력하여야 하고, madd와 mmult는 2차원 배열 A와 B에 대해 직접 연산을 수행하도록 작성되어야 한다.

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

2.7 Strings

2.7.1 the Abstract Data type

As an ADT, we define a string to have the form, $S = s_0, \dots, s_{n-1}$, 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 $s, t \in \text{String}$, $i, j, m \in \text{non-negative integers}$

String Null(m) ::= **return** a string whose maximum length is m characters, but is initially set to *NULL*
We write *NULL* as "".

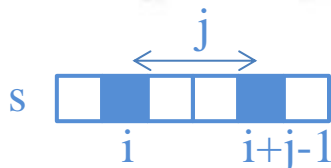
Integer Compare(s, t) ::= **if** s equals t **return** 0
else if s precedes t **return** -1
else return +1

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** (($j > 0$) && ($i + j - 1 < \text{Length}(s)$))
return the string containing the characters of s at positions $i, i + 1, \dots, i + j - 1$.
else return *NULL*.



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.

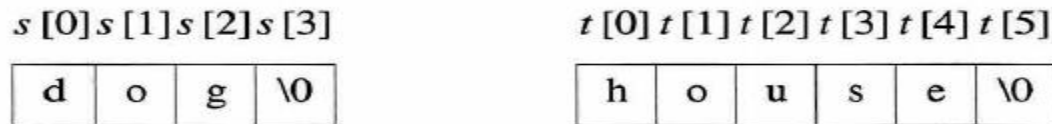


Figure 2.9: String representation in C

<아래 형태 모두 가능>

<code>char s[MAX_SIZE] = {"dog"};</code>	<code>char s[MAX_SIZE] = "dog";</code>	← 배열 s의 크기 = MAX_SIZE로 고정됨.
<code>char s[] = {"dog"};</code>	<code>char s[] = "dog";</code>	← 배열 s의 크기
<code>const char *s = "dog";</code>		←

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

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

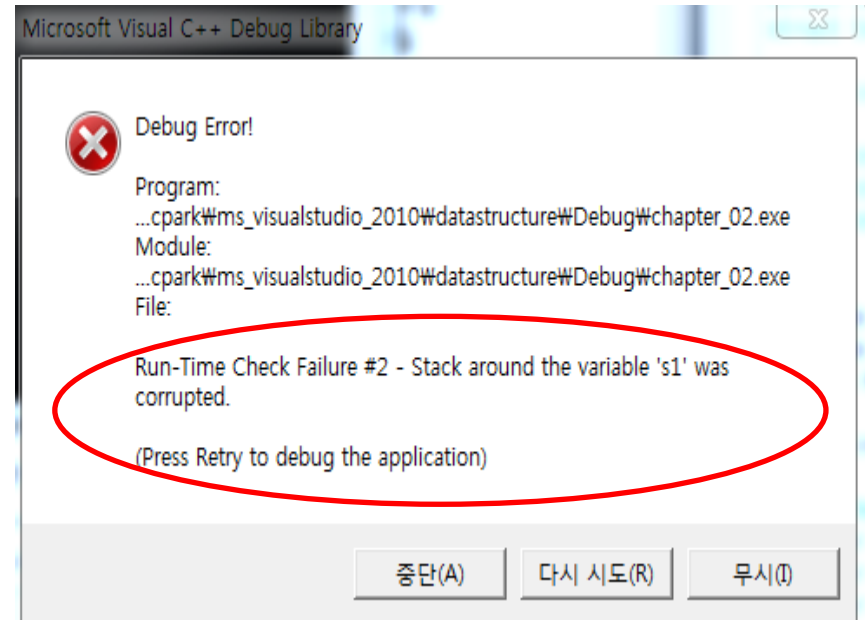
```
char t [] = {"house"}
```

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

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 string1[MAX_SIZE], *s = string1;
char string2[MAX_SIZE], *t = string2;
```

The call **strnins(s, t, 0)** in the next page 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

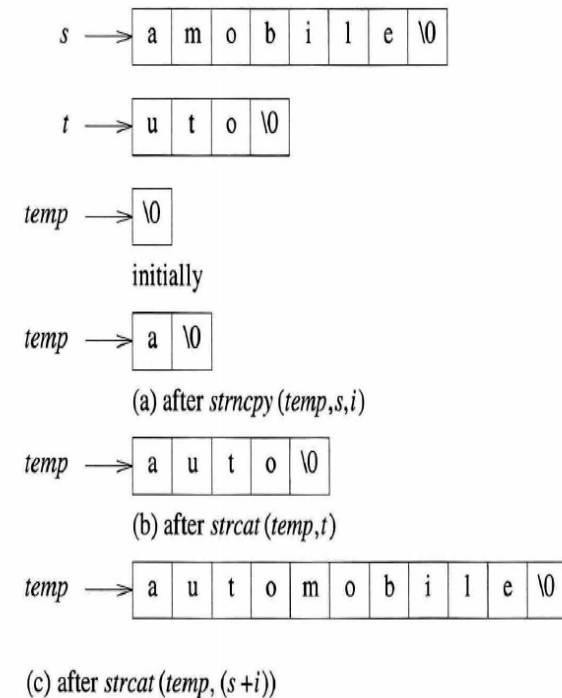


Figure 2.10: String insertion example

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

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

Program : Brute-force pattern matching

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

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 $O(mn)$

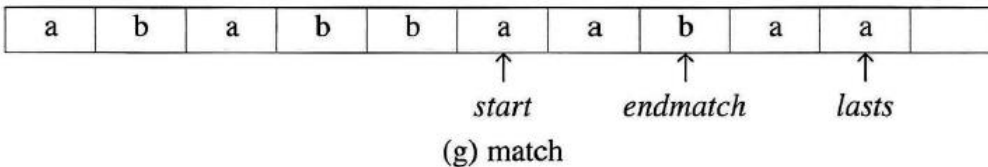
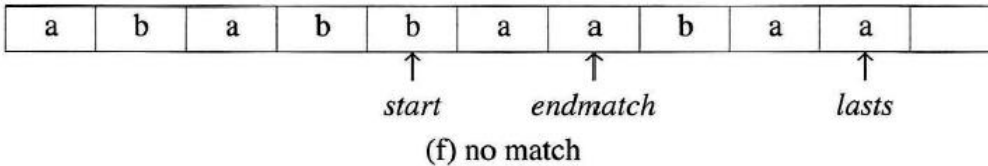
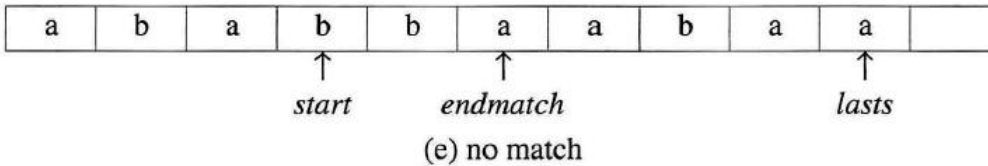
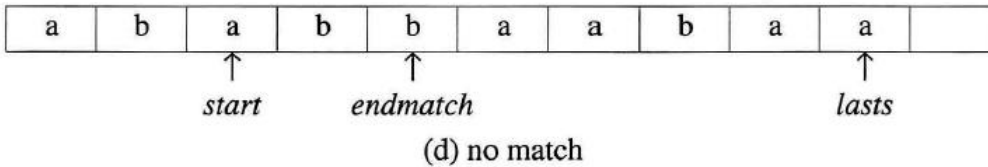
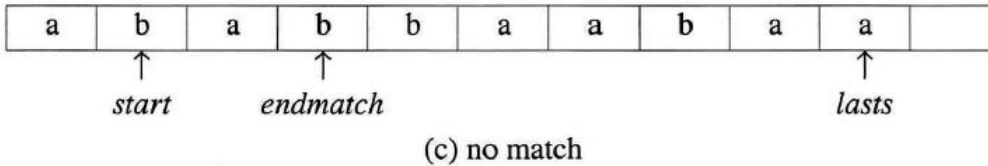
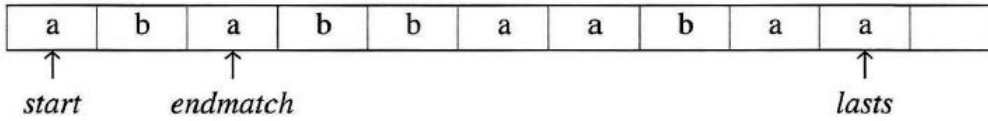
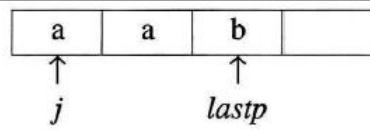
Simple algorithm

```
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

< nfind 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 length of *string* is m and the length of *pat* is n .

Time complexity:

$O(m)$ – linear, for the best case

- string : aa...a

- pattern: a...ab

$O(m \cdot n)$, for the worst case

- String : bbbbbbbbbbb...bbbbbb

- pattern: bb...bab

Figure 2.11: Simulation of *nfind*

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

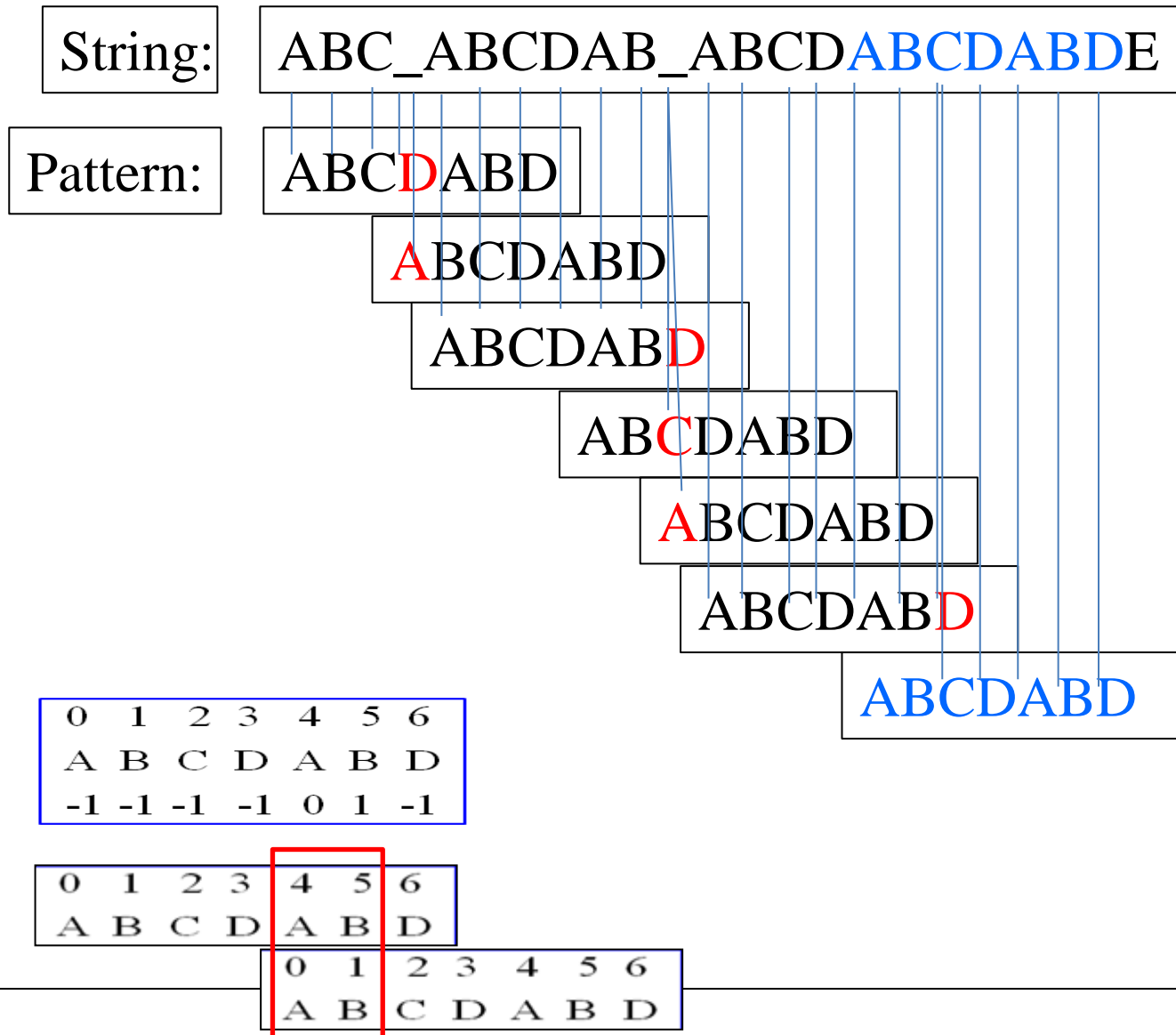
Ideally, we would like an algorithm that works in $O(\text{strlen}(\text{string}) + \text{strlen}(\text{pat}))$ time. This is optimal for this problem as in the worst case it is necessary to look at all characters in the pattern and string at least once.

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**

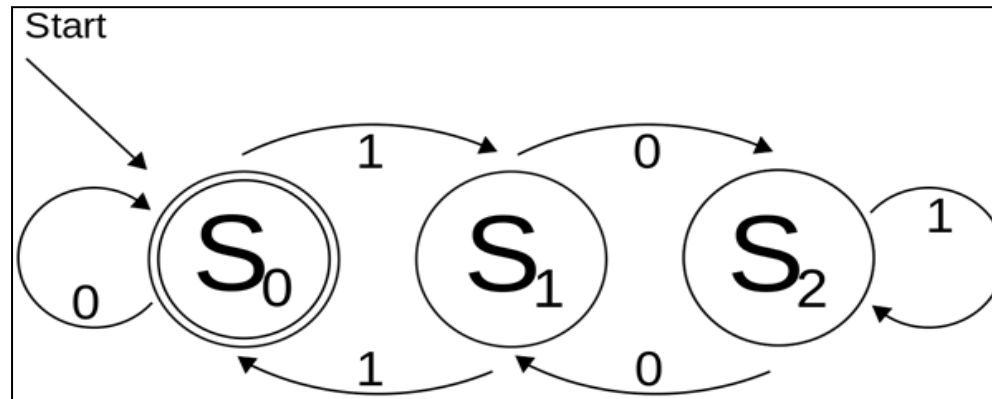
<https://www.cs.princeton.edu/~rs/AlgsDS07/21PatternMatching.pdf>

Basic Idea



Deterministic finite automaton (결정 유한 오토마틴)

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.

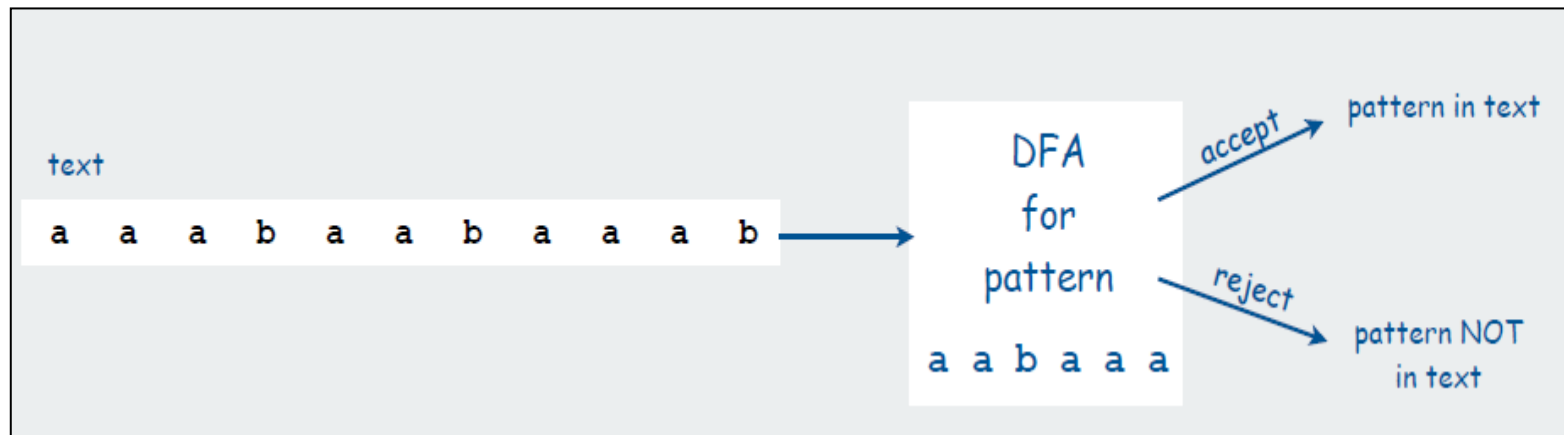
Knuth-Morris-Pratt (KMP) exact pattern-matching algorithm

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

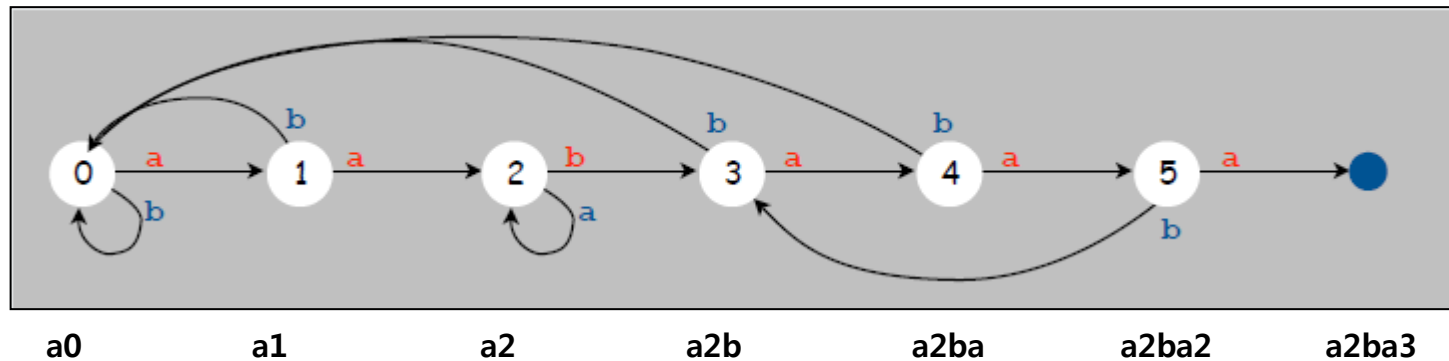


Knuth-Morris-Pratt DFA example

One state for each pattern character

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

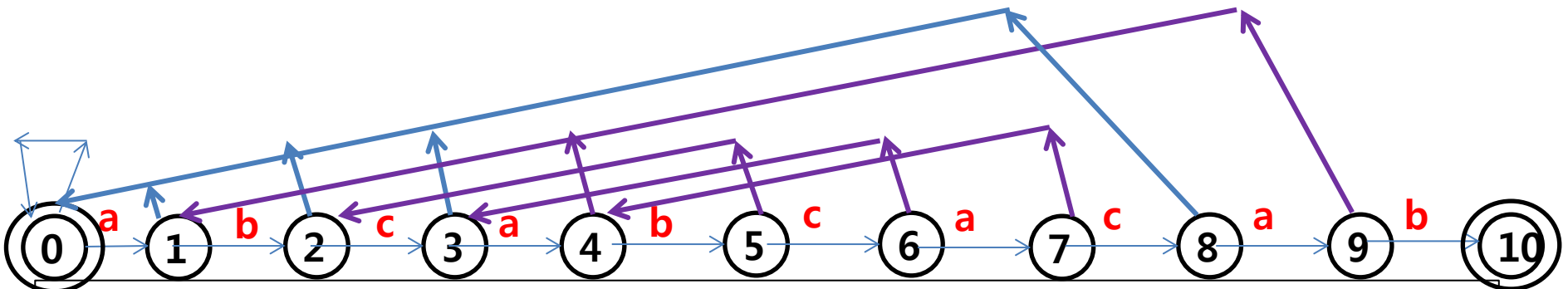
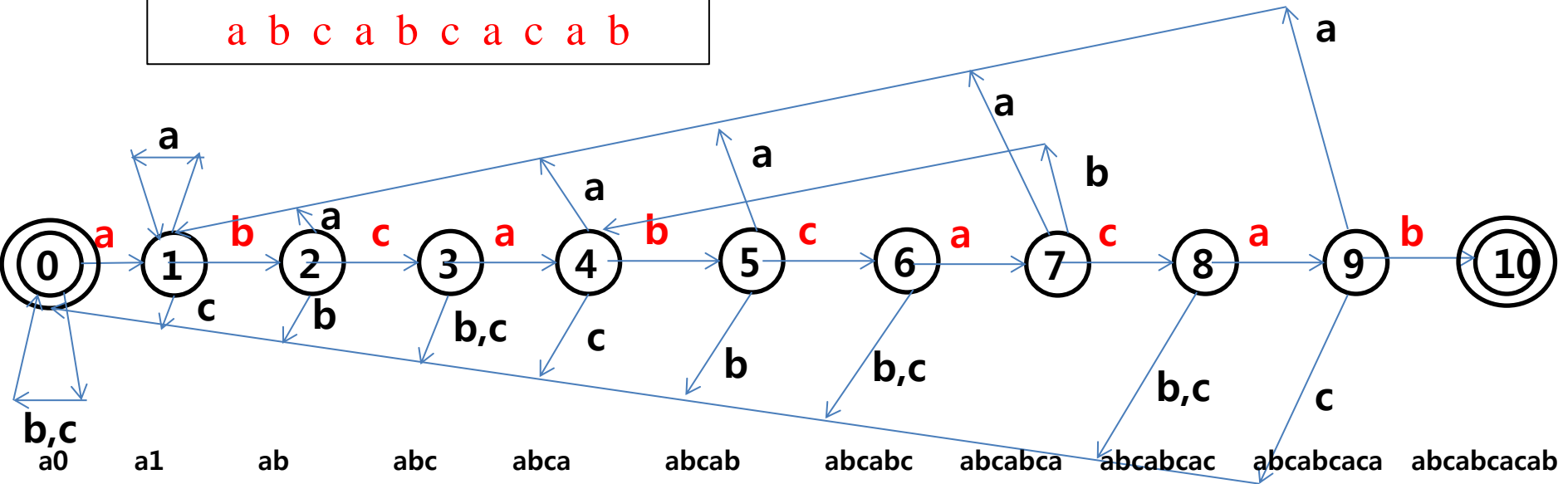
DFA
for
pattern
a a b a a a



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.

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

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

0	1	2	3	4	5	6	7	8	9
<i>a</i>	<i>b</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>

0	1	2	3	4	5	6	7	8	9
<i>a</i>	<i>b</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>

0	1	2	3	4	5	6	7	8	9
<i>a</i>	<i>b</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>

0	1	2	3	4	5	6	7	8	9
<i>a</i>	<i>b</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>

Using their example, suppose $pat = 'abcabcacab'$.

Let $s = s_0 s_1 s_2 \cdot \cdot \cdot s_{m-1}$ 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 $s_{i+1} \neq b$ 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 have the situation:

$s = \quad ' - \textcolor{blue}{a} \textcolor{blue}{b} ? ? ? ? '$

$pat = \quad '\textcolor{blue}{a} \textcolor{blue}{b} c a b c a c a b'$

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

- 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 $p = p_0 p_1 \dots p_{n-1}$ is a pattern,
then its **failure function**, f , is defined as:

$$f(j) = \begin{cases} \text{largest } i < j \text{ such that } p_0 p_1 \dots p_i = p_{j-i} p_{j-i+1} \dots p_j & \text{if such an } i \geq 0 \text{ exists} \\ -1 & \text{otherwise} \end{cases}$$

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

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

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

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 $s[i-j]=p[0], \dots, s[i-1]=p[j-1]$,

Upon $s[i] \neq p[j]$,

Matching may be resumed by comparing $s[i]$ and $p[f[j-1]+1]$ if $j \neq 0$.

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

\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}	

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

0	1	2	3	4	5	6
<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>a</i>	<i>b</i>	<i>d</i>
-1	-1	-1	-1	0	1	-1

0	1	2	3	4	5	6
<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>a</i>	<i>b</i>	<i>d</i>

0	1	2	3	4	5	6	7	8	9
<i>a</i>	<i>b</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>
-1	-1	-1	0	1	2	3	-1	0	1

0	1	2	3	4	5	6	7	8	9
<i>a</i>	<i>b</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>

0	1	2	3	4	5	6	7	8	9
<i>a</i>	<i>b</i>	<i>c</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>a</i>	<i>c</i>	<i>a</i>	<i>b</i>

0	1	2	3	4	5	6	7	8	9	0	1	2
<i>a</i>	<i>b</i>	<i>k</i>	<i>a</i>	<i>b</i>	<i>m</i>	<i>a</i>	<i>b</i>	<i>k</i>	<i>a</i>	<i>b</i>	<i>k</i>	<i>a</i>
-1	-1	-1	0	1	-1	0	1	2	3	4	2	3

0	1	2	3	4	5	6	7	8	9	0	1	2
<i>a</i>	<i>b</i>	<i>k</i>	<i>a</i>	<i>b</i>	<i>k</i>	<i>a</i>	<i>b</i>	<i>k</i>	<i>a</i>	<i>b</i>	<i>k</i>	<i>a</i>

0	1	2	3	4	5	6	7	8	9	0	1	2
<i>a</i>	<i>b</i>	<i>k</i>	<i>a</i>	<i>b</i>	<i>k</i>	<i>a</i>	<i>b</i>	<i>k</i>	<i>a</i>	<i>b</i>	<i>k</i>	<i>a</i>

Got from Wikipedia

```

      1           2
i: 01234567890123456789012
S: ABC ABCDAB ABCDABCDABDE
P: ABCDABD
j: 0123456
    ABCDABD
    0123456
  
```

- $S[i=3] \neq P[j=3]$.
 - $j \neq 0$, so, **resume match** $S[i=3]$ and $P[j=f[j-1]+1]=P[j=0]$.
 - $S[i=3] \neq P[j=0]$.
- 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: 01234567890123456789012
S: ABC ABCDAB ABCDABCDABDE
P:      ABCDABD
j:      0123456
  
```

- $S[i=10] \neq P[j=6]$.
- $j \neq 0$, so, **resume match** $S[i=10]$ and $P[j=f[j-1]+1]=P[j=2]$.

```

      1           2
i: 01234567890123456789012
S: ABC ABCDAB ABCDABCDABDE
P:      ABCDABD
j:      0123456
    ABCDABD
    0123456
  
```

- $S[i=10] \neq P[j=2]$.
 - $j \neq 0$, so, **resume match** $S[i=10]$ and $P[j=f[j-1]+1]=P[j=0]$.
 - $S[i=10] \neq P[j=0]$.
- Advance i (now 11) but j is still 0.

```

      1           2
i: 01234567890123456789012
S: ABC ABCDAB ABCDABCDABDE
P:      ABCDABD
j:      0123456
  
```

- $S[i=17] \neq P[j=6]$.
- $j \neq 0$, so, **resume match** $S[i=17]$ and $P[j=f[j-1]+1]=P[j=2]$.

```

      1           2
i: 01234567890123456789012
S: ABC ABCDAB ABCDABCDABDE
P:      ABCDABD
j:      0123456
  
```

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

<i>j</i>	0	1	2	3	4	5	6
<i>pat</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>a</i>	<i>b</i>	<i>d</i>
<i>f</i>	-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];
```

0	1	2	3	4	5	6	7	8	9	0	1	2
a	b	k	a	b	m	a	b	k	a	b	k	a
-1	-1	-1	0	1	-1	0	1	2	3	4	2	3

0	1	2	3	4	5	6	7	8	9	0	1	2
a	b	k	a	b	k	a	b	k	a	b	k	a

0	1	2	3	4	5	6	7	8	9	0	1	2
a	b	k	a	b	m	a	b	k	a	b	k	a

/* 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 *pattern)
{ /* Knuth, Morris, Pratt string matching algorithm */
    int i = 0, j = 0;
    int length_str= strlen(string);
    int length_pat = strlen(pattern);

    while ( i < length_str && j < length_pat ) {
        if (string[i] == pattern[j]) { i++; j++; }
        else {
            if (j == 0) i++;
            else j = failure[j-1]+1;
        }
    }
    return ( (j == length_pat) ? (i-length_pat), -1);
}

```

Program 2.14: Knuth, Morris, Pratt pattern matching algorithm

Time complexity : $O(\text{strlen}(\text{string}))$

Time complexity of fail and pmatch: $O(\text{strlen}(\text{string}) + \text{strlen}(\text{pattern}))$

Pattern Matching 문제 1/2 [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.

```
text를 담은 파일명을 입력하시오:sample.txt
검색할 단어를 입력하시오:program
```

```
288 programming development environments has
488 programming systems areas of computer sc
821 programs using ANSI C. ANSI C, adopted i
903 programming language by permitting a var
1113 programs.
```

```
For those instructors who hav
1793 program readability, we define macros su
1964 program termination. The discussion of s
계속하려면 아무 키나 누르십시오 . . .
```

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 reliability 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.

Pattern Matching 문제 2/2 [4 점]

우리는 한글, 영어, 한글 + 영어로 된 내용에 대해 한글, 영어, 한글 + 영어로 검색하고자 한다. 다음 페이지의 내용은 데이터베이스 교재의 번역 본의 일부이다. 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 “최적화”를 입력한다.
3. Simple algorithm으로 그 단어를 그 text에서 찾으시오.
4. Fast algorithm by Knuth, Morris, Pratt 으로 그 단어를 그 text에서 찾으시오.

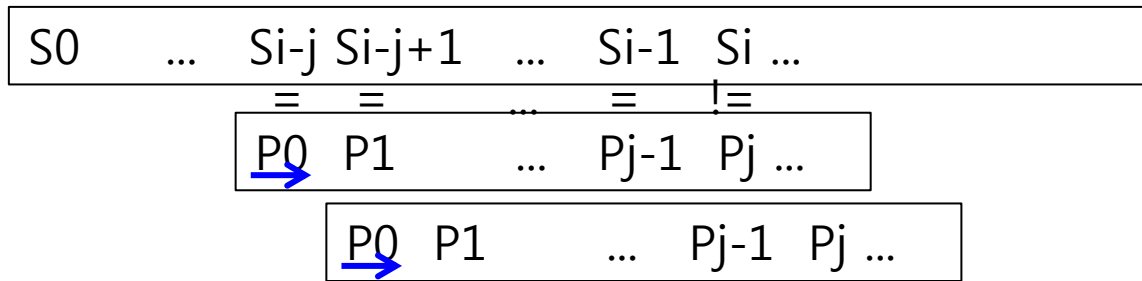
3과 4의 수행에서 그 단어가 나오는 모든 곳을 찾아야 하며 각 발견마다 그 단어를 포함하여 모두 40 자를 출력하여야 한다. 아래 결과는 내가 수행한 결과이다.

이 장에서는 고수준의 질의를 처리하기 위해서 DBMS가 내부적으로 사용하는 기술들을 논의한다. SQL과 같은 고수준의 질의어로 표현된 질의는 어휘 분석과 구문 분석을 거친 뒤 검증 과정을 거쳐야 한다. 스캐너(scanner)는 질의에 나타나는 질의 토큰들, 예를 들어 SQL 키워드, 애트리뷰트 이름, 릴레이션 이름들을 식별한다. 파서(parser)는 질의의 구문이 질의어의 문법 규칙에 맞게 작성되었는지 검사한다. 또한 질의는 질의에 사용된 모든 애트리뷰트와 릴레이션의 이름들이 질의의 대상이 되는 특정 데이터베이스의 스키마에서 유효하고 의미적으로 합당한지를 검사함으로써 검증해야 한다. 그리고 나서 질의의 내부 표현을 생성하는데 일반적으로 질의 트리(query tree)라고 하는 트리 자료구조로 나타낸다. 질의를 질의 그래프(query graph)라고 하는 일반적으로 방향성 비사이클 그래프라는 그래프 자료구조를 이용하여 표현할 수도 있다. 그 다음에 DBMS는 데이터베이스 파일에서 질의 결과를 검색하기 위해 실행 전략(execution strategy) 또는 질의 계획(query plan)을 만들어야 한다. 하나의 질의는 다수의 가능한 실행 전략을 가질 수 있으며, 그 질의를 처리하기 위해 하나의 적절한 실행 전략을 선택하는 과정을 질의 **최적화**(query optimization)라고 한다.

질의 **최적화**에 대한 상세한 논의는 다음 장으로 미룬다. 이 장에서는 어떻게 질의가 처리되고 그 질의 내부의 개별 연산들을 수행하기 위해 어떤 알고리즘들이 사용되는 지를 주로 집중한다. 그림 17.1은 고수준 질의를 처리하는 여러 단계를 보여준다. 질의 **최적화**기(query optimizer) 모듈은 좋은 실행 계획을 만들며, 코드 생성기(code generator)는 그 계획을 실행하기 위한 코드를 생성하는 작업을 수행한다. 런타임 데이터베이스 처리기(runtime database processor)는 컴파일되었거나 인터프리트된 질의 코드를 수행(실행)하여 질의 결과를 생성한다. 만약 런타임 오류가 발생하면 런타임 데이터베이스 처리기가 오류 메시지를 발생시킨다.

최적화라는 용어는 실제로는 잘못된 명칭이다. 왜냐하면 몇몇 경우에는 선택한 실행 계획이 최적의(또는 절대적 최고의) 전략이 아닐 수 있기 때문이다. 그것은 그 질의를 실행하기에 적절히 효율적이거나 이용 가능한 최선의 전략일 뿐이다. 가장 단순한 형태의 질의들을 제외하고는 최적의 전략을 찾는 작업은 일반적으로 너무 많은 시간을 소모한다. 또한 최적의 질의 실행 전략을 찾고자 하는 작업은 테이블의 크기와 칼럼 값과 같은 것들의 분포에 대한 정밀하고 상세한 정보를 필요로 한다. 이런 정보가 DBMS의 카탈로그에서 항상 제공된다고 보장할 수는 없다. 더욱이, 예상되는 결과의 크기와 같은 추가 정보는 그 질의의 술어에 기초하여 얻어져야 한다. 그러므로 질의 **최적화**라는 용어보다 좋은 실행 전략 계획(planning of a good execution strategy)이라는 용어가 더 적합할 것이다.

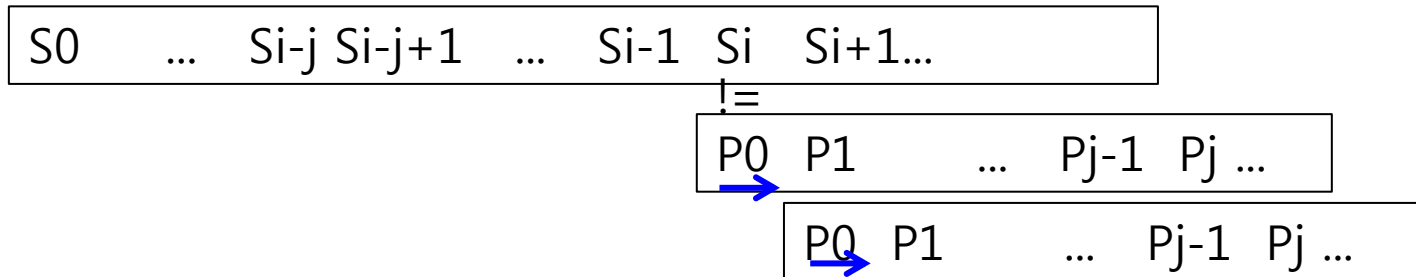
- Brute-Force Algorithm



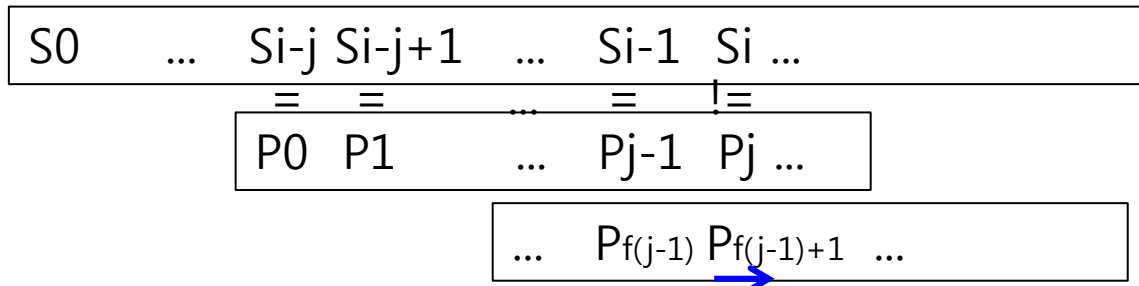
If $s_i \neq p_j$ then matching may be resumed by comparing s_{i-j+1} and p_0 .

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

Case $j = 0$



Case $j \neq 0$



If a partial match is found such that $s_{i-j} \dots s_{i-1} = p_0 p_1 \dots p_{j-1}$ 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 .

A B C v A B C D A B v A B C D A B C D A B D E

!=

A B C D A B D

!=

A B C D A B D

!=

A B C D A B D

!=

A B C D A B D

!=

A B C D A B D

!=

A B C D A B D

A B C D A B D

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