

Chapter 2 : ARRAYS AND STRUCTURES

A decorative L-shaped line consisting of a vertical segment on the left and a horizontal segment extending to the right, both in a dark blue color, positioned below the chapter title.

2.1 ARRAYS

2.1.1 The Abstract Data Type

- An *array* is usually viewed as "a consecutive set of memory locations" which is a usual implementation.
- An *array* as an ADT is a set of pairs, $\langle index, value \rangle$, such that each index that is defined has a value associated with it.
- Aside from creating a new array, most languages provide only two standard operations for arrays,
 - (1) *retrieving a value*
 - (2) *storing a value*

■ <Abstract Data Type *Array*>

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 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 $A \in Array, i \in index, x \in item, j, size \in integer$

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 $\langle i, x \rangle$ has been inserted
else return error.

end *Array*

2.1.2 Arrays in C

- Declaration of one-dimensional arrays in C :

```
int list[5], *plist[5];
```

- Memory allocation of arrays :

Variable	Memory address
list[0]	base address = a
list[1]	a + sizeof(int)
list[2]	a + 2·sizeof(int)
list[3]	a + 3·sizeof(int)
list[4]	a + 4·sizeof(int)

C interprets list[i] as a pointer to an integer.

- Observe the difference between a declaration such as

```
int *list1;  
and  
int list2[5];
```

Variables *list1* and *list2* are both pointers to an integer type object.
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*].

Thus, (*list2*+*i*) equals &*list2*[*i*].

And, *(*list2*+*i*) equals *list2*[*i*].

■ [Program 2.1] Example array program

```
#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;
}
```

- When `sum` is invoked, `input = &input[0]` is copied into a temporary location and associated with the formal parameter `list`.
- When `list[i]` occurs on the right-hand side of '=' in an assignment statement, a dereference takes place and the value pointed at by `(list+i)` is returned.
- If `list[i]` appears on the left-hand side of '=', then the value produced on the right-hand side is stored in the location `(list+i)`.

■ **Example 2.1 [One-dimensional array addressing]**

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


A function that prints out both the address of the i th element of this array and the value found at this address.

■ **[Program 2.2]**

```
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("%8u%5d\n", ptr + i, *(ptr + i));
    printf("\n");
}
```


- **[Figure 2.1] One-dimensional array addressing**

sizeof(int)==4



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

2.2 DYNAMICALLY ALLOCATED ARRAYS

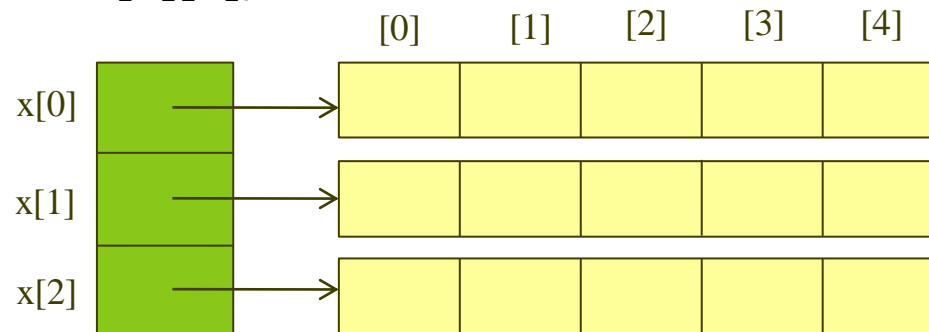
2.2.1 ONE-DIMENSIONAL ARRAYS

- If the user wishes to change array size, we have to change the definition of *MAX_SIZE* and recompile the program.
- A good solution to this problem is to defer this decision to run time and allocate the array when we have a good estimate of the required array size.

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

2.2.2 TWO-DIMENSIONAL ARRAYS

- A 2-D array is represented as a 1-D array in which each element is itself a 1-D array
- (e.g.) `int x[3][5];`



- A 3-D array is represented as a 1-D array in which each element is itself a 2-D array

■ [Program 2.3]

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

    /* get memory for row pointers */
    x = (int**)malloc(rows*sizeof(int*));

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

=> int x[rows][cols]

- `void* calloc(elt_count, elt_size)`
→ allocates a region of memory large enough to hold an array of *elt_count* elements, each of size *elt_size*, and the region of memory is set to zero
- `void* realloc(p, s)`
→ changes the size of memory block pointed at by *p* to *s*

2.3 STRUCTURES AND UNIONS

2.3.1 Structures

- A *structure* (called a *record* in many other programming language) is a collection of data items, where each item is identified as to its *type* and *name*.
- For example, the following declaration creates a variable whose name is *person* with three fields.

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

a name that is a character array

an integer value representing the age of the person

a float value representing the salary of the individual

- The structure member operator `.` is used to select a particular member of the structure.

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

- Creating new structure data types by using the *typedef* statement :

definition	<pre>typedef struct humanBeing { char name[10]; int age; float salary; };</pre>	<pre>typedef struct { char name[10]; int age; float salary; } humanBeing;</pre>
declaration	<pre>struct humanBeing person1;</pre>	<pre>humanBeing person2;</pre>

```
if (strcmp(person1.name, person2.name))  
    printf("The two people do not have the same name");  
else  
    printf("The two people have the same name");
```

→ only compares names

* Entire structure operation?

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

```
<person1 == person2>  
    if (humans_equal(person1, person2))  
        printf("The two human beings are the same");  
    else  
        printf("The two human beings are not the same");
```


■ [Program 2.4]

```
#define FALSE 0
```

```
#define TRUE 1
```

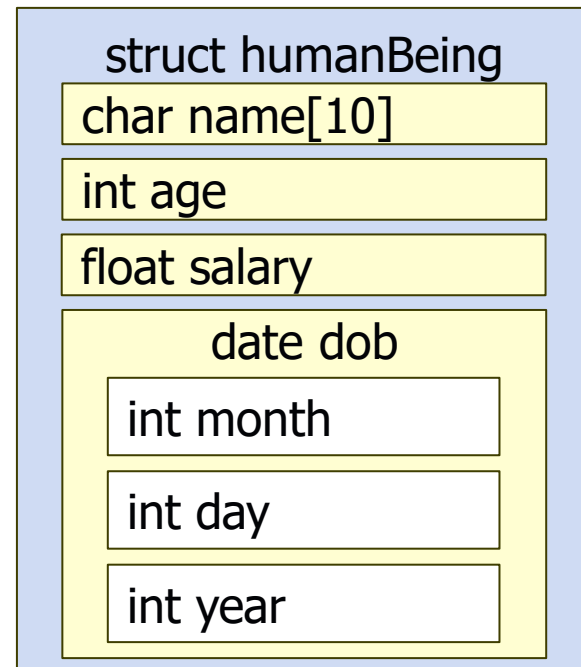
```
int humans_equal(humanBeing person1, humanBeing person2)
{
    /* return TRUE if person1 and person2 are the same human being
    otherwise return FALSE */
    if (strcmp(person1.name, person2.name))
        return FALSE;
    if (person1.age != person2.age)
        return FALSE;
    if (person1.salary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

■ A structure within a structure

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

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

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

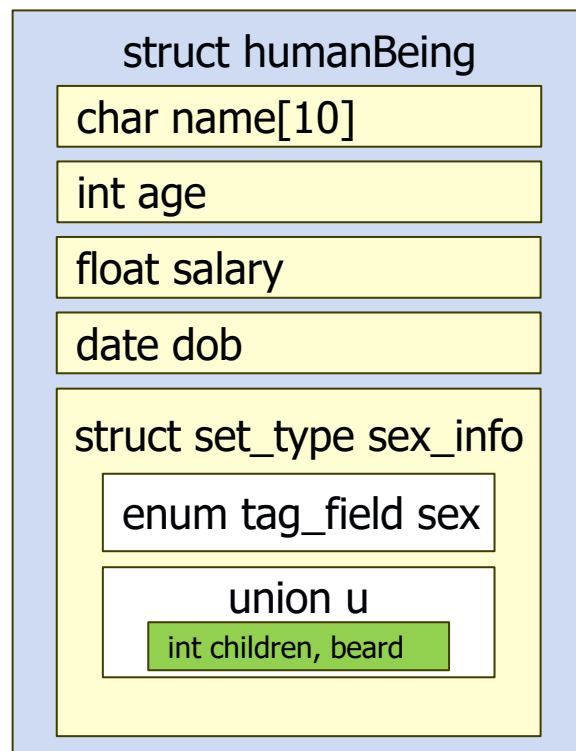


2.3.2 Unions

- Fields share their memory space → only one field of union is active at any given time

```
typedef struct sex_type {  
    enum tag_field {female, male} sex;  
    union {  
        int children;  
        int beard;  
    } u;  
};
```

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



```
struct humanBing person1, person2;
```

```
person1.sex_info.sex = male;
```

```
person1.sex_info.u.beard = FALSE;
```

```
person2.sex_info.sex = female;
```

```
person2.sex_info.u.children = 4;
```

2.3.3 Internal Implementation of Structures

- In most cases we need not be concerned with exactly how the C compiler will store the fields of structure in memory.
- Generally, the values will be stored in the same way using increasing address locations in the order specified in the structure definition.

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.
- Self-referential structure usually require dynamic storage management routines (*malloc* and *free*) to explicitly obtain and release memory.

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

```
struct list item1, item2, item3;
```

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

```
item1.link = &item2;  
item2.link = &item3;
```

item1

data	link

item2

data	link

item3

data	link

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

```
struct list item1, item2, item3;
```

```
item1.data = 'a';
```

```
item2.data = 'b';
```

```
item3.data = 'c';
```

```
item1.link = item2.link = item3.link = NULL;
```

```
item1.link = &item2;
```

```
item2.link = &item3;
```

item1

data	link
'a'	

item2

data	link
'b'	

item3

data	link
'c'	


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

```
struct list item1, item2, item3;
```

```
item1.data = 'a';
```

```
item2.data = 'b';
```

```
item3.data = 'c';
```

```
item1.link = item2.link = item3.link = NULL;
```

```
item1.link = &item2;
```

```
item2.link = &item3;
```

item1

data	link
'a'	NULL

item2

data	link
'b'	NULL

item3

data	link
'c'	NULL

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

```
struct list item1, item2, item3;
```

```
item1.data = 'a';
```

```
item2.data = 'b';
```

```
item3.data = 'c';
```

```
item1.link = item2.link = item3.link = NULL;
```

```
item1.link = &item2;
```

```
item2.link = &item3;
```

item1

data	link
'a'	&item2

item2

data	link
'b'	NULL

item3

data	link
'c'	NULL

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

```
struct list item1, item2, item3;
```

```
item1.data = 'a';
```

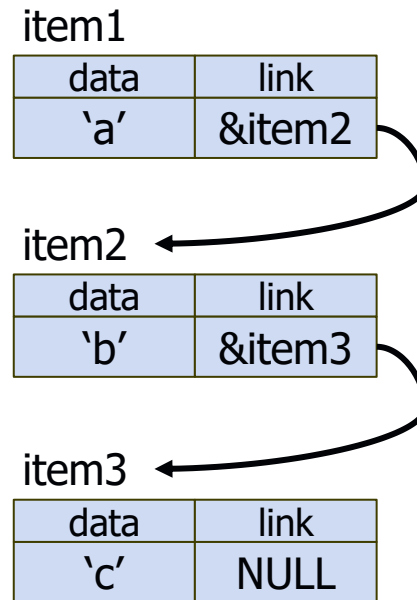
```
item2.data = 'b';
```

```
item3.data = 'c';
```

```
item1.link = item2.link = item3.link = NULL;
```

```
item1.link = &item2;
```

```
item2.link = &item3;
```



2.4 POLYNOMIALS

2.4.1 The Abstract Data Type

- Arrays are not only data structures in their own right, we can also use them to implement other abstract data types.
- One of the simplest and most commonly found data structures:
ordered list or *linear list*.
$$(item_0 , item_1 , \dots , item_{n-1})$$
- Examples :
 - Days of the week : (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday)
 - Values in a deck of cards :
(Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King)
 - Floors of the building :
(basement, lobby, mezzanine, first, second)etc.

- Possible operations on the ordered lists :
 - Finding the length, n , of a list.
 - Reading the items in a list from left to right (or right to left).
 - Retrieving the i th item from a list, $0 \leq i < n$.
 - Replacing the item in the i th position of a list, $0 \leq i < n$.
 - Inserting 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 $i+1, i+2, \dots, n$.

- Deleting an item from the i th position of a list, $0 \leq i < n$.

The items previously numbered $i+1, \dots, n-1$ become items numbered $i, i+1, \dots, n-2$.

- Implementations (ways to represent an ordered list) :
 - Sequential mapping
 - commonly represented as an **array**
 - storing $item_i, item_{i+1}$ into consecutive slots i and $i+1$ of the array.
(assuming the standard implementation of an array)
 - Retrieving an item, replacing an item, or finding the length of a list can be done in constant time.
 - requires data movement for insertion/deletion.
 - Nonsequential mapping: will be covered in Chapter 4.

- A *polynomial* (viewed from a mathematical perspective) is a sum of terms, where each term has a form ax^e , where x is a variable, a is the coefficient, and e is the exponent.

For example :

$$A(X) = 3x^{20} + 2x^5 + 4$$

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

Standard mathematical definitions for sum and product of polynomials.

$$\text{For } A(x) = \sum a_i x^i \text{ and } B(x) = \sum b_i x^i$$

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

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

■ [ADT 2.2] Abstract Data Type *Polynomial*

ADT *Polynomial* is

Objects : $p(x) = a_1x^{e_1} + \dots + a_nx^{e_n}$; a set of ordered pairs of $\langle e_i, a_i \rangle$
where a_i in *Coefficients* and e_i in *Exponents*, e_i are integers ≥ 0 .

Functions :

for all $poly, poly1, poly2 \in Polynomial, coef \in Coefficients, expon \in Exponents$

Polynomial Zero() $::=$ **return** the polynomial, $p(x)=0$

Boolean IsZero(*poly*) $::=$ **if** (*poly*) **return** FALSE
else return TRUE

Coefficients Coef(*poly*,*expon*) $::=$ **if** (*expon* \in *poly*) **return** its coefficient
else return zero

Exponent LeadExp(*poly*) $::=$ **return** the largest exponen in *poly*.

<i>Polynomial Attach</i> (<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 $\langle \textit{coef}, \textit{expon} \rangle$ inserted
<i>Polynomial Remove</i> (<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 SingleMult</i> (<i>poly</i> , <i>coef</i> , <i>expon</i>)	::= return the polynomial $\textit{poly} \cdot \textit{coef} \cdot x^{\textit{expon}}$
<i>Polynomial Add</i> (<i>poly1</i> , <i>poly2</i>)	::= return the polynomial $\textit{poly1} + \textit{poly2}$
<i>Polynomial Mult</i> (<i>poly1</i> , <i>poly2</i>)	::= return the polynomial $\textit{poly1} \cdot \textit{poly2}$
end <i>Polynomial</i>	

2.4.2 Polynomial Representation

■ [Program 2.5] Initial version of padd function

```
/* d = a + b, where a, b, and d are polynomials */
d = Zero();
While(!IsZero(a) && ! IsZero(b)) do {
    switch COMPARE(LeadExp(a), LeadExp(b)) {
        case -1 :    d = Attach(d, Coef(b, LeadExp(b)), LeadExp(b));
                     b = Remove(b, LeadExp(b));
                     break;

        case 0 :    sum = Coef(a, LeadExp(a)) + Coef(b, LeadExp(b));
                     if (sum)
                         Attach(d, sum, LeadExp(a));
                     a = Remove(a, LeadExp(a));
                     b = Remove(b, LeadExp(b));
                     break;

        case 1 :    d = Attach(d, Coef(a, LeadExp(a)), LeadExp(a));
                     a = Remove(a, LeadExp(a));
    }
}
insert any remaining terms of a or b into d
```

2.4.2 Polynomial Representation

■ [Program 2.5] Initial version of padd function

/* d = a + b, where a, b, and d are polynomials */

d = Zero();

While(!IsZero(a) && ! IsZero(b)) do {

 switch COMPARE(LeadExp(a), LeadExp(b)) { -> 1

 case -1 : d = Attach(d, Coef(b, LeadExp(b)), LeadExp(b));

 b = Remove(b, LeadExp(b));

 break;

 case 0 : sum = Coef(a, LeadExp(a)) + Coef(b, LeadExp(b));

 if (sum)

 Attach(d, sum, LeadExp(a));

 a = Remove(a, LeadExp(a));

 b = Remove(b, LeadExp(b));

 break;

 case 1 : d = Attach(d, Coef(a, LeadExp(a)), LeadExp(a));

 a = Remove(a, LeadExp(a));

 }

}

insert any remaining terms of a or b into d

$$a = 3x^3 - 2x^2 + x$$

$$b = 2x^2 + 5x + 1$$

$$d = 0$$

2.4.2 Polynomial Representation

■ [Program 2.5] Initial version of padd function

/* d = a + b, where a, b, and d are polynomials */

d = Zero();

While(!IsZero(a) && ! IsZero(b)) do {

switch COMPARE(LeadExp(a), LeadExp(b)) {

case -1 : d = Attach(d, Coef(b, LeadExp(b)), LeadExp(b));

b = Remove(b, LeadExp(b));

break;

case 0 : sum = Coef(a, LeadExp(a)) + Coef(b, LeadExp(b));

if (sum)

Attach(d, sum, LeadExp(a));

a = Remove(a, LeadExp(a));

b = Remove(b, LeadExp(b));

break;

case 1 : d = Attach(d, Coef(a, LeadExp(a)), LeadExp(a));

a = Remove(a, LeadExp(a));

}

}

insert any remaining terms of a or b into d

$$a = -2x^2 + x$$

$$b = 2x^2 + 5x + 1$$

$$d = 3x^3$$

2.4.2 Polynomial Representation

■ [Program 2.5] Initial version of padd function

/* d = a + b, where a, b, and d are polynomials */

d = Zero();

While(!IsZero(a) && ! IsZero(b)) do {

 switch COMPARE(LeadExp(a), LeadExp(b)) { -> 0

 case -1 : d = Attach(d, Coef(b, LeadExp(b)), LeadExp(b));

 b = Remove(b, LeadExp(b));

 break;

 case 0 : sum = Coef(a, LeadExp(a)) + Coef(b, LeadExp(b));

 if (sum)

 Attach(d, sum, LeadExp(a));

 a = Remove(a, LeadExp(a));

 b = Remove(b, LeadExp(b));

 break;

 case 1 : d = Attach(d, Coef(a, LeadExp(a)), LeadExp(a));

 a = Remove(a, LeadExp(a));

 }

}

insert any remaining terms of a or b into d

$$a = -2x^2 + x$$

$$b = 2x^2 + 5x + 1$$

$$d = 3x^3$$

2.4.2 Polynomial Representation

■ [Program 2.5] Initial version of padd function

/* d = a + b, where a, b, and d are polynomials */

d = Zero();

While(!IsZero(a) && ! IsZero(b)) do {

 switch COMPARE(LeadExp(a), LeadExp(b)) {

 case -1 : d = Attach(d, Coef(b, LeadExp(b)), LeadExp(b));

 b = Remove(b, LeadExp(b));

 break;

 case 0 : sum = Coef(a, LeadExp(a)) + Coef(b, LeadExp(b));

 if (sum)

 Attach(d, sum, LeadExp(a));

 a = Remove(a, LeadExp(a));

 b = Remove(b, LeadExp(b));

 break;

 case 1 : d = Attach(d, Coef(a, LeadExp(a)), LeadExp(a));

 a = Remove(a, LeadExp(a));

 }

}

insert any remaining terms of a or b into d

$$\text{sum} = 0$$

$$a = x$$

$$b = 5x + 1$$

$$d = 3x^3$$

2.4.2 Polynomial Representation

■ [Program 2.5] Initial version of padd function

/* d = a + b, where a, b, and d are polynomials */

d = Zero();

While(!IsZero(a) && ! IsZero(b)) do {

 switch COMPARE(LeadExp(a), LeadExp(b)) { -> 0

 case -1 : d = Attach(d, Coef(b, LeadExp(b)), LeadExp(b));

 b = Remove(b, LeadExp(b));

 break;

 case 0 : sum = Coef(a, LeadExp(a)) + Coef(b, LeadExp(b));

 if (sum)

 Attach(d, sum, LeadExp(a));

 a = Remove(a, LeadExp(a));

 b = Remove(b, LeadExp(b));

 break;

 case 1 : d = Attach(d, Coef(a, LeadExp(a)), LeadExp(a));

 a = Remove(a, LeadExp(a));

 }

}

insert any remaining terms of a or b into d

$$a = x$$

$$b = 5x + 1$$

$$d = 3x^3$$

2.4.2 Polynomial Representation

■ [Program 2.5] Initial version of padd function

/* d = a + b, where a, b, and d are polynomials */

d = Zero();

While(!IsZero(a) && ! IsZero(b)) do {

switch COMPARE(LeadExp(a), LeadExp(b)) {

case -1 : d = Attach(d, Coef(b, LeadExp(b)), LeadExp(b));

b = Remove(b, LeadExp(b));

break;

case 0 : sum = Coef(a, LeadExp(a)) + Coef(b, LeadExp(b));

if (sum)

Attach(d, sum, LeadExp(a));

a = Remove(a, LeadExp(a));

b = Remove(b, LeadExp(b));

break;

case 1 : d = Attach(d, Coef(a, LeadExp(a)), LeadExp(a));

a = Remove(a, LeadExp(a));

}

}

insert any remaining terms of a or b into d

$$a = 0$$

$$b = 1$$

$$d = 3x^3 + 6x$$

2.4.2 Polynomial Representation

■ [Program 2.5] Initial version of padd function

/* d = a + b, where a, b, and d are polynomials */

d = Zero();

While(!IsZero(a) && ! IsZero(b)) do {

 switch COMPARE(LeadExp(a), LeadExp(b)) {

 case -1 : d = Attach(d, Coef(b, LeadExp(b)), LeadExp(b));

 b = Remove(b, LeadExp(b));

 break;

 case 0 : sum = Coef(a, LeadExp(a)) + Coef(b, LeadExp(b));

 if (sum)

 Attach(d, sum, LeadExp(a));

 a = Remove(a, LeadExp(a));

 b = Remove(b, LeadExp(b));

 break;

 case 1 : d = Attach(d, Coef(a, LeadExp(a)), LeadExp(a));

 a = Remove(a, LeadExp(a));

 }

}

insert any remaining terms of a or b into d

Finally

$$a = 0$$

$$b = 0$$

$$d = 3x^3 + 6x + 1$$

- Representation decision:
Exponents are uniquely arranged in decreasing order.

<Dense Representation>

Include all the terms in a polynomial :

$$A(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0, \text{ where } a_n \neq 0.$$

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

Let a be a variable of type polynomial.

We can represent the polynomial $A(x) = \sum_{i=0}^n a_i x^i$ in a ,
by setting $a.degree = n$ and $a.coef[i] = a_{n-i}, 0 \leq i \leq n$.

- Although this representation leads to very simple algorithms for most of the operations, it wastes a lot of space.
- For instance, if $a.\text{degree} \ll \text{MAX_DEGREE}$ or if the polynomial is sparse.

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

■ <Sparse Representation>

To preserve space, we use only one global array to store all our polynomials.

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

■ [Figure 2.3] : Array representation of two polynomials

	<u>starta</u>	<u>finisha</u>	<u>startb</u>		<u>finishb</u>	<u>avail</u>				
	↓	↓	↓		↓	↓				
<i>coef</i>	2	1	1	10	3	1				
<i>exp</i>	1000	0	4	3	2	0				
	0	1	2	3	4	5	6	7	8	

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

To represent a zero polynomial c , set $startc > finishc$.

2.4.3 Polynomial Addition

■ [Program 2.6] : Function to add two polynomials

```
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 2.7] : Function to add a new term

```
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");
        exit(1);
    }
    terms[avail].coef = coefficient;
    terms[avail++].expon = exponent;
}
```


- Analysis of *padd* :

Time complexity is $O(n+m)$, where m and n are the number of terms in A and B, respectively.

When *avail* > MAX_TERMS, must we quit?

→ Given the current representation, we must unless there are some polynomials that we no longer need.

2.5 SPARSE MATRICES

2.5.1 The Abstract Data Type

- A *sparse matrix* is a matrix which contains many zero entries.
- If a two-dimensional array is used to represent a sparse matrix, a lot of space is used to store the same value 0 and this implementation does not work when the matrices are large since most compilers impose limits on array sizes.

[Figure 2.3]

	<u>col 0</u>	<u>col 1</u>	<u>col 2</u>
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)

	<u>col 0</u>	<u>col 1</u>	<u>col 2</u>	<u>col 3</u>	<u>col 4</u>	<u>col 5</u>
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) → sparse matrix

■ [ADT 2.3] ADT *Sparse Matrix*

ADT *Sparse_Matrix* 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 $a, b \in \text{Sparse_Matrix}, x \in \text{item}, i, j, \text{max_col}, \text{max_row} \in \text{index}$

Sparse_Matrix Create(*max_row*, *max_col*) ::=

return a *Sparse_Matrix* that can hold up to $\text{max_items} = \text{max_row} \times \text{max_col}$ and whose maximum row size is *max_row* and whose maximum column size is *max_col*.

Sparse_Matrix Transpose(*a*) ::=

return the matrix produced by interchanging the row and column value of every triple.

Sparse_Matrix Add(*a*, *b*) ::=

if the dimension 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.

Sparse_Matrix 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.

end *Sparse_Matrix*

2.5.2 Sparse Matrix Representation

- We can characterize uniquely any element within a matrix by a triple $\langle row, col, value \rangle$.
Thus we can use an array of triples.
- We organize the triples so that row indices are in ascending order and among those with the same row indices are ordered in ascending order of column indices.
- To insure that the operations terminate,
we must know the number of rows and columns,
and the number of nonzero elements in the matrix.

Sparse_Matrix Create(*max_row*, *max_col*) ::=

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

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

[Figure 2.5] For example,

	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)

	row	col	value
<i>b</i> [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)

transpose

2.5.3 Transposing A Matrix

- < A simple algorithm >

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;

We will not know exactly where to place element $\langle j, i, \text{value} \rangle$ in the transpose until we have processed all the elements that precede it.

For instance,

$(0, 0, 15)$ becomes $(0, 0, 15)$

$(0, 3, 22)$ becomes $(3, 0, 22)$

$(0, 5, -15)$ becomes $(5, 0, -15)$

Consecutive insertions are required.

We must move elements to maintain the correct order.

- We can avoid this data movement by using the column indices to determine the placement of elements in the transpose matrix.

for all elements in column j

place element $\langle i, j, \text{value} \rangle$ in element $\langle j, i, \text{value} \rangle$;

- **[Program 2.8] Transpose of a sparse matrix**

```
void transpose (term a[], term b[])
```

```
{ /* b is set to the transpose of a */
```

```
    int n, i, j, currentb;
```

```
    n = a[0].value;           /* total number of elements */
```

```
    b[0].row = a[0].col;      /* rows in b = columns in a */
```

```
    b[0].col = a[0].row;      /* columns in b = rows in a */
```

```
    b[0].value = n;
```

```

if (n > 0) { /* nonzero matrix */
    currentb = 1;
    for (i=0; i<a[0].col; i++) /* transpose by columns in a */
        for (j=1; j<=n; j++) /* find elements from the current column */
            if (a[j].col == i) { /*element is in current column, add it to b*/
                b[currentb].row = a[j].col;
                b[currentb].col = a[j].row;
                b[currentb].value = a[j].value;
                currentb++;
            }
        }
    }
}

```

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

If $\text{elements} = O(\text{rows} \cdot \text{columns})$, then

$O(\text{columns} \cdot \text{elements})$ becomes $O(\text{rows} \cdot \text{columns}^2)$.

```

if (n > 0) { /* nonzero matrix */
    currentb = 1;
    for (i=0; i<a[0].col; i++) /* transpose by columns in a */
        for (j=1; j<=n; j++) /* find elements from the current column */
            if (a[j].col == i) { /*element is in current column, add it to b*/
                b[currentb].row = a[j].col;
                b[currentb].col = a[j].row;
                b[currentb].value = a[j].value;
                currentb++;
            }
    }
}

```

3	2	4
0	0	9
0	1	6
1	1	10
2	0	7

a

2	3	4

b

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

If $\text{elements} = O(\text{rows} \cdot \text{columns})$, then

$O(\text{columns} \cdot \text{elements})$ becomes $O(\text{rows} \cdot \text{columns}^2)$.

```

if (n > 0) { /* nonzero matrix */
    currentb = 1;
    for (i=0; i<a[0].col; i++) /* transpose by columns in a */
        for (j=1; j<=n; j++) /* find elements from the current column */
            if (a[j].col == i) { /*element is in current column, add it to b*/
                b[currentb].row = a[j].col;
                b[currentb].col = a[j].row;
                b[currentb].value = a[j].value;
                currentb++;
            }
    }
}

```

3	2	4
0	0	9
0	1	6
1	1	10
2	0	7

a

2	3	4
0	0	9

b

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

If $\text{elements} = O(\text{rows} \cdot \text{columns})$, then

$O(\text{columns} \cdot \text{elements})$ becomes $O(\text{rows} \cdot \text{columns}^2)$.

```

if (n > 0) { /* nonzero matrix */
    currentb = 1;
    for (i=0; i<a[0].col; i++) /* transpose by columns in a */
        for (j=1; j<=n; j++) /* find elements from the current column */
            if (a[j].col == i) { /*element is in current column, add it to b*/
                b[currentb].row = a[j].col;
                b[currentb].col = a[j].row;
                b[currentb].value = a[j].value;
                currentb++;
            }
    }
}

```

3	2	4
0	0	9
0	1	6
1	1	10
2	0	7

a

b

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

If $\text{elements} = O(\text{rows} \cdot \text{columns})$, then

$O(\text{columns} \cdot \text{elements})$ becomes $O(\text{rows} \cdot \text{columns}^2)$.

```

if (n > 0) { /* nonzero matrix */
    currentb = 1;
    for (i=0; i<a[0].col; i++) /* transpose by columns in a */
        for (j=1; j<=n; j++) /* find elements from the current column */
            if (a[j].col == i) { /*element is in current column, add it to b*/
                b[currentb].row = a[j].col;
                b[currentb].col = a[j].row;
                b[currentb].value = a[j].value;
                currentb++;
            }
    }
}

```

3	2	4
0	0	9
0	1	6
1	1	10
2	0	7

a

2	3	4
0	0	9

b

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

If $\text{elements} = O(\text{rows} \cdot \text{columns})$, then

$O(\text{columns} \cdot \text{elements})$ becomes $O(\text{rows} \cdot \text{columns}^2)$.

```

if (n > 0) { /* nonzero matrix */
    currentb = 1;
    for (i=0; i<a[0].col; i++) /* transpose by columns in a */
        for (j=1; j<=n; j++) /* find elements from the current column */
            if (a[j].col == i) { /*element is in current column, add it to b*/
                b[currentb].row = a[j].col;
                b[currentb].col = a[j].row;
                b[currentb].value = a[j].value;
                currentb++;
            }
    }
}

```

3	2	4
0	0	9
0	1	6
1	1	10
2	0	7

a

2	3	4
0	0	9

b

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

If $\text{elements} = O(\text{rows} \cdot \text{columns})$, then

$O(\text{columns} \cdot \text{elements})$ becomes $O(\text{rows} \cdot \text{columns}^2)$.


```

if (n > 0) { /* nonzero matrix */
    currentb = 1;
    for (i=0; i<a[0].col; i++) /* transpose by columns in a */
        for (j=1; j<=n; j++) /* find elements from the current column */
            if (a[j].col == i) { /*element is in current column, add it to b*/
                b[currentb].row = a[j].col;
                b[currentb].col = a[j].row;
                b[currentb].value = a[j].value;
                currentb++;
            }
    }
}

```

3	2	4	2	3	4
0	0	9	0	0	9
0	1	6	0	2	7
1	1	10			
2	0	7			

a

b

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

If $\text{elements} = O(\text{rows} \cdot \text{columns})$, then

$O(\text{columns} \cdot \text{elements})$ becomes $O(\text{rows} \cdot \text{columns}^2)$.

```

if (n > 0) { /* nonzero matrix */
    currentb = 1;
    for (i=0; i<a[0].col; i++) /* transpose by columns in a */
        for (j=1; j<=n; j++) /* find elements from the current column */
            if (a[j].col == i) { /*element is in current column, add it to b*/
                b[currentb].row = a[j].col;
                b[currentb].col = a[j].row;
                b[currentb].value = a[j].value;
                currentb++;
            }
    }
}

```

3	2	4
0	0	9
0	1	6
1	1	10
2	0	7

a

2	3	4
0	0	9
0	2	7

b

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

If $\text{elements} = O(\text{rows} \cdot \text{columns})$, then

$O(\text{columns} \cdot \text{elements})$ becomes $O(\text{rows} \cdot \text{columns}^2)$.

```

if (n > 0) { /* nonzero matrix */
    currentb = 1;
    for (i=0; i<a[0].col; i++) /* transpose by columns in a */
        for (j=1; j<=n; j++) /* find elements from the current column */
            if (a[j].col == i) { /*element is in current column, add it to b*/
                b[currentb].row = a[j].col;
                b[currentb].col = a[j].row;
                b[currentb].value = a[j].value;
                currentb++;
            }
    }
}

```

3	2	4
0	0	9
0	1	6
1	1	10
2	0	7

a

2	3	4
0	0	9
0	2	7

b

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

If $\text{elements} = O(\text{rows} \cdot \text{columns})$, then

$O(\text{columns} \cdot \text{elements})$ becomes $O(\text{rows} \cdot \text{columns}^2)$.

```

if (n > 0) { /* nonzero matrix */
    currentb = 1;
    for (i=0; i<a[0].col; i++) /* transpose by columns in a */
        for (j=1; j<=n; j++) /* find elements from the current column */
            if (a[j].col == i) { /*element is in current column, add it to b*/
                b[currentb].row = a[j].col;
                b[currentb].col = a[j].row;
                b[currentb].value = a[j].value;
                currentb++;
            }
    }
}

```

3	2	4
0	0	9
0	1	6
1	1	10
2	0	7

a

2	3	4
0	0	9
0	2	7
1	0	6

b

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

If $\text{elements} = O(\text{rows} \cdot \text{columns})$, then

$O(\text{columns} \cdot \text{elements})$ becomes $O(\text{rows} \cdot \text{columns}^2)$.

```

if (n > 0) { /* nonzero matrix */
    currentb = 1;
    for (i=0; i<a[0].col; i++) /* transpose by columns in a */
        for (j=1; j<=n; j++) /* find elements from the current column */
            if (a[j].col == i) { /*element is in current column, add it to b*/
                b[currentb].row = a[j].col;
                b[currentb].col = a[j].row;
                b[currentb].value = a[j].value;
                currentb++;
            }
    }
}

```

3	2	4
0	0	9
0	1	6
1	1	10
2	0	7

a

2	3	4
0	0	9
0	2	7
1	0	6

b

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

If $\text{elements} = O(\text{rows} \cdot \text{columns})$, then

$O(\text{columns} \cdot \text{elements})$ becomes $O(\text{rows} \cdot \text{columns}^2)$.

```

if (n > 0) { /* nonzero matrix */
    currentb = 1;
    for (i=0; i<a[0].col; i++) /* transpose by columns in a */
        for (j=1; j<=n; j++) /* find elements from the current column */
            if (a[j].col == i) { /*element is in current column, add it to b*/
                b[currentb].row = a[j].col;
                b[currentb].col = a[j].row;
                b[currentb].value = a[j].value;
                currentb++;
            }
    }
}

```

3	2	4
0	0	9
0	1	6
1	1	10
2	0	7

a

2	3	4
0	0	9
0	2	7
1	0	6
1	1	10

b

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

If $\text{elements} = O(\text{rows} \cdot \text{columns})$, then

$O(\text{columns} \cdot \text{elements})$ becomes $O(\text{rows} \cdot \text{columns}^2)$.

```

if (n > 0) { /* nonzero matrix */
    currentb = 1;
    for (i=0; i<a[0].col; i++) /* transpose by columns in a */
        for (j=1; j<=n; j++) /* find elements from the current column */
            if (a[j].col == i) { /*element is in current column, add it to b*/
                b[currentb].row = a[j].col;
                b[currentb].col = a[j].row;
                b[currentb].value = a[j].value;
                currentb++;
            }
    }
}

```

3	2	4	2	3	4
0	0	9	0	0	9
0	1	6	0	2	7
1	1	10	1	0	6
2	0	7	1	1	10

a

b

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

If $\text{elements} = O(\text{rows} \cdot \text{columns})$, then

$O(\text{columns} \cdot \text{elements})$ becomes $O(\text{rows} \cdot \text{columns}^2)$.

- A transpose algorithm using dense representation :

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

Time complexity : $O(\text{rows} \cdot \text{columns})$.

- <A much better algorithm by using a little more storage>

This algorithm, *fast_transpose*, proceeds by first determining the number of elements in each column of the original matrix.

This gives the number of elements in each row of the transpose matrix.

■ [Program 2.9]

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

2	3	4
0	0	9
0	1	6
1	0	10
1	2	7

a

3	2	4

b

0	2
1	1
2	1

row_terms

0	1
1	
2	

starting_pos

```

for (i = 0; i < num_cols; i++)    row_terms[i] = 0;
for (i = 1; i <= num_terms; i++) row_terms[a[i].col]++;
starting_pos[0] = 1;
for (i = 1; i < num_cols; i++)
    starting_pos[i] = starting_pos[i-1] + row_terms[i-1];
for (i = 1; i <= num_terms; i++) {
    j = starting_pos[a[i].col]++;
    b[j].row = a[i].col;    b[j].col = a[i].row;
    b[j].value = a[i].value;
}

```

2	3	4
0	0	9
0	1	6
1	0	10
1	2	7

a

3	2	4

b

0	2
1	1
2	1

row_terms

0	1
1	1+2=3
2	

starting_pos

```

for (i = 0; i < num_cols; i++)    row_terms[i] = 0;
for (i = 1; i <= num_terms; i++) row_terms[a[i].col]++;
starting_pos[0] = 1;
for (i = 1; i < num_cols; i++)
    starting_pos[i] = starting_pos[i-1] + row_terms[i-1];
for (i = 1; i <= num_terms; i++) {
    j = starting_pos[a[i].col]++;
    b[j].row = a[i].col;    b[j].col = a[i].row;
    b[j].value = a[i].value;
}

```

2	3	4
0	0	9
0	1	6
1	0	10
1	2	7

a

3	2	4

b

0	2
1	1
2	1

row_terms

0	1
1	3
2	3+1=4

starting_pos

```

for (i = 0; i < num_cols; i++)    row_terms[i] = 0;
for (i = 1; i <= num_terms; i++) row_terms[a[i].col]++;
starting_pos[0] = 1;
for (i = 1; i < num_cols; i++)
    starting_pos[i] = starting_pos[i-1] + row_terms[i-1];
for (i = 1; i <= num_terms; i++) {
    j = starting_pos[a[i].col]++;
    b[j].row = a[i].col;    b[j].col = a[i].row;
    b[j].value = a[i].value;
}

```

2	3	4
0	0	9
0	1	6
1	0	10
1	2	7

a

3	2	4

b

0	2
1	1
2	1

row_terms

0	1→2
1	3
2	4

starting_pos

```

for (i = 0; i < num_cols; i++)    row_terms[i] = 0;
for (i = 1; i <= num_terms; i++) row_terms[a[i].col]++;
starting_pos[0] = 1;
for (i = 1; i < num_cols; i++)
    starting_pos[i] = starting_pos[i-1] + row_terms[i-1];
for (i = 1; i <= num_terms; i++) {
    j = starting_pos[a[i].col]++;
    b[j].row = a[i].col;  b[j].col = a[i].row;
    b[j].value = a[i].value;
}

```

j=1

2	3	4
0	0	9
0	1	6
1	0	10
1	2	7

a

3	2	4
0	0	9

b

0	2
1	1
2	1

row_terms

0	2
1	3
2	4

starting_pos

```

for (i = 0; i < num_cols; i++)    row_terms[i] = 0;
for (i = 1; i <= num_terms; i++) row_terms[a[i].col]++;
starting_pos[0] = 1;
for (i = 1; i < num_cols; i++)
    starting_pos[i] = starting_pos[i-1] + row_terms[i-1];
for (i = 1; i <= num_terms; i++) {
    j = starting_pos[a[i].col]++;
    b[j].row = a[i].col;  b[j].col = a[i].row;
    b[j].value = a[i].value;
}

```

j=1

2	3	4
0	0	9
0	1	6
1	0	10
1	2	7

a

3	2	4
0	0	9

b

0	2
1	1
2	1

row_terms

0	2
1	3→4
2	4

starting_pos

```

for (i = 0; i < num_cols; i++)    row_terms[i] = 0;
for (i = 1; i <= num_terms; i++) row_terms[a[i].col]++;
starting_pos[0] = 1;
for (i = 1; i < num_cols; i++)
    starting_pos[i] = starting_pos[i-1] + row_terms[i-1];
for (i = 1; i <= num_terms; i++) {
    j = starting_pos[a[i].col]++;
    b[j].row = a[i].col;  b[j].col = a[i].row;
    b[j].value = a[i].value;
}

```

j=3

2	3	4
0	0	9
0	1	6
1	0	10
1	2	7

a

3	2	4
0	0	9
1	0	6

b

0	2
1	1
2	1

row_terms

0	2
1	4
2	4

starting_pos

```

for (i = 0; i < num_cols; i++)    row_terms[i] = 0;
for (i = 1; i <= num_terms; i++) row_terms[a[i].col]++;
starting_pos[0] = 1;
for (i = 1; i < num_cols; i++)
    starting_pos[i] = starting_pos[i-1] + row_terms[i-1];
for (i = 1; i <= num_terms; i++) {
    j = starting_pos[a[i].col]++;
    b[j].row = a[i].col;  b[j].col = a[i].row;
    b[j].value = a[i].value;
}

```

j=3

2	3	4
0	0	9
0	1	6
1	0	10
1	2	7

a

3	2	4
0	0	9
1	0	6

b

0	2
1	1
2	1

row_terms

0	2→3
1	4
2	4

starting_pos

```

for (i = 0; i < num_cols; i++)    row_terms[i] = 0;
for (i = 1; i <= num_terms; i++) row_terms[a[i].col]++;
starting_pos[0] = 1;
for (i = 1; i < num_cols; i++)
    starting_pos[i] = starting_pos[i-1] + row_terms[i-1];
for (i = 1; i <= num_terms; i++) {
    j = starting_pos[a[i].col]++;
    b[j].row = a[i].col;  b[j].col = a[i].row;
    b[j].value = a[i].value;
}

```

j=2

2	3	4
0	0	9
0	1	6
1	0	10
1	2	7

a

3	2	4
0	0	9
0	1	10
1	0	6

b

0	2
1	1
2	1

row_terms

0	3
1	4
2	4

starting_pos

```

for (i = 0; i < num_cols; i++)    row_terms[i] = 0;
for (i = 1; i <= num_terms; i++) row_terms[a[i].col]++;
starting_pos[0] = 1;
for (i = 1; i < num_cols; i++)
    starting_pos[i] = starting_pos[i-1] + row_terms[i-1];
for (i = 1; i <= num_terms; i++) {
    j = starting_pos[a[i].col]++;
    b[j].row = a[i].col;  b[j].col = a[i].row;
    b[j].value = a[i].value;
}

```

j=2

2	3	4
0	0	9
0	1	6
1	0	10
1	2	7

a

3	2	4
0	0	9
0	1	10
1	0	6

b

0	2
1	1
2	1

row_terms

0	2
1	4
2	4→5

starting_pos

```

for (i = 0; i < num_cols; i++)    row_terms[i] = 0;
for (i = 1; i <= num_terms; i++) row_terms[a[i].col]++;
starting_pos[0] = 1;
for (i = 1; i < num_cols; i++)
    starting_pos[i] = starting_pos[i-1] + row_terms[i-1];
for (i = 1; i <= num_terms; i++) {
    j = starting_pos[a[i].col]++;
    b[j].row = a[i].col;  b[j].col = a[i].row;
    b[j].value = a[i].value;
}

```

j=4

2	3	4
0	0	9
0	1	6
1	0	10
1	2	7

a

3	2	4
0	0	9
0	1	10
1	0	6
2	1	7

b

0	2
1	1
2	1

row_terms

0	2
1	4
2	5

starting_pos

```

for (i = 0; i < num_cols; i++)    row_terms[i] = 0;
for (i = 1; i <= num_terms; i++) row_terms[a[i].col]++;
starting_pos[0] = 1;
for (i = 1; i < num_cols; i++)
    starting_pos[i] = starting_pos[i-1] + row_terms[i-1];
for (i = 1; i <= num_terms; i++) {
    j = starting_pos[a[i].col]++;
    b[j].row = a[i].col;  b[j].col = a[i].row;
    b[j].value = a[i].value;
}

```

j=4

- Time complexity : $O(\text{columns} + \text{elements})$.
If $\text{elements} = O(\text{rows} \cdot \text{columns})$, then
 $O(\text{columns} + \text{elements})$ becomes $O(\text{rows} \cdot \text{columns})$.
- Additional arrays, *row_terms* and *starting_pos*, are used.
- We can reduce this space to one array
if we put the starting positions into the space used by *row_terms*.

2.5.4 Matrix Multiplication

■ Definition :

Given two matrices 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 $0 \leq j < p$.

- <Matrix Multiplication Algorithm using dense representation>

```
for (i = 0; i < rows_a; i++) {  
    for (j = 0; j < cols_b; j++) {  
        sum = 0;  
        for (k = 0; k < cols_a; k++)  
            sum += a[i][k]*b[k][j];  
        d[i][j] = sum;  
    }  
}
```

Time Complexity : $O(rows_a \cdot cols_a \cdot cols_b)$

Note that the product of two sparse matrices may no longer be sparse.

For example :
$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

- <Multiplying two sparse matrices represented as an ordered list>

Need to compute the elements of D by rows so that we can store them in their proper place without moving previously computed elements.

A

rows_a	cols_a	totala
rows_a		

B

rows_b	cols_b	totalb

B^T

cols_b	rows_b	totalb
cols b	-1	

D

rows_a	cols_b	totald

■ [Program 2.10]

Matrices A, B, and D are stored in the arrays *a*, *b*, and *d*, respectively.
Transpose of B is stored in *new_b*.

Variables used :

row - the row of A that we are currently multiplying with the columns in B.

row_begin - the position in *a* of the first element of the current row.

column - the column of B that we are currently multiplying with a row in A.

totald - the current number of elements in the product matrix D.

i, *j* - pointers which are used to examine successively elements
from a row of A and a column B.

```
void mmult(term a[], term b[], term d[])
/* multiply two sparse matrices */
{
    int i, j, column, totalb = b[0].value, totald = 0;
    int rows_a = a[0].row, cols_a = a[0].col, totala = a[0].value;
    int cols_b = b[0].col;
    int row_begin = 1, row = a[1].row, sum = 0;
    term new_b[MAX_TERMS];
    if (cols_a != b[0].row) {
        fprintf(stderr, "Incompatible matrices\n");
        exit(1);
    }
    fast_transpose(b, new_b);
    /* set boundary condition */
    a[totala+1].row = rows_a;
    new_b[totalb+1].row = cols_b;    new_b[totalb+1].col = -1;
```

```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++)
                ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
    }
}

```

```

else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
        break;
    case 1 : /* go to next term in b */
        j++;
}
} /* end of for j <= totalb+1 */
for ( ; a[i].row == row; i++)
    ;
row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;
}

```

```

int i, j, column, totalb = b[0].value, totald = 0;
int rows_a = a[0].row, cols_a = a[0].col, totala = a[0].value;
int cols_b = b[0].col;
int row_begin = 1, row = a[1].row, sum = 0;
term new_b[MAX_TERMS];
if (col_a != b[0].row) {
    fprintf(stderr, "Incompatible matrices\n");
    exit(1);
}
fast_transpose(b, new_b);
/* set boundary condition */
a[totala+1].row = rows_a;
new_b[totalb+1].row = cols_b;  new_b[totalb+1].col = -1;

```

row_begin=1

row=0

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

2	3	3
0	0	4
0	2	5
1	1	1

a

$$\begin{bmatrix} 1 & 0 \\ 0 & 2 \\ 1 & 0 \end{bmatrix}$$

3	2	3
0	0	1
1	1	2
2	0	1

b

new_b

```

int i, j, column, totalb = b[0].value, totald = 0;
int rows_a = a[0].row, cols_a = a[0].col, totala = a[0].value;
int cols_b = b[0].col;
int row_begin = 1, row = a[1].row, sum = 0;
term new_b[MAX_TERMS];
if (col_a != b[0].row) {
    fprintf(stderr, "Incompatible matrices\n");
    exit(1);
}
fast_transpose(b, new_b);
/* set boundary condition */
a[totala+1].row = rows_a;
new_b[totalb+1].row = cols_b;  new_b[totalb+1].col = -1;

```

row_begin=1

row=0

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

2	3	3
0	0	4
0	2	5
1	1	1

a

$$\begin{bmatrix} 1 & 0 \\ 0 & 2 \\ 1 & 0 \end{bmatrix}$$

3	2	3
0	0	1
1	1	2
2	0	1

b

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

2	3	3
0	0	1
0	2	1
1	1	2

new_b

```

int i, j, column, totalb = b[0].value, totald = 0;
int rows_a = a[0].row, cols_a = a[0].col, totala = a[0].value;
int cols_b = b[0].col;
int row_begin = 1, row = a[1].row, sum = 0;
term new_b[MAX_TERMS];
if (col_a != b[0].row) {
    fprintf(stderr, "Incompatible matrices\n");
    exit(1);
}
fast_transpose(b, new_b);
/* set boundary condition */
a[totala+1].row = rows_a;
new_b[totalb+1].row = cols_b;  new_b[totalb+1].col = -1;

```

row_begin=1

row=0

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

2	3	3
0	0	4
0	2	5
1	1	1
2		

a

$$\begin{bmatrix} 1 & 0 \\ 0 & 2 \\ 1 & 0 \end{bmatrix}$$

3	2	3
0	0	1
1	1	2
2	0	1

b

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

2	3	3
0	0	1
0	2	1
1	1	2
2	-1	

new_b

```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
i	0	0	4
	0	2	5
	1	1	1
	2		

a

	2	3	3
j	0	0	1
	0	2	1
	1	1	2
	2	-1	

new_b

row=0

column=0

row_begin=1

sum=0

i=1

j=1

totald=0

d


```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
i	0	0	4
	0	2	5
	1	1	1
	2		

a

	2	3	3
j	0	0	1
	0	2	1
	1	1	2
	2	-1	

new_b

row=0

column=0

row_begin=1

sum=0

i=1

j=1

totald=0

d

```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
	0	0	4
i	0	2	5
	1	1	1
	2		

a

	2	3	3
	0	0	1
j	0	2	1
	1	1	2
	2	-1	

new_b

row=0

column=0

row_begin=1

sum=4

i=2

j=2

totald=0

d

```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
	0	0	4
i	0	2	5
	1	1	1
	2		

a

	2	3	3
	0	0	1
j	0	2	1
	1	1	2
	2	-1	

new_b

row=0

column=0

row_begin=1

sum=4

i=2

j=2

totald=0

d

```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
	0	0	4
	0	2	5
i	1	1	1
	2		

a

	2	3	3
	0	0	1
	0	2	1
j	1	1	2
	2	-1	

new_b

row=0

column=0

row_begin=1

sum=9

i=3

j=3

totald=0

d

```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
	0	0	4
	0	2	5
i	1	1	1
	2		

a

	2	3	3
	0	0	1
	0	2	1
j	1	1	2
	2	-1	

new_b

row=0

column=0

row_begin=1

sum=9

i=3

j=3

totald=0

d

```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
i	0	0	4
	0	2	5
	1	1	1
	2		

a

	2	3	3
	0	0	1
	0	2	1
	1	1	2
	2	-1	

new_b

row=0

column=1

row_begin=1

sum=0

i=1

j=3

totald=1

	0	0	9

d

```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
i	0	0	4
	0	2	5
	1	1	1
	2		

a

	2	3	3
	0	0	1
	0	2	1
	1	1	2
	2	-1	

new_b

row=0

column=1

row_begin=1

sum=0

i=1

j=3

totald=1

0	0	9

d

```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
	0	0	4
i	0	2	5
	1	1	1
	2		

a

	2	3	3
	0	0	1
	0	2	1
	1	1	2
	2	-1	

new_b

row=0

column=1

row_begin=1

sum=0

i=2 j=3

totald=1

0	0	9

d


```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
	0	0	4
i	0	2	5
	1	1	1
	2		

a

	2	3	3
	0	0	1
	0	2	1
	1	1	2
	2	-1	

new_b

row=0

column=1

row_begin=1

sum=0

i=2

j=3

totald=1

0	0	9

d

```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
	0	0	4
i	0	2	5
	1	1	1
	2		

a

	2	3	3
	0	0	1
	0	2	1
	1	1	2
	2	-1	

j

new_b

row=0

column=1

row_begin=1

sum=0

i=2

j=4

totald=1

0	0	9

d

```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
	0	0	4
i	0	2	5
	1	1	1
	2		

a

	2	3	3
	0	0	1
	0	2	1
	1	1	2
j	2	-1	

new_b

row=0

column=1

row_begin=1

sum=0

i=2

j=4

totald=1

0	0	9

d

```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
i	0	0	4
	0	2	5
	1	1	1
	2		

a

	2	3	3
	0	0	1
	0	2	1
	1	1	2
j	2	-1	

new_b

row=0

column=2

row_begin=1

sum=0

i=1

j=4

totald=1

0	0	9

d

```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
i	0	0	4
	0	2	5
	1	1	1
	2		

a

	2	3	3
	0	0	1
	0	2	1
	1	1	2
	2	-1	

j

new_b

row=0

column=2

row_begin=1

sum=0

i=1

j=4

totald=1

0	0	9

d

```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

i

2	3	3
0	0	4
0	2	5
1	1	1
2		

a

2	3	3
0	0	1
0	2	1
1	1	2
2	-1	

new_b

row=0

column=2

row_begin=1

sum=0

i=1

j=5

totald=1

0	0	9

d

```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) { → break
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
	0	0	4
	0	2	5
i	1	1	1
	2		

a

	2	3	3
	0	0	1
	0	2	1
	1	1	2
	2	-1	

new_b

row=1

column=2

row_begin=3

sum=0

i=3

j=5

totald=1

	0	0	9

d

```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
	0	0	4
	0	2	5
i	1	1	1
	2		

a

	2	3	3
	0	0	1
	0	2	1
	1	1	2
	2	-1	

j

new_b

row=1

column=0

row_begin=3

sum=0

i=3

j=1

totald=1

0	0	9

d


```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
	0	0	4
	0	2	5
i	1	1	1
	2		

a

	2	3	3
	0	0	1
	0	2	1
	1	1	2
	2	-1	

j

new_b

row=1

column=0

row_begin=3

sum=0

i=3

j=1

totald=1

0	0	9

d

```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
	0	0	4
	0	2	5
i	1	1	1
	2		

a

	2	3	3
	0	0	1
	0	2	1
	1	1	2
	2	-1	

j

new_b

row=1

column=0

row_begin=3

sum=0

i=3

j=2

totald=1

0	0	9

d

```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
	0	0	4
	0	2	5
i	1	1	1
	2		

a

	2	3	3
	0	0	1
	0	2	1
	1	1	2
	2	-1	

new_b

row=1

column=0

row_begin=3

sum=0

i=3

j=2

totald=1

	0	0	9

d

```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
	0	0	4
	0	2	5
	1	1	1
i	2		

a

	2	3	3
	0	0	1
	0	2	1
	1	1	2
	2	-1	

j

new_b

row=1

column=0

row_begin=3

sum=0

i=4

j=2

totald=1

0	0	9

d

```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
	0	0	4
	0	2	5
	1	1	1
i	2		

a

	2	3	3
	0	0	1
	0	2	1
	1	1	2
	2	-1	

j

new_b

row=1

column=0

row_begin=3

sum=0

i=4

j=2

totald=1

0	0	9

d

```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
	0	0	4
	0	2	5
i	1	1	1
	2		

a

	2	3	3
	0	0	1
	0	2	1
j	1	1	2
	2	-1	

new_b

row=1

column=1

row_begin=3

sum=0

i=3

j=3

totald=1

	0	0	9

d

```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
	0	0	4
	0	2	5
i	1	1	1
	2		

a

	2	3	3
	0	0	1
	0	2	1
j	1	1	2
	2	-1	

new_b

row=1

column=1

row_begin=3

sum=0

i=3

j=3

totald=1

	0	0	9

d

```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
	0	0	4
	0	2	5
	1	1	1
i	2		

a

	2	3	3
	0	0	1
	0	2	1
	1	1	2
j	2	-1	

new_b

row=1

column=1

row_begin=3

sum=2

i=4

j=4

totald=1

0	0	9

d


```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
	0	0	4
	0	2	5
	1	1	1
i	2		

a

	2	3	3
	0	0	1
	0	2	1
	1	1	2
j	2	-1	

new_b

row=1

column=1

row_begin=3

sum=2

i=4

j=4

totald=1

0	0	9

d

```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
	0	0	4
	0	2	5
i	1	1	1
	2		

a

	2	3	3
	0	0	1
	0	2	1
	1	1	2
j	2	-1	

new_b

row=1

column=2

row_begin=3

sum=0

i=3

j=4

totald=2

	0	0	9
	1	1	2

d

```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
	0	0	4
	0	2	5
i	1	1	1
	2		

a

	2	3	3
	0	0	1
	0	2	1
	1	1	2
j	2	-1	

new_b

row=1

column=2

row_begin=3

sum=0

i=3

j=4

totald=2

	0	0	9
	1	1	2

d

```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) {
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
	0	0	4
	0	2	5
i	1	1	1
	2		

a

	2	3	3
	0	0	1
	0	2	1
	1	1	2
	2	-1	

new_b

row=1

column=2

row_begin=3

sum=0

i=3

j=5

totald=2

	0	0	9
	1	1	2

d

```

for (i = 1; i <= totala; ) {
    column = new_b[1].row;
    for (j = 1; j <= totalb+1; ) { → break
        /* multiply row of a by column of b */
        if (a[i].row != row) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            for ( ; new_b[j].row == column; j++) ;
            column = new_b[j].row;
        }
        else if (new_b[j].row != column) {
            storesum(d, &totald, row, column, &sum);
            i = row_begin;
            column = new_b[j].row;
        }
        else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
                break;
            case 1 : /* go to next term in b */
                j++;
        }
    } /* end of for j <= totalb+1 */
    for ( ; a[i].row == row; i++) ;
    row_begin = i; row = a[i].row;
} /* end of for i <= totala */
d[0].row = row_a; d[0].col = cols_b;
d[0].value = totald;

```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
	0	0	4
	0	2	5
	1	1	1
i	2		

a

	2	3	3
	0	0	1
	0	2	1
	1	1	2
	2	-1	

new_b

row=2

column=2

row_begin=4

sum=0

i=4

j=5

totald=2

0	0	9
1	1	2

d

```
for (i = 1; i <= totala; ) { → break
```

```
    column = new_b[1].row;
```

```
    for (j = 1; j <= totalb+1; ) {
```

```
        /* multiply row of a by column of b */
```

```
        if (a[i].row != row) {
```

```
            storesum(d, &totald, row, column, &sum);
```

```
            i = row_begin;
```

```
            for ( ; new_b[j].row == column; j++) ;
```

```
            column = new_b[j].row;
```

```
        }
```

```
    else if (new_b[j].row != column) {
```

```
        storesum(d, &totald, row, column, &sum);
```

```
        i = row_begin;
```

```
        column = new_b[j].row;
```

```
    }
```

```
    else switch (COMPARE(a[i].col, new_b[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 * new_b[j++].value);
```

```
            break;
```

```
        case 1 : /* go to next term in b */
```

```
            j++;
```

```
    }
```

```
    } /* end of for j <= totalb+1 */
```

```
    for ( ; a[i].row == row; i++) ;
```

```
    row_begin = i; row = a[i].row;
```

```
    } /* end of for i <= totala */
```

```
d[0].row = row_a; d[0].col = cols_b;
```

```
d[0].value = totald;
```

$$\begin{bmatrix} 4 & 0 & 5 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 2 & 0 \end{bmatrix}$$

	2	3	3
	0	0	4
	0	2	5
	1	1	1
i	2		

a

	2	3	3
	0	0	1
	0	2	1
	1	1	2
	2	-1	

new_b

row=2

column=2

row_begin=4

sum=0

i=4

j=5

totald=2

2	2	2
0	0	9
1	1	2

d

■ [Program 2.11]

```
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 exceeds %d", MAX_TERMS);
            exit(1);
        }
}
```

- Notice that we have introduced an additional term into both a and new_b :

```
a[totala+1].row = rows_a;  
new_b[totalb+1].row = cols_b;  
new_b[totalb+1].col = -1;
```

- Time complexity :

lines before the *for* loop:

fast transpose - $O(cols_b + totalb)$ time.

the outer *for* loop is iterated $total_a$ times:

at each iteration - one row of the product matrix D is computed
by the inner *for* loop in which at each iteration
either i or j or both increase by 1,
or i is reset to row_begin .

The maximum total increment in j is $totalb+1$.

Let r_k be the number of terms in row k .

Then when row k is processed, i can increase at most r_k times

and i is reset to row_begin at most $cols_b$ times.

Thus the maximum total increment in i is $cols_b \cdot r_k$.

The inner *for* loop requires $O(cols_b \cdot r_k + totalb)$ time.

Therefore the outer *for* loop requires

$$\begin{aligned} & \sum_{k=0}^{rows_a-1} O(cols_b \cdot r_k + totalb) \\ &= O(cols_b \cdot \sum_{k=0}^{rows_a-1} r_k + rows_a \cdot totalb) \\ &= O(cols_b \cdot totala + rows_a \cdot totalb). \end{aligned}$$

Note that if $totala = O(rows_a \cdot cols_a)$ and $totalb = O(rows_b \cdot cols_b)$,
its complexity becomes $O(rows_a \cdot cols_a \cdot cols_b)$.

2.6 REPRESENTATION OF MULTIDIMENSIONAL ARRAYS

- If an array is declared $a[\text{upper}_0][\text{upper}_1]\cdots[\text{upper}_{n-1}]$,

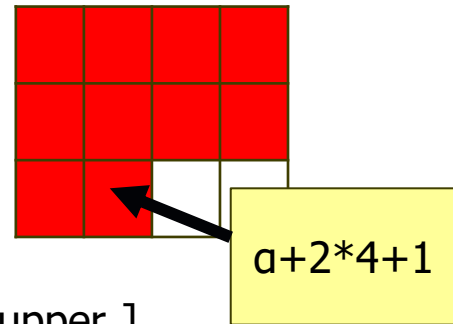
the number of elements in the array is $\prod_{i=0}^{n-1} \text{upper}_i$

- Two common ways to represent multidimensional arrays :
row major order : stores multidimensional arrays by rows.
column major order : stores multidimensional arrays by columns.

- <row major order>

We interpret the two-dimensional array $a[\text{upper}_0][\text{upper}_1]$
 as upper_0 rows, $\text{row}_0, \text{row}_1, \dots, \text{row}_{\text{upper}_0-1}$,
 each row containing upper_1 elements.

If we assume that α is the address of $a[0][0]$,
 then the address of $a[i][j]$ is $\alpha + i \cdot \text{upper}_1 + j$.



To represent a three-dimensional array $a[\text{upper}_0][\text{upper}_1][\text{upper}_2]$,
 we interpret the array as upper_0 two-dimensional arrays
 of dimension $\text{upper}_1 \times \text{upper}_2$.

Then the address of $a[i][j][k]$ is
 $\alpha + i \cdot \text{upper}_1 \cdot \text{upper}_2 + j \cdot \text{upper}_2 + k$.

- Generalizing on the preceding discussion, we can obtain the addressing formula for any element $a[i_0][i_1]\cdots[i_{n-1}]$ in an array declared as $a[upper_0][upper_1]\cdots[upper_{n-1}]$.

If α is the address of $a[0][0] \dots [0]$, the address of $a[i_0][i_1]\cdots[i_{n-1}]$ is :

$$\begin{aligned}
 & \alpha + i_0 \cdot upper_1 \cdot upper_2 \cdots upper_{n-1} \\
 & \quad + i_1 \cdot upper_2 \cdot upper_3 \cdots upper_{n-1} \\
 & \quad + i_2 \cdot upper_3 \cdot upper_4 \cdots upper_{n-1} \\
 & \quad \cdot \\
 & \quad \cdot \\
 & \quad \cdot \\
 & \quad + i_{n-2} \cdot upper_{n-1} \\
 & \quad + i_{n-1} \\
 & = \alpha + \sum_{j=0}^{n-1} i_j a_j \quad \text{where} \begin{cases} a_j = \prod_{k=j+1}^{n-1} upper_k, & 0 \leq j < n-1, \\ a_{n-1} = 1 \end{cases}
 \end{aligned}$$

2.7 STRINGS

2.7.1 The Abstract Data Type

A *string* is a finite sequence of zero or more characters,
 $S = s_0, \dots, s_{n-1}$, where s_i are characters.

■ [ADT2.4] Abstract data type *String*:

ADT *String* is

objects : a finite sequence of zero or more characters.

functions : for all $s, t \in \text{String}$, $i, j, m \in \text{non-negative integers}$

$\text{String Null}(m) ::= \text{return}$ 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, ..., *i*+*j*-1.
 else return *NULL*.

2.7.2 Strings in C

■ <Representation>

In C, we represent strings as character arrays terminated with the null character .

For instance,

```
#define MAX_SIZE 100
char s[MAX_SIZE] = "dog";
char t[MAX_SIZE] = "house";
```

Internal representation in C :

[Figure 2.9]

s[0]	s[1]	s[2]	s[3]	t[0]	t[1]	t[2]	t[3]	t[4]	t[5]
d	o	g	\0	h	o	u	s	e	\0

- Alternative declaration :

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

Concatenating these two strings by calling *strcat*(s,t) which stores the result in *s*. This produces the new string, "doghouse".

Although *s* has increased in length by five,
we have no additional space in *s* to store the extra five characters.

Most of *C* compilers simply *overwrite* the memory
to fit in the extra five characters.

- C provides built-in other string functions which we access through the statement
#include <string.h>

- **Example 2.2 [String insertion]**

```
# include <string.h>
# define MAX_SIZE 100
char string1 [MAX_SIZE], *s = string1;
char string2 [MAX_SIZE], *t = string2;
```

■ [Program 2.12]

```
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 ");
        exit(1);
    }
    if (!strlen(s))
        strcpy(s, t);
    else if (strlen(t)) {
        strncpy(temp, s, i);
        strcat(temp, t);
        strcat(temp, (s+i));
        strcpy(s, temp);
    }
}
```

s →

a	m	o	b	i	l	e	\0
---	---	---	---	---	---	---	----

t →

u	t	o	\0
---	---	---	----

temp →

\0

initially

temp →

a	\0
---	----

(a) after strncpy(temp, s, i)

temp →

a	u	t	o	\0
---	---	---	---	----

(b) after strcat(temp,t)

temp →

a	u	t	o	m	o	b	i	l	e	\0
---	---	---	---	---	---	---	---	---	---	----

(c) after strcat(temp,(s+i))

■ [Program 2.12]

```
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 ");
```

```
        exit(1);
```

```
    }
```

```
    if (!strlen(s))
```

```
        strcpy(s, t);
```

```
    else if (strlen(t)) {
```

```
        strncpy(temp, s, i);
```

```
        strcat(temp, t);
```

```
        strcat(temp, (s+i));
```

```
        strcpy(s, temp);
```

```
    }
```

```
}
```

s

a	m	o	b	i	l	e	\0
---	---	---	---	---	---	---	----

t

u	t	o	\0
---	---	---	----

temp

\0

■ [Program 2.12]

```
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 ");  
    exit(1);
```

```
  }
```

```
  if (!strlen(s))
```

```
    strcpy(s, t);
```

```
  else if (strlen(t)) {
```

```
    strncpy(temp, s, i);
```

```
    strcat(temp, t);
```

```
    strcat(temp, (s+i));
```

```
    strcpy(s, temp);
```

```
  }
```

```
}
```

s

a	m	o	b	i	l	e	\0
---	---	---	---	---	---	---	----

t

u	t	o	\0
---	---	---	----

temp

a	\0
---	----

■ [Program 2.12]

```
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 ");
```

```
    exit(1);
```

```
}
```

```
if (!strlen(s))
```

```
    strcpy(s, t);
```

```
else if (strlen(t)) {
```

```
    strncpy(temp, s, i);
```

```
    strcat(temp, t);
```

```
    strcat(temp, (s+i));
```

```
    strcpy(s, temp);
```

```
}
```

```
}
```

s

a	m	o	b	i	l	e	\0
---	---	---	---	---	---	---	----

t

u	t	o	\0
---	---	---	----

temp

a	u	t	o	\0
---	---	---	---	----

■ [Program 2.12]

```
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 ");  
    exit(1);
```

```
  }  
  if (!strlen(s))
```

```
    strcpy(s, t);
```

```
  else if (strlen(t)) {
```

```
    strncpy(temp, s, i);
```

```
    strcat(temp, t);
```

```
    strcat(temp, (s+i));
```

```
    strcpy(s, temp);
```

```
  }
```

```
}
```

s

a	m	o	b	i	l	e	\0
---	---	---	---	---	---	---	----

t

u	t	o	\0
---	---	---	----

temp

a	u	t	o	m	o	b	i	l	e	\0
---	---	---	---	---	---	---	---	---	---	----

■ [Program 2.12]

```
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 ");
```

```
        exit(1);
```

```
    }
```

```
    if (!strlen(s))
```

```
        strcpy(s, t);
```

```
    else if (strlen(t)) {
```

```
        strncpy(temp, s, i);
```

```
        strcat(temp, t);
```

```
        strcat(temp, (s+i));
```

```
        strcpy(s, temp);
```

```
    }
```

```
}
```

s

a	u	t	o	m	o	b	i	l	e	\0
---	---	---	---	---	---	---	---	---	---	----

t

u	t	o	\0
---	---	---	----

temp

a	u	t	o	m	o	b	i	l	e	\0
---	---	---	---	---	---	---	---	---	---	----

2.7.3 Pattern Matching

```
char pat[MAX_SIZE], string[MAX_SIZE], *t;
```

To determine if *pat* is in *string* :

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

The call (*t = strstr(string, pat)*) returns
a null pointer if *pat* is not in *string*.
a pointer to the start of *pat* in *string* if *pat* is in *string*.

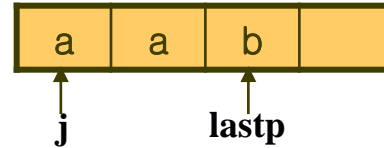
- Reasons of developing our own pattern matching function:
 - (1) The function *strstr* may not be available with the compiler we are using.
 - (2) There are several different methods for implementing a pattern matching function.
- A simple matching algorithm :

Sequentially examines each character of the string until it finds the pattern or it reaches the end of the string.
- If *pat* is not in *string*, this algorithm has a computing time of $O(nm)$, where n is the length of *pat* and m is the length of *string*.
- Improvements (incorporated in **Program 2.13**) :
 - 1. Quitting when *strlen(pat)* is greater than the number of remaining characters in the string.
 - 2. Checking the first and last characters of *pat* before we checking the remaining characters.

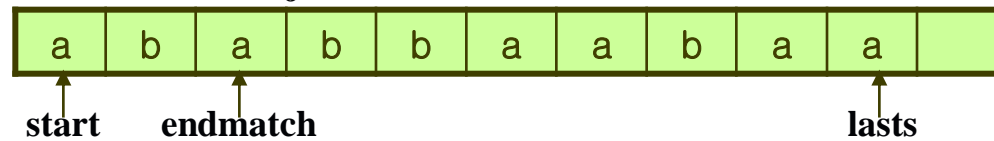
■ [Program 2.13 Pattern matching by checking end indices first]

```
int nfind(char * string, char *pat)
{
    /* match the last character of the 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;
}
```

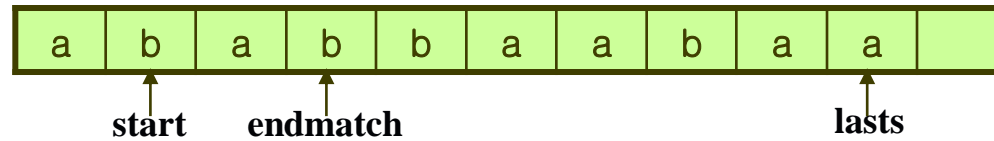
(a) pattern



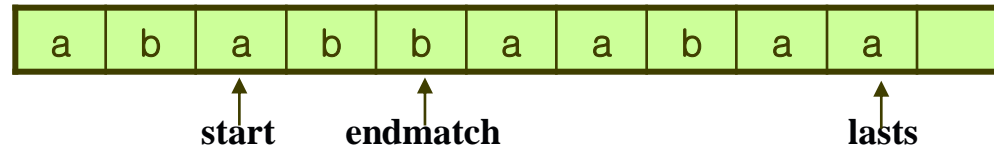
(b) no match



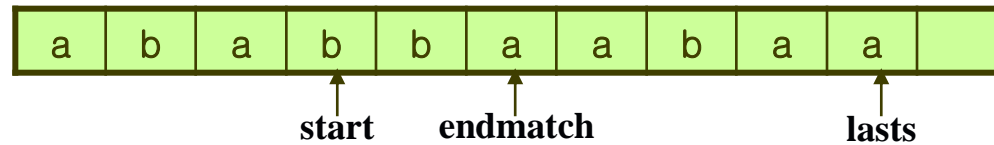
(c) no match



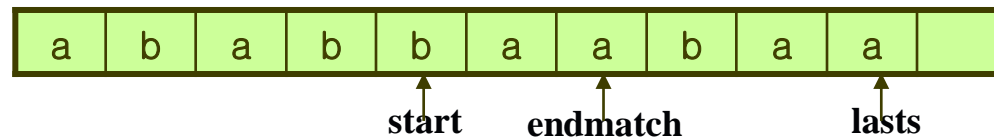
(d) no match



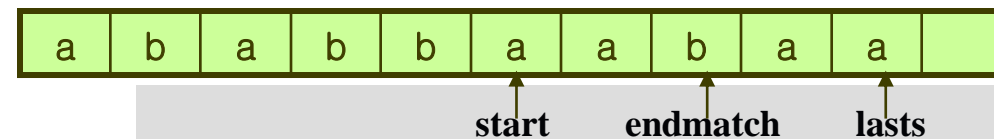
(e) no match



(f) no match



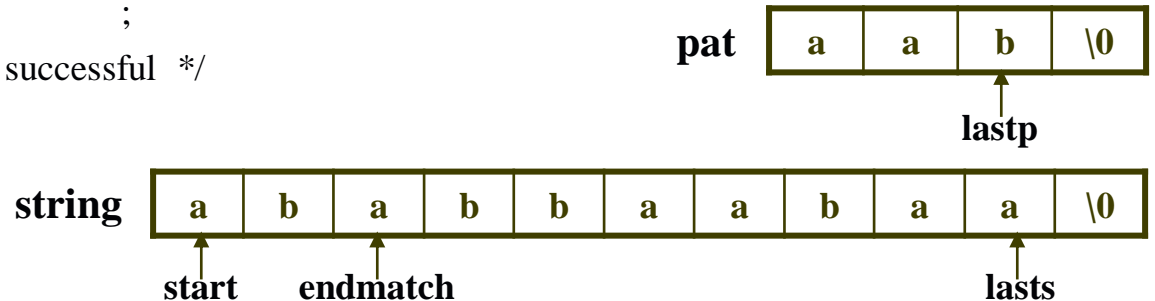
(g) match



```

int nfind(char * string, char *pat)
{
    /* match the last character of the 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;
}

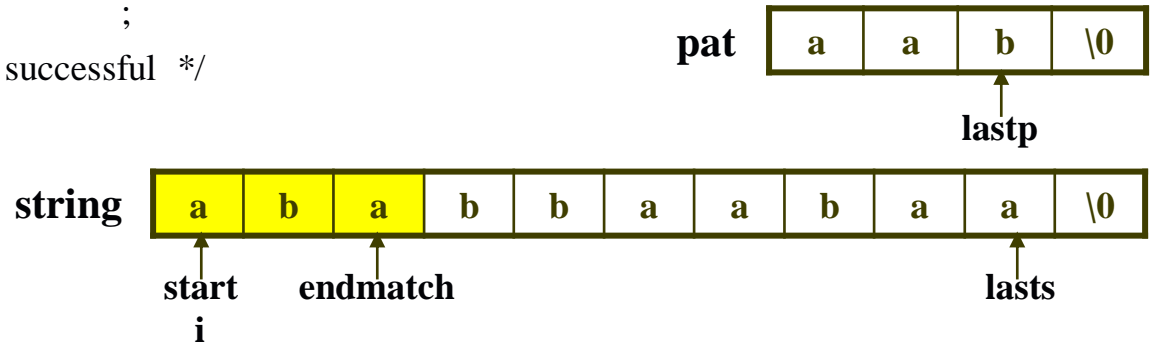
```



```

int nfind(char * string, char *pat)
{
    /* match the last character of the 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;
}

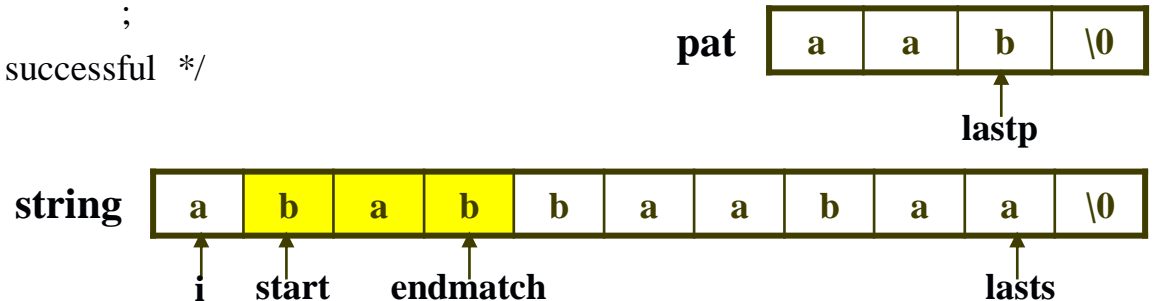
```



```

int nfind(char * string, char *pat)
{
    /* match the last character of the 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;
}

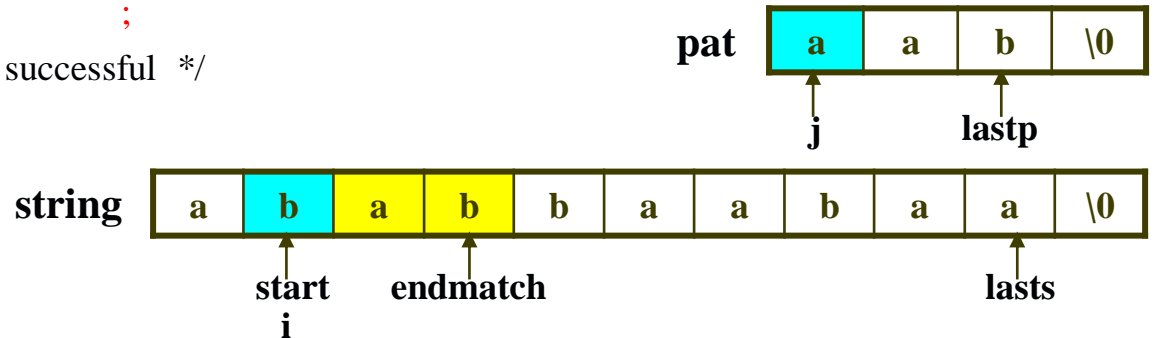
```



```

int nfind(char * string, char *pat)
{
    /* match the last character of the 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;
}

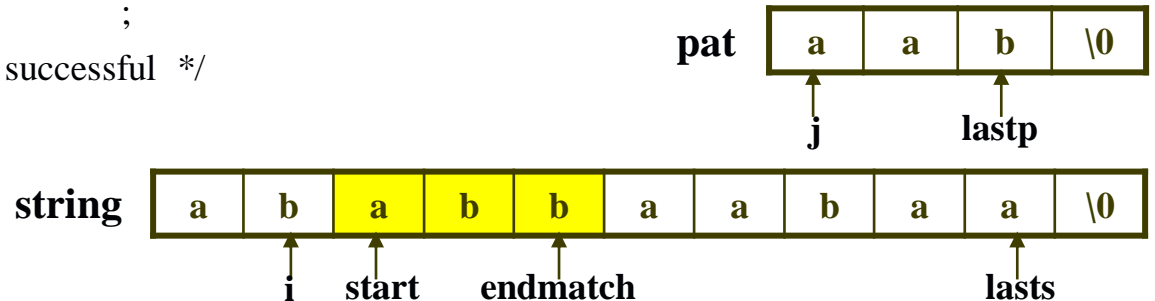
```




```

int nfind(char * string, char *pat)
{
    /* match the last character of the 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;
}

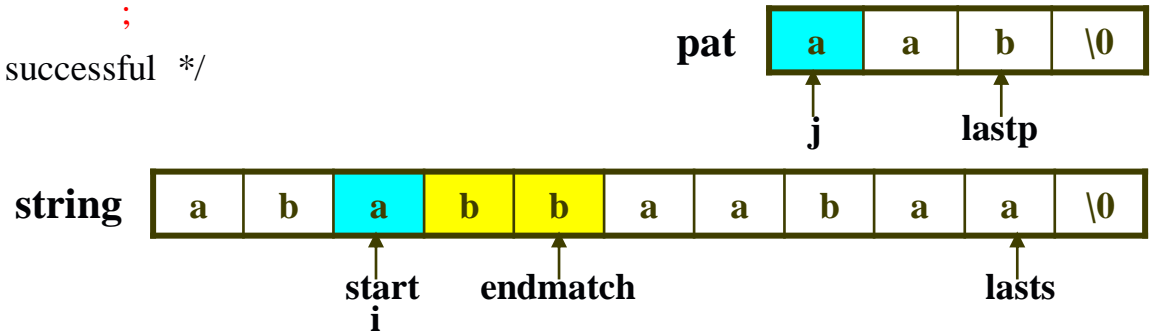
```



```

int nfind(char * string, char *pat)
{
    /* match the last character of the 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;
}

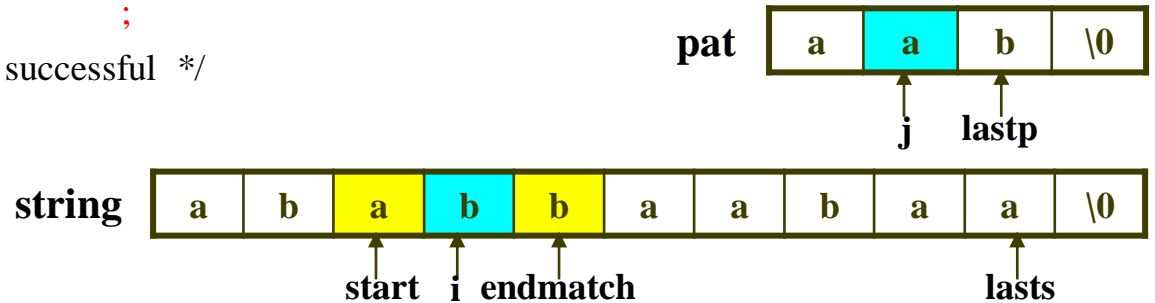
```



```

int nfind(char * string, char *pat)
{
    /* match the last character of the 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;
}

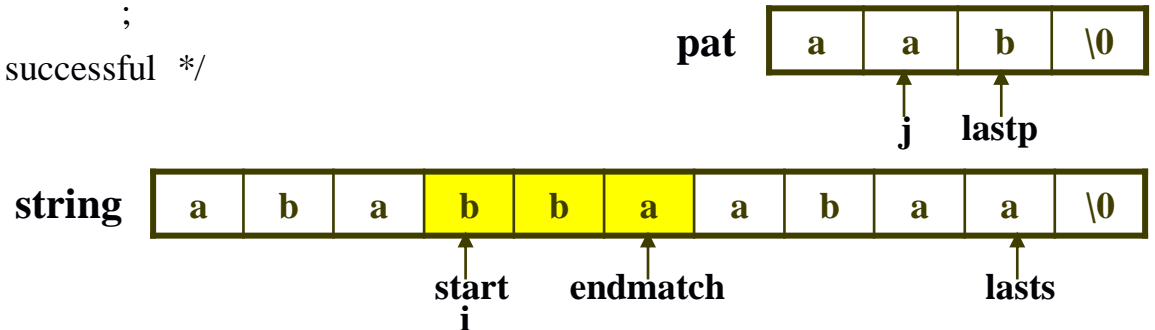
```



```

int nfind(char * string, char *pat)
{
    /* match the last character of the 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;
}

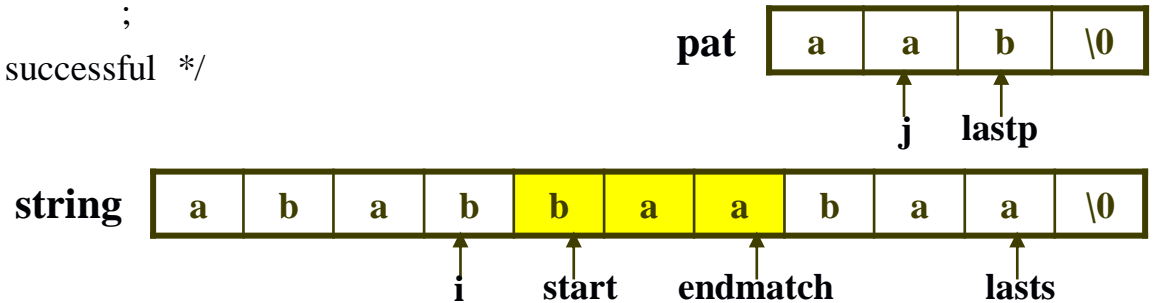
```



```

int nfind(char * string, char *pat)
{
    /* match the last character of the 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;
}

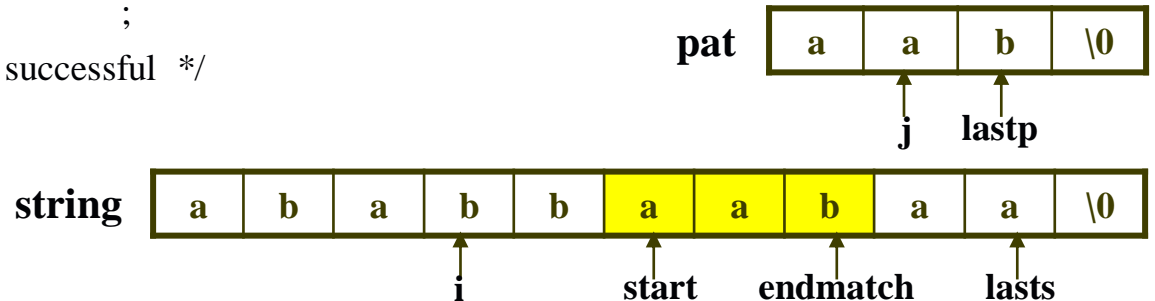
```



```

int nfind(char * string, char *pat)
{
    /* match the last character of the 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;
}

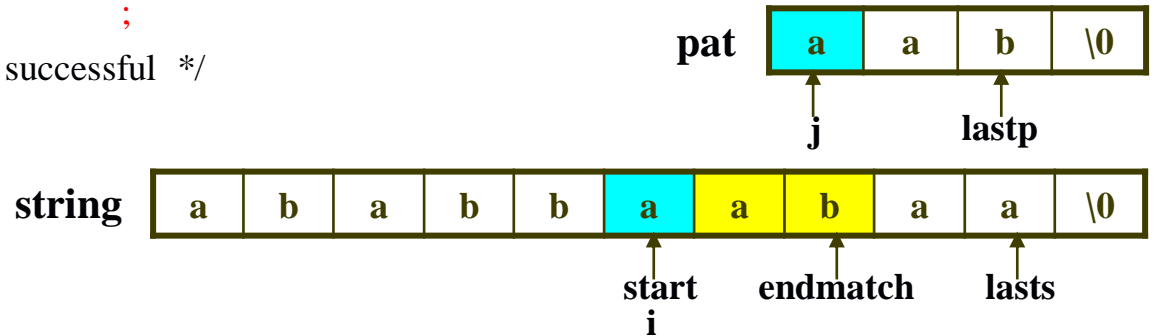
```



```

int nfind(char * string, char *pat)
{
    /* match the last character of the 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;
}

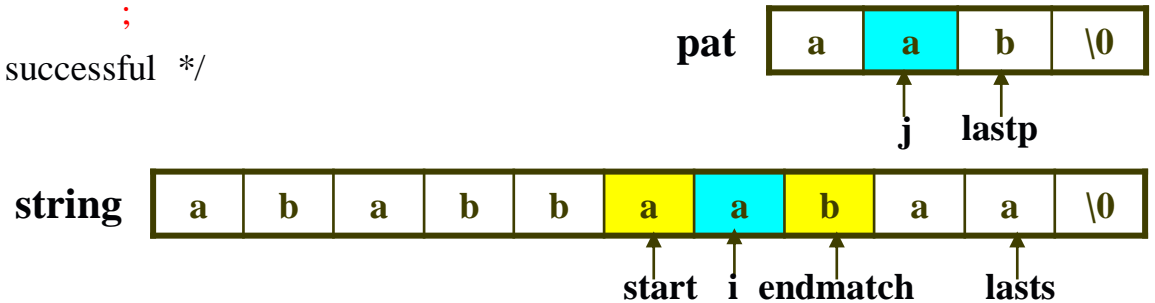
```



```

int nfind(char * string, char *pat)
{
    /* match the last character of the 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;
}

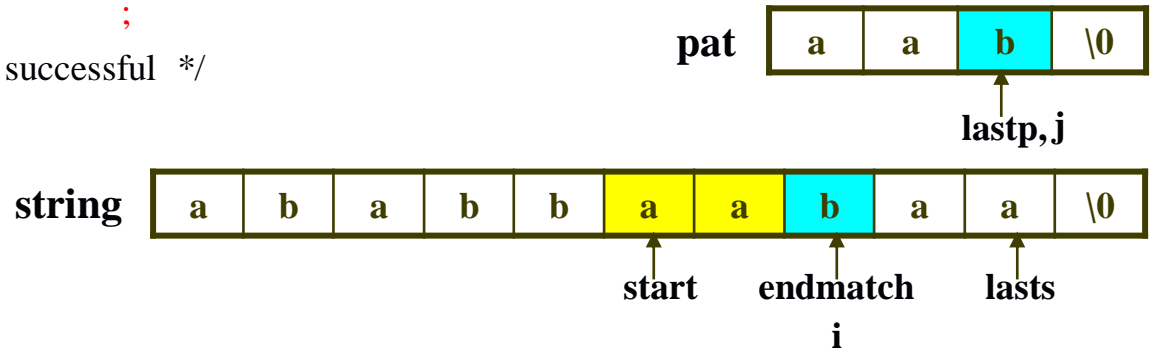
```




```

int nfind(char * string, char *pat)
{
    /* match the last character of the 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;
}

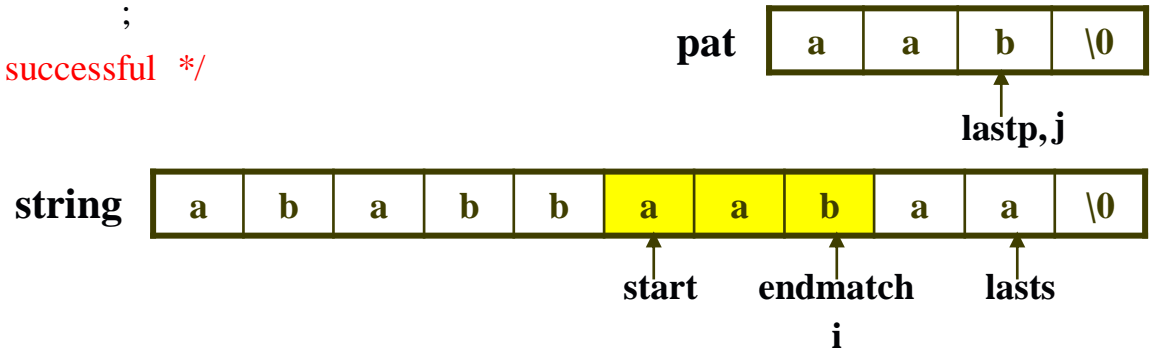
```



```

int nfind(char * string, char *pat)
{
    /* match the last character of the 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;
}

```



■ Analysis of *nfind* :

For *string* = "*aa...a*" and *pat* = "*aa...ab*", the computing time is $O(m)$.

Bur for *string* = "*aa...a*" and *pat* = "*aa...aba*",

the computing time is still $O(nm)$.

■ <KMP Algorithm>

- When a mismatch occurs, 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.
- We want to search the string for the pattern without moving backwards in the string

$pat = 'a \quad b \quad c \quad a \quad b \quad c \quad a \quad c \quad a \quad b'$

$p_0 \quad p_1 \quad p_2 \quad p_3 \quad p_4 \quad p_5 \quad p_6 \quad p_7 \quad p_8 \quad p_9$

if $s_i \neq p_0$, ?

if $s_i = p_0$ and $s_{i+1} \neq p_1$, ?

if $s_i = p_0$, $s_{i+1} = p_1$, and $s_{i+2} \neq p_2$, ?

if $s_i = p_0$, $s_{i+1} = p_1$, $s_{i+2} = p_2$, $s_{i+3} = p_3$, and $s_{i+4} \neq p_4$, ?

$pat = 'a \ b \ c \ a \ b \ c \ a \ c \ a \ b'$

$p_0 \ p_1 \ p_2 \ p_3 \ p_4 \ p_5 \ p_6 \ p_7 \ p_8 \ p_9$

if $s_i \neq p_0$, then proceed by comparing s_{i+1} and p_0

if $s_i = p_0$ and $s_{i+1} \neq p_1$, then proceed by comparing s_{i+1} and p_0

if $s_i = p_0$, $s_{i+1} = p_1$, and $s_{i+2} \neq p_2$,

then proceed by comparing s_{i+2} and p_0

(No need to compare with s_{i+1} as we already know that $s_{i+1} \neq p_0$)

if $s_i = p_0$, $s_{i+1} = p_1$, $s_{i+2} = p_2$, $s_{i+3} = p_3$, and $s_{i+4} \neq p_4$,

then proceed by comparing s_{i+4} and p_1

(We know that $s_{i+3} = p_3 = a = p_0$)

■ **Definition :**

If $p = p_0p_1p_2 \dots p_{n-1}$ is a pattern, then its *failure function*, f , is defined as :

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

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

■ **A rule for pattern matching :**

If a partial match is found such that $s_{i-j} \dots s_{i-1} = p_0p_1 \dots p_{j-1}$ 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 .

- Assumed declarations:

```
#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];
```

■ [Program 2.14]

```
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]

```
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);
}
```

string	b	a	b	c	a	b	...
--------	---	---	---	---	---	---	-----

pattern	a	b	c	a	c
---------	---	---	---	---	---

failure	-1	-1	-1	0	-1
---------	----	----	----	---	----

■ [Program 2.14]

```
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);
}
```

string	b	a	b	c	a	b	...
--------	---	---	---	---	---	---	-----

pattern	a	b	c	a	c
---------	---	---	---	---	---

failure	-1	-1	-1	0	-1
---------	----	----	----	---	----

■ [Program 2.14]

```

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);
}

```

string	b	a	b	c	a	b	...
--------	---	---	---	---	---	---	-----

pattern	a	b	c	a	c
---------	---	---	---	---	---

failure	-1	-1	-1	0	-1
---------	----	----	----	---	----

■ [Program 2.14]

```
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);
}
```

string	b	a	b	c	a	b	...
--------	---	---	---	---	---	---	-----

pattern	a	b	c	a	c
---------	---	---	---	---	---

failure	-1	-1	-1	0	-1
---------	----	----	----	---	----

■ [Program 2.14]

```
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;
}
```

string	b	a	b	c	a	b	...
--------	---	---	---	---	---	---	-----

pattern	a	b	c	a	c
---------	---	---	---	---	---

failure	-1	-1	-1	0	-1
---------	----	----	----	---	----

■ [Program 2.14]

```

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);
}

```

string	b	a	b	c	a	b	...
--------	---	---	---	---	---	---	-----

pattern	a	b	c	a	c
---------	---	---	---	---	---

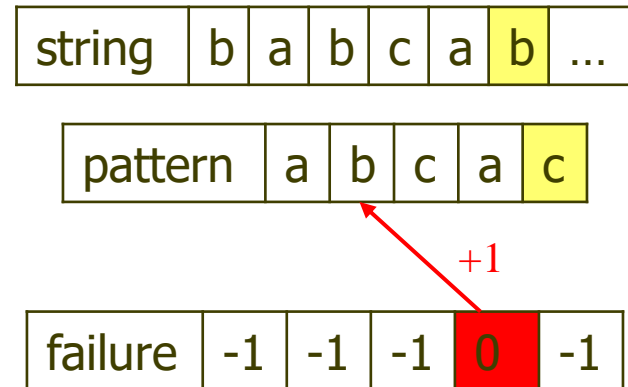
failure	-1	-1	-1	0	-1
---------	----	----	----	---	----

■ [Program 2.14]

```

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]

```
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);
}
```

string	b	a	b	c	a	b	...
--------	---	---	---	---	---	---	-----

pattern	a	b	c	a	c
---------	---	---	---	---	---

failure	-1	-1	-1	0	-1
---------	----	----	----	---	----

- Analysis of *pmatch* :

The *while* loop is iterated until the end of either the string or the pattern is reached.

In each iteration, one of the following three actions occurs :

- 1) increment *i*.
- 2) increment both *i* and *j*.
- 3) reset *j* to `failure[j-1]+1`
 - this cannot be done more than *j* is incremented by the statement `j++` as otherwise, *j* falls off the pattern.

Note that *j* cannot be incremented more than $m = \text{strlen}(\text{string})$ times.

Hence the complexity of *pmatch* is $O(m)$.

■ **Another definition of the failure function:**

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

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

■ [program 2.15]

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

■ [program 2.15]

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

pattern	a	b	a	a	b	a	b
failure	-1						

■ [program 2.15]

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

i = -1 **j = 1**

pattern	a	b	a	a	b	a	b
---------	---	---	---	---	---	---	---

failure	-1						
---------	----	--	--	--	--	--	--

■ [program 2.15]

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

i = -1 **j = 1**

pattern	a	b	a	a	b	a	b
---------	---	---	---	---	---	---	---

failure	-1						
---------	----	--	--	--	--	--	--

■ [program 2.15]

```

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

```

i = -1 **j = 1**

pattern	a	b	a	a	b	a	b
---------	---	---	---	---	---	---	---

failure	-1	-1					
---------	----	----	--	--	--	--	--

■ [program 2.15]

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

i = -1 **j = 2**

pattern	a	b	a	a	b	a	b
---------	---	---	---	---	---	---	---

failure	-1	-1					
---------	----	----	--	--	--	--	--

■ [program 2.15]

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

i = -1 **j = 2**

pattern	a	b	a	a	b	a	b
---------	---	---	---	---	---	---	---

failure	-1	-1					
---------	----	----	--	--	--	--	--

■ [program 2.15]

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

i = -1 j = 2

pattern	a	b	a	a	b	a	b
---------	---	---	---	---	---	---	---

failure	-1	-1	0				
---------	----	----	---	--	--	--	--

■ [program 2.15]

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

i = 0

j = 3

pattern	a	b	a	a	b	a	b
---------	---	---	---	---	---	---	---

failure	-1	-1	0				
---------	----	----	---	--	--	--	--

■ [program 2.15]

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

i = -1 **j = 3**

pattern	a	b	a	a	b	a	b
---------	---	---	---	---	---	---	---

failure	-1	-1	0				
---------	----	----	---	--	--	--	--

■ [program 2.15]

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

i = -1 **j = 3**

pattern	a	b	a	a	b	a	b
---------	---	---	---	---	---	---	---

failure	-1	-1	0	0			
---------	----	----	---	---	--	--	--

■ [program 2.15]

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

i = 0

j = 4

pattern	a	b	a	a	b	a	b
---------	---	---	---	---	---	---	---

failure	-1	-1	0	0			
---------	----	----	---	---	--	--	--

■ [program 2.15]

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

i = 0

j = 4

pattern	a	b	a	a	b	a	b
---------	---	---	---	---	---	---	---

failure	-1	-1	0	0			
---------	----	----	---	---	--	--	--

■ [program 2.15]

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

i = 0

j = 4

pattern	a	b	a	a	b	a	b
---------	---	---	---	---	---	---	---

failure	-1	-1	0	0	1		
---------	----	----	---	---	---	--	--

■ [program 2.15]

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

i = 1

j = 5

pattern	a	b	a	a	b	a	b
---------	---	---	---	---	---	---	---

failure	-1	-1	0	0	1		
---------	----	----	---	---	---	--	--

■ [program 2.15]

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

i = 1

j = 5

pattern	a	b	a	a	b	a	b
---------	---	---	---	---	---	---	---

failure	-1	-1	0	0	1		
---------	----	----	---	---	---	--	--

■ [program 2.15]

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

i = 1

j = 5

pattern	a	b	a	a	b	a	b
---------	---	---	---	---	---	---	---

failure	-1	-1	0	0	1	2	
---------	----	----	---	---	---	---	--

■ [program 2.15]

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

i = 2

j = 6

pattern	a	b	a	a	b	a	b
---------	---	---	---	---	---	---	---

failure	-1	-1	0	0	1	2	
---------	----	----	---	---	---	---	--

■ [program 2.15]

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

i = 0

j = 6

pattern	a	b	a	a	b	a	b
---------	---	---	---	---	---	---	---

failure	-1	-1	0	0	1	2	
---------	----	----	---	---	---	---	--

■ [program 2.15]

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

i = 0

j = 6

pattern	a	b	a	a	b	a	b
---------	---	---	---	---	---	---	---

failure	-1	-1	0	0	1	2	1
---------	----	----	---	---	---	---	---

■ Analysis of *fail* :

In each iteration of the *while* loop, the value of i decreases (by the definition of f).

The variable i is reset at the beginning of each iteration of the *for* loop.

However, it is either reset to -1

or it is reset to a value 1 greater than

its terminal value on the previous iteration.

Since the *for* loop is iterated only $n-1$ times,

the value of i has a total increment of at most $n-1$.

Hence it cannot be decremented more than $n-1$ times.

Consequently, the *while* loop is iterated at most $n-1$ times over the whole algorithm

Hence the complexity of *fail* is $O(n) = O(\text{strlen}(\text{pat}))$