제 2 장. 배열과 구조

- 배열을 연속적인 메모리 위치로만 생각하는 것은
  - 구현 측면만 강조한 생각 -> 효율적이지 않은 경우 발생
  - 항상 연속적인 메모리로 구현하지는 않음

- 배열은 : <index,value>의 쌍 집합
  - index -> value : 대응, 사상(mapping)
  - standard operations : retrieve, store

- 구조 2.1 ADT 정의
  - 연속적인 메모리 위치의 집합보다 더 일반적인 구조라는 사실을 명확히 나타냄

  - .

Structure *Array*

Objects : index의 각 값에 대하여 집합 item에 속한 한 값이 존재하는 <index, value>쌍의 집합. index는 일차원 또는 다차원의 유한 순서 집합이다. 예를 들면, 일차원의 경우 0, , n-1과 이차원 배열 (0,0), (0,1), (0,2), (1,0), (1,1), (1,2), (2,0), (2,1), (2,2) 등

Functions : 모든 A ∈ *Array, i* ∈ *index, x* ∈ *item, j, size* ∈ *integer*

Array Create(j, list) ::= return j차원의 배열.
여기서 list는 i번째 원소가 i번째 차원이고,
크기 j-tuple이며 item들은 정의되지 않았음.

Item Retrieve(A, i) ::= if (*i* ∈ *index*)
return 배열 A의 인덱스 i값과 관련된 항목.
else return 에러.

Array Store(A,i,x) ::= if (*i* ∈ *index*)
return 새로운 쌍<*i, x*>가 삽입된 배열 A.
else return 오류

end *Array*

- C의 일차원 배열

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

정수값 : list[0], list[1], list[2], list[3], list[4]

정수포인터 : plist[0], plist[1], plist[2], plist[3], plist[4]

변수	메모리 주소
list[0]	기본주소 = a
list[1]	a + sizeof(int)
list[2]	a + 2 sizeof(int)
list[3]	a + 3 sizeof(int)
list[4]	a + 4 sizeof(int)

- 포인터 해석

int *list1; int list2[5];

- list2는 list2[0]를 가리키는 포인터
- list2 + I는 list2[i]를 가리키는 포인터
- (list2+i) == &list2[i], *(list2+i) == list2[i]

```
#define MAX_SIZE 100
float sum(float [], int);
float input[MAX_SIZE], answer;
int i;
void main(void)
{
   for (i = 0; i < MAX_SIZE; i++)
      input[i] = i;
   answer = sum(input, MAX_SIZE);
   printf("The sum is: %f", answer);
}
float sum(float list[], int n)
{
   int i;
   float tempsum = 0;
   for (i = 0; i < n; i++)
      tempsum += list[i];
   return tempsum;
 } // 프로그램 2.1 배열 프로그램의 예
```

호출시 input(=&input[0])은 임시 장소에 복사

- 형식 매개 변수 list와 연관

역참조(dereference)

- list[i]가 "="기호 우측 (list + i)가 가리키는 값 반환
- list[i]가 "="기호 좌측 값을 (list + i) 위치에 저장

- C 언어의 매개변수 전달방식

  call-by-value임에도 불구하고 배열 매개변수는 그 값을 변경함

예제 2.1 [일차원 배열의 주소 계산] :

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

```
void print1(int *ptr, int row)
{/* 포인터를 사용한 일차원 배열의 출력 */
    int i;
    printf("Address Contents");
    for (i=0; i<rows; i++)
        printf("%8u%5d", ptr+i, *(ptr+i) );
     printf("");
}
```

주소	내용
1228	0
1230	1
1232	2
1234	3
1236	4

구조 및 유니언

- 구조(structure) : struct
  타입이 다른 데이타를 그룹화
  데이타 항목의 집단 – 각 항목은 타입과 이름으로 식별

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

구조의 멤버 연산자

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

- typedef 문장을 사용한 구조 데이타 타입 생성

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

- 변수 선언

```
    human_being  person1,  person2 ;
    if (strcmp(person1.name, person2.name))
        printf("두 사람의 이름은 다르다.");
    else
        printf("두 사람의 이름은 같다.");
```

- 전체 구조의 동등성 검사 : if (person1 == person2)
- 구조 치환 : person1 = person2

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

- 구조체 속의 또 다른 구조체 정의

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

[예] human_being  person1 ;
        person1.dob.month = 2;
        person1.dob.day = 11;
        person1.dob.year = 1944;
```

- 유니언(union) : union의 필드들은 메모리 공간을 공용
  - 한 필드만이 어느 한 시점에 활동적이 되어 사용 가능

```
typedef struct sex_type {
enum tag_field {female, male} sex;
union {
        int children;
        int beard;
        } u ;
 } ;
typedef struct human_being {
    char name[10];
    int age;
    float salary;
    date dob;
    sex_type sex_info;
   } ;
human_being person1, person2;
```

- 값 할당

```
person1.sex_info.sex = male; person1.sex_info.u.beard = FALSE;
person2.sex_info.sex = female; person2.sex_info.u.children = 4;
```

- 구조의 내부 구현

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

  채워넣기, 가장 큰 구성 요소에 맞추기, 메모리 경계에서 시작하고 끝나도록 함

- 자체참조 구조 (self_referential structure)
  - 구성 요소 중 자신을 가리키는 포인터가 존재하는 구조

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

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

item1.link = &item2;        구조들을 서로 연결
item2.link = &item3;        (item1 → item2 → item3)
```

다항식 추상 데이타 타입

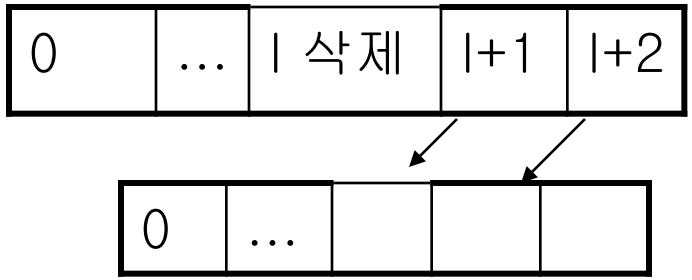
- 순서 리스트, 선형 리스트
  - 원소들의 순서가 있는 모임
  - examples
    - 한 주일의 요일: (일,월,화,수, .., 토)
    - 카드 한 벌의 값: (Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King)
    - 건물 층: (지하, 로비, 1층, 2층)
    - 스위스의 2차 세계대전 참전 년도: ()
- 리스트 형태: $(a_0, a_1, \ldots, a_{n-1})$
- 공백 리스트: () -> 포함된 항목이 없음

- 리스트에 대한 연산
  - 길이 n의 계산
  - 리스트의 항목을 왼쪽에서 오른쪽 (오른쪽에서 왼쪽)으로 읽기
  - i번째 항목을 가져오기, $0 \le i < n$
  - i번째 항목을 저장, $0 \le i < n$
  - i번째 위치에 새 항목 삽입, $0 \le i < n$, l번째 부터 뒤로 밀림
  - i번째 항목을 삭제, $0 \le i < n$, l+1부터 당겨짐

- 순서 리스트의 구현
  - 순차적 사상 : 물리적으로 인접한 배열을 이용

    인덱스 $i \rightarrow a_i$, $0 \le i < n$

    문제점: 삽입, 삭제 시 오버헤드



  - 비순차 사상 : 비연속적 기억장소 위치, 4장 리스트

순서리스트 사용 예: 다항식 (polynomial)

- A(x) = $\sum a_i \, x^{e_i}$ 다항식
  a_i : 계수(coefficient)
  e_I : 지수(exponent)
  x : 변수(variable)
  A(x) = $3x^{20} + 2x^5 + 4$,
  B(x) = $x^4 + 10x^3 + 3x^2 + 1$

- 차수(degree): 가장 큰 지수
- 다항식의 합과 곱
  - $A(x) = \sum a_i x^i$
  - $B(x) = \sum b_i x^i$
  - $A(x) + B(x) = \sum(a_i + b_i)x^i$
  - A(x) B(x) = $\sum(a_i x^i \, . \sum(b_j x^j))$

polynomial 추상 데이타 타입

Structure *Polynomial*

objects : $P(x)=a_1 x^{e1} + \ +a_n x^{en}$; $<e_i , a_i>$의 순서쌍으로 된 집합, 여기서 $a_i \in$Coefficient 이고, $e_i \in$ *Exponent,* Exponent는 ≥0 인 정수로 가정

Functions : 모든 *poly, poly1, poly2* ∈ *polynomial, coef* ∈ *Coefficients, expon* ∈ *Exponents*에 대해

Polynomial Zero() ::= return 다항식, *p(x) = 0*

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

Coefficient Coef(poly, expon) ::= if (*expon* ∈ *poly*) return 계수
else return 0

Exponent Lead_Exp(poly) ::= return *poly*에서 제일 큰 지수

Polynomial Attach(poly, coef, expon) ::= if (*expon* ∈ *poly*) return 오류
else return <*coef, exp*>항이 삽입된 다항식 *poly*

Polynomial Remove(poly, expon) ::= if (*expon* ∈ *poly*)
return 지수가 *expon*인 항이 삭제된 다항식 *poly*
else return 오류

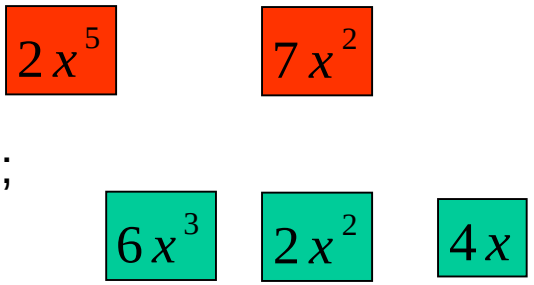
Polynomial SingleMult(poly, coef, expon) ::= return 다항식 *poly coef* x^{expon}

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

Polynomial Mult(poly1, poly2) ::= return 다항식 *poly1 poly2*

- 표현에 독립적인 함수 padd 초기 버전

```
/* d = a + b, 여기서 a, b, d는 다항식이다. */
d = Zero();
while (! IsZero(a) && ! IsZero(b)) do  {
    switch COMPARE(Lead_Exp(a), Lead_Exp(b))  {
        case -1:
            d = Attach (d, Coef(b, Lead_Exp(b)), Lead_Exp(b));
            b = Remove(b, Lead_Exp(b));
            break;
        case 0:
            sum = Coef(a, Lead_Exp(a)) + Coef(b, Lead_Exp(b));
            if (sum)
                    Attach(d, sum, Lead_Exp(a));
            a = Remove(a, Lead_Exp(a));
            b = Remove(b, Lead_Exp(b));
            break;
        case 1:
            d = Attach(d, Coef(a, Lead_Exp(a)), Lead_Exp(a));
            a = Remove(a, Lead_Exp(a));
    }
}
```



a 또는 b의 나머지 항을 d에 삽입한다.

- [표현 1] 모든 차수에 대한 계수만 저장
  - 지수들은 내림차순으로 정돈

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

  - *a* 가 polynomial 타입이고 n < MAX_DEGREE 이면,
  - 다항식 $\sum a_i x^i$는 *a.degree = n, a.coef[i] =* a_{n-i}, *0<= i <= n*

    예) *A(x) = x^4+10x^3+3x^2+1* : n = 4

    A = (4, 1, 10, 3, 0, 1) : 6 elements

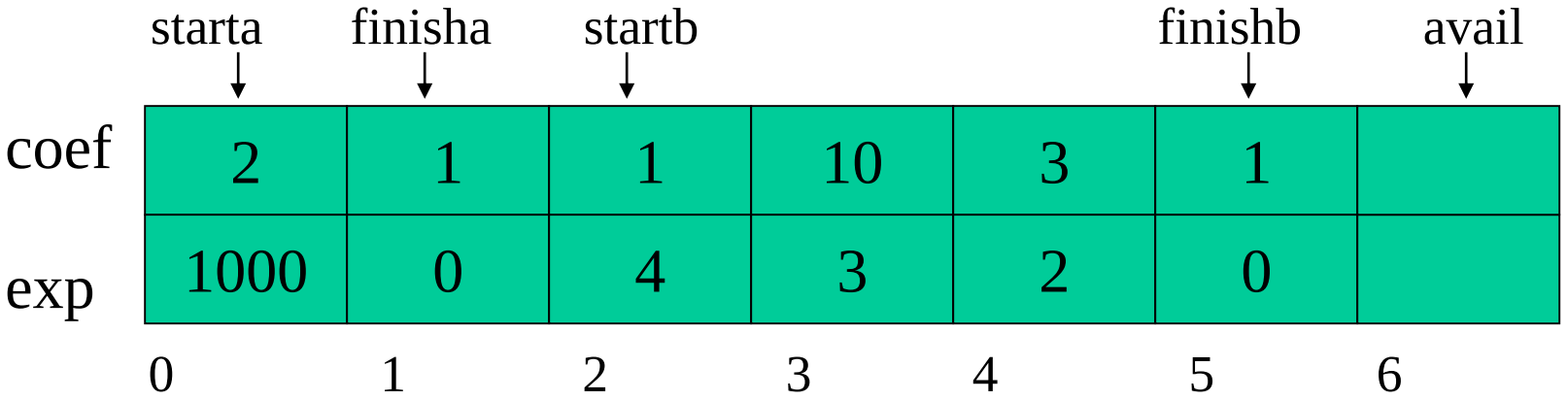
- 표현 1의 장단점
  - 연산을 위한 알고리즘은 간단하나, 많은 기억 공간의 낭비 초래
  - n << MAX_DEGREE이거나, 희소 다항식

- a.degree << MAX_DEGREE
  - a.coef[MAX_DEGREE]의 대부분이 필요 없음
- A(x) = x^1000 + 1 : n = 1000
  A = (1000, 1, 0,..., 0, 1) : 1002 elements
  999

- 다항식의 표현 2
  - 모든 다항식을 저장하는 전역배열 사용
  - 0 이 아닌 계수-지수 쌍만 저장

```
#define MAX_TERMS 100 /* 항 배열의 크기 */
 typedef struct  {
        float coef;
        int expon;
       }  Polynomial;
 Polynomial terms[MAX_TERMS];
 int avail = 0;
```

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



- 명세에서의 다항식 poly는 표현에서는 <start, finish>쌍으로 변환
- n개의 0이 아닌 항을 가진 다항식 A는
  - finish =start+n-1을 만족
  - terms에 저장될 수 있는 다항식수는 0이 아닌 항의 수가 MaxTerms을 넘지 않는 한도 내 표현 가능
  - 많은 항이 0인 경우 우수하나, 모든 항이 0이 아닌 경우 표현 2보다 두 배의 저장 장소 사용

```
 다항식 덧셈 D = A + B
void padd(int starta, int finisha, int startb, int finishb, int *startd, int *finishd);
{   /* A(x) 와 B(x)를 더하여 D(x)를 생성한다 */
    float coefficient;
    *startd = avail;
    while (starta <= finisha && startb <= finishb)
        switch (COMPARE(terms[starta].expon, terms[startb].expon)) {
            case -1: /* a의 expon이 b의 expon보다 작은 경우 */
                     attach(terms[startb].coef, terms[startb].expon);
                     startb++;  break;
            case 0: /* 지수가 같은 경우 */
                     coefficient = terms[starta].coef + terms[startb].coef;
                     if (coefficient) attach(coefficient, terms[starta].expon);
                     starta++;  startb++; break;
            case 1: /*  a의 expon이 b의 expon보다 큰 경우 */
                     attach(terms[starta].coef, terms[starta].expon);
                     starta++;
        }
    /* A(x)의 나머지 항들을 첨가한다 */
    for(; starta <= finisha; starta++)
        attach(terms[starta].coef, terms[starta].expon);
    /* B(x)의 나머지 항들을 첨가한다 */
    for(; startb <= finishb; startb++)
        attach(terms[startb].coef, terms[startb].expon);
    *finishd = avail-1;
}
```

```
void attach(float coefficient, int exponent)
{     /* 새 항을 다항식에 첨가한다. */
    if (avail >= MAX_TERMS) {
        fprintf(stderr, "다항식에 항이 너무 많다.");
        exit(1);
    }
    terms[avail].coef = coefficient;
    terms[avail++].expon = exponent;
}
```

- 알고리즘 padd 의 분석 :

  m, n (>0) : 각각 A와 B의 0이 아닌 항의 수

  while 루프

  - 각 반복마다 starta나 startb 또는 둘 다 값이 증가
  - 반복 종료 -> *starta <= finisha && startb <= finishb*
  - 반복 횟수는 최대 m+n-1

  나머지 두 루프 : 첫번째는 m번 이하, 두번째는 n번 이하

  전체 연산시간 = $O(n+m)$

- 문제점

  avail = = MAX_TERMS 일때

  불필요한 다항식 제거후

  배열 끝에 연속적인 가용공간 생성 (압축 함수)

  - 데이타 이동시간, 각 다항식의 *starti*, *finishi* 변경

희소 행렬(Sparse matrix)

- 행렬: 행과 열로 이루어진 숫자들
- 행렬의 2차원 배열 표현
  - A[m][n] : m × n 행렬, m : 행의 수, n : 열의 수
  - m × n : 원소의 수, m = n : 정방 행렬
  - 0인 원소가 많은 경우 희소 행렬(sparse matrix) 일 경우 → 0이 아닌 원소만 저장할 필요 있음
  - 그림 2.3의 (b)가 sparse matrix

- 행렬에 대한 연산
  - creation, transposition, addition, multiplication

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

(a)

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

(b)

희소 행렬 ADT

structure Sparse_Matrix
  objects : 3원소쌍 <행, 열, 값>의 집합이다. 여기서, 행과 열은 정수이고
        이 조합은 유일하며, 값은 item 집합의 한 원소이다.
  functions : 모든 a, b ∈ Sparse_Matrix, x ∈ item,
            I, j, max_col, max_row ∈ index에 대해,
   Sparse_Matrix Create(max_row, max_col) ::= return max_items =
                max_row x max_co까지 저장할 수 있는 SparseMatrix
   Sparse_Matrix Transpose(a) ::= return 모든 3원소 쌍의 행과 열의 값을
                서로 교환하여 얻은 행렬
   Sparse_Matrix Add(a, b)   ::= if a와 b의 차원이 같으면
                return 대응 항들, 즉 같은 행과 열의 값을
                  가진 항들을 더해서 만들어진 행렬
                else return 오류
   Sparse_Matrix Multiply(a, b) ::= if a의 열의 수와 b의 행의 수가 같으면
                return 다음 공식에 따라 a와 b를
                  곱해서 생성된 행렬 d를 반환하고
                else 오류
    //   행렬 d : $d(i,j)=\sum (a[i][k] \times b[k][j])$ 여기서 d(i,j)는 (i,j)번째 요소

- 희소행렬 표현 방법 설계
  - 0이 아닌 원소만 표시 → <행, 열, 값> 3원소 쌍 모두 표시하여야 함
  - 행의 인덱스를 오름차순으로 배치하여 전치 연산 효율적으로 수행하도록 하고, 행 내에서 열번호도 오름차
  - a[0].row = 행의 수, a[0].col = 열의 수,
  - a[0].value = non-zero 원소의 수도 저장

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

#define MAX_TERMS 101 /* 최대 항의 수 + 1 */
typedef struct {
        int col ;
        int row ;
        int value ;
        } term ;
term a[MAX_TERMS] ;
```

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

(a)

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

(b) transpose

그림 2.4 원소쌍으로 저장된 희소 행렬과 전치 행렬

행렬의 전치 (transpose)

- 행렬 각 행 i에 대해서 원소 (i, j, 값)이 전치행렬에서는 원소 (j, i, 값)
- 2차원 배열을 사용한 일반적인 행렬표현의 전치

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

- 우리가 선택한 희소행렬 표현법에서의 행렬의 전치 알고리즘 (첫번째)

```
for (each row i)
    take element (I,j,value) and
    store it in (j,I,value) of the transpose
```

  - 문제점: <j, i, 값>을 저장할 위치는 앞선 다른 모든 원소를 처리하기 전에는 찾기 어려움

(예)

(0, 0, 15) -> (0, 0, 15)
(0, 3, 22) -> (3, 0, 22)
(0, 5, -15) -> (5, 0, -15)
(1, 1, 11) -> (1, 1, 11)

- 전치행렬에서 원소의 위치를 결정하는 방법
  - 열 인덱스를 사용하여, 열번호 0의모든 원소를 전치행렬 행 0에 저장하고, 열 1을 행 1에 저장하는 다음의 알고리즘 사용
  - 원래 행렬이 행 우선으로 정렬되어 있으므로, 전치 행렬의 각 행에 있는 열 역시 오름차순으로 정렬됨
  - a의 열 I로 부터 0이 아닌 항을 수집하는 것은 b의 행 I에 대해 0이 아닌 항을 수집하는 것이 된다.

  열 j에 있는 모든 원소에 대해
  원소 (I,j,값)을 원소 (j,I,값)에 저장

- 프로그램 2.7 희소행렬의 전치
  - currentb : 다음 전치될 항이 저장될 b의 위치

```
void transpose(term a[], term b[]) // 희소 행렬의 전치
{ /* a를 전치시켜 b를 생성 */
    int n, i, j, currentb;
    n = a[0].value;     /* 총 원소 수 */
    b[0].row = a[0].col; /* b의 행 수 = a의 열 수 */
    b[0].col = a[0].row; /* b의 열 수 = a의 행 수 */
    b[0].value = n;
    if (n > 0) {   /* 0이 아닌 행렬 */
        currentb = 1;
        for (i =0; i < a[0].col; i++) /* a에서 열별로 전치*/
          for (j = 1; j <= n; j++)   /* 현재의 열로부터 원소를 찾는다. */
              if (a[j].col ==i ) {
              /* 현재 열에 있는 원소를 b에 첨가 */
                  b[currentb].row = a[j].col;
                  b[currentb].col = a[j].row;
                  b[currentb].value = a[j].value;
                  currentb++;
              }
     }
}
```

- program 2.7의 연산 시간
  - O(elements•columns)

  -> elements 가 rows*columns일 때, O(rows·columns ·columns)이 되어, 공간 절약을 위해 시간을 희생

- 단순 2차원 배열 표현 시는 O(rows·columns)

```
for (int j = 0; j < columns; j++)
    for (int i = 0; i < rows; i++)
        B[j][i] = A[i][j];
```

- 메모리를 조금 더 사용하여 연산 시간이 O(columns+ elements)인 개선 알고리즘: FastTranspose

- FastTranspose
  - 행렬 a의 각 열의 원소가 옮겨질 위치를 미리 계산 → 전치 행렬 b의 각 행의 원소 수를 결정 : row_terms[]
  - row_terms[]를 이용 전치행렬 b의 각행의 시작위치 starting_pos[] 구함
  - 원래 행렬 a에 있는 원소를 하나씩 전치 행렬 b의 올바른 위치로 옮김

	row_terms	starting_pos
[0]	2	1
[1]	1	3
[2]	2	4
[3]	2	6
[4]	0	8
[5]	1	8
	↑	↑
	# of elements in b's row (a's col)	starting position of b's row

```
#define MAX_COL  50  /* 최대 열의 수 + 1 */
void fast_transpose(term a[], term b[])
{     /* a를 전치시켜 b에 저장 */
    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) { /* 0이 아닌 행렬 */
        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;
        }
    }
}
```

- 연산 시간
  - 첫번째 for loop: O(num_cols)
  - 두번째 for loop: O(num_terms)
  - 세번째 for loop: O(num_cols)
  - 네번째 for loop: O(num_terms)
  - 합하면 O(columns + elements)

- 이차원 배열 표현과 비교
  - elements가 columns • row일 경우 상수인자가 크지만 연산시간은 O(columns • row)로 동일
  - 원소의 수가 columns • row 보다 작을 경우 fast_transpose가 시간과 공간이 절약됨
  - transpose()는 연산시간이 O(elements•columns)이고, elements는 작더라도 {columns, rows}의 최대값보다 보통 크므로 연산시간은 elements•columns > rows•columns가 되어, 이차원 배열표현보다 작지 않다.

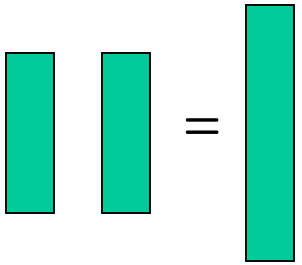
행렬 곱셈

- D_m×p ← A_m×n × B_n×p

$$d_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj} \qquad ,0 \le i < m, 0 \le j < p$$

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

$$\begin{bmatrix} 2 & 1 & 0 \\ 3 & 5 & 6 \\ 1 & 0 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 3,3,6 \\ 0,0,2 \\ 0,1,1 \\ 1,0,3 \\ 1,1,5 \\ 1,2,5 \\ 2,0,1 \end{bmatrix}$$



- 순서 리스트로 표현된 두 희소 행렬의 곱셈
  - D의 (I,j)원소는 A의 I행과 B의 j열의 원소를 곱하여 합함
  - B의 j열을 원소를 계산할 때마다 찾기 보다는 transpose를 하여 j행으로 바꾸고 사용
  - A의 i행과 B의 j열의 원소들이 정해지면 다항식 덧셈과 유사한 합병연산 수행

Program 2.9

- storesum ()함수는 d에 하나의 3원소쌍을 저장하고, sum을 0으로 재설정

- row : 현재 곱해질 A의 행
- row_begin : 현재 행 row의 처음 원소의 인덱스
- column : 현재 곱해질 B의 열
- totald : 곱셈 결과 행렬인 D내에 있는 원소의 현재 갯수
- i, j : A의 행, B의 열 원소를 차례대로 처리하는 인덱스
- new_b는 행렬 b의 전치 행렬

- end condition을 위한
  - a[totald+1].row = rows_a; new_b[totald+1].row = cols_b;

```
void storesum(term d[], int *totald, int row, int column, int *sum){
//sum≠0이면 행과 열의 위치와 함께 행렬 d의 (*totald+1)원소로 저장
// *totald는 증가한다.
// 새로운 항을 위한 가용 메모리가 없으면 에러

    if (*sum)
        if (*totald  < MAX_TERMS) {
            d[++*totald].row = row ;
            d[*totald].col = column ;
            d[*totald]. value = *sum ;
            *sum = 0 ;
        }
        else {
            fprintf (stderr, "Number of terms in product exceeds %d₩n”,
                             MAX_TERMS) ;
            exit (1) ;
        }
}
```

```
void mmult (term a[], term b[], term d[])
{// 두 개의 희소 행렬 a와 b를 곱하여 d를 생성
        int i, j, column, tatalb = 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, totalb = b[0].value;
        int row_begin = 1, row = a[1].row, sum = 0 ;
        int new_b[MAX_TERMS];

        if (cols_a != b[0].row) {
                fpritf(stderr, “Incompatible matricses₩n”); exit(1);
        }

        fast_transpose(b, new_b);

        /* 경계조건 설정 */
        a[totala+1].row = rows_a;
        new_b[totalb+1].row = cols_b;
        new_b[totalb+1].col = 0;
```

```
for (i = 1; i <= totala; ) {
        column = new_b[1].row;
        for (j = 1; j <= totalb + 1; ){// a의 행 row에 b의 열 column를 곱함
          if (a[i].row != row) { //a의 row행을 모두 사용
                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) {
                                        //new_b의 column행 모두 사용
                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: /* a의 다음 항으로 이동 */
                       i++;
                       break;
                    case 0:
                    /* 항을 더하고, a와 b를 다음 항으로 이동 */
                       sum += (a[i++].value * new_b[j++].value);
                       break;
                    case 1: /* b의 다음 항으로 이동 */
                       j++;
               }
          } /* for j <= totalb + 1문의 끝 */
          for (; a[i].row == row; i++) )//다음 행으로 감
                    ;
          row_begin = i;  //다음 행의 처음원소의 a[]내의 인덱스
          row = a[i].row;
    } /* end of for I <= totala */
    d[0].row = rows_a;
    d[0].col = cols_b; d[0].value = totald;
}
```

- mmult 알고리즘의 분석
  - fast_transpose : O(cols_b + totalb)
  - 내부 for : i나 j 또는 둘 다 증가하거나, i와 column0이 재설정
    - totalb(j의 증가) + cols_b * termsrow (i를 row_begin으로 재설정하고 column0이 다음 행으로 전진) + cols_b(column0이 다음 행으로 전진)
  - 외부 for도 고려 $O(\Sigma_{row}(cols\_b*termsrow + totalb))$
    $= O(cols\_b*totala + rows\_a*totalb)$

- 표준 배열 표현법을 사용할 경우

```
 for (i = 0; i < row_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;
}
```

  - Complexity: O(rows_a * cols_a * cols_b)

- Non-sparse matrix
  - totala = rows_a * cols_a
  - totalb = rows_b * cols_b