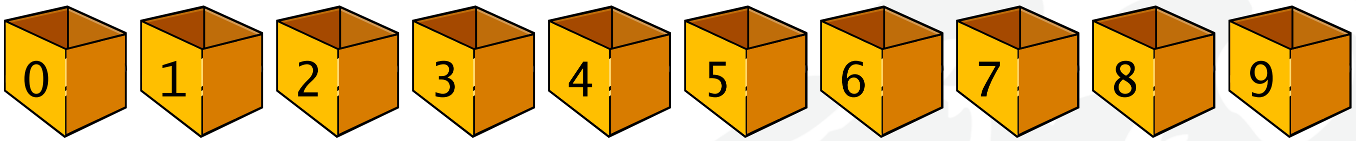
CSE2010 자료구조론

Week 2: Array

ICT융합학부 한진영

배열

- 동일한 데이터 타입의 데이터를 여러 개 만드는 경우에 사용
  - int A0, A1, A2, A3, …,A9;
  - int A[10];



- 반복 코드 등에서 배열을 사용하면 효율적인 프로그래밍이 가능
  - 예: 최대값을 구하는 프로그램, 만약 배열이 없었다면?

```
tmp=score[0];
for(i=1;i<n;i++){
        if( score[i] > tmp )
                tmp = score[i];
}
```

배열 ADT

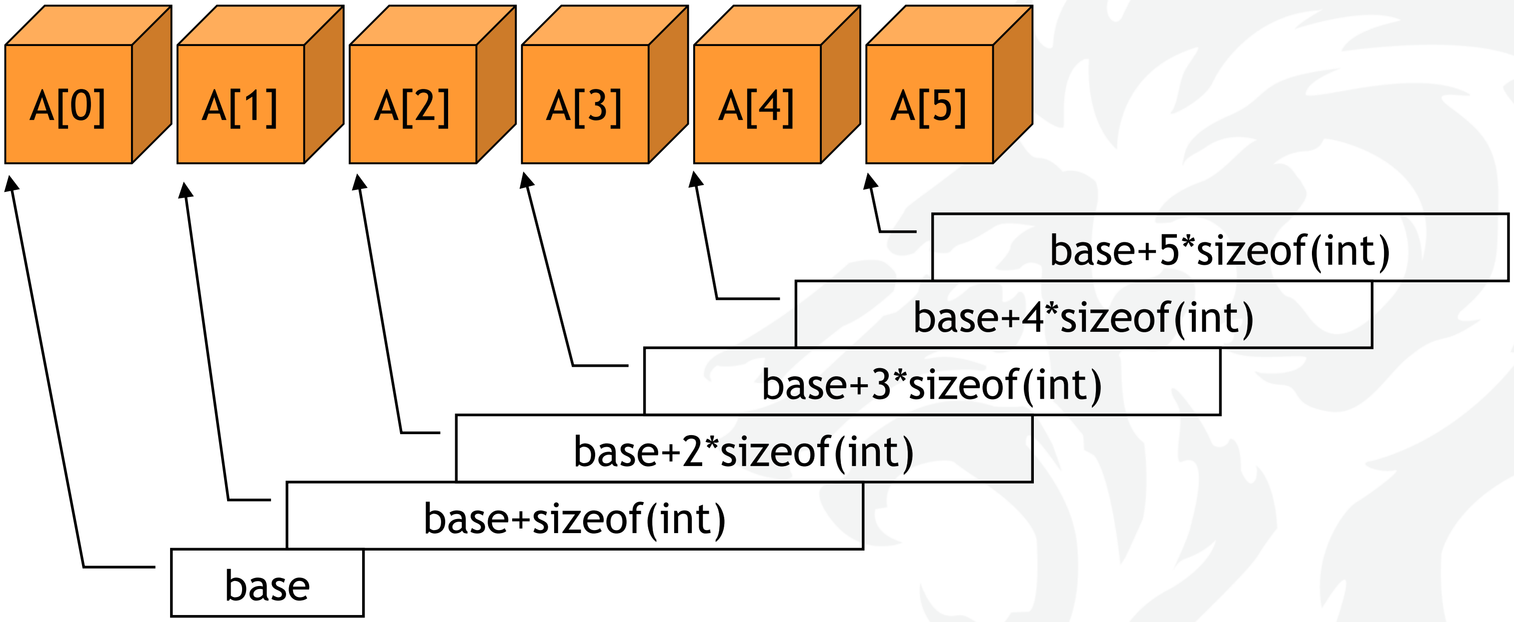
- 배열 ADT

배열 ADT

객체: <인덱스, 요소> 쌍의 집합
연산:
- create(n) ::= n개의 요소를 가진 배열의 생성.
- retrieve(A, i) ::= 배열 A의 i번째 요소 반환.
- store(A, i, item) ::= 배열 A의 i번째 위치에 item 저장.

1차원 배열

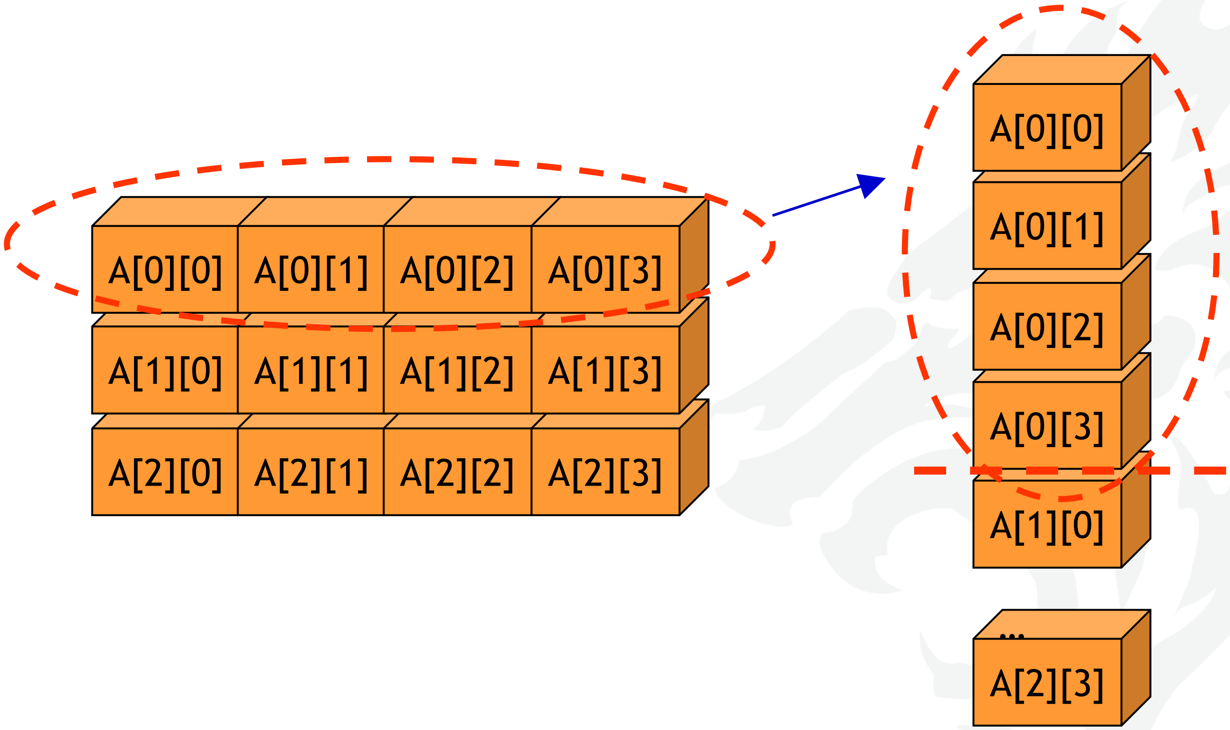
- int A[6];



- 배열 인덱스는 0부터 시작
- base: 메모리상의 기본 주소
- 배열은 메모리의 연속된 위치에 구현 되기때문에, A[0]의 주소가 base가 됨

2차원 배열

- int A[3][4];



실제 메모리 안에서의 위치

배열의 응용: 다항식

- 다항식의 일반적인 형태

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

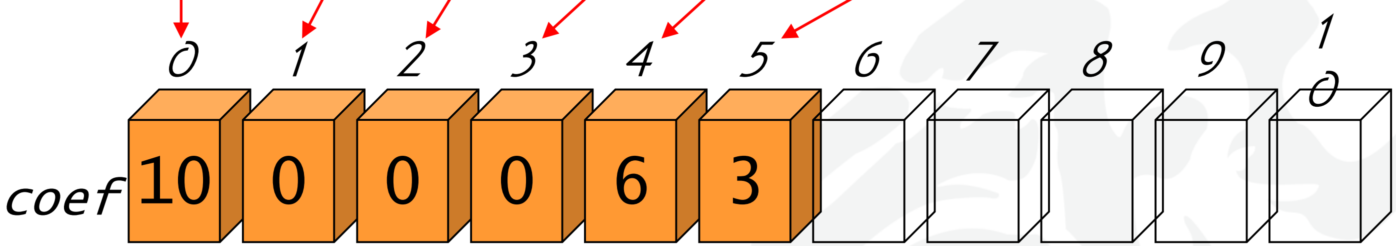
- 프로그램에서 다항식을 처리하려면 다항식을 위한 자료구조가 필요함
  - 어떤 자료구조를 사용해야 다항식의 덧셈, 뺄셈 ,곱셈, 나눗셈 등의 연산을 할 때 편리하고 효율적일까?

- 배열을 사용한 2가지 방법
  - 다항식의 모든 항을 배열에 저장
  - 다항식의 0이 아닌 항만을 배열에 저장

다항식 표현 방법 #1

- 모든 차수에 대한 계수값을 배열로 저장
- 하나의 다항식을 하나의 배열로 표현

$$10x^5 + 0x^4 + 0x^3 + 0x^2 + 6x^1 + 3x^0$$




```
typedef struct {
	int degree;
	float coef[MAX_DEGREE];
} polynomial;
polynomial a = { 5, {10, 0, 0, 0, 6, 3} };
```

장점: 다항식의 각종 연산이 간단해짐
단점: 대부분의 항의 계수가 0이면 공간의 낭비가 심함

다항식 덧셈 연산 #1 (1)

```
#include <stdio.h>
#define MAX(a,b) (((a)>(b))?(a):(b))
#define MAX_DEGREE 101
typedef struct {                    // 다항식 구조체 타입 선언
    int degree;                     // 다항식의 차수
    float coef[MAX_DEGREE];         // 다항식의 계수
} polynomial;
```

다항식 덧셈 연산 #1 (2)

```
// C = A+B 여기서 A와 B는 다항식이다.
polynomial poly_add1(polynomial A, polynomial B)
{
    polynomial C;                    // 결과 다항식
    int Apos=0, Bpos=0, Cpos=0;     // 배열 인덱스 변수
    int degree_a=A.degree;
    int degree_b=B.degree;
    C.degree = MAX(A.degree, B.degree); // 결과 다항식 차수
    while( Apos<=A.degree && Bpos<=B.degree ){
        if( degree_a > degree_b ){ // A항 > B항
          C.coef[Cpos++]= A.coef[Apos++];
          degree_a--;
        }
```

다항식 덧셈 연산 #1 (3)

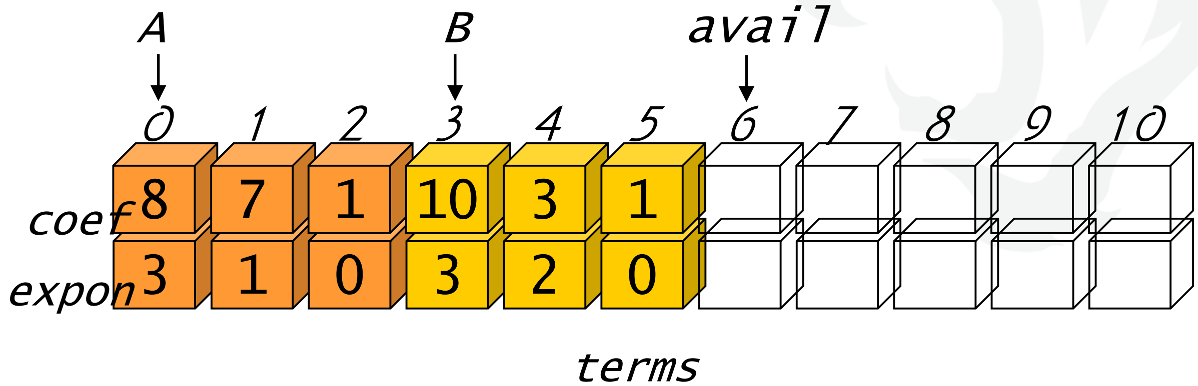
```
        else if( degree_a == degree_b ){ // A항 == B항
                C.coef[Cpos++]=A.coef[Apos++]+B.coef[Bpos++];
                degree_a--; degree_b--;
            }
        else {                      // B항 > A항
                C.coef[Cpos++]= B.coef[Bpos++];
                degree_b--;
            }
        }
        return C;
}
// 주함수
main()
{
        polynomial a = { 5, {3, 6, 0, 0, 0, 10} };
        polynomial b = { 4, {7, 0, 5, 0, 1} };
        polynomial c;
        c = poly_add1(a, b);
}
```

다항식 표현 방법 #2

- 다항식에서 0이 아닌 항만을 배열에 저장
- (계수, 차수) 형식으로 배열에 저장
  - 예: $10x^5+6x+3$ -> ((10,5), (6,1), (3,0))

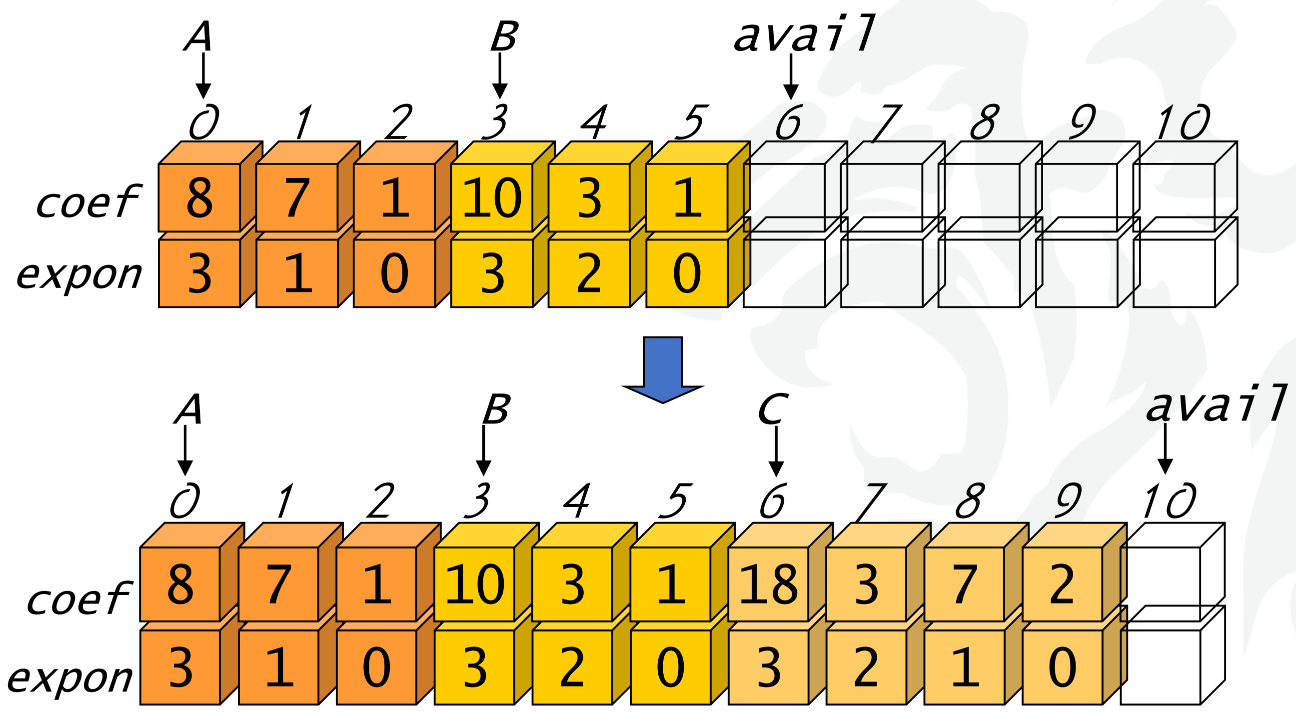
```
struct {
        float coef;
        int expon;
} terms[MAX_TERMS]={ {10,5}, {6,1}, {3,0} };
```

- 하나의 배열로 여러 개의 다항식을 나타낼 수 있음



다항식 표현 방법 #2 (계속)

- 장점: 메모리 공간의 효율적인 이용
- 단점: 다항식의 연산들이 복잡해짐
  - 예: 다항식의 덧셈 $A=8x^3+7x+1$, $B=10x^3+3x^2+1$, $C=A+B$



다항식 덧셈 연산 #2 (1)

```
#define MAX_TERMS 101
struct {
    float coef;
    int expon;
} terms[MAX_TERMS]={ {8,3}, {7,1}, {1,0}, {10,3}, {3,2},{1,0} };
int avail=6;
// 두 개의 정수를 비교
char compare(int a, int b)
{
    if( a>b ) return '>';
    else if( a==b ) return '=';
    else return '<';
}
```

다항식 덧셈 연산 #2 (2)

```
// 새로운 항을 다항식에 추가한다.
void attach(float coef, int expon)
{
    if( avail>MAX_TERMS ){
            fprintf(stderr, "항의 개수가 너무 많음\n");
            exit(1);
    }
    terms[avail].coef=coef;
    terms[avail++].expon=expon;
}
```

다항식 덧셈 연산 #2 (3)

```
// C = A + B
poly_add2(int As, int Ae, int Bs, int Be, int *Cs, int *Ce)
{
        float tempcoef;
        *Cs = avail;
        while( As <= Ae && Bs <= Be )
        switch(compare(terms[As].expon,terms[Bs].expon)){
        case '>':        // A의 차수 > B의 차수
                attach(terms[As].coef, terms[As].expon);
                As++;                   break;
        case '=':        // A의 차수 == B의 차수
                tempcoef = terms[As].coef + terms[Bs].coef;
                if( tempcoef )
                 attach(tempcoef,terms[As].expon);
                As++; Bs++;             break;
        case '<':        // A의 차수 < B의 차수
                attach(terms[Bs].coef, terms[Bs].expon);
                Bs++;                   break;
        }
```

다항식 덧셈 연산 #2 (4)

```
// A의 나머지 항들을 이동함
    for(;As<=Ae;As++)
        attach(terms[As].coef, terms[As].expon);
    // B의 나머지 항들을 이동함
    for(;Bs<=Be;Bs++)
        attach(terms[Bs].coef, terms[Bs].expon);
    *Ce = avail -1;
}
//
void main()
{
    int Cs, Ce;
    poly_add2(0,2,3,5,&Cs,&Ce);
}
```

희소 행렬

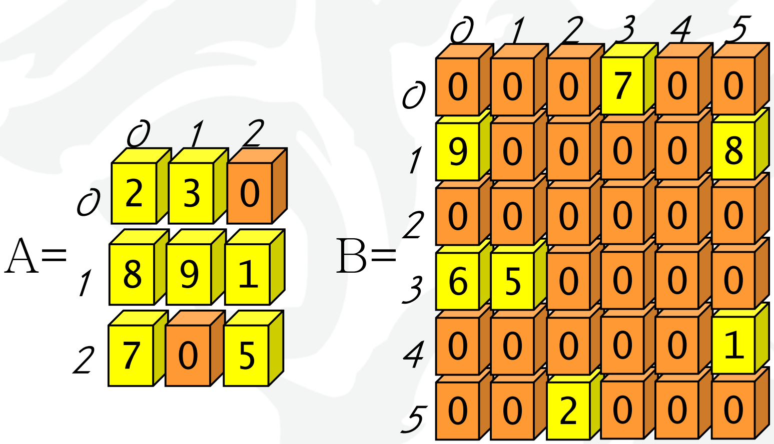
- 배열을 이용하여 행렬(matrix)을 표현하는 2가지 방법
  - (1) 2차원 배열을 이용하여 배열의 전체 요소를 저장하는 방법
  - (2) 0이 아닌 요소들만 저장하는 방법
- 희소행렬: 대부분의 항들이 0인 배열

$$A = \begin{bmatrix} 2 & 3 & 0 \\ 8 & 9 & 1 \\ 7 & 0 & 5 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 0 & 0 & 7 & 0 & 0 \\ 9 & 0 & 0 & 0 & 0 & 8 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 6 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$

희소 행렬 표현 방법 #1

- 2차원 배열을 이용하여 배열의 전체 요소를 저장하는 방법
  - 장점: 행렬의 연산들을 간단하게 구현할 수 있음
  - 단점: 대부분의 항들이 0인 희소 행렬의 경우 많은 메모리 공간 낭비

$$A = \begin{bmatrix} 2 & 3 & 0 \\ 8 & 9 & 1 \\ 7 & 0 & 5 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 0 & 0 & 7 & 0 & 0 \\ 9 & 0 & 0 & 0 & 0 & 8 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 6 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$



희소 행렬 덧셈 연산 #1 (1)

```
#include <stdio.h>
#define ROWS 3
#define COLS 3
// 희소 행렬 덧셈 함수
void sparse_matrix_add1(int A[ROWS][COLS],
        int B[ROWS][COLS], int C[ROWS][COLS]) // C=A+B
{
    int r,c;
    for(r=0;r<ROWS;r++)
        for(c=0;c<COLS;c++)
            C[r][c] = A[r][c] + B[r][c];
}
```

희소 행렬 덧셈 연산 #1 (2)

```
main()
{
    int array1[ROWS][COLS] = {    { 2,3,0 },
                                  { 8,9,1 },
                                  { 7,0,5 } };
    int array2[ROWS][COLS] = {    { 1,0,0 },
                                  { 1,0,0 },
                                  { 1,0,0 } };
    int array3[ROWS][COLS];
    sparse_matrix_add1(array1,array2,array3);
}
```

희소 행렬 표현 방법 #2

- 0이 아닌 요소들만 저장하는 방법
  - 장점: 희소 행렬의 경우, 메모리 공간의 절약
  - 단점: 각종 행렬 연산들의 구현이 복잡해짐

$$A = \begin{bmatrix} 2 & 3 & 0 \\ 8 & 9 & 1 \\ 7 & 0 & 5 \end{bmatrix} \quad B = \begin{bmatrix} 0 & 0 & 0 & 7 & 0 & 0 \\ 9 & 0 & 0 & 0 & 0 & 8 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 6 & 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$

A=

	행	열	값
0	0	0	2
1	0	1	3
2	1	0	8
3	1	1	9
4	1	2	1
5	2	0	7
6	2	2	5

B=

	행	열	값
0	0	3	7
1	1	0	9
2	1	5	8
3	3	0	6
4	3	1	5
5	4	5	1
6	5	2	2

희소 행렬 덧셈 연산 #2 (1)

```
#define ROWS 3
#define COLS 3
#define MAX_TERMS 10
typedef struct {
    int row;
    int col;
    int value;
} element;
typedef struct SparseMatrix {
    element data[MAX_TERMS];
    int rows;   // 행의 개수
    int cols;   // 열의 개수
    int terms;  // 항의 개수
} SparseMatrix;
```

희소 행렬 덧셈 연산 #2 (2)

```
// 희소 행렬 덧셈 함수
// c = a + b
SparseMatrix sparse_matrix_add2(SparseMatrix a, SparseMatrix b)
{
    SparseMatrix c;
    int ca=0, cb=0, cc=0; // 각 배열의 항목을 가리키는 인덱스
    // 배열 a와 배열 b의 크기가 같은지를 확인
    if( a.rows != b.rows || a.cols != b.cols ){
        fprintf(stderr,"희소행렬 크기에러\n");
        exit(1);
    }
    c.rows = a.rows;
    c.cols = a.cols;
    c.terms = 0;
```

희소 행렬 덧셈 연산 #2 (3)

```
while( ca < a.terms && cb < b.terms ){
        // 각 항목의 순차적인 번호를 계산한다.
                        int inda = a.data[ca].row * a.cols + a.data[ca].col;
                        int indb = b.data[cb].row * b.cols + b.data[cb].col;
        if( inda < indb) {
             // a 배열 항목이 앞에 있으면
             c.data[cc++] = a.data[ca++];
        }
        else if( inda == indb ){
             // a와 b가 같은 위치
             if( (a.data[ca].value+b.data[cb].value)!=0){
               c.data[cc].row = a.data[ca].row;
               c.data[cc].col = a.data[ca].col;
               c.data[cc++].value = a.data[ca++].value +
                          b.data[cb++].value;
             }
             else {
               ca++; cb++;
             }
        }
        else   // b 배열 항목이 앞에 있음
             c.data[cc++] = b.data[cb++];
     }
```

희소 행렬 덧셈 연산 #2 (4)

```
// 배열 a와 b에 남아 있는 항들을 배열 c로 옮긴다.
    for(; ca < a.terms; )
            c.data[cc++] = a.data[ca++];
    for(; cb < b.terms; )
            c.data[cc++] = b.data[cb++];
    c.terms = cc;
    return c;
}
// 주함수
main()
{
    SparseMatrix m1 = {     {{ 1,1,5 },{ 2,2,9 }}, 3,3,2 };
    SparseMatrix m2 = {     {{ 0,0,5 },{ 2,2,9 }}, 3,3,2 };
    SparseMatrix m3;
    m3 = sparse_matrix_add2(m1, m2);
}
```

Week 2: Array