Array Implementaion: Polynomials & Sparse Matrix

Amiya Ranjan Panda

Polynomials

A single variable polynomial p(x) = $4x^6 + 10x^4 - 5x + 3$

Remark: order of this polynomial is 6 (highest exponent)

- Representing Polynomials
  - In general, the polynomial are represented as:

$$A(x) = a_{m-1}x^{e_{m-1}} + \cdots + a_0 x^{e_0}$$

    - where the a_i are nonzero coefficients and the e_i are nonnegative integer exponents such that

$$e_{m-1} > e_{m-2} > \ldots > e_1 > e_0 \geqq 0 .$$

Polynomial

How to implement this?

There are different ways of implementing the polynomial ADT:

- Array (not recommended)
- Double Array (inefficient)
- Array of Structure (inefficient)
- Linked List (preferred and recommended)

Polynomial

Array Implementation:

$p_1(x) = 8x^3 + 3x^2 + 2x + 6$

$p_2(x) = 23x^4 + 18x - 3$

$p_1(x)$

0	1	2	3
6	2	3	8

$p_2(x)$

0	1	2	3	4
-3	18	0	0	23

0 2 0 2 4

Index represents exponents

Polynomial

This is why arrays are not good to represent polynomials:

$p_3(x) = 16x^{21} - 3x^5 + 2x + 6$

0	1	2	3	4	5	. . .	20	21
6	2	0	0	0	-3	. . .	0	16

WASTE OF SPACE!

Polynomial

- Advantages of using an Array
  - good for non-sparse polynomials.
  - easy to store and retrieve.

- Disadvantages of using an Array:
  - Allocate array size ahead of time.
  - huge array size required for sparse polynomials. Waste of space and runtime.

Polynomial Addition

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

float a[50], b[50], c[50], d[50];
int main() {
  int i;
  int deg1, deg2;
  int k=0, l=0, m=0;
  printf("Enter the highest degree of polynomial1: ");
  scanf("%d", &deg1);
  for(i=0; i<=deg1; i++)  {
     printf("\nEnter the coeff of x^%d :", i);
     scanf("%f", &a[i]);
 }
```

Polynomial Addition

```
printf("\nEnter the highest degree of polynomial2: ");
 scanf("%d",  &deg2);
 for(i=0; i<=deg2; i++)  {
     printf("\nEnter the coeff of x^%d : ", i);
     scanf("%f", &b[i]);
 }
 printf("\nPolynomial 1 = %.1f", a[0]);
 for(i=1; i<=deg1; i++)
      printf("+ %.1fx^%d", a[i], i);
 printf("\nPolynomial 2 = %.1f", b[0]);
 for(i=1; i<=deg2; i++)
      printf("+ %.1fx^%d",b[i], i);
```

Polynomial Addition

```
if(deg1>deg2) {
  for(i=0; i<=deg2; i++) {
     c[m] = a[i] + b[i];
     m++;
  }
  for(i=deg2+1; i<=deg1; i++) {
     c[m] = a[i];
     m++;
  }
}
```

```
else  {
  for(i=0; i<=deg1; i++)  {
     c[m] = a[i] + b[i];
     m++;
  }
  for(i=deg1+1; i<=deg2; i++)  {
     c[m] = b[i];
     m++;
  }
}
```

Polynomial Addition

```
  printf("\npolynomial after addition  = %.1f", c[0]);
  for(i=1; i<m; i++)
     printf("+ %.1fx^%d", c[i], i);
   return 0;
}
```

Output

Enter the highest degree of polynomial1: 3
Enter the coeff of x^0 :2
Enter the coeff of x^1 :3
Enter the coeff of x^2 :5
Enter the coeff of x^3 :1
Enter the highest degree of polynomial2: 2
Enter the coeff of x^0 :7
Enter the coeff of x^1 :8
Enter the coeff of x^2 :5
polynomial 1 = 2.0+ 3.0x^1+ 5.0x^2+ 1.0x^3
polynomial 2 = 7.0+ 8.0x^1+ 5.0x^2
polynomial after addition = 9.0+ 11.0x^1+ 10.0x^2+ 1.0x^3

Polynomial

Double Array Implementation:

Represent the following two polynomials:

$p_1(x) = 23x^9 + 18x^7 - 41x^6 + 163x^4 - 5x + 3$
$p_2(x) = 4x^6 + 10x^4 + 12x + 8$

Polynomial

$p_1(x) = 23x^9 + 18x^7 - 41x^6 + 163x^4 - 5x + 3$

$p_2(x) = 4x^6 + 10x^4 + 12x + 8$

Start $p_1(x)$

Start $p_2(x)$

Coefficient

23	18	-41	163	-5	3	4	10	12	8
9	7	6	4	1	0	6	4	1	0
0		2						8	

Exponent

End $p_1(x)$

End $p_2(x)$

Polynomial

Advantages of using two array:

- save space (compact)

Disadvantages of using two Array:

- difficult to maintain
- have to allocate array size ahead of time
- more code required for misc. operations.

Polynomial using structure

Structure Implementation:

```
struct poly  {
    float coeff;
    int exp;   };
struct poly p[50];
```

Polynomial Addition

```
#include<stdio.h>
#include<math.h>
struct poly  {
    float coeff;
    int exp;   };
struct poly a[50], b[50], c[50], d[50];
int main() {
   int nterm1, nterm2, nterm3;
   int i, k=0, l=0, m=0;
   printf("Enter the number of non-zero terms in Polynomial1: ");
   scanf("%d", &nterm1);
   for(i=0; i<nterm1; i++)  {
       printf("\nEnter the coeff of %d th term: ", i);
       scanf("%f", &a[i].coeff);
       printf("\nEnter the exp of %d th term: ", i);
       scanf("%f", &a[i].exp);
   }
```

Polynomial Addition

```
printf("\nEnter the number of non-zero terms in Polynomial2: ");
 scanf("%d",  &nterm2);
 for(i=0; i<nterm2; i++)  {
    printf("\nEnter the coeff of %d th term: ", i);
    scanf("%f", &b[i].coeff);
    printf("\nEnter the exp of %d th term: ", i);
    scanf("%f", &b[i].exp);
 }
 printf("\nPolynomial 1 = %.1f", a[0].coeff);
 for(i=1; i<nterm1; i++)
     printf("+ %.1fx^%d", a[i].coeff, a[i].exp);
 printf("\nPolynomial 2 = %.1f", b[0].coeff);
 for(i=1; i<nterm2; i++)
     printf("+ %.1fx^%d", b[i].coeff, b[i].exp);
```

Polynomial Addition

```
while(k<nterm1 && l<nterm2)  {
    if(a[k].exp < b[l].exp) {
        c[m].coeff = a[k].coeff;
        c[m].exp = a[k].exp;
        k++; m++;        }
     else if(a[k].exp > b[l].exp) {
        c[m].coeff = b[l].coeff;
        c[m].exp = b[l].exp;
        l++; m++;        }
      else {
        c[m].coeff = a[k].coeff + b[l].coeff;
        c[m].exp = a[k].exp;
        k++; l++; m++;   }
 }
```

Polynomial Addition

```
while(k<nterm1)  {
    c[m].coeff = a[k].coeff;
    c[m].exp = a[k].exp;
    k++; m++;
}
 while(l<nterm2)  {
    c[m].coeff = b[l].coeff;
    c[m].exp = b[l].exp;
    l++; m++;
 }
 nterm3 = m-1;
 printf("\npolynomial after addition  = %.1f", c[0].coeff);
 for(i=1; i<nterm3; i++)
     printf("+ %.1fx^%d", c[i].coeff, c[i].exp);
 return 0;
}
```

Polynomial Multiplication

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

float a[50], b[50], c[50], d[50];
int main() {
   int i;
   int deg1,deg2;
   int k=0,l=0,m=0;
   printf("Enter the highest degree of polynomial1: ");
   scanf("%d", &deg1);
   for(i=0; i<=deg1; i++)  {
       printf("\nEnter the coeff of x^%d :", i);
       scanf("%f", &a[i]);
  }
```

Polynomial Multiplication

```
printf("\nEnter the highest degree of polynomial2: ");
 scanf("%d",  &deg2);
 for(i=0; i<=deg2; i++)  {
     printf("\nEnter the coeff of x^%d : ", i);
     scanf("%f", &b[i]);
 }
 printf("\nPolynomial 1 = %.1f", a[0]);
 for(i=1; i<=deg1; i++)
      printf("+ %.1fx^%d", a[i], i);
 printf("\nPolynomial 2 = %.1f", b[0]);
 for(i=1; i<=deg2; i++)
      printf("+ %.1fx^%d", b[i], i);
```

Polynomial Multiplication

```
        deg3 = deg1+deg2;
        for (int i = 0; i<=deg3; i++)
            c[i] = 0;
        for (int i=0; i<=deg1; i++) {
             for (int j=0; j<=deg2; j++)
                 c[i+j] += a[i] * b[j];
        }
        printf("\nPolynomial after multiplication = %.1f", c[0]);
        for(i=1; i<=deg3; i++)
           printf("+ %.1fx^%d", c[i], i);
        return 0;
}
```

Polynomial Multiplication

Output

Enter the highest degree of polynomial1:2
Enter the coeff of x^0 :2
Enter the coeff of x^1 :3
Enter the coeff of x^2 :4
Enter the highest degree of polynomial2:3
Enter the coeff of x^0 :5
Enter the coeff of x^1 :6
Enter the coeff of x^2 :7
Enter the coeff of x^3 :2
Polynomial 1 = 2.0+ 3.0x^1+ 4.0x^2
Polynomial 2 = 5.0+ 6.0x^1+ 7.0x^2+ 2.0x^3
Polynomial after multiplication = 10.0+ 27.0x^1+ 52.0x^2+ 49.0x^3+ 34.0x^4+ 8.0x^5

Polynomial Multiplication

```
#include<stdio.h>
#include<math.h>
struct poly  {
    float coeff;
    int exp;   };
struct poly a[50], b[50], c[50], d[50];
int main() {
   int nterm1, nterm2, nterm3;
   int i, j, k, l=0, m=0;
   float prod;
   printf("Enter the number of non-zero terms in Polynomial1: ");
   scanf("%d", &nterm1);
   for(i=0; i<nterm1; i++)  {
       printf("\nEnter the coeff of %d th term: ", i);
       scanf("%f", &a[i].coeff);
       printf("\nEnter the exp of %d th term: ", i);
       scanf("%f", &a[i].exp);       }
```

Polynomial Multiplication

```
printf("\nEnter the number of non-zero terms in Polynomial2: ");
scanf("%d",  &nterm2);
for(i=0; i<nterm2; i++)  {
    printf("\nEnter the coeff of %d th term: ", i);
    scanf("%f", &b[i].coeff);
    printf("\nEnter the exp of %d th term: ", i);
    scanf("%f", &b[i].exp);
 }
 printf("\nPolynomial 1 = %.1f", a[0].coeff);
 for(i=1; i<nterm1; i++)
     printf("+ %.1fx^%d", a[i].coeff, a[i].exp);
 printf("\nPolynomial 2 = %.1f", b[0].coeff);
 for(i=1; i<nterm2; i++)
     printf("+ %.1fx^%d", b[i].coeff, b[i].exp);
```

Polynomial Multiplication

```
for (int i=0; i<nterm1; i++) {
    for (int j=0; j<ntern2; j++) {
        prod = a[i].coeff*b[j].coeff;
        for (int k=0; k<m; k++) {
            if(a[i].exp+b[j].exp == c[k].exp)  {
                c[k].coeff += prod;
                break;  }
        }
        c[m].exp=a[i].exp+b[j].exp;
        c[m++].coeff = prod;
    }
}
nterm3 = m-1;
printf("\nPolynomial after multiplication = %.1f", c[0].coeff);
for(i=1; i<nterm3; i++)
    printf("+ %.1fx^%d", c[i].coeff, c[i].exp);
return 0;  }
```

Sparse Matrix

Sparse Matrix

- A matrix is a two-dimensional data object made of *m* rows and *n* columns having total m × n values.
- If most of the elements of the matrix have 0 value, then it is called a sparse matrix.

- Why to use Sparse Matrix instead of simple matrix?
  - Storage: less memory used to store only those non-zero elements.
  - Computing time: Computing time can be reduced by logically designing a data structure traversing only non-zero elements.

Sparse Matrix

$$
\begin{array}{c c}
 & \begin{array}{ccc} \text{col 1} & \text{col 2} & \text{col 3} \end{array} \\
\begin{array}{c} \text{row 1} \\ \text{row 2} \\ \text{row 3} \\ \text{row 4} \\ \text{row 5} \end{array} &
\begin{bmatrix} -27 & 3 & 4 \\ 6 & 82 & -2 \\ 109 & -64 & 11 \\ 12 & 8 & 9 \\ 48 & 27 & 47 \end{bmatrix}
\end{array}
\quad 5 \times 3
$$

$$
\begin{array}{c c}
 & \begin{array}{cccccc} \text{col0} & \text{col1} & \text{col2} & \text{col3} & \text{col4} & \text{col5} \end{array} \\
\begin{array}{c} \text{row0} \\ \text{row1} \\ \text{row2} \\ \text{row3} \\ \text{row4} \\ \text{row5} \end{array} &
\begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}
\end{array}
\quad 6 \times 6
$$

(a) 15/15

(b) 8/36

Two matrices

sparse matrix

data structure?

Sparse Matrix: Array Representation

- Represented by a two-dimensional array.
- Sparse matrix wastes space
- Each element is characterized by <row, col, value>.

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

# of rows (columns)

# of nonzero terms

row, column in ascending order

Sparse Matrix: Program

```
#include<stdio.h>
int main() {
    // Assume 4x5 sparse matrix
    int smat[4][5] =
    {  {0 , 0 , 3 , 0 , 4 },
       {0 , 0 , 5 , 7 , 0 },
       {0 , 0 , 0 , 0 , 0 },
       {0 , 2 , 6 , 0 , 0 }
    };
    int i, j, k, size = 0;
    for (i = 0; i < 4; i++)
        for (j = 0; j < 5; j++)
            if (smat[i][j] != 0)
                size++;
    int sm[size+1][3];
```

```
k = 0;
sm[k][0] = 4; sm[k][1] = 5; sm[k][2] = size;
k++;
for (i = 0; i < 4; i++)
    for (j = 0; j < 5; j++)
        if (smat[i][j] != 0) {
            sm[k][0] = i;  sm[k][1] = j;
            sm[k][2] = smat[i][j];   k++;
        }
for (int i=0; i<=size; i++) {
    for (int j=0; j<3; j++)
        printf("%d ", sm[i][j]);
    printf("\n");
}
return 0;  }
```

Output:

```
4 5 6
0 2 3
0 4 4
1 2 5
1 3 7
3 1 2
3 2 6
```

Sparse Matrix: Transpose

of rows (columns)

# of nonzero 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)

transpose

	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)

row, column in ascending order

Sparse matrix and its transpose stored as triples

Sparse Matrix: Representation

Sparse_matrix Create(max_row, max_col) :

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

# of rows (columns)
# of nonzero terms

Transpose a Matrix

(1) for each row i

element <i, j, value> store

in element <j, i, value> of the transpose

Difficulty: where to put <j, i, value>

(0, 0, 15) → (0, 0, 15)

(0, 3, 22) → (3, 0, 22)

(0, 5, -15) → (5, 0, -15)

(1, 1, 11) → (1, 1, 11)

Move elements down very often.

Approach:

(2) For all elements in column j,

place element <i, j, value> in element <j, i, value>

Sparse Matrix: Transpose

```
void transpose (Sparse a[], Sparse b[])  {
    int n, i, j, k;
    n = a[0].value;
    b[0].row = a[0].col;
    b[0].col = a[0].row;
    b[0].value = n;
    if (n > 0) {      /*non zero matrix */
        k = 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 */
```

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

Sparse Matrix: Transpose

columns elements

```
      b[k].row = a[j].col;
      b[k].col  = a[j].row;
      b[k].value = a[j].value;
      k++
    }
  }
}
```

Transpose of a sparse matrix

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

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

Scan the array "columns" times.
The array has "*elements*" elements.

==> O(columns×elements)

Sparse Matrix: Transpose

Discussion: compared with 2-D array representation

- O(columns×elements) vs. O(columns×rows)

- elements → columns × rows when nonsparse
  - O(columns×columns×rows)

Inefficient: Scan the array "*columns*" times.

Solution:

- Determine the # of elements in each column of the original matrix.
- Determine what will be the the starting positions of each row in the transpose matrix.

Sparse Matrix

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

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

	[0]	[1]	[2]	[3]	[4]	[5]
row_terms =	2	1	2	2	0	1
starting_pos =	1	3	4	6	8	8

Sparse Matrix: Transpose

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

```
void fast_transpose(term a[ ], term 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*/
```

columns

```
    for (i = 0; i < num_cols; i++)
        row_terms[i] = 0;
```

elements

```
    for (i = 1; i  <= num_terms; i++)
        row_terms[a[i].col]++
    starting_pos[0] = 1;
```

columns

```
    for (i =1; i < num_cols; i++)
        starting_pos[i]=starting_pos[i-1] + row_terms[i-1];
```

	[0]	[1]	[2]	[3]	[4]	[5]
row_terms =	2	1	2	2	0	1
starting_pos =	1	3	4	6	8	8

Sparse Matrix: Transpose

```
    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;
    }
  }
}
```

elements

Fast transpose of a sparse matrix

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

	[0]	[1]	[2]	[3]	[4]	[5]
row_terms =	2	1	2	2	0	1
starting_pos =	1	3	4	6	8	8

Compared with 2-D array representation: O(columns+elements) vs. O(columns×rows)

elements → columns×rows: O(columns+elements) → O(columns×rows)

Cost: Additional *row_terms* and *starting_pos* arrays are required.

Let the two arrays row_terms and starting_pos be shared.

Sparse Matrix Addition

```
void read_sp_mat(int sp[][3])  {
    int r, c, i, j, n;
    printf("\nEnter r and c : ");
    scanf("%d %d", &r, &c);
    printf("\nEnter # of nonzero elements : ");
    scanf("%d", &n);
    printf("\nEnter the elements \n");
    j=1;
    for(i=1; i <= n; i++)  {
        printf("\nEnter row no : ");
        scanf("%d", &sp[j][0]);
        printf("\nEnter col no : ");
        scanf("%d", &sp[j][1]);
        printf("\nEnter the value : ");
        scanf("%d", &sp[j][2]);
        j++;
    }
        sp[0][0] = r;
        sp[0][1] = c;
        sp[0][2] = n;
}
```

Matrix 1: (4×4)

Row	Col	Value
4	4	5
1	2	10
1	4	12
3	3	5
4	1	15
4	2	12

Matrix 2: (4×4)

Row	Col	Value
4	4	5
1	3	8
2	4	23
3	3	9
4	1	20
4	2	25

Sparse Matrix Addition

```
int add_sp_mat(int sp1[][3], sp2[][3], sp3[][3])  {
   int r, c, i, j, k1, k2, k3, tot1 ,tot2;
   if( sp1[0][0] != sp2[0][0] || sp1[0][1] != sp2[0][1] )  {
      printf("Invalid matrix size ");
      exit(0);
   }
   tot1 = sp1[0][2];  tot2 = sp2[0][2];
   k1 = k2 = k3 = 1;
   while ( k1 <= tot1 && k2 <= tot2)  {
      if ( sp1[k1][0] < sp2[k2][0] )  {
         sp3[k3][0] = sp1[k1][0];
         sp3[k3][1] = sp1[k1][1];
         sp3[k3][2] = sp1[k1][2];
         k3++; k1++;
      }
      else
      if ( sp1[k1][0] > sp2[k2][0] )  {
         sp3[k3][0] = sp2[k2][0];
         sp3[k3][1] = sp2[k2][1];
         sp3[k3][2] = sp2[k2][2];
         k3++;k2++;
      }
      else
      if(sp1[k1][1] < sp2[k2][1])  {
         sp3[k3][0]=sp1[k1][0];
         sp3[k3][1]=sp1[k1][1];
         sp3[k3][2]=sp1[k1][2];
         k1++;    k3++;
      }
```

Sparse Matrix Addition

```
 else
 if(sp1[k1][1] > sp2[k2][1])  {
     sp3[k3][0]=sp2[k2][0];
     sp3[k3][1]=sp2[k2][1];
     sp3[k3][2]=sp2[k2][2];
     k2++;
     k3++;
 }
else  //if ( sp1[k1][0] == sp2[k2][0] )
{
     sp3[k3][0] = sp2[k2][0];
     sp3[k3][1] = sp2[k2][1];
     sp3[k3][2] = sp1[k1][2] + sp2[k2][2];
     k3++;k2++;k1++;
 }
}
```

```
 while ( k1 <=tot1 )  {
    sp3[k3][0] = sp1[k1][0];
    sp3[k3][1] = sp1[k1][1];
    sp3[k3][2] = sp1[k1][2];
    k3++;k1++;
 }
 while ( k2 <= tot2 )  {
    sp3[k3][0] = sp2[k2][0];
    sp3[k3][1] = sp2[k2][1];
    sp3[k3][2] = sp2[k2][2];
    k3++;k2++;
 }
 sp3[0][0] = sp1[0][0];
 sp3[0][1] = sp1[0][1];
 sp3[0][2] = k3-1;
}
```

Sparse Matrix Addition: Example

Matrix 1: (4×4)

Row	Col	Value
4	4	5
1	2	10
1	4	12
3	3	5
4	1	15
4	2	12

Matrix 2: (4×4)

Row	Col	Value
4	4	5
1	3	8
2	4	23
3	3	9
4	1	20
4	2	25

Result of Addition: (4×4)

Row	Col	Value
4	4	7
1	2	10
1	3	8
1	4	12
2	4	23
3	3	14
4	1	35
4	2	37