



순천향대학교  
SOON CHUN HYANG  
UNIVERSITY

# Mid Term Project

## 자료구조1

담당교수	홍민 교수님
학과	컴퓨터소프트웨어공학과
학번	20204005
이름	김필중
제출일	2021년 04월 23일

# | 목 차 |

## 1. 희소행렬의 전치, 연산 프로그램

1.1 문제 분석

1.2 소스 코드

1.3 소스코드 분석

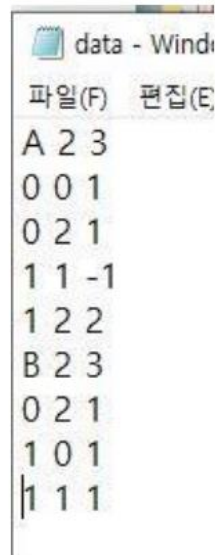
1.4 실행결과

1.5 느낀점

## 1. 문제분석

- 두개의 희소 행렬 데이터를 **data.txt**파일에서 입력 받아, 두 행렬의 **+**, **-**, **\*** 연산과 전치행렬을 구하는 프로그램을 작성하시오. 단, 동적 할당으로 2차원 배열을 생성하여 작성하여야 함. 제출일: 4월 23일(금)

- data.txt** 파일 예



```
data - Windi
파일(F)  편집(E)
A 2 3
0 0 1
0 2 1
1 1 -1
1 2 2
B 2 3
0 2 1
1 0 1
1 1 1
```

- 결과 예:

A+B

0 0 : 1

0 2 : 2

1 0 : 1

1 2 : 2

문제를 보고 풀 매트릭스로 바꿔 연산하지 말아야 한다는 조건도 걸려있었다. 그래서 처음부터 희소행렬을 받아서 이중 동적 할당에 저장을 해 연산과정을 거쳐야 한다고 분석했다. 특히 희소행렬 자체로 주어진 경우는 일반 행렬과는 다르게 라인 수를 세서 그만큼의 동적할당을 줘야 하는 부분이 있다. 그로 인해 우선적으로 라인을 세는 구문부터 해결해야 한다고 판단했다. Fgets 구문을 이용해 하는 것이 합리적이라고 분석했다.

희소 행렬의 연산 같은 경우는 일반 행렬과는 다르게 덧셈이나 뺄셈은 같은 행과 열을 가진 A B 행렬을 연산하고 만약 A행렬에는 존재하지만 B행렬엔 존재하지 않는 행과 열을 가지고 있으면 B행렬은 0이라 생각하고 계산하는 부분을 고려해야했다. 이것 또한 동적 메모리가 낭비되지 않게 필요한 개수를 찾고 그만큼 동적할당을 해야한다 분석을 했다.

전치 행렬은 간단하게 행과 열을 바꿔주면 되는 것이라 간단히 해결이 가능하다고 분석했다.

행렬의 곱은 마찬가지로 필요한 동적 메모리를 이용하기 위해서 개수를 세서 그만큼의 동적할당을 해줘야 했다. 희소행렬의 곱은 전치행렬을 이용해 오름차순으로 정렬해 계산하는 것이 계산과정이 줄어들 수 있다 판단했다.

## 1.2 소스코드

```
1  /*
2     작성일 : 2021년 4월 16일 ~ 4월 22일
3     작성자 : 김필중
4     프로그램명 : 중간 과제 희소행렬 데이터를 입력받아 연산과 전치행렬을 구하는 프로그램
5  */
6
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <string.h>
10 #pragma warning(disable : 4996) // 경고 제거 구문
11
12 void sparse_matrix_setting(); // 기본 세팅 함수
13 void sparse_matrix_insert(FILE *fp, int num, int **matrix); // 동적할당을 하고 동적으로 만든 배열에 행렬의 값을 집어넣는 함수
14 void sparse_matrix_add(int row_A, int col_A, int num_A, int **matrix_A, int row_B, int col_B, int num_B, int **matrix_B); // 희소행렬의 덧셈 함수
15 void sparse_matrix_sub(int row_A, int col_A, int num_A, int **matrix_A, int row_B, int col_B, int num_B, int **matrix_B); // 희소행렬의 뺄셈 함수
16 void sparse_matrix_show(int num, int **matrix); // 행렬을 보여주는 함수
17 void sparse_matrix_transpose(int num, int **matrix); // 행렬을 전치하는 전치행렬 함수
18 void sparse_matrix_mul(int col_A, int num_A, int **matrix_A, int row_B, int num_B, int **matrix_B); // 행렬의 곱 함수
19 void sparse_matrix_sort(int num, int **matrix); // 행렬을 정렬해주는 함수
20 int sparse_matrix_mul_check(int **matrix_A, int num_A, int **matrix_B, int num_B); // 행렬의 곱의 개수를 확인해주는 함수
21
22 int main(void)
23 {
24     sparse_matrix_setting(); // 세팅 함수를 호출한다.
25     return 0;
26 }
27
28 void sparse_matrix_setting()
29 {
30     // -----
31     // 필요한 변수들을 선언한다.
32     // -----
33     FILE *fp; // 파일포인터 함수
34     int A_row, A_col, num_A = 0; // A배열에 저장할 변수
35     int B_row, B_col, num_B = 0; // B배열에 저장할 변수
36     int set = 0, end = 0, cur = 0; //파일을 읽으면서 위치를 저장하는 변수
37     int **matrix_A; // A 행렬을 저장하기 위해 이중포인터로 변수 선언
38     int **matrix_B; // B 행렬을 저장하기 위해 이중포인터로 변수 선언
39     int i;
40     char A_name, B_name;
41     char check[100]; //fgets의 한줄을 읽기 위한 변수
42     char B_Check; // B가 들어오는지 확인하기 위한 변수
43
44     // -----
45     // 파일 포인터 관련 부분
46     // -----
47     fp = fopen("data.txt", "r");
48     if(fp==NULL) // 만약 fp가 null 일 경우 파일 오픈이 실패를 확인하는 if문
49     {
50         printf("파일에 열리지 않았습니다.\n");
51         exit(1); // 프로그램 종료
52     }
53
54     fscanf(fp, "%c %d %d ", &A_name, &A_row, &A_col); // 파일의 첫번째에 들어있는 값을 읽어 저장한다.
55     while(1) // A의 희소행렬의 개수를 확인하기 위한 반복문
56     {
57         fgets(check, sizeof(check), fp); // 한 문장을 읽는다
58         B_Check = check[0]; // 처음 값을 읽는다
59
60         if(B_Check == 'B' || B_Check == 'b') // 처음값이 B일경우 즉시 종료한다.
61         {
62             break;
63         }
64         else
65         {
66             num_A++; // 아닐경우 A의 라인을 증가 시킨다.
67             set = ftell(fp); // 파일의 위치를 체크해준다.
68         }
69     }
70
71     end = ftell(fp); //B를 만나 반복문이 끝날경우 끝난 위치를 체크한다.
72     cur = set - end; // 위치를 계산한다.
73     fseek(fp, cur, SEEK_CUR); //B의 희소행렬의 개수를 세기 위해 시작지점을 세팅한다.
74
75     fscanf(fp, "%c %d %d ", &B_name, &B_row, &B_col); // 파일의 값을 읽어 저장한다.
76     while(!feof(fp)) // 파일 끝까지 반복을 한다.
77     {
78         fgets(check, sizeof(check), fp); // 파일을 한줄 읽는다
79         num_B++; // B의 라인을 증가 시킨다.
80     }
81     rewind(fp); // 파일 위치를 처음으로 돌린다
82     // -----
83     // 이중 동적 할당을 해준다. A행렬과 B행렬에
84     // -----
85     matrix_A = (int**) malloc(sizeof(int *) * num_A); // 이중포인터 matrix_A에 동적 메모리 할당(행)
86     for(i = 0; i < num_A; i++) // 행의 크기만큼 반복
87     {
88         matrix_A[i] = (int*) malloc(sizeof(int) * 3); // 열에 동적 메모리 할당
89     }
90
91     matrix_B = (int**) malloc(sizeof(int *) * num_B); // 이중포인터 matrix_B에 동적 메모리 할당(행)
92     for(i = 0; i < num_B; i++) // 행의 크기만큼 반복
93     {
94         matrix_B[i] = (int*) malloc(sizeof(int) * 3); // 열에 동적 메모리 할당
95     }
96     // -----
97     // 이중 동적 할당을 해준다. A행렬과 B행렬에
98     // -----
99     fgets(check, sizeof(check), fp); //파일을 읽어서 희소 행렬 부분으로 넘어간다.
100
101     sparse_matrix_insert(fp, num_A, matrix_A); // 첫번째 행렬에 파일에 있는 희소 행렬의 값을 넣는 함수 호출
102     sparse_matrix_sort(num_A, matrix_A); // A 희소행렬을 정렬한다.
103     printf("==== A 행렬 ====\n\n");
104     printf("행 열 값\n\n");
105     sparse_matrix_show(num_A, matrix_A); // 행렬을 화면에 출력한다.
106     printf("\n\n");
107
108     fgets(check, sizeof(check), fp); //파일을 읽어서 희소 행렬 부분으로 넘어간다.
```

```

107 fgetc(check, sizeof(check), fp); //파일을 읽어서 희소 행렬 부분으로 넘어간다.
108 fgetc(check, sizeof(check), fp); //파일을 읽어서 희소 행렬 부분으로 넘어간다.
109
110 sparse_matrix_insert(fp, num_B, matrix_B); // 두번째 행렬에 파일에 있는 희소 행렬의 값을 넣는 함수 호출
111 sparse_matrix_sort(num_B, matrix_B); // B 희소행렬을 정렬한다.
112 printf("===== B 행렬 =====\n\n");
113 printf("행 열 값\n\n");
114 sparse_matrix_show(num_B, matrix_B); // 행렬을 화면에 출력한다.
115 printf("\n\n");
116 // -----
117 // -----
118 sparse_matrix_add(A_row, A_col, num_A, matrix_A, B_row, B_col, num_B, matrix_B); // 희소 행렬의 덧셈 함수 호출
119 sparse_matrix_sub(A_row, A_col, num_A, matrix_A, B_row, B_col, num_B, matrix_B); // 희소 행렬의 뺄셈 함수 호출
120 sparse_matrix_mul(A_col, num_A, matrix_A, B_row, num_B, matrix_B); // 희소 행렬의 곱셈 함수 호출
121 printf("===== A 전치 행렬 =====\n\n");
122 sparse_matrix_transpose(num_A, matrix_A); // A전치행렬 호출
123 printf("===== B 전치 행렬 =====\n\n");
124 sparse_matrix_transpose(num_B, matrix_B); // B전치행렬 호출
125
126
127 for(i = 0; i < num_A; i++) // 행의 크기만큼 반복
128     free(matrix_A[i]); // 열의 메모리 해제
129 free(matrix_A); //행의 메모리 해제
130 for(i = 0; i < num_B; i++) // 행의 크기만큼 반복
131     free(matrix_B[i]); // 열의 메모리 해제
132 free(matrix_B); //행의 메모리 해제
133
134
135 fclose(fp); // 파일을 닫는다.
136 }
}

138 void sparse_matrix_insert(FILE *fp, int num, int **matrix) // 이중 동적 메모리에 희소행렬을 집어 넣는다
139 {
140     int i, j;
141     for(i = 0; i < num; i++) // A 행렬의 라인 수 만큼만 반복한다.
142     {
143         for(j = 0; j < 3; j++) // 3번만 반복한다
144         {
145             fscanf(fp, "%d", &matrix[i][j]); // J = 0 행, J = 1 열 J = 2 값을 가르킨다.
146         }
147     }
148 }
}

151 void sparse_matrix_sort(int num, int **matrix) // 행렬을 오름차순으로 정렬한다.
152 {
153     int i, j;
154     int save_row, save_col, save_val;
155     for(i = 0; i < num-1; i++) //행렬의 라인수 - 1 만큼 반복한다
156     {
157         for(j = 0; j < num-1-i; j++) // 행렬의 라인수 - 1 - 1값만큼 반복한다
158         {
159             if(matrix[j][0] > matrix[j+1][0]) // 만약 행의 값이 클경우 위치를 바꿔준다.
160             {
161                 save_row = matrix[j][0];
162                 save_col = matrix[j][1];
163                 save_val = matrix[j][2];
164                 matrix[j][0] = matrix[j+1][0];
165                 matrix[j][1] = matrix[j+1][1];
166                 matrix[j][2] = matrix[j+1][2];
167                 matrix[j+1][0] = save_row;
168                 matrix[j+1][1] = save_col;
169                 matrix[j+1][2] = save_val;
170             }
171         }
172     }
}

173 for(i = 0; i < num-1; i++) // 행의 숫자대로 정렬했으니 다음으로는 열의 크기대로 정렬한다.
174 {
175     for(j = 0; j < num-1-i; j++)
176     {
177         if(matrix[j][0] == matrix[j+1][0]) // 행의 크기가 같을경우
178         {
179             if(matrix[j][1] > matrix[j+1][1]) // 열의 크기를 비교해 클경우 위치를 바꿔준다.
180             {
181                 save_row = matrix[j][0];
182                 save_col = matrix[j][1];
183                 save_val = matrix[j][2];
184                 matrix[j][0] = matrix[j+1][0];
185                 matrix[j][1] = matrix[j+1][1];
186                 matrix[j][2] = matrix[j+1][2];
187                 matrix[j+1][0] = save_row;
188                 matrix[j+1][1] = save_col;
189                 matrix[j+1][2] = save_val;
190             }
191         }
192     }
193 }
194 }
}

```

```

172     },
173     for(i = 0; i < num-1; i++) // 행의 숫자대로 정렬했으니 다음으로는 열의 크기대로 정렬한다.
174     {
175         for(j = 0; j < num-1-i; j++)
176         {
177             if(matrix[j][0] == matrix[j+1][0]) // 행의 크기가 같을경우
178             {
179                 if(matrix[j][1] > matrix[j+1][1]) // 열의 크기를 비교해 클경우 위치를 바꿔준다.
180                 {
181                     save_row = matrix[j][0];
182                     save_col = matrix[j][1];
183                     save_val = matrix[j][2];
184                     matrix[j][0] = matrix[j+1][0];
185                     matrix[j][1] = matrix[j+1][1];
186                     matrix[j][2] = matrix[j+1][2];
187                     matrix[j+1][0] = save_row;
188                     matrix[j+1][1] = save_col;
189                     matrix[j+1][2] = save_val;
190                 }
191             }
192         }
193     }
194 }
195
222     },
223     else
224     {
225         for(i = 0; i < num_A; i++) // matrix_C의 몇개의 동적할당을 해줄지 결정하는 부분이다.
226         {
227             for(j = 0; j < num_B; j++)
228             {
229                 if((matrix_A[i][0] == matrix_B[j][0]) && (matrix_A[i][1] == matrix_B[j][1])) // 만약 행과 열이 같을 경우 연산이 가능하다.
230                 {
231                     if((matrix_A[i][2] + matrix_B[j][2]) == 0)
232                     break;
233                     num_C++; // 동적할당 갯수 하나 추가
234                 }
235             }
236         }
237         i = 0;
238         while(i < num_A) // A의 라인 수 만큼 반복한다.
239         {
240             for(j = 0; j < num_B; j++) // B의 라인수 만큼 반복한다.
241             {
242                 if((matrix_A[i][0] != matrix_B[j][0]) || (matrix_A[i][1] != matrix_B[j][1])) // 만약 행 혹은 열을 비교해서 다를경우 check 값을 증가시킨다
243                 check++;
244             }
245             if(check == num_B) //B행렬의 라인수가 check와 같을 시 행과 열이 겹치는게 없다는 뜻이므로 동적할당 갯수를 하나 추가한다.
246             num_C++;
247             i++;
248             check = 0;
249         }
250         i = 0;
251         while(i < num_B) // 반대로 B를 기준으로 다른지 찾아본다.
252         {
253             for(j = 0; j < num_A; j++) // a의 라인수 만큼 반복한다.
254             {
255                 if((matrix_B[j][0] != matrix_A[i][0]) || (matrix_B[j][1] != matrix_A[i][1])) // 만약 행 혹은 열을 비교해서 다를경우 check 값을 증가시킨다
256                 check++;
257             }
258             if(check == num_A) //A행렬의 라인수가 check와 같을 시 행과 열이 겹치는게 없다는 뜻이므로 동적할당 갯수를 하나 추가한다.
259             num_C++;
260             i++;
261             check = 0;
262         }
263
264         matrix_C = (int**) malloc(sizeof(int *) + num_C); // 위에서 동적할당 갯수를 찾았기 때문에 갯수만큼 동적 메모리를 할당해준다.
265         for(i = 0; i < num_C; i++) // 갯수 만큼 반복한다.
266         {
267             matrix_C[i] = (int*) malloc(sizeof(int) * 3); // 행, 열, 값 부분을 만들어 준다.
268         }
269         for(i = 0; i < num_C; i++) // 행의 크기만큼 반복
270         {
271             for(j = 0; j < 3; j++) // 열의 크기만큼 반복
272             {
273                 matrix_C[i][j] = 0; // 행렬을 0으로 초기화(쓰레기 값 처리 구문)
274             }
275         }
276
277         for(i = 0; i < num_A; i++)
278         {
279             for(j = 0; j < num_B; j++)
280             {
281                 if((matrix_A[i][0] == matrix_B[j][0]) && (matrix_A[i][1] == matrix_B[j][1])) // 만약 A와 B의 행과 열이 모두 같을 경우 덧셈이 가능하기 때문에 조건문을 걸어준다.
282                 {
283                     if((matrix_A[i][2] + matrix_B[j][2]) == 0) // 단 둘의 합이 0일경우는 제외한다. 최소행렬은 값이 0일 경우 없애줘야한다.
284                     break;
285                     matrix_C[k][0] = matrix_A[i][0];
286                     matrix_C[k][1] = matrix_A[i][1];
287                     matrix_C[k][2] = matrix_A[i][2] + matrix_B[j][2];
288                     k++;
289                 }
290             }
291         }
292         i = 0;
293         check = 0;
294         while(i < num_A) // A의 라인 수 만큼 반복한다.
295         {
296             for(j = 0; j < num_B; j++) // B의 라인 수 만큼 반복한다.
297             {
298                 if((matrix_A[i][0] != matrix_B[j][0]) || (matrix_A[i][1] != matrix_B[j][1])) // 행과 열 모두 다른경우
299                 check++; // 증가시킨다.

```

```

290     }
291 }
292 i = 0;
293 check = 0;
294 while(i < num_A) // A의 라인 수 만큼 반복한다.
295 {
296     for(j = 0; j < num_B; j++) // B의 라인 수 만큼 반복한다.
297     {
298         if((matrix_A[i][0] != matrix_B[j][0]) || (matrix_A[i][1] != matrix_B[j][1])) // 행과 열 모두 다른경우
299             check++; // 증가시킨다.
300     }
301     if(check == num_B) // 만약 같을 경우 i를 기점으로 한 A행렬은 0과 더해진것이다.
302     {
303         matrix_C[k][0] = matrix_A[i][0];
304         matrix_C[k][1] = matrix_A[i][1];
305         matrix_C[k][2] = matrix_A[i][2];
306         k++;
307     }
308     check = 0;
309     i++;
310 }
311 i = 0;
312 check = 0;
313 while(i < num_B) // B의 라인 수 만큼 반복한다.
314 {
315     for(j = 0; j < num_A; j++) // A의 라인 수 만큼 반복한다.
316     {
317         if((matrix_A[j][0] != matrix_B[i][0]) || (matrix_A[j][1] != matrix_B[i][1])) // 행과 열 모두 다른경우
318             check++;
319     }
320     if(check == num_A) // 만약 같을 경우 i를 기점으로 한 B행렬은 0과 더해진것이다.
321     {
322         matrix_C[k][0] = matrix_B[i][0];
323         matrix_C[k][1] = matrix_B[i][1];
324         matrix_C[k][2] = matrix_B[i][2];
325         k++;
326     }
327     check = 0;
328     i++;
329 }

```

```

330     printf("===== A+B 행렬 =====\n\n");
331     printf("행 열 값\n\n");
332     sparse_matrix_sort(num_C, matrix_C); // 정렬한다.
333     sparse_matrix_show(num_C, matrix_C); // 행렬의 결과를 출력한다.
334     printf("\n\n");
335
336     for(i = 0; i < num_C; i++) // 동적할당을 풀어준다.
337         free(matrix_C[i]);
338     free(matrix_C);
339 }
340 }
341
342 void sparse_matrix_sub(int row_A, int col_A, int num_A, int **matrix_A, int row_B, int col_B, int num_B, int **matrix_B) // 뺄셈 함수
343 {
344     int **matrix_C; // 행렬의 뺄셈을 저장할 이중 포인터
345     int i, j, k = 0;
346     int num_C = 0;
347     int check = 0;
348     // -----
349     if((row_A != row_B) || (col_A != col_B)) // 우선 행렬의 뺄셈을 하기 위해서는 행과 열의 값이 같아야 한다.
350     {
351         printf("===== A - B 행렬 =====\n\n\n");
352         printf("A - B는 불가능 합니다\n\n");
353         printf("\n\n");
354     }
355     else
356     {
357         for(i = 0; i < num_A; i++) // matrix_C의 몇개의 동적할당을 해줄지 결정하는 부분이다.
358         {
359             for(j = 0; j < num_B; j++)
360             {
361                 if((matrix_A[i][0] == matrix_B[j][0]) && (matrix_A[i][1] == matrix_B[j][1])) // 만약 행과 열이 같을 경우 연산이 가능하다.
362                 {
363                     if((matrix_A[i][2] - matrix_B[j][2]) == 0)
364                         break;
365                     num_C++; // 동적할당 갯수 하나 추가
366                 }
367             }
368             i = 0;
369         }
370         while(i < num_A) // A의 라인 수 만큼 반복한다.
371         {

```

```

372             while(i < num_A) // A의 라인 수 만큼 반복한다.
373             {
374                 for(j = 0; j < num_B; j++) // B의 라인 수 만큼 반복한다.
375                 {
376                     if((matrix_A[i][0] != matrix_B[j][0]) || (matrix_A[i][1] != matrix_B[j][1])) // 만약 행 혹은 열을 비교해서 다를경우 check 값을 증가시킨다
377                         check++;
378                 }
379                 if(check == num_B) // B행렬의 라인수가 check와 같을 시 행과 열이 겹치는게 없다는 뜻이므로 동적할당 갯수를 하나 추가한다.
380                     num_C++;
381                 i++;
382                 check = 0;
383             }
384             i = 0;
385             while(i < num_B) // 반대로 B를 기준으로 다른지 찾아본다.
386             {
387                 for(j = 0; j < num_A; j++) // A의 라인 수 만큼 반복한다.
388                 {
389                     if((matrix_B[i][0] != matrix_A[j][0]) || (matrix_B[i][1] != matrix_A[j][1])) // 만약 행 혹은 열을 비교해서 다를경우 check 값을 증가시킨다
390                         check++;
391                 }
392                 if(check == num_A) // A행렬의 라인수가 check와 같을 시 행과 열이 겹치는게 없다는 뜻이므로 동적할당 갯수를 하나 추가한다.
393                     num_C++;
394                 i++;
395                 check = 0;
396             }
397         }
398         matrix_C = (int**) malloc(sizeof(int) * num_C); // 위에서 동적할당 갯수를 찾았기 때문에 갯수만큼 동적 메모리를 할당해준다.
399         for(i = 0; i < num_C; i++)
400         {
401             matrix_C[i] = (int*) malloc(sizeof(int) * 3); // 행, 열, 값 부분을 만들어 준다.
402         }
403         for(i = 0; i < num_C; i++) // 행의 크기만큼 반복
404         {
405             for(j = 0; j < 3; j++) // 열의 크기만큼 반복
406             {
407                 matrix_C[i][j] = 0; // 행렬을 0으로 초기화(쓰레기 값 처리 구문)
408             }
409         }
410     }

```



```

406     for(i = 0; i < num_A; i++)
407     {
408         for(j = 0; j < num_B; j++)
409         {
410             if((matrix_A[i][0] == matrix_B[j][0]) && (matrix_A[i][1] == matrix_B[j][1]))// 만약 A와 B의 행과 열이 모두 같을 경우 덧셈이 가능하기
411             {
412                 if((matrix_A[i][2] - matrix_B[j][2]) == 0)// 단 둘의 차이가 0일경우는 제외한다. 최소행렬은 값이 0일 경우 없애줘야한다.
413                     break;
414                 matrix_C[k][0] = matrix_A[i][0];
415                 matrix_C[k][1] = matrix_A[i][1];
416                 matrix_C[k][2] = matrix_A[i][2] - matrix_B[j][2];
417                 k++;
418             }
419         }
420     }
421     i = 0;
422     check = 0;
423     while(i < num_A)// A의 라인 수 만큼 반복한다.
424     {
425         for(j = 0; j < num_B; j++)// B의 라인 수 만큼 반복한다.
426         {
427             if((matrix_A[i][0] != matrix_B[j][0]) || (matrix_A[i][1] != matrix_B[j][1]))// 행과 열 모두 다른경우
428                 check++;
429         }
430         if(check == num_B)// 만약 같을 경우 i를 기점으로 한 A행렬은 0과 더해진것이다.
431         {
432             matrix_C[k][0] = matrix_A[i][0];
433             matrix_C[k][1] = matrix_A[i][1];
434             matrix_C[k][2] = matrix_A[i][2];
435             k++;
436         }
437         check = 0;
438         i++;
439     }
440     i = 0;
441     check = 0;
442     while(i < num_B)// B의 라인 수 만큼 반복한다.
443     {
444         for(j = 0; j < num_A; j++)// A의 라인 수 만큼 반복한다.
445         {

```

```

446             if((matrix_A[j][0] != matrix_B[i][0]) || (matrix_A[j][1] != matrix_B[i][1]))// 행과 열 모두 다른경우
447                 check++;
448             }
449             if(check == num_A)// 만약 같을 경우 i를 기점으로 한 B행렬은 0과 더해진것이다.
450             {
451                 matrix_C[k][0] = matrix_B[i][0];
452                 matrix_C[k][1] = matrix_B[i][1];
453                 matrix_C[k][2] = -matrix_B[i][2];
454                 k++;
455             }
456             check = 0;
457             i++;
458         }
459         printf("===== A-B 행렬 =====\n\n");
460         printf("행 열 값\n\n");
461         sparse_matrix_sort(num_C, matrix_C); // 정렬한다
462         sparse_matrix_show(num_C, matrix_C); // 행렬의 결과를 출력한다.
463         printf("\n\n");
464
465         for(i = 0; i < num_C; i++) // 동적할당을 풀어준다.
466             free(matrix_C[i]);
467         free(matrix_C);
468     }
469 }
470
471 void sparse_matrix_transpose(int num, int **matrix) // 전치 행렬 함수
472 {
473     int **matrix_C;
474     int i, j;
475     matrix_C = (int**) malloc(sizeof(int) * num);
476     for(i = 0; i < num; i++)
477     {
478         matrix_C[i] = (int*) malloc(sizeof(int) * 3);
479     }
480     for(i = 0; i < num; i++) // 행의 크기만큼 반복
481     {
482         for(j = 0; j < 3; j++) // 열의 크기만큼 반복
483         {
484             matrix_C[i][j] = 0; // 행렬을 0으로 초기화(쓰레기 값 처리 구문)

```

```

485         }
486     }
487     for(i = 0; i < num; i++) // 행과 열을 바꿔주는 함수이다
488     {
489         matrix_C[i][0] = matrix[i][1]; // 행을 열로
490         matrix_C[i][1] = matrix[i][0]; // 열을 행으로
491         matrix_C[i][2] = matrix[i][2]; // 값은 그대로
492     }
493     sparse_matrix_sort(num, matrix_C); // 정렬하날
494     printf("행 열 값\n\n");
495     sparse_matrix_show(num, matrix_C); // 값을 출력한다
496     printf("\n\n");
497     for(i = 0; i < num; i++) // 라인만큼 반복한다
498         free(matrix_C[i]); // 열을 해제한다
499     free(matrix_C); // 행을 해제한다
500 }
501
502 int sparse_matrix_mul_check(int **matrix_A, int num_A, int **matrix_B, int num_B) // 행렬의 곱의 갯수를 파악하는 함수
503 {
504     int i, j, n, m;
505     int num = 0, sum = 0;
506     int row, col;
507     int save_row, save_col, save_val; // 정렬을 위한 변수
508     int **matrix_C;
509     // -----
510     // matrix_C는 B행렬을 전치해주는 이중 동적 할당이다.
511     // -----
512     matrix_C = (int**) malloc(sizeof(int) * num_B);
513     for(i = 0; i < num_B; i++)
514     {
515         matrix_C[i] = (int*) malloc(sizeof(int) * 3);
516     }
517     for(i = 0; i < num_B; i++) // 행의 크기만큼 반복
518     {
519         for(j = 0; j < 3; j++) // 열의 크기만큼 반복
520         {
521             matrix_C[i][j] = 0; // 행렬을 0으로 초기화(쓰레기 값 처리 구문)
522         }
523     }

```

```

523     }
524     for(i = 0; i < num_B; i++) // 전치 행렬 부분
525     {
526         matrix_C[i][0] = matrix_B[i][1];
527         matrix_C[i][1] = matrix_B[i][0];
528         matrix_C[i][2] = matrix_B[i][2];
529     }
530     // -----
531     // 행렬의 곱의 갯수를 세기 위해서는 전치행렬로 만든 값을 정렬을 해서 오름차순으로 바꿔서 비교해야한다.
532     // -----
533     for(i = 0; i < num_B-1; i++)//행렬의 라인수 - 1 만큼 반복한다
534     {
535         for(j = 0; j < num_B-1-i; j++)// 행렬의 라인수 - 1 - i값만큼 반복한다
536         {
537             if(matrix_C[j][0] > matrix_C[j+1][0]) // 만약 행의 값이 클경우 위치를 바꿔준다.
538             {
539                 save_row = matrix_C[j][0];
540                 save_col = matrix_C[j][1];
541                 save_val = matrix_C[j][2];
542                 matrix_C[j][0] = matrix_C[j+1][0];
543                 matrix_C[j][1] = matrix_C[j+1][1];
544                 matrix_C[j][2] = matrix_C[j+1][2];
545                 matrix_C[j+1][0] = save_row;
546                 matrix_C[j+1][1] = save_col;
547                 matrix_C[j+1][2] = save_val;
548             }
549         }
550     }
551 }
552 for(i = 0; i < num_B-1; i++)// 행의 숫자대로 정렬했으니 다음으로는 열의 크기대로 정렬한다.
553 {
554     for(j = 0; j < num_B-1-i; j++)
555     {
556         if(matrix_C[j][0] == matrix_C[j+1][0])// 행의 크기가 같을경우
557         {
558             if(matrix_C[j][1] > matrix_C[j+1][1])// 열의 크기를 비교해 클경우 위치를 바꿔준다.
559             {
560                 save_row = matrix_C[j][0];
561                 save_col = matrix_C[j][1];
562                 save_val = matrix_C[j][2];

```

```

562                 save_val = matrix_C[j][2];
563                 matrix_C[j][0] = matrix_C[j+1][0];
564                 matrix_C[j][1] = matrix_C[j+1][1];
565                 matrix_C[j][2] = matrix_C[j+1][2];
566                 matrix_C[j+1][0] = save_row;
567                 matrix_C[j+1][1] = save_col;
568                 matrix_C[j+1][2] = save_val;
569             }
570         }
571     }
572 }
573 // -----
574 // 행렬의 곱의 갯수를 카운트 하는 부분이다.
575 // -----
576 for(i=0; i<num_A; )//A 라인을 기준으로 출발한다
577 {
578     row = matrix_A[i][0]; // 기본 행을 설정한다.
579     for(j=0; j<num_B; )//같은 B 라인을 기준으로 한다
580     {
581         col = matrix_C[j][0]; //기본 열을 설정한다.
582         sum=0;
583         for(n=0, m=0; n < num_A && m < num_B && matrix_A[n][0] <= row && matrix_C[m][0] <= col; ) // 기본 행과 열을 비교하면서 동적 할당된 부분을 넘어가지 않게 막아주는 반복문이다.
584         {
585             if (matrix_A[n][0] != row) // 만약 행의 값이 다를경우 n값을 증가시킨다
586                 n++;
587             if(matrix_C[m][0] != col) // 기본 열의 값이 다를경우 m값을 증가시킨다
588                 m++;
589             if (matrix_A[n][0] == row && matrix_C[m][0] == col) // 두개의 행과 열이 동시에 같을 경우 진행한다
590             {
591                 if (matrix_A[n][1] < matrix_C[m][1]) // A 열이 C열보다 작을경우 n값을 증가한다
592                     n++;
593                 else if (matrix_A[n][1] > matrix_C[m][1]) //반대로 A열이 클경우 m 값을 증가한다.
594                     m++;
595                 else
596                 {
597                     sum = sum + (matrix_A[n][2] * matrix_C[m][2]); // 둘다 아닐경우 같다는 의미로 sum 값에 값을 곱해준다.
598                     n++;
599                     m++;
600                 }
601             }
602         }

```

```

605         if(sum!=0) // sum이 0이 아니면 num 값을 증가시켜 행렬의 곱의 갯수를 증가시켜준다.
606         {
607             sum = 0;
608             num++;
609         }
610         while(j<num_B-1 && matrix_C[j][0]!=col) // 반복문이 종료되고 j값이 B라인수-1 보다 작으면서 B행 값이 기본 열과 같을때까지 j값을 올리면서 체크한다.
611             j++;
612     }
613     while(i<num_A-1 && matrix_A[i][0]==row) // 반복문이 종료되고 i값이 A라인수-1 보다 작으면서 A행 값이 기본 열과 같을때까지 i값을 올리면서 체크한다.
614         i++;
615 }
616 }
617 return num;// 행렬의 곱 갯수 값을 돌려준다.
618 }
619
620 void sparse_matrix_mul(int col_A, int num_A, int **matrix_A, int row_B, int num_B, int **matrix_B) // 행렬의 곱 함수
621 {
622     int num_D;
623     int **matrix_C;
624     int **matrix_D;
625     int i, j, k = 0;
626     int row, col;
627     int sum = 0;
628     int n, m;
629     if(col_A != row_B) // 행렬의 곱의 조건에 맞춰 A의 열과 B의 행이 다르면 계산이 불가능하다.
630     {
631         printf("===== A * B 행렬 =====\n\n\n");
632         printf("A * B는 불가능 합니다 \n\n");
633         printf("\n\n\n");
634     }
635     else
636     {
637         num_D = sparse_matrix_mul_check(matrix_A, num_A, matrix_B, num_B); // 행렬의 곱을 저장하기 위해 이중 동적 행렬의 갯수를 정해줘야 하기 때문에 호출한다.
638         matrix_C = (int**) malloc(sizeof(int *) * num_B); // 전치행렬로 곱을 처리하는 것이 편해 전치행렬 matrix_C를 만들어준다.
639         for(i = 0; i < num_B; i++) // B의 전치행렬이기에 B라인 으로 계산한다.
640         {
641             matrix_C[i] = (int*) malloc(sizeof(int ) * 3);
642         }
643         for(i = 0; i < num_B; i++) // 행의 크기만큼 반복

```

```

605         if(sum!=0) // sum이 0이 아니면 num 값을 증가시켜 행렬의 곱의 갯수를 증가시켜준다.
606         {
607             sum = 0;
608             num++;
609         }
610
611         while(j<=num_B-1 && matrix_C[j][0]==col) // 반복문이 종료되고 j값이 B라인수-1 보다 작으면서 B행 값이 기본 열과 같을때까지 j값을 올리면서 체크한다.
612             j++;
613     }
614     while(i<=num_A-1 && matrix_A[i][0]==row) // 반복문이 종료되고 i값이 A라인수-1 보다 작으면서 A행 값이 기본 열과 같을때까지 j값을 올리면서 체크한다.
615         i++;
616 }
617 return num; // 행렬의 곱 갯수 값을 돌려준다.
618 }
619
620 void sparse_matrix_mul(int col_A, int num_A, int **matrix_A, int row_B, int num_B, int **matrix_B) // 행렬의 곱 함수
621 {
622     int num_D;
623     int **matrix_C;
624     int **matrix_D;
625     int i, j, k = 0;
626     int row, col;
627     int sum = 0;
628     int n, m;
629     if(col_A != row_B) // 행렬의 곱의 조건에 맞춰 A의 열과 B의 행이 다르면 계산이 불가능하다.
630     {
631         printf("===== A * B 행렬 =====\n\n\n");
632         printf(" A * B는 불가능 합니다 \n\n");
633         printf("\n\n\n");
634     }
635     else
636     {
637         num_D = sparse_matrix_mul_check(matrix_A, num_A, matrix_B, num_B); // 행렬의 곱을 저장하기 위해 이중 동적 행렬의 갯수를 정해줘야 하기 때문에 호출한다.
638         matrix_C = (int**) malloc(sizeof(int) * num_B); // 전치행렬로 곱을 처리하는 것이 편해 전치행렬 matrix_C를 만들어준다.
639         for(i = 0; i < num_B; i++) // B의 전치행렬이기에 B라인 으로 계산한다.
640         {
641             matrix_C[i] = (int*) malloc(sizeof(int) * 3);
642         }
643         for(i = 0; i < num_B; i++) // 행의 크기만큼 반복

```

```

682
683         for(n=0, m=0; n < num_A && m < num_B && matrix_A[n][0] <= row && matrix_C[m][0] <= col;) // 기본 행과 열을 비교하면서 동적 할당된 부분을 넘어가지 않게 막아주는 반복문이다.
684         {
685             if (matrix_A[n][0] != row) // 만약 행의 값이 다들 경우 n값을 증가시킨다
686                 n++;
687             if(matrix_C[m][0] != col) // 기본 열의 값이 다들 경우 m값을 증가시킨다
688                 m++;
689             if (matrix_A[n][0] == row && matrix_C[m][0] == col) // 두개의 행과 열이 동시에 같을 경우 진행한다
690             {
691                 if (matrix_A[n][1] < matrix_C[m][1]) // A 열이 C열보다 작을 경우 n값을 증가한다
692                     n++;
693                 else if (matrix_A[n][1] > matrix_C[m][1]) //반대로 A열이 클 경우 m 값을 증가한다.
694                     m++;
695                 else
696                 {
697                     sum = sum + (matrix_A[n][2] * matrix_C[m][2]); // 둘다 아닐 경우 같다는 의미로 sum 값에 값을 곱해준다.
698                     n++;
699                     m++;
700                 }
701             }
702         }
703         if(sum!=0) // sum이 0이 아니면 위에서 만들어 둔 희소행렬의 저장 공간에 행 열 값을 저장해준다
704         {
705             matrix_D[k][0] = row;
706             matrix_D[k][1] = col;
707             matrix_D[k][2] = sum;
708             sum = 0;
709             k++;
710         }
711         while(j<=num_B-1 && matrix_C[j][0]==col) // 반복문이 종료되고 j값이 B라인수-1 보다 작으면서 B행 값이 기본 열과 같을때까지 j값을 올리면서 체크한다.
712             j++;
713     }
714     while(i<=num_A-1 && matrix_A[i][0]==row) // 반복문이 종료되고 i값이 A라인수-1 보다 작으면서 A행 값이 기본 열과 같을때까지 j값을 올리면서 체크한다.
715         i++;
716 }
717 printf("===== A+B 행렬 =====\n\n\n");
718 printf(" 행 열 값 \n\n\n");
719 sparse_matrix_show(num_D, matrix_D); // 희소행렬의 곱을 출력한다.
720
721 printf("\n\n");

```

## 1.3 소스코드 분석

```
// -----  
//      필요한 변수들을 선언한다.  
// -----  
FILE *fp; // 파일포인터 함수  
int A_row, A_col, num_A = 0; // A배열에 저장할 변수  
int B_row, B_col, num_B = 0; // B배열에 저장할 변수  
int set = 0, end = 0, cur = 0; //파일을 읽으면서 위치를 저장하는 변수  
int **matrix_A; // A 행렬을 저장하기 위해 이중포인터로 변수 선언  
int **matrix_B; // B 행렬을 저장하기 위해 이중포인터로 변수 선언  
int i;  
char A_name, B_name;  
char check[100]; //fgets의 한줄을 읽기 위한 변수  
char B_Check; // B가 들어오는지 확인하기 위한 변수
```

1. 필요한 변수들을 선언해 준다.
2. check[100] fgets 함수를 위한 저장소이다.
3. set end cur 변수는 위치를 통해 라인을 세기 위한 변수이다.
4. b\_check 변수는 b가 들어오면 반복을 멈추고 b행렬로 넘어가기 위한 조건이다

```
fscanf(fp, "%c %d %d ", &A_name, &A_row, &A_col); // 파일의 첫번째에 들어있는 값을 읽어 저장한다.  
while(1) // A의 최소행렬의 갯수를 확인하기 위한 반복문  
{  
    fgets(check, sizeof(check), fp); // 한 문장을 읽는다  
    B_Check = check[0]; // 처음 값을 읽는다  
  
    if(B_Check == 'B' || B_Check == 'b') // 처음값이 B일경우 즉시 종료한다.  
    {  
        break;  
    }  
    else  
    {  
        num_A++; // 아닐경우 A의 라인을 증가 시킨다.  
        set = ftell(fp); // 파일의 위치를 체크해준다.  
    }  
}  
end = ftell(fp); //B를 만나 반복문이 끝날경우 끝난 위치를 체크한다.  
cur = set - end; //위치를 계산한다.  
fseek(fp, cur, SEEK_CUR); //B의 최소행렬의 갯수를 세기 위해 시작지점을 세팅한다.
```

5. 우선 한줄을 입력 받아 행과 열 이름을 기록한다.
6. 반복문으로 들어간다. 한줄씩 읽어서 check[100]에 저장을 하면서 한줄씩 내려 간다
7. num\_A라는 값을 하나씩 올려준다
8. B가 나올경우 반복문에서 빠져나옴으로써 라인 개수를 확인한다.
9. ftell()을 이용해 시작 위치를 조정해준다. 조정을 하게 되면 나중에 시작할 위치

를 정해 B 위치에서 시작할 수 있게 할 수 있다.

10. fseek() 함수를 통해 B 행렬의 시작지점을 설정해준다.

```
fscanf(fp, "%c %d %d ", &B_name, &B_row, &B_col); // 파일의 값을 읽어 저장한다.
while(!feof(fp)) // 파일 끝까지 반복을 한다.
{
    fgets(check, sizeof(check), fp); // 파일을 한줄 읽는다
    num_B++; // B의 라인을 증가 시킨다.
}
rewind(fp); // 파일 위치를 처음으로 돌린다
..
```

11. B도 마찬가지로 희소행렬의 개수를 카운트 해준다

12. 파일에 끝에 도달하면 반복을 멈추고 파일 위치를 처음으로 돌린다.

```
matrix_A = (int**) malloc(sizeof(int *) * num_A); // 이중포인터 matrix_A에 동적 메모리 할당(행)
for(i = 0; i < num_A; i++) // 행의 크기만큼 반복
{
    matrix_A[i] = (int*) malloc(sizeof(int) * 3); // 열에 동적 메모리 할당
}

matrix_B = (int**) malloc(sizeof(int *) * num_B); // 이중포인터 matrix_B에 동적 메모리 할당(행)
for(i = 0; i < num_B; i++) // 행의 크기만큼 반복
{
    matrix_B[i] = (int*) malloc(sizeof(int) * 3); // 열에 동적 메모리 할당
}
..
```

13. 각자 희소행렬의 개수를 알고있기 때문에 각각 동적 메모리를 할당한다.

14. matrix[a][i] 에서 i값 0 = 행 1 = 열 2 = 값이다.

```
for(i = 0; i < num_A; i++) // 행의 크기만큼 반복
    free(matrix_A[i]); // 열의 메모리 해제
free(matrix_A); //행의 메모리 해제
for(i = 0; i < num_B; i++) // 행의 크기만큼 반복
    free(matrix_B[i]); // 열의 메모리 해제
free(matrix_B); //행의 메모리 해제

fclose(fp); // 파일을 닫는다.
```

추후 구문들이 모두 돌게 되면 메모리를 해제한다 또한 파일도 닫는다.

## Insert 함수

```
void sparse_matrix_insert(FILE *fp, int num, int **matrix) // 이중 동적 메모리에 희소행렬을 집어 넣는다
{
    int i, j;
    for(i = 0; i < num; i++) // A 행렬의 라인 수 만큼만 반복한다.
    {
        for(j = 0; j < 3; j++) // 3번만 반복한다
        {
            fscanf(fp, "%d", &matrix[i][j]); // J = 0 행, J = 1 열 J = 2 값을 가르킨다.
        }
    }
}
```

## Sort 함수

```
void sparse_matrix_sort(int num, int **matrix) // 행렬을 오름차순으로 정렬한다.
{
    int i, j;
    int save_row, save_col, save_val;
    for(i = 0; i < num-1; i++) //행렬의 라인수 - 1 만큼 반복한다
    {
        for(j = 0; j < num-1-i; j++) // 행렬의 라인수 - 1 - i값만큼 반복한다
        {
            if(matrix[j][0] > matrix[j+1][0]) // 만약 행의 값이 클경우 위치를 바꿔준다.
            {
                save_row = matrix[j][0];
                save_col = matrix[j][1];
                save_val = matrix[j][2];
                matrix[j][0] = matrix[j+1][0];
                matrix[j][1] = matrix[j+1][1];
                matrix[j][2] = matrix[j+1][2];
                matrix[j+1][0] = save_row;
                matrix[j+1][1] = save_col;
                matrix[j+1][2] = save_val;
            }
        }
    }

    for(i = 0; i < num-1; i++) // 행의 숫자대로 정렬했으니 다음으로는 열의 크기대로 정렬한다.
    {
        for(j = 0; j < num-1-i; j++)
        {
            if(matrix[j][0] == matrix[j+1][0]) // 행의 크기가 같을경우
            {
                if(matrix[j][1] > matrix[j+1][1]) // 열의 크기를 비교해 클경우 위치를 바꿔준다.
                {
                    save_row = matrix[j][0];
                    save_col = matrix[j][1];
                    save_val = matrix[j][2];
                    matrix[j][0] = matrix[j+1][0];
                    matrix[j][1] = matrix[j+1][1];
                    matrix[j][2] = matrix[j+1][2];
                    matrix[j+1][0] = save_row;
                    matrix[j+1][1] = save_col;
                    matrix[j+1][2] = save_val;
                }
            }
        }
    }
}
```

1. 우선 희소행렬을 받은 후 희소행렬의 개수만큼 반복을 한다.
2. 처음 값을 저장한 후 조건에 부합하면 위치를 바꿔주는 정렬을 진행한다.
3. 처음에는 행을 기준으로 출발을 한다. 행이 클 경우 뒤로 작을 경우 앞으로 간다.
4. 모두 정리하면 행이 같은거끼리 열을 비교해 열이 크면 뒤로 열이 작으면 앞으로 옮기는 정렬을 한번 더 해준다.
5. 이 정렬은 추후 계산에 큰 도움을 준다.

## ADD, SUB 함수

(add와 sub 함수는 + - 차이)

```

...
if((row_A != row_B) || (col_A != col_B)) // 우선 행렬의 덧셈을 하기 위해서는 행과 열의 값이 같아야 한다.
{
    printf("===== A + B 행렬 =====\n\n\n");
    printf("    A + B는 불가능 합니다    \n");
    printf("\n\n");
}
else

```

1. 행렬의 덧셈과 뺄셈을 하기 위해서는 행과 열 둘중 하나라도 다를경우 계산이 불가능 해진다.

```

for(i = 0; i < num_A; i++) // matrix_C의 몇개의 동적할당을 해줄지 결정하는 부분이다.
{
    for(j = 0; j < num_B; j++)
    {
        if((matrix_A[i][0] == matrix_B[j][0]) && (matrix_A[i][1] == matrix_B[j][1])) // 만약 행과 열이 같을 경우 연산이 가능하다.
        {
            if((matrix_A[i][2] + matrix_B[j][2]) == 0)
                break;
            num_C++; // 동적할당 갯수 하나 추가
        }
    }
}
,

```

2. 만약 조건에 부합했다면 조건을 통해 동적할당 개수를 정해준다.
3. 첫번째 조건은 행과 열이 같으면 덧셈과 뺄셈이 가능하기에 골라주는 조건이다.
4. 단 둘이 덧셈에서는 더해서 0이되거나 뺄셈에서는 빼면 0 이 되는 부분은 희소행렬에 넣을 수 없기에 중간에 조건을 걸고 빠져나온다

```

i = 0;
while(i < num_A) // A의 라인 수 만큼 반복한다.
{
    for(j = 0; j < num_B; j++) // B의 라인수 만큼 반복한다.
    {
        if((matrix_A[i][0] != matrix_B[j][0]) || (matrix_A[i][1] != matrix_B[j][1])) // 만약 행 혹은 열을 비교해서 다를경우 check 값을 증가시킨다
            check++;
    }
    if(check == num_B) //B행렬의 라인수가 check와 같을 시 행과 열이 겹치는게 없다는 뜻이므로 동적할당 갯수를 하나 추가한다.
        num_C++;
    i++;
    check = 0;
}

```

5. 라인 수를 받아 이번에는 둘중에 하나만 같을 경우 check를 올려준다.

한 희소행렬의 행과 열을 가지고 다른 행렬의 희소행렬을 모두 비교를 통해 같은 것이 없을 경우 0과 더해주면 된다. 뺄셈도 마찬가지이다.

```

matrix_C = (int**) malloc(sizeof(int *) * num_C); // 위에서 동적할당 갯수를 찾았기 때문에 갯수만큼 동적 메모리를 할당해준다.
for(i = 0; i < num_C; i++) // 갯수 만큼 반복한다.
{
    matrix_C[i] = (int*) malloc(sizeof(int) * 3); // 행, 열, 값 부분을 만들어 준다.
}

```

6. 구해진 덧셈의 동적할당 개수를 가지고 동적 메모리를 할당한다.

```

for(i = 0; i < num_A; i++)
{
    for(j = 0; j < num_B; j++)
    {
        if((matrix_A[i][0] == matrix_B[j][0]) && (matrix_A[i][1] == matrix_B[j][1])) // 만약 A와 B의 행과 열이 모두 같을 경우 덧셈이 가능하기 때문에 조건문을 걸어준다.
        {
            if((matrix_A[i][2] + matrix_B[j][2]) == 0) // 단 둘의 합이 0일경우는 제외한다. 희소행렬은 값이 0일 경우 없애줘야한다.
                break;
            matrix_C[k][0] = matrix_A[i][0];
            matrix_C[k][1] = matrix_A[i][1];
            matrix_C[k][2] = matrix_A[i][2] + matrix_B[j][2];
            k++;
        }
    }
}

```

7. 위에 조건처럼 같을 경우는 그냥 계산을 진행하면 되는 반복 조건문이다. 단 둘의 합 혹은 차가 0이면 희소행렬에 들어갈 수 없기에 조건을 걸어 확인해야한다.

```

i = 0;
check = 0;
while(i < num_A) // A의 라인 수 만큼 반복한다.
{
    for(j = 0; j < num_B; j++) // B의 라인 수 만큼 반복한다.
    {
        if((matrix_A[i][0] != matrix_B[j][0]) || (matrix_A[i][1] != matrix_B[j][1])) // 행과 열 모두 다른경우
            check++; // 증가시킨다.
    }
    if(check == num_B) // 만약 같을 경우 i를 기점으로 한 A행렬은 0과 더해진것이다.
    {
        matrix_C[k][0] = matrix_A[i][0];
        matrix_C[k][1] = matrix_A[i][1];
        matrix_C[k][2] = matrix_A[i][2];
        k++;
    }
    check = 0;
    i++;
}

```

8. 아무것도 겹치지 않을 경우는 0과 더하는 방법을 표현하기 위해 빈 공간에 겹치지 않는 열, 행, 값을 대입한다.



## Translate 함수

```
matrix_C = (int**) malloc(sizeof(int *) * num);
for(i = 0; i < num; i++)
{
    matrix_C[i] = (int*) malloc(sizeof(int) * 3);
}
for(i = 0; i < num; i++) // 행의 크기만큼 반복
{
    for(j = 0; j < 3; j++) // 열의 크기만큼 반복
    {
        matrix_C[i][j] = 0; // 행렬을 0으로 초기화(쓰레기 값 처리 구문)
    }
}
```

1. 전치하고 싶은 희소행렬을 입력 받아 희소행렬의 개수 만큼 동적 메모리를 만든다.

```
for(i = 0; i < num; i++) // 행과 열을 바꿔주는 함수이다
{
    matrix_C[i][0] = matrix[i][1]; // 행을 열로
    matrix_C[i][1] = matrix[i][0]; // 열을 행으로
    matrix_C[i][2] = matrix[i][2]; // 값은 그대로
}
```

2. 전치 행렬은 행과 열의 값을 서로 바꿔주는 것이기 때문에 C 매트릭스에 바꿔서 대입해준다. 단 값은 그대로이다,

## Mul\_Check 함수

```
// -----
matrix_C = (int**) malloc(sizeof(int *) * num_B);
for(i = 0; i < num_B; i++)
{
    matrix_C[i] = (int*) malloc(sizeof(int) * 3);
}
for(i = 0; i < num_B; i++) // 행의 크기만큼 반복
{
    for(j = 0; j < 3; j++) // 열의 크기만큼 반복
    {
        matrix_C[i][j] = 0; // 행렬을 0으로 초기화(쓰레기 값 처리 구문)
    }
}
for(i = 0; i < num_B; i++) // 전치 행렬 부분
{
    matrix_C[i][0] = matrix_B[i][1];
    matrix_C[i][1] = matrix_B[i][0];
    matrix_C[i][2] = matrix_B[i][2];
}
```

1. C 매트릭스에 전치된 B행렬을 대입해주는 구문을 작성한다. 이유는 행렬의 곱

을 전치해서 계산하면 훨씬 계산하기 수월해지기 때문이다. 그러므로 B행렬을 전치하는 구문을 작성한다.

```
for(i = 0; i < num_B-1; i++)// 행의 숫자대로 정렬했으니 다음으로는 열의 크기대로 정렬한다.
{
    for(j = 0; j < num_B-1-i; j++)
    {
        if(matrix_C[j][0] == matrix_C[j+1][0])// 행의 크기가 같을경우
        {
            if(matrix_C[j][1] > matrix_C[j+1][1])// 열의 크기를 비교해 클경우 위치를 바꿔준다.
            {
                save_row = matrix_C[j][0];
                save_col = matrix_C[j][1];
                save_val = matrix_C[j][2];
                matrix_C[j][0] = matrix_C[j+1][0];
                matrix_C[j][1] = matrix_C[j+1][1];
                matrix_C[j][2] = matrix_C[j+1][2];
                matrix_C[j+1][0] = save_row;
                matrix_C[j+1][1] = save_col;
                matrix_C[j+1][2] = save_val;
            }
        }
    }
}
// -----
for(i = 0; i < num_B-1; i++)//행렬의 라인수 - 1 만큼 반복한다
{
    for(j = 0; j < num_B-1-i; j++)// 행렬의 라인수 - 1 - 1값만큼 반복한다
    {
        if(matrix_C[j][0] > matrix_C[j+1][0]) // 만약 행의 값이 클경우 위치를 바꿔준다.
        {
            save_row = matrix_C[j][0];
            save_col = matrix_C[j][1];
            save_val = matrix_C[j][2];
            matrix_C[j][0] = matrix_C[j+1][0];
            matrix_C[j][1] = matrix_C[j+1][1];
            matrix_C[j][2] = matrix_C[j+1][2];
            matrix_C[j+1][0] = save_row;
            matrix_C[j+1][1] = save_col;
            matrix_C[j+1][2] = save_val;
        }
    }
}
```

2. 전치된 행렬을 오름차순으로 정렬한다.

```

...
for(i=0; i<num_A; )//A 라인을 기준으로 출발한다
{
    row = matrix_A[i][0]; // 기본 행을 설정한다.
    for(j=0; j<num_B; )//같은 B 라인을 기준으로 한다
    {
        col = matrix_C[j][0]; //기본 열을 설정한다.
        sum=0;
        for(n=0, m=0; n < num_A && m < num_B && matrix_A[n][0] <= row && matrix_C[m][0] <= col; ) // 기본 행과 열을 비교하면서 동적 할당된 부분을 넘어가지 않게 막아주는 반복문이다.
        {
            if (matrix_A[n][0] != row) // 만약 행의 값이 다를경우 n값을 증가시킨다
                n++;
            if(matrix_C[m][0] != col) // 기본 열의 값이 다를경우 m값을 증가시킨다
                m++;
            if (matrix_A[n][0] == row && matrix_C[m][0] == col) // 두개의 행과 열이 동시에 같을 경우 진행한다
            {
                if (matrix_A[n][1] < matrix_C[m][1]) // A 열이 C열보다 작을경우 n값을 증가한다
                    n++;
                else if (matrix_A[n][1] > matrix_C[m][1]) //반대로 A열이 클경우 m 값을 증가한다.
                    m++;
                else
                {
                    sum = sum + (matrix_A[n][2] * matrix_C[m][2]); // 둘다 아닐경우 같다는 의미로 sum 값에 값을 곱해준다.
                    n++;
                    m++;
                }
            }
        }
    }
}

```

3. 조건문을 통해 기본 행과 열을 잡는다.

4. 기본 행과 열을 비교하면서 동적 할당된 메모리 그 이상을 건드리지 않게 조절하는 반복문을 적어둔다.

5. 크기를 비교하면서 크기가 같으면 둘의 합을 저장한다.

```

if(sum!=0) // sum이 0이 아니면 num 값을 증가시켜 행렬의 곱의 갯수를 증가시켜준다.
{
    sum = 0;
    num++;
}

```

6. 둘의 곱이 0이 아닐경우 num 값을 증가시킨다

7 증가된 num 값은 추후 행렬의 곱의 동적 메모리의 개수로 활용이 된다.

## Mul 함수

```

...
if(col_A != row_B) // 행렬의 곱의 조건에 맞춰 A의 열과 B의 행이 다르면 계산이 불가능하다.
{
    printf("==== A * B 행렬 ==== \n\n\n");
    printf("  A * B는 불가능 합니다  \n");
    printf("\n\n");
}

```

1. 행렬의 곱을 하기 위해서 조건문을 통해 곱이 연산이 가능한지 불가능 한지 판단을 한다.

```

num_D = sparse_matrix_mul_check(matrix_A, num_A, matrix_B, num_B); // 행렬의 곱을 저장하기 위해 이중 동적 행렬의 개수를 정해줘야 하기 때문에 호출한다.
matrix_C = (int**) malloc(sizeof(int *) * num_B); // 전치행렬로 곱을 처리하는 것이 편해 전치행렬 matrix_C 를 만들어준다.
for(i = 0; i < num_B; i++) // B의 전치행렬이기에 B라인 으로 계산한다.
{
    matrix_C[i] = (int*) malloc(sizeof(int ) * 3);
}
for(i = 0; i < num_B; i++) // 행의 크기만큼 반복
{
    for(j = 0; j < 3; j++) // 열의 크기만큼 반복
    {
        matrix_C[i][j] = 0; // 행렬을 0으로 초기화(쓰레기 값 처리 구문)
    }
}
for(i = 0; i < num_B; i++)
{
    matrix_C[i][0] = matrix_B[i][1];
    matrix_C[i][1] = matrix_B[i][0];
    matrix_C[i][2] = matrix_B[i][2];
}
sparse_matrix_sort(num_B, matrix_C);

```

2. matrix\_C에 B를 전치를 해서 집어 넣고 정렬을 진행한다.

3. 아까 위에서 구한 행렬의 곱의 개수 만큼 동적 메모리를 할당한다.

```

for(i=0; i < num_A; )//A 라인을 기준으로 출발한다
{
    row = matrix_A[i][0]; // 기본 행을 설정한다.
    for(j=0; j < num_B; ) //j값은 B 라인을 기준으로 한다
    {
        col = matrix_C[j][0]; //기본 열을 설정한다.
        sum=0;

        for(n=0, m=0; n < num_A && m < num_B && matrix_A[n][0] <= row && matrix_C[m][0] <= col; ) // 기본 행과 열을 비교하면서 동적 할당된 부분을 넘어가지 않게 막아주는 반복문이다.
        {
            if (matrix_A[n][0] != row) // 만약 행의 값이 다를경우 n값을 증가시킨다
                n++;
            if(matrix_C[m][0] != col) // 기본 열의 값이 다를경우 m값을 증가시킨다
                m++;
            if (matrix_A[n][0] == row && matrix_C[m][0] == col)// 두개의 행과 열이 동시에 같을 경우 진행한다
            {
                if (matrix_A[n][1] < matrix_C[m][1]) // A 열이 C열보다 작을경우 n값을 증가한다
                    n++;
                else if (matrix_A[n][1] > matrix_C[m][1]) //반대로 A열이 클경우 m 값을 증가한다.
                    m++;
                else
                {
                    sum = sum + (matrix_A[n][2] * matrix_C[m][2]); // 둘다 아닐경우 같다는 의미로 sum 값에 값을 곱해준다.
                    n++;
                    m++;
                }
            }
        }
        // 이 부분에서 sum의 값을 matrix_C[i][j]에 저장한다.
        matrix_C[i][j] = sum;
        i++;
        j++;
    }
}

```

4. 조건문을 통해 기본 행과 열을 잡는다.

5. 기본 행과 열을 비교하면서 동적 할당된 메모리 그 이상을 건드리지 않게 조절하는 반복문을 적어둔다.

6. 크기를 비교하면서 크기가 같으면 둘의 곱을 저장한다.

```

{
    if(sum!=0) // sum이 0이 아니면 위에서 만들어 둔 최소행렬의 저장 공간에 행 열 값을 저장해 준다
    {
        matrix_D[k][0] = row;
        matrix_D[k][1] = col;
        matrix_D[k][2] = sum;
        sum = 0;
        k++;
    }
}

```

7. 둘의 곱이 0이 아닐 경우 기본 행과 열을 저장하고 계산한 값을 대입한다.

8. 반복문은 동적 메모리 이상을 건들지 않게 계속 반복되다가 종료한다.

## 1.4 실행 결과

```
data - Windows D C:\Windows\system32\cmd.exe
파일(F) 편집(E) 서
A 2 3
0 0 1
0 2 1
1 1 -1
1 2 2
B 3 2
0 1 1
1 0 1
1 1 1

===== A 행렬 =====
행 열 값
0 0 1
0 2 1
1 1 -1
1 2 2

===== B 행렬 =====
행 열 값
0 1 1
1 0 1
1 1 1

===== A + B 행렬 =====
A + B는 불가능 합니다

===== A - B 행렬 =====
A - B는 불가능 합니다

===== A*B 행렬 =====
행 열 값
0 1 1
1 0 -1
1 1 -1

===== A 전치 행렬 =====
행 열 값
0 0 1
2 0 1
1 1 -1
2 1 2

===== B 전치 행렬 =====
행 열 값
1 0 1
0 1 1
1 1 1

계속하려면 아무 키나 누르십시오 . . .

data - Windows D C:\Windows\system32\cmd.exe
파일(F) 편집(E) 서
A 2 3
0 0 1
0 2 1
1 1 -1
1 2 2
B 2 3
0 2 1
1 0 1
1 1 1

===== A 행렬 =====
행 열 값
0 0 1
0 2 1
1 1 -1
1 2 2

===== B 행렬 =====
행 열 값
0 2 1
1 0 1
1 1 1

===== A+B 행렬 =====
행 열 값
0 0 1
0 2 2
1 0 1
1 2 2

===== A-B 행렬 =====
행 열 값
0 0 1
1 0 -1
1 1 -2
1 2 2

===== A * B 행렬 =====
A * B는 불가능 합니다

===== A 전치 행렬 =====
행 열 값
0 0 1
2 0 1
1 1 -1
2 1 2

===== B 전치 행렬 =====
행 열 값
2 0 1
0 1 1
1 1 1
```

## 1.5 느낀점

과제 3번과는 완전히 다른 느낌을 받았다. 우선 단순히 입력을 받는것도 동적 메모리를 이용해야하는 부분이 생기면서 임의로 동적 할당을 할 경우 낭비가 되거나 부족할 수 있었기에 여러가지 경우를 계속해서 고려를 해줘야 해서 중간중간 코드가 계속 오류가 나왔다. 특히 행렬의 연산 과정에서 나는 결과값을 따로 저장해주기 위해 새로운 이중 동적할당 변수를 만들어서 계산하려 했다. 하지만 덧셈을 같은 행과 열을 찾아 계산을 하게 되면 2 0 1 같은 혼자 남아있는 희소행렬은 0 이랑 합을 해줘야 하지만 그부분의 조건을 생각하느라 많이 힘들었다.

행렬의 곱도 마찬가지로 여러가지 경우가 많았고 특히 계속해서 희소행렬의 넘버를 바꿔주면서 계산을 하기가 많이 힘들었다.

교수님께서 배열의 크기를 추가해 주셔서 다행히 행의 제일 큰 부분과 열의 제일 큰 부분을 찾아 그것을 행과 열로 추가하는 복잡한 방법을 사용하지 않아 다행이었다. 이번 중간 프로젝트를 통해 희소행렬이 정말 좋은가에 대해 생각을 해봤는데 책에 나온대로 코드가 많이 복잡해지고 알고리즘도 더 복잡해지게 되었다. 하지만 그만큼 쓸모없는 부분이 사라지면서 메모리 낭비가 줄어들었다는 것은 정말 좋은 방법이다 라고 생각했다. 교수님이 수업시간에 말씀하신대로 코드가 길어져도 메모리 아끼는 것이 더 좋은 방법이다 라는 것을 다시한번 상기시킬 수 있는 기회가 되었다. 다음에는 코드를 조금 더 간략하고 조금 더 줄여 보는 도전을 해보고 싶다.