실습과제 2.1

[과제 설명]

희소행렬A의 전치행렬B를 구하는 알고리즘을 코드로 구현한다.
A의 0이 아닌 원소를 구조체 Element를 이용해서 배열로 main에 입력한다. 구조체 Element는 각각 행, 열, 값을 저장한다. 단, 0이 아닌 원소의 개수+1만큼 배열을 선언하는데, 첫 번째 행에 희소행렬의 크기와 0이 아닌 원소의 개수를 입력하기 위해서다.
또한, 다음 조건에 있는 두 함수를 구현해서 사용한다.

> Element* Transpose_Triple1(Element S_a[])
> Element의 값에서 전치행렬의 행 기준으로 원소를 순차적으로 찾아서 행과 열의 위치를 바꿔주는 함수
> void Print_Sparse_Mat (Element arr[])
> 구조체 Element에 입력된 데이터를 이용해 완전한 행렬을 출력해주는 함수

첫 번째 함수에서는 반환 타입이 정해져 있기 때문에 main에서 새로운 배열로 리턴값을 받아줘야 한다. 하지만, Transpose_Triple1에서 새롭게 만든 S_b를 리턴하려고 해도, 배열이기 때문에 주소만 넘겨주고 값은 사라지게 된다.
이것을 해결하기 위해서 해당하는 함수가 종료되어도 값이 유지되게 만들어야 하는데, 그 방법에는 새로운 값을 저장할 S_b를 static 변수로 선언하거나, 동적 할당을 하면 된다. 후술할 코드에서는 malloc을 이용한 동적 할당을 하는 방법으로 해결했다.
두 번째 함수는 미리 제시된 코드를 보고 만든 것은 아니기 때문에 형태가 다소 다르다. 값을 받아와서 새로운 변수에 저장하지 않은 차이가 있다. 중첩 반복문을 통해서 돌리면서, 행과 열이 일치할 때 넘겨받은 배열에 값이 있다면 출력하고, 없다면 0을 출력하는 방식이다.

[코드]

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

#define MAX_ELEMENTS 8
//0이 아닌 원소의 개수

typedef struct Element
{
  int row;
```

```
  int col;
  int value; //열, 행, 값
}Element;

Element* Transpose_Triple1(Element S_a[]) {
  //static Element S_b[MAX_ELEMENTS];
  //동적 할당 대신 static 변수를 만들 때
  Element* S_b = (Element*)malloc(sizeof(Element) * MAX_ELEMENTS);
  //함수가 끝나고도 유지되도록 동적 할당을 이용

  int num = S_a[0].value; //희소 행렬 중 0이 아닌 값의 개수
  S_b[0].row = S_a[0].col;
  S_b[0].col = S_a[0].row; //행과 열의 크기를 바꿔서 대입
  S_b[0].value = num;

  if (num > 0) { //0이 아닌 값의 개수가 1개라도 있을 때 실행
    int cnt = 1;

    for (int i = 0; i < num; i++) {
      for (int j = 1; j <= num; j++) {
        if (S_a[j].col == i) {
          S_b[cnt].row = S_a[j].col;
          S_b[cnt].col = S_a[j].row;
          S_b[cnt].value = S_a[j].value;
          cnt++;
        }
      }
    }
    //중첩 반복문을 이용해 오름차순으로 전치된 새로운 값을 대입
    //한 번 탐색하고 나면 cnt를 1 증가시켜서 다음 요소에 대입할 수 있게 한다.
  }

  return S_b;
  //배열을 넘기는 것이기 때문에 값을 넘기는 것이 아니라 시작 주소를 넘기는 것이다.
}

void Print_Sparse_Mat(Element arr[]) {
  int cnt = 1;

  for (int i = 0; i < arr[0].row; i++) {
    for (int j = 0; j < arr[0].col; j++) {
      if (i == arr[cnt].row && j == arr[cnt].col){
        printf("%d ", arr[cnt].value);
        cnt++;
      }
      else
        printf("0 "); //해당하는 값이 없으면 0을 대신 출력한다.
    }
    puts(""); //행이 바뀔 때마다 줄바꿈
  }
}

int main(void) {
  Element Sparse_A[MAX_ELEMENTS] = { {6,6,7},
    {0,2,6},
    {1,0,5},
    {1,4,7},
    {2,3,3},
```

```
    {4,0,8},
    {4,1,9},
    {5,3,2} }; //원본 행렬의 0이 아닌 원소의 행과 열, 값의 정보

  Print_Sparse_Mat(Sparse_A); //원본 행렬 출력
  puts(""); //줄바꿈
  Element* Sparse_B = Transpose_Triple1(Sparse_A); //전치행렬로 데이터를 변환
  Print_Sparse_Mat(Sparse_B); //전치행렬 출력
}
```

[과제에 대한 고찰]

희소 행렬은 행렬의 대부분이 0인 행렬이라, 0인 부분까지 모두 배열 안에 저장하면 메모리가 많이 들어서 0이 아닌 원소들만 따로 행, 열, 값으로 구조체를 이용해서 따로 저장하는 방법을 앞서 사용했다.

하지만 이렇게 하면 한 원소를 저장할 때 정수형 3개를 사용하는 형태다. 실습 예제에 제시된 36개 중 7개만 들어가도, 행렬의 크기와 0이 아닌 원소의 값을 저장하는 0행까지 8개, 즉 24개의 정수형을 쓴 격이 된다. 그렇다면, 원소가 전부 int를 이용해 저장하는 것을 가정하면, 행렬의 크기에 비해서 얼마나 원소가 적어야 공간적 측면에서 더 효율적일지도 생각해보았다.

가로의 크기를 a, 세로의 크기를 b, 0이 아닌 원소의 개수가 n이라고 할 때, a x b가 3(n+1)보다 커야 공간적인 효율성이 난다고 볼 수 있다.

처음에 공간적 효율과 시간적 효율을 고려하지 않았을 때는 비교적 코드가 간단하고 이해하기 쉬웠는데, 공간적 효율만 고려해도 그렇지 않을 때와 비교해서 코드가 어려워진 것을 체감하게 되었다. 게다가 경우에 따라서 오히려 메모리의 사용을 더 할 수도 있는 알고리즘인데, 다양한 경우에 이렇게 공간을 고려한 알고리즘을 쓰는 게 좋을 때가 많을 것이라는 생각도 했다.

이렇기 때문에 Big-O와 Omega의 예시 그래프에, 일정 기준 이전에는 Omega 그래프가 오히려 시간을 더 많이 쓰는 것처럼 최대값 라인보다 위에 있는 구간도 있는 것이 아닐까 하는 추측도 하게 되었다.