

알고리즘 2021 보고서

보고서 제출서약서

나는 송실대학교 컴퓨터학부의 일원으로 명예를 지키면서 생활하고 있습니다.

나는 보고서를 작성하면서 다음과 같은 사항을 준수하였음을 엄숙히 서약합니다.

1. 나는 자력으로 보고서를 작성하였습니다.
 - 1.1. 나는 동료의 보고서를 베끼지 않았습니다.
 - 1.2. 나는 비공식적으로 얻은 해답/해설을 기초로 보고서를 작성하지 않았습니다.
2. 나는 보고서에서 참조한 문헌의 출처를 밝혔으며 표절하지 않았습니다. (나는 특히 인터넷에서 다운로드한 내용을 보고서에 거의 그대로 복사하여 사용하지 않았습니다.)
3. 나는 보고서를 제출하기 전에 동료에게 보여주지 않았습니다.
4. 나는 보고서의 내용을 조작하거나 날조하지 않았습니다.

과목	알고리즘(나) 2021
과제명	SIC/XE 어셈블러 구현 (Project #1b)
담당교수	최 재 영 교 수
제출인	컴퓨터학부 20163340 강원경 (출석번호 201번)
제출일	2021년 09월 23일

차 례

1장 프로젝트 동기/목적 ----- 3p

2장 설계/구현 아이디어 ----- 3p

3장 문제 a 수행 결과 ----- 4p

4장 문제 b 수행 결과 ----- 8p

5장 문제 c, d 수행 결과 ----- 12p

6장 디버깅 ----- 13p

1장. 과제 동기/목적

이번 과제는 6가지 정렬 알고리즘을 직접 구현하고 각각의 알고리즘의 수행 과정을 살펴보며 속도를 비교해보는 과제이다. 컴퓨터를 전공하는 사람으로써 각각의 알고리즘을 이해하는 것은 어렵지 않았다. 하지만 그 이해한 내용을 코드로 풀어내는 것은 단순히 배우는 것과는 다른 영역이고, 나에게는 어려운 것 중 하나이다. 머릿속으로는 어떤 방식으로 동작을 해야하는지 알고 있더라도, 그걸 풀어서 코드화 한다는 것은 그만한 프로그래밍 실력과 알고리즘을 작성하는 논리력이 필요하기 때문이다. 따라서, 이번 과제를 통해 내가 이론으로 배운 내용을 나만의 방식으로 정리하고, 그 정리한 내용을 코드화 할 수 있는 논리력을 기르는 것을 첫 번째 목표로 삼았다.

또한, 평소같으면 라이브러리나 이미 짜여진 함수들을 그대로 가져다 쓰기 때문에 이러한 알고리즘들이 어떻게 동작하는지 크게 신경 쓰지 않으면서 코드를 작성했다. 그렇기 때문에 이번 과제를 통해 알고리즘을 직접 구현해봄으로써 평소에 라이브러리만 사용했다면 몰랐을 동작 방식에 대한 이해도를 높이는 것도 이번 과제를 통해 달성할 또 다른 목표로 삼았다.

2장. 설계/구현 아이디어

이번 과제에서는 각각의 문제들의 데이터가 서로 다른 방식으로 주어지기 때문에 각 정렬에 대한 주요 코드는 같더라도 각각의 문제에 맞도록 코드를 수정하였다. a번 문제의 경우는 고정된 정수형 데이터 8개가 주어지기 때문에 별도의 데이터 입력 과정 없이 이미 선언한 int형 배열에 주어진 데이터를 넣고 그 데이터를 정렬하여 결과를 출력하는 형식으로 하였다.

b번 문제의 경우는 정렬 알고리즘을 실행하기 전 시드값이 고정된 8개의 랜덤 데이터를 생성하였고, 그 데이터의 내용을 정렬 알고리즘 실행 전 한 번 출력한다. 데이터가 실수형이기 때문에 정수형 데이터를 다루는 a번 문제와는 달리 코드의 수정이 필요했고, 따라서 실수형 데이터를 취급할 수 있도록 코드를 수정하여 별도의 폴더에 별도의 코드를 저장하였다.

c번 문제는 앞선 문제에서의 랜덤 데이터를 천 개 단위로 생성하여 정렬을 수행하는 문제다. 그러면서 20000개 까지 각각의 데이터 개수에 따른 코드 실행시간을 알아야 한다. 조금 더 깔끔하고 보기 좋게 작성한다면 한번의 코드 실행으로 1000~20000개의 데이터를 자동으로 생성하여 모든 결과를 출력하는 방식을 취할 수도 있겠지만, 과제를 수행하면서 각각의 초기 데이터를 출력하고 그 데이터들의 연산 결과가 정상적으로 이뤄지는지 확인하고 싶었다. 따라서 본인이 작성한 프로그램의 경우에는 코드 실행의 맨 처음에 랜덤 생성할 데이터 개수를 사용자가 입력하는 과정이 있으며, 그 개수를 입력하면 프로그램에서 자동으로 초기의 랜덤 데이터를 생성하고 화면에 출력한다. 그 후 정렬 연산을 수행한 후 정렬된 데이터와 그 데이터들을 생성하고 정렬하여 화면에 출력하기까지 걸린 시간을 측정한 후 마지막에 시간을 출력해주는 방식을 택했다.

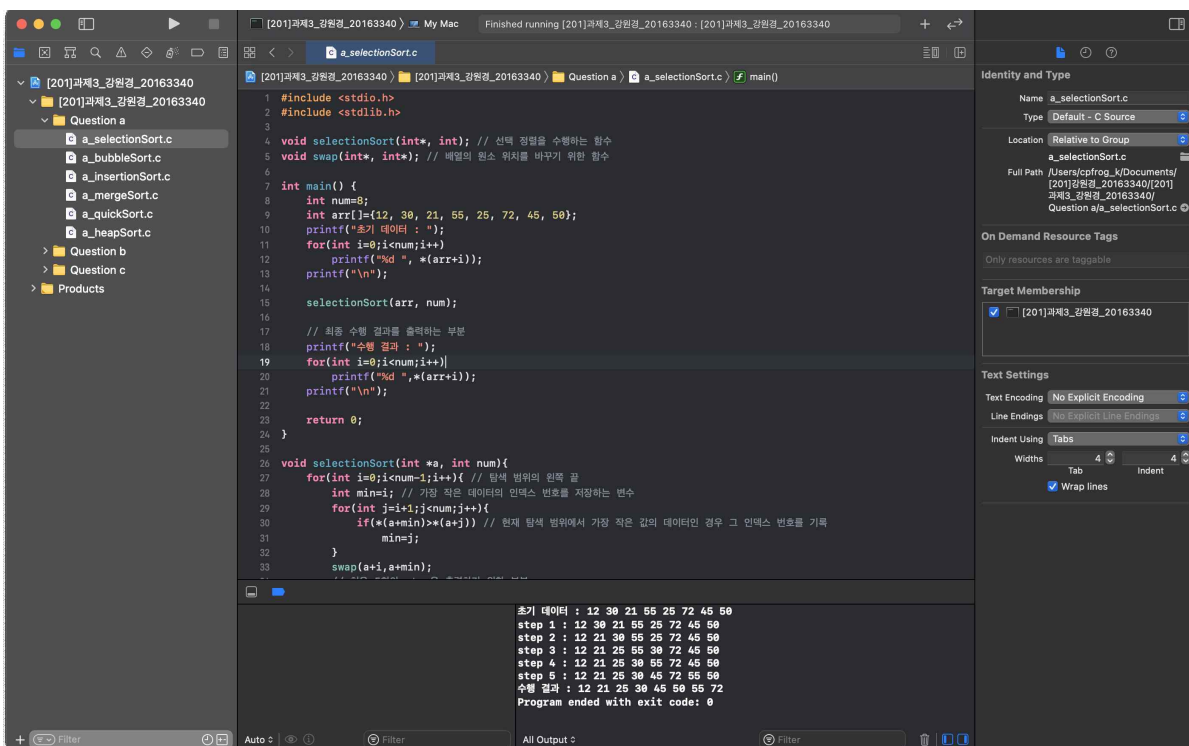
d번 문제는 c번 문제에서 각각의 실행을 통해 얻은 시간들을 표로 정리한 후, 그 표를 바탕으로 꺾은선 그래프로 표현하여 각 정렬 알고리즘과 데이터의 개수 별 시간의 차이가 파악하기 쉽도록 작성하였다.

이번 과제는 macOS 버전 11.6 (Big Sur)에서 Xcode 버전 12.5.1 버전을 이용하여 수행하였다.

3장. 문제 a 수행 결과

결과에 대한 내용을 보기에 앞서 본인의 코드 수행 결과에 대한 설명을 덧붙이고자 한다. 이 설명의 내용은 a번 문제뿐만 아니라 이후 문제들에서도 공통되는 내용이다. 과제 명세에서는 최종 결과에서 앞의 데이터 2개만을 출력하라고 명시되어있다는 것은 본인도 인지하고 있는 부분이지만, 앞의 두 데이터만 출력하는 것으로는 본인이 작성한 코드가 모든 데이터에 대해 정상적으로 수행되었는지 확인할 수 없었다. 따라서 a~c번 문제는 초기 데이터의 모든 내용을 1회 출력하고, 정렬이 완료된 후의 전체 데이터를 출력하는 것으로 하였다.

먼저 선택 정렬을 수행한 결과이다.



```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void selectionSort(int*, int); // 선택 정렬을 수행하는 함수
5 void swap(int*, int*); // 배열의 원소 위치를 바꾸기 위한 함수
6
7 int main() {
8     int num=8;
9     int arr[]={12, 30, 21, 55, 25, 72, 45, 50};
10    printf("초기 데이터 : ");
11    for(int i=0;i<num;i++)
12        printf("%d ", *(arr+i));
13    printf("\n");
14
15    selectionSort(arr, num);
16
17    // 최종 수행 결과를 출력하는 부분
18    printf("수행 결과 : ");
19    for(int i=0;i<num;i++)
20        printf("%d ", *(arr+i));
21    printf("\n");
22
23    return 0;
24 }
25
26 void selectionSort(int *a, int num){
27     for(int i=0;i<num-1;i++){ // 탐색 범위의 왼쪽 끝
28         int min=i; // 가장 작은 데이터의 인덱스 번호를 저장하는 변수
29         for(int j=i+1;j<num;j++){
30             if(*(a+min)>*(a+j)) // 현재 탐색 범위에서 가장 작은 값의 데이터인 경우 그 인덱스 번호를 기록
31                 min=j;
32         }
33         swap(a+i,a+min);
34     }
35 }
```

초기 데이터 : 12 30 21 55 25 72 45 50
step 1 : 12 30 21 55 25 72 45 50
step 2 : 12 21 30 55 25 72 45 50
step 3 : 12 21 25 55 30 72 45 50
step 4 : 12 21 25 30 55 72 45 50
step 5 : 12 21 25 30 45 72 55 50
수행 결과 : 12 21 25 30 45 50 55 72
Program ended with exit code: 0

결과 화면에 출력된 step 1~5를 보면 알 수 있듯, 일반적인 선택정렬과 같이 배열의 맨 앞에서부터 작은 데이터들이 순차적으로 정렬되어지고 있다.

다음은 버블 정렬을 수행한 결과이다.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void bubbleSort(int*, int); // 버블 정렬을 수행하는 함수
5 void swap(int*, int*); // 배열의 원소 위치를 바꾸기 위한 함수
6
7 int main() {
8     int num=8;
9     int arr[]={12, 30, 21, 55, 25, 72, 45, 50};
10    printf("초기 데이터 : ");
11    for(int i=0;i<num;i++)
12        printf("%d ", *(arr+i));
13    printf("\n");
14
15    bubbleSort(arr, num);
16
17    // 최종 수행 결과를 출력하는 부분
18    printf("수행 결과 : ");
19    for(int i=0;i<num;i++)
20        printf("%d ", *(arr+i));
21    printf("\n");
22
23    return 0;
24 }
25
26 void bubbleSort(int *arr, int num){
27     int step=1;
28     for(int j=num-1;j>0;j--){ // 탐색 범위의 오른쪽 끝
29         for(int i=0;i<j;i++){
30             if(*(arr+i)>*(arr+i+1)) // i번째와 i+1번째 원소를 비교하여 필요에 따라 위치를 교환
31                 swap((arr+i),(arr+i+1));
32         }
33         // 처음 5회의 step을 출력하기 위한 부분
34     }
35 }
36
37 void swap(int *a, int *b){
38     int temp=*a;
39     *a=*b;
40     *b=temp;
41 }

```

초기 데이터 : 12 30 21 55 25 72 45 50
step 1 : 12 21 30 25 55 45 50 72
step 2 : 12 21 25 30 45 50 55 72
step 3 : 12 21 25 30 45 50 55 72
step 4 : 12 21 25 30 45 50 55 72
step 5 : 12 21 25 30 45 50 55 72
수행 결과 : 12 21 25 30 45 50 55 72
Program ended with exit code: 0

이웃한 데이터의 크기를 비교하여 데이터를 정렬한다는 버블정렬의 정의에 부합하도록 코드를 작성하였고, 그 과정은 step 1~5에서 확인이 가능하다.

세 번째로 삽입정렬이다.

```

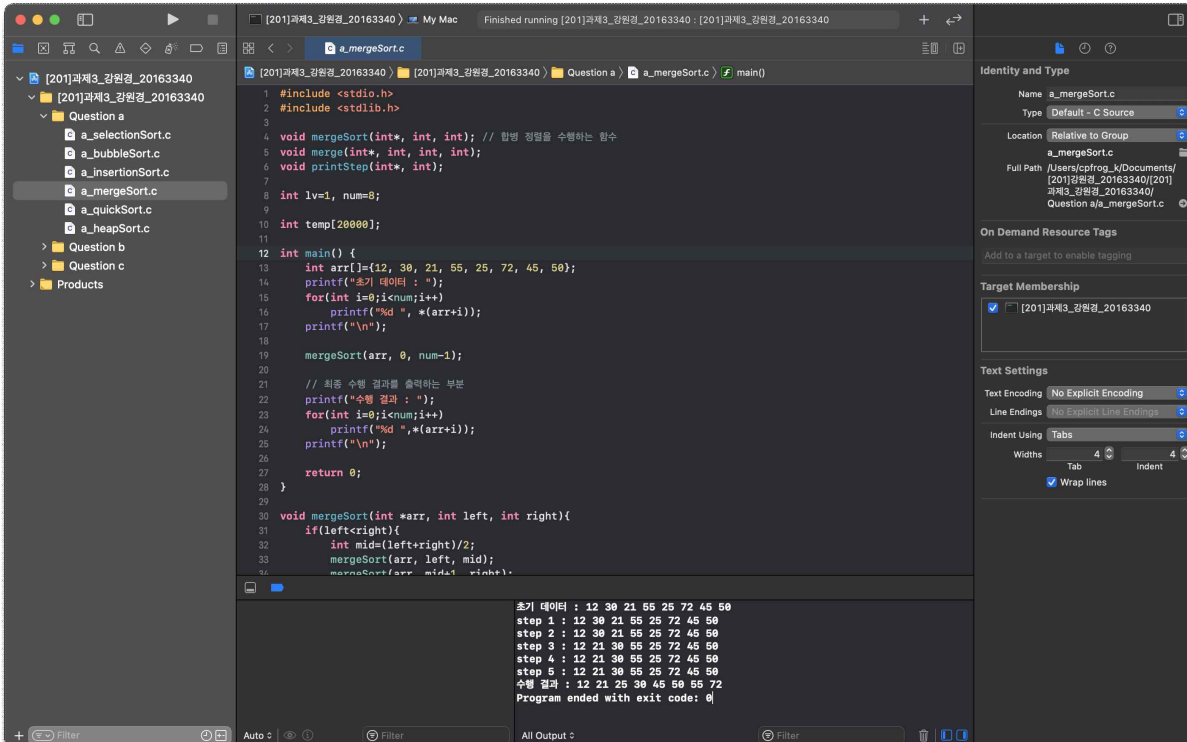
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void insertionSort(int*, int); // 삽입 정렬을 수행하는 함수
5
6 int main() {
7     int num=8;
8     int arr[]={12, 30, 21, 55, 25, 72, 45, 50};
9     printf("초기 데이터 : ");
10    for(int i=0;i<num;i++)
11        printf("%d ", *(arr+i));
12    printf("\n");
13
14    insertionSort(arr, num);
15
16    // 최종 수행 결과를 출력하는 부분
17    printf("수행 결과 : ");
18    for(int i=0;i<num;i++)
19        printf("%d ", *(arr+i));
20    printf("\n");
21
22    return 0;
23 }
24
25 void insertionSort(int *arr, int num){
26     for(int j=1;j<num;j++){ // 탐색 범위의 오른쪽 끝
27         for(int i=0;i<j;i++){
28             if(*(arr+i)>*(arr+j)) { // i번째와 j번째 원소를 비교하여 j번째 원소의 삽입 위치를 결정
29                 int temp=*(arr+j);
30                 for(int k=j;k>i;k--)
31                     *(arr+k)=*(arr+k-1);
32                 *(arr+i)=temp;
33             }
34         }
35     }
36 }

```

초기 데이터 : 12 30 21 55 25 72 45 50
step 1 : 12 30 21 55 25 72 45 50
step 2 : 12 21 30 55 25 72 45 50
step 3 : 12 21 30 55 25 72 45 50
step 4 : 12 21 25 30 55 72 45 50
step 5 : 12 21 25 30 55 72 45 50
수행 결과 : 12 21 25 30 45 50 55 72
Program ended with exit code: 0

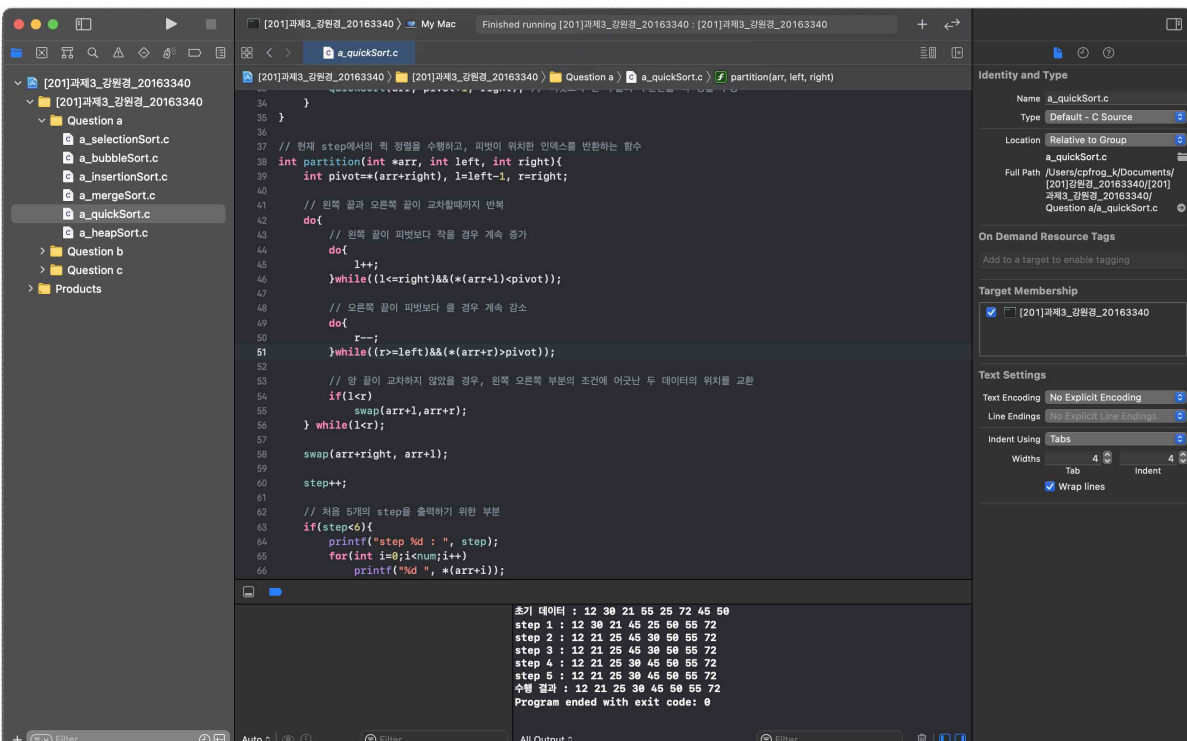
step이 늘어날수록 왼쪽의 데이터들부터 크기에 맞게 정렬이 이뤄지고 있음을 알 수 있다.

네 번째로 합병 정렬이다.



짝수번째 step에서는 데이터의 분할이 이뤄지기 때문에 정렬 내용에 변화가 없지만, 홀수번째 step으로 넘어가면서 데이터의 병합이 이뤄지기 때문에 데이터의 순서에 변화가 있음이 확인 가능하다.

다섯 번째, 퀵 정렬이다.



퀵 정렬은 각 partition의 맨 오른쪽 데이터를 피벗으로 잡고 수행하였다. step 1에서는 맨 오른쪽 데이터인 50이 피벗이므로 그에 맞게 정렬이 수행되었고, 이후 step에서도 각 피벗(25, 72)에 맞게 퀵 정렬이 반복해서 이루어져 최종적으로 정렬된 결과가 출력된다.

마지막으로 힙정렬이다.

The screenshot shows a C++ IDE with a project named "[201]과제3_강원경_20163340". The file explorer on the left shows a directory structure with files like "a_selectionSort.c", "a_bubbleSort.c", "a_insertionSort.c", "a_mergeSort.c", "a_quickSort.c", and "a_heapSort.c". The main editor displays the code for "a_heapSort.c", which implements a heap sort algorithm. The code includes headers, defines swap and heapify functions, and a main function that sorts an array of 10 numbers. The output window at the bottom shows the execution results, including the initial array, the state of the array at each step of the sorting process, and the final sorted array.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void heapSort(int*, int); // 재귀 호출 방식을 통해 힙 정렬을 수행하는 함수
5 void swap(int*, int*); // 배열의 원소 위치를 바꾸기 위한 함수
6 void heapify(int*, int, int);
7
8 int num=8, step=0;
9
10 int main() {
11     int arr[]={12, 30, 21, 55, 25, 72, 45, 50};
12     printf("초기 데이터 : ");
13     for(int i=0;i<num;i++)
14         printf("%d ", *(arr+i));
15     printf("\n");
16
17     // leaf node가 아닌것부터 시작
18     for (int i=num-1;i>=0;i--)
19         heapify(arr,num,i);
20
21     heapSort(arr,num);
22
23     // 최종 수행 결과를 출력하는 부분
24     printf("수행 결과 : ");
25     for(int i=0;i<num;i++)
26         printf("%d ", *(arr+i));
27     printf("\n");
28
29     return 0;
30 }
31
32 // 힙 자료구조의 성질을 유지할 수 있도록 하는 함수
33 void heapify(int *arr, int num, int pos){
34     // ...
35 }
```

초기 데이터 : 12 30 21 55 25 72 45 50
step 1 : 55 50 45 30 25 21 12 72
step 2 : 50 30 45 12 25 21 55 72
step 3 : 45 30 21 12 25 50 55 72
step 4 : 30 25 21 12 45 50 55 72
step 5 : 25 12 21 30 45 50 55 72
수행 결과 : 12 21 25 30 45 50 55 72
Program ended with exit code: 0

오름 차순으로 정렬하기 위해 최대 힙 방식을 이용한 힙 정렬을 사용하였고, 그에 맞게 맨 오른쪽부터 가장 큰 데이터가 순서대로 정렬되어짐을 알 수 있다.

4장. 문제 b 수행 결과

문제 b는 -1 ~ 1 사이의 데이터를 무작위로 8개 추출하기 때문에 실수형 데이터를 취급해야 하고, 이에 따라 정수형 데이터를 다뤘던 문제 a와는 달리 실수형 데이터를 다룰 수 있는 별도의 수정된 코드가 필요하다고 판단했다. 따라서 문제 a와는 다른 소스코드 파일을 생성하여 문제 b를 해결할 수 있는 코드를 작성하였다.

```
double arr[8]; // -1 ~ 1 사이의 실수 8개가 저장될 배열
printf("랜덤한 숫자 8개 : ");
for(int i=0;i<8;i++){
    srand(5+i); // 5부터 시작하는 시드값으로 고정시키기 위한 코드
    arr[i]=(5-(double)(rand()%1000)/100)/10; // -1~1 사이의 실수가 나올 수 있도록 한 부분
    printf("%.31f ", arr[i]);
}
printf("\n");
```

위 코드와 같이 double형 데이터 8개를 무작위로 생성하였고, 기존에 작성된 정렬 함수들 또한 double형 데이터를 다룰 수 있도록 수정하였다.

선택 정렬을 예시로 하여 기존 a번 문제의 함수와 어떻게 달라졌는지 살펴보기로 한다.

```
void selectionSort(double *a, int num){
    for(int i=0;i<num-1;i++){ // 탐색 범위의 왼쪽 끝
        int min=i; // 가장 작은 데이터의 인덱스 번호를 저장하는 변수
        for(int j=i+1;j<num;j++){
            if(0>*(a+j)-*(a+min)) // 현재 탐색 범위에서 가장 작은 값의 데이터인 경우 그 인덱스 번호를 기록
                min=j;
        }
        swap(a+i, a+min);
    }
}
```

우선 함수의 argument로 int형 배열이 아닌 double형 배열을 받아오는 것이 필요하였고, 데이터의 대소비교의 경우에도 실수형 데이터의 구조상 논리 연산자(< , >)를 사용하면 의도치 않은 오류를 발생시킬 수 있기 때문에 밑에서 두 번째 줄의 if문과 같이 크기를 비교하려고 하는 데이터를 서로 빼서 음수인지 양수인지의 여부로 크기 비교를 수행하였다.

위와 같이 초기 데이터의 생성, 그리고 데이터 형태의 차이로 인한 코드 수정 이외에는 주요한 정렬 알고리즘 동작 방식은 동일하기 때문에 문제 b번의 수행 결과는 코드를 생략하고 살펴보기로 한다.

우선 선택 정렬의 결과이다.

```
랜덤한 숫자 8개 : 0.465 -0.342 -0.149 0.044 0.237 0.430 -0.377 -0.184
step 1 : -0.377 -0.342 -0.149 0.044 0.237 0.430 0.465 -0.184
step 2 : -0.377 -0.342 -0.149 0.044 0.237 0.430 0.465 -0.184
step 3 : -0.377 -0.342 -0.184 0.044 0.237 0.430 0.465 -0.149
step 4 : -0.377 -0.342 -0.184 -0.149 0.237 0.430 0.465 0.044
step 5 : -0.377 -0.342 -0.184 -0.149 0.044 0.430 0.465 0.237
수행 결과 : -0.377 -0.342 -0.184 -0.149 0.044 0.237 0.430 0.465
Program ended with exit code: 0
```

최초로 생성된 8개의 데이터는 첫 번째 줄에 나와있고, 이 데이터는 시드값이 정해져 있는 값이기

때문에 이후에 수행될 정렬 함수의 초기 데이터에서도 동일하게 사용되어질 것이다.

마지막 줄의 수행 결과와 같이 데이터의 크기에 따라 정상적으로 정렬이 수행되었음을 확인할 수 있다. 또, step을 통해 살펴본 과정에서도 알 수 있듯이 선택정렬의 정의에 맞는 과정이 수행되어졌음이 확인 가능하다.

두 번째로 버블 정렬의 결과를 보도록 한다.

```
랜덤한 숫자 8개 : 0.465 -0.342 -0.149 0.044 0.237 0.430 -0.377 -0.184
step 1 : -0.342 -0.149 0.044 0.237 0.430 -0.377 -0.184 0.465
step 2 : -0.342 -0.149 0.044 0.237 -0.377 -0.184 0.430 0.465
step 3 : -0.342 -0.149 0.044 -0.377 -0.184 0.237 0.430 0.465
step 4 : -0.342 -0.149 -0.377 -0.184 0.044 0.237 0.430 0.465
step 5 : -0.342 -0.377 -0.184 -0.149 0.044 0.237 0.430 0.465
수행 결과 : -0.377 -0.342 -0.184 -0.149 0.044 0.237 0.430 0.465
Program ended with exit code: 0
```

선택정렬과 동일한 초기 데이터가 사용됐음이 첫 번째 줄을 통해 확인할 수 있고, 이후의 정렬 과정은 버블 정렬에 맞게 수행되어져 최종적으로 같은 정렬 결과를 출력하였다.

다음은 삽입 정렬이다.

```
랜덤한 숫자 8개 : 0.465 -0.342 -0.149 0.044 0.237 0.430 -0.377 -0.184
step 1 : -0.342 0.465 -0.149 0.044 0.237 0.430 -0.377 -0.184
step 2 : -0.342 -0.149 0.465 0.044 0.237 0.430 -0.377 -0.184
step 3 : -0.342 -0.149 0.044 0.465 0.237 0.430 -0.377 -0.184
step 4 : -0.342 -0.149 0.044 0.237 0.465 0.430 -0.377 -0.184
step 5 : -0.342 -0.149 0.044 0.237 0.430 0.465 -0.377 -0.184
수행 결과 : -0.377 -0.342 -0.184 -0.149 0.044 0.237 0.430 0.465
Program ended with exit code: 0
```

삽입정렬도 마찬가지로 동일한 초기 데이터에 대해 정의에 부합하는 과정을 통해 정렬이 이루어졌음을 알 수 있다.

네 번째로 합병 정렬이다.

```
랜덤한 숫자 8개 : 0.465 -0.342 -0.149 0.044 0.237 0.430 -0.377 -0.184
step 1 : -0.342 0.465 -0.149 0.044 0.237 0.430 -0.377 -0.184
step 2 : -0.342 0.465 -0.149 0.044 0.237 0.430 -0.377 -0.184
step 3 : -0.342 -0.149 0.044 0.465 0.237 0.430 -0.377 -0.184
step 4 : -0.342 -0.149 0.044 0.465 0.237 0.430 -0.377 -0.184
step 5 : -0.342 -0.149 0.044 0.465 0.237 0.430 -0.377 -0.184
수행 결과 : -0.377 -0.342 -0.184 -0.149 0.044 0.237 0.430 0.465
Program ended with exit code: 0
```

첫 번째 step에서 왼쪽 2개의 데이터의 정렬이 이뤄졌고, 이후 step 3에서 3, 4번째 데이터에 대해 정렬이 수행되었다. 이러한 방식으로 데이터의 partition이 점점 커지면서 합병 정렬이 수행되어져 최종적으로는 이전 정렬 알고리즘들과 동일한 정렬 결과가 출력되었음을 확인할 수 있다.

다섯 번째로 퀵 정렬이다.

```
랜덤한 숫자 8개 : 0.465 -0.342 -0.149 0.044 0.237 0.430 -0.377 -0.184
step 1 : -0.377 -0.342 -0.184 0.044 0.237 0.430 0.465 -0.149
step 2 : -0.377 -0.342 -0.184 0.044 0.237 0.430 0.465 -0.149
step 3 : -0.377 -0.342 -0.184 -0.149 0.237 0.430 0.465 0.044
step 4 : -0.377 -0.342 -0.184 -0.149 0.044 0.430 0.465 0.237
step 5 : -0.377 -0.342 -0.184 -0.149 0.044 0.237 0.465 0.430
수행 결과 : -0.377 -0.342 -0.184 -0.149 0.044 0.237 0.430 0.465
Program ended with exit code: 0
```

All Output ↕

Filter

🗑️ | 📄 📄

step 1에서는 전체 데이터를 하나의 partiton으로 보고 맨 오른쪽 데이터인 -0.184를 피벗으로 하여 퀵 정렬이 수행되었고, 이후 피벗을 중심으로 partiton을 분할하여 퀵 정렬이 수행되었다. 최종 정렬 수행 결과는 이전과 동일하다.

마지막으로 힙 정렬의 수행 결과이다.

```
랜덤한 숫자 8개 : 0.465 -0.342 -0.149 0.044 0.237 0.430 -0.377 -0.184
step 1 : 0.430 0.237 -0.149 0.044 -0.342 -0.184 -0.377 0.465
step 2 : 0.237 0.044 -0.149 -0.377 -0.342 -0.184 0.430 0.465
step 3 : 0.044 -0.184 -0.149 -0.377 -0.342 0.237 0.430 0.465
step 4 : -0.149 -0.184 -0.342 -0.377 0.044 0.237 0.430 0.465
step 5 : -0.184 -0.377 -0.342 -0.149 0.044 0.237 0.430 0.465
수행 결과 : -0.377 -0.342 -0.184 -0.149 0.044 0.237 0.430 0.465
Program ended with exit code: 0
```

All Output ↕

Filter

🗑️ | 📄 📄

b번 문제의 힙 정렬도 a번 문제의 코드와 동일하게 max heap 방식을 사용하였기 때문에 오른쪽에서부터 큰 값의 데이터가 순서대로 정렬되기 시작해 최종적으로는 이전 정렬 결과와 동일한 결과가 출력되어졌음을 알 수 있다.

5장. 문제 c, d 수행 결과

문제 c는 이전 문제에서 데이터의 생성 개수가 달라지게 된다. 따라서 이에 대응할 수 있는 코드의 수정이 필요했고, 지난 문제의 해결 방식과 마찬가지로 별도의 소스코드 파일에 수정된 코드를 저장하여 문제의 조건을 만족할 수 있도록 하였다.

```
time_t start, end;
int num; // 생성할 데이터의 갯수가 저장될 변수
printf("랜덤으로 생성할 데이터 갯수를 입력하세요 : ");
scanf("%d", &num);
double arr[num]; // -1 ~ 1 사이의 실수가 저장될 배열
start=clock();
printf("초기 데이터 : ");
for(int i=0;i<num;i++){
    srand(5+i); // 5부터 시작하는 시드값으로 고정시키기 위한 코드
    arr[i]=((double)(rand()%1000)/100)/10; // -1~1 사이의 실수가 나올 수 있도록 한 부분
    if(rand()%2==0) arr[i]*=-1;
    printf("%.3lf ", arr[i]);
}
printf("\n");
```

위 사진의 코드는 사용자로부터 랜덤으로 생성할 데이터의 개수를 입력받고, 랜덤 함수를 이용해 사용자가 요구하는 개수 만큼 랜덤 데이터를 생성해 정렬을 수행한다.

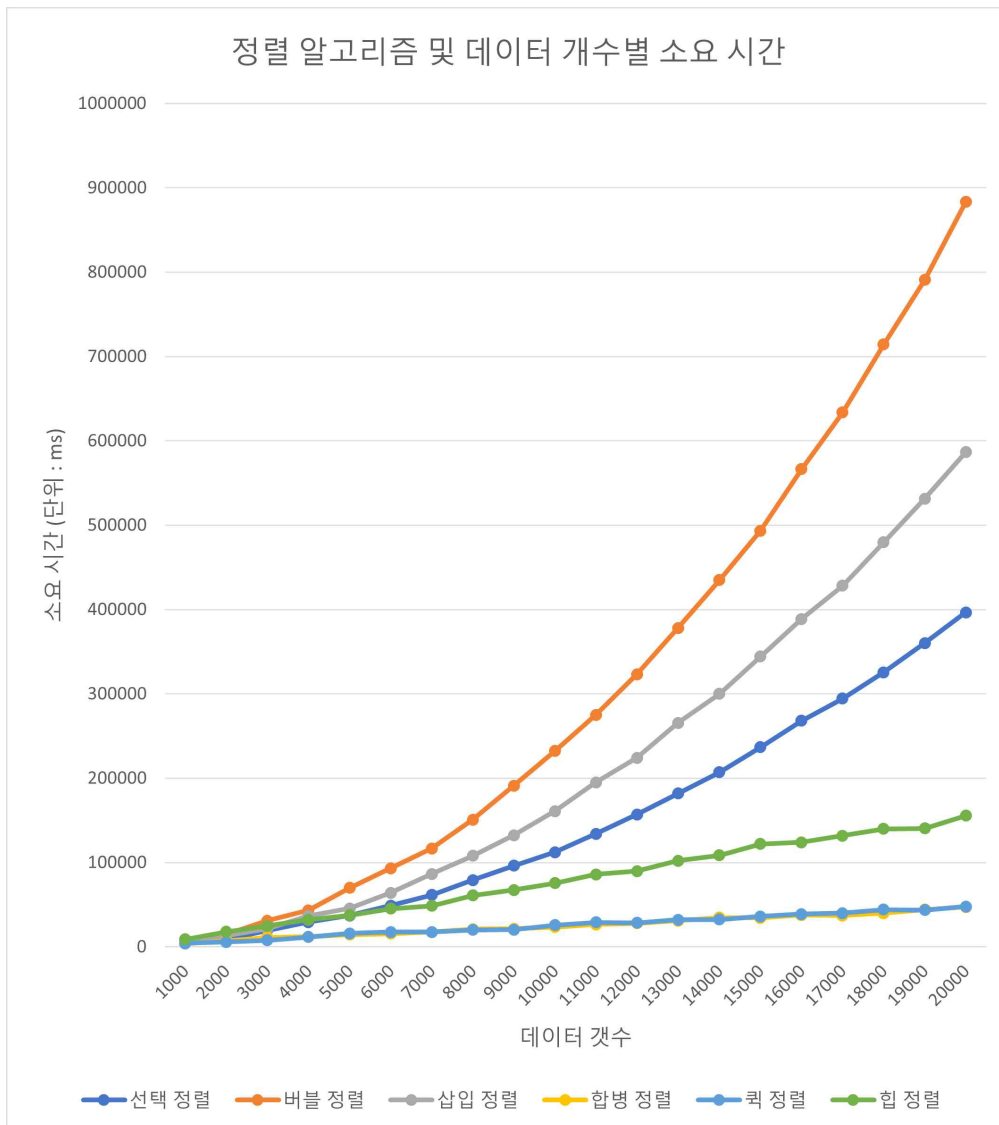
본인이 작성한 코드에서의 수행시간 기준은 "랜덤 데이터를 생성하기 시작한 시점에서부터 정렬된 데이터를 모두 출력하기까지의 시간" 으로 하였고, 이 조건은 각각의 정렬 알고리즘의 수행시간 측정에 동일하게 적용되었음을 밝힌다.

문제 c의 수행결과를 수행된 결과표를 통해 확인하도록 한다.

	선택 정렬	버블 정렬	삽입 정렬	합병 정렬	퀵 정렬	힙 정렬
1000	5401	6826	6329	4006	4316	8944
2000	8742	14780	13091	7315	5780	17854
3000	19614	30981	21601	11134	7917	25016
4000	29451	43317	36807	12113	11749	32301
5000	37281	70010	45662	14690	15813	37453
6000	49102	93091	64135	15765	17756	45438
7000	61522	116800	86447	17535	17716	48736
8000	79153	150825	107946	20957	20205	61147
9000	96363	191156	132513	21502	20395	67498
10000	112112	232211	160877	23587	25797	75468
11000	133970	275005	195105	26455	29138	86016
12000	157175	323248	224355	27918	28456	89719
13000	182012	378078	265568	31147	32100	102327
14000	207153	435046	299970	34777	32386	108615
15000	236836	493432	344240	34590	36248	121936
16000	268146	566501	388678	37846	38841	123942
17000	294554	633720	428337	37297	40140	131889
18000	325659	714238	479768	40093	44333	139972
19000	360213	790861	531498	44171	43757	140602
20000	396699	883435	586806	47244	47780	155646

표에서 확인할 수 있듯, $n \log n$ 의 복잡도를 가지는 병합 정렬, 퀵 정렬, 힙 정렬은 상승폭이 비교적 크지 않은 반면에 n^2 의 복잡도를 가지는 선택 정렬, 버블 정렬, 삽입 정렬은 상승폭이 매우 크다는 것을 확인할 수 있다.

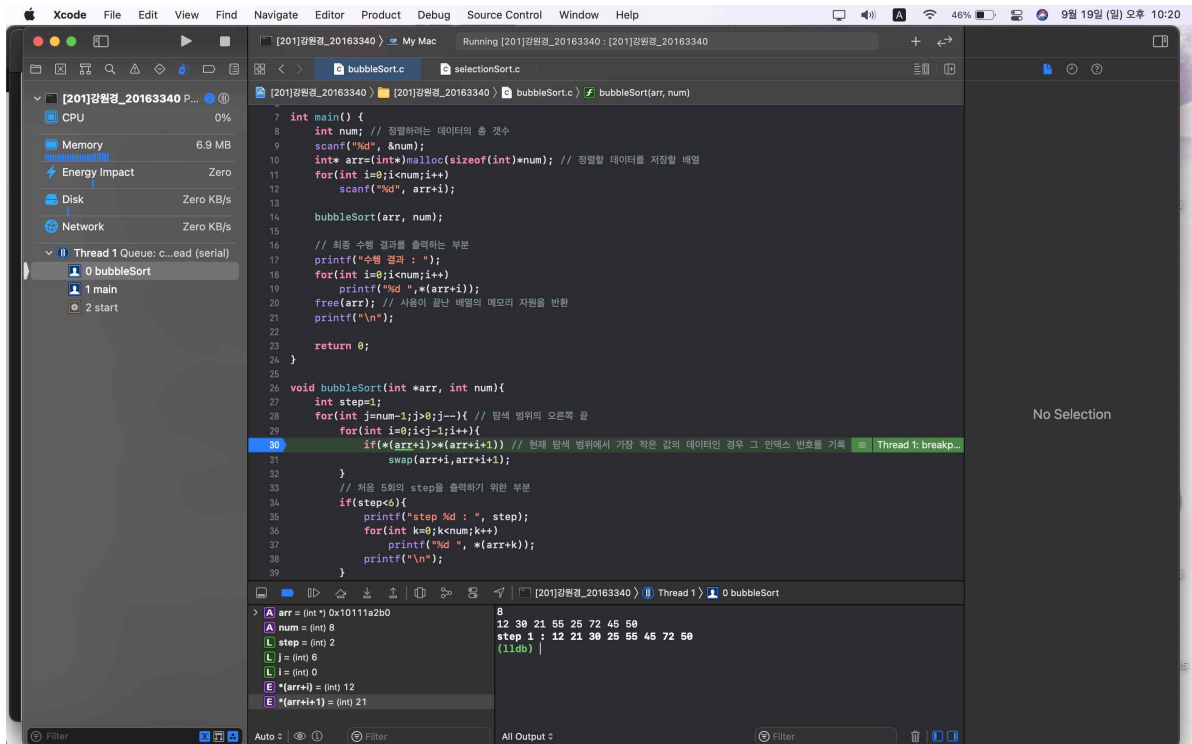
단순히 표의 숫자로 확인하기에는 한번에 확인하기 어려우므로 시각화된 자료를 통해 다시 한 번 살펴보도록 하겠다. (문제 d)



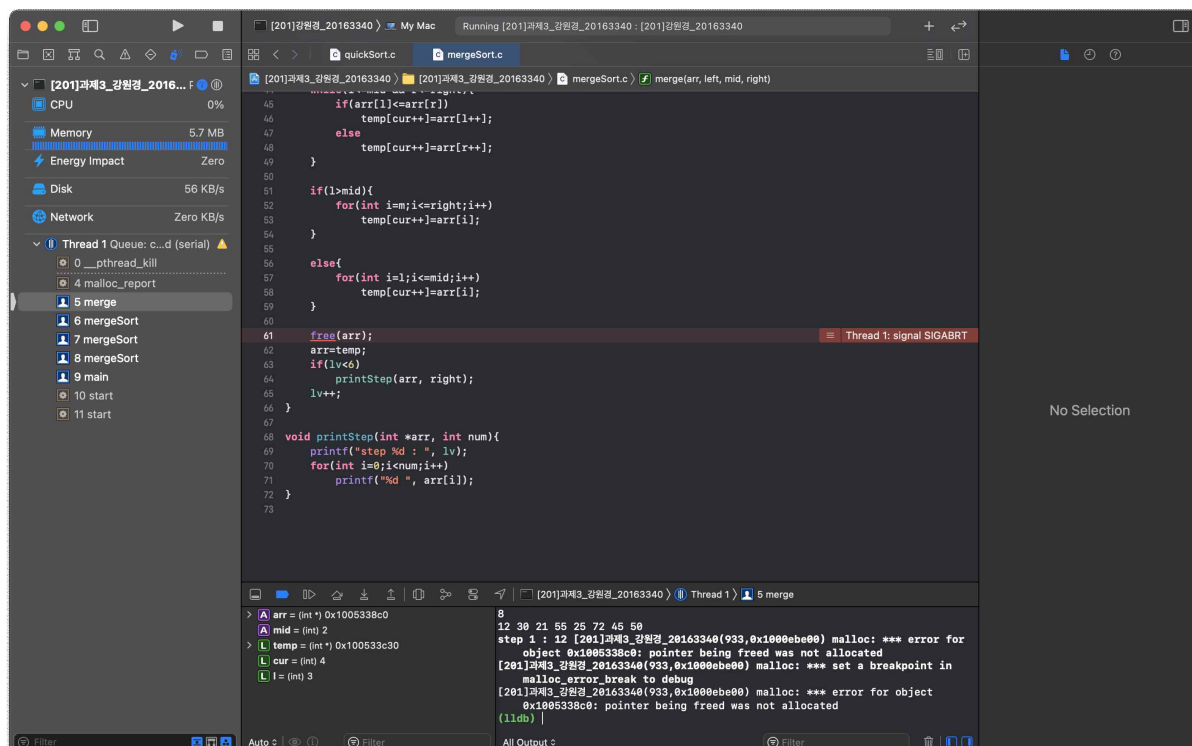
그래프를 통해 확인해보면, 합병 정렬과 퀵 정렬은 수행 시간에 있어서 차이가 그리 크지 않았고, 힙 정렬은 두 정렬 방식보다는 조금 시간이 소요되기는 했으나, n^2 의 복잡도를 가지는 정렬 알고리즘에 비해서는 빠르게 결과를 출력하였다.

6장. 디버깅

이번 과제에서도 코드를 작성하는 중에 발생하는 오류는 디버거를 이용하여 수정을 진행하였다. 여러 과정에서 활용을 하였으나, 두 가지 예시만 살펴보도록 하겠다.



디버거를 활용한 첫 번째 방법은 중단점이다. 버블 정렬 코드를 작성하는 중 step에서의 과정이 본인이 알고 있는 과정과는 다르게 출력되어 어떤 방식으로 동작되고 있었는지를 확인할 필요가 있었다. 따라서 반복문 위치에 중단점을 설정하고 한 단계씩 수행하여 코드 수행과정을 살펴보았다.



두 번째로 활용한 방법은 오류 코드의 검출이다. 합병 정렬을 수행하는 중 갑자기 실행중에 프로세스가 kill이 되는 상황이 발생했다. 디버거를 통해 살펴본 결과 malloc 방식으로 배열을 생성하던 코드에서 일반적인 선언문으로 방식을 변경하는 과정에서 완벽하게 이전 코드를 제거하지 않고 수행하던 중 발생한 문제였다.