



ICE2004 자료구조론

9

제 목

자료구조론 과제 4 보고서

보고서 작성 서약서

1. 나는 타학생의 보고서를 베끼거나 여러 보고서의 내용을 짜집기하지 않겠습니다.
2. 나는 보고서의 주요 내용을 인터넷사이트 등을 통해 얻지 않겠습니다.
3. 나는 보고서의 내용을 조작하지 않겠습니다.
4. 나는 보고서 작성에 참고한 문헌의 출처를 밝히겠습니다.
5. 나는 나의 보고서를 제출 전에 타학생에게 보여주지 않겠습니다.

나는 보고서 작성시 윤리에 어긋난 행동을 하지 않고 정보통신공학인으로서 나의 명예를 지킬 것을 맹세합니다.

2021년 11월 21일

학부

학년

성명

성시열

학번

[개요 및 구현상 특징]

- 파일은 `sorting.h` 헤더파일과 `keydata.h` 헤더파일. 총 두 개의 헤더파일과 이 헤더파일의 기능을 활용할 수 있는 `main.cpp` 드라이버 파일로 구성되어있다.

- `sorting.h` 헤더파일은 과제의 핵심인 4가지 정렬을 수행할 수 있게 종류별 정렬이 함수로 정의되어있다. (`insertion_sort`, `selection_sort`, `merge_sort`, `quick_sort`) 또한, 중복을 허용하는 데이터셋의 정렬에는 하나의 숫자가 배열에 있는 것이 아니라 하나의 문자와 하나의 숫자가 배열에 있기 때문에 그에 해당하는 정렬을 해줄 `pair_insertion_sort`, `pair_selection_sort`, `pair_merge_sort`, `pair_quick_sort`로 구성되어있다. 이를 위해 `keydata.h`를 `#include` 하였다. 모든 함수는 `main`함수에서 사용시 `parameter`를 동일하게 하기위해 보조함수인 `merge_sorting`, `quick_sorting` 함수가 있다. 총 10개의 함수로 구성되어있다.

- `keydata.h` 헤더파일은 `key`와 `data`를 넣을 수 있는 `class`가 있다. 중복을 허용하는 데이터셋의 정렬에 `key`와 `data`가 하나의 배열 원소에 들어가기 때문에 `class`의 멤버를 `key`와 `data`로 지정하여 사용하였다. 값을 대부분 `main`함수에서 컨트롤하기 때문에 `data`나 `key`값이 필요한 함수는 `friend` 함수로 지정하여 `private` 멤버를 일부 허용하였다. (`print_pair`, `pair_insertion_sort`, `pair_selection_sort`, `pair_merge_sort`, `pair_quick_sort`)

- 드라이버 파일인 `main.cpp` 파일은 정상임을 보이는 것과 중복을 허용한 데이터 정렬, 알고리즘 성능 비교 이 세가지를 구분하고 간결하게 구성하기 위해 함수로 묶어서 `main`함수 내부를 구성했다. `main`함수 및 세 가지 핵심 함수를 실행하는데 필요한 함수들은 `main`함수 앞에 `prototype`을 남기고 뒷 부분에 코드를 작성하였다. 시간을 표현하기 위해서 `time()`과 `clock()`을 모두 사용해보았으나 초단위와 `ms` 단위이기 때문에 0이 나와서 `ns` 단위의 시간을 얻기 위해 `chrono`를 사용하였다. 알고리즘의 성능을 비교하기 위해 배열의 크기가 클수록 좋은데, 배열의 크기가 매우 크기 때문에 `random_av`와 `test_av`는 전역 `scope`에 선언한다.

[코드 분석]

1. keydata.h

코드, 목적 및 상세설명
<pre>friend void print_pair(const keydata* array, int size); friend void pair_insertion_sort(keydata* av, int size); friend void pair_selection_sort(keydata* av, int size); friend void pair_merge_sorting(keydata* array, int start_index, int end_index); friend void pair_quick_sorting(keydata* array, int start_index, int</pre>
해당 클래스의 private값인 key값과 data 값을 비교해야하기 때문에 friend 함수로 지정

2. sorting.h

코드, 목적 및 상세설명
<pre>#include "keydata.h"</pre>
<key, data> 비교를 위해 생성한 keydata 클래스를 사용.
<pre>void insertion_sort(int* av, int size) { for (int i = 1; i < size; i++) { const int key = av[i]; int j = i - 1; while (j < size && key < av[j]) { av[j + 1] = av[j]; j--; } av[j + 1] = key; } }</pre>
insertion_sort 함수. [0]은 이미 정렬이 되었다는 가정하에 시작하기 때문에 i는 1부터 for문을 시작함. 두번째 칸의 값부터 key로 설정하여 값을 비교함. j는 같은 반복문에서 i의 앞칸, j의 값을 줄여가며 key보다 앞에 있는 모든값과 비교를 할 수 있게 함. 원소를 오른쪽으로 한칸 이동시키는데 j--가 반복되면서 비교하는 값 중 연속된 큰 값들이 모두 오른쪽으로 한칸 이동함. 조건이 탈출한 곳에서 즉 key보다 작은 값을 만났을 때 key 값을 av[j+1]에 대입한다. av[j+1] = key 문장이 끝난 뒤 이후 i가 1이 추가 되고 key = av[i]를 통해 그 아래 원소를 key로 하여 비교를 반복한다.
<pre>void pair_insertion_sort(keydata* av, int size){ ... }</pre>
insertion_sort함수와 동작은 일치하나 정렬하는 배열의 원소가 keydata임

```

void selection_sort(int* av, int size) {
    for (int i = 0; i < size; i++) {
        int min_elem = av[i];
        int index = i;
        for (int j = i + 1; j < size; j += 1) {
            if (min_elem > av[j]) {
                min_elem = av[j];
                index = j;
            }
        }
        int temp;
        temp = av[i];
        av[i] = av[index];
        av[index] = temp;
    }
}

```

selection_sort 함수, 가장 먼저 첫 인덱스의 값이 최소값이라는 가정을 함. j는 최소값으로 지정한 다음값으로 j부터 끝까지 for을 통해 최소값을 찾는 스캔작업을 함. 그 최소값을 처음 지정한 최소값에 대입함. 여기서 대입을 해주는 이유는 반복을 하며 이 값보다 더 작은 값을 찾기 위해서이다. 따로 min_elem에 저장했기 때문에 실제 배열에서의 값은 바뀌지 않는다. temp를 통해 값을 서로 바꿔준다.

```

void pair_selection_sort(keydata* av, int size) {
...
}

```

selection_sort함수와 동작은 일치하나 정렬하는 배열의 원소가 keydata임

```

if (size > 1) {
...
}

```

merge_sorting 함수. merge_sort는 배열을 절반으로 나누고 이를 반복한 뒤 양분된 배열에서 정렬을 해주는 sorting이다. size가 1이 될 때까지 divide를 해야하므로 size>1이라는 조건을 사용한다.

```

const int left_start = start_index;
const int right_start = left_start + (size / 2);
const int left_end = right_start - 1;
const int right_end = end_index;

```

직관적으로 파악하기 위해 const int를 총 4개선언

left_start : 양분한 배열중 왼쪽 첫번째 배열

right_start : 양분한 배열중 오른쪽 첫번째 배열

left_end : 양분한 배열중 왼쪽 마지막 배열

right_end : 양분한 배열중 오른쪽 마지막 배열

```

merge_sorting(array, left_start, left_end);
merge_sorting(array, right_start, right_end);

```

배열의 범위를 통해 절반으로 나눈 뒤 merge_sorting을 재귀하면서 1개가 남을 때 까지 양분하고 정렬함.

```
int* new_array = new int[size];
```

merge_sorting의 정렬을 양분된 배열 중 같은 선상의 원소를 하나씩 비교해보며 오름차순일 경우 작은 값을 새로운 배열에 넣어주고 한쪽 배열이 모두 옮겨졌을 때 남은 배열을 그 뒤에 그대로 붙여주는 방법이다. 이를 위해 새로운 배열을 할당하는 코드를 작성.

```
int left_index = left_start;  
int right_index = right_start;  
int new_index = 0;
```

양분된 배열을 비교하기 위해 두개의 인덱스를 선언

left_index : 왼쪽 배열의 시작 인덱스

right_index : 오른쪽 배열의 시작 인덱스

새로 생성된 배열에 들어갈 값의 위치를 지정해주기 위해 new_index 선언

```
while (left_index <= left_end && right_index <= right_end) {  
    if (array[left_index] < array[right_index]) {  
        new_array[new_index] = array[left_index];  
        left_index++;  
    }  
    else {  
        new_array[new_index] = array[right_index];  
        right_index++;  
    }  
    new_index ++;  
}
```

while 반복문을 통해 left_index나 right_index가 비교를 모두 마치고 계속 +1이 되어서 양분된 배열을 벗어날 때 까지 반복하여 계산. 더 작은쪽의 원소가 new_array로 순서대로 옮겨지고, 그 배열의 인덱스는 한칸씩 오른쪽으로 이동함. new_array에 값이 하나씩 추가될 때마다 다음 칸에 값을 넣기 위해 인덱스도 1씩 증가.

```
if (left_index > left_end) {  
    copy(array + right_index, array + right_end + 1, new_array + new_index);  
}  
else {  
    copy(array + left_index, array + right_start, new_array + new_index);  
}  
copy(new_array, new_array + size, array + start_index);
```

while문을 마치고 인덱스가 남은 배열의 남은 값들을 new_array의 남은 부분에 copy 기능을 활용하여 저장함.

copy기능, copy(a,b,c) a부터 b까지 c의 시작점부터 차례대로 대입함. 즉 남아있는 모든 배열을 new_array의 남은 부분에 저장

```
copy(new_array, new_array + size, array + start_index);
```

```
delete[] new_array;
```

정렬된 배열을 만들기 위해 사용되었던 new_array는 배열을 원래 배열에 옮겨주고 delete를 통해 소멸시킴.

```
void merge_sort(int* array, int size) {  
    merge_sorting(array, 0, size - 1);  
}
```

main함수에서 다른 정렬들과 같이 동일한 형태로 값을 받기 위해 merge_sort로 배열과 크기만을 받아오고 실제 merge_sorting을 통해 실제 합병정렬을 수행함.

```
void pair_merge_sorting(keydata* array, int start_index, int end_index) {  
    ...  
}
```

merge_sorting함수와 동작은 일치하나 정렬하는 배열의 원소가 keydata임

```
void pair_merge_sort(keydata* array, int size) {  
    ...  
}
```

merge_sort함수와 동작은 일치하나 정렬하는 배열의 원소가 keydata임

```
const int pivot = array[last_index];
```

배열의 가장 끝 자리의 값을 pivot으로 지정

```
while (left_index < right_index) {  
    ...  
}
```

왼쪽에서는 값을 비교하며 오른쪽으로 다가오고 오른쪽에서는 값을 비교하며 왼쪽으로 다가오고 있는데 이 값이 교차되어 조건이 성립하지 않을 경우 while문을 탈출한다.

```
while (left_index <= right_index && left_index < last_index - 1 && array[left_index] < pivot) {  
    left_index ++;  
}
```

양쪽 인덱스가 만나지 않고 배열 범위 내에 있을 때, pivot보다 큰 값을 만나면 왼쪽 배열 인덱스의 오른쪽 이동을 멈춘다.

```
while (left_index <= right_index && right_index > 0 && array[right_index] > pivot) {  
    right_index --;
```

양쪽 인덱스가 만나지 않고 배열 범위 내에 있을 때, pivot보다 작은 값을 만나면 오른쪽 배열 인덱스의 왼쪽 이동을 멈춘다.

```
if (left_index < right_index) {  
    int temp;  
    temp = array[left_index];  
    array[left_index] = array[right_index];  
    array[right_index] = temp;  
}
```

만약 left_index가 right_index보다 더 오른쪽으로 갔다면(교차했다면) array[left_index]와 array[right_index]의 값을 바꿔준다.
<pre> if (left_index == last_index - 1 && left_index == right_index) { int mid = (last_index - start_index + 1) / 2; // int temp; temp = array[last_index]; array[last_index] = array[last_index - mid]; array[last_index - mid] = temp; } </pre>
<p>pivot이 배열에서 가장 큰 값을 차지하여 양분이 되지 않을때이다. 왼쪽 인덱스가 자신보다 큰 값을 만나지 못해 결국 pivot 바로 앞에 오른쪽 인덱스는 자신보다 작은 값이 없어 인덱스에 변화가 없게 되면 두 인덱스가 같아진다.</p> <p>배열의 중간 인덱스를 나타내는 mid 배열의 중간 인덱스를 찾기 위해 전체 크기에서 /2를 해준다. temp를 통해 배열의 마지막 값과 중간값을 바꿔준다.</p>
<pre> int temp; temp = array[left_index]; array[left_index] = array[last_index]; array[last_index] = temp; </pre>
<p>pivot보다 큰 값을 가르키는 left_index를 가장 뒤로 보내기 위해 가장 뒤에 위치한 last_index의 값과 바꿔줌.</p>
<pre> void quick_sort(int* array, int size) { quick_sorting(array, 0, size - 1); } </pre>
<p>main함수에서 다른 정렬들과 같이 동일한 형태로 값을 받기 위해 quick_sort로 배열과 크기만을 받아오고 실제 quick_sorting을 통해 실제 빠른정렬을 수행함.</p>
<pre> void pair_quick_sorting(keydata* array, int start_index, int last_index) { ... } </pre>
<p>quick_sorting함수와 동작은 일치하나 정렬하는 배열의 원소가 keydata임</p>
<pre> void pair_quick_sort(keydata* array, int size) { pair_quick_sorting(array, 0, size - 1); } </pre>
<p>quick_sort함수와 동작은 일치하나 정렬하는 배열의 원소가 keydata임</p>

3. main.cpp

코드, 목적 및 상세설명
<pre> #include <chrono> #include <stdlib.h> #include <time.h> </pre>

시작시간과 끝시간 측정을 위해 chrono 라이브러리 사용

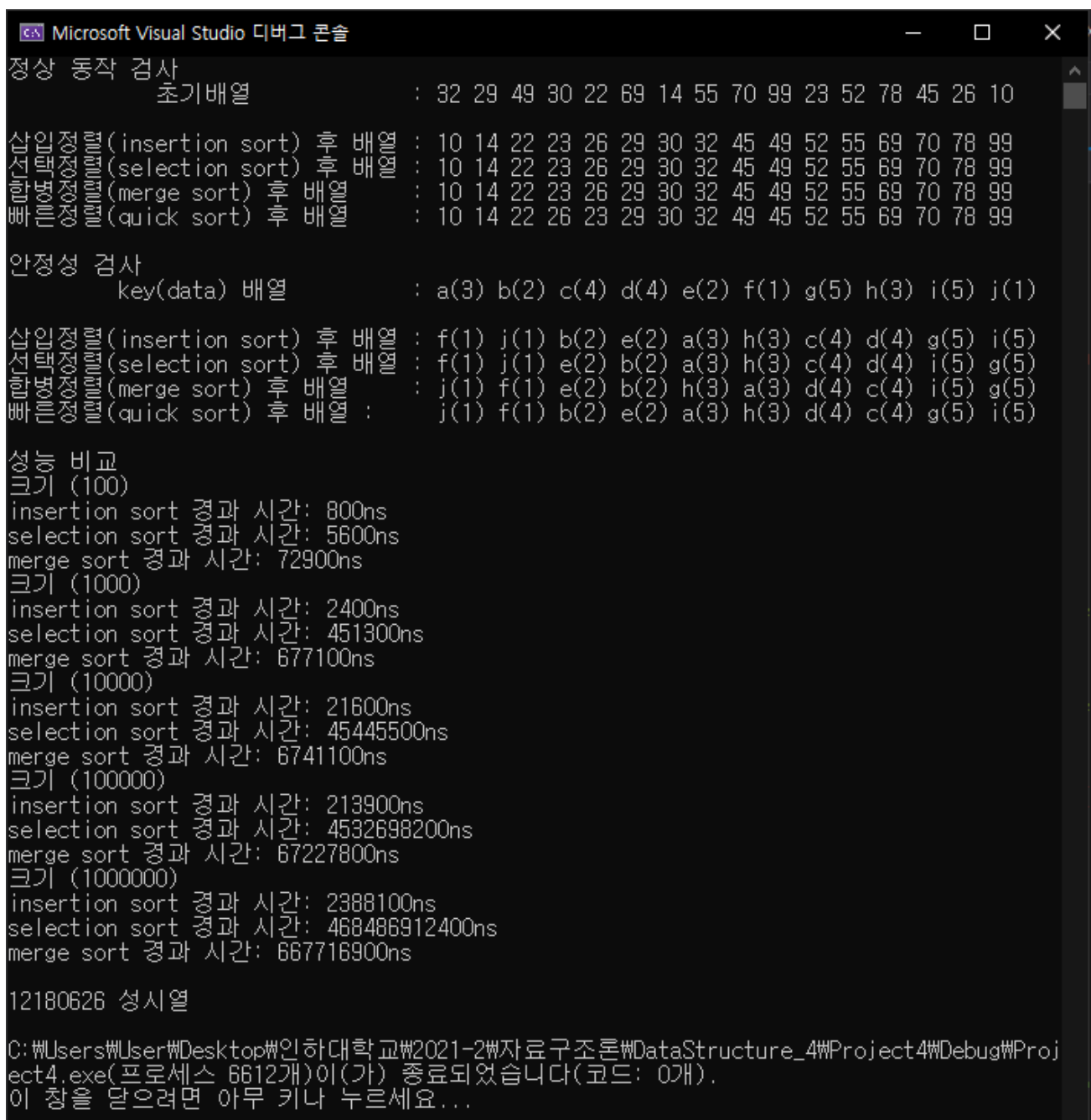
배열의 값을 랜덤으로 배치하기 위해 stdlib.h, time.h 사용

```
for (int i = 0; i < size; i++) { test[i] = S[i]; }
```

정렬을 할 때마다 배열이 정렬되므로 test 배열을 각 정렬기능을 통해 정렬하고 다시 S를 대입하여 초기화함.

이 외의 코드에 대한 모든 설명은 주석으로 자세히 작성함.

[실행결과분석]



```
Microsoft Visual Studio 디버그 콘솔

정상 동작 검사
초기배열          : 32 29 49 30 22 69 14 55 70 99 23 52 78 45 26 10

삽입정렬(insertion sort) 후 배열 : 10 14 22 23 26 29 30 32 45 49 52 55 69 70 78 99
선택정렬(selection sort) 후 배열 : 10 14 22 23 26 29 30 32 45 49 52 55 69 70 78 99
합병정렬(merge sort) 후 배열    : 10 14 22 23 26 29 30 32 45 49 52 55 69 70 78 99
빠른정렬(quick sort) 후 배열    : 10 14 22 26 23 29 30 32 49 45 52 55 69 70 78 99

안정성 검사
key(data) 배열    : a(3) b(2) c(4) d(4) e(2) f(1) g(5) h(3) i(5) j(1)

삽입정렬(insertion sort) 후 배열 : f(1) j(1) b(2) e(2) a(3) h(3) c(4) d(4) g(5) i(5)
선택정렬(selection sort) 후 배열 : f(1) j(1) e(2) b(2) a(3) h(3) c(4) d(4) i(5) g(5)
합병정렬(merge sort) 후 배열    : j(1) f(1) e(2) b(2) h(3) a(3) d(4) c(4) i(5) g(5)
빠른정렬(quick sort) 후 배열    : j(1) f(1) b(2) e(2) a(3) h(3) d(4) c(4) g(5) i(5)

성능 비교
크기 (100)
insertion sort 경과 시간: 800ns
selection sort 경과 시간: 5600ns
merge sort 경과 시간: 72900ns
크기 (1000)
insertion sort 경과 시간: 2400ns
selection sort 경과 시간: 451300ns
merge sort 경과 시간: 677100ns
크기 (10000)
insertion sort 경과 시간: 21600ns
selection sort 경과 시간: 45445500ns
merge sort 경과 시간: 6741100ns
크기 (100000)
insertion sort 경과 시간: 213900ns
selection sort 경과 시간: 4532698200ns
merge sort 경과 시간: 67227800ns
크기 (1000000)
insertion sort 경과 시간: 2388100ns
selection sort 경과 시간: 468486912400ns
merge sort 경과 시간: 667716900ns

12180626 성시열

C:\Users\User\Desktop\인하대학교\2021-2\자료구조론\DataStructure_4\Project4\Debug\Project4.exe(프로세스 6612개)이(가) 종료되었습니다(코드: 0개).
이 창을 닫으려면 아무 키나 누르세요...
```


먼저 4가지 정렬이 잘 작동하는지 파악하기 위해 초기에 생성한 배열을 출력함.

이후 4가지 정렬 후 배열을 순서대로 출력함

다음은 안정성 검사를 위해 문자하나와 숫자하나가 있는 배열을 출력함.

이후 4가지 정렬 후 배열을 순서대로 출력함.

마지막으로 성능비교를 위해 배열의 크기별로 각 정렬의 경과 시간을 출력함.

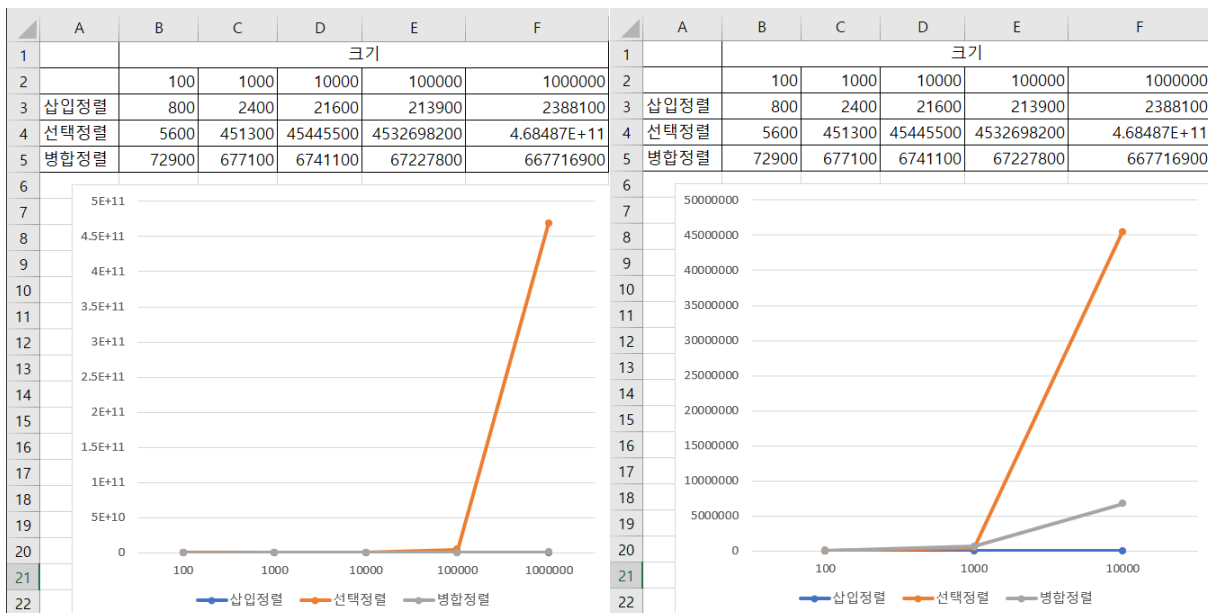
크기는 100, 1000, 10000, 100000, 1000000까지 성능을 확인해봄.

이후 학번과 이름을 출력한다.

[Quick Sort에서 pivot을 어떻게 선정하였는가?]

보통의 경우 가장 마지막 자리의 값을 pivot으로 지정했다. pivot이 배열에서 가장 큰 값을 차지하여 양분이 되지 않을 때 pivot을 배열의 전체 크기에서 중간에 위치한 값과 pivot 위치인 가장 끝 값을 바꿔준다. 비록 바꿔준 뒤에도 값이 가장 클 확률이 있지만 이미 오른쪽에서 가장 큰 값이 있었기 때문에 중앙 지점에 극히 드물다고 생각한다. 양분이 정확히 효율적으로 되지 않더라도 최악의 경우를 피할 수 있다. 또한, 결국 배열의 크기가 커지면 배열 값의 중앙값을 찾는 데도 많은 작업을 필요로 하는데 배열의 중간 위치의 값은 $size/2$ 을 통해 원하는 인덱스의 값을 한번에 찾을 수 있다. 그렇기 때문에 pivot이 극단적이어서 양분이 되지 않을 때, 중간 위치의 값과 가장 끝의 값을 바꿔준다.

[성능비교 그래프]



선택정렬이 너무 압도적으로 큰 값을 가지고 있어 파악이 불가해 크기가 10000까지인 그래프도 함께 첨부하였다. 이를 통해 내가 구성한 정렬 중 선택 정렬이 가장 성능이 좋지 않고 삽입 정렬의 성능이 가장 좋은 것을 알 수 있다.

[결론 및 느낀점, 어려웠던 점]

자료구조론 네번째 과제를 하면서 4가지 종류의 정렬을 모두 구현해볼 수 있었다.

설명으로 들었을 때 정말 이해할 수 있고, 구현할 수 있겠다고 생각했지만 실제로 너무 어려웠던 과제 중 하나였다. 특히 merge sort와 quick sort는 구현이 너무 어려워서 실제로도 많은 시간이 걸렸다. 결과값을 볼 때 quick sort는 완벽히 정렬되지 않았음을 알 수 있다. pivot의 선정에 따라 값이 바뀔 수 있다고 생각은 했으나 교수님의 설명으로 sorting을 하면 가장 대표적인 것이 quick sort이며 많이 사용한다고 알고 있는데, 정렬을 성공시키지 못해 아쉬움이 많이 남는다. 과제를 제출한 후에도 올바른 정렬을 위해 코드를 좀 더 분석해봐야겠다.

이후 안정성 검사에서도 재밌는 결과를 얻을 수 있었다. 정말 정렬의 방법마다 숫자는 잘 정렬된 듯 보이나 모두 다르게 정렬된 것을 알 수 있었다. 이런 경우는 중복된 값이 있을 때 필요한 알고리즘인데, 추후에 따로 프로그램을 구성할 때 이러한 정렬이 필요하다면 Stable하면서 추가적인 공간이 필요하지 않은 insertion sort 방법을 사용해야겠다.

그리고 성능비교를 통해 정렬에 걸린 시간을 확인해보았다. 먼저 quick sort 알고리즘이 걸린 시간을 출력하는 코드를 구성하였으나 코드에 포함시켜서 실행을 시키면 quick sort에서 출력이 되지 않고 멈추는 현상이 계속 발생해 주석처리 후 실행을 시켰다.

선택정렬과 삽입정렬은 $O(n)$, 병합정렬과 빠른정렬은 $O(n\log n)$ 의 시간복잡도를 가지고 있으므로 큰 수일수록 병합정렬이 더욱 빠르게 실행되어야 한다. 하지만 내가 구성한 코드상으로는 크기가 1,000,000일 때 삽입 정렬보다는 크고 선택 정렬보다는 작은 값이 출력되었다. 이보다 더 삽입 정렬보다 오래 걸린 이유에 대해서도 코드를 분석하며 공부하는 시간을 가져야겠다.

그래도 복잡한 코드를 작성하였지만 깔끔한 결과물을 보여주는 이번 과제를 하면서 코딩의 재미를 한 층 더 느낄 수 있었다.