



Object-Oriented Programming Report

Assignment 1-3

Professor	Donggyu Sim
Department	Computer engineering
Student ID	2022202065
Name	박나림
Class (Design / Laboratory)	2 / C
Submission Date	2023. 3. 30

Program 1

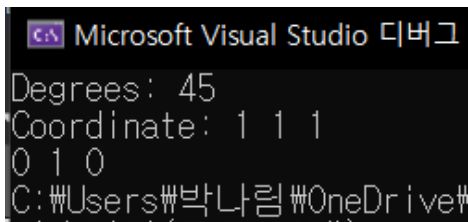
□ 문제 설명

행렬 변환 프로그램으로, 차원 좌표 값들을 입력 받으면 연산자 T 를 통해 행렬의 곱셈을 계산하는 프로그램이다. 연산자 T 는 3 가지 T 의 값들로 구성되어 있는데, $T1$ 은 z 축에 대한 회전, $T2$ 는 yz 평면에 대한 반사이며 $T3$ 는 xy 평면의 직교 투영 값들로 구성된다. 이러한 3 가지 행렬 값들을 모두 곱한게 연산자 T 로, 이 프로그램에서는 정해져 있는 T 를 이용해 연산을 진행한다. T 는 3×3 의 삼각함수 행렬 배열이며 사용자로부터 각도와 좌표 값 3 개를 입력 받아서 T 에 각도를 대입하고, 좌표 값은 3×1 의 행렬로써 곱셈 연산을 진행하고 출력하는 역할을 한다.

-구현 방법

먼저 각도를 계산하기 위해 제일 기본이 되는 파이 값을 `const double` 형으로 고정시킨다. 그리고 각도, 라디안 변수와 입력 받을 좌표 값 변수, `sin`과 `cos` 변수를 선언한다. 사용자로부터 각도와 좌표 값들을 차례대로 입력 받으면 `int` 형 배열에다가 그 좌표 값들을 저장한다. 그 다음 삼각함수를 계산하는데, 이때 `sin`과 `cos` 함수를 이용해 연산을 진행하면 오차가 발생하므로 공식에서 쓰이는 기본 각도들(0, 30, 45, 60, 90)은 따로 스위치문을 통해 직접 연산을 진행한다. 그 외 다른 각도인 경우에만 각도를 라디안으로 변환하고 `sin` 함수와 `cos` 함수로 값을 전달한다. 이러한 연산을 위해 거듭제곱, 팩토리얼 함수를 구현하고 그 함수들을 이용하여 `sin` 함수, `cos` 함수를 테일러 급수 공식의 급수 합 전개식을 써서 구현한다. 이렇게 계산 된 값들은 `double` 형 3×3 배열로 T 연산자 배열에 저장하고 행렬 곱셈 연산을 진행하는 함수로 전달된다. 행렬 곱셈 함수에서는 T 배열과 좌표 값들을 저장한 배열을 매개변수로 전달받아 연산을 진행하고 출력한다.

□ 결과 화면



```
Microsoft Visual Studio 디버그
Degrees: 45
Coordinate: 1 1 1
0 1 0
C:\Users\박나림\OneDrive\...
```

사용자로부터 각도, 좌표 값들을 입력 받으면 행렬 곱셈을 진행하여 출력하게 된다.

□ 고찰

구현할 당시에는 오차 허용이나 내장된 \sin , \cos 함수 허용 등을 몰라서 직접 구현하려니 어려운 점들이 있었다. 먼저 \sin , \cos 함수들에 대해서 다시 공부해보니 테일러 급수를 이용한다는 사실을 깨달아, 공식을 코드로 옮기는 형태로 구현하였다. 그러다 보니 값을 입력했을 때 오차 값이 발생하였다. 그래서 이 점에 대한 해결방안을 생각해보다가, 기본 각도들에 대해서는 직접 계산하면 값이 정확하게 나올 수 있다는 점을 생각하여 스위치문으로 나누어서 진행했다. 하지만 그 외 값들은 어쩔 수 없이 함수를 사용하여 오차가 발생한다. 이러한 오차를 줄이는 방법으로는 식의 형태를 바꾸는 방식을 취하면 될 것 같은데, 어떻게 할 수 있는지 더 알아보고 싶다.

Program 2

□ 문제 설명

회로의 출력 전압 값을 구하는 프로그램으로, 저항이 직렬연결, 병렬연결인지에 따라 크게 2 가지 경우로 나누어서 계산을 진행한다. 사용자로부터 입력 전압 값, 2 개의 저항 값, 저항의 병렬 여부, 총 4 개의 변수 값들을 입력 받으면 공식을 이용하여 총 출력 전압 값을 출력한다. 이때 나눗셈 연산을 진행하게 되면 무한소수가 나올 가능성이 있다. 따라서 먼저 약분하기 전인 분수의 형태로 출력, 그리고 무한 소수의 여부를 같이 출력하는 형태로 진행한다. 병렬연결이 있을 경우에는 부하 전력 비율 값도 퍼센트 형식으로 같이 출력하도록 한다. 모든 공식은 옴의 법칙을 기반으로 한 공식들을 이용한다.

-구현 방법

사용자로부터 전압, 저항 값들을 입력 받아서 R_{load} 의 값이 0 인경우와 아닌 경우로 조건문을 사용하여 각 함수로 전달한다. 0 일 경우 직렬 회로 함수로 전달 하고, 아닐 경우 병렬 회로 함수로 전달하는 것이다. 직렬 회로 함수에서는 전압과 두개의 저항 변수를 매개변수로 받아서 V_{out} 을 구하는 공식에 대입한다. 이때 처음부터 온전히 대입하면 바로 순환소수로 넘어갈 가능성도 있으므로 첫 단계는 약분을 하지 않고 분수의 형태로 출력되게 한다. 또한 순환 소수 여부를 구분하기 위해 소인수를 구하는 조건문을 만든다. 여기서 소인수를 저장할 새 배열이 필요하므로 출력 전압 값의 제곱근 식을 하여 구한 값만큼 동적 할당을 시킨다. 그리고 소인수를 구하여 배열에 저장한 다음, 배열의 원소 값을 처음부터 끝까지 검사하면서 2 또는 5 가 나올 때에만 카운트를 증가시킨다. 반복문이 다 끝나고 이때 증가한 카운트 값이 배열의 크기와 같을 경우, 소인수가 전부 2 또는 5 밖에 없는 유한소수이므로 출력 전압 값을

그대로 계산하여 같이 출력한다. 만약 배열의 크기와 다를 경우 중간에 다른 소인수가 있다는 뜻이므로 순환소수가 된다. 이 경우에는 출력 전압 값의 나눗셈을 반복하여 나머지의 값들을 새롭게 또 동적 할당된 배열에 저장하여 for 문을 통해 배열의 값 중 중복되는 값이 어디서부터 있는지 검사한다. 결과적으로 중복된 값이 나오면, 그 부분을 ()로 묶어주어서 순환소수를 표기하고 최종 출력하는 형식으로 진행한다. 병렬 회로 함수에서도 이와 같은 원리로 진행하는데, 여기서는 전압을 구하는 공식이 달라지는 점에 유의하여 통분 변수를 새롭게 지정해주어 계산한다. 각 함수에서는 동적할당 했던 배열의 메모리를 해제해주는 작업도 꼭 넣도록 한다. 또한 병렬 회로에서는 전력을 구하는 것도 추가하여 같이 출력해준다.

□ 결과 화면

```
Microsoft Visual Studio 디버그 콘솔
Vs: 40
R1: 4
R2: 2
R(Load): 0

Vout: 80/6 = 13(.순환)

C:\Users\박나림\OneDrive\바탕 화면>
```

직렬 회로에서 값들을 입력했을 때 출력 전압 값을 계산한 결과이다. 이때 순환소수로 나올 시 순환되는 부분을 같이 알린다.

```
Microsoft Visual Studio 디버그 콘솔
Vs: 10
R1: 4
R2: 2
R(Load): 2

Vout: 40/20 = 2
Load power ratio: 10.0%
C:\Users\바나나\OneDrive\바나나
```

병렬 회로에서 값들을 입력했을 때 출력 전압 값과 전력을 계산하여 나온 결과이다.

□ 고찰

제일 어려웠던 점이 순환소수 부분을 구현하는 것이었는데, 먼저 순환소수를 판단하기 위해 소인수를 구하는 작업이 필요했다. 다 하고 실행을 시켜보니 직렬 회로 함수에서는 제대로 순환소수를 판단해냈지만 병렬 회로 함수에서는 버퍼 오버런 에러가 발생하였다. 같은 코드로 진행을 하였는데 이쪽에만 에러가 난 걸 보아 아마도 병렬 회로에서 새롭게 쓰는 공식으로 인해 에러가 발생한 것 같다. 소인수를 저장할 배열이 문제였는데, 배열의 크기를 제공근을 취하든 말든 동일한 에러가 발생해서 크기의 관한 에러를 어떻게 고쳐야 할 지 아직 잘 모르겠다. 배열의 크기가 추가될 때 자동으로 메모리 할당을 늘려주는 방법에 대해서 고려해 봐야 될 것 같다. 또한 순환소수를 표기하는 쪽에 대해서도 시행착오를 겪었는데, 여기서도 마찬가지로 새 배열을 할당했을 때 버퍼 오버런이 발생하였기에 이에 대한 해결방안을 찾아봐야 될 것 같다.

Program 3

□ 문제 설명

사용자로부터 입력 받은 수를 원하는 진수로 변환하는 프로그램이다. 2 진수, 8 진수, 10 진수, 16 진수로 이루어진다. 값을 입력하고 현재 값의 진법, 바꾸고 싶은 진법까지 3 개의 수를 입력하면 처리를 해야 하는 것이다.

- 1) 10 진법 변환: 해당 수를 원하는 진법의 숫자로 더 이상 나눌 수 없을 때까지 나눗셈 연산을 진행한다. 그리고 최종적으로 나온 몫을 첫번째 값으로 하여 차례대로 나왔던 나머지 값들을 거꾸로 배열하면 진법을 변환할 수 있다.
- 2) 2 진법 변환: 8 진법으로 변환할 때는 뒤에서부터 3 자리씩, 16 진법으로 변환할 때는 4 자리씩으로 끊어서 각각 뒤에서부터 순서대로 1 인 부분만 2^n ($n=0$ 부터)의 합으로 해당 자릿수를 계산한다. 따라서 8 진법은 3 자릿수, 16 진법은 2 자릿수(1~9, A~F 표현)로 나오게 된다.

-구현 방법

10 진법용 long 형 변수, 그 외 진법용인 char 형 배열, 원래 진법과 변환할 진법 변수를 선언한다. 진법을 차례로 입력 받고 바꿀 수도 입력 받는다. 이때 원래 진법이 어떤 것인지에 따라 2 경우로 나누어서 다른 변수로 입력 받는다. 10 진법에서는 나눗셈 연산을, 그 외 진법에서는 배열 각 자릿수 치환 연산을 하기위해서이다. 그 다음 스위치문으로 원래 진법 종류에 따라 각 함수로 입력 받은 수와 바꿀 진법 변수를 인수로 전달한다. 10 진수에서 변환하는 Decimal 함수에서는 입력 받은 수가 0 이하가 될 때까지 재귀적으로 나눗셈을 반복한다. 동시에 나머지도 차례로 출력하여 바로 변환할 수

있도록 한다. 8 진수에서 변환하는 Octal 함수와 16 진수에서 변환하는 Hexadecimal 함수에서는 각각 자릿수의 문자열을 스위치문으로 2 진수로 치환하여 새 문자열에 이어 붙이는 형식으로 다시 저장한다. 이것들은 모두 2 진수에서 변환하는 Binary 함수로 보내진다. 그래서 최종 바꿀 진법이 2 진법인 경우 그대로 출력, 8 진법일 경우 처음의 남은 2 자리 먼저 변환하여 출력 후 나머지 3 자리씩 나누어서 변환 후 출력한다. 10 진법일 경우 전부 변환하여 합하여 출력, 16 진법일 경우 4 자리씩 나누어서 9 이하의 숫자로 출력하고 10 부터는 스위치문으로 알파벳으로 변환하여 각 자릿수를 출력한다. 이러한 함수들 내에서 각 자릿수를 벗어나는 오버 플로우 연산이 발생할 경우에는 안내문을 띄우고 종료시키는 코드도 추가한다.

□ 결과 화면

```

Microsoft Visual Studio 디버그 콘솔
10 2 181
10110101
C:\Users\박나림\OneDrive\바탕 화면>
있습니니다(코드: 0개).

Microsoft Visual Studio 디버그 콘솔
2 8 10110101
265
C:\Users\박나림\OneDrive\바탕 화면>
습니니다(코드: 0개).

```

10 진수(181) -> 2 진수(10110101) 2 진수(10110101) -> 8 진수(265)

```

Microsoft Visual Studio 디버그 콘솔
8 16 265
B5
C:\Users\박나림\OneDrive\바탕 화면>
습니니다(코드: 0개).

Microsoft Visual Studio 디버그 콘솔
16 10 B5
181
C:\Users\박나림\OneDrive\바탕 화면>
습니니다(코드: 0개).

```

8 진수(265) -> 16 진수(B5) (추가) 16 진수(B5) -> 10 진수(181)

□ 고찰

진법을 변환하는 함수를 구현할 때 어떻게 하면 제일 간단하게 축약할 수 있는지가 제일 문제였다. 이 부분에서 많은 시행착오를 겪었는데, 10 진법은 재귀함수로서 간단화 시켰지만 8 진법과 16 진법에서 특히 어려웠던 것 같다. 진법을 변환하는 것은 여러가지 경우의 수가 있기 때문에 먼저 모든 진법을 2 진법으로 바꾸고 거기서부터 다시 변환하여 출력하려고 하였다. 그러다 보니 8 진법과 16 진법에서는 각 자릿수를 나누어서 계산해야 한다는 규칙이 있었는데, 여기서 반복문을 쓰려고

하자 문자열 배열의 반복이 까다로웠다. 그래서 어쩔 수 없이 스위치문으로 상황을 나누어서 각각 치환을 하였다. 실행속도는 괜찮았지만 아무래도 코드를 보기에 조금 복잡해 보인다는 점이 문제이다. 이 부분은 문자열에 관한 반복문을 더 공부해야 될 것 같다.

또한 10 진법과 그 외 진법의 경우의 수를 나누어서 long 형과 char 형으로 입력 받은 것도 하나로 통일하는 방식이 더 좋아 보인다. 10 진법도 똑같이 char 형으로 받아야 문제의 테스트케이스처럼 값을 순서대로 받을 수 있을 것 같은데, 10 진법 함수의 연산을 간단히 하기 위해 경우의 수를 나누었더니 입력 값 순서가 다르다는 문제점이 발생하였다. 문제와 달리 진법 변수들을 먼저 입력 후 수를 입력하는 방식으로 하였다. 따라서 이 부분도 문자열에 대한 공부가 더 필요할 것으로 보인다.

Program 4

□ 문제 설명

사용자로부터 입력 받은 정수 값들을 오름차순으로 정렬하고 중앙 값을 같이 구하여 출력하는 프로그램이다. Insertion sort, quick sort, merge sort, bubble sort 로, 총 4 가지의 정렬 방법을 이용하여 각각 구했을 때 걸리는 시간을 비교하도록 한다. 입력한 개수는 홀수인 경우만 고려하며, 프로그램에서는 사전 입력 개수를 알려주는 것으로 한다. 그래서 15 개의 값으로 진행하면, 사용자로부터 15 개의 정수 값들을 입력 받고 각각의 정렬 방법들을 이용하여 정렬된 수들을 출력하고 마지막으로 중앙 값까지 출력하고 끝내는 형식이다.

- 1) Insertion sort: 두번째 값부터 시작하여 앞의 배열 중 알맞은 위치로 삽입되는 정렬
- 2) quick sort: pivot 값을 기준으로 양측을 분할하여 각각 값을 비교하는 정렬
- 3) merge sort: 절반으로 계속 나누다가 2 개씩 남는 시점부터 합치면서 값을 교환하는 정렬
- 4) bubble sort: 끝에서부터 비교하여 그 전의 수가 더 클 경우 서로 위치를 교환하는 정렬

-구현 방법

먼저 사용자로부터 입력할 숫자들의 개수를 입력 받아 그 크기만큼 배열을 동적 할당시킨다. 그 후 숫자들을 입력 받은 뒤 정렬 함수로 배열과 배열의 크기-1 을 전달해준다. 정렬이 끝나면 차례로 배열을 출력하고 크기의 절반 값을 인덱스로 가지는 배열 값을 출력하여 중앙 값도 같이 출력 되도록 한다. 마지막으로 동적 할당했던 배열의 메모리를 해제시키고 프로그램을 종료한다.

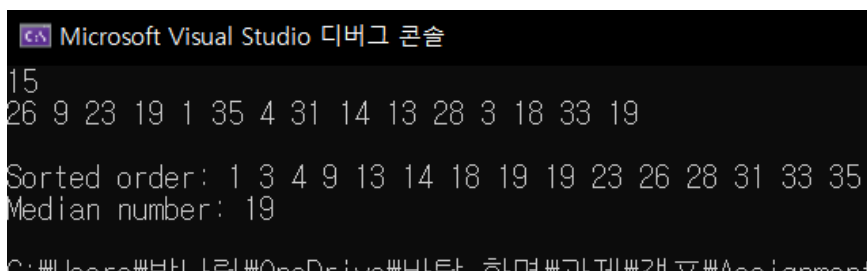
Insertion sort, 삽입 정렬 함수에서는 배열의 두번째 값부터 시작하기 위해 처음 시점을 0 이 아닌 1 부터 시작하여 마지막에 도달할 때까지 정렬 함수 호출을 반복하는 형태로 진행한다. 또한 bool 형으로 위치 변수를 false 로 저장하여, 반복문을 돌 때 조건을 추가하는 형식으로 해준다. 현재 가리키고 있는 배열의 값이 설정한 배열의 값보다 더 큰 경우 정렬이 된 상태이니 다음 칸을 가리키는 것으로 바꾼다. 만약 그러지 않을 경우 위치 변수에 true 를 저장하여 반복문을 탈출시키고 다음 배열 값에 현재 배열 값을 넣어 서로 값을 교환시킨다.

Quick sort 함수에서는 pivot 이란 변수와 함께 left, right 변수를 추가로 만들어 준다. 처음을 가리키는 pivot 을 기준으로 그 다음 칸을 left, 마지막 칸을 right 으로 설정하여 각각 가운데로 모이게 하면서 pivot 보다 큰 값, 작은 값을 찾아내어 그때마다 교환해주는 형태로 한다. 찾은 시점에서 left 와 right 이 순서대로 있을 경우 서로 값을 교환하고, 만약 순서가 엇갈릴 시 pivot 과 right 의 값을 교환해주며 정렬하고 반복문을 벗어나 분할 재귀적 호출로 다시 양측을 반복하여 계산한다.

Merge sort, 합병 정렬 함수에서는 절반으로 계속 나누면서 다시 합칠 수 있는 새 배열이 필요하기 때문에 이 함수 내부에서 쓸 목적으로 동적 할당을 통해 초기화까지 시켜준다. 또한 배열의 첫번째 값을 갖는 변수 2 개와 배열의 중간 값을 계산하여 인자로 받는 변수까지 다 초기화를 시켜준다. 그래서 처음 칸이 중간 칸까지, 중간+1 칸이 마지막 칸에 도달할 때까지 반복하여 첫번째 값과 중간+1 값이 정렬된 상태일 경우 새 배열에 그 값을 저장한다. 정렬이 필요한 경우에는 새 배열에 작은 값을 먼저 저장한다. 다시 합치는 과정까지 진행하면 새로 동적 할당했던 배열의 메모리 할당을 해제시킨다. 이 과정도 마찬가지로 양측을 분할하여 재귀적 호출로 계속 반복하여 계산하도록 한다.

Bubble sort 함수에서는 첫번째 값부터 비교연산에 바로 들어가므로 삽입 정렬과 달리 0 번째부터 시작하여 마지막 전까지 반복시킨다. 또한 이때 위치를 가리키는 변수는 마지막 값부터 시작하여 왼쪽 방향으로 진행하면서 그 전의 값이 더 클 시 서로 값을 교환해주며 정렬한다.

□ 결과 화면



```
Microsoft Visual Studio 디버그 콘솔
15
26 9 23 19 1 35 4 31 14 13 28 3 18 33 19
Sorted order: 1 3 4 9 13 14 18 19 19 23 26 28 31 33 35
Median number: 19
C:\Users\user\Desktop\OneDrive\바탕 화면\과제\2개 과제\Assignment
```

사용자로부터 입력할 숫자의 개수를 입력 받고, 그 수만큼 숫자들을 입력 받으면 오름차순으로 정렬되어 출력한 뒤 중앙 값도 같이 출력해 준 형태이다.

□ 고찰

-각 정렬 방법마다 배열의 원소의 개수를 10000 개로 설정하여 랜덤 값을 넣었을 때, 정렬하는 데에 걸리는 시간을 2 번씩 측정한 결과와 그에 따른 시간복잡도 비교

1) Insertion sort

```
Microsoft Visual Studio 디버그 콘솔
10000
sorting time: 0.048
C:\Users\박나림\OneDrive\바탕 화면>
업데이트(코드 : 0개)
디버깅이 중지될 때 코스를
Microsoft Visual Studio 디버그 콘솔
10000
sorting time: 0.05
C:\Users\박나림\OneDrive\바탕 화면>
업데이트(코드 : 0개)
```

배열 값들이 모두 역순으로 정렬된 최악의 경우, 외부 루프를 $n-1$ 번 도는 동안 비교 연산은 처음부터 마지막 전까지 반복하게 된다. 따라서 시간 복잡도는 $(n-1)(\text{회전 1}) + (n-2)(\text{회전 2}) + \dots + 2 + 1 = \frac{n(n-1)}{2}$ 로 나오게 되어 $O(n^2)$ 으로 나타낼 수 있다. 그러나 정렬할 배열의 값들이 이미 정렬된 최선의 경우일 시, 외부 루프를 $n-1$ 번 도는 동안 비교 연산이 1 번씩만 수행되므로 $O(n)$ 이라는 높은 효율성을 가진다. (최악의 경우가 $O(n^2)$) 실제로 결과 값을 보면 0.048로 빠른 편에 속하는 걸 볼 수 있는데, 이는 아래의 쿼 정렬과 비교했을 때 조금밖에 차이가 안 난다. 삽입 정렬에서는 최선의 경우와 최악의 경우의 시간이 차이가 꽤 나므로, 이번 측정에서는 최선에 조금 더 가까운 상황이었다는 걸 알 수 있다.

2) Quick sort

```

Microsoft Visual Studio 디버깅 콘솔
10000
sorting time: 0.03
C:\Users\박나름\OneDrive\바탕 화면>
순서입니다.(코드는 0개)

Microsoft Visual Studio 디버깅 콘솔
10000
sorting time: 0.039
C:\Users\박나름\OneDrive\바탕 화면>
있습니다.(코드는 0개).

```

처음 값을 pivot 으로 설정하여 정렬을 시킬 때, 이 pivot 값이 중간 값에 가까우면 최선, 전혀 가깝지 않은 경우 최악의 상황이 되어 시간 복잡도가 여러가지로 표현될 수 있다. 따라서 최악의 경우에는 한번씩 루프를 다 돌아야 되므로 $O(n^2)$ 이라는 시간 복잡도를 가지게 되며, 최선의 상황과 평균적 상황일 때에는 $O(N\log N)$ 으로 나타낼 수 있다. 그래서 정렬하고자 하는 배열이 오름차순 또는 내림차순으로 반대로 정렬되어 있는, 최악의 상황인 경우에는 배열에서의 가장 첫번째 원소와 중간 원소의 값을 서로 교환시킨다면 어느정도는 시간복잡도를 최선의 상황에 가깝게 개선시킬 수 있다. 물론 확률적인 이론이기에 완전히 개선할 수는 없어보인다. 따라서 불균형 분할에 의해 시간이 더 오래 걸릴 수 있으므로 불안정 정렬로 분류되기도 한다. 하지만 이번에 측정한 결과를 보면, 최악의 상황의 시간복잡도와 같은 시간복잡도를 가지는 버블 정렬에

3) Merge sort

```

10000
sorting time: 0.01
C:\Users\박나림\OneDrive\업슨입니다(코드: 0개)

10000
sorting time: 0.008
C:\Users\박나림\OneDrive\업슨입니다(코드: 0개).

```

4) Bubble sort

```

C# Microsoft Visual Studio 디버깅
10000
sorting time: 0.121
C:\Users\박나름\OneDrive\바탕 화면>
업니다(코드: 0개).

C# Microsoft Visual Studio 디버깅 콘솔
10000
sorting time: 0.202
C:\Users\박나름\OneDrive\바탕 화면>
업니다(코드: 0개).

```

것이다. 최선의 상황이나 최악의 상황이나 어떤 상황에서든 값은 동일하게 n^2 이 나오기 때문에 여러가지 정렬 방법들 중에서 제일 효율성이 떨어진다고 볼 수 있다. 다만 타 정렬 방법들과 비교하여 구현하기에는 간단한 편이기 때문에 시간복잡도를 크게 따질 필요가 없을 때, 배열의 원소의 개수가 크지 않은 상황 같은 때에는 적절할 것으로 보인다. 실제로도 각 함수들 코드 중에서 제일 간단하지만 시간을 측정한 결과로는 0.202 등으로 제일 오래 걸린 것을 확인할 수 있다.

이번 측정에서는 합병 정렬(0.008), 퀵 정렬(0.03), 삽입 정렬(0.048), 버블 정렬(0.121)순으로 시간 빠르기를 알 수 있었다. 평균적인 시간으로 따질 때, 이론 상으로는 합병 정렬($N\log_2 N$), 퀵 정렬($N\log_2 N$), 삽입 정렬(N^2), 버블 정렬(N^2)이기 때문에 실험과 거의 비슷한 결과가 나왔다는 것도 알 수 있었다. 하지만 평균적인 시간이 이렇게 나올 뿐, 가장 최선의 상황에서는 N 이라는 시간복잡도를 가지는 삽입 정렬이 가장 빠를 것이다. ($O(1) > O(\log N) > O(N) > O(N\log N) > O(N^2) > O(2^N) > O(N!)$ 순) 그래서 이러한 결과 값이 나오는 상황도 보고싶다. 또한 평소에는 구현이 비교적 간단한 버블 정렬만 알고 있었는데 이번 기회에 다른 정렬 방법들의 각각 상황 별 장단점을 알게 되면서 앞으로는 상황에 맞게 적절한 정렬 방법을 사용해야겠다고 느꼈다.