



Object-Oriented Programming Report

Assignment 1-3

Professor	Donggyu Sim
Department	Computer engineering
Student ID	2020202031
Name	Jaehyun Kim

Program 1

□ 문제 설명

$$T = \begin{matrix} & \begin{matrix} x & y & z \end{matrix} \\ \begin{matrix} -\cos\theta & \sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 \end{matrix} \end{matrix}$$

에서 θ 에 들어갈 육십분법 각도와 T와 y 값을 입력 받고 둘을 곱한 값을 출력하는 문제입니다. θ 는 육십분법으로 입력 받지만 $\cos()$ 과 $\sin()$ 함수가 인자로 호도법으로 표현된 각도를 사용하기 때문에, 각도를 호도법으로 변환하여 사용합니다.

$$x' = -\cos\theta x + \sin\theta y + 0z = -\cos\theta + \sin\theta y$$

$$y' = \sin\theta x + \cos\theta y + 0z = \sin\theta x + \cos\theta y$$

$$z = 0x + 0y + 0z = 0$$

□ 결과 화면

```
Degrees: 45
Coordinate: 1 1 1
-0 1 0
```

$$x' = -\cos 45^\circ + \sin 45^\circ, y' = \sin 45^\circ + \cos 45^\circ, z = 0$$

수식 상으로는 $\cos 45^\circ = \sin 45^\circ$ 이므로 $x'=0$ 이 나와야 합니다. 하지만 $\cos 45^\circ$, $\sin 45^\circ$ 가 double 형 실수이면서 값이 근사하기 때문에 round-off error 가 발생합니다.

round() 함수를 사용해서 반올림해주었기에 -0 1 0 이 출력되었습니다.

```
Degrees: 30
Coordinate: 1 2 3
0 2 0
```

$x' = -\cos 30^\circ + 2\sin 30^\circ = 0.133 \dots$, $y' = \sin 30^\circ + 2\cos 30^\circ = 2.232 \dots$, $z = 0$ 를 반올림하여 0 2 0 이라는 값이 출력되었습니다.

□ 고찰

cos, sin 함수가 육십분법이 아닌 호도법을 사용한다는 사실을 간과하고 코드를 짜는 바람에 올바르지 않은 출력이 나왔었습니다. 그래서 사용자로부터 육십분법으로 각도를 입력 받고 이를 호도법으로 바꿔서 계산하는 방향으로 다시 코드를 짚습니다.

예시와 같이 degrees: 45, coordinate: 1 1 1 을 입력했더니 round- off error 때문에

$0 \sqrt{2} 0$ 이 아닌 그 근사치가 출력이 되었습니다. 따라서 예시와 같이 해주기 위해 round 함수로 반올림 해주었고 그 결과 올바른 값이 출력됨을 확인했습니다.

Program 2

□ 문제 설명

전원전압과 저항들의 크기를 입력 받은 후 R_L 의 유무에 따라 다른 두 회로의 출력전압을 출력하고, R_L 이 존재하는 경우엔 전원전력에 대한 R_L 의 비율까지 출력해주는 문제입니다.

1) R_L 이 없을 경우

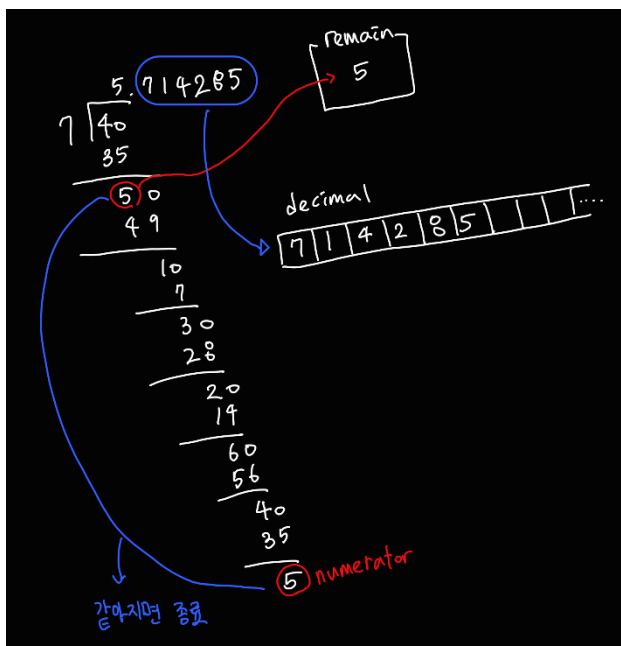
$$V_{out} = I \cdot R_2 = \frac{V_s}{R_1 + R_2} \cdot R_2$$

2) R_L 이 있을 경우

$$V_{out} = \frac{V_s}{R_1 + R_2 || R_L} \cdot R_2 || R_L = \frac{V_s R_2 R_L}{R_1(R_2 + R_L) + R_2 R_L}$$

$$\frac{P_L}{P_s} = \frac{V_L I_{R_L}}{V_s I} = \frac{R_2 V_L}{V_s(R_2 + R_L)}$$

printDecimal 함수에 대해 설명하겠습니다.



나눗셈의 과정을 그대로 코드로 구현했습니다. 40 / 7 을 예시로 설명하면, 처음 40 이었던 numerator 을 denominator 로 나눈 몫은 decimal 배열에 저장해주고 나눗셈 후 남은 나머지를 numerator 에 계속해서 저장합니다. 이때, 나눗셈을 시작하고 처음으로 나온 나머지는 remain 변수에 저장해둡니다. numerator 가 처음으로 나온 나머지 remain 과 같아지는 시점이 오면, 순환소수의 순환마디가 끝났다는 뜻이므로 반복문을 탈출합니다. decimal 을 출력합니다.

□ 결과 화면

```
Vs: 40
R1: 4
R2: 2
R(Load): 0
Vout: 80/6 = 13.(3)
```

$$V_{out} = \frac{40V}{4\Omega + 2\Omega} \cdot 2\Omega = 13.\dot{3}$$

```
Vs: 10
R1: 4
R2: 2
R(Load): 2
Vout: 40/20
Load power ratio: 10.00%
```

$$V_{out} = \frac{10V \cdot 2\Omega \cdot 2\Omega}{4\Omega(2\Omega + 2\Omega) + 2\Omega \cdot 2\Omega} = \frac{40}{20} = 2V$$

$$\frac{P_L}{P_s} = \frac{(40/20)V \cdot 2\Omega}{10V(2\Omega + 2\Omega)} = \frac{1}{10}$$

$$\text{Load power ratio} = 100 \times \frac{P_L}{P_s} = 10.00\%$$

□ 고찰

Load power ratio 를 출력하는 과정에서 자꾸 엉뚱한 값이 나와서 그 이유를 찾던 중 double Vout = numerator / denominator; 에서 오류가 발생했다는 사실을 발견했습니다. numerator 과 denominator 은 int 형 변수이기에 / 연산을 진행하면 소수부분은 버려지고 몫만이 Vout 에 저장되어, 뒤이어 나오는 연산들의 결과가 잘못 출력되는 것이었습니다. 그렇기에 double Vout = (double)numerator / denominator; 강제 형 변환을 통해 double 형 /연산으로 소수부분까지 잘 저장되게끔 코드를 수정했습니다.

Program 3

□ 문제 설명

사용자로부터 Input 으로 수와 그 수의 진법, 변환하여 출력할 진법을 입력 받고, 입력 받은 수를 출력하고자 하는 진법으로 변환하여 출력합니다.

일단 진법과 상관없이 입력 받은 모든 수를 10 진법 정수로 변환하고 이를 출력하고자 하는 진법으로 바꿔 출력했습니다.

□ 결과 화면

```
Input: 181 10 2
Output: 10110101
```

10 진법 181 을 2 진법으로 변환한
10110101 이 출력된 모습이다.

```
Input: 10110101 2 8
Output: 265
```

2 진법 10110101 을 8 진법으로 변환한
265 가 출력된 모습이다.

```
Input: 265 8 16
Output: B5
```

8 진법 265 를 16 진법으로 변환한 265 가
출력된 모습이다.

```
Input: B5 16 10
Output: 181
```

16 진법 B5 를 10 진법으로 변환한 181 이
출력된 모습이다.

□ 고찰

입력 받은 수를 원하는 진법으로 변환하기 전 거쳐가는 진법을 2 진법으로 할지 10 진법으로 할지 고민했습니다. 8, 16 진법을 2 진법으로 변환할 때와 10 진법을 2 진법으로 변환할 때 방식이 다릅니다. 하지만 2, 8, 16 을 10 진법으로 변환할 때, 그리고 출력할 때 모두 같은 방식이므로 코드를 간결하게 짜기 위해 모든 수를 10 진법으로 변환하고 출력하고자 하는 진법으로 변환하도록 코드를 짰습니다.

Program 4

□ 문제 설명

사용자로부터 수들을 입력 받고, 그 수들을 오름차순으로 정렬하는 프로그램을 작성하는 문제입니다. 구현해야 할 정렬함수들은

1. Bubble sort, 2. Insertion sort, 3. Merge sort, 4. Quick sort 가 있습니다.

1) Bubble sort

배열의 마지막 두개의 값(list[last], list[last - 1])끼리 비교해서 swap 하고 한 칸 앞 값(list[last - 1], list[last - 2])끼리도 비교하면서 첫번째 값까지 비교합니다. 그럼 배열에서 가장 작은 수가 첫번째 인덱스에 저장됩니다. 그럼 첫번째 인덱스는 제쳐두고, 마지막 인덱스부터 두번째 인덱스까지 비교, swap 하면 두번째로 작은 수가 두번째 인덱스에 저장됩니다. 이러한 과정을 끝까지 반복하여 배열이 정렬되도록 코드를 작성했습니다.

2) Insertion sort

배열의 두번째 값부터 차례로 자신의 앞쪽 인덱스들과 크기를 비교해가며 자신이 위치해야 할 인덱스를 탐색해갑니다. 자신의 값을 temp 변수에 저장해 놓고 앞자리 값들과 비교해서 앞자리 값들이 더 크면 앞자리 값들을 오른쪽으로 한 칸씩 밀어주고 자신이 더 크면 located 를 true 로 바꿈으로써 반복문을 탈출하고 비교대상의 오른쪽에 temp 를 저장합니다. 만약 앞의 모든 비교대상보다 자신이 작다면 비교대상의 인덱스가 < 0 이 되면서 반복문을 탈출하게 되므로 인덱스 0 에 temp 를 저장합니다. 이러한 과정을 끝까지 반복하여 배열이 정렬되도록 코드를 작성했습니다.

3) Merge sort

Merge sort 는 크기가 2 가 될 때까지 배열을 반으로 나누는 과정을 반복하여 정렬 및 합병하는 방법입니다. 크기가 2 가 될 때까지 반으로 나누는 과정이 큰 문제를 작은 문제로 나누어 해결하는 재귀의 상황과 맞아떨어져, 재귀함수로 구현했습니다. 매개변수로 배열의 왼쪽 끝 인덱스, 오른쪽 끝 인덱스를 받음으로써 나뉜 배열의 구간을 알 수 있도록 했습니다. 이 두개의 인덱스 매개변수를 사용하여 재귀의 조건을 ($left < right$)로 주어 배열의 크기가 2 이상일 때만 재귀가 실행되도록 했고, 재귀를 왼쪽, 오른쪽 두 번 실행한 후에는 merge 함수를 사용하여 정렬해줬습니다. 두 개의 배열을 동시에 정렬해야 하므로 동적할당을 통해 두 배열을 합친 크기만큼의 배열 arr 을 할당 받고 두 배열의 가장 앞 인덱스부터 비교해가며 작은 수를 arr 의 첫번째 인덱스부터 차례로 저장했습니다.

4) Quick sort

Quick sort 역시 기준(pivot)을 잡고 pivot 을 기준으로 작은 것은 왼쪽 큰 것은 오른쪽에 위치하도록 나눠서 배열하므로 역시 재귀를 통해 구현했습니다. Merge sort 와 다른 점은, Merge sort 는 배열을 먼저 나누고 정렬했다면 Quick sort 는 배열을 조건에 맞게 정렬한 후 나눈다는 점입니다. Quick sort 에서 정렬하는 방법은 Insertion sort 와 비슷한데, pivot 의 왼쪽 배열에서 pivot 보다 큰 값이 있다면 그 수를 변수 tmp 에 저장해두고 그 수부터 pivot 까지의 값들을 왼쪽으로 한 칸씩 밀어준 후 pivot 의 오른쪽에 tmp 를 저장합니다. 마찬가지로 pivot 의

오른쪽 배열에서 pivot 보다 작은 값이 있다면 그 수를 변수 tmp 에 저장해두고 pivot 부터 그 수까지의 값들을 오른쪽으로 한 칸씩 밀어주고 pivot 의 왼쪽에 tmp 를 저장했습니다.

□ 결과 화면

```
Input: 15
26 9 23 19 1 35 4 31 14 13 28 3 18 33 19

Bubble Sorted order: 1 3 4 9 13 14 18 19 19 23 26 28 31 33 35
Insertion Sorted order: 1 3 4 9 13 14 18 19 19 23 26 28 31 33 35
Merge Sorted order: 1 3 4 9 13 14 18 19 19 23 26 28 31 33 35
Quick Sorted order: 1 3 4 9 13 14 18 19 19 23 26 28 31 33 35
Media number: 19
```

```
Input: 7
1 9 20 239 111 76 77

Bubble Sorted order: 1 9 20 76 77 111 239
Insertion Sorted order: 1 9 20 76 77 111 239
Merge Sorted order: 1 9 20 76 77 111 239
Quick Sorted order: 1 9 20 76 77 111 239
Media number: 76
```

무작위로 홀수 개의 수들을 입력했을 때, 수들이 오름차순으로 크기에 맞게 잘 정렬된 채로 출력되는 모습이다. 그리고 중간값 또한 알맞게 출력된다.

□ 고찰

1) Bubble sort

Bubble sort 는 current 가 0 부터 last-1 까지 반복문을 실행하게 됩니다. 그런데 그 반복문 안에서 walker 가 last 부터 current+1 까지 반복문을 한번 더 실행하는 이중반복문이 됩니다. 이 중 반복문 내에서는 조건에 따라 실행되는 swap 함수가 존재하므로 매 반복마다 3 번의 대입연산이 일어나게 됩니다. 따라서 최악의 경우(예를 들어, 입력 받은 배열이 내림차순 정렬 돼있어서 내부 반복문 실행마다 모두 swap 을 해줘야하는 경우)

Bubble sort 는 $3\{(n-1) + (n-2) + (n-3) + \dots + 3 + 2 + 1\} = 3 \sum_{k=1}^{n-1} k = \frac{3n(n-1)}{2}$ 번의 연산이 실행되므로 Bubble sort 의 시간복잡도 $T(n) = O(n^2)$ 입니다.

※일반적인 경우나 최선의 경우에는 Bubble sort 가 swap 은 실행하지 않더라도 내부반복문을 끝까지 전부 실행하기 때문에 계산복잡도 $T(n) = \Omega(n^2) = \theta(n^2) = O(n^2)$ 입니다.

2) Insertion sort

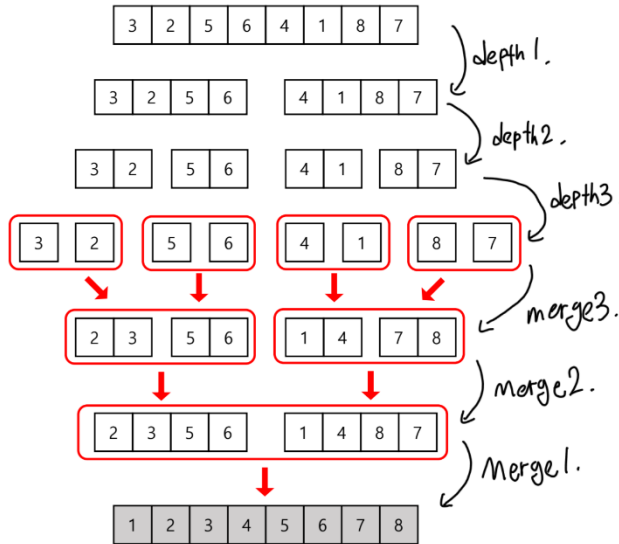
Insertion sort 는 current 가 1 부터 last 까지 반복문을 실행하게 됩니다. 그런데 그 반복문 안에서 walker 가 current-1 부터 0 까지 반복문을 한번 더 실행하는 이중반복문이 됩니다. 하지만 조건($\text{temp} > \text{list}[\text{walker}]$)에 따라 walker 가 0 까지 가지 못한 채 반복문이 끝나는 경우가 있습니다. 반면 $\text{temp} < \text{list}[\text{walker}]$ 인 경우 walker 가 0 이 될 때까지 2 번의 대입연산을 반복적으로 실행합니다. 따라서 최악의 경우(예를 들어, 입력 받은 배열이 내림차순 정렬 돼있어서 모든 내외부 반복문을 실행해야 할 경우)

Insertion sort 는 $2\{1 + 2 + 3 + \dots + (n-3) + (n-2) + (n-1)\} = 2 \sum_{k=1}^{n-1} k = \frac{2n(n-1)}{2}$ 번의 연산이 실행되므로 Insertion sort 의 시간복잡도 $T(n) = O(n^2)$ 입니다.

※일반적인 경우 역시 시간복잡도 $T(n) = \theta(n^2)$ 이고, 최선의 경우(입력 받은 배열이 오름차순 정렬 돼있어서 내부 반복문이 한번의 실행만에 종료될 때)에는 시간복잡도 $T(n) = \Omega(n)$ 입니다.

3) Merge sort

Merge sort 는 크기가 n 인 배열을 크기가 1 이 될 때까지 반으로 나누는 개념으로 재귀를 사용한다. 따라서 재귀가 $\log_2 n$ 만큼의 깊이로 재귀가 호출된다.



그리고 각 깊이에서 정렬되는 시간은 n 으로 같은데 그 이유를 그림의 예로 설명하자면, $n=8$ 일때, merge3 에서 각 파트들이 정렬되는데 총 2 번의 대입연산이 진행되고 파트가 총 4 개 있으므로 $4 \times 2 = 8 = n$, merge2 에서 각 파트들이 정렬되는데 총 4 번의 대입연산이 진행되고 파트가 총 2 개 있으므로 $2 \times 4 = 8 = n$. 이런 식으로 각 깊이에서 정렬되는 시간이 n 이므로 시간복잡도 $T(n) = O(n \log_2 n)$ 이다.


※일반적인 경우나 최선의 경우에도 Merge sort 는 분할과 정렬을 모두 같은 횟수만큼 실행시키므로

계산복잡도 $T(n) = \Omega(n \log_2 n) = \theta(n \log_2 n) = O(n \log_2 n)$ 입니다.

4) Quick sort

Quick sort 의 경우 재귀를 사용한다. 가장 이상적인 경우, pivot 으로 고르는 수들이 여러 수들 중 중간 값을 가진다고 하면 깊이는 $\log_2 n$ 이 된다. 그리고 각 깊이당 총 n 번의 연산을 진행한다.

따라서 시간복잡도 $T(n) = \Omega(n \log_2 n) = \theta(n \log_2 n)$ 이다. 하지만 최악의 경우 pivot 으로 고르는 수마다 가장 작은 값 혹은 가장 큰 값을 가진다고 가정하면 그 깊이는 $\log_2 n$ 가 아니라 n 이 된다. 따라서 각 깊이당 n 번의 연산이 진행되므로 시간복잡도 $T(n) = O(n^2)$ 이다.



시간복잡도를 비교해봤을 때, $T(n) = \Omega(n \log_2 n) = \theta(n \log_2 n) = O(n \log_2 n)$ 인 merge sort 가 가장 효율적이라고 생각할 수 있습니다. 하지만 일반적인 상황에서 quick sort 또한

$T(n) = \theta(n \log_2 n)$ 이므로 두 정렬법을 비교해보겠습니다.

Merge sort 는 두 배열을 새로운 배열에 정렬시키고 정렬시킨 배열을 다시 원래 배열에 저장하므로 n^2 의 연산이 진행됩니다.

Quick sort 는 한번의 연산으로 하나의 수를 정렬할 수 있으므로 n 번의 연산이 진행됩니다. 따라서 일반적인 상황에서는 quick sort 가 월등히 빠른 속도를 가진다고 할 수 있습니다. 또한 공간복잡도 면에서 봐도 quick sort 가 우수합니다.

따라서 수들이 랜덤하게 주어진다고 가정할 때, 최선의 상황과 최악의 상황이 나타날 확률은 매우 낮으므로 quick sort 를 사용하는 것이 좋다고 할 수 있습니다.