EECE372 HW#1

20190445 허수범

March 17, 2024

1 Source Code에 대한 설명

1.1 Flow

전체적인 시스템의 flow는 Figure 1의 flowchart로 시각화할 수 있습니다. 먼저 두 번의 Exception handling을 거치며 커맨드라인 input들을 모두 int 변수에 입력합니다. 이후, 배열을 n의 값에 맞도록 동적으로 할당해 주고, prod_MultofThree 함수를 이용하여 할당해 두었던 배열에 1 이상 n 이하인 3의 배수를 저장해 주었습니다. 'prod_MultofThree' 함수에서는 'srand(time(NULL))'과 for, while loop을 통해 랜덤으로 겹치지 않는 3의 배수를 생성하였습니다. 'prod_MultofThree' 함수의 코드는 Listing 1에서 확인할 수 있는데, while loop을 통해 겹치는 숫자는 피하고, for loop을 통해 배열의 원소 수만큼 숫자를 생성하여 배열에 입력하도록 한 것을 볼 수 있습니다.

```
// produce random multiples of three using srand
void prod_MultofThree(int* arr, int n){
    srand(time(NULL));

    int i;
    for(i = 0; i < n / 3; i++){
        int gen_num = 3 * (rand() % (n / 3) + 1);// generate random number

        int j = 0;
        while(j < i){
            if(arr[j] == gen_num){// check same number
                gen_num = 3 * (rand() % (n / 3) + 1);
                j = -1;// restart loop from beginning
            }
            j++;
        }
        arr[i] = gen_num;// save generated random number in input array
    }
}
```

Listing 1: prod_MultofThree Function.

이후 n이 30 이하인 경우를 대비해 생성된 랜덤 정수 배열을 다른 배열에 복사해 주고, switch문을 통해 sorting 방법에 따른 sorting 함수 호출, 알고리즘 실행 시간 측정, 그리고 실행시간 결과 출력을 하도록 하였습니다. Listing 2에서 해당 code snippet을 확인할 수 있는데, 각 sorting 함수의 호출 앞뒤에 clock

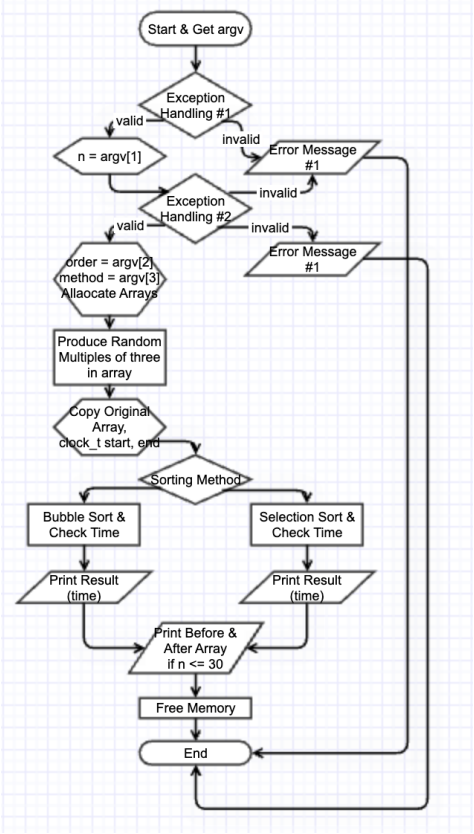


Figure 1: Flowchart.

함수를 호출하여 start, end 시간을 측정하도록 하여 정확한 알고리즘 수행 시간을 측정하였다는 것을 볼 수 있습니다.

이후 만약 n이 30 이하의 정수라면, printer 함수를 호출하여 sorting 이전과 이후의 배열을 출력해 주도록 하였습니다. 여기서 printer 함수는 for loop으로 구현하였습니다. 마지막으로, main함수는 두 배열

```
clock_t start, end;
    switch(method){// sort by input method
        case 0:
            start = clock();
            bubble_sort(arr, n, order);
            end = clock();
            printf("bubble sort: %lf sec\n", (double) (end - start) / CLOCKS_PER_SEC);
            break;
        case 1:
            start = clock();
            selection_sort(arr, n, order);
            end = clock();
            printf("selection sort: %lf sec\n", (double) (end - start) /
    CLOCKS_PER_SEC);
            break;
    }
```

Listing 2: Switch문.

(sorting된 것과 되기 전의 것)에 할당된 메모리를 free()함수를 통해 해제해주고 끝이 납니다.

1.2 Sorting Algorithm 동작 원리

Sorting 함수들은 수업에 사용된 lecture note에 적혀 있는 코드를 토대로 작성되었습니다. 가장 큰 차이점들은 argv[2]의 값에 따라 오름차순과 내림차순이 모두 가능하도록 구현하였다는 점과 swapping하는 것은 함수로 따로 구현을 하여 반복되는 코드를 줄였다는 점입니다.

swap함수는 Listing 3과 같이 구현하였습니다. 두 정수 포인터를 argument로 받아 각 주소에 있는 정수의 값을 서로 바꾸도록 하였습니다.

```
// number swapper
void swap(int* a, int* b){
    int temp = *b;
    *b = *a;
    *a = temp;
}
```

Listing 3: swap Function.

1.2.1 Bubble Sort

Bubble Sort 함수는 Listing 4와 같이 구현하였습니다.

```
// bubble sort
void bubble_sort(int* arr, int n, int order){
    int i, j, swapped = 1;

    for(i = 0; i < n / 3 - 1 && swapped == 1; i++){
        swapped = 0;
        for(j = 0; j < n / 3 - i - 1; j++){
            if(order == 0){
                if(arr[j] > arr[j + 1]){
                    swap(&arr[j], &arr[j + 1]);
                    swapped = 1;
                }
            } else if(order == 1){
                if(arr[j] < arr[j + 1]){
                    swap(&arr[j], &arr[j + 1]);
                    swapped = 1;
                }
            }
        }
    }
}
```

Listing 4: Bubble Sort Function.

여기서 ‘bubble_sort’ 함수는 세 개의 매개변수를 받도록 하였습니다. 첫 번째 매개변수인 ‘arr’은 정렬할 배열, 두 번째 매개변수인 ‘n’은 입력 받은 n의 값, 세 번째 매개변수인 ‘order’는 정렬 순서입니다. 함수 내에서 선언된 변수 ‘i’와 ‘j’는 밑의 이중 for문에서 사용되고, ‘swapped’는 bubble sort의 improved version에서 사용되는, 배열에서 원소의 교환 여부를 나타내는 변수입니다. ‘swapped’의 초기값은 ‘1’로 설정됩니다. ‘swapped’ 변수가 사용되는 이유는, bubble sort 회차 반복을 배열의 길이만큼 해 줄 필요 없이, 원소의 교환이 일어나지 않았다면, 함수를 끝내도록 하여 조금이라도 알고리즘 실행 시간을 줄이기 위해서입니다.

바깥쪽 for loop는 배열의 길이에서 1을 뺀만큼 반복하도록 설정하였습니다. 이는 bubble sort에서 마지막 원소는 자동으로 정렬되기 때문입니다. 또한, 이 loop는 위에서 설명했듯, 원소의 교환이 이루어진

경우, 즉 'swapped'가 '1'인 동안에만 반복됩니다. 바깥쪽 for loop이 시작될 때마다 'swapped'를 '0'으로 초기화합니다.

안쪽 반복문은 배열을 순회하며 인접한 원소들을 비교하여 필요한 경우 교환하도록 구현하였습니다. 여기서 if - else if문을 활용하여 함수의 argument로 입력된 'order'가 '0'인 경우(오름차순)와 '1'인 경우(내림차순)을 나누어 처리하도록 하였습니다. 인접한 두 원소를 비교해 순서가 올바르지 않다면 위에서 설명한 'swap' 함수를 통해 두 원소를 교환하고, 'swapped'를 '1'로 설정하여 원소의 교환이 일어났다는 flag를 세우도록 하였습니다.

1.2.2 Selection Sort

Selection Sort 함수는 Listing 5와 같이 구현하였습니다.

```
// selection sort
void selection_sort(int* arr, int n, int order){
    int i, j, minmax_idx;

    for(i = 0; i < n / 3 - 1; i++){
        minmax_idx = i;

        for(j = i + 1; j < n / 3; j++){
            if(order == 0){
                if(arr[j] < arr[minmax_idx]){
                    minmax_idx = j;
                }
            } else if(order == 1){
                if(arr[j] > arr[minmax_idx]){
                    minmax_idx = j;
                }
            }
        }

        swap(&arr[i], &arr[minmax_idx]);
    }
}
```

Listing 5: Selection Sort Function.

여기서 'selection_sort' 함수는 위의 'bubble_sort' 함수와 동일한 세 개의 매개변수를 받도록 하였습니다. 함수 내에서 선언된 변수 'i'와 'j'는 밑의 이중 for문에서 사용되고, 'minmax_idx'는 현재 반복에서 최솟값 혹은 최댓값이 위치한 인덱스를 저장합니다.

'selection_sort'에서도 역시 마지막 원소는 자동으로 정렬되기 때문에, 바깥쪽 for loop를 첫 번째 원소부터 마지막에서 두 번째 원소까지 반복하도록 하였습니다. 바깥쪽 loop에서는 각 반복마다 최솟값 또는 최댓값의 인덱스를 현재 인덱스로 먼저 초기화합니다. 안쪽 for loop는 현재 위치의 다음 원소부터 배열의 끝까지 반복되며 최소 혹은 최댓값의 인덱스를 찾습니다. 'bubble_sort' 함수와 똑같이, if - else if 문을 통해 오름차순과 내림차순에 대해 다른 처리를 하도록 구현하였습니다. 안쪽 for loop에서 최솟값 혹은 최댓값을 찾는 방법은 현재 원소 또는 이전 반복에서 찾은 최소 혹은 최대값과 비교해 더 작거나 더 큰 값이 있다면 'minmax_idx'에 해당 원소의 인덱스를 업데이트하는 것입니다.

안쪽 for loop가 종료되면, 찾은 최솟값 혹은 최대값을 현재 위치('i')에 있는 값과 교환하도록 하였습니다. 이 때 역시 'swap' 함수를 호출하여 처리하도록 구현하였습니다.

```
[sbh408@programming2 HW1]$ ./HW1_20190445 0 0 0
Error: wrong input type.
[sbh408@programming2 HW1]$ ./HW1_20190445 300001 0 0
Error: wrong input type.
[sbh408@programming2 HW1]$ ./HW1_20190445 300001 2 0
Error: wrong input type.
[sbh408@programming2 HW1]$ ./HW1_20190445 300001 0 2
Error: wrong input type.
[sbh408@programming2 HW1]$ ./HW1_20190445 1 0 0
No multiples of three in range.
[sbh408@programming2 HW1]$ ./HW1_20190445 2 0 0
No multiples of three in range.
```

Figure 2: 예외 처리 메시지 출력 결과 예시.

1.3 n의 입력 방식

n의 입력은 'int main(int argc, char* argv[])'으로 main 함수가 커맨드라인 인자를 받도록 하였습니다. 커맨드라인 인자는 argv[]배열에 string으로 저장되는데, 받은 string은 예외 처리 이후 atoi()함수를 이용하여 int 변수 n에 입력하였습니다.

1.4 Error와 예외 처리

예외 처리 출력 결과의 예시는 Figure 2에서 확인할 수 있습니다. 이 출력 결과들은 main함수에서 두 차례의 예외 처리를 통해 이루어지는데, 이는 밑에서 설명하도록 하겠습니다.

1.4.1 예외 처리 1

Exception Handling #1에서는 총 세가지 조건을 검사하여 처리해 주었습니다. 첫째 조건은 입력 받은 커맨드라인 인자가 총 3개가 맞는지, 둘째 조건은 argv[2]가 "0" 또는 "1"이 맞는지, 마지막 셋째 조건은 argv[3]가 "0" 또는 "1"이 맞는지였습니다. 위 세 조건을 만족하지 않는 argument가 들어왔다 판단되면, "Error: wrong input type.\n" 메세지를 출력하고, 프로그램을 끝내도록 하였습니다. Listing 6에서 확인할 수 있듯이, if문으로 구현하였습니다.

```
1 // exception handling #1
2 if(argc != 4|| (strcmp(argv[2], "0") != 0 && strcmp(argv[2], "1") != 0) || \
3 (strcmp(argv[3], "0") != 0 && strcmp(argv[3], "1") != 0)) {
4     printf("Error: wrong input type.\n");
5     return 1;
6 }
```

Listing 6: Exception Handling #1.

1.4.2 예외 처리 2

Exception Handling #1 에서 valid한 입력이 들어왔다고 판단하였다면, int n에 argv[1]을 입력한 후, Exception Handling #2를 진행하였습니다. 이 때 int n만 먼저 정수 변수에 입력한 것은 Exception Handling #2에서 예외 상황이 발생한 경우 불필요한 메모리 할당을 진행하지 않게 하기 위해서였습니다.

Exception Handling #2에서는 먼저 n에 입력된 정수 값이 인자로 받은 string인 argv[1]와 동일한지와 n의 값이 1 이상 300000이하인지 확인하도록 하였습니다. 위 두 조건을 통과하지 못한 경우 Exception

```
[sbh408@programming2 HW1]$ ./HW1_20190445 30 0 0
bubble sort: 0.000000 sec
before sort: 12 27 6 24 9 21 30 3 18 15
after sort: 3 6 9 12 15 18 21 24 27 30
[sbh408@programming2 HW1]$ ./HW1_20190445 30 1 0
bubble sort: 0.000000 sec
before sort: 18 6 30 15 9 27 3 24 21 12
after sort: 30 27 24 21 18 15 12 9 6 3
[sbh408@programming2 HW1]$ ./HW1_20190445 30 0 1
selection sort: 0.000000 sec
before sort: 15 30 3 9 6 18 21 24 27 12
after sort: 3 6 9 12 15 18 21 24 27 30
[sbh408@programming2 HW1]$ ./HW1_20190445 30 1 1
selection sort: 0.000000 sec
before sort: 27 15 21 3 12 18 9 24 6 30
after sort: 30 27 24 21 18 15 12 9 6 3
```

Figure 3: n이 30일 때 각 상황에 대한 결과.

Handling #1과 같이 “Error: wrong input type.\n” 메세지를 출력하고 프로그램을 종료하도록 하였고, 위 조건을 통과하였다고 하더라도 else if문을 통해 n의 값이 1 혹은 2라면 ”No multiples of three in range.\n” 를 출력하고 프로그램을 종료하도록 하였습니다. 이 과정의 구현은 Listing 7에서 확인할 수 있습니다.

```
// exception handling #2
if(digit_nums(n) != strlen(argv[1]) || n < 1 || n > 300000){
    printf("Error: wrong input type.\n");
    return 1;
} else if(n < 3){
    printf("No multiples of three in range.\n");
    return 1;
}
```

Listing 7: Exception Handling #2.

여기서 n에 할당된 정수 값이 입력 받은 string인 argv[1]와 동일한지 확인하였던 방법은, 정수 변수의 자릿수를 세는 함수인 ‘digit_num’ 함수를 호출하여, 정수 변수의 자릿수와 받은 커맨드라인 인자 string의 글자 수가 같은지 확인하는 것이었습니다. 이 때 string의 자릿수를 확인하기 위해 string.h 라이브러리의 strlen함수를 사용하였습니다. ‘digit_num’ 함수의 구현은 Listing 8에서 확인할 수 있습니다. 재귀함수를 이용하여 정수의 자릿수를 고하도록 구현한 것을 볼 수 있습니다.

```
// get number of digits of an integer recursively
int digit_nums(int n){
    if(n == 0){
        return 0;
    }
    while(n != 0){
        return 1 + digit_nums(n / 10);
    }
    return 0;
}
```

Listing 8: 정수의 자릿수를 구하는 Function.

2 실행 과정과 결과

먼저 n이 30일 때 각 sorting 방법에 대한 오름차순, 내림차순 결과는 Figure 3과 같습니다.

그 외 n이 30 이상일 때의 실행 결과의 예시는 Figure 4에서 확인할 수 있습니다.

```
[sbh408@programming2 HW1]$ ./HW1_20190445 300 0 0
bubble sort: 0.000000 sec
[sbh408@programming2 HW1]$ ./HW1_20190445 3000 0 0
bubble sort: 0.010000 sec
[sbh408@programming2 HW1]$ ./HW1_20190445 3000 1 0
bubble sort: 0.010000 sec
[sbh408@programming2 HW1]$ ./HW1_20190445 3000 1 1
selection sort: 0.000000 sec
[sbh408@programming2 HW1]$ ./HW1_20190445 3000 0 1
selection sort: 0.000000 sec
```

Figure 4: n이 30이상일 때 결과 예시.

```
[sbh408@programming2 HW1]$ bash script_bubble
Average result: 0 seconds for 30 0 0
Average result: 0 seconds for 300 0 0
Average result: .004800 seconds for 3000 0 0
Average result: .295200 seconds for 30000 0 0
Average result: 31.711500 seconds for 300000 0 0
[sbh408@programming2 HW1]$ bash script_selection
Average result: 0 seconds for 30 0 1
Average result: 0 seconds for 300 0 1
Average result: .000700 seconds for 3000 0 1
Average result: .102400 seconds for 30000 0 1
Average result: 10.205500 seconds for 300000 0 1
```

(a) 오름차순

```
[sbh408@programming2 HW1]$ bash script_bubble
Average result: 0 seconds for 30 1 0
Average result: 0 seconds for 300 1 0
Average result: .005400 seconds for 3000 1 0
Average result: .297400 seconds for 30000 1 0
Average result: 32.089500 seconds for 300000 1 0
[sbh408@programming2 HW1]$ bash script_selection
Average result: 0 seconds for 30 1 1
Average result: 0 seconds for 300 1 1
Average result: .001400 seconds for 3000 1 1
Average result: .114200 seconds for 30000 1 1
Average result: 11.275500 seconds for 300000 1 1
```

(b) 내림차순

Figure 5: 알고리즘 수행시간 출력 결과.

두 경우 모두 HW#1 pdf에서 명시한 출력 형태를 따르고 있는 것을 확인할 수 있습니다. sorting 함수가 수행되는 시간을 초 단위로 소수점 아래 6자리까지 출력하였으며, n이 30 이하인 경우 sorting 전 / 후의 sequence를 화면에 출력한 것을 볼 수 있습니다. 또한, n이 30인 경우의 출력 결과를 보면, 숫자들이 겹치지 않고 생성되었으며, 매번 생성시마다 무작위로 생성되었고, sorting 역시 정상적으로 되었다는 것을 확인할 수 있습니다.

3 알고리즘 수행시간 계산과 분석

알고리즘 수행 시간을 여러번 측정하여 평균 내기 위해 Linux Shell Script를 작성하여 프로그램을 반복하여 실행하고, 결과의 평균 값을 출력하도록 하였습니다. Listing 9는 밑의 결과들을 도출해내기 위한 Shell Script의 일부로, 필요에 따라 수정하여 실행하였습니다. Shell Script를 통한 알고리즘 수행시간 출력 결

```
#!/bin/bash

total_30=0

for ((i=0; i<100; i++))
do
    result=$(./HW1_20190445 30 1 0 | awk '/bubble sort:/ {print $3}')
    total_30=$(echo "scale=6; $total_30 + $result" | bc)
done

average_30=$(echo "scale=6; $total_30 / 100" | bc)

echo "Average result: $average_30 seconds for 30 1 0"
```

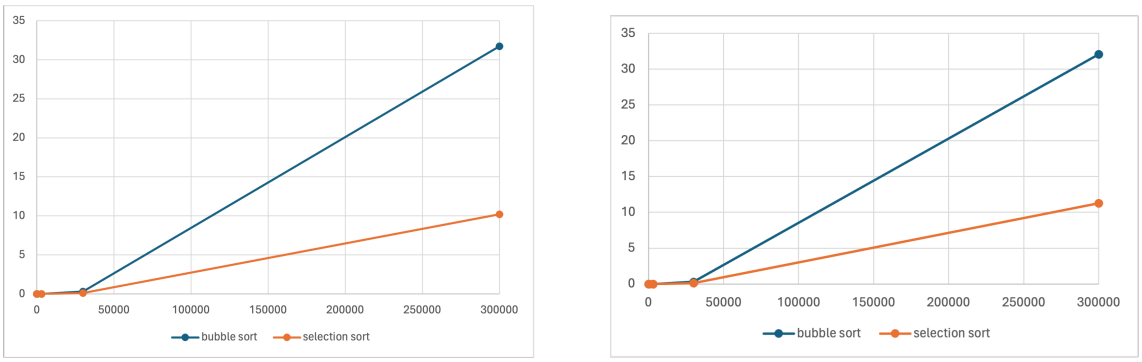
Listing 9: Linux Shell Script.

과는 Figure 5와 같습니다. 그 결과는 Figure 6과 같은 표로 정리하였습니다. Figure 6의 표를 그래프로 plot 것은 Figure 7와 같습니다. 그래프의 개형을 더욱 자세히 확인하기 위해 n의 값을 3000부터 6000

	bubble sort	selection sort
30	0	0
300	0	0
3000	0.0048	0.0007
30000	0.2952	0.1024
300000	31.7115	10.2055

	bubble sort	selection sort
30	0	0
300	0	0
3000	0.0054	0.0014
30000	0.2974	0.1142
300000	32.0895	11.2755

Figure 6: 알고리즘 수행시간 출력 결과 table(왼쪽이 오름차순, 오른쪽이 내림차순).



(a) 오름차순 (b) 내림차순

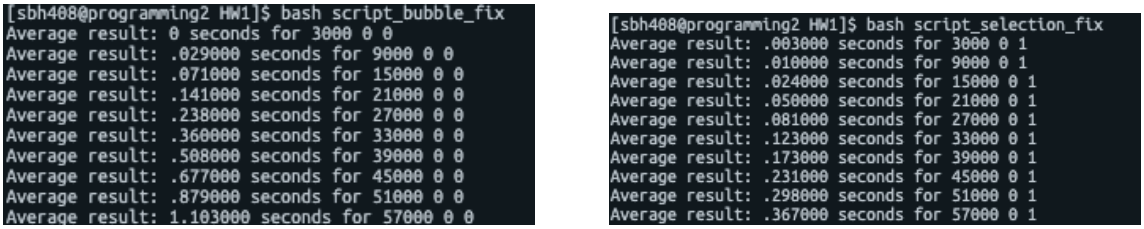
Figure 7: 알고리즘 수행시간 출력 결과 그래프.

식 늘려 57000까지로 설정하여 실행한 결과는 Figure 8와 같습니다. 이 때 정렬 순서는 오름차순 정렬로 하였습니다. 그 결과를 그래프로 plot한 것은 Figure 9와 같습니다.

이러한 결과들로 확인할 수 있는 bubble sort와 selection sort 알고리즘의 특징은 크게 세가지가 있습니다.

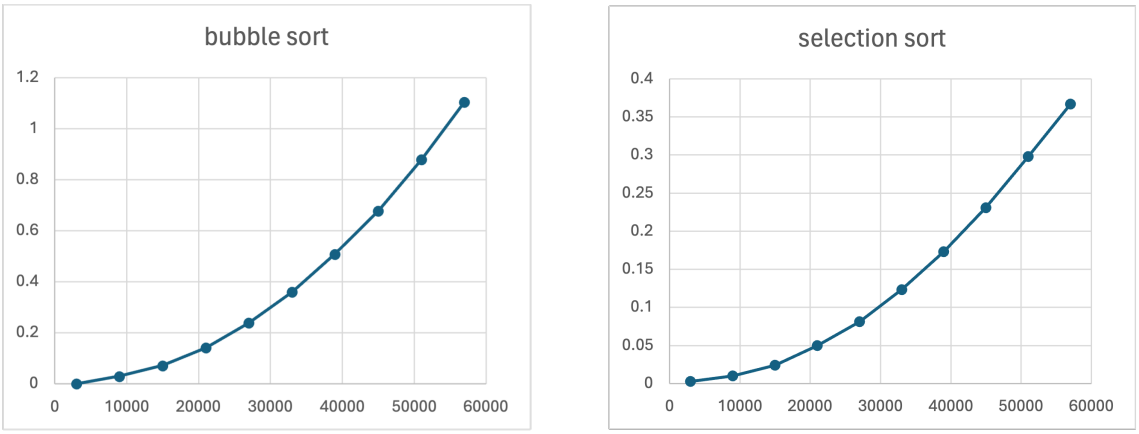
첫째, 수업시간에 다룬 대로, 두 알고리즘 모두 Average case complexity가 $O(n^2)$에 가깝다는 점입니다. 이는 n의 크기 변화에 따른 알고리즘 수행 시간의 변화를 통해 확인할 수 있는데, n의 크기가 작을 때 ($n \leq 3000$)는 정확히 드러나지 않지만, 그보다 클 경우에는 알고리즘 수행 시간이 n의 제곱에 비례하는 경향성을 확실히 파악할 수 있고, HW의 조건을 modify하여 따로 진행한 결과인 Figure 8에서는 더욱 명확히 파악할 수 있습니다. Figure 9에서는 알고리즘 수행시간 plot 결과 그래프의 개형 역시 이차함수와 유사하다는 것을 확인할 수 있습니다.

둘째, n의 크기가 작은 경우, 두 알고리즘이 비슷한 성능을 보이지만, n이 커지자 bubble sort의 improved version을 구현하였음에도 불구하고, selection sort가 더 좋은 성능을 보여준다는 점입니다. 그 이유를 두 가지 관점에서 찾아보도록 하겠습니다. 먼저 비교 횟수입니다. 제가 구현한 대로라면, 비교 횟수는 두 sorting method 간에 큰 차이가 없습니다. 그 이유는 두 method 모두 이중 for loop을 이용하여



(a) Bubble Sort (b) Selection Sort

Figure 8: 알고리즘 수행시간 출력 결과(mod).



(a) Bubble Sort (b) Selection Sort

Figure 9: 알고리즘 수행시간 출력 결과 그래프(mod).

비슷한 횟수만큼 반복해 비교하도록 구현하였기 때문입니다. 오히려 bubble sort가 improved version이기 때문에, 더욱 적은 횟수의 비교를 할 수 있습니다. 하지만 두 번째 관점인 교환 횟수를 통해 두 방식을 비교해 보면, 그 차이는 확연해집니다. selection sort는 각 회차에서 최솟값(혹은 최댓값)을 찾은 후에 한번 원소를 교환합니다. 즉, 최대 $n/3-1$번의 교환만이 이루어지는 것입니다. 평균적으로도 n에 비례하는 횟수의 교환만이 이루어집니다. 하지만 bubble sort의 경우, 인접한 원소가 순서에 맞지 않을 때마다 교환을 수행하기 때문에, n^2에 비례하는 교환 횟수가 필요합니다. 그렇기에, big O notation에 의한 complexity는 같지만, 실제로는 평균적으로 selection sort가 bubble sort에 비해 더 빠른 알고리즘 수행시간을 보여주는 것입니다.

셋째, 두 알고리즘 모두 입력 데이터의 크기가 커질수록 성능이 크게 저하된다는 점입니다. 이는 첫째 발견과 비슷한 맥락의 이야기이지만, 두 알고리즘 모두 시간 복잡도가 $O(n^2)$이기에, 실제로 확인해 보았을 때도 n의 크기가 300000에 다다르면, 알고리즘 수행 속도가 급격히 저하됩니다. 이에 더불어 랜덤 숫자 생성 함수의 시간 복잡도 역시 $O(n^2)$이기에, n이 30000에만 도달해도 프로그램 실행 시간이 매우 오래 걸리는 것을 확인할 수 있었습니다.

HW#1을 통해 크게 위의 세가지 사항을 발견하고, 분석할 수 있었습니다. 과제 수행과 위의 분석을 통해 basic sorting algorithm과 간단한 c programming(포인터, 재귀함수 등의 사용)에 익숙해질 수 있었으며, 알고리즘의 시간 복잡도(big O notation)이 실제 execution과 어떤 차이가 있고, big O notation은 실제 프로그램의 수행 시간에 대한 어떤 인사이트를 주는지 알 수 있었습니다.