알고리즘

- 02. Sort (Merge, Quick) -

※ 과제 목표 및 해결 방법

- Merge, Quick Sort를 구현하여 입출력 파일 방식으로 결과를 도출하여라.

※ 주요 부분 코드 설명 (알고리즘 부분 코드 캡쳐)

```
public static final int insertStart = 4; // variable for insertion start
```

[그림1. Field_variable]

➔ insert_sort를 실행하기 위한 조건이다. 배열의 크기가 4가 되면 더 이상 분할을 하지 않고 insert_sort를 실행한다. 이 변수를 사용했을 경우와 사용하지 않았을 경우에 따라 시간의 차이가 있을 거라고 생각한다.

```
public static int[] Mergesort(int[] Array, int left, int right) {
    if (right - left > insertStart) {
        int mid = (left + right) / 2;
        Mergesort(Array, left, mid);
        Mergesort(Array, mid + 1, right);
        Merge(Array, left, mid, right);
    }
    else {
        insertionSort(Array, left, right);
    }
    return Array;
}
```

[그림2. Mergesort]

➔ Mergersort를 실행하는 재귀로 실행하는 함수이다. 인자로는 배열과 왼쪽 오른쪽의 인덱스 값을 받는다.
만약, (right – left)의 값이 insertStart변수보다 크면 분할을 하고 그렇지 않으면 insert_sort를 실행한다.
양쪽 인덱스의 값을 더한 값에 2를 나눈 값으로 mid(중간값)을 구해준다. 분할을 위해서 인자값을 (배열, 왼쪽, 중간값)으로 중간값을 기준으로 왼쪽을 분할하고 (배열, 중간값+1, 오른쪽)으로 중간값을 기준으로 오른쪽을 분할한다.
Merge함수는 분할이 완료되면 다시 정렬된 배열을 합치기 위한 함수이다.

```
public static void insertionSort(int Array[], int left, int right) {
    for (int i = left; i < right; i++) {
        int tempVal = Array[i + 1];
        int j = i + 1;
        while (j > left && Array[j - 1] > tempVal) {
            Array[j] = Array[j - 1];
            j--;
        }
        Array[j] = tempVal;
    }
}
```

[그림3. insertionSort]

➔ insertsort를 실행하는 함수로 이전의 Mergesort함수에서 제한된 배열의 크기가 도달하면 실행하는 함수이다.
입력받은 인자값부터 반복문을 실행해서 Array[I+1]의 값을 저장한다. 비교를 하면서 저장한 값이 작다면 자리를 바꾸어 준다. 그렇게 비교를 진행해서 j의 값이 left와 같아지면 while문을 종료하고 저장한 값을 Array[j]에 저장한다.

```java
public static void Merge(int[] Array, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int[] LA = Arrays.copyOfRange(Array, left, mid +1);
    int[] RA = Arrays.copyOfRange(Array, mid+1, right +1);
    int RIDX = 0;
    int LIDX = 0;

    for (int i = left; i < right - left + 1; i++) {
        if (RIDX == n2) {
            Array[i] = LA[LIDX];
            LIDX++;
        } else if (LIDX == n1) {
            Array[i] = RA[RIDX];
            RIDX++;
        } else if (RA[RIDX] > LA[LIDX]) {
            Array[i] = LA[LIDX];
            LIDX++;
        } else {
            Array[i] = RA[RIDX];
            RIDX++;
        }
    }
}
```

[그림4. Merge]

➔ Merge함수는 분할된 후 정렬을 마친 배열을 합쳐주는 함수이다. Arrays의 클래스를 사용해서 지정한 범위를 복사하여 배열에 저장을 하였다. 아래 반복문의 과정을 통해서 하나의 배열로 도출한다. 최종적으로 나눠진 배열이 4개이고 정렬이 되어있을 것이다. 이제 이 배열을 합쳐야한다. 우선 각 배열의 작은 값이 앞에 위치하였기 때문에 앞에 있는 값끼리 비교하면서 누가 더 작냐 판단하면 된다.

※ 실행 결과(시간 복잡도 포함)

```
Select File name...
1. test_100.txt
2. test_1000.txt
3. exit
Input : 1

Hey!! File read finish...
TIME : 1.030258(ms)
Hey!! File save finish...

Select File name...
1. test_100.txt
2. test_1000.txt
3. exit
Input : 2

Hey!! File read finish...
TIME : 0.259414(ms)
Hey!! File save finish...

Select File name...
1. test_100.txt
2. test_1000.txt
3. exit
Input : 3

Program terminate
```

[그림5. insertStart = 4]

```
Select File name...
1. test_100.txt
2. test_1000.txt
3. exit
Input : 1

Hey!! File read finish...
TIME : 1.137208(ms)
Hey!! File save finish...

Select File name...
1. test_100.txt
2. test_1000.txt
3. exit
Input : 2

Hey!! File read finish...
TIME : 0.392533(ms)
Hey!! File save finish...

Select File name...
1. test_100.txt
2. test_1000.txt
3. exit
Input : 3

Program terminate
```

[그림6. insertStart = 7]

```
Select File name...
1. test_100.txt
2. test_1000.txt
3. exit
Input : 1

Hey!! File read finish...
TIME : 1.53088(ms)
Hey!! File save finish...

Select File name...
1. test_100.txt
2. test_1000.txt
3. exit
Input : 2

Hey!! File read finish...
TIME : 0.360107(ms)
Hey!! File save finish...

Select File name...
1. test_100.txt
2. test_1000.txt
3. exit
Input : 3

Program terminate
```

[그림7. insertStart = 15]

insertStart의 값을 변화를 주면서 실행을 했는데 데이터의 크기가 적어서 그런지 변수에 따라서 시간의 차이는 크게는 없지만 변수의 값이 커질수록 시간이 좀 더 걸리는 것을 알 수 있다.

```java
public static int[] Quicksort(int[] Array, int start, int end) {
    if(start < end) {
        int p = partition(Array, start, end);
        Quicksort(Array, start, p-1);
        Quicksort(Array, p+1, end);
    }
    return Array;
}
```

[그림8. Quicksort]

➔ 기본적인 Quicksort를 실행하는 함수이다. start의 값이 end값보다 작을 경우 partition함수를 실행해서 pivot을 설정한다. 설정한 pivot을 사용해서 pivot 앞에는 pivot보다 작은 모든 원소들이 오고, pivot보다 큰 원소들은 pivot 뒤에 오도록 둘로 나눈다. 분할된 두 개의 배열에 대해서 재귀적으로 반복한다. 정렬이 완료되면 배열을 반환한다.

```java
public static int partition(int[] Array, int start, int end) {
    int pivot = Array[end];
    int i = start-1;

    for(int j=start; j < end; j++) {
        if(Array[j] <= pivot) {
            i++;
            int tmp = Array[i];
            Array[i] = Array[j];
            Array[j] = tmp;
        }
    }
    int tmp = Array[i+1];
    Array[i+1] = Array[end];
    Array[end] = tmp;
    return i+1;
}
```

[그림9. partition]

➔ partition함수는 pivot을 정해주는 함수이다. 이 함수에서는 pivot을 배열의 끝 위치의 값을 지정하였다. pivot보다 작은 값을 왼쪽, 큰 값을 오른쪽으로 나누어 분할한다. 반복문을 통해서, 만약 j 인덱스 배열의 값이 pivot보다 작다면 i 인덱스을 1증가시킨다. I 인덱스에 위치한 배열의 값을 저장하고 j 인덱스에 있는 값을 I 인덱스에 위치한 배열로 Swap해준다. 반복문이 종료되고 나면 아래의 코드에 명시되어 있든이 오른쪽의 pivot보다 큰 값중 제일 앞 인덱스와 pivot의 값을 교체한 후에 pivot의 인덱스 값을 반환하고 다시 Quicksort를 재귀호출한다.

```java
public static int[] Quicksort_useRandom(int[] Array, int start, int end) {
    if(start < end) {
        int p = randomizePartition(Array, start, end);
        Quicksort(Array, start, p-1);
        Quicksort(Array, p+1, end);
    }
    return Array;
}
```

[그림10. Quicksort_useRandom]

➔ 기본적인 Quicksort와 다르게 pivot의 값을 랜덤으로 주어져서 sort를 진행하는 함수이다. 조건은 이전과 동일한 대신에 분할의 기준을 정해주는 부분이 randomizePartition 함수를 통해서 분할된다.

```java
public static int randomizePartition(int[] Array, int start, int end) {
    Random random = new Random();
    int i = random.nextInt(end - start + 1) + start;
    int pivot = Array[i];
    Array[i] = Array[end];
    Array[end] = pivot;

    return partition(Array, start, end);
}
```

[그림11. randomizePartition]

➔ randomizePartition함수는 pivot을 정해주는 함수이다. 이전의 partition함수와 비슷하지만 pivot의 값을 랜덤으로 주어지는 인덱스의 값을 이용해서 분할한다

※ 실행 결과(시간 복잡도 포함)

```
Select File name...
1. test_100.txt
2. test_1000.txt
3. exit
Input : 1

Hey!! File read finish...
TIME(use Random) : 1.299911(ms)
Hey!! File save finish...

Hey!! File read finish...
TIME(not use Random) : 0.438045(ms)
Hey!! File save finish...

Select File name...
1. test_100.txt
2. test_1000.txt
3. exit
Input : 2

Hey!! File read finish...
TIME(use Random) : 0.499485(ms)
Hey!! File save finish...

Hey!! File read finish...
TIME(not use Random) : 0.180338(ms)
Hey!! File save finish...

Select File name...
1. test_100.txt
2. test_1000.txt
3. exit
Input : 3

Program terminate
```

[그림12. 첫 번째 실행]

```
Select File name...
1. test_100.txt
2. test_1000.txt
3. exit
Input : 1

Hey!! File read finish...
TIME(use Random) : 1.598578(ms)
Hey!! File save finish...

Hey!! File read finish...
TIME(not use Random) : 0.410168(ms)
Hey!! File save finish...

Select File name...
1. test_100.txt
2. test_1000.txt
3. exit
Input : 2

Hey!! File read finish...
TIME(use Random) : 0.302649(ms)
Hey!! File save finish...

Hey!! File read finish...
TIME(not use Random) : 0.10752(ms)
Hey!! File save finish...

Select File name...
1. test_100.txt
2. test_1000.txt
3. exit
Input : 3

Program terminate
```

[그림13. 두 번째 실행]

```
Select File name...
1. test_100.txt
2. test_1000.txt
3. exit
Input : 1

Hey!! File read finish...
TIME(use Random) : 6.005759(ms)
Hey!! File save finish...

Hey!! File read finish...
TIME(not use Random) : 0.935822(ms)
Hey!! File save finish...

Select File name...
1. test_100.txt
2. test_1000.txt
3. exit
Input : 2

Hey!! File read finish...
TIME(use Random) : 0.575147(ms)
Hey!! File save finish...

Hey!! File read finish...
TIME(not use Random) : 0.150187(ms)
Hey!! File save finish...

Select File name...
1. test_100.txt
2. test_1000.txt
3. exit
Input : 3

Program terminate
```

[그림14. 세 번째 실행]

3번의 실행을 한 결과 대체적으로 Random값으로 pivot을 설정하지 않았을 경우가 시간이 적게 걸렸다.
마지막의 100개의 데이터를 가지고 정렬한 실행결과를 보면 첫 번째 실행했던 시간보다 5배의 시간이 걸린 것을 확인 할 수 있는데 이 결과를 통해서 pivot의 값에 따라서 정렬 시간 크게 차이난다라는 결과를 도출할 수 있었다.