알고리즘

- 03. Sort (Max-Min Heap, Counting) -

※ 과제 목표 및 해결 방법

- 배열로 구현.
- Max-Min Heap, Counting Sort를 구현하여 입출력 파일 방식으로 결과를 도출하여라.

※ 주요 부분 코드 설명 (알고리즘 부분 코드 캡쳐)

1. minHeap

```
8●      public minHeap(int[] arr) {
9           this.arr = arr;
10          this.size = arr.length;
11      }
```

[그림1. 생성자]

➔ 생성자로서 인자로 받은 값으로 배열을 생성한다.

```
22●     public static int[] heapSort(int[] arr) {
23          int length = arr.length;
24          build_recu_min_heap(arr, 0, length);
25          for (int i = length-1; i >= 1; i--) {
26              swap(arr, 0, i);
27              length -= 1;
28              heapify(arr, 0, length);
29          }
30          return arr;
31      }
```

[그림2. heapSort]

➔ heap 정렬을 하기 위해서 heap을 생성한다. 생성된 heap은 루트에 가장 작은 원소가 위치했을 것이다. 이 원소를 heap의 맨 마지막 위치시키고 배열의 크기를 1 감소 시켜서 정렬된 원소에 영향을 미치지 않도록 한다.

```
13      // n: 히프의 원소 개수, i는 마지막 내부 노드 인덱스
14●     public static void build_recu_min_heap(int[] arr, int i, int n) {
15          if(i >= n/2) // 리프노드이면
16              return;
17          build_recu_min_heap(arr, 2*i+1, n);
18          build_recu_min_heap(arr, 2*i+2, n);
19          heapify(arr,i,n);
20      }
```

[그림3. build_recu_min_heap]

➔ minHeap을 만들기 위한 과정이다. 재귀로 실행하여 왼쪽 자식과 오른쪽 자식에 대하여 heap을 구성하고 heapify로 가장 작은 숫자가 루트에 위치하도록 한다. 종료 조건은 리프 노드이면 함수를 종료한다.

※ 왼쪽 자식 : 2 * I + 1
※ 오른쪽 자식 : 2 * I + 2

```
33    public static void heapify(int[] arr, int i, int n) {
34        int temp = arr[i];
35        while(i < n/2) // 리프가 아닌 동안
36        {
37            int j = 2*i +1; // 왼쪽 자식
38            if(j + 1 < n && arr[j +1] < arr[j])
39                ++j;
40            if(arr[j] > temp) // 왼쪽 자식이 부모보다 크면
41                break;
42            arr[i] = arr[j]; // 왼쪽 자식을 부모노드로
43            i = j;
44            arr[i] = temp; // 값 바꿔주기
45        }
46    }
```

[그림4. heapify(min)]

➔ heapify 함수는 생성된 heap에 대해서 가장 작은 원소가 루트에 위치하도록 해주는 함수이다.
자식들과 비교하여 자식이 부모보다 작으면 부모 노드와 값을 바꿔주는 형식으로 진행해서 리프가 아닐 때까지 반복한다. 이 함수가 종료되면 맨 위에 가장 작은 원소가 위치한다.

```
48    public static void swap(int arr[], int i, int j) {
49        int temp = arr[i];
50        arr[i] = arr[j];
51        arr[j] = temp;
52    }
```

[그림5. swap]

➔ 배열의 값을 교환해주는 함수이므로 루트에 위치한 값과 맨 마지막에 위치한 값을 바꿔준다. 이 과정을 통해서 배열이 오름차순으로 정렬된다.

2. maxHeap

```
8     public maxHeap(int[] arr) {
9         this.arr = arr;
10        this.size = arr.length;
11    }
```

[그림1. 생성자]

➔ 생성자로서 인자로 받은 값으로 배열을 생성한다.

```
22    public static int[] heapSort(int[] arr) {
23        int length = arr.length;
24        build_recu_max_heap(arr, 0, length);
25        for (int i = length-1; i >= 1; i--) {
26            swap(arr, 0, i);
27            length-=1;
28            heapify(arr, 0, length);
29        }
30        return arr;
31    }
```

[그림2. heapSort]

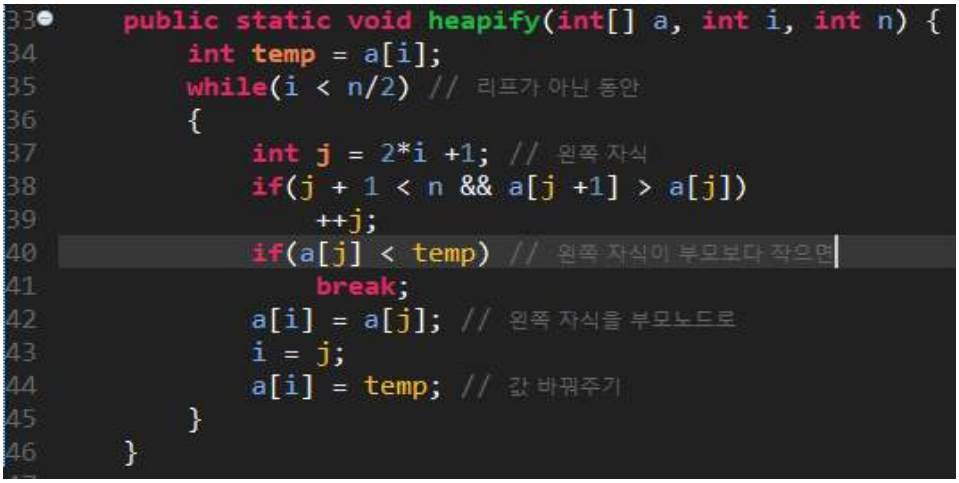
➔ heap 정렬을 하기 위해서 heap을 생성한다. 생성된 heap은 루트에 가장 큰 원소가 위치했을 것이다. 이 원소를 heap의 맨 마지막 위치시키고 배열의 크기를 1 감소 시켜서 정렬 된 원소에 영향을 미치지 않도록 한다.

```
    // n: 히프의 원소 개수, i는 마지막 내부 노드 인덱스
    public static void build_recu_max_heap(int[] a, int i, int n) {
15      if(i >= n/2) // 리프노드이면
16          return;
17      build_recu_max_heap(a, 2*i+1, n);
18      build_recu_max_heap(a, 2*i+2, n);
19      heapify(a,i,n);
20  }
```

[그림3. build_recu_max_heap]

➔ maxHeap을 만들기 위한 과정이다. 재귀로 실행하여 왼쪽 자식과 오른쪽 자식에 대하여 heap을 구성하고 heapify로 가장 큰 숫자가 루트에 위치하도록 한다. 종료 조건은 리프 노드이면 함수를 종료한다.

※ 왼쪽 자식 : 2 * I + 1
※ 오른쪽 자식 : 2 * I + 2



```
33    public static void heapify(int[] a, int i, int n) {
34        int temp = a[i];
35        while(i < n/2) // 리프가 아닌 동안
36        {
37            int j = 2*i +1; // 왼쪽 자식
38            if(j + 1 < n && a[j +1] > a[j])
39                ++j;
40            if(a[j] < temp) // 왼쪽 자식이 부모보다 작으면
41                break;
42            a[i] = a[j]; // 왼쪽 자식을 부모노드로
43            i = j;
44            a[i] = temp; // 값 바꿔주기
45        }
46    }
```

[그림4. heapify(max)]

➔ heapify 함수는 생성된 heap에 대해서 가장 큰 원소가 루트에 위치하도록 해주는 함수이다.
자식들과 비교하여 자식이 부모보다 크면 부모 노드와 값을 바꿔주는 형식으로 진행해서 리프가 아닐 때까지 반복한다. 이 함수가 종료되면 맨 위에 가장 큰 원소가 위치한다.

```
48    public static void swap(int arr[], int i, int j) {
49        int temp = arr[i];
50        arr[i] = arr[j];
51        arr[j] = temp;
52    }
```

[그림5. swap]

➔ 배열의 값을 교환해주는 함수이므로 루트에 위치한 값과 맨 마지막에 위치한 값을 바꿔준다. 이 과정을 통해서 배열이 내림차순으로 정렬된다.

3. Counting

```java
public static int[] countingSort(int[] arr) {
    int maxValue = 0;
    int length = arr.length;

    /* search maxValue in Not Arranged array*/
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] > maxValue)
            maxValue = arr[i];
    }

    /* create array for counting */
    int[] countArr = new int[maxValue+1];

    /* counting */
    for (int i = 0; i < length; i++) {
        countArr[arr[i]]++;
    }

    for (int i = 1; i < countArr.length; i++) {
        countArr[i] += countArr[i-1];
    }

    /* create array for sort */
    int[] sortArr = new int[length];

    /* sort start */
    for (int i = 0; i < sortArr.length; i++) {
        int index = countArr[arr[i]]; // search index
        sortArr[index-1] = arr[i]; // insert in sort Array
        countArr[arr[i]]--; // minus count
    }
    return sortArr;
}
```

[그림1. countingSort]

➔ 계수정렬은 숫자가 등장한 횟수를 카운트하여 그 기준으로 정렬하는 방법이다. 말 그대로 계수를 해서 정렬하는 것이다. 먼저 정렬되지 않은 배열에서 값이 가장 큰 값을 찾는다. 찾은 값이 배열의 마지막인덱스를 의미하기 때문에 1을 더해서 Count 배열을 생성해준다. 생성한 Count 배열에 정렬되지 않은 배열를 인덱스로 사용하여 Counting을 한다. 카운팅한 배열을 반복문을 실행하여 0부터 Count한 값을 누적하여 더해준다.
마지막으로 정렬한 값을 저장할 배열을 생성한다. Count 배열에서 정렬할 배열의 값으로 index를 찾는다. 정렬할 값을 sortArr 배열의 index-1 위치에 저장한다. 저장한 후 count 배열에서는 해당 인덱스에 위치한 값을 1 감소시킨다. 이 과정을 마치면 오름차순으로 값이 정렬된다. 정렬된 배열을 반환해준다.

※ 실행 결과(시간 복잡도 포함)

Heap, counting 정렬의 결과를 분석한 결과

100개의 데이터 - minHeap, maxHeap의 정렬 시간은 거의 동일.

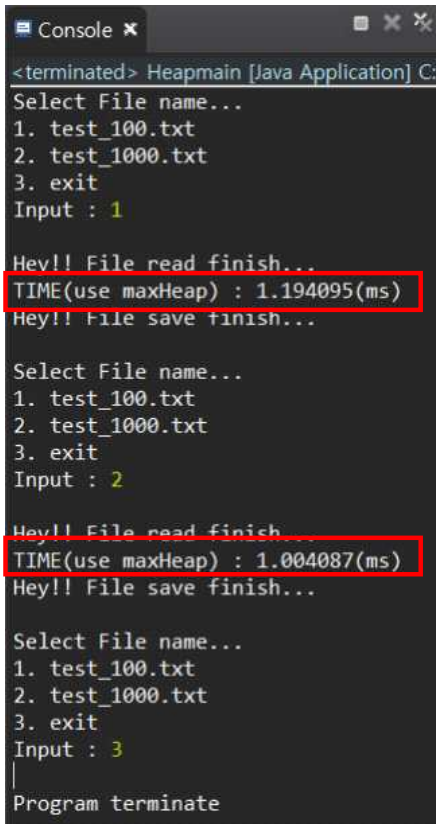
1000개의 데이터 – maxHeap이 minHeap의 정렬 시간보다 조금 빠르다는 것을 알 수 있음.

하지만 이 둘의 함수의 차이는 루트의 위치하는 값이 가장 작은 값이냐, 큰 값이냐에 따른 차이점이기 때문에 정렬 시간에 대해서는 크게 차이가 없는 것으로 생각한다.

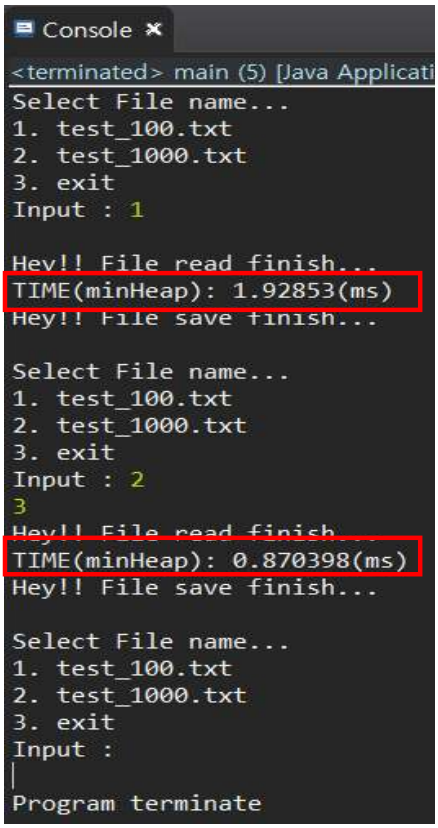
100개의 데이터 – counting 정렬의 시간은 Heap정렬보다 2배의 시간이 걸렸다.

1000개의 데이터 – counting 정렬의 시간은 Heap정렬보다 적은 시간이 걸렸다.

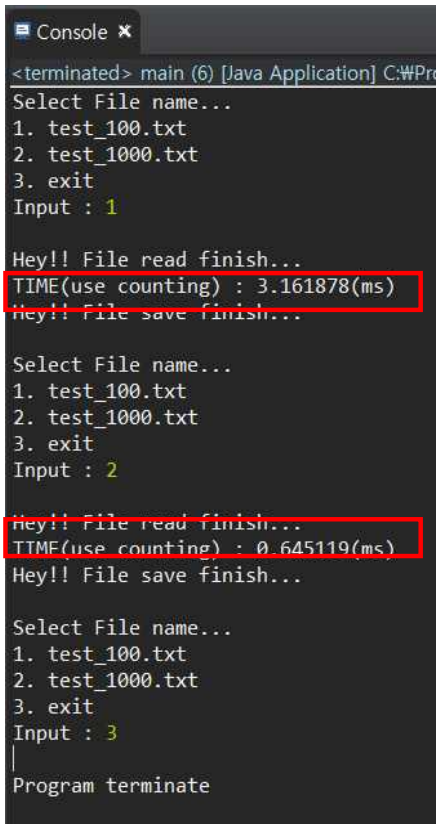
Heap은 O(nlogn)의 시간복잡도를 가지지만 counting은 O(n)의 시간복잡도를 가지기 때문에 Heap보다는 빠르다는 결과를 도출할 수 있다.



[그림5. maxHeap_result]



[그림6. minHeap_result]



[그림7. counting_result]