자료구조 및 실습
보고서

[제11주] DS_11_201502273_김현종

2018.06.01.

201502273/김현종

1.내용

```
import java.io.*;
import java.nio.file.Paths;
public class IO_Manager_201502273 {
    BufferedReader br;
    String filename = Paths.get("src","Random", "1000000.txt").toString(); //데이터 경로
    StringBuffer sb;
    String readData() {
        try {
            br = new BufferedReader(new FileReader(filename));          //파일 읽어오기
            sb = new StringBuffer();
            int i;
            while ((i = br.read()) != -1) {     //읽은 데이터가 있으면
                sb.append((char)i);  //StringBuffer에 append
            }
        } catch (IOException e) {    //IOErrorHandler
            e.printStackTrace();
        }
        return sb.toString();          //만든 String 반환
    }
}
```

IO_Manager_201502273.java

```
public class Sort_201502273 {
    private int[] dataSet;   //데이터가 들어갈 어레이
    private int[] bucket;    //임시 저장 어레이
    private int size;        //데이터의 길이(크기)

    void initDataSet(String dataSet) {          //dataSet을 받은값으로 초기화하는 메소드
        String dataset[] = dataSet.split(" ");  //공백을 기준으로 String을 잘라 String Array를
만든다.
        this.size = dataset.length;   //어레이의 길이가 size이다.
        this.dataSet = new int[this.size];      //size만큼의 길이를 갖는 int array 생성
        this.bucket = new int[this.size];       //size만큼의 길이를 갖는 int array 생성
        for (int i = 0; i < this.size; i++) {     //size만큼 순회하며
            this.dataSet[i] = Integer.parseInt(dataset[i]);         //Sting을 int로 Parce하여
저장
        }
    }

    int getSize() { //size의 getter
        return this.size;
    }

    void bubbleSort() {     //버블정렬을 수행하는 메소드
        //TODO
        for(int i = 0; i < this.size; i++) {     //size만큼 반복하며
            for(int j = this.size - 1; j > i; j--) {//정렬되지 않은 부분만 실행
                if(this.dataSet[j] < this.dataSet[j - 1]) {//현재값이 측 값보다 크면
                    this.swap(j, j - 1);//현재값과 좌측값의 위치를 바꾼다.
                }
            }
        }
    }

    void insertionSort() {  //삽입정렬을 수행하는 메소드
        //TODO
        for(int j = 1; j < this.size; j++) {//size - 1만큼 수행하며
            int key = dataSet[j];         //움직일 값을 설정

            int i = j - 1;       //움직일 값의 좌측 index
            while(i >= 0 && dataSet[i] > key) { //index가 0혹은 양수이거나 해당
정렬된 위치의 값이 key값보다 크면 반복
                dataSet[i + 1] = dataSet[i]; //key값의 위치를 덮어 씌우며 Right
Shift
                i--;       //index = index - 1;
            }
            dataSet[i + 1] = key;        //처음으로 정렬된 곳의 값이 더 큰곳의 우측에,
혹은 큰값이 없으면 0번 위치에 key값 저장
```

```
        }
    }

    void mergeSort(int begin, int end) {      //merge sort를 수행하는 메소드
        //TODO
        if(end - begin < 2) return;//남은 Element가 1개이면 함수 종료

        int middle = (begin + end)/2;        //가운데 인덱스 추출

        mergeSort(begin, middle); //처음부터 중간까지를 재귀적 호출
        mergeSort(middle, end);              //중간부터 끝까지를 재귀적 호출
        merge(begin, middle, end);//나누어진 조각을 합치며 정렬
        copyArray(begin, end);     //임시 어레이에서 원래 어레이로 어레이 복사
    }

    private void merge(int begin, int middle, int end) {        //정렬을 하며 합병하는 함수
        //TODO
        int i = begin, j = middle;  //합병하려는 두 덩어리 내부의 index
        for(int k = begin; k < end; k++) {  //begin부터 end까지 순회
                /*
                 * 두 덩어리에 있는 값을 크기순으로 bucket에 저장
                 * 작은값부터 bucket에 넣는다.
                 */
                if(i < middle && (j >= end || dataSet[i] <= dataSet[j])) {
                        bucket[k] = dataSet[i];
                        i++;
                } else {
                        bucket[k] = dataSet[j];
                        j++;
                }
        }
    }

    private void copyArray(int begin, int end) {       //어레이를 복사하는 메소드
        System.arraycopy(bucket, begin, dataSet, begin, end - begin);//어레이 복사
    }

    void quickSort(int p, int r) {    //quick sort를 수행하는 메소드
        //TODO
        if(p < r) {          //파티션 첫 인덱스(피봇)보다 파티션 마지막 인덱스가 크면
                int q = partition(p, r);      //r위치의 값을 정렬한 후 해당 인덱스를
받아온다.
                quickSort(p, q - 1);          //q값을 기준으로 좌측 파티션을 재귀로 돌린다.
                quickSort(q + 1, r);          //q값을 기준으로 우측 파티션을 재귀로 돌린다.
        }
    }

    int partition(int p, int r) {
        //TODO
        int x = dataSet[r];           //r인덱스위치의 값을 저장
        int i = p - 1;
        for(int j = p; j < r; j++) {  //p에서 r 전까지 돌며
                if(dataSet[j] < x) {//j위치의 값이 r위치의 값보다 작으면
                        i++;
                        this.swap(i, j);     //i위치의 값과 j위치의 값을 swap
                }
        }         //r의 값보다 작은 값들은 i + 1기준 좌측에, 같거나 큰 값들은 우측에 두게
해준다.
        this.swap(i + 1, r);          //r위치의 값보다 작은 값들의 마지막 인덱스 우측의 값과
r위치의 값을 바꾼다.
        return i + 1;      //r위치에 있던 값이 있는 인덱스 반환, 정렬 완료된 인덱스 반환
    }

    void swap(int index1, int index2) {       //두 인덱스에 있는 값을 서로 바꾸는 함수
        int temp = dataSet[index1]; //임시값에 저장
        dataSet[index1] = dataSet[index2];   //index1의 값을 index2의 값으로 덮어 씌우기
        dataSet[index2] = temp;      //임시값으로 index2의 값을 덮어 씌우기
    }
}
```

Sort_201502273.java

```java
public class MainClass_11_201502273 {
    public static void main(String args[]){
        IO_Manager_201502273 io = new IO_Manager_201502273();
        Sort_201502273 sort = new Sort_201502273();
        long start, end;
        System.out.println("Start to Compare Each Sorting Methods Performance");
        System.out.println("================BubbleSort_O(n^2)================");
        sort.initDataSet(io.readData());
        start = System.currentTimeMillis();
        sort.bubbleSort();
        end = System.currentTimeMillis();
        System.out.println("Taken Time(ms) : " + (end-start));

        System.out.println("================InsertionSort_O(n^2)================");
        sort.initDataSet(io.readData());
        start = System.currentTimeMillis();
        sort.insertionSort();
        end = System.currentTimeMillis();
        System.out.println("Taken Time(ms) : " + (end-start));

        System.out.println("================MergeSort_O(nlog2(n))================");
        sort.initDataSet(io.readData());
        start = System.currentTimeMillis();
        sort.mergeSort(0,sort.getSize());
        end = System.currentTimeMillis();
        System.out.println("Taken Time(ms) : " + (end-start));

        System.out.println("================QuickSort_O(nlog2(n))================");
        sort.initDataSet(io.readData());
        start = System.currentTimeMillis();
        sort.quickSort(0, sort.getSize()-1);
        end = System.currentTimeMillis();
        System.out.println("Taken Time(ms) : " + (end-start));
    }
}
```

MainClass_11_201502273.java

2.결과

```
Start to Compare Each Sorting Methods Performance
================BubbleSort_O(n^2)===============
Taken Time(ms) : 1385957
================InsertionSort_O(n^2)================
Taken Time(ms) : 360909
================MergeSort_O(nlog2(n))================
Taken Time(ms) : 133
================QuickSort_O(nlog2(n))================
Taken Time(ms) : 87


Start to Compare Each Sorting Methods Performance
input : 5 1 2 8 9 7 4 6 3 10 15 50 30 22 34 52
================BubbleSort_O(n^2)================
Taken Time(ms) : 0
1 2 3 4 5 6 7 8 9 10 15 22 30 34 50 52
================InsertionSort_O(n^2)================
Taken Time(ms) : 0
1 2 3 4 5 6 7 8 9 10 15 22 30 34 50 52
================MergeSort_O(nlog2(n))================
Taken Time(ms) : 0
1 2 3 4 5 6 7 8 9 10 15 22 30 34 50 52
================QuickSort_O(nlog2(n))================
Taken Time(ms) : 0
1 2 3 4 5 6 7 8 9 10 15 22 30 34 50 52
```

Bubble Sort가 가장 오래 걸린 것을 볼 수 있고 Insertion Sort, Merge Sort, Quick Sort순으로 빨라지는 것을 볼 수 있다. O(n^2)인 Insertion Sort와 O(nlog2(n))인 Merge Sort사이에 큰 갭이 있는 것을 확인할 수 있다. 임의의 짧은 수열을 입력해주고 각각의 정렬 결과를 보면 모두 동일하게 오름차순으로 정렬하는 것을 볼 수 있다.

깨달은 점 및 결론

재귀로 처리되는 Quick Sort와 Merge Sort는 재귀 특성상 역시 구현하기 상대적으로 쉽다는 것을 볼 수 있었고 O(n^2)과 O(nlog2(n))사이에 엄청난 성능차이가 있다는 것을 볼 수 있었다. 하지만 여기서 성능이 좋지 않은 Bubble Sort나 Insertion Sort도 특정 상황에서는 Merge Sort나 Quick Sort보다 성능이 좋을 수 있다. 예를 들어 이미 정렬되어있는 수열에 새로 들어온 값을 정렬이 유지되도록 삽입하는 경우에는 Insertion Sort를 사용하면 O(n)으로 수행하게 된다. 즉, 위의 정렬 방법들은 각각 쓰이는 방향이 있으며 무조건적인 장단점이 있다고는 할 수 없다.