자료구조 실습 보고서

[제12주] 정렬 성능 비교

2021년 05월 24일

201702039 오명주

1. 프로그램 설명서

(1) 프로그램의 전체 설계 구조

➔ MVC (Model – View – Controller) 구조

Model : 프로그램이 "무엇"을 할 것인지 정의. 사용자의 요청에 맞는 알고리즘을 처리하고 DB와 상호작용하여 결과물을 산출하고 Controller에게 전달.

View : 화면에 무엇인가를 "보여주기 위한" 역할. 최종 사용자에게 "무엇"을 화면으로 보여줌.

Controller : 모델이 "어떻게" 처리할 지 알려주는 역할. 사용자로부터 입력을 받고 중개인 역할. Model과 View는 서로 직접 주고받을 수 없음. Controller을 통해 이야기함.

➔ 정렬 결과 검증 프로그램에서의 각 클래스 별 MVC 구조 역할

Model :

- DataGenerator : 데이터를 생성하고 배열을 만든다.
- Sort<E> (Abstract) : 삽입정렬과 퀵정렬의 공통 기능.
- InsertionSort<E> : 삽입정렬의 기능을 구현한다.
- QuickSort<E> : 퀵정렬의 기능을 구현한다.
- ListOrder (Enum) : 오름차순/내림차순/랜덤값 세가지 방법에 대한 Enum 클래스.
- ExperimentManager : 실험 준비, 실험 시작을 진행한다.
- Experiment : 정렬 성능 측정 실험을 실행한다.
- ParameterSet : 단순히 실험에서 필요한 매개변수들 모아 놓는 역할.
- Timer : 실험 수행시간을 측정할 메소드를 구성한다.

View :

- AppView : 프로그램의 입/출력을 담당한다.

Controller :

- AppController : Model을 통해 생성된 결과물을 AppView를 통해 출력한다.

(2) 함수 설명서

➔ 주요 알고리즘

1) 리스트 정렬 시간 측정 후 측정값 배열 반환

```
public long[] durationsOfSort(Sort<Integer> aSort, Integer[] experimentList) {
    // 정렬 방법이 매개변수로 주어져 있음에 유의할 것 : Class "Sort" 를 볼 것
    int numberOfSteps = this.parameterSet().numberOfSizeIncreasingSteps(); // 크기 별로 실행할 측정 횟수
    long[] durations = new long[numberOfSteps]; // 측정 결과를 저장할 곳
    int sortingSize = this.parameterSet().startingSize(); // 정렬 데이터의 시작 크기
    int incrementSize = this.parameterSet().incrementSize(); // 정렬 데이터 증가 크기
    for (int step = 0; step < numberOfSteps; step++) {
        Integer[] listForSorting = this.copyListOfGivenSize(experimentList, sortingSize); // 측정에 사용할 데이터 리스트 복사
        durations[step] = this.durationOfSingleSort(aSort, listForSorting); // 측정하여 , 그 결과를 저장
        sortingSize += incrementSize; // 다음 단계의 정렬 데이터 크기를 얻는다
    }
    return durations;
}
```

- 정렬 방법과 리스트를 매개변수로 받아와 측정 결과를 배열로 반환하는 함수이다.
- 측정 횟수, 시작크기, 증가크기를 parameterSet에서 받아온다.
- 측정 횟수가 저장된 numberOfSteps만큼 반복하여 for문을 반복한다.
- 측정에 사용할 데이터 리스트를 복사한 후, durationOfSingleSort 함수를 이용하여 측정한 후 그 결과를 durations 배열에 저장한다. 다음 단계의 정렬 데이터 크기를 얻은 후 모두 끝나면 durations 배열을 반환한다.

```
private long durationOfSingleSort(Sort<Integer> aSort, Integer[] aList) { // 정렬 실행 시간 측정
    Timer timer = new Timer(); // Timer 객체 생성
    timer.start(); // 시작
    { aSort.sort(aList, aList.length); } // 시간 측정할 코드
    timer.stop(); // 끝
    return timer.duration(); // 측정된 결과
}
```

- 정렬 실행 시간을 측정하는 durationOfSingleSort 함수
- Timer 객체를 생성한다.
- start() 함수로 시간 측정을 시작하고 stop() 함수로 시간 측정을 종료하는데 사이에 시간 측정을 실행할 코드를 삽입한다.
- 측정 후, 측정 시간 값의 배열 (long) duration[] 이 반환된다.

2) Experiment Manager

```
// Constants:실험에서 사용할 정렬들을 상수 객체로 선언
private static final InsertionSort<Integer> INSERTION_SORT = new InsertionSort<Integer>();
private static final QuickSort<Integer> QUICK_SORT = new QuickSort<Integer>();

// Private instance variables
private Experiment _experiment;// 측정 실험을 실시할 객체
private ParameterSet _parameterSet; // 측정 실험에 사용할 매개변수 집합
private Integer[] ascendingOrderList; // 측정에서 정렬에 사용할 오름차순 데이터 리스트
private Integer[] descendingOrderList; // 측정에서 정렬에 사용할 내림차순 데이터 리스트
private Integer[] randomOrderList;;// 측정에서 정렬에 사용할 무작위 데이터 리스트
private long[] _measuredResultForInsertionSort; // 삽입 정렬의 측정 결과 저장할 곳
private long[] _measuredResultForQuickSort;;// 퀵 정렬의 측정 결과 저장할 곳
```

⇨ 실험에 필요한 변수.

⇨ 실험할 객체, 매개변수 집합, 오름차순/내림차순/무작위 데이터 리스트, 결과 저장 변수 등을 private으로 선언한다.

```
public void prepareExperiment(ParameterSet aParameterSet) { // 실험을 준비한다
    if (aParameterSet != null) {
        // 객체 생성할 때 , 매개변수 집합은 기본 값으로 설정되어 있다
        // 실험 준비 단계에서 , 이렇게 새로운 매개변수 집합을 주어 변경할 수 있다
        this.setParameterSet(aParameterSet);
    }
    this.setExperiment(new Experiment(this.parameterSet()));
    // 현재 상태의 매개변수 집합을 사용하여 Experiment 객체를 생성한다
    this.prepareExperimentLists();
    // 측정 실험에서 정렬에 사용할 데이터 리스트를 생성하여 보관한다
    // 다음의 내용은 생략 가능한 , 내용적으로는 의미가 없는 실행이다
    // 단지 실험 측정 결과를 안정화시키기 위한 목적일 뿐이다
    this.performExperiment(ListOrder.Random);
    this.performExperiment(ListOrder.Random);
}
```

- 실험을 준비하는 prepareExperiment 함수
- 실험 준비 단계에서 새로운 매개변수 집합을 주어 변경한다.
- Experiment 객체를 생성하여 설정한다.
- 실험 측정 결과를 안정화하기 위해 performExperiment 를 통한 Random 리스트 실험을 2번 진행한다.

(3) 종합 설명서

➔ 프로그램 실행 순서대로 설명해보자.

```
public class _DS12_201702039_오명주 {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        AppController appController = new AppController() ;
        // AppController가 실질적인 main class 이다
        appController.run();
        //여기 main() 에서는 앱 실행이 시작되도록 해주는 일이 전부이다

    }

}
```

main에서 AppController 의 객체를 생성하여 run 한다. 프로그램을 실행한다.

```
public void run() {
    AppView.outputLine("<<< 정렬 성능 비교 프로그램을 시작합니다 >>>");
    AppView.outputLine("");
    {
        AppView.outputLine(">> 2가지 정렬의 성능 비교 : 삽입 , 퀵 <<");
        this.manager().prepareExperiment(null);
        // ExperimentManager객체에게 실험을 준비시킨다
        // 이번 실험에서는 매개변수 값으로 기본 설정 값을 사용한다
        // 기본 설정 값은 Class "ExperimentManager" 에 설정되어 있다
        this.measureAndShowFor(ListOrder.Ascending); // 오름차순 정렬 측정 결과
        this.measureAndShowFor(ListOrder.Descending); // 내림차순 정렬 측정 결과
        this.measureAndShowFor(ListOrder.Random); // 무작위 정렬 측정 결과
    }
    AppView.outputLine("<<< 정렬 성능 비교 프로그램을 종료합니다 >>>");
}
```

⇨ AppController의 run 함수이다. 프로그램 실행

⇨ 오름차순, 내림차순, 랜덤 값 리스트에 대한 정렬 측정을 실행한다.

```
private void measureAndShowFor(ListOrder anOrder) {
    this.manager().performExperiment(anOrder);
    this.showResultTable(anOrder);
}
```

- 배열 생성 방법 anOrder을 인자로 받고 ExperimentManager()의 performExperiment를 실행한다.

```
private void showResultTable(ListOrder anOrder) { // 주어진 anOrder에 대하여, 성능 측정 결과를 보여준다
    this.showTableTitle(anOrder);
    this.showTableHead();
    this.showTableContent();
    AppView.outputLine("");
}
```

- 결과를 출력하도록 하는 showResultTable() 함수이다. anOrder가 주어져 해당 결과를 반환한다.
- showTableTitle() 함수는 테이블 제목을, showTableHead() 함수는 테이블 헤드를, showTableContent() 함수는 측정 결과를 출력한다.

```
private void showTableContent() { // 측정 결과 출력
    int startingSize = this.manager().parameterSet().startingSize(); // 시작크기
    int incrementSize = this.manager().parameterSet().incrementSize(); // 증가크기
    int numberOfSteps = this.manager().parameterSet().numberOfSizeIncreasingSteps(); // 스텝 횟수
    for (int step = 0; step < numberOfSteps; step++) {
        int sortingSize = startingSize + (incrementSize * step);
        AppView.outputLine("[" + String.format("%5d", sortingSize) + "]"
                + String.format("%16d", this.manager().measuredResultForInsertionSortAt(step))
                + String.format("%16d", this.manager().measuredResultForQuickSortAt(step)));
    }
}
```

- 데이터 결과에 맞게 형식을 맞추어 출력하게 된다.
- parameterSet() 의 각종 매개변수를 받아와 진행한다. 횟수를 받아와 횟수만큼 for문을 반복한다.

2. 프로그램 장단점 / 특이점 분석

➔ 장점

- 정렬에 공통부분을 추상클래스로 정의하여 새로운 정렬을 추가하거나 정렬을 수정하고 싶을 때, Sort나 Swap 함수에 대한 수정은 하지 않고 사용하여 구현할 수 있다.
- 각 정렬에 대한 검증을 해볼 수 있어서 구현한 정렬이 올바르게 작동하는지 확인할 수 있다.
- 실험, 실험준비, 실험에 사용되는 매개변수, 시간 측정 등 각각의 기능을 갖는 클래스들로 기능을 나누어 설계하였기 때문에 가독성이 높고 수정이 용이하다.

➔ 단점

- 기능을 많이 나누어 헷갈릴 수 있다. 클래스가 많아지며 코드가 길어진다.

3. 실행 결과 분석

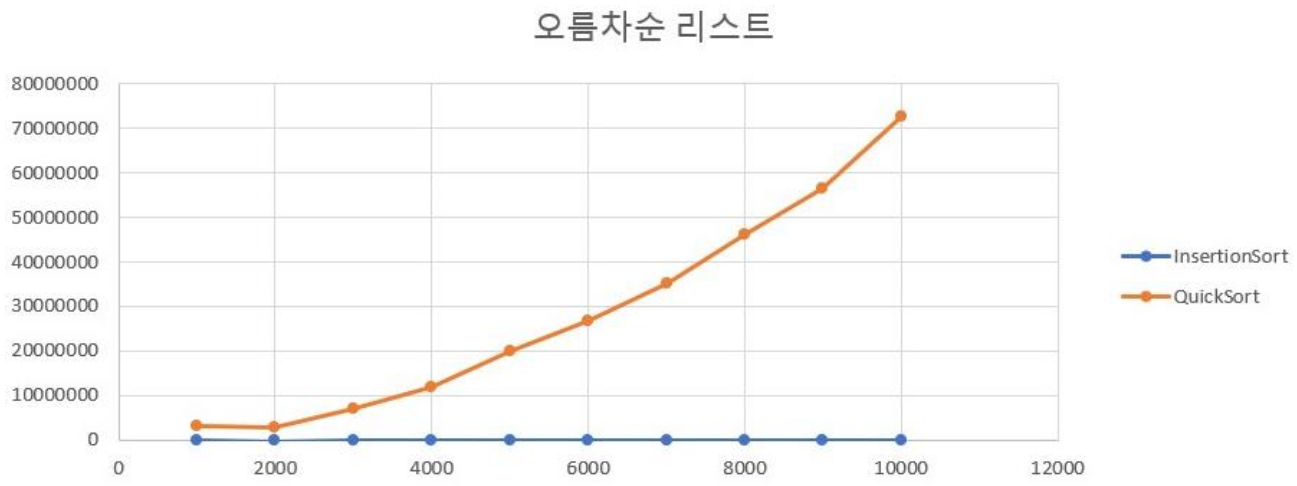
(1) 입력과 출력

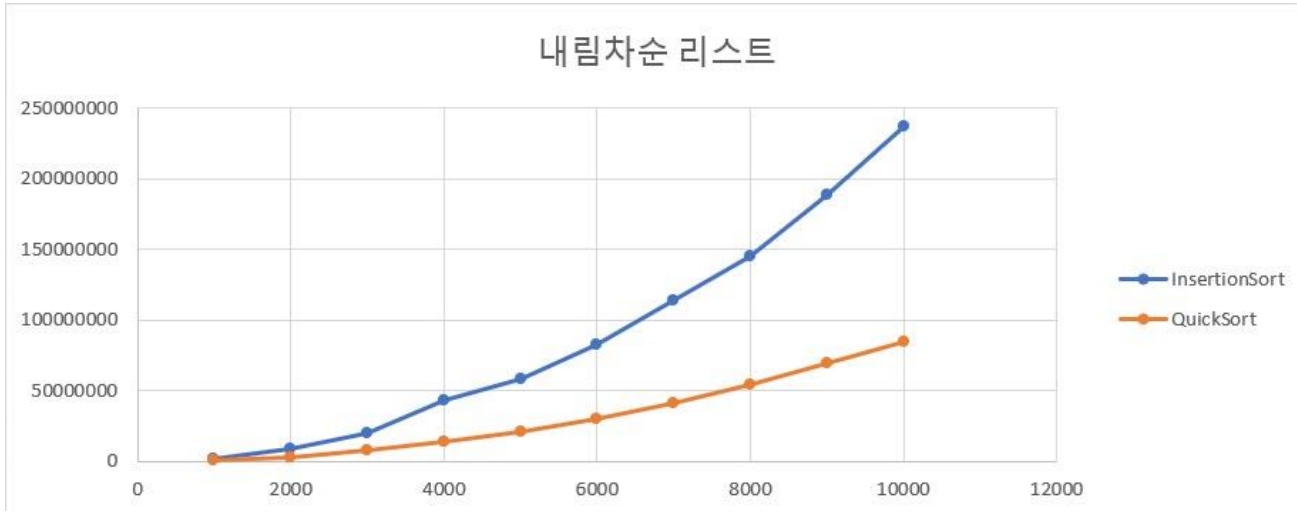
```
> 오름차순 데이터를 사용하여 실행한 측정:
        <Insertion Sort>    <Quick Sort>
[ 1000]            18800         3282000
[ 2000]            15600         2885800
[ 3000]            16600         7062600
[ 4000]            21200        11744300
[ 5000]            26300        20009600
[ 6000]            31400        26840200
[ 7000]            36500        35277600
[ 8000]            41800        46249700
[ 9000]            46600        56633900
[10000]            52200        72639700

> 내림차순 데이터를 사용하여 실행한 측정:
        <Insertion Sort>    <Quick Sort>
[ 1000]          2227400          831500
[ 2000]          9266300         3281900
[ 3000]         20007300         7974300
[ 4000]         43272800        13552900
[ 5000]         57954600        21012400
[ 6000]         82505300        30080400
[ 7000]        113644100        41149300
[ 8000]        145066900        54053800
[ 9000]        188713300        69457400
[10000]        237316600        84513000

> 무작위 데이터를 사용하여 실행한 측정:
        <Insertion Sort>    <Quick Sort>
[ 1000]          2074100          113400
[ 2000]          5415100          171400
[ 3000]          9340200          265200
[ 4000]         17451700          397200
[ 5000]         31462500          466100
[ 6000]         42560600          612400
[ 7000]         73767800          711500
[ 8000]         84180600          787000
[ 9000]         88014000         1631800
[10000]        109295200         1048300

<<< 정렬 성능 비교 프로그램을 종료합니다 >>>
```

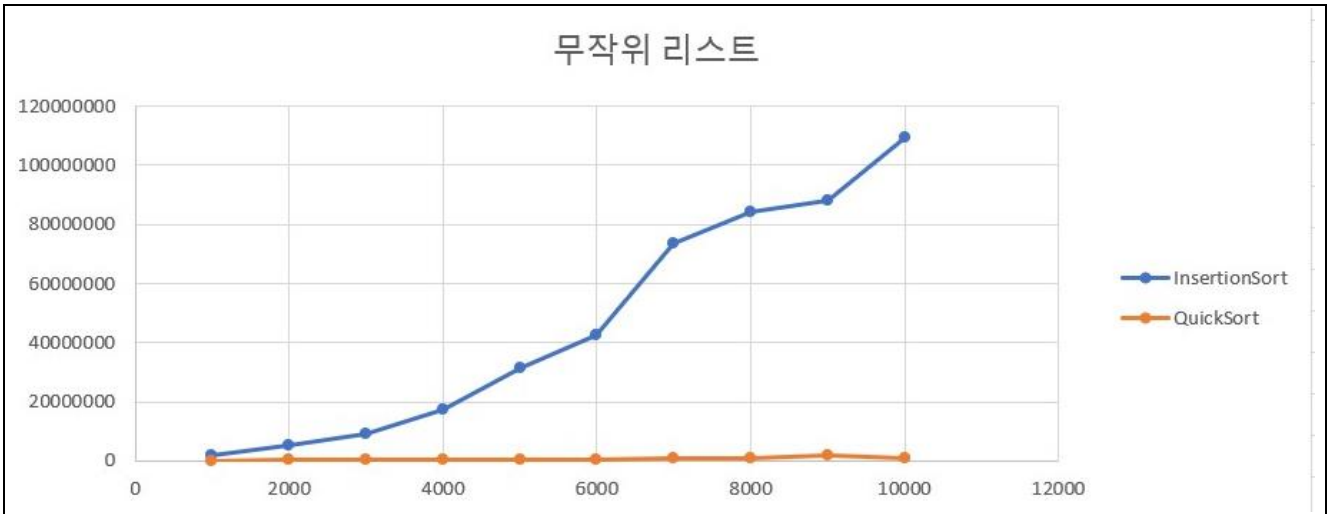
(2) 결과 분석 (자신의 논리적 평가, 기타 느낀 점)

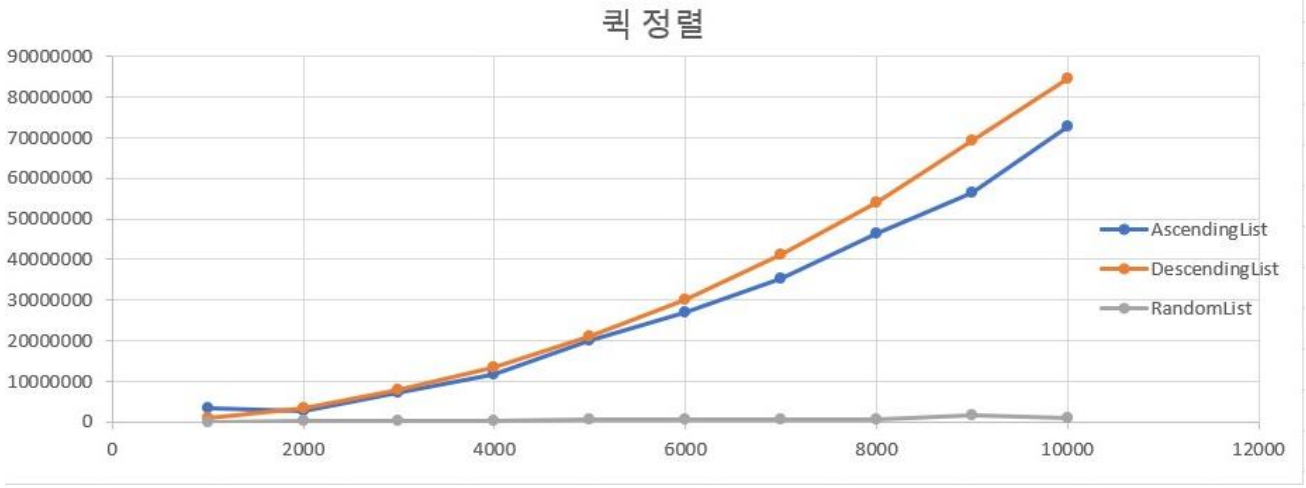
➔ 이론적 시간 복잡도

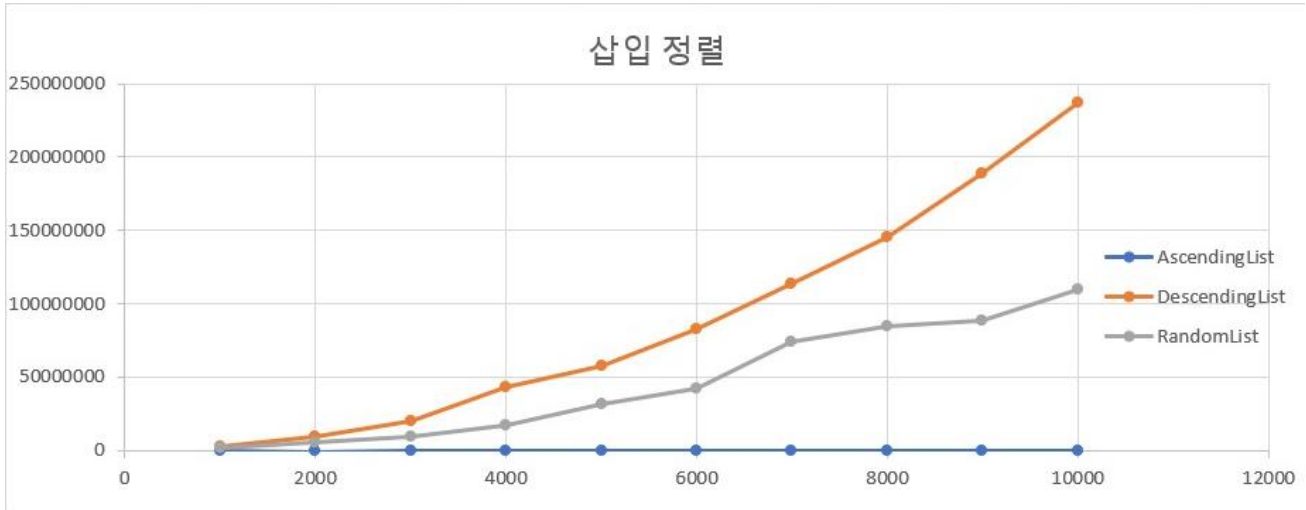
- 삽입 정렬 : 최선의 경우, 이미 정렬된 리스트에 대하여 O(N) 만큼이며, 최악의 경우 배열 전체를 비교해야 하므로 O(N^2)이다. 알고리즘이 간단하므로 배열의 크기가 작을수록, 배열이 이미 정렬되어 있을수록 효과적이다.
- 퀵 정렬 : 평균적으로 O(NlogN) 이고, 오름차순 일 때 최악의 경우 O(N^2)이다.











- 주어진 예시에 비해 성능 측정 값이 너무 높게 나와 잘 못 측정된 것은 아닌가 생각했는데 그래프로 나타내보니 맞게 측정되었음을 확인할 수 있었다.

4. 소스코드

```
import java.util.Scanner;

public class AppView {
    private static Scanner scanner = new Scanner(System.in);

    // 생성자
    public AppView() {}

    // 출력 관련 함수
    // 한줄을 출력하는 함수 (한줄이 띄워지지않는다)
    public static void output(String message) {
        System.out.print(message); // 입력받은 message를 출력한다
    }
    // 한줄을 출력하는 함수 (한줄이 띄워진다)
    public static void outputLine(String message) {
        System.out.println(message); // 입력받은 message를 출력한다
    }

    // 입력 관련 함수
    public static char inputChar() {
        String line = AppView.scanner.nextLine().trim();
        while(line.equals("")) {
            line = AppView.scanner.nextLine().trim();
        }
        return line.charAt(0);
    }
}
```

[AppView]

```
public class InsertionSort<E extends Comparable<E>> extends Sort<E> {
    // Constructor
    public InsertionSort() {}

    // Public Method
    public boolean sort(E[] aList, int aSize) {
        if ((aSize < 1) || (aSize > aList.length)) { // aSize 값이 유효한지 판단
            return false;
        }
        int minLoc = 0; // 가장 작은 값이 존재하는 위치를 저장할 변수
        for (int i = 1; i < aSize; i++) { // 제일 작은 값을 찾아 minLoc에 저장
            if (aList[i].compareTo(aList[minLoc]) < 0) {
                minLoc = i;
            }
        }
        this.swap(aList, 0, minLoc); // 0을 삽입하여 실질적인 -무한대 값 역할을 한다.
        // Abstract class "Sort" 에 구현된 것을 그대로 사용하고 있다
        for (int i = 2; i < aSize; i++) {
            E insertedElement = aList[i]; // 삽입할 원소
            int insertionLoc = i - 1; // 삽입할 위치
            while (aList[insertionLoc].compareTo(insertedElement) > 0) { // 삽입할 위치의 원소 값이 삽입할 원소보다 크다면
                aList[insertionLoc + 1] = aList[insertionLoc]; // 그 뒤 값 뒤로 이동
                insertionLoc--;
            }
            // While loop조건이 false 라서 loop 종료 . 따라서 , (insertionLoc+1) 이 원소 삽입 위치
            aList[insertionLoc + 1] = insertedElement;
        }
        return true;
    }
} // End of "Sort"
```

[InsertionSort]

```
    // Static Class.
    // 더이상 상속할 필요가 없으므로 "final" 선언

    // 생성자는 private. 외부에서 객체를 만들 수 없다.
    private DataGenerator() {};

    // 모든 공개함수는 static
    public static Integer[] ascendingOrderList(int aSize) { // 오름차순 리스트 생성
        Integer[] list = null; // 초기화
        if (aSize > 0) { // aSize가 0보다 크다면
            list = new Integer[aSize]; // 배열 생성
            for (int i = 0; i < aSize; i++) { // i에 맞게 리스트에 값 생성
                list[i] = i;
            }
        }
        return list;
    }

    public static Integer[] descendingOrderList(int aSize) { // 내림차순 리스트 생성
        Integer[] list = null; // 초기화
        int count = aSize - 1; // 값을 넣을 때 사용할 count 생성
        if (aSize > 0) { // aSize가 0보다 크다면
            list = new Integer[aSize]; // 배열생성
            for (int i = 0; i < aSize; i++) { // i에 맞게 리스트에 값 생성
                list[i] = count;
                count--;
            }
        }
        return list;
    }

    public static Integer[] randomOrderList(int aSize) { // 랜덤 값 리스트 생성
        // 겹치는 원소가 없는 무작위 리스트를 생성하여 , 돌려준다
        Integer[] list = null;
        if (aSize > 0) {
            // 일단 Ascending order list 를 만든다
            list = new Integer[aSize];
            for (int i = 0; i < aSize; i++) {
                list[i] = i;
            }
            // 각 원소 list[i] 에 대해 무작위 위치 r 을 생성하여 list[i] 와 list[r] 를 맞바꾼다
            Random random = new Random();
            for (int i = 0; i < aSize; i++) {
                int r = random.nextInt(aSize);
                Integer temp = list[i];
                list[i] = list[r];
                list[r] = temp;
            }
        }

        return list;

    }
}
```

[DataGenerator]

```
public abstract class Sort<E extends Comparable<E>> {
    // Protected Method
    protected void swap(E[] aList, int i, int j) {
        E tempElement = aList[i];
        aList[i] = aList[j];
        aList[j] = tempElement;
    }

    // Constructor
    protected Sort() {
    }

    // Public Method
    public abstract boolean sort(E[] aList, int aSize);
} // End of "Sort"
```

[Sort]

```
    // Constructor
    public QuickSort() {
    }

    // Private methods
    private int pivot(E[] aList, int left, int right) { // pivot값 반환
        return left;
    }

    private int partition(E[] aList, int left, int right) {
        int toRight = left; // 오른쪽으로 갈 toRight는 left 위치에 지정
        int toLeft = right + 1; // 왼쪽으로 갈 toLeft는 right+1 위치에 지정
        int pivot = this.pivot(aList, left, right);
        do {
            do {toRight++;} while (aList[toRight].compareTo(aList[pivot]) < 0); // Left에서 Right로 갈 위치 선정
            do {toLeft--;} while (aList[toLeft].compareTo(aList[pivot]) > 0); // Right에서 Left로 갈 위치 선정
            if (toRight < toLeft) { // toRight < toLeft라면
                this.swap(aList, toRight, toLeft); // 두개의 배열을 바꾼다.
            }
        } while (toRight < toLeft); // toRight > toLeft가 되는 순간 탈출
        this.swap(aList, pivot, toLeft); // pivot과 toLeft를 바꾼다.
        return toLeft; // pivot 위치가 toLeft 이다.
    }

    private void quickSortRecursively(E[] aList, int left, int right) {
        if (left < right) { // left<right이면
            int mid = this.partition(aList, left, right); // 파티션 후의 pivot 위치
            this.quickSortRecursively(aList, left, mid - 1); // 나누어진 반을 다시 퀵정렬한다.
            this.quickSortRecursively(aList, mid + 1, right); // 나누어진 반을 다시 퀵정렬한다.
        }
    }

    @Override
    public boolean sort(E[] aList, int aSize) {
        if ((aSize < 1) || (aSize > aList.length)) {
            return false;
        }
        int maxLoc = 0;
        for (int i = 1; i < aSize; i++) {
            if (aList[i].compareTo(aList[maxLoc]) > 0) {
                maxLoc = i;
            }
        }
        this.swap(aList, maxLoc, aSize - 1);
        this.quickSortRecursively(aList, 0, aSize - 2);
        return true;
    }
} // End of "Sort"
```

[QuickSort]

```
public enum ListOrder {
    // 이번 실험에서는 , 3 가지 유형의 데이터 리스트를 구분하고 있다
    // 이 구분을 표현할 목적으로 Enum "ListOrder" 를 사용한다
    Ascending, // 오름차순 데이터 리스트의 유형을 표현
    Descending, // 내림차순 데이터 리스트의 유형을 표현
    Random; // 무작위 데이터 리스트의 유형을 표현
    // Enum은 Class 의 특수한 경우로 간주된다
    // 따라서 아래와 같이 상수를 선언할 수 있다

    public static final String[] ORDER_NAMES = { "오름차순", "내림차순", "무작위" };

    // 또한 아래와 같이 member method 역시 선언할 수 있다
    // 즉 Enum 안에 선언된 값들은 Enum 의 객체 인스턴스로 인식된다
    // Ascending, Descending, Random 각각이 객체 인스턴스이다
    public String orderName() {
        return ListOrder.ORDER_NAMES[this.ordinal()];
        // "ordinal()" 은 모든 Enum 에 미리 정의되어 있는 함수로
        // 선언된 값의 Enum 안에서의 순서를 정수로 얻을 수 있다
        // 즉, Ascending.ordinal()은 0, Descending.ordinal() 은 1,
        // Random.ordinal()은 2 를 얻는다
    }

}
```

[ListOrder]

```
public class AppController {
    private ExperimentManager _manager;

    // Getters/Setters
    private ExperimentManager manager() {
        return this._manager;
    }
    private void setManager(ExperimentManager newManager) {
        this._manager = newManager;
    }

    // 생성자
    public AppController() {
        this.setManager(new ExperimentManager());
    }

    private void showTableTitle(ListOrder anOrder) { // 테이블 제목을 출력
        AppView.outputLine("> " + anOrder.orderName() + " 데이터를 사용하여 실행한 측정:");
    }

    private void showTableHead() { // 테이블 헤드를 출력 <Insertion Sort> <Quick Sort>
        AppView.outputLine(String.format("%8s", "") + String.format("%16s", "<Insertion Sort>")
                + String.format("%16s", "<Quick Sort>"));
    }

    private void showTableContent() { // 측정 결과 출력
        int startingSize = this.manager().parameterSet().startingSize(); // 시작크기
        int incrementSize = this.manager().parameterSet().incrementSize(); // 증가크기
        int numberOfSteps = this.manager().parameterSet().numberOfSizeIncreasingSteps(); // 스텝 횟수
        for (int step = 0; step < numberOfSteps; step++) {
            int sortingSize = startingSize + (incrementSize * step);
            AppView.outputLine("[" + String.format("%5d", sortingSize) + "]"
                    + String.format("%16d", this.manager().measuredResultForInsertionSortAt(step))
                    + String.format("%16d", this.manager().measuredResultForQuickSortAt(step)));
        }
    }

    private void showResultTable(ListOrder anOrder) { // 주어진 anOrder에 대하여, 성능 측정 결과를 보여준다
        this.showTableTitle(anOrder);
        this.showTableHead();
        this.showTableContent();
        AppView.outputLine("");
    }

    private void measureAndShowFor(ListOrder anOrder) {
        this.manager().performExperiment(anOrder);
        this.showResultTable(anOrder);
    }

    public void run() {
        AppView.outputLine("<<< 정렬 성능 비교 프로그램을 시작합니다 >>>");
        AppView.outputLine("");
        {
            AppView.outputLine(">> 2가지 정렬의 성능 비교 : 삽입 , 퀵 <<");
            this.manager().prepareExperiment(null);
            // ExperimentManager객체에게 실험을 준비시킨다
            // 이번 실험에서는 매개변수 값으로 기본 설정 값을 사용한다
            // 기본 설정 값은 Class "ExperimentManager" 에 설정되어 있다
            this.measureAndShowFor(ListOrder.Ascending); // 오름차순 정렬 측정 결과
            this.measureAndShowFor(ListOrder.Descending); // 내림차순 정렬 측정 결과
            this.measureAndShowFor(ListOrder.Random); // 무작위 정렬 측정 결과
        }
        AppView.outputLine("<<< 정렬 성능 비교 프로그램을 종료합니다 >>>");
    }
}
```

[AppController]

```
    private final ParameterSet _parameterSet;
    // Class의 함수에서 이 값을 변경할 수 없다
    // 오로지, 생성자 안에서만 값을 설정할 수 있다
    // 즉, 객체를 생성할 때 정해진 값을 그대로 유지한다

    // 생성자
    public Experiment(ParameterSet givenParameterSet) {
            this._parameterSet = givenParameterSet;
    }

    private ParameterSet parameterSet() {
        return this._parameterSet;
    }

    private Integer[] copyListOfGivenSize(Integer[] aList, int copiedSize) {
        Integer[] copiedList = null;
        if (copiedSize <= aList.length) {
            copiedList = new Integer[copiedSize];
            for (int i = 0; i < copiedSize; i++) {
                copiedList[i] = aList[i];
            }
        }
        return copiedList; // aList[]에서 copiedSize만큼 복사한 리스트를 얻는다
    }

    private long durationOfSingleSort(Sort<Integer> aSort, Integer[] aList) { // 정렬 실행 시간 측정
        Timer timer = new Timer(); // Timer 객체 생성
        timer.start(); // 시작
        { aSort.sort(aList, aList.length); } // 시간 측정할 코드
        timer.stop(); // 끝
        return timer.duration(); // 측정된 결과
    }

    public long[] durationsOfSort(Sort<Integer> aSort, Integer[] experimentList) {
        // 정렬 방법이 매개변수로 주어져 있음에 유의할 것 : Class "Sort" 를 볼 것
        int numberOfSteps = this.parameterSet().numberOfSizeIncreasingSteps(); // 크기 별로 실행할 측정 횟수
        long[] durations = new long[numberOfSteps]; // 측정 결과를 저장할 곳
        int sortingSize = this.parameterSet().startingSize(); // 정렬 데이터의 시작 크기
        int incrementSize = this.parameterSet().incrementSize(); // 정렬 데이터 증가 크기
        for (int step = 0; step < numberOfSteps; step++) {
            Integer[] listForSorting = this.copyListOfGivenSize(experimentList, sortingSize); // 측정에 사용할 데이터 리스트 복사
            durations[step] = this.durationOfSingleSort(aSort, listForSorting); // 측정하여 , 그 결과를 저장
            sortingSize += incrementSize; // 다음 단계의 정렬 데이터 크기를 얻는다
        }
        return durations;
    }
}
```

[Experiment]

```
public class ParameterSet {
    private int _startingSize;
    private int _numberOfSizeIncreasingSteps;
    private int _incrementSize;

// public Getters & Setters
    public int startingSize() {
        return this._startingSize;
    }
    public void setStartingSize(int newStaringSize) {
        this._startingSize = newStaringSize;
    }
    public int numberOfSizeIncreasingSteps() {
        return this._numberOfSizeIncreasingSteps;
    }
    public void setNumberOfSizeIncreasingSteps(int newNumberOfSizeIncreasingSteps) {
        this._numberOfSizeIncreasingSteps = newNumberOfSizeIncreasingSteps;
    }
    public int incrementSize() {
        return this._incrementSize;
    }
    public void setIncrementSize(int newIncrementSize) {
        this._incrementSize = newIncrementSize;
    }
    public int maxDataSize() {
        return (this.startingSize() + (this.incrementSize() * (this.numberOfSizeIncreasingSteps() - 1)));
    }

    // 생성자
    public ParameterSet(int givenStartingSize, int givenNumberOfSizeIncreasingSteps, int givenIncrementSize) {
        this.setIncrementSize(givenIncrementSize);
        this.setNumberOfSizeIncreasingSteps(givenNumberOfSizeIncreasingSteps);
        this.setStartingSize(givenStartingSize);
    }
}
```

[ParameterSet]

```
public class ExperimentManager {
    private static final int DEFAULT_NUMBER_OF_SIZE_INCREASING_STEPS = 10;
    private static final int DEFAULT_INCREMENT_SIZE = 1000;
    private static final int DEFAULT_STARTING_SIZE = DEFAULT_INCREMENT_SIZE;

    // Constants:실험에서 사용할 정렬들을 상수 객체로 선언
    private static final InsertionSort<Integer> INSERTION_SORT = new InsertionSort<Integer>();
    private static final QuickSort<Integer> QUICK_SORT = new QuickSort<Integer>();

    // Private instance variables
    private Experiment _experiment;// 측정 실험을 실시할 객체
    private ParameterSet _parameterSet; // 측정 실험에 사용할 매개변수 집합
    private Integer[] ascendingOrderList; // 측정에서 정렬에 사용할 오름차순 데이터 리스트
    private Integer[] descendingOrderList; // 측정에서 정렬에 사용할 내림차순 데이터 리스트
    private Integer[] randomOrderList;;// 측정에서 정렬에 사용할 무작위 데이터 리스트
    private long[] _measuredResultForInsertionSort; // 삽입 정렬의 측정 결과 저장할 곳
    private long[] _measuredResultForQuickSort;;// 퀵 정렬의 측정 결과 저장할 곳

    // Getter/Setter
    private Experiment experiment() {
        return this._experiment;
    }
    private void setExperiment(Experiment newExperiment) {
        this._experiment = newExperiment;
    }
    public ParameterSet parameterSet() {
        return this._parameterSet;
    }
    private void setParameterSet(ParameterSet newParameterSet) {
        this._parameterSet = newParameterSet;
    }
    private Integer[] ascendingOrderList() {
        return this.ascendingOrderList;
    }
    private void setAscendingOrderList(Integer[] newAscendingOrderList) {
        this.ascendingOrderList = newAscendingOrderList;
    }
    private Integer[] descendingOrderList() {
        return this.descendingOrderList;
    }
    private void setDescendingOrderList(Integer[] newDescendingOrderList) {
        this.descendingOrderList = newDescendingOrderList;
    }
    private Integer[] randomOrderList() {
        return this.randomOrderList;
    }
    private void setRandomOrderList(Integer[] newRandomOrderList) {
        this.randomOrderList = newRandomOrderList;
    }
    private long[] measuredResultForInsertionSort() {
        return this._measuredResultForInsertionSort;
    }
    private void setMeasuredResultForInsertionSort(long[] newMeasuredResultForInsertionSort) {
        this._measuredResultForInsertionSort = newMeasuredResultForInsertionSort;
    }
    private long[] measuredResultForQuickSort() {
        return this._measuredResultForQuickSort;
    }
    private void setMeasuredResultForQuickSort(long[] newMeasuredResultForQuickSort) {
        this._measuredResultForQuickSort = newMeasuredResultForQuickSort;
    }

    // Constructor
    public ExperimentManager() {
        this.setParameterSetWithDefaults(); // 기본 값으로 매개변수 집합을 초기화한다
    }

    private void prepareExperimentLists() {
        int maxDataSize = this.parameterSet().maxDataSize();
        this.setAscendingOrderList(DataGenerator.ascendingOrderList(maxDataSize));
        this.setDescendingOrderList(DataGenerator.descendingOrderList(maxDataSize));
        this.setRandomOrderList(DataGenerator.randomOrderList(maxDataSize));
    }

    private void setParameterSetWithDefaults() {
        this.setParameterSet(new ParameterSet(DEFAULT_STARTING_SIZE, DEFAULT_NUMBER_OF_SIZE_INCREASING_STEPS,
                DEFAULT_INCREMENT_SIZE));
    }

    private Integer[] experimentListOfOrder(ListOrder anOrder) { // 주어진 anOrder 에 해당하는 리스트를 돌려준다
        switch (anOrder) {
        case Ascending:
            return this.ascendingOrderList();
        case Descending:
            return this.descendingOrderList();
        default:
            return this.randomOrderList();
        }
    }

    public void prepareExperiment(ParameterSet aParameterSet) { // 실험을 준비한다
        if (aParameterSet != null) {
            // 객체 생성할 때 , 매개변수 집합은 기본 값으로 설정되어 있다
            // 실험 준비 단계에서 , 이렇게 새로운 매개변수 집합을 주어 변경할 수 있다
            this.setParameterSet(aParameterSet);
        }
        this.setExperiment(new Experiment(this.parameterSet()));
        // 현재 상태의 매개변수 집합을 사용하여 Experiment 객체를 생성한다
        this.prepareExperimentLists();
        // 측정 실험에서 정렬에 사용할 데이터 리스트를 생성하여 보관한다
        // 다음의 내용은 생략 가능한 , 내용적으로는 의미가 없는 실행이다
        // 단지 실험 측정 결과를 안정화시키기 위한 목적일 뿐이다
        this.performExperiment(ListOrder.Random);
        this.performExperiment(ListOrder.Random);
    }

    public long measuredResultForInsertionSortAt(int sizeStep) {
        return this.measuredResultForInsertionSort()[sizeStep];
    }

    public long measuredResultForQuickSortAt(int sizeStep) {
        return this.measuredResultForQuickSort()[sizeStep];
    }

    public void performExperiment(ListOrder anOrder) {
        // 측정 실험을 실행한다 .
        // 주어진 anOrder 의 실험 리스트를 얻는다
        Integer[] experimentList = this.experimentListOfOrder(anOrder);
        // 이 실험 리스트로 삽입과 퀵 각각의 정렬의 성능을 측정해서 , 그 결과를 얻는다
        this.setMeasuredResultForInsertionSort(this.experiment().durationsOfSort(INSERTION_SORT, experimentList));
        this.setMeasuredResultForQuickSort(this.experiment().durationsOfSort(QUICK_SORT, experimentList));
    }
}
```

[ExperimentManager]