자료구조 실습 보고서

[제 11주]

제출일: 19.05.29
학번/이름: 201603867/조성환

1. 프로그램 설명서

1) 주요 알고리즘/ 자료구조/ 기타

정렬 알고리즘의 결과를 비교. 정렬 알고리즘은 insertion과 quick 두 종료가 있음.

2) 함수 설명서

AppContorller

AppController()
setManager()를 통해 new ExperimentManager()로 새로운 experimentManager를 만듬.

run()
시작을 알림.
manager().setExperiment()를 통해 실험을 준비함
measureAndShowFor()으로 ascending, descending, random의 측정값을 보여줌.

measureAndShowFor()
manager().perfromExperiment()를 통해 해당 순서의 실험을 시랭함.
showResultTable()을 통해 순서의 결과를 보여줌.

showResultTable()
showTableTitle()을 통해 해당 데이터의 순서를 알려줌
show Table Head()를 통해 head를 보여줌
showTableContent()를 통해 측정값을 보여줌

showTableTitle()
해당 순서의 이름을 가져오며 무슨 순서인지를 보여줌

show Table Head()
형식에 맞게 insertion, quick임을 보여줌

showtableContent()
startingSize를 manager().parameterSet().startingSize()를 통해 가져옴.
마찬가지로 incrementSize, numberOfSteps를 같은 형식으로 가져옴

format을 통해 sortingSize, measureREsultForInsertionSortAtStep, measureResultForQuickSortAt 으로 해당 크기의 측정값을 나타냄

ExperimentManager

prepareExperiment()
입력받은 parameterSet이 null이 아니라면 set함.
setExperiment()를 통해 새로운 Experiment를 만듬

prepareExperimentList()를 통해 리스트를 만듬
performExperiment()를 통해 Random을 실행

prepareExperimentLists()
madataSize에 parameterSet().maxDataSize()를 통해 가져옴
set~List를 오름차, 내림차, 랜덤 순으로 DataGenerator.~List(maxDataSize)로 만듬

set~List()
List 자체는 Integer[]를 통해 배열을 만듬. 인자값을 통해 데이터를 넣음. 데이터는 DataGenerator를 통해 생성함.

performExperiment()
setMeasureResultFor~Sort()를 통해 해당 정렬의 결과를 입력함. 정렬의 결과는 experiment().durationOfSort()를 통해 insertion, quick을 진행 가능함.

exprimentListOforder()
order를 입력받아 해당 순서의 리스트를 반환함.

Experiment

durationsOfSort()
 해당 step에 맞는 순서에서 listforSorting을 copyListOfGivenSize()를 통해 복사,
duaration에 해당 step에 맞는 곳에 넣은 후 sortingSize를 더해줌.
마지막엔 duration을 리턴함.

durationOfSingleSort()
샤ㄷㄱ를 통해 sorting한 시간을체크

copyListOfGivenSize()
copiedsize에 맞게 해당 위치의 요소를 copiedList에 집어넣음.

Sort

sort()
list를 정렬함. 6주차 과제를 응용해서 작성함.
만약 list의 사이즈가 유효하지 않다면 false를 리턴함
size만큼 for문을 돌림. for문을 통해 가장 큰 요소를 가진 위치를 가져옴.
가져오면 맨 마지막에 가장 큰 요소를 넣고, 0번째부터 뒤에서2번째 위치까지를 quicksortreculsively에 집어넣음.

swap()
E를 하나 만든 후 aList의 I번째를 넣은 뒤 I번째에 j를 넣고, j번째에 E를 집어넣어 서로 바

꿔줌.

quickSortRecursively()
left가 right보다 작으면 mid를 partition()을 통해 계산한 후 left에서 mid-1까지와 mid+1에서 right까지 quickRecursively()를 통해 재귀적으로 정렬을 해줌.

partition()
pivot을 설정한 뒤 toRight에 left, toLeft에 right+1을 넣어 do while을 돌림.

Data Generator

ascendingList()
list를 만든 후 size만큼 for문을 돌림. I번째에 I를 차례로 집어넣음
list를 반환함

descendingList()
list를 만든 후 size만큼 for문을 돌림. I번째에 size-1을 넣은 후 차례로 - -를 통해 역순으로 집어넣음
list를 반환함

randomList()
list를 만든 후 size만큼 for문을 돌림. I번째에 I를 차례로 집어넣음. 이후 Random객체를 생성한 뒤. size만큼 for문을 만들어 size만큼 random을 돌림. temp에 I번째 리스트를 가져옴. I번째에 r번째 값을 집어넣은 후 r번째에 temp를 집어넣음.
list를 반환함

InsertionSort

sort()
본 함수를 통해 정렬을 시작함.
size가 유효한지 검사한 후 틀리면 false를 리턴함
유효하면 minLoc를 선언한 뒤 1번째부터size의 크기만큼 for문을 돌림
해당 위치의 요소가 minLoc위치의 요소보다 작으면 minLoc를 해당 위치로 바꿈
for문이 끝난 후 swap()을 통해 minLoc의 요소를 0번째로 보냄
for문을 2번째부터 size의 크기 만큼 돌림
aList가 E의 배열임. E를 하나 선언한 후 해당 위치의 요소를 넣음
insertionLoc를 I-1로 초기화함. insertionLoc위치에 있는 요소와 I에 위치한 요소를 비교해 insertionLoc에 위치한 요소가 더 크면 insertionLoc+1위치에 insertionLoc의 요소를 넣고 insertionLoc를 1빼줌. 이런식으로 계속 전에있는 요소가 현재 삽입하려는 요소보다 크다면 뒤로 보내줘서 공간을 만듬. 마지막으로 while문을 빠져냐가면 insertionLoc는 insertedElement가 들어가야할 위치 전의 숫자임. +1위치에 insertedElement를 넣어줌.
이런식을 hahems 요소를 정렬해준 뒤 true를 리턴함.

swap()
E를 만들어 I위치의 요소를 담음. I위치에 j위치 요소를 넣고 j위치에 E에 담은 I요소를 넣음

QuickSort

sort()
size가 유효한지 검사한 후 틀리면 false를 리턴함
유효하면 maxLoc를 선언한 뒤 for문을 1부터 size만큼 돌림
aList의 I번째와 maxLoc의 위치와 비교한 후 I가 더 크면 maxLoc를 I로 초기화함
끝나면 swap을 통해 maxLoc를 맨 마지막에 보냄
quickSortRecursively()에 0번째와 뒤에서 두 번째까지를 넣은 후 재귀적으로 정렬이 되도록 함. 마지막으로 true를 반환함

swap()
앞서 설명한 내용

quickSortRecursively()
left가 right보다 작으면 mid를 partition을 통해 구함.
quickSortRecursively에 left에서 mid-1, mid+1에서 right까지를 2개 해서 넣음
이제 재귀적으로 1개단위까지 쪼개진 뒤 다시 모두 정렬이 됨.

partition()
pivot을 기준으로 양쪽을 나눔
pivot을 left로 초기화함. toRight는 left, toLeft는 right로 가야할 위치를 정함
do while을 통해 toRight에 위치한 요소가 클때까지 toRight을 ++, toLeft에 위치한 요소가 pivot보다 작을 때 까지 toLeft를ㅡ함 만약 toRight이 toLeft보다 작으면 toRight와 toLeft를 swap()함 이는 toRight가 toLeft보다 클때까지 반복함.
마지막으로 left를 toLeft와 swap한 뒤 toLeft를 반환함.

pivot()
left를 리턴함

ListOrder
enum 클래스로 Ascending, Descending, Random 만듬.
Order_Names 배열에 각 오름차순 내림차순 무작위를 넣음
orderName()
ListOrder.ODER_NAMES[this.ordinal()]를 통해 한글 이름을 가져옴

ParameterSet

각 매개변수를 설정함
ParameterSet()
주어진 startingSize, numberOfSizeIncreasingSteps, incrementSize를 set을 통해 설정함.

maxDataSize()
startingSize, incrementSize, numberOFSizeIncreasingSteps를 연산해 최대 사이즈를 리턴함.

3) 종합 설명서

전체적으로 data를 생성한 후 배열에 집어넣는데, 순서를 오름차로, 내림차, 랜덤 3가지를 insertion과 quick으로 진행함. sort는 저번 과제의 class를 이용함. 정렬을 하면서 timer를 이용해 시간을 측정해 각 순과 정렬 종류에 따른 시간을 출력함으로써 능력을 비교함.

2. 프로그램 장단점/ 특이점 분석

결과분석에 결과를 그래프로 나타낸 사진을 첨부함. 이를 통해 sort는 quick sort, 순서는 오름차순으로 실행하는 것이 더 성능이 좋다고 할 수 있음. 프로그램의 장점은 아무래도 정렬을 quick sort로 하면서 partition을 재귀적으로 하기 때문에 일반적으로 random의 자료에서는 속도가 더 빠르다는 장점이 있음. 다만 descending의 경우는 insertion이 미미하게나마 더 효율적으로 정렬 가능함. ascending에서는 insertion이 훨씬 빠름. 이 점을 참고하면 데이터의 크기와 순서에 따라 어떤 알고리즘을 선택해야 하는지 고민해야함을 알게됨.

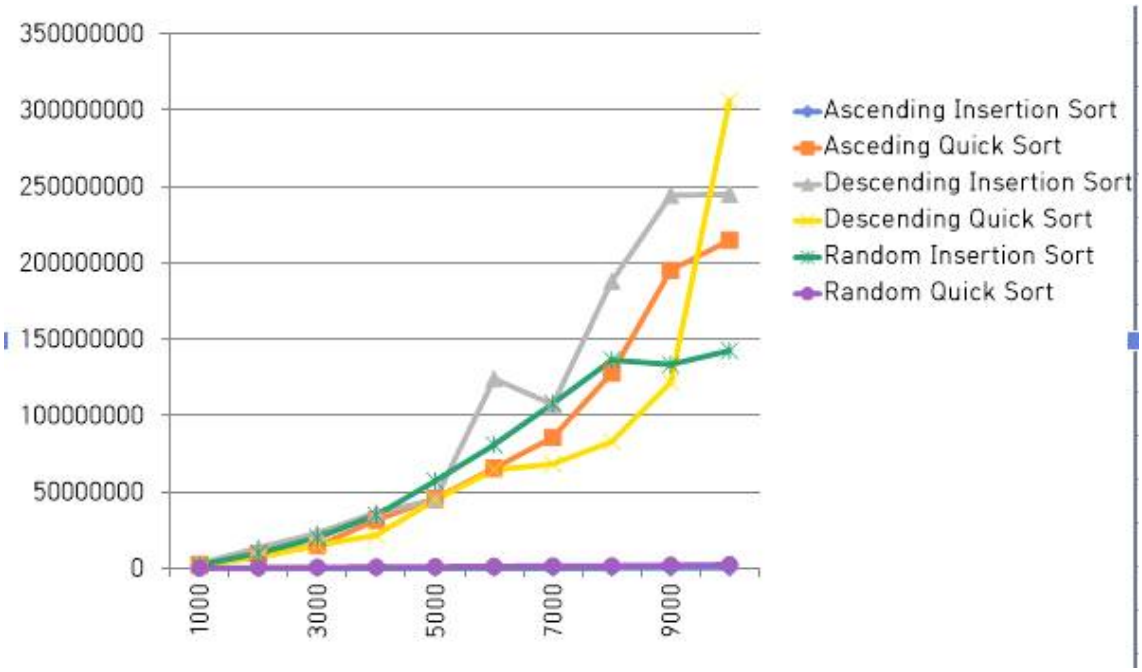
3. 실행 결과 분석

1) 입력과 출력

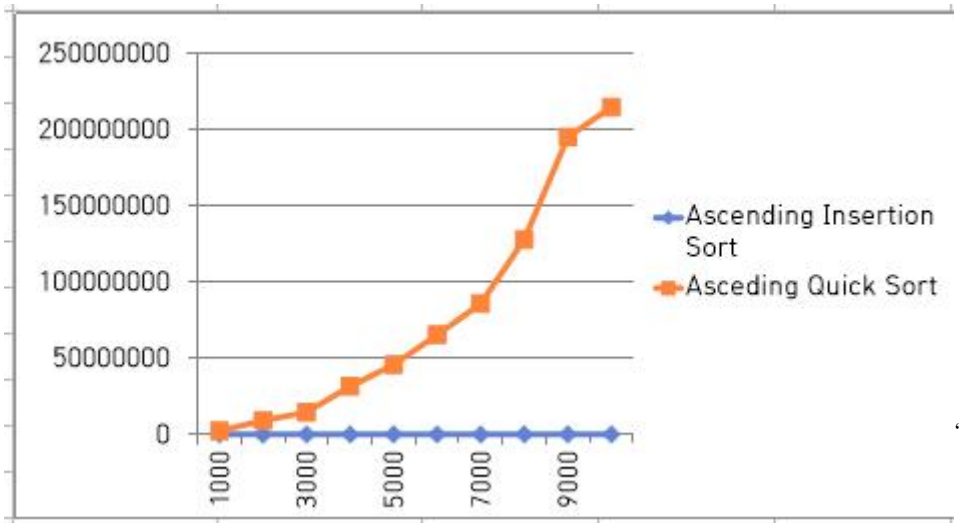
```
<<< 정렬 성능 비교 프로그램을 시작합니다 >>>

>> 2가지 정렬의 성능 비교: 삽입, 퀵 <<
> 오름차순데이터를 사용하여 실행한 측정:
        <Insertion Sort>    <Quick Sort>
[ 1000]           74538         2423777
[ 2000]           57060         9203155
[ 3000]           86362        14667064
[ 4000]           97156        31601630
[ 5000]          138795        45710943
[ 6000]          104354        65456886
[ 7000]          110008        85823809
[ 8000]          144964       127961544
[ 9000]          143422       195138977
[10000]          160386       214765660

> 내림차순데이터를 사용하여 실행한 측정:
        <Insertion Sort>    <Quick Sort>
[ 1000]         2992324         2142588
[ 2000]        13589601         7048744
[ 3000]        22988612        15150277
[ 4000]        36605972        21710154
[ 5000]        44884339        44756854
[ 6000]       124001766        64458074
[ 7000]       107306235        68323266
[ 8000]       187934989        82980048
[ 9000]       244137312       121752768
[10000]       244911996       306143850
```

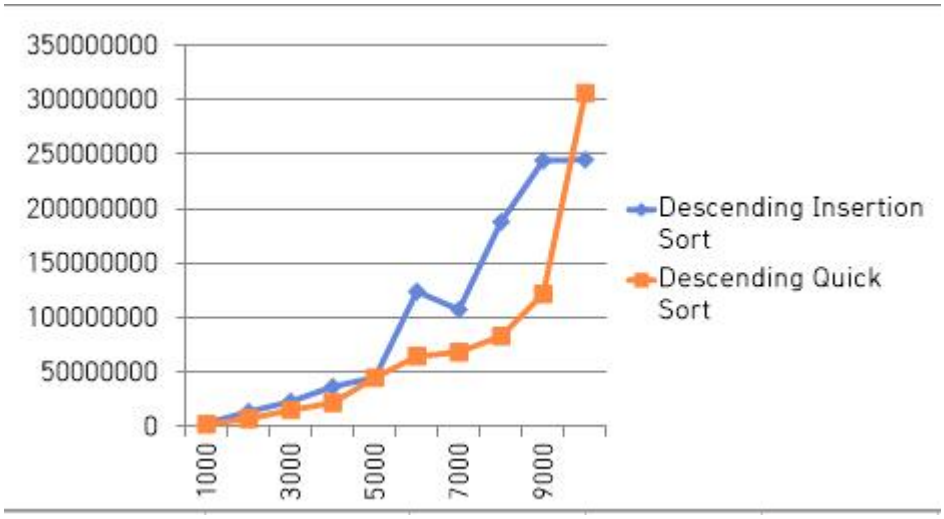
```
> 무작위데이터를 사용하여 실행한 측정:
        <Insertion Sort>    <Quick Sort>
[ 1000]         2160580          267310
[ 2000]         9992233          476017
[ 3000]        20138168          712996
[ 4000]        34810372          853334
[ 5000]        57182631         1057928
[ 6000]        80874472         1302105
[ 7000]       107882493         1542684
[ 8000]       136332956         1656290
[ 9000]       133441388         2090154
[10000]       142507805         2436115

<<< 정렬 성능 비교 프로그램을 종료합니다 >>>
```

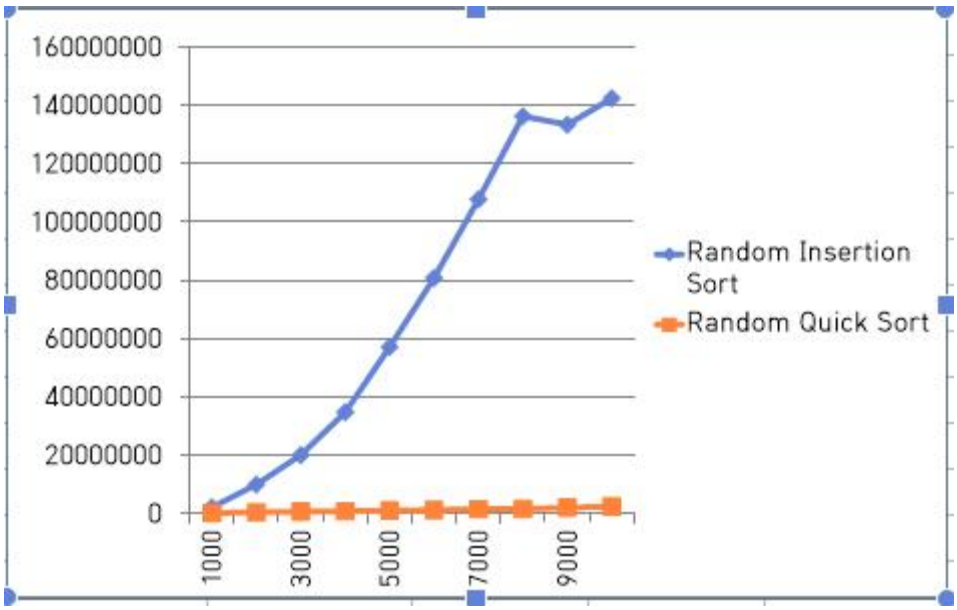
2) 결과 분석



random의 경우 quick Sort가 insertionSort보다 훨씬 빠르고, 오름차의 경우는 insertionSort가 훨씬 빠름. 둘다 비슷함.


250000000
200000000
150000000
100000000
50000000
0
1000
3000
5000
7000
9000
Ascending Insertion Sort
Asceding Quick Sort

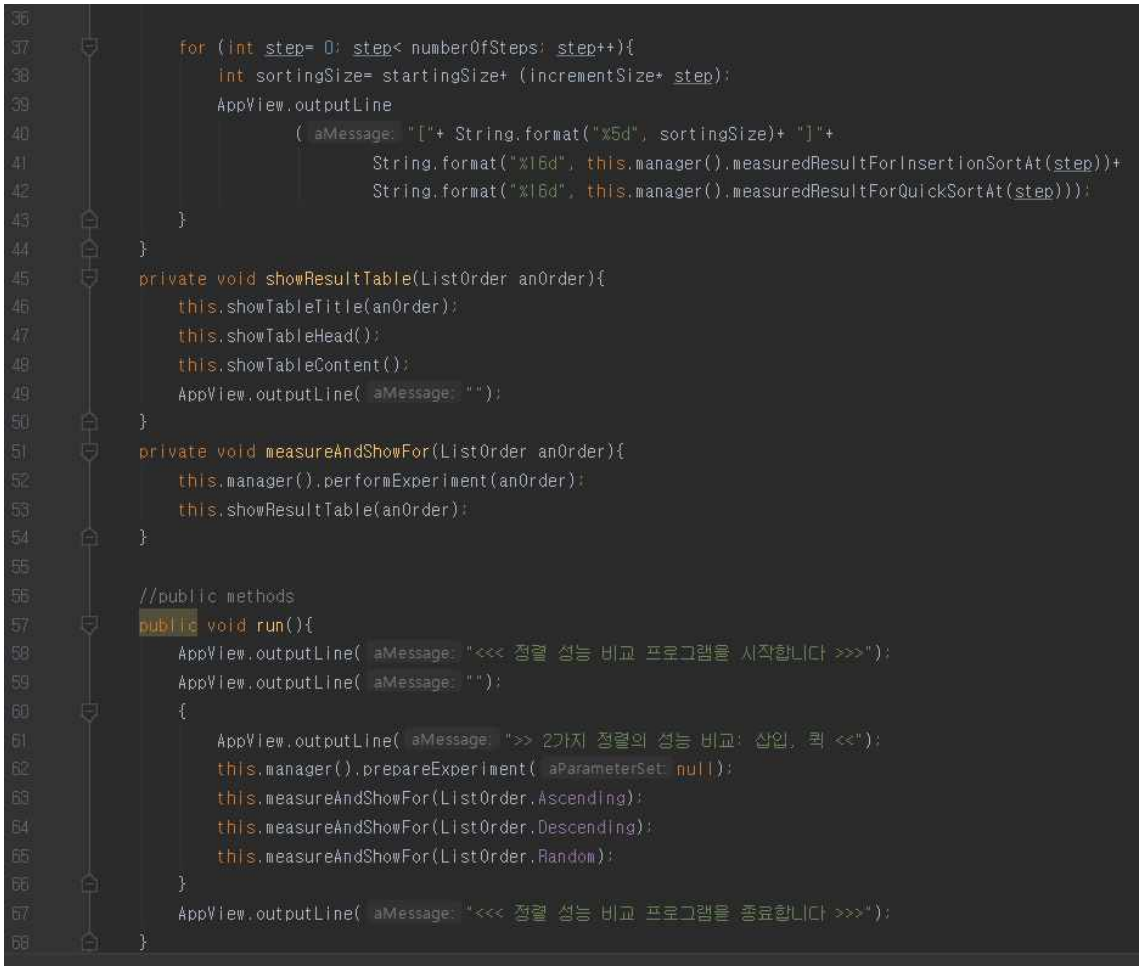

350000000
300000000
250000000
200000000
150000000
100000000
50000000
0
1000
3000
5000
7000
9000
Descending Insertion Sort
Descending Quick Sort


160000000
140000000
120000000
100000000
80000000
60000000
40000000
20000000
0
1000
3000
5000
7000
9000
Random Insertion Sort
Random Quick Sort

4. 소스 코드

AppController

```
package controller;

import model.ExperimentManager;
import view.AppView;

public class AppController {
    //private var
    private ExperimentManager _manager;

    //getter setter

    public ExperimentManager manager() { return _manager; }
    public void setManager(ExperimentManager newManager) { this._manager = newManager; }

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

    //private methods
    private void showTableTitle(ListOrder anOrder){
        AppView.outputLine
                ( aMessage: "> "+ anOrder.orderName()+ "데이터를 사용하여 실행한 측정:");
    }
    private void showTableHead(){
        AppView.outputLine
                ( aMessage: String.format("%8s", " ")+
                        String.format("%16s", "<Insertion Sort>")+
                        String.format("%16s", "<Quick Sort>"));
    }
    private void showTableContent(){
        int startingSize= this.manager().parameterSet().startingSize();
        int incrementSize= this.manager().parameterSet().incrementSize();
        int numberOfSteps=
                this.manager().parameterSet().numberOfSizeIncreasingSteps();
```



```
        for (int step= 0; step< numberOfSteps; step++){
            int sortingSize= startingSize+ (incrementSize* step);
            AppView.outputLine
                    ( aMessage: "["+ String.format("%5d", sortingSize)+ "]"+
                            String.format("%16d", this.manager().measuredResultForInsertionSortAt(step))+
                            String.format("%16d", this.manager().measuredResultForQuickSortAt(step)));
        }
    }
    private void showResultTable(ListOrder anOrder){
        this.showTableTitle(anOrder);
        this.showTableHead();
        this.showTableContent();
        AppView.outputLine( aMessage: "");
    }
    private void measureAndShowFor(ListOrder anOrder){
        this.manager().performExperiment(anOrder);
        this.showResultTable(anOrder);
    }

    //public methods
    public void run(){
        AppView.outputLine( aMessage: "<<< 정렬 성능 비교 프로그램을 시작합니다 >>>");
        AppView.outputLine( aMessage: "");
        {
            AppView.outputLine( aMessage: ">> 2가지 정렬의 성능 비교: 삽입, 퀵 <<");
            this.manager().prepareExperiment( aParameterSet: null);
            this.measureAndShowFor(ListOrder.Ascending);
            this.measureAndShowFor(ListOrder.Descending);
            this.measureAndShowFor(ListOrder.Random);
        }
        AppView.outputLine( aMessage: "<<< 정렬 성능 비교 프로그램을 종료합니다 >>>");
    }
```

ExperimentManager

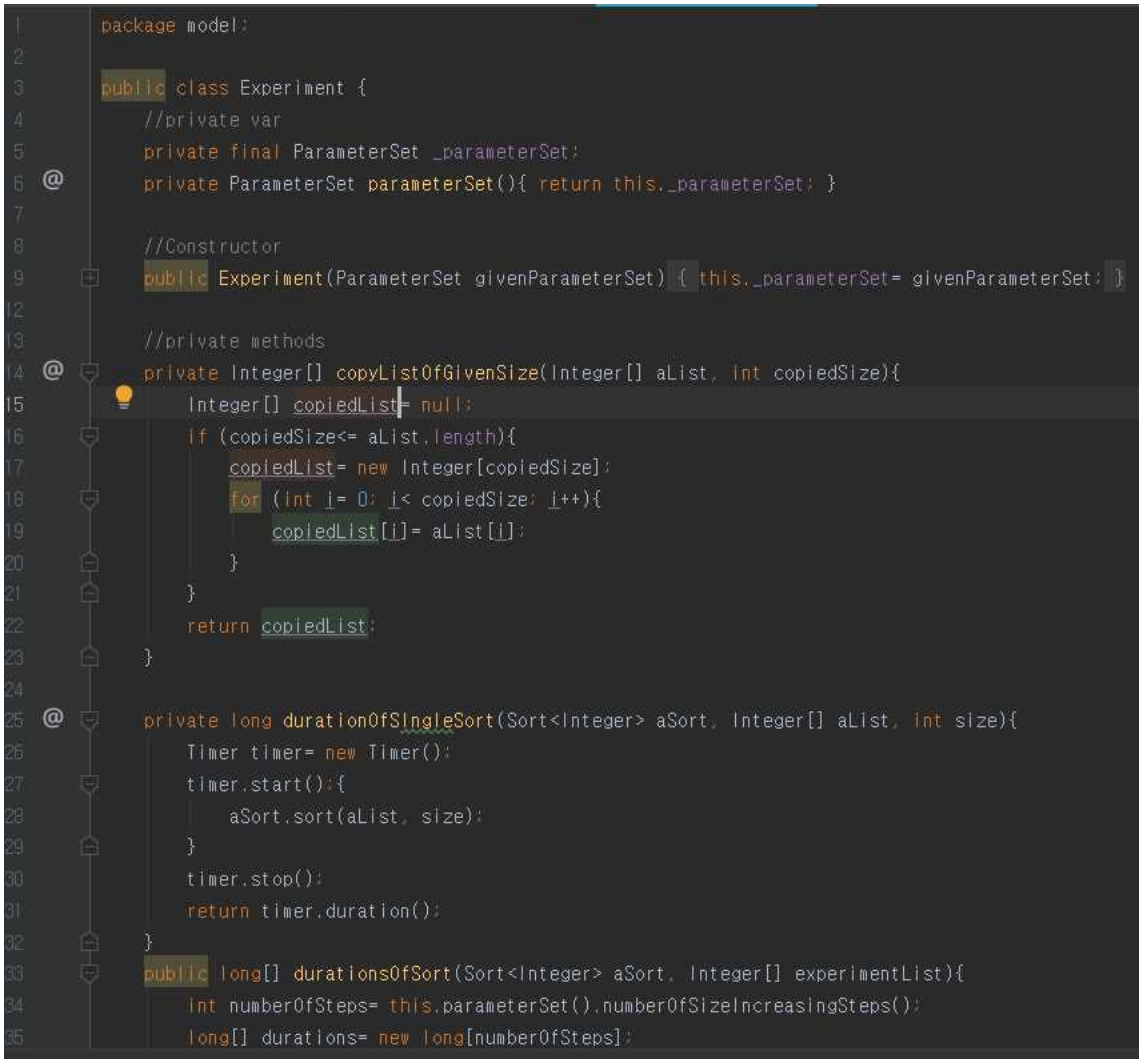
```
34      public Integer[] descendingList() { return _descendingList; }
35      public void setDescendingList(Integer[] newDescendingList) { this._descendingList = newDescendingList; }
36
37      public Integer[] randomList() { return _randomList;}
38      public void setRandomList(Integer[] newRandomList) { this._randomList = newRandomList; }
39
40      public long[] measuredResultForInsertionSort() { return _measuredResultForInsertionSort; }
43      public void setMeasuredResultForInsertionSort
44              (long[] newMeasuredResultForInsertionSort) {
45          this._measuredResultForInsertionSort = newMeasuredResultForInsertionSort;
46      }
47
48      public long[] measuredResultForQuickSort() {return _measuredResultForQuickSort; }
49      public void setMeasuredResultForQuickSort(long[] newMeasuredResultForQuickSort) {
50          this._measuredResultForQuickSort = newMeasuredResultForQuickSort;
51      }
52
53      //Constructor
54      public ExperimentManager(){ this.setParameterSetWithDefaults();}
55
56      //private methods
57      private void prepareExperimentLists(){
58          int maxDataSize= this.parameterSet().maxDataSize();
59
60          this.setAscendingList(DataGenerator.ascendingList(maxDataSize));
61          this.setDescendingList(DataGenerator.descendingList(maxDataSize));
62          this.setRandomList(DataGenerator.randomList(maxDataSize));
63      }
64
65      private void setParameterSetWithDefaults(){
66          this.setParameterSet(new ParameterSet(
67                  DEFAULT_STARTING_SIZE,
68                  DEFAULT_NUMBER_OF_SIZE_INCREASING_STEPS,
```

```java
package model;

import controller.ListOrder;

public class ExperimentManager {
    //constants
    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;

    private static final InsertionSort<Integer>
     INSERTION_SORT= new InsertionSort<~>();
    private static final QuickSort<Integer>
     QUICK_SORT= new QuickSort<~>();

    //private var
    private Experiment _experiment ;// 측정 실험을 실시할 객체
    private ParameterSet _parameterSet ; // 측정 실험에 사용할 매개변수 집합
    private Integer[] _ascendingList ; // 측정에서 정렬에 사용할 오름차순 데이터 리스트
    private Integer[] _descendingList ; // 측정에서 정렬에 사용할 내림차순 데이터 리스트
    private Integer[] _randomList ; // 측정에서 정렬에 사용할 무작위 데이터 리스트
    private long[] _measuredResultForInsertionSort ; // 삽입 정렬의 측정 결과 저장할 곳
    private long[] _measuredResultForQuickSort ; // 퀵 정렬의 측정 결과 저장할 곳

    public Experiment experiment() { return _experiment; }
    public void setExperiment(Experiment newExperiment) { this._experiment = newExperiment; }

    public ParameterSet parameterSet() { return _parameterSet; }
    public void setParameterSet(ParameterSet newParameterSet) { this._parameterSet = newParameterSet; }

    public Integer[] ascendingList() { return _ascendingList; }
    public void setAscendingList(Integer[] newAscendingList) { this._ascendingList = newAscendingList; }

```

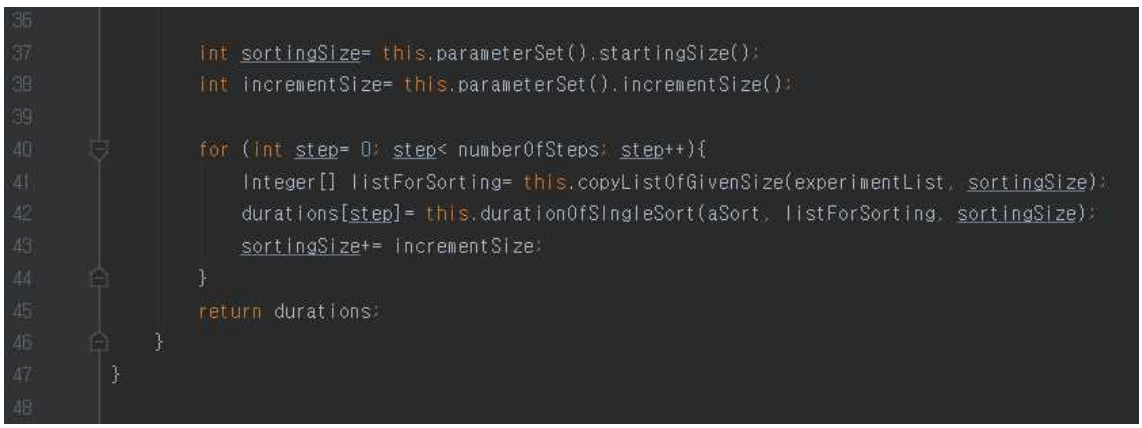
```
                DEFAULT_INCREMENT_SIZE));
    }

    private Integer[] exprimentListOfOrder(ListOrder anOrder){
        switch (anOrder){
            case Ascending:
                return this.ascendingList();
            case Descending:
                return this.descendingList();
            default:
                return this.randomList();
        }
    }
    public void prepareExperiment(ParameterSet aParameterSet){
        if (aParameterSet!= null){
            this.setParameterSet(aParameterSet);
        }
        this.setExperiment(new Experiment(this.parameterSet()));
        this.prepareExperimentLists();

        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){
        Integer[] experimentList= this.exprimentListOfOrder(anOrder);

        this.setMeasuredResultForInsertionSort(
```

```
                    this.experiment().durationsOfSort(INSERTION_SORT, experimentList));
            this.setMeasuredResultForQuickSort
                    (this.experiment().durationsOfSort(QUICK_SORT, experimentList));
        }
    }

```

Experiment



```
package model;

public class Experiment {
    //private var
    private final ParameterSet _parameterSet;
    private ParameterSet parameterSet(){ return this._parameterSet; }

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

    //private methods
    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;
    }

    private long durationOfSingleSort(Sort<Integer> aSort, Integer[] aList, int size){
        Timer timer= new Timer();
        timer.start();{
            aSort.sort(aList, size);
        }
        timer.stop();
        return timer.duration();
    }
    public long[] durationsOfSort(Sort<Integer> aSort, Integer[] experimentList){
        int numberOfSteps= this.parameterSet().numberOfSizeIncreasingSteps();
        long[] durations= new long[numberOfSteps];
```



```
        int sortingSize= this.parameterSet().startingSize();
        int incrementSize= this.parameterSet().incrementSize();

        for (int step= 0; step< numberOfSteps; step++){
            Integer[] listForSorting= this.copyListOfGivenSize(experimentList, sortingSize);
            durations[step]= this.durationOfSingleSort(aSort, listForSorting, sortingSize);
            sortingSize+= incrementSize;
        }
        return durations;
    }
}
```

ParameterSet

```java
package model;

public class ParameterSet {
    //private var
    private int _startingSize;
    private int _numberOfSizeIncreasingSteps;
    private int _incrementSize;

    //getter setter
    public int startingSize() { return _startingSize; }
    public void setStartingSize(int newStartingSize) { this._startingSize = newStartingSize; }

    public int numberOfSizeIncreasingSteps() { return _numberOfSizeIncreasingSteps; }
    public void setNumberOfSizeIncreasingSteps(int newNumberOfSizeIncreasingSteps) { this._numberOfSizeIncreasingSteps = newNumberOfSizeIncreasingSteps; }

    public int incrementSize() { return _incrementSize; }
    public void setIncrementSize(int newIncrementSize) { this._incrementSize = newIncrementSize; }

    //
    public int maxDataSize(){
        return (this.startingSize()+
                (this.incrementSize()*(this.numberOfSizeIncreasingSteps()-1)));
    }
    //Contsoructor
    public ParameterSet( int givenStartingSize,
                         int givenNumberOfSizeIncreasingSteps,
                         int givenIncrementSize){
        //작성
        this.setStartingSize(givenStartingSize);
        this.setNumberOfSizeIncreasingSteps(givenNumberOfSizeIncreasingSteps);
        this.setIncrementSize(givenIncrementSize);
    }
}
```

Timer

```java
package model;

public class Timer {
    private long _start;
    private long _stop;

    public long Start() { return _start; }
    private void setStart(long newStart) { this._start = newStart; }

    public long Stop() { return _stop; }
    private void setStop(long newStop) { this._stop = newStop; }

    public void start() { this.setStart(System.nanoTime()); }
    public void stop() { this.setStop(System.nanoTime()); }
    public long duration() { return this.Stop() - this.Start(); }
}
```