자료구조 실습 보고서

[제11주] 정렬 결과 검증

2021년 05월 17일

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
- InsertionSort<E> : 삽입정렬의 기능을 구현한다.
- QuickSort<E> : 퀵정렬의 기능을 구현한다.
- ListOrder (Enum) : 오름차순/내림차순/랜덤값 세가지 방법에 대한 Enum 클래스.

View :

- AppView : 프로그램의 입/출력을 담당한다.

Controller :

- AppController : Model을 통해 생성된 결과물을 AppView를 통해 출력한다.

(2) 함수 설명서

➔ 주요 알고리즘

1) 오름차순/내림차순/무작위 리스트 생성

```
private void validateWithAscendingOrderList() { // 오름차순 정렬 리스트
    this.setListOrder(ListOrder.Ascending); // ListOrder를 Ascending으로 지정
    this.setList(DataGenerator.ascendingOrderList(AppController.TEST_SIZE)); // 크기 10000의 오름차순 리스트 생성
    this.showFirstPartOfDataList(); // 첫 5개 원소 출력
    this.validateSortsAndShowResult(); // 결과값
}
```

각각의 오름차순/내림차순/무작위 리스트를 출력하는 함수가 존재하는데, 이 때 DataGenerator을 통해 주어진 크기만큼의 리스트를 생성한다.

```
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;
}
```

⇨ 오름차순 리스트 생성.

⇨ 먼저 배열 크기를 매개변수로 받아와서 유효한지 판단한다. (aSize>0)

⇨ for문을 통해 1-aSize만큼 각각의 인덱스에 각각의 값을 삽입한다.

```
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;
}
```

⇨ 내림차순 리스트 생성

⇨ 오름차순과 다른 부분은 aSize-1 만큼의 count 변수를 따로 생성하여 인덱스와 다르게 내림차순으로 수를 삽입한다.

```
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;
        }
```

⇨ 무작위로 랜덤값 리스트 생성

⇨ 먼저 오름차순으로 리스트를 생성한다.

```
    // 각 원소 list[i] 에 대해 무작위 위치 r 를 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;
```

⇨ 각 원소 list[i]에 대해 무작위 위치 r을 생성하여 바꾼다.

⇨ 해당 리스트를 반환

2) 리스트 정렬

- 삽입정렬

```
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.
```

⇨ 먼저 aSize값이 유효한지 판단

⇨ 가장 작은 값이 존재하는 위치를 저장할 변수 minLoc을 선언하고 0으로 초기화한다.

⇨ 주어진 배열에서 가장 작은 값을 찾고 0으로 삽입하여 -무한대 값 역할을 하도록 한다.

```
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;
```

⇨ 삽입할 원소를 i번째 값으로 선언하고 삽입할 위치를 i-1로 선언하여 둘을 비교한다.

⇨ 삽입할 위치의 원소값이 삽입할 원소보다 크다면 그 후 모든 값을 한칸씩 뒤로 옮기고 해당 위치에 값을 삽입한다.

- 퀵정렬

```
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.
    }
```

⇨ 반으로 나누어 각각의 반에 대해 퀵정렬을 실행한다. (재귀)

```
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.
}
```

⇨ partition 함수

⇨ left를 toRight로, right+1 위치를 toLeft로 설정한다. pivot은 함수를 통해 left 값을 받는다.

⇨ toRight은 left로 걸어가며 pivot보다 적은 값을 찾아 멈추고, toLeft는 right로 걸어가며 pivot보다 큰 값을 찾아 멈춘다.

⇨ 둘을 비교하여 swap한다.

⇨ toRight가 toLeft보다 커지면 pivot과 toLeft를 swap 한다.

(3) 종합 설명서

➔ 프로그램 실행 순서대로 설명해보자.

```
public class _DS11_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 알고리즘의 정렬 결과를 검증하는 프로그램을 시작합니다 >>>");
    AppView.outputLine("");
    AppView.outputLine(">정렬 결과의 검증 :");
    AppView.outputLine("");
    this.validateWithAscendingOrderList(); // 오름차순 정렬 리스트
    this.validateWithDescendingOrderList(); // 내림차순 정렬 리스트
    this.validateWithRandomOrderList(); // 랜덤 값 정렬 리스트
    AppView.outputLine("<<<정렬 알고리즘의 정렬 결과를 검증하는 프로그램을 종료합니다 >>>");
}
```

⇨ AppController의 run 함수이다. 프로그램 실행

⇨ 오름차순, 내림차순, 랜덤 값 리스트에 대한 실행을 각각의 함수를 통해 진행한다.

```
private void validateWithRandomOrderList() { // 랜덤 값 정렬 리스트
    this.setListOrder(ListOrder.Random); // ListOrder을 Random으로 지정
    this.setList(DataGenerator.randomOrderList(AppController.TEST_SIZE)); // 크기 10000의 랜덤 값 리스트 생성
    this.showFirstPartOfDataList(); // 첫 5개 원소 출력
    this.validateSortsAndShowResult(); // 결과값
}
```

⇨ 랜덤값 리스트를 실행하는 validateWithRandomOrderList 함수이다.

⇨ ListOrder에서 Random으로 지정한다.

⇨ 크기가 TEST_SIZE인 크기의 배열을 생성한다.

⇨ 처음 5개 원소를 출력한다.

⇨ 결과값을 출력한다.

```
private void showFirstPartOfDataList() { // 리스트의 첫 5개 원소를 출력
    AppView.output("[" + this.listOrder().orderName() + " 리스트] 의 앞 부분 : ");
    for (int i = 0; i < AppController.FIRST_PART_SIZE; i++) { // FIRST_PART_SIZE만큼 반복
        AppView.output(copyList(list())[i] + " "); // 복사된 리스트의 원소 출력
    }
    AppView.outputLine("");
}
```

⇨ 리스트의 첫 5개 원소를 출력하는 showFirshPartOfDataList 함수이다.

⇨ 미리 정의한 FIRST_PART_SIZE (5개) 만큼 반복문을 반복하여 원소를 출력한다.

```
private void validateSort(Sort<Integer> aSort) {
    Integer[] list = this.copyList(this._list);
    aSort.sort(list, list.length);
    this.showValidationMessage(aSort, list);
```

⇨ 동일한 리스트로 2번 정렬한다 (Quick, Insert)

⇨ 매번 원본 리스트를 복사하여 정렬한다.

⇨ copyList : 주어진 배열 객체의 복사본을 만든다.

```
private boolean sortedListIsValid(Integer[] aList) {
    // 주어진 aList 의 원소들이 오름차순으로 되어 있으면 true 를 돌려준다
    for (int i = 0; i < (aList.length - 1); i++) {
        if (aList[i].compareTo(aList[i + 1]) > 0) {
            return false; // 오름차순이 아닌 순서를 발견
        }
    }
    return true; // 리스트 전체가 오름차순으로 되어 있다
```

⇨ 주어진 aList 원소들이 오름차순으로 되어있으면 True를 반환한다.

⇨ 해당 함수가 true면 검증이 올바르다는 출력문을, false면 검증이 잘 못되었다는 출력문을 출력한다.

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.

➔ 단점

-

3. 실행 결과 분석

(1) 입력과 출력

```
<terminated> _DS11_201702039_오명주 [Java Application] C:₩Program Files₩Java₩jdk-12.0.1₩bin₩javaw.exe (2021. 5. 17.
<<<정렬 알고리즘의 정렬 결과를 검증하는 프로그램을 시작합니다 >>>

>정렬 결과의 검증 :

[오름차순 리스트] 의 앞 부분 : 0 1 2 3 4
[오름차순 리스트]를 [InsertionSort] 한 결과는 올바릅니다.
[오름차순 리스트]를 [QuickSort] 한 결과는 올바릅니다.

[내림차순 리스트] 의 앞 부분 : 9999 9998 9997 9996 9995
[내림차순 리스트]를 [InsertionSort] 한 결과는 올바릅니다.
[내림차순 리스트]를 [QuickSort] 한 결과는 올바릅니다.

[무작위 리스트] 의 앞 부분 : 8443 1927 1898 3494 3968
[무작위 리스트]를 [InsertionSort] 한 결과는 올바릅니다.
[무작위 리스트]를 [QuickSort] 한 결과는 올바릅니다.

<<<정렬 알고리즘의 정렬 결과를 검증하는 프로그램을 종료합니다 >>>
```

(2) 결과 분석 (자신의 논리적 평가, 기타 느낀 점)

- 구현한 코드에 대한 검증을 할 수 있어서 유용한 프로그램이라고 생각했다.
- .

➔ 생각해 볼 점

⇨ 추상클래스와 인터페이스의 차이점

- Abstract : 추상 메소드가 하나이상 포함 된 경우. new 연산자를 이용하여 객체를 생성할 수 있다. 단일 상속만 가능하다.
- Interface : 모든 메소드가 추상 메소드인 경우. 구현 객체가 같은 동작을 하는 것을 보장한다. 다중 상속이 가능하다.

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]

```
    private static final int TEST_SIZE = 10000;
    private static final int FIRST_PART_SIZE = 5;
    private static final InsertionSort<Integer> INSERTION_SORT = new InsertionSort<Integer>();
    private static final QuickSort<Integer> QUICK_SORT = new QuickSort<Integer>();

    // 비공개 변수들
    private Integer[] _list;
    private ListOrder _listOrder;

    // Getters/Setters
    private Integer[] list() {
        return this._list;
    }
    private void setList(Integer[] newList) {
        this._list = newList;
    }
    private ListOrder listOrder() {
        return this._listOrder;
    }
    private void setListOrder(ListOrder newListOrder) {
        this._listOrder = newListOrder;
    }

    // 생성자
    public AppController() {}

    public void run() {
        AppView.outputLine("<<<정렬 알고리즘의 정렬 결과를 검증하는 프로그램을 시작합니다 >>>");
        AppView.outputLine("");
        AppView.outputLine(">정렬 결과의 검증 :");
        AppView.outputLine("");
        this.validateWithAscendingOrderList(); // 오름차순 정렬 리스트
        this.validateWithDescendingOrderList(); // 내림차순 정렬 리스트
        this.validateWithRandomOrderList(); // 랜덤 값 정렬 리스트
        AppView.outputLine("<<<정렬 알고리즘의 정렬 결과를 검증하는 프로그램을 종료합니다 >>>");
    }

    private void validateWithAscendingOrderList() { // 오름차순 정렬 리스트
        this.setListOrder(ListOrder.Ascending); // ListOrder을 Ascending으로 지정
        this.setList(DataGenerator.ascendingOrderList(AppController.TEST_SIZE)); // 크기 10000의 오름차순 리스트 생성
        this.showFirstPartOfDataList(); // 첫 5개 원소 출력
        this.validateSortsAndShowResult(); // 결과값
    }

    private void validateWithDescendingOrderList() { // 내림차순 정렬 리스트
        this.setListOrder(ListOrder.Descending); // ListOrder을 Descending으로 지정
        this.setList(DataGenerator.descendingOrderList(AppController.TEST_SIZE)); // 크기 10000의 내림차순 리스트 생성
        this.showFirstPartOfDataList(); // 첫 5개 원소 출력
        this.validateSortsAndShowResult(); // 결과값
    }

    private void validateWithRandomOrderList() { // 랜덤 값 정렬 리스트
        this.setListOrder(ListOrder.Random); // ListOrder을 Random으로 지정
        this.setList(DataGenerator.randomOrderList(AppController.TEST_SIZE)); // 크기 10000의 랜덤 값 리스트 생성
        this.showFirstPartOfDataList(); // 첫 5개 원소 출력
        this.validateSortsAndShowResult(); // 결과값
    }

    private void showFirstPartOfDataList() { // 리스트의 첫 5개 원소를 출력
        AppView.output("[" + this.listOrder().orderName() + " 리스트] 의 앞 부분 : ");
        for (int i = 0; i < AppController.FIRST_PART_SIZE; i++) { // FIRST_PART_SIZE만큼 반복
            AppView.output(copyList(list())[i] + " "); // 복사된 리스트의 원소 출력
        }
        AppView.outputLine("");
    }

    private void validateSortsAndShowResult() { // 결과 출력
        this.validateSort(AppController.INSERTION_SORT); // 삽입정렬에 대한 결과
        this.validateSort(AppController.QUICK_SORT); // 퀵정렬에 대한 결과
        AppView.outputLine("");
    }

    private void validateSort(Sort<Integer> aSort) {
        Integer[] list = this.copyList(this._list);
        // 동일한 리스트로 여러 번 (실제로는 2번) 정렬하게 된다
        // 매번 원본 리스트를 복사하여 정렬한다
        aSort.sort(list, list.length);
        this.showValidationMessage(aSort, list);
    }

    private Integer[] copyList(Integer[] aList) {
        // 주어진 배열 객체 aList[] 의 복사본을 만들어 돌려준다
        // aList[]자체는 복사하지만
        // 배열의 원소 객체 자체는 복사하지 않고 공유한다
        Integer[] copiedList = new Integer[aList.length];
        for (int i = 0; i < aList.length; i++) {
            copiedList[i] = aList[i];
        }
        return copiedList;
    }

    private boolean sortedListIsValid(Integer[] aList) {
        // 주어진 aList 의 원소들이 오름차순으로 되어 있으면 true 를 돌려준다
        for (int i = 0; i < (aList.length - 1); i++) {
            if (aList[i].compareTo(aList[i + 1]) > 0) {
                return false; // 오름차순이 아닌 순서를 발견
            }
        }
        return true; // 리스트 전체가 오름차순으로 되어 있다
    }

    private void showValidationMessage(Sort<Integer> aSort, Integer[] aList) {
        AppView.output("[" + this.listOrder().orderName() + "리스트]를 [" + aSort.getClass().getSimpleName() + "] 한 결과는 ");
        if (this.sortedListIsValid(aList)) {
            AppView.outputLine("올바릅니다.");
        } else {
            AppView.outputLine("올바르지 않습니다.");
        }
    }
}
```

[AppController]