알고리즘 과제 보고서

- AL11, 정렬 성능 측정 과제

충남대학교 컴퓨터공학과
분반: 알고리즘 04 분반
학번: 201701975
이름: 구건모

과제의 주요이론

이번 과제는 정렬 알고리즘인 heapSort, insertionSort, quickSort 에 대하여 3가지 종류의 데이터(오름차순,내림차순,무작위)가 들어왔을 때 각각의 정렬 알고리즘의 복잡도가 어떠할지 측정값과 이론적인 복잡도를 예측한 예측값을 산출해보는 것이 과제의 주요 내용이었습니다.
정렬 알고리즘에 따라서 정렬을 하는 방식이 다르기 때문에 초기에 들어온 데이터가 어떤 모양이냐에 따라 빠르게 해결하거나 또는 다른 알고리즘에 비해 오래걸리게 될 수 있습니다. 예를 들면, insertion sort와 같이 이웃을 비교하여 Swap 하는 방식의 정렬 알고리즘의 경우 정렬된 데이터에 대한 성능을 비교했을 때 단 한번씩만의 비교로 알고리즘이 끝나버리기 때문에 최선의 효율을 보여주며, pivot을 기준으로 Divde and Conquer 하는 방식인 quick sort보다 유리합니다. 하지만 일반적인 random 한 종류의 dataSet이 들어왔을 때에는 값의 위치를 하나씩 바꿔가면서 다시 비교를 반복하는 Insertion Sort 보다 QuickSort가 평균적으로 더 좋은 성능을 보여줍니다.
이번 과제에서 이러한 데이터 set의 종류와 특정 sorting 알고리즘에 대해서 정렬이 완료되는 때까지 소요되는 시간을 측정하여 확인하고 이론적인 복잡도를 계산하여 두 값을 비교해보는 과정을 구현하는 것이 주제였습니다.

Timer

```java
package experiment;

public final class Timer {
    private static long startTime;
    private static long stopTime;

    private Timer() {
    };

    public static void start() {
        Timer.startTime = System.nanoTime();
    }

    public static void stop() {
        Timer.stopTime = System.nanoTime();
    }

    public static long duration() {
        return (Timer.stopTime - Timer.startTime) / 1000;
    }


}
```

- Class Timer

sorting에 걸린 시간을 측정하기 위해 정의된 Timer 클래스 입니다.

System.nanoTime을 이용하여 끝난 시점과 시작한 시점의 차이를 구하여 sorting 시 duration을 구하는데 사용됩니다.

타이머는 동시에 여러개를 측정하지 않고 하나가 끝나면 다음을 측정하는 방식으로 사용될 것이므로 Static class로 정의됩니다.

ListOrder

```java
package experiment;

public enum ListOrder {
    Random, Ascending, Descending;

    public String orderName() {
        if (this.equals(ListOrder.Random)) {
            return "무작위";
        } else if (this.equals(ListOrder.Ascending)) {
            return "오름차순";
        } else {
            return "내림차순";
        }
    }
}

```

- Enum ListOrder

Random, Ascending, Descending 등 사용할 order에 대한 한글명을 return 하기 위해 정의된 enum class 입니다

QuickSortByPivotLeft

```java
package sort;

public class QuickSortByPivotLeft <E extends Comparable<E>> extends QuickSort<E> {

    public QuickSortByPivotLeft(boolean givenSortingOrder) {
        super(givenSortingOrder);
        // TODO Auto-generated constructor stub
    }

    @Override
    protected int pivot(E[] aList, int left, int right) {
        // TODO Auto-generated method stub
        return left;

    }


}

```

- Class QuickSortByPivotLeft

QuickSort 중 pivot을 left로 하여 sorting 하는 QuickSortByPivotLeft 클래스 입니다. QuickSort를 extends 하며 pivot() 메서드가 left 값을 리턴하도록 오버라이드하여 구현됩니다.

DataGenerator

```java
package experiment;

import java.util.Random;

public final class DataGenerator {
    private DataGenerator() {

    };

    public static Integer[] ascendingList(int aSize) {
        if (aSize > 0) {
            Integer[] list = new Integer[aSize];
            for (int i = 0; i < aSize; i++) {
                list[i] = i;

            }
            return list;

        }

        return null;
    }

    public static Integer[] descendingList(int aSize) {
        if (aSize > 0) {
            Integer[] list = new Integer[aSize];
            Random random = new Random();
            for (int i = 0; i < aSize; i++) {
                list[i] = aSize - i - 1;
            }
            return list;

        }

        return null;
    }

    public static Integer[] randomList(int aSize) {
        if (aSize > 0) {
            Integer[] list = new Integer[aSize];
            Random random = new Random();
            for (int i = 0; i < aSize; i++) {
                list[i] = random.nextInt(aSize);
```

Class DataGenerator

DataGenerator 클래스는 이번 실습에 필요한 데이터를 Random 라이브러리를 이용해 난수를 발생시켜서 만들어줍니다.
이번 실습에 사용될 데이터 종류는 Ascending, Descending, Random 세 가지이므로 값들을 각각의 방식에 맞게 구성하여 list를 return 합니다. randomList를 만들어 낼 때 random을 이용하게 됩니다.

ExperimentDataSet

```
42    public ExperimentDataSet() {
43        this.setRandomList(null);
44        this.setAscendingList(null);
45        this.setDescendingList(null);
46    }
47
48    public ExperimentDataSet(int givenMaxDataSize) {
49        if (!this.generate(givenMaxDataSize)) {
50            this.setRandomList(null);
51            this.setAscendingList(null);
52            this.setDescendingList(null);
53        }
54    }
55
56    public boolean generate(int aMaxDataSize) {
57
58        if (aMaxDataSize <= 0) {
59            return false;
60
61        } else {
62            this.setMaxDataSize(aMaxDataSize);
63            this.setRandomList(DataGenerator.randomList(this.maxDataSize()));
64            this.setAscendingList(DataGenerator.ascendingList(this.maxDataSize()));
65            this.setDescendingList(DataGenerator.descendingList(this.maxDataSize()));
66            return true;
67        }
68    }
69
70    public Integer[] listWithOrder(ListOrder anOrder) {
71        if (anOrder.equals(ListOrder.Random)) {
72            return this.randomList();
73        } else if (anOrder.equals(ListOrder.Ascending)) {
74            return this.ascendingList();
75        } else if (anOrder.equals(ListOrder.Descending)) {
76            return this.descendingList();
77        } else {
78            return null;
79        }
80    }
81
82 }
```

- Class ExperimentDataSet

ExperimentDataSet 클래스는 sorting에 사용될 dataset을 DataGenerator 클래스를 이용해 실험에 사용될 3가지의 데이터인 Random, Ascending, Descending을 만들고 초기화하는 역할을 수행합니다.

ParameterSetForMeasurement

```java
package experiment;

public class ParameterSetForMeasurement extends ParameterSetForIteration {
    private static final int DEFAULT_NUMBER_OF_REPETITION_OF_SINGLE_SORT = 1;

    private int _numberOfRepititionOfSingleSort;

    public int numberOfRepititionOfSingleSort() {
        return this._numberOfRepititionOfSingleSort;

    }

    public void setNumberOfRepititionOfSingleSort(int newNumberOfRepititionOfSingleSort) {
        this._numberOfRepititionOfSingleSort = newNumberOfRepititionOfSingleSort;
    }

    public ParameterSetForMeasurement() {
        super();
        this.setNumberOfRepititionOfSingleSort(DEFAULT_NUMBER_OF_REPETITION_OF_SINGLE_SORT);
    }

    public ParameterSetForMeasurement(int givenStartingSize, int givenNumberOfIteration, int givenIncrementSize,
            int givenNumberOfRepititionOfSingleSort) {
        super(givenStartingSize, givenNumberOfIteration, givenIncrementSize);
        this.setNumberOfRepititionOfSingleSort(givenNumberOfRepititionOfSingleSort);
    }
}
```

- Class ParameterSetForMeasurement

ParameterSetForMeasurement 클래스는 실험을 정해진 횟수만큼 반복하면서 초기 데이터 크기에서부터 설정한 증가 크기만큼씩 데이터를 늘려가기 위해 필요한 기능을 수행하는 클래스입니다.

실험을 얼마나 반복지에 대한 값인 _numberOfReptitionOfSingleSort 라는 멤버변수가 정의되어 있으며 ParameterForIteration을 extends 하였기 때문에 해당 클래스에 정의된 시작 데이터 사이즈와 iteration 마다 증가시킬 크기, 그리고 총 반복 횟수를 기본적으로 가지고 있으며, 각 값들을 사용자가 설정하여 초기화할 수 있도록 위의 값을 받는 생성자도 정의되어 있습니다.

Estimation

```java
package experiment;

public final class Estimation {
    private Estimation() {
    };

    public static long[] estimateByLinear(long[] measuredTimes, ParameterSetForMeasurement aParameterSet) {
        int length = aParameterSet.numberOfIteration();
        long[] estimatedTimes = new long[length];
        double estimatedCoefficient = (double) measuredTimes[length - 1] / (double) length;
        for (int i = 1; i <= length; i++) {
            estimatedTimes[i - 1] = (long) (estimatedCoefficient * (double) (i));
        }
        return estimatedTimes;

    }

    public static long[] estimateByQudratic(long[] measuredTimes, ParameterSetForMeasurement aParameterSet) {
        int length = aParameterSet.numberOfIteration();
        long[] estimatedTimes = new long[length];
        double estimatedCoefficient = (double) measuredTimes[length - 1] / (double) (length * length);
        for (int i = 1; i <= length; i++) {
            estimatedTimes[i - 1] = (long) (estimatedCoefficient * (double) (i * i));
        }
        return estimatedTimes;

    }

    public static long[] estimateByNLogN(long[] measuredTimes, ParameterSetForMeasurement aParameterSet) {
        int length = aParameterSet.numberOfIteration();
        int incrementSize = aParameterSet.incrementSize();
        int N = aParameterSet.maxDataSize();
        long[] estimatedTimes = new long[length];
        double estimatedCoefficient = (double) measuredTimes[length - 1] / ((double) (N) * Math.log((double) (N)));
        for (int i = 1; i <= length; i++) {
            estimatedTimes[i - 1] = (long) (estimatedCoefficient * (double) (i * incrementSize)
                    * (Math.log((double) (i * incrementSize))));
        }
        return estimatedTimes;
    }

}
```

- Class Estimation

Estimation은 각 데이터의 종류와 Sorting 방법에 따른 이론적인 시간복잡도 계산하여 성능을 추정하기 위한 클래스로, 주어진 데이터 set과 적용되는 정렬 알고리즘에 따라서 복잡도를 선택하여 estimate 하게 됩니다. 이번 실습에서 필요한 복잡도의 종류 3가지 (Qudratic(n^2), NlogN, Linear(n))를 구하는 과정이 구현되어 있습니다. Estimation은 데이터의 크기가 클수록 오차가 적으므로 estimatedCoefficient를 구할 때 가장 큰 데이터의 측정 시간을 가지고 구하도록 구현되어 있습니다.

Experiment

```
1  package experiment;
2
3  import app.AppView;
6  public class Experiment {
7      private static final boolean DEBUG_MODE = false;
8
9      private static void showDebugMessage(String aMessage) {
10         if (Experiment.DEBUG_MODE) {
11             AppView.outputDebugMessage(aMessage);
12         }
13     }
14
15     private static Integer[] copyListOfGivenSize(Integer[] aList, int givenSize) {
16         if (givenSize <= aList.length) {
17             Integer[] copiedList = new Integer[givenSize];
18             for (int i = 0; i < givenSize; i++) {
19                 copiedList[i] = aList[i];
20             }
21             return copiedList;
22         }
23         return null;
24     }
25
26     public static long durationOfSingleSort(Sort<Integer> aSort, Integer[] aList) {
27         Timer.start();
28         {
29             aSort.sort(aList);
30         }
31         Timer.stop();
32         return Timer.duration();
33     }
34
35     private ParameterSetForMeasurement _parameterSetForMeasurement;
36
37     private ParameterSetForMeasurement parameterSetForMeasurement() {
38         return this._parameterSetForMeasurement;
39     }
40
41     private void setParameterSetForMeasurement(ParameterSetForMeasurement newParameterSet) {
42         this._parameterSetForMeasurement = newParameterSet;
43     }
44
45     public Experiment(ParameterSetForMeasurement givenParameterSet) {
```

- Class Experiment

정렬방법에 따라 데이터를 sorting 하고 시간을 측정한느 기능을 담당하는 Experiment Class 입니다. DurationOfSingleSort 메서드는 Sorting 방법에 대해 정렬을 수행하는데 걸린 시간을 Timer를 이용하여 측정하고 리턴합니다. DurationOfSort에서는 list를 특정 sorting 알고리즘으로 sorting 하였을 때의 duration을 Data 크기별로 측정하여 duration 배열에 저장한 후 return 합니다.

ExperimentManager

```java
package experiment;

import app.AppView;

public abstract class ExperimentManager {

    private static final boolean DEBUG_MODE = false;

    private static void showDebugMessage(String aMessage) {
        if (ExperimentManager.DEBUG_MODE) {
            AppView.outputDebugMessage(aMessage);
        }
    }

    protected static final int DEFUALT_NUMBER_OF_ITERATION = 10;
    protected static final int DEFUALT_INCREMENT_SIZE = 1000;
    protected static final int DEFUALT_STARTING_SIZE = DEFUALT_INCREMENT_SIZE;
    protected static final int DEFUALT_NUMBER_OF_REPITITION_OF_SINGLE_SORT = 1;

    private ExperimentDataSet _dataSet;
    private Experiment _experiment;
    private ParameterSetForMeasurement _parameterSetForMeasurement;

    public ExperimentDataSet dataSet() {
        return this._dataSet;

    }

    protected void setDataSet(ExperimentDataSet newDataSet) {
        this._dataSet = newDataSet;
    }

    protected Experiment experiment() {
        return this._experiment;
    }

    protected void setExperiment(Experiment newExperiment) {
        this._experiment = newExperiment;
    }

    public ParameterSetForMeasurement parameterSetForMeasurement() {
        return this._parameterSetForMeasurement;
    }
```

- Class ExperimentManager

사용자가 실험을 구성하고 운영하는 기능을 담당하는 ExperimentManager Class 입니다. 해당 클래스에는 실험에 사용할 변수들이 멤버 변수로 정의되어 있고, 실험에 쓰일 데이터들을 저장하는 변수가 정의되어 있습니다. prepareExperiment 메서드로 실험에 필요한 데이터를 만들어주고, 실험을 시행하는 performExperiment 메소드가 abstract method로 정의되어 있습니다. 이후에 ExperimentManagerForThreeSorts 클래스에서 구현하여 사용됩니다.

ExperimentManagerForThreeSorts

```
77      protected void performMeasuring(ListOrder anOrder) {
78          // TODO Auto-generated method stub
79
80          Integer[] experimentList = this.dataSet().listWithOrder(anOrder);
81          this.setMeasurementForInsertionSort(
82                  this.experiment().durationOfSort(ExperimentManagerForThreeSorts.InsertionSort, experimentList));
83          ExperimentManagerForThreeSorts.showDebugMessage("[Debug] end of Insertion Sort\n");
84
85          this.setMeasurementForQuickSort(
86                  this.experiment().durationOfSort(ExperimentManagerForThreeSorts.QuickSort, experimentList));
87          ExperimentManagerForThreeSorts.showDebugMessage("[Debug] end of Quick Sort\n");
88
89          this.setMeasurementForHeapSort(
90                  this.experiment().durationOfSort(ExperimentManagerForThreeSorts.HeapSort, experimentList));
91          ExperimentManagerForThreeSorts.showDebugMessage("[Debug] end of HeapSort Sort\n");
92
93      }
94
95      private void estimateForRandomList() {
96          this.setEstimationForInsertionSort(
97                  Estimation.estimateByQudratic(this.measurementForInsertionSort(), this.parameterSetForMeasurement()));
98          this.setEstimationForQuickSort(
99                  Estimation.estimateByNLogN(this.measurementForQuickSort(), this.parameterSetForMeasurement()));
100         this.setEstimationForHeapSort(
101                 Estimation.estimateByNLogN(this.measurementForHeapSort(), this.parameterSetForMeasurement()));
102     }
103
104     private void estimateForAscendingList() {
105         this.setEstimationForInsertionSort(
106                 Estimation.estimateByLinear(this.measurementForInsertionSort(), this.parameterSetForMeasurement()));
107         this.setEstimationForQuickSort(
108                 Estimation.estimateByQudratic(this.measurementForQuickSort(), this.parameterSetForMeasurement()));
109         this.setEstimationForHeapSort(
110                 Estimation.estimateByNLogN(this.measurementForHeapSort(), this.parameterSetForMeasurement()));
111     }
112
113     private void estimateForDescendingList() {
114         this.setEstimationForInsertionSort(
115                 Estimation.estimateByQudratic(this.measurementForInsertionSort(), this.parameterSetForMeasurement()));
116         this.setEstimationForQuickSort(
117                 Estimation.estimateByQudratic(this.measurementForQuickSort(), this.parameterSetForMeasurement()));
118         this.setEstimationForHeapSort(
119                 Estimation.estimateByNLogN(this.measurementForHeapSort(), this.parameterSetForMeasurement()));
```

- Class ExperimentManagerForThreeSorts

ExperimentManagerForThreeSorts 는 실제로 이번 과제에서 세가지 sorting 방식(InsertionSort, QuickSort, HeapSort)에 대해 실험하기 위해 구현된 클래스로 ExperimentManager를 extends 하여 구현되었으며, 각각의 데이터 종류와 Sorting 방식에 따라 적절한 Estimation 이 이루어집니다.

결과화면

```
<terminated> _Main_AL11_201701975_구건모 [Java Application] C:₩Users₩gmku1₩.p2₩pool₩plugins₩org.eclipse.justj.openjdk.hotspot.jre.full.win3
<<< 정렬 성능 비교 프로그램을 시작합니다 >>>

>> 3 가지 정렬의 성능 비교: 삽입, 퀵, 힙 <<
> 무작위 데이터에 대한 측정:
                  <Insertion Sort>          <Quick Sort>            <Heap Sort>
 데이터 크기        Measure (Estimate)      Measure (Estimate)      Measure (Estimate)
[   1000]            1482 (     775)         136 (      81)          131 (     140)
[   2000]            3712 (    3100)         200 (     179)          297 (     309)
[   3000]            6661 (    6975)         454 (     282)          469 (     489)
[   4000]           11670 (   12400)         499 (     390)          583 (     675)
[   5000]           18917 (   19376)         552 (     501)          726 (     867)
[   6000]           27786 (   27901)         567 (     614)         1133 (    1063)
[   7000]           37521 (   37976)         736 (     730)         1109 (    1262)
[   8000]           49380 (   49602)         817 (     846)         1278 (    1464)
[   9000]           74549 (   62778)         934 (     965)         1680 (    1669)
[  10000]           77504 (   77504)        1085 (    1085)         1876 (    1876)

> 오름차순 데이터에 대한 측정:
                  <Insertion Sort>          <Quick Sort>            <Heap Sort>
 데이터 크기        Measure (Estimate)      Measure (Estimate)      Measure (Estimate)
[   1000]               5 (       4)          738 (     321)          222 (     108)
[   2000]               8 (       9)         2887 (    1285)          242 (     239)
[   3000]              11 (      13)         3188 (    2892)          370 (     377)
[   4000]              14 (      18)         5422 (    5141)          604 (     521)
[   5000]              18 (      22)         8610 (    8033)          776 (     669)
[   6000]              21 (      27)        12375 (   11568)          995 (     821)
[   7000]              25 (      31)        17065 (   15746)         1037 (     975)
[   8000]              29 (      36)        20991 (   20566)         1216 (    1131)
[   9000]              34 (      40)        26994 (   26029)         1322 (    1289)
[  10000]              45 (      45)        32135 (   32135)         1449 (    1449)

> 내림차순 데이터에 대한 측정:
                  <Insertion Sort>          <Quick Sort>            <Heap Sort>
 데이터 크기        Measure (Estimate)      Measure (Estimate)      Measure (Estimate)
[   1000]            1503 (    1368)         424 (     356)          101 (      73)
[   2000]            5387 (    5475)        1490 (    1424)          159 (     160)
[   3000]           12182 (   12318)         3317 (    3205)          245 (     254)
[   4000]           21782 (   21900)         5582 (    5698)          344 (     351)
[   5000]           35643 (   34219)         8728 (    8903)          437 (     450)
[   6000]           49088 (   49275)        12534 (   12821)          537 (     552)
[   7000]           66935 (   67069)        17133 (   17450)          639 (     656)
[   8000]           87091 (   87600)        22476 (   22792)          733 (     761)
[   9000]          116086 (  110869)        28794 (   28847)          842 (     867)
[  10000]          136876 (  136876)        35614 (   35614)          975 (     975)

<<< 정렬 성능 비교 프로그램을 종료합니다 >>>
```

결과분석 및 생각할 점

결과를 확인해보면 데이터량이 커지면 커질수록 측정값과

이론적인 복잡도 사이의 오차가 적어지는 것을 확인할 수 있습니다.

```
> 오름차순 데이터에 대한 측정:
                <Insertion Sort>          <Quick Sort>            <Heap Sort>
 데이터 크기       Measure (Estimate)       Measure (Estimate)       Measure (Estimate)
[   1000]             5 (       4)           738 (     321)           222 (     108)
[   2000]             8 (       9)          2887 (    1285)           242 (     239)
[   3000]            11 (      13)          3188 (    2892)           370 (     377)
[   4000]            14 (      18)          5422 (    5141)           604 (     521)
[   5000]            18 (      22)          8610 (    8033)           776 (     669)
[   6000]            21 (      27)         12375 (   11568)           995 (     821)
[   7000]            25 (      31)         17065 (   15746)          1037 (     975)
[   8000]            29 (      36)         20991 (   20566)          1216 (    1131)
[   9000]            34 (      40)         26994 (   26029)          1322 (    1289)
[  10000]            45 (      45)         32135 (   32135)          1449 (    1449)
```

InsertionSort의 경우 오름차순 데이터를 sorting 할 때,

이미 오름차순으로 정렬된 데이터이기 때문에 한번씩만의 비교가 이루어진 후 마치므로(비교횟수 n-1번) 최선의 성능인 복잡도 O(n) 을 가지게 됩니다. 실제 이번 실습 결과에서도 다른 sorting 알고리즘과는 확연히 낮은 값의 measurement를 보여주는 것을 확인해 볼 수 있습니다.

```
> 내림차순 데이터에 대한 측정:
                <Insertion Sort>          <Quick Sort>            <Heap Sort>
 데이터 크기       Measure (Estimate)       Measure (Estimate)       Measure (Estimate)
[   1000]          1503 (    1368)           424 (     356)           101 (      73)
[   2000]          5387 (    5475)          1490 (    1424)           159 (     160)
[   3000]         12182 (   12318)          3317 (    3205)           245 (     254)
[   4000]         21782 (   21900)          5582 (    5698)           344 (     351)
[   5000]         35643 (   34219)          8728 (    8903)           437 (     450)
[   6000]         49088 (   49275)         12534 (   12821)           537 (     552)
[   7000]         66935 (   67069)         17133 (   17450)           639 (     656)
[   8000]         87091 (   87600)         22476 (   22792)           733 (     761)
[   9000]        116086 (  110869)         28794 (   28847)           842 (     867)
[  10000]        136876 (  136876)         35614 (   35614)           975 (     975)
```

데이터가 내림차순일 경우 이론적으로 heap 정렬이 가장 좋은

성능을 보이는데 실제 결과에서도 heap에 대한 measurement가 다른 두개의 알고리즘보다 현저히 빠른 것을 볼 수 있습니다.

```
> 무작위 데이터에 대한 측정:
                <Insertion Sort>          <Quick Sort>            <Heap Sort>
 데이터 크기       Measure (Estimate)       Measure (Estimate)       Measure (Estimate)
[   1000]          1482 (     775)           136 (      81)           131 (     140)
[   2000]          3712 (    3100)           200 (     179)           297 (     309)
[   3000]          6661 (    6975)           454 (     282)           469 (     489)
[   4000]         11670 (   12400)           499 (     390)           583 (     675)
[   5000]         18917 (   19376)           552 (     501)           726 (     867)
[   6000]         27786 (   27901)           567 (     614)          1133 (    1063)
[   7000]         37521 (   37976)           736 (     730)          1109 (    1262)
[   8000]         49380 (   49602)           817 (     846)          1278 (    1464)
[   9000]         74549 (   62778)           934 (     965)          1680 (    1669)
[  10000]         77504 (   77504)          1085 (    1085)          1876 (    1876)
```

무작위 데이터의 경우에서는 Quick sort와 Heap Sort가 전반적으로 좋은 성능을 가지는데, 그에 반해 모든 이웃값을 비교 후 swap 하는 과정을 반복하는 삽입정렬의 경우, 퀵이나 힙 정렬에 비해 무작위 데이터가 들어왔을 때는 전반적으로 느린 모습을 보여주는 것을 확인해 볼 수 있습니다.