DATA STRUCTURES AND ALGORITHMS

LAB #3

Thanh-Sach LE
`LTSACH@hcmut.edu.vn`

Introduction

You study sorting algorithms this week. You learned how to sort arrays of data elements of **specific data type**, for examples, sorting points in 2D-space by their x-coordinates. In the example, each data element is an instance of class **Point2D**, and we hardcode the comparison between the x-coordinate of points. Thereby, we have to **re-write sorting algorithms** for each type of data elements; it's boring to do by such the way.

How do we code the sorting algorithm in a **general way** and then we can use them for different type of the data element? It is important to do in such the way for the data structures being studied in subsequent lectures/chapters, for examples, list, tree, and graph of any data type.

Therefore, the objectives of this lab are as follows:

1. Practice generics programming in Java. You will be required to design sorting algorithms that are able to work with any type of data elements.
2. Evaluate the executation time of sorting algorithms and then show it on console and create graphs of the executation time (wrt the count of data elements) for each sorting algorithms.

1 Design of sorting algorithms

It is important to note that we **do not want to hardcode** the type of data elements in sorting algorithms, so we have to **parameterize** it the definition of the sorting algorithms. We need **generics programming** for that task.

We should think about the way that programmers using our sorting API to guide the design of the API. In this lab, programmers can perform the sorting by code given in Figure 1.

In Figure 1:

- Line 1-2: The programmer create a array of instances, each is of type **Point2D**. He can cteate array of **any data type**, Point2D is just for example.
- Line 3: He use **Straight Insertion Sort** to sort the data given in array points. Now, **Point2D** is the argument passed to **StraightInsertionSort**.
- Line 4: He invokes method sort in **StraightInsertionSort** to do sorting. We expect that he must pass the array being sorted and a **comparator object**. **comparator object - why and how?**. To implement the sorting task, all sorting algorithms must know how to compare between any pair of data elements. The comparator object is designed for that purpose. Fortunately, Java have an interface named Comparator[1] for supporting the comparison capability for any other data type.

[1]java.util.Comparator

```java
Point2D[] points = new Point2D[N]; //N: data count
//here: initialize data elements in 'points'
StraightInsertSort<Point2D> algorithm = new StraightInsertSort<>();
algorithm.sort(points, new PointComparator());
```

Figure 1: How to use sorting API

1.1 Sorting Interface

We create a standard for sorting implementation in this lab. The standard says that any class can have the ability of sorting if it implements interface **ISort**. Interface **ISort** is defined as in Figure 2. Method sort has two parameters:

- The first is array; the element type is parameterized by type **E**.
- The second is an comparator object; programmers can pass any object instantiated from the class that implements interface **java.util.Comparator** for data type **E**.

```java
package sorting;
import java.util.Comparator;
public interface ISort<E> {
    public void sort(E[] array, Comparator<E> comparator);
}
```

Figure 2: Sorting Interface

1.2 Class **StraigtInsertionSort**

Straight Insertion Sort is implemented as in Figure 3.

- Line 3: we declare **StraigtInsertionSort** implementing interface **ISort** and parameterizing element type with parameter **E**.
- Therefore, we have to add an implementation for method sort, from Line 5-19. The code from Line 5 to Line 19 is almost the same as the one given in slide and the demo during lecture, except the element type is changed from **Point2D** to **E**.

1.3 Using **StraigtInsertionSort**

1.3.1 Comparison of data element

In order to use **StraigtInsertionSort**, we can follow steps given in Figure 1.

Assume that we use **StraigtInsertionSort** to sort an array of points of type **Point2D**. Class **Point2D** have been defined in previous Labs.

The question now is **How do we compare two points?**. Clearly, there is no standard way to compare two points in 2D-space. So, how they are compared together is totally depending on you. Actually, we have several approaches, depending on your applications. They are:

- We can compare points by using the x-coordinate of the points.
- We can compare points by using the y-coordinate of the points.
- We can compare points by using the distance to the origin or to their center of mass.

In this lab, we compare by using the x-coordinate. Remember, class **Point2D** supports method getX to obtain the x-coordinate of points. The comparison is coded in class **PointComparator**, defined in Figure 4.

In Figure 4:

```
 1 package sorting;
 2 import java.util.Comparator;
 3 public class StraightInsertionSort<E> implements ISort<E>{
 4     @Override
 5     public void sort(E[] array, Comparator<E> comparator){
 6         int current, walker;
 7         E temp;
 8         current = 1;
 9         while(current < array.length){
10             temp = array[current];
11             walker = current - 1;
12             while((walker >= 0) && comparator.compare(temp, array[walker])
                   < 0 ){
13                 array[walker + 1] = array[walker]; //shift to right
14                 walker -= 1;
15             }
16             array[walker + 1] = temp;
17             current += 1;
18         }
19     }
20 }
```

Figure 3: Class **StraigtInsertionSort**

- Line 3: We declare that **PointComparator** implementing interface **Comparator** to compare two objects of type **Point2D**.
- Line 6: if two values of the x-coordinates are so similar, e.g., the distance is less than $10^{-}7$ then we can consider them are the same (**equals**). As defined by interface **Comparator**, returning 0 means that an equality.

```
 1 package geom;
 2 import java.util.Comparator;
 3 public class PointComparator implements Comparator<Point2D>{
 4     @Override
 5     public int compare(Point2D o1, Point2D o2) {
 6         if(Math.abs(o1.getX() - o2.getX()) < 1e-7) return 0;
 7         else if(o1.getX() < o2.getX()) return -1;
 8         else return 1;
 9     }
10 }
```

Figure 4: Class **PointComparator**

1.3.2 Data generation and sorting

Now, we are ready for testing class **StraightInsertionSort** using steps as shown in Figure 5.

- Line 9: we generate an array of points. Lab1 tell you how to do the generation.
- Line 12-19: we show the list of generated points on screen. Please try format to format your data!
- Line 21-22: **do the sorting** by using **StraightInsertionSort**
- Line 25-31: we show the list after sorted.

We can invoke method demo (in Figure 5) by creating a main entry point as shown in Figure 6. The output is shown in Figure 7.

```java

package sorting;
import geom.Point2D;
import geom.PointComparator;

public class StraightInsertionSortDemo {
    public static void demo() {
        int N = 30;
        Point2D[] points = Point2D.generate(N, -10, 20);

        //Print points
        System.out.println("DEMO FOR INSERTION SORT:");
        System.out.println(new String(new char[80]).replace('\0', '='));
        System.out.println("Unsorted list points:");
        System.out.println(new String(new char[80]).replace('\0', '-'));
        for(int idx=0; idx < N; idx++){
            String line = String.format("%3d | %s", idx, points[idx]);
            System.out.println(line);
        }
        //Sort: insertion sort
        StraightInsertionSort<Point2D> sortAlg = new StraightInsertionSort
            <>();
        sortAlg.sort(points, new PointComparator());

        //Print point
        System.out.println("");
        System.out.println("Sorted list of points (sorted by x-cooridinates
            , ascending)");
        System.out.println(new String(new char[80]).replace('\0', '-'));
        for(int idx=0; idx < N; idx++){
            String line = String.format("%3d | %s", idx, points[idx]);
            System.out.println(line);
        }
    }
}
```

Figure 5: Class **StraigtInsertionSortDemo**

```java
package sorting;
import geom.*;
public class Sorting {
    public static void main(String[] args) {
        StraightInsertionSortDemo.demo();
    }
}
```

Figure 6: Class **StraigtInsertionSortDemo**

1.4 Put them all together

```
DEMO FOR INSERTION SORT:
==============================================================
Unsorted list points:
--------------------------------------------------------------
0 | P(  4.43,   4.95)
1 | P( -4.21,  18.91)
2 | P(  8.26,  18.07)
3 | P(  6.96,  17.94)
4 | P( 11.54,  18.92)
5 | P( -9.83,  17.27)
6 | P( 11.04,  -0.95)
7 | P(  5.93,  -7.67)
8 | P( 19.89,  17.16)
9 | P( -2.46,  -5.54)

Sorted list of points (sorted by x-cooridinates, ascending)
--------------------------------------------------------------
0 | P( -9.83,  17.27)
1 | P( -4.21,  18.91)
2 | P( -2.46,  -5.54)
3 | P(  4.43,   4.95)
4 | P(  5.93,  -7.67)
5 | P(  6.96,  17.94)
6 | P(  8.26,  18.07)
7 | P( 11.04,  -0.95)
8 | P( 11.54,  18.92)
9 | P( 19.89,  17.16)
```

Figure 7: **Output of StraigtInsertionSortDemo**

Question 1

Do the tasks below:

1. Using guideline in Lab1, define class **Point2D** in package **geom**.
2. Define class **PointComparator** as shown in Figure 4.
3. Define interface **ISort** as shown in Figure 2.
4. Define class **StraightInsertionSort** as shown in Figure 3.
5. Define class **StraightInsertionSortDemo** as shown in Figure 5.
6. Create a program with the main entry-point as shown in Figure 6.
7. Compile, Run, and Try to understand the following main points:
   - Generics programming with Java
   - The principle of the design of sorting algorithms
   - Straight Insertion Sort algorithm.
8. Do the analysis for the complexity of Straight Insertion Sort. Write the complexity Straight Insertion Sort in a comment block placed preceding the method sort in Figure3, using the format specified in Lab1.

2 Measuring the time of executation

In this lab, we try to measure the execution time of sorting algorithms. There are several ways to measure time of execution in Java. In this Lab, we do the measurement with API **System.nanoTime()** (in nano seconds), as shown in Figure 8. The difference between two calls to **System.nanoTime()** is the execution time measured in nano-seconds; therefore, we need to divide for a million (1000000) to meaure time in mili-seconds. In Line 22, we divide for $(nExec * 1000000)$ because we perform the sorting $nExec$ times for each of data size.

The question is **Why do we execute $nExec$ times for each data size?** Yes, because the sorting time of some algorithms depends on the order of data elements stored in the input array. Therefore, for each data size (for-loop in Line 14), we try $nExec$ times, each we generate an array of points and then do the sorting.

```
 1  package sorting;
 2  import geom.Point2D;
 3  import geom.PointComparator;
 4
 5  public class SortingEval {
 6      /*
 7      Typical value:
 8      * nElementMax = 500
 9      * nExec = 100;
10      */
11      public static Point2D[] timeit(ISort algorithm, int nElementMax, int
            nExec){
12          Point2D[] algPerfomance = new Point2D[nElementMax];
13          algPerfomance[0] = new Point2D(0,0); //list of size =0 => time = 0
14          for(int idx=1; idx < nElementMax; idx ++){
15              double timeElapsed=0;
16              for(int c=0; c<nExec; c++){
17                  //Try c times for each size
18                  Point2D[] points = Point2D.generate(idx, -20, 20);
19                  long startTime = System.nanoTime();
20                  algorithm.sort(points, new PointComparator());
21                  long endTime = System.nanoTime();
22                  timeElapsed += (double)(endTime - startTime)/(nExec
                        *1000000);
23              }
24              algPerfomance[idx] = new Point2D(idx, timeElapsed);
25          }
26          return algPerfomance;
27      }
28  }
```

Figure 8: Class **SortingEval**

Question 2

Write a program to output the execution time of Straight insertion sort as shown Figure 9.

Guidelines:

- Create an instance of class **StraightInsertionSort** with element type **Point2D**.
- Invoke method timeit in **SortingEval** to get the execution time for each data size. The result from timeit is array of Point2D; the x-coordinate is the data size (casting to double), and the y-coordinate is the execution time. Remember, timeit is static, so you can call it without refering to any object.
- Print the result using **for-lop**. Try **String.format** for formating the data.

```
Straigt Insertion Sort: Time measurement
Size    Time (msec)
----------------------------------------
0       0.00000000
1       0.01073895
2       0.07710172
3       0.00277745
4       0.00804701
5       0.00366332
6       0.00472653
7       0.00567456
8       0.00138266
9       0.00159747
```

Figure 9: The timing output

3 Implement other sorting algorithms

3.1 From programmer's view-point

Assume that we want to allow programmers to sort their data by using API as given in Figure 10. Line 3-6 are for creating sorting objects. Almost API are similar to **StraightInsertionSort**, has been discussed in previous sections, except ShellSort. Why? because in order to use ShellSort we need to pass to the ShellSort algorithm a sequence of integers, each is the count of segments (a concept in Shell-sort algorithm) used in the algorithm. Rememer, **this first is always number 1**, this is the last pass of the sorting process - in the last pass, Shell-sort uses Straight Insertion Sort.

3.2 Implementation

Question 3

Develop classes for other soring algorithms have been learnt. You are required to design your sorting API to support code given in Figure 10 being compiled successfully.

Guidelines: Actually, the instructor has shared with you the source code of all algorithms studied in the last lecture (see announcement in Blackboard). In the shared source code, the sorting algorithms were hardcoded to work with only Point2D object. You can copy the ideas in preivous section for Straight insertion sort to create your own implementation.

```java
int[] num_segments = {1, 3, 7};
ISort[] algorithms = {
    new StraightInsertionSort<Point2D>(),
    new ShellSort<Point2D>(num_segments),
    new StraightSelectionSort<Point2D>(),
    new BubbleSort<Point2D>()
};

for(int aIdx=0; aIdx < alg.length; aIdx++){
    Point2D points = Point2D.generate(100, -20, 20);
    //If you want to sort ...
    algorithms[aIdx].sort(points, new PointComparator()); //do sorting
    //If you want to time it ...
    Point2D[] time = SortingEval.timeit(algorithms[aIdx], 500, 100);
    //here: more code for other purpose.
}
```

Figure 10: Sorting API for some algorithms

4 Extending your implementation

Question 4

The above sorting alorithms **always sort the input data ascendingly**. This question ask you to modify the code developed above to allow programmers to select the sorting direction (ascending or descending).

Guidelines:

- You must think your API supplied to programmers. A alot ways for doing a task like this, you should select an easy and but efficent way.
  - You can allow programmers to specify the sorting direction when they instantiate an sorting object. Thereby, you have to add one more constructor to sorting class. You also need to add one more data field to maintain the sorting direction. Sure, you need to add setter/getter for it too.
- In method sort you need to test the value of the sorting direction (data field) and change the comparison accordingly.

5 Using sorting algorithms

Question 5

The above section help you to sort points in 2D-space using the x-coordinate. This question ask you to support the sorting of points ascendingly/descendingly with respect to their distance to the origin of the space.

Guidelines:

- You need to create a new comparator for points, for example, a class with name **O2PointComparator**; copy the idea from Figure 4.
- In method compare of **O2PointComparator**, you compute the distance from the origin $(0, 0)$ to two points, and then use these distances for comparison. The distance between two points you have studied in high-school; you can ask Dr. Google if forget!

Question 6

The above section help you to sort points in 2D-space using the x-coordinate. This question ask you to support the sorting of points ascendingly/descendingly with respect to their distance to the center of mass of points stored in the array being sorted.

Guidelines:

- You need to create a new comparator for points, for example, a class with name **M2PointComparator**; copy the idea from Figure 4. Different with previous question, you know that the origin is at $(0, 0)$. In this question, the comparator if created as **O2PointComparator**, it **does not** know where the center of mass is.
  - Therefore, you need to add a new constructor to **M2PointComparator**. This new contructor receives a point in 2D-space. Before instantiating an object of **M2PointComparator**, you need to compute the center of mass for points stored in the input array, and then pass the center to the constructor.
  - You also add a data field to maintain the center in **M2PointComparator**; add setter/getter if you want.
  - You use the center stored in the data field (instead of the origin as in previous question) to compute the distance between two points to the center.
  - How to compute the center of mass? **you consult Lab for this question**.
- In method compare of **M2PointComparator**, you compute the distance from the center of mass to two points, and then use these distances for comparison. The distance between two points you have studied in high-school; you can ask Dr. Google if forget!

6 Ploting the time of execution

Question 7

Use the source code of the sample GUI-project shared with you and your modification in Lab2 to create a diagram shown in Figure 11 for plotting the time of execution of sorting algorithms. **Guidelines**:

The code fragment for ploting the time of execution. You should think about where the code should be.

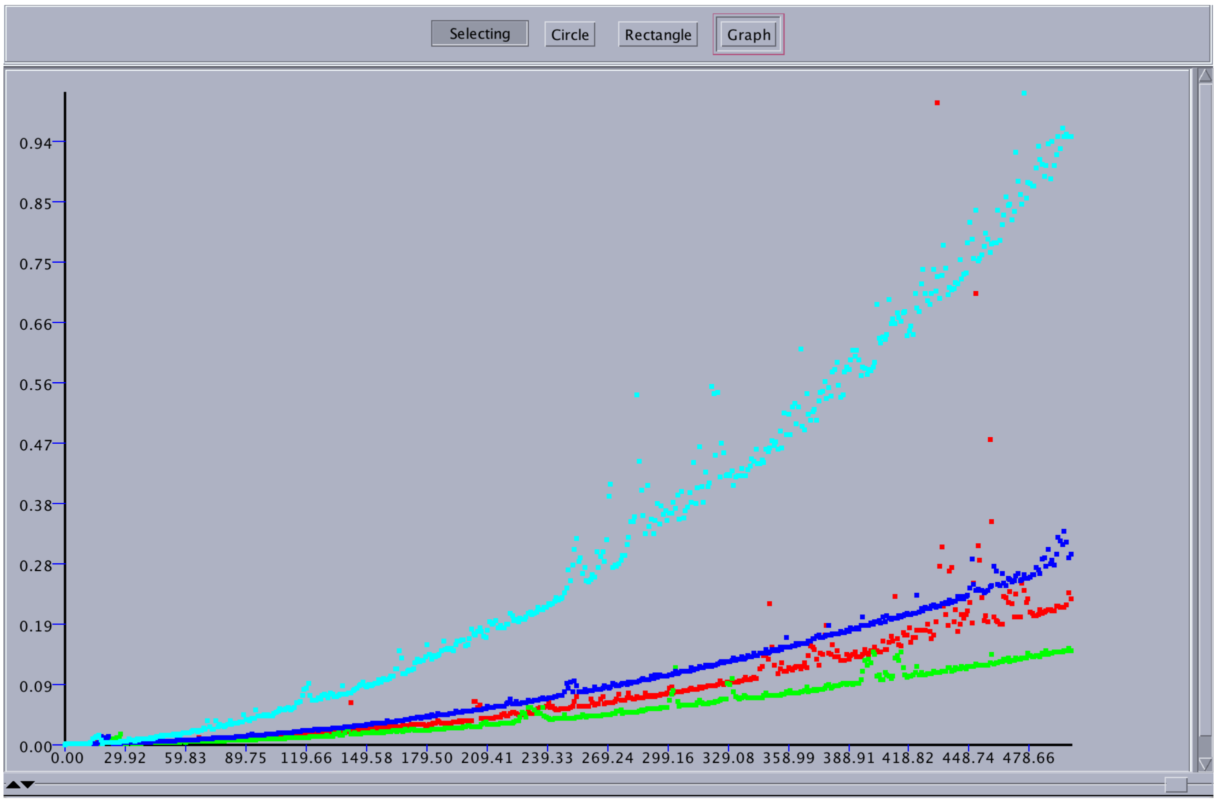

Figure 11: Sorting time: StraightInsertionSort, ShellSort, StraightSelectionSort, BubbleSort

Question 8

As shown in Figure 11, ShellSort is best one, and BubbleSort the worst. **Why?** Write you explanation to the above question to a block comment placed at the begining of each corresponding file.

7 Adding features to your graph

Question 9

The source code of the sample GUI-project shared with you **does not** support features: (a) showing the legend for each graph, and (b) add label for the x- and the y-axis, as you can see in Figure 11.

You are required to add such the features to your project.

```java
int[] num_segments = {1, 3, 7};
ISort[] alg = {
    new StraightInsertionSort<Point2D>(),
    new ShellSort<Point2D>(num_segments),
    new StraightSelectionSort<Point2D>(),
    new BubbleSort<Point2D>()
};

Color[] color = {
    Color.red, Color.green, Color.blue, Color.cyan
};
for(int aIdx=0; aIdx < alg.length; aIdx++){
    Point2D[] time = SortingEval.timeit(alg[aIdx], 500, 500);
    Graph graph = new Graph(time);
    graph.setMode(Graph.GraphMode.SCATTER);
    this.axis.addGraph(graph, color[aIdx], 1);
}

//update viewport
this.spaceMapping.updateLogViewPort(this.axis.getViewport());
this.repaint();
```

Figure 12: Code fragment to generate the drawing as shown in Figure 11

.