Programmierung III (I-320) - WiSe 2020/2021 Exam replacement – research paper

Oleg, Bissing, Matr.-Nr.: 47309

***Abstract*—Topic of this research is „Combsort algorithm with step-by-step visual output and runtime analysis“. Comb sort is one of the huge number of sorting algorithms, which are used in informatics. It is practically an upgrade of the bubble sort algorithm and theoretically it should be much faster. In this article both algorithms are programmed using Java language. For comb sort is also created a visual output. Then comb sort is compared with bubble sort in the same conditions to prove whether it is faster or not. Output is based on thousands of tests combined into a table.**

I. Motivation and Introduction

To start talking about algorithms it is essential to understand what they are. "Informally, an algorithm is any well-defined computional procedure that takes some value, or set of values, as input and produces some value, or set of values, as output [...] We can also view an algorithm as a tool for solving a well-specified computational problem. The statement of the problem specifies in general terms the desired input/output relationship. The algorithm describes a specific computational procedure for achieving that input/output relationship"([1], P. 5).

Algorithms are extremely important in informatics, moreover informatics practically exist because of algorithms, so their importance cannot be underestimated. Among them are sorting algorithms, which are used to sort data in the order needed. In our case it is all about a sorting algorithm called comb sort. It is made as an upgrade of a wide known bubble sort algorithm. The goal here is to provide a reader with a sufficient information about comb sort, to show how does it work using a graphical user interface and to compare it with the bubble sort - otherwise it would be impossible to estimate it. "Algorithms devised to solve the same problem often differ dramatically in their efficiency. These differences can be much more significant than differences due to hardware and software"([1], P. 11).

II. Theory and related works

As already mentioned, there are lots of different sorting algorithms: bubble sort, shell sort, quicksort, merge sort, etc. (at least 26 sorting algorithms can be found on the Internet). Such a diversity grew up together with informatics itself: while bubble sort was analysed as early as 1956[2], some new algorithms came out after 2000. To understand what makes comb sort differ from the other algorithms we need to dive deeper into a theory.

Comb sort is a relatively simple sorting algorithm originally designed by Włodzimierz Dobosiewicz and Artur Borowy in 1980[3], later rediscovered (and given the name "Combsort") by Stephen Lacey and Richard Box in 1991[4]. It is practically an upgrade of a bubble sort algorithm.

In the "bubble", when iterating over the array, adjacent elements are compared. The main idea of the "comb" is to initially take a sufficiently large distance between the compared elements and, as the array is ordered, narrow this distance down to the minimum. Thus, we sort of comb the array, gradually smoothing it into more and more neat strands. It is better to take the initial gap between the compared elements considering a special value called the reduction factor, the optimal value of which is approximately 1.3. This value came out as a result of empirical tests of comb sort on over 200.000 random lists([4], P. 316).

First, the distance between the elements is maximum, that is, equal to the size of the array minus one. Then, after going through the array with this step, you need to divide the step by the decrease factor and go through the list again. This continues until the difference between the indices reaches one. In this case, neighbouring elements are compared, as in bubble sort, but there is only one such iteration.

Stephen Lacey and Richard Box in their article "A Fast, Easy Sort: A novel enhancement makes a bubble sort into one of the fastest sorting routines" also explain the hidden hint, that makes comb much faster than bubble sort. "Bubble sorts are also slow because they are susceptible to the birth of elements, we call turtles. A turtle (in an ascending sort) is a relatively low value located near the end of a list. During a bubble sort, this element moves up only one position for each pass (or stroke), so a single turtle can cause maximal slowing. Almost every long random list contains a turtle. One the other hand, a high-value element near the top of a list (a rabbit) is harmless. If you reverse the direction of the stroke, turtles become rabbits and rabbits become turtles. The worst possible turtle – the lowest relative value at the end of a list – forces a bubble sort to make $(n-1)^2$ comparisons. This means that a bubble sort of a 1000-item list could require nearly a million comparisons. Combsort is our simple modification of a bubble sort. It eliminates turtles by allowing the distance between compared elements to be greater than 1" ([4], P. 315).

It also makes sense to mention, that despite of a first-glance similarity, comb sort is not the same as another sorting algorithm – shell sort. “A shell sort does a complete sort (until there are no more swaps to be made) for each gap size. Comb sort makes only a single pass for each gap size [...]. The ideal shrink factor for a shell sort is around 1.7, compared with 1.3 for comb sort” ([4], P. 320). There are some other differences from a mathematical point of view as well.

III. Comb sort algorithm with step-by-step visual output and runtime analysis

A. From theory to code

One of the most important steps in the beginning is to think over the way to put theoretical ideas into the lines of the code.

1) Approach: Thinking about coding in the context of this topic it is obvious, that there are several things to consider about. First, it is of course needed to implement the algorithm. While it is relatively simple there will not be many ways to code it. Second, there are actually not one but two tasks to make with this implemented algorithm, but visual output would slower the runtime, which means that it optimal to make those two parts separately.

2) Two tasks: Because the ideas behind those two tasks are not directly connected with each other it is logically correct to realise them apart. Both parts would share same class with the algorithm. In the visual part of code, I am not going to concentrate on efficiency and runtimes of the code, it should only deliver a correct result with an understandable graphical output. The other part, on the contrary, will output all the results in the console and main goal would be to analyse runtimes.

B. Coding

As I mentioned before I decided to separate visualisation and calculation parts in two programs. It is also important to implement the algorithm first.

1) Algorithm: Using the theoretical information about comb sort it is now time to code it. A while-loop is needed to run iterations untill the gap's size is equal to one while shrinking it using a separate method, for-loop is needed to go through the array, an if-statement – to check which of two numbers in the array is bigger and swap them when needed. Implemented in Java comb sort algorithm looks wie follow:

```
public int[] sortIntegers() {
   int save;
   while(gap > 1 || flag) {
      gap = calculateGap(gap);
      flag = false;
      for(int i = 0; i < array.length - gap; i++) {
         if(array[i] > array[i + gap]) {
            save = array[i];
            array[i] = array[i + gap];
            array[i + gap] = save;
            flag = true;
         }
      }
   }
   return sortedArray;
}

private int calculateGap(int gap) {
   gap = (gap * 10) / 13;
   return Math.max(gap, 1);
}
```

2) Visual output: For a propriate visual output are used .awt and .swing libraries. CombSortVisualization class serves for creating MyFrame class only. MyFrame class extends JFrame class and creates MyPanel class. It is also needed for some basic settings like closing operator, visibility, and location of the window. Those are written in the constructor:

```
MyPanel panel;

public MyFrame() {
   panel = new MyPanel();
   this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   this.add(panel);
   pack();
   this.setLocationRelativeTo(null);
   setVisible(true);
}
```

MyPanel class extends JPanel class and does a lot. It creates the next class (MyGraph), a panel and a button. It sets all the settings needed such as background, size, positioning of the elements, borders, visibility, etc. It also implements ActionListener which is needed to interact with the user using 'Restart' button to restart the visualisation. It works wie follow: when user clicks the button an existing JPanel is deleted and new one is created at the same place.

```
@Override
public void actionPerformed(ActionEvent e) {
   if(e.getSource() == button) {
      this.remove(graph);
      graph = new MyGraph();
      this.add(graph);
      SwingUtilities.updateComponentTreeUI(this);
      }
}
```

MyGraph class is the most important one. It extends JPanel class, creates graphics and calculates comb sort algorithm. The idea behind it is the following one: to create an array of 1000 random numbers, use comb sort algorithm to sort those numbers in ascending order while saving each step inside an empty array with arrays previously created and then use this array to paint step by step the sorting using a timer.

```
public void paint(Graphics g) {
   super.paint(g);
   g.setColor(new Color(184, 227, 223));
   for(int i = 0; i < 1000; i++) {
      g.drawLine(i, 500, i, 500-array[i]);
   }
}
```

Code above only paints one picture with the lines. To bring it to action there is once more needed the ActionListener, but now the action would be performed not by the user, but by the timer. With the last repaint backgroud color is changed to underline the end of the sorting animation (see Figure 1).

```
@Override
public void actionPerformed(ActionEvent e) {
   System.arraycopy(vault[count], 0, array, 0, vault[
        count].length);
   count++;
   if(vault[count][w-1] == 0) {
      timer.stop();
      setBackground(new Color(255, 253, 208));
   }
   repaint();
}
```

3) Runtime analysis: For runtime analysis I have created another program to keep the task logically fitting the concept. CombSortTestingBuild class calls the Menu class which is needed to interact with user by means of the console. Menu class uses switch statement with endless while-statement (ends only when 0-button is pushed) and calls other classes depend on the user's choice.

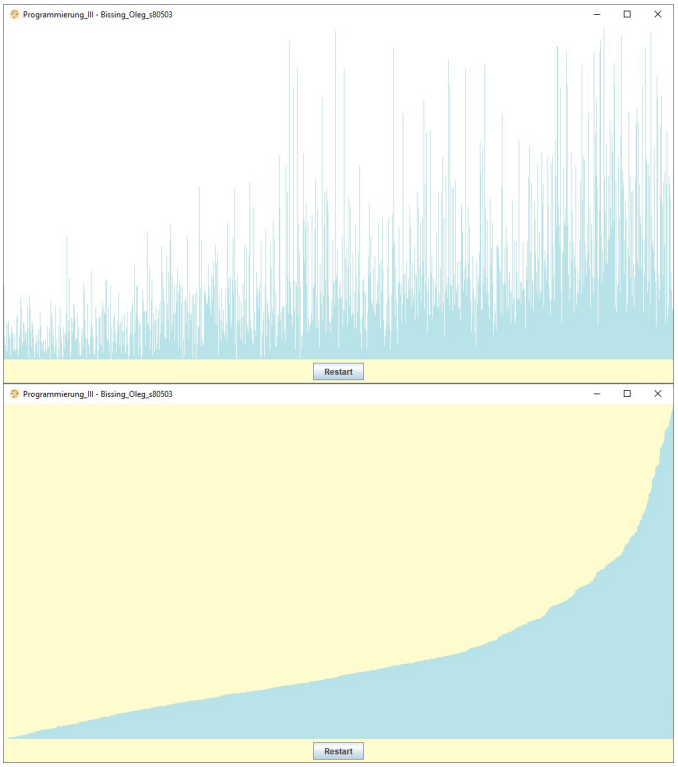

Figure 1. Example of visualising

Output class prints information in the console. It uses different formatting options to create an output needed. For example, chooses a matching format for each element of the array while printing.

```
for(int i = 0; i < array.length; i++) {
   int n = i + 1;
   if(array.length < 10) {
      System.out.println("Element " + (i + 1) + "
          contents " + array[i]);
   } else if(array.length < 100) {
      System.out.printf("Element " + "%2d" + " contents
          " + array[i] + "\n", n);
   } else if(array.length < 1000) {
      System.out.printf("Element " + "%3d" + " contents
          " + array[i] + "\n", n);
   } else if(array.length < 10000) {
      System.out.printf("Element " + "%4d" + " contents
          " + array[i] + "\n", n);
   } else {
      System.out.printf("Element " + "%5d" + " contents
          " + array[i] + "\n", n);
   }
}
```

Runtime class is needed to calculate runtime. To do so, start time and end time of the calculation needs to be saved and then start time must be subtracted from the end time using casting to get double out of long.

```
startTime = System.nanoTime();
result = bubbleSort.sortIntegers();
endTime = System.nanoTime();
duration = (double) (endTime - startTime) / 1000000;  //
    divide by 1000000 to get milliseconds
```

GenerateArray class can either create a random array or ask the user to input the numbers. It uses Scanner class to get data from the user, then creates an array. It also uses Random class to create random numbers for the array. To make a created number not to huge and to randomize it more, there are actually two numbers created and one is divided by the other. To make sure, that the dividing number is not equal to zero (to not to divide by zero) one is added "by default". For example, to create an array with the given by user length the following code is used.

```
public int[] createIntegers() {
   Scanner scanner = new Scanner(System.in);
   System.out.println("How many numbers should contain
       your array: ");
   quantity = scanner.nextInt();
   Random rand = new Random();
   array = new int[quantity];
   for(int i = 0; i < array.length; i++) {
      array[i] = rand.nextInt(9999) / (rand.nextInt(99)
          + 1);
   }
   return array;
}
```

BubbleSort and CombSort classes are used to respectively sort an array. Comb sort algorithm is given in the part 3.2.1. Implementation of the bubble sort algorithm should also be mentioned.

When we compare these two algorithms it can be seen that they are similar, so it is interesting to see, how much of a difference this addition of the gap within the comb sort algorithm makes.

```
public int[] sortIntegers() {
   while (flag) {
      flag = false;
      for(int i = 0; i < sortedArray.length - 1; i++) {
         if(sortedArray[i] > sortedArray[i + 1]) {
            int save = sortedArray[i];
            sortedArray[i] = sortedArray[i + 1];
            sortedArray[i + 1] = save;
            flag = true;
         }
      }
   }
   return sortedArray;
}
```

AutoTest class is the most important one. To analyse information or statistics you need to have these. In this case it is necessary to conduct a sufficient amount of runs to get enough data to analyse. At this point AutoTest class is quite useful: it not only conducts a given number of tests with a given amount of numbers but also calculates the fastest, longest and average runtime.

In this class are implemented some small tricks to make the code look better, shorter and work faster. For example, while it is needed to calculate minimal and maximal values of test runtime those variables are given in constructor with a predefined value: maximal possible double value for a minimal value variable and minimal possible double value for a maximal value variable. This approach guarantees, that on the one hand this variable can from the very beginning be used in logical operations (for example with if-operator) while it is not empty, one the other hand any possible minimal or maximal value of the calculations will be respectively lower or bigger than this placeholding value.

```
public AutoTest() {
   this.count = 0;
   this.durationSum = 0.0;
   this.minValue = Double.MAX_VALUE;
   this.maxValue = Double.MIN_VALUE;
}
```

	Bubble sort	Comb sort
1000 runs	average/longest/fastest, ms	average/longest/fastest, ms
10 numbers	0.0008 / 0.0123 / 2.0^{-4}	0.00098 / 0.0098 / 2.0^{-4}
100 numbers	0.0170 / 0.0813 / 0.0122	0.0025 / 0.0036 / 0.0019
1000 numbers	0.82683 / 0.9526 / 0.7814	0.03386 / 0.0553 / 0.0318
10000 numbers	94.4668 / 100.52 / 90.834	0.39589 / 0.4822 / 0.371

Table I
COMPARISON OF THE BUBBLE SORT WITH THE COMB SORT

Amount of test runs is given by the user and then a for-loop is used to conduct these.

```
for(int i = 0; i < testsCount; i++) {
   int[] array = generateArray.createIntegers(quantity);
   duration = run.runTimeBubbleSort(array, false);
   stats[i] = duration;
   if(duration < minValue) {
      minValue = duration;
   } else if (duration > maxValue) {
      maxValue = duration;
   }
}
```

To find out an average value an array of double values is created. Then all the values this array consist are added together and within the output are divided by their count.

```
double[] stats = new double[testsCount];
...
for(int i = 0; i < stats.length; i++) {
   durationSum += stats[i];
   count++;
}
```

Another small trick is to use formatted console output to control its length, while by double value it can be much longer than needed for this purpose.

```
System.out.printf("Average runtime is %.5f", (
     durationSum / count));
```

When all the iterations done user sees the following output in the console.

```
BubbleSort: 1000 runs are  done.
Average runtime is 0,82496 ms.
Longest runtime is 0.9138 ms.
Fastest runtime is 0.783 ms.

CombSort: 1000 runs are  done.
Average runtime is 0,03515 ms.
Longest runtime is 0.1045 ms.
Fastest runtime is 0.0309 ms.
```

As you can see, AutoTest is used in a bubble sort as well. As we remember from the part 2, comb sort is an upgrade of the bubble sort. This means that comb sort should be faster and to analyse this idea properly we need to compare them in the *same* conditions.

For the analysis had been chosen the following conditions: 1000 runs for each numbers count, both sorting algorithms always sorting a randomly generated array of the same length, test starts with 10 numbers per array and this count increases tenfold each new autotest.

As you can see (Table I) there is almost no difference between two algorithms dealing with only 10 numbers long arrays. But when we increase it to even 100 numbers per array comb sort starts to show efficiency. With relatively huge arrays (10000 numbers) the difference is tremendous.

IV. SUMMARY AND OUTLOOK

When creating a program, it is important to choose carefully all the instruments needed. One of these instruments are algorithms. To make a choice which algorithm to use a programmer must see the difference clearly. In this research in details is inspected comb sort algorithm: were covered both theoretical and practical aspects.

First, theoretical background was introduced, and related works were mentioned, among them an interesting article in the Byte Magazine "A Fast, Easy Sort: A novel enhancement makes a bubble sort into one of the fastest sorting routines" where its authors Stephen Lacey and Richard Box gave a comprehensive look on the comb sort and that it is a better version of the bubble sort. In this research were mentioned their arguments why comb sort is better, and some mathematical justification was given.

Second, coding approach was explained and supported by numerous code examples. The whole coding task was separated in two parts to show both visual and statistical tasks at their best. For the visualisation Java .awt and .swing libraries were used as well as the ActionListener class. There are also added screenshots of a launched program to show the reader the visual output.

To analyse the runtime another program has been written. It uses different Java built-in classes (ActionListener, Timer, Scanner, etc.) and the power of object-oriented programming to apply needed functionality. It also offers an opportunity to conduct different auto tests to get required data. To do so user needs to give the number of runs and the length of arrays. In this research different variations of these two parameters were used to get enough data to analyse. An example of the console output is added as well.

More than a dozen small code passages with comments and explanations are added to clarify every step of this research. They cover not only algorithm itself, but all the important parts of both programs to show the reader step-by-step the code part of the research.

As comb sort is an upgrade of bubble sort it was also compared with it, results are presented in the Table 1. Head-to-head comparison proved it to be significantly faster than bubble sort in the most cases, and which is also important, in all three parameters compared: average, longest, and fastest sort.

In conclusion it should be said that choice of the algorithm is based on the purpose: if it is needed to sort small groups of numbers – perhaps bubble sort is a better choice because it is shorter and easier, but it is obvious, that for huge arrays, especially bigger than 1000 numbers it is incomparably better to use the comb sort.

REFERENCES

[1] Cormen T.H., Leiserson C.E., Rivest R.L., Stein C.: „*Introduction To Algorithms*“, The MIT Press, 2001

[2] Demuth, Howard B.: „*Electronic Data Sorting*“, PhD thesis, Stanford University, 1956

[3] Dobosiewicz W.: „*An efficient variation of bubble sort*“, Information Processing Letters 11, 1980

[4] Lacey S., Box R.: „*A Fast, Easy Sort: A novel enchancemet makes a bubble sort into one of the fastest sorting routines*“, Byte Magazine Vol. 16, no. 4., 1991