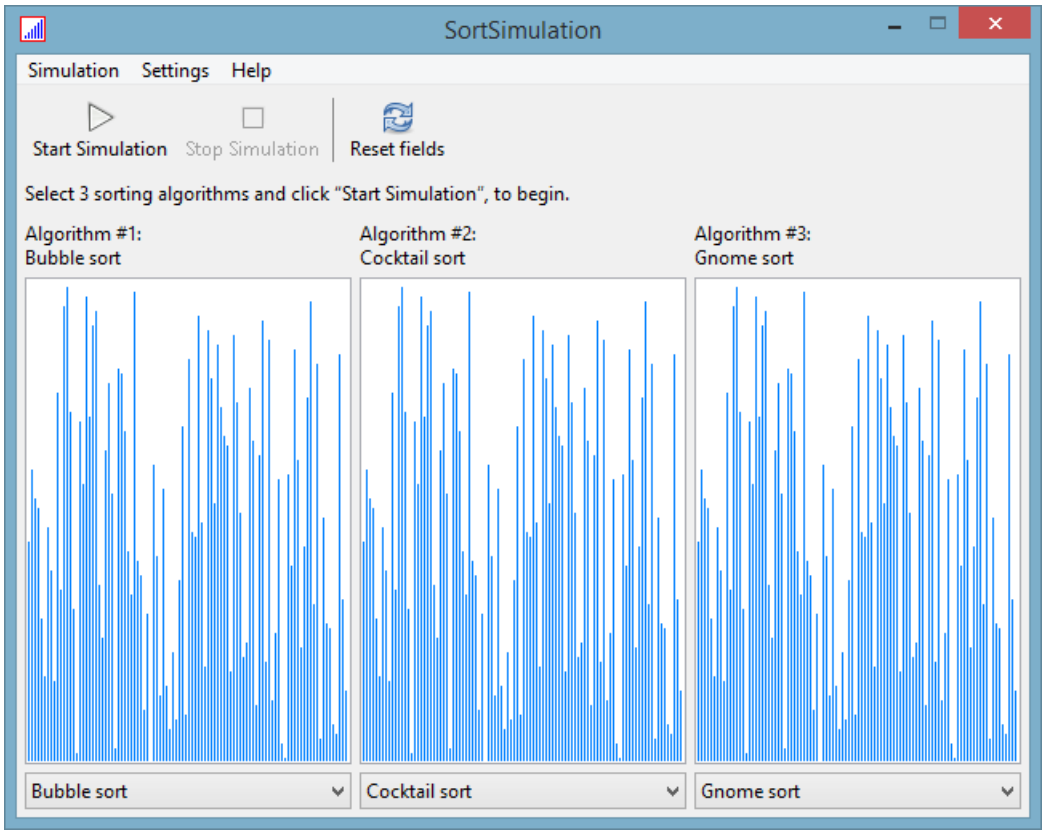
SortSimulation

Documentation

– English –

Version 2.0.0

http://www.peterfolta.net/software/sortsimulation



Copyright © 2008–2014 Peter Folta. All rights reserved.

PETER FOLTA

Contents

1 Introduction

SortSimulation is a Java application that provides the user with a visual representation of various sorting algorithms. This gives users a better understanding of the various operating modes of said algorithms. The application is also useful for demonstrating the runtime differences without simply comparing numbers.

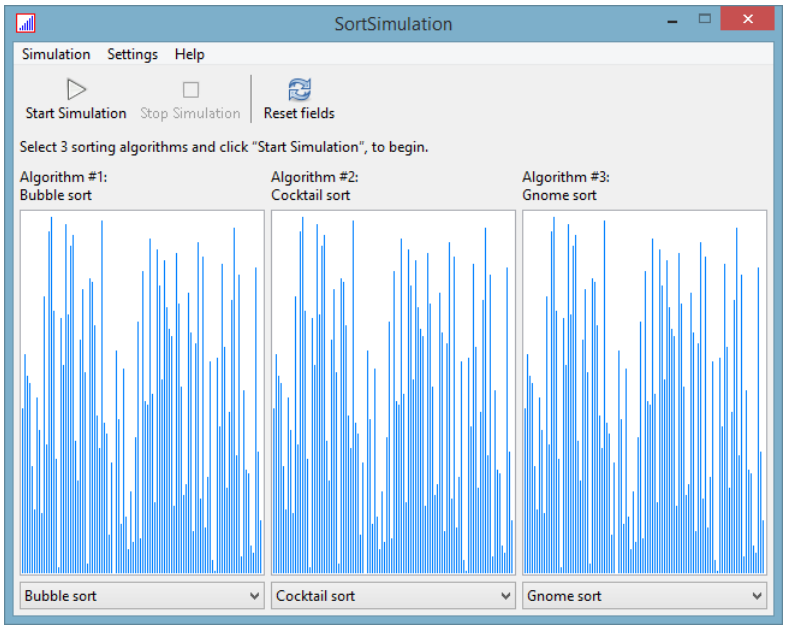


Figure 1: *The main window of SortSimulation*

This documentation contains a brief insight into the usage of SortSimulation and presents the supported sorting algorithms, including their java implementation.

2 Usage

Using SortSimulation is really simple: the sorting fields are filled randomly immediately after the launch of the program—every field containing the same initial situation. Three sorting algorithms are preselected also following the program launch, which the user can then change to suit their needs.

To compare the three particular sorting algorithms of your choice, select them from the combo boxes below the fields. Click on "*Start simulation*" in the toolbar, the menu, or, alternatively, press the enter key on your keyboard to begin. Cancelling an active simulation is just as easy: Press the escape key or click on "*Stop simulation*". To reset the fields after a finished or cancelled simulation, click on the button "*Reset fields*" or press `Ctrl+N`.

SortSimulation allows you to configure the simulation as you wish. The available settings are discussed in the following section.

In addition, advanced users have the possibility to provide SortSimulation with arguments upon program launch, read more about this in section 2.2 on page 3.

2.1 Settings

SortSimulation offers a variety of options to configure the sorting simulations in the way the user wants it. The available settings are listed in the "*Settings*" menu:

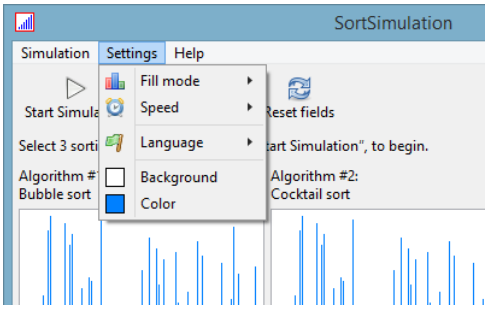


Figure 2: *The settings menu*

"*Fill mode*" offers the user four possibilities to configure the way the sorting fields are filled to begin with: "*Randomly*" (preselected; `Ctrl+R`), "*Inverse*" (`Ctrl+I`), "*Almost sorted*" (`Ctrl+A`) and "*Presorted*" (`Ctrl+P`). If "*Randomly*" is selected, the elements are ordered completely randomly in the fields when the user commits the action "*Reset fields*". This option allows the user to try the different sorting algorithms on data that is not ordered whatsoever, in a very realistic fashion.

In contrast, the "*Inverse*" mode begins with the fields already sorted- in a descending, as opposed to ascending, order. This setting gives the user a better insight to look at the efficiency of the algorithms sorting backwards, pre-ordered series.

The "*Almost sorted*" mode generates a randomly sorted sequence that only differs slightly from its real order. The rough structure of the sorted data is already recognizable. This option helps the user to visualize which sorting algorithms are particularly useful for sorting data sets that only marginally differ from their sorted order.

The last option, "*Presorted*", fills all fields with already sorted data. This might be useful for understanding how different sorting algorithms work on an already sorted set of data. However, this only has an effect on some algorithms.

The submenu "*Speed*" allows the user to configure the speed of the simulation. The user can choose their desired speed from five prechosen levels. As algorithms like Bubble sort are quite slow, it may be worth selecting one of the higher speed levels. However, whilst running a sorting algorithm like Quicksort, it is better to choose a slower speed to be able to take a closer look at the functionality. It's also possible to set the different speed levels by keyboard using the sortcuts `Ctrl+Shift+(1-5)`.

SortSimulation does not create or modify any of the files stored on the user's computer. That is why SortSimulation's user interface language is always set to English after the program launch, even

if the language was previously changed. To switch between the offered languages, go to the submenu “*Language*”.

The last two submenus “*Background*” and “*Color*” allow the user to change the color of the bars as well as to adjust the background color of the panes, so that the program is aesthetically pleasing to the user as well as being easily readable.

2.2 Parameter

By default, SortSimulation displays three different fields alongside each other. This layout makes it easier for the user to compare the various algorithms by viewing them at the same time. It is possible to adjust the number of displayed fields by using a parameter at the program launch. To do this, simply provide the desired amount, whilst keeping in mind that, valid values are between 2 and 9.

For instance, to launch SortSimulation with 5 fields, please run the following command from a terminal window:

```
1 SortSimulation 5
```

SortSimulation will continue to start as usual, but shows the provided 5 fields this time instead of the default 3:

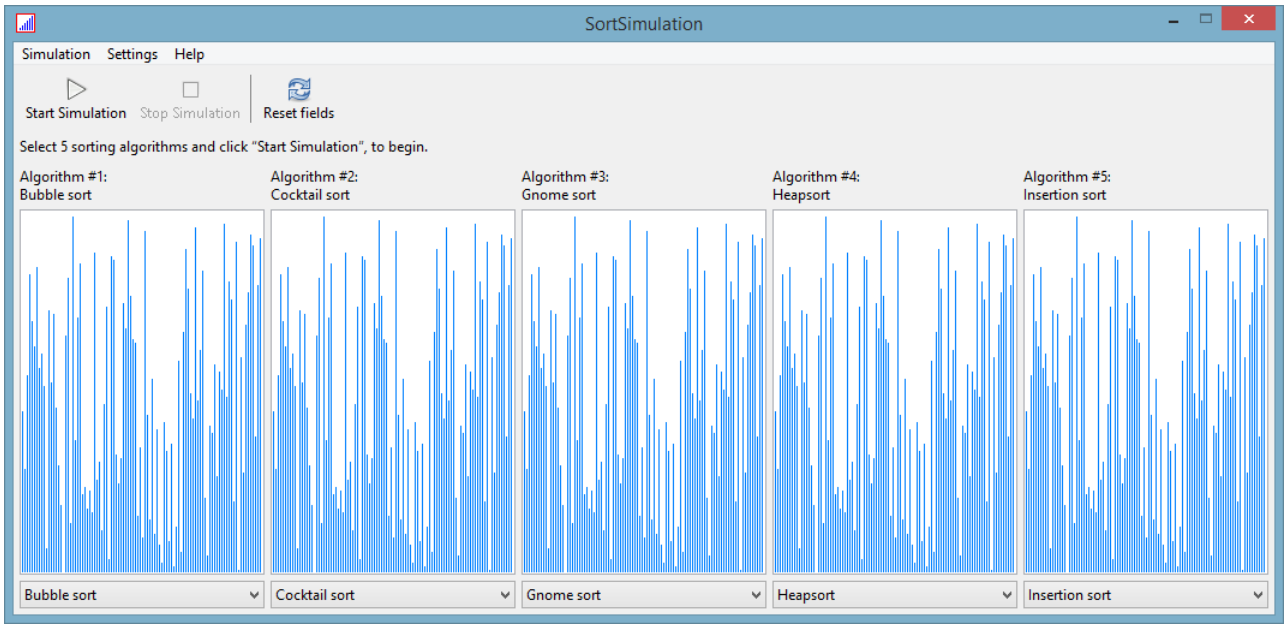


Figure 3: *SortSimulation with 5 fields*

Please note that when using this method, SortSimulation’s main window may become much wider than usual which may be larger than what the user’s screen is able to display.

If the users provides an invalid value, the following error message will occur:

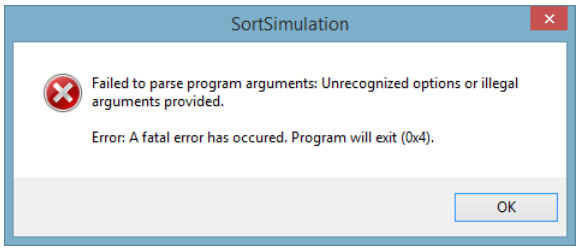


Figure 4: *Error message when using illegal parameters*

3 Sorting algorithms

3.1 Bubble sort

Bubble sort is a very simple sorting algorithm that sorts elements by using a *step-by-step comparison*. As bubble sort is not very efficient, it is often used to demonstrate a weak sorting algorithm.

3.1.1 Idea of Bubble sort

Bubble sort works by comparing two of the listed items and swapping them if they are in the wrong order. This procedure is repeated until all elements are in the correct position.

3.1.2 Performance

Bubble sort has typical complexity $\mathcal{O}(n^2)$, where n is the number of items being sorted. The performance of bubble sort over an already-sorted list (best-case) is $\mathcal{O}(n)$.

3.1.3 Java implementation

```
1  public class Bubblesort {
2      public void sort(int[] a) {
3          int tmp;
4
5          for(int i = a.length-1; i >= 0; i--) {
6              for(int j = 0; j <= i-1; j++) {
7                  if(a[j] > a[j+1]) {
8                      tmp = a[j];
9                      a[j] = a[j+1];
10                     a[j+1] = tmp;
11                 }
12             }
13         }
14     }
15 }
```

Listing 1: *Implementation of Bubble sort*

3.2 Cocktail sort

Cocktail sort, also known as *Shaker sort*, is another simple sorting algorithm, which is a variant of the popular bubble sort.

3.2.1 Idea of Cocktail sort

The sequence of elements that are to be sorted are iterated upwards and downwards alternately. In each step, two neighboring elements are compared and swapped if they are in the wrong order. Because of this, a cluster of large elements forms on the upper end of the array and a cluster of small elements form on the lower end.

3.2.2 Performance

Cocktail sort has typical complexity $\mathcal{O}(n^2)$, where n is the number of items being sorted. It strives towards $\mathcal{O}(n)$ for almost sorted sequences.

3.2.3 Java implementation

```
1  public class CocktailSort {
2      public void sort(int[] a) {
3          int start = -1;
4          int end = a.length - 2;
5          int tmp;
6
7          boolean swapped;
8
9          do {
10             swapped = false;
11             start++;
12
13             for(int i = start; i <= end; i++) {
14                 if(a[i] > a[i+1]) {
15                     tmp = a[i];
16                     a[i] = a[i+1];
17                     a[i+1] = tmp;
18                     swapped = true;
19                 }
20             }
21
22             if(!swapped) {
23                 break;
24             }
25
26             swapped = false;
```

```
27              end--;
28
29              for(int i = end; i >= start; i--) {
30                  if(a[i] > a[i+1]) {
31                      tmp = a[i];
32                      a[i] = a[i+1];
33                      a[i+1] = tmp;
34                      swapped = true;
35                  }
36              }
37          } while(swapped);
38      }
39  }
```

Listing 2: *Implementation of Cocktail sort*

3.3 Gnome sort

Gnome sort is yet another very simple sorting algorithm, that sorts elements by *step-by-step comparison*. It was originally proposed by *Hamid Sarbazi-Azad* in 2000 under the name *Stupid Sort*. Later the name was changed into Gnome sort. Gnome sort has the advantage of requiring just one loop (and thus no nested loops).

3.3.1 Idea of Gnome sort

Gnome sort works in a way one might imagine a garden gnome: Suppose a garden gnome wants to sort flower pots. He initially stands at the left end of the sequence of pots. The gnome compares two neighboring flower pots: If they are in the correct order, he moves one pot to the right. If they are not, he swaps them and makes one step to the left, unless he is standing at the left end of the pot sequence. In that case, the garden gnome makes one step to the right. This procedure is repeated until he gets to the right end of the sequence (and therefore reaches the last flower pot).

3.3.2 Performance

Gnome sort has typical complexity $\mathcal{O}(n^2)$, where n is the number of items being sorted. It strives towards $\mathcal{O}(n)$ for almost sorted sequences.

3.3.3 Java implementation

```
1  public class Gnomesort {
2      public void sort(int[] a) {
3          int tmp;
4          int pos = 1;
5
6          while(pos < a.length) {
```

```
7               if(a[pos] >= a[pos-1]) {
8                   pos++;
9               } else {
10                  tmp = a[pos];
11                  a[pos] = a[pos-1];
12                  a[pos-1] = tmp;
13
14                  if(pos > 1) {
15                      pos--;
16                  }
17              }
18          }
19      }
20  }
```

Listing 3: *Implementation of Gnome sort*

3.4 Heapsort

Heapsort is a fast sorting algorithm which was developed by *Robert W. Floyd* and *J. W. J. Williams* in 1964. Heapsort is an improvement of *Selection sort*.

3.4.1 Idea of Heapsort

Heapsort uses the *heap* as a specialized data structure for sorting. This data structure is based on an (almost) complete *binary tree*. A binary tree is (almost) complete, if all levels except possibly the last are completed.

If the sequence exists as a heap, the largest element can be taken out of the *root* of the tree. To get to the next item, the heap has to be *rearranged* before.

3.4.2 Performance

Heapsort always has complexity $\mathcal{O}(n \log n)$ and is therefore one of the best comparison sorts.

3.4.3 Java implementation

```
1   public class Heapsort {
2       private void sift(int[] a, int l, int r) {
3           int i;
4           int j;
5           int x;
6
7           i = l;
8           x = a[l];
9           j = 2 * i + 1;
10
```

```
11          if((j < r) && (a[j+1] > a[j])) j++;
12
13          while((j <= r) && (a[j] > x)) {
14              a[i] = a[j];
15              i = j;
16              j = 2 * j + 1;
17
18              if((j < r) && (a[j+1] > a[j])) j++;
19          }
20
21          a[i] = x;
22      }
23
24      public void sort(int[] a) {
25          int l;
26          int r;
27          int tmp;
28
29          for(l = (a.length - 2) / 2; l >= 0; l--) {
30              sift(a, l, a.length-1);
31          }
32
33          for(r = a.length - 1; r > 0; r--) {
34              tmp = a[0];
35              a[0] = a[r];
36              a[r] = tmp;
37              sift(a, 0, r-1);
38          }
39      }
40  }
```

Listing 4: *Implementation of Heapsort*

3.5 Insertion sort

Insertion sort is quite a simple sorting algorithm. It is not as efficient as other more complex algorithms, but it is *easy to implement* and has a very short run time in case of a small amount of data or already presorted data.

3.5.1 Idea of Insertion sort

Insertion sort removes an element of the unsorted set and inserts it in the correct place of the output sequence. If the sequence is empty, the element will be inserted at the first position.

This sorting algorithm is rather inefficient, because it often needs to move elements over long distances.

3.5.2 Performance

Insertion sort has typical complexity $\mathcal{O}(n^2)$, where n is the number of items being sorted. It strives towards $\mathcal{O}(n)$ for almost sorted sequences.

3.5.3 Java implementation

```
1  public class Insertionsort {
2      public void sort(int[] a) {
3          int tmp;
4          int j;
5
6          for(int i = 1; i < a.length; i++) {
7              tmp = a[i];
8              j = i;
9
10             while(j > 0 && a[j-1] > tmp) {
11                 a[j] = a[j-1];
12                 j--;
13             }
14
15             a[j] = tmp;
16         }
17     }
18 }
```

Listing 5: *Implementation of Insertion sort*

3.6 Merge sort

Merge sort is a recursive and stable sorting algorithm, which is based on the *divide and conquer principle*, similar to *Quicksort*. Merge sort was presented by John von Neumann in 1945.

3.6.1 Idea of Merge sort

Merge sort splits the whole sequence into several smaller sequences, which will each be sorted individually. Following this, the small sequences are united in one completely sorted sequence.

3.6.2 Performance

Merge sort always has complexity $\mathcal{O}(n \log n)$ and is therefore, arguably, one of the best comparison sorts.

3.6.3 Java implementation

```java
public class Mergesort {
    private int[] a;
    private int[] b;
    private int n;

    public void sort(int[] a) {
        this.a = a;
        n = a.length;
        b = new int[n];
        mergesort(0, n-1);
    }

    private void mergesort(int lo, int hi) {
        if(lo < hi) {
            int m = (lo + hi) / 2;
            mergesort(lo, m);
            mergesort(m+1, hi);
            merge(lo, m, hi);
        }
    }

    private void merge(int lo, int m, int hi) {
        int i = lo;
        int j = hi;
        int k = lo;

        while(i <= m) b[k++] = a[i++];
        while(j > m) b[k++] = a[j--];

        i = lo;
        j = hi;
        k = lo;

        while(i <= j) {
            if(b[i] <= b[j]) a[k++] = b[i++];
            else a[k++] = b[j--];
        }
    }
}
```

Listing 6: *Implementation of Merge sort*

3.7 Quicksort

Quicksort is one of the *fastest sorting algorithms*, based on the *divide and conquer principle*. The recursive Quicksort algorithm, in its original version, was developed by *C. Antony R. Hoare* in 1960.

3.7.1 Idea of Quicksort

The sequence will initially be split into two parts. The first section's elements are all less than or equal to all elements in the second section (*divide*). Then, the divided sequences are independently, recursively sorted using the same procedure (*conquer*). The last step combines the smaller sequences to a sorted one (*combine*).

The division is realized using a *pivot element*, which is selected from the array in the first step. All elements of the sequence, which are *smaller* than the pivot element, will be put in the first section. All the elements, that are *larger* than the pivot element, will be put in the second section. For elements which have the same value as the pivot element, it does not matter in what part they are put.

3.7.2 Performance

Quicksort has typical complexity $\mathcal{O}(n \log n)$, where n is the number of items being sorted. It is therefore, arguably, one of the best comparison sorts. The worst case performance of Quicksort has complexity $\mathcal{O}(n^2)$.

3.7.3 Java implementation

```
1  public class Quicksort {
2      private void quicksort(int[] a, int bottom, int top) {
3          int tmp;
4          int i = bottom;
5          int j = top;
6          int middle = (bottom + top) / 2;
7          int x = a[middle];
8
9          do {
10             while(a[i] < x) i++;
11             while(a[j] > x) j--;
12
13             if(i <= j) {
14                 tmp = a[i];
15                 a[i] = a[j];
16                 a[j] = tmp;
17                 i++;
18                 j--;
19             }
20         } while(i <= j);
21
22         if(bottom < j) quicksort(a, bottom, j);
23         if(i < top) quicksort(a, i, top);
24     }
25
26     public void sort(int[] a) {
```

```
27          quicksort(a, 0, a.length-1);
28      }
29  }
```

Listing 7: *Implementation of Quicksort*

3.8 Selection sort

Selection sort is a naive sorting algorithm that works in place. It is compareable to insertion sort.

3.8.1 Idea of Selection sort

Selection sort divides the sequence in a *sorted* field and an *unsorted* field. The sorted part is empty at the beginning. Selection sort examines the smallest element in the unsorted sequence and swaps it with the first element. The sequence is sorted until this position after the step is made. The procedure is repeated until the entire sequence is sorted.

3.8.2 Performance

Selection sort always has complexity $\mathcal{O}(n^2)$.

3.8.3 Java implementation

```
1  public class Selectionsort {
2      public void sort(int[] a) {
3          int tmp;
4
5          for(int i = 0; i < a.length; i++) {
6              for(int j = i+1; j < a.length; j++) {
7                  if(a[j] < a[i]) {
8                      tmp = a[i];
9                      a[i] = a[j];
10                     a[j] = tmp;
11                 }
12             }
13         }
14     }
15 }
```

Listing 8: *Implementation of Selection sort*

3.9 Shell sort

The sorting algorithm Shell sort is based on *Insertion sort*. It was invented in 1959 by *Donald L. Shell*.

3.9.1 Idea of Shellsort

Shellsort compensates for the disadvantage of Insertion sort to move elements over large distances. Shellsort creates a *k-column matrix*, whose columns are sorted separately. After these steps, the result is already sorted roughly. The step is repeated every time whilst the number of columns is reduced, until the matrix consists of only a single column.

3.9.2 Performance

Shell sort has best case complexity $\mathcal{O}(n \log n)$ and worst case complexity $\mathcal{O}(n^2)$. The average case is in between and has complexity $\mathcal{O}(n \log^2 n)$.

3.9.3 Java implementation

```
1  public class Shellsort {
2      public void sort(int[] a) {
3          int[] cols = {
4              1391376, 463792, 198768, 86961, 33936, 13776,
5              4592, 1968, 861, 336, 112, 48, 21, 7, 3, 1
6          };
7
8          for(int i = 0; i < cols.length; i++) {
9              int h = cols[i];
10
11             for(int j = h; j < a.length; j++) {
12                 int k = j;
13                 int tmp = a[k];
14
15                 while(k >= h && a[k-h] > tmp) {
16                     a[k] = a[k-h];
17                     k = k - h;
18                 }
19
20                 a[k] = tmp;
21             }
22         }
23     }
24 }
```

Listing 9: *Implementation of Shell sort*

4 Contributors

This is to thank the following persons, who have vigorously supported the development of SortSimulation. New contributors (translators, designers, writers of documentation, etc.) are needed constantly – if you are interested in contributing, please get in touch with Peter Folta.

– Allison, Chloë Louise
– Folta, Lucia Sonja
– Folta, Peter
– Müllner, Jan Sebastian

References

[1] Lang, Prof. Dr. Hans Werner: *Algorithmen in Java*. 2nd Edition 2006. Munich: Oldenbourg Wissenschaftsverlag GmbH 2006. ISBN 978-3-486-57938-3, pp. 5–52

[2] Sarbazi-Azad, Dr. Hamid: *Stupid Sort: A new sorting algorithm*. In: *Department of Computer Science Newsletter, University of Glasgow*. No. 4, October 2, 2000. http://sina.sharif.edu/˜azad/stupid-sort.PDF

List of Figures

Listings