SortSimulation

Documentation

– English –

Peter Folta

Version 1.2.1

Internet: http://www.peterfolta.de/software/sortsimulation

Contents

1 Introduction

SortSimulation is a java application, that visualizes different sorting algorithms. This gives you a better understanding of their operating mode and illustrates runtime differences without just comparing dull numbers.

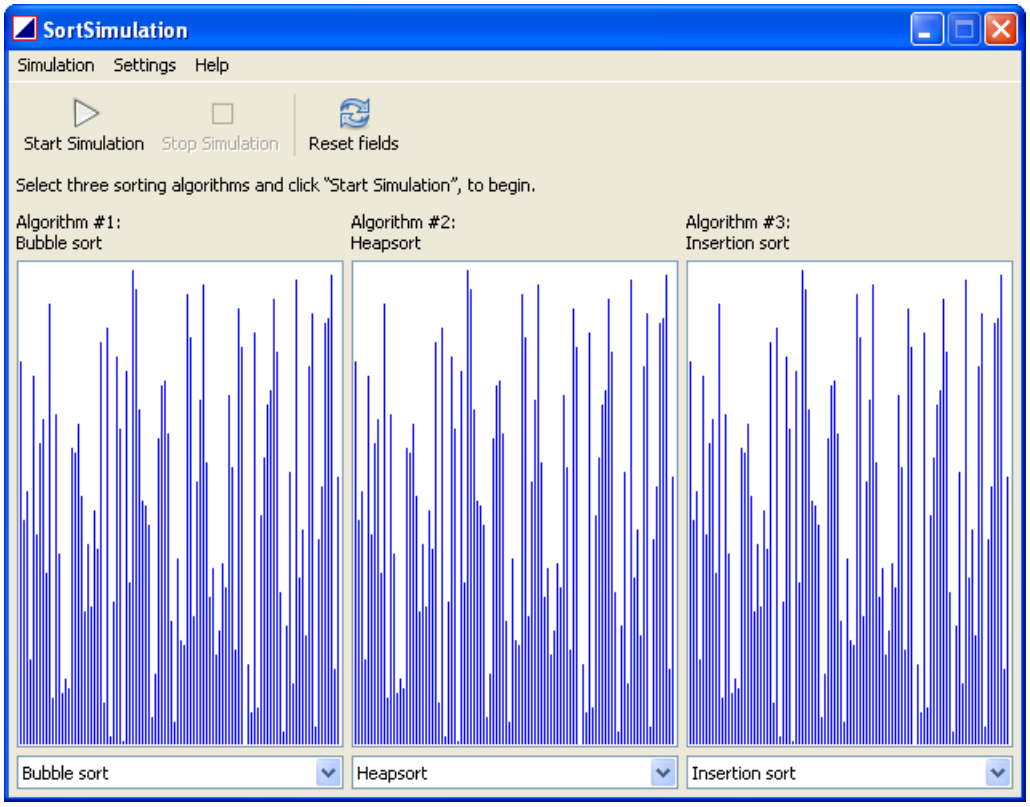

Figure 1: *Main window of SortSimulation*

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

2 Usage

Using SortSimulation is really simple: The sorting fields are randomly filled right after the program launch—every field contains the same initial situation. Three sorting algorithms are also preselected just after the program launch.

To compare three particular sorting algorithms, select them from the combo boxes below the fields. Click on **Start simulation** in the toolbar or the menu, or press the enter key on your keyboard alternatively, 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 paragraph.

2.1 Settings

SortSimulation offers you miscellaneous options to configure the sorting simulations in the way you want it. The available settings are listed in the menu **Settings**:

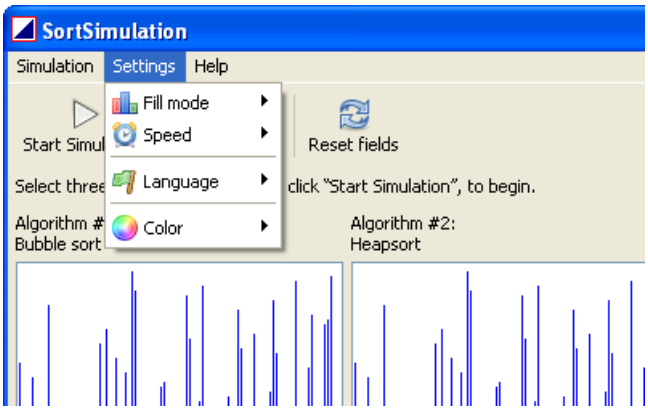

Figure 2: *The settings menu*

To configure the way the sorting fields are filled, the option **Fillmode** offers you two possibilities: Randomly (preselected; `Ctrl+R`) and Inverse (`Ctrl+I`). If **Randomly** is selected, the elements are ordered randomly in the fields when the user commits the action **Reset fields**. This allows you to try the different sorting algorithms on totally unsorted data in a realistic way.

In contrast, the mode **Inverse** fills the fields already sorted—but backwards (descending instead of ascending). Using this setting gives you the possibility to take a closer look on the efficiency of the algorithms sorting backwards pre-ordered series.

The submenu **Speed** allows you to configure the speed of the simulation. You can choose the speed between five levels. While algorihms like Bubble sort are quite slow, it is worth to select one of the higher speed levels; while running a sorting algorithm like Quicksort, it's 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 doesn't create or modify any of the files stored on your computer, that's why SortSimulation's user interface language is always set to English after the program launch, even if you changed it before. To switch between the offered languages, go to the submenu **Language**.

The last submenu **Color** is used to configure the color of the bars. Blue is set by default, but you have the choice between eight different colors.

3 Sorting algorithms

3.1 Bubble sort

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

3.1.1 Idea of Bubble sort

Bubblesort works by comparing two list 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 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 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**.

3.2.1 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.

3.2.2 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 2: *Implementation of Heapsort*

3.3 Insertion sort

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

3.3.1 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.

Insertion sort is inefficient, because this sorting algorithm often needs to move elements over long distances.

3.3.2 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 3: *Implementation of Insertion sort*

3.4 Merge sort

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

3.4.1 Idea of Merge sort

Merge sort splits the sequence into several smaller sequences, which will be sorted each for itself. After that the sorted, small sequences are united in one completly sorted sequence.

3.4.2 Java implementation

```
1  public class Mergesort {
2      private int[] a;
3      private int[] b;
4      private int n;
5
6      public void sort(int[] a) {
7          this.a = a;
8          n = a.length;
9          b = new int[n];
10         mergesort(0, n-1);
11     }
12
13     private void mergesort(int lo, int hi) {
14         if(lo < hi) {
15             int m = (lo + hi) / 2;
16             mergesort(lo, m);
17             mergesort(m+1, hi);
18             merge(lo, m, hi);
19         }
20     }
21
22     private void merge(int lo, int m, int hi) {
23         int i = lo;
24         int j = hi;
```

```
25          int k = lo;
26
27          while(i <= m) b[k++] = a[i++];
28          while(j > m) b[k++] = a[j--];
29
30          i = lo;
31          j = hi;
32          k = lo;
33
34          while(i <= j) {
35              if(b[i] <= b[j]) a[k++] = b[i++];
36              else a[k++] = b[j--];
37          }
38      }
39  }
```

Listing 4: *Implementation of Merge sort*

3.5 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.

3.5.1 Idea of Quicksort

The sequence will initially be split into two part. The first section contains all elements less than or equal to all elements in the second section (**divide**). Then the two pieces 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.

3.5.2 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 5: *Implementation of Quicksort*

3.6 Selection sort

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

3.6.1 Idea of Selection sort

Selection sort divides the sequence in a **sorted** and a **unsorted** field. The sorted part is empty at the beginning. Selection sort examines the smallest element in the unsorted part 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.

3.6.2 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 6: *Implementation of Selection sort*

3.7 Shell sort

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

3.7.1 Idea of Shellsort

Shellsort compensates 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 coarse. The step is repeated every time while the number of columns is reduced, until the matrix consists of only a single column.

3.7.2 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 7: *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.

4.1 Translators

- Folta, Lucia Sonja – Russian
- Folta, Peter – English, German
- Müllner, Jan Sebastian – French, Spanish

5 Contact

Peter Folta
Humboldtstrasse 9
34497 Korbach
Germany

E-mail: mail@peterfolta.de
Internet: http://www.peterfolta.de/

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

List of Figures

Listings