SortSimulation

Dokumentation

– Deutsch –

Version 2.0.0 Alpha

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

Peter Folta

Inhaltsverzeichnis

1 Einführung

SortSimulation ist ein Java-Programm, das diverse Sortierverfahren visuell darstellt. Dies ermöglicht einerseits ein besseres Verständnis der Funktionsweise der verschiedenen Sortieralgorithmen und verdeutlicht andererseits die Laufzeitunterschiede, ohne dabei lediglich trockene Zahlen zu vergleichen.

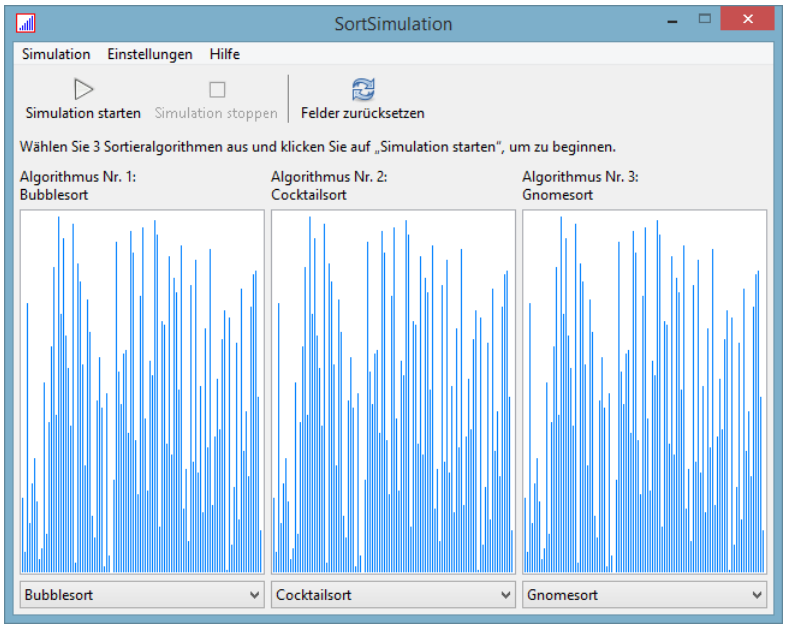

Abbildung 1: *Das Hauptfenster von SortSimulation*

Diese Dokumentation enthält eine kurze Einführung in die Bedienung von SortSimulation und stellt anschließend die unterstützten Sortierverfahren inklusive Implementierung in Java vor.

2 Bedienung

Die Bedienung von SortSimulation ist denkbar einfach: Direkt nach dem Programmstart sind die Sortierfelder bereits zufällig gefüllt. Dabei liegt in jedem der drei Felder die gleiche Ausgangssituation vor. Ebenfalls sind nach dem Programmstart drei verschiedene Sortierverfahren voreingestellt; dies ist an den Auswahlboxen unterhalb der Sortierfelder erkennbar.

Um nun drei Sortieralgorithmen miteinander zu vergleichen, wählen Sie aus den Auswahlboxen die entsprechenden Verfahren aus. Die Simulation startet, sobald Sie auf die Schaltfläche *Simulation starten* in der Toolbar klicken, den entsprechenden Eintrag im Menü *Simulation* auswählen oder die Eingabetaste drücken. Möchten Sie eine laufende Simulation abbrechen, genügt ein Druck auf die Escape-Taste oder ein Klick auf die Schaltfläche *Simulation stoppen*. Um nach einer abgeschlossenen oder abgebrochenen Simulation wieder unsortierte Felder zu erhalten, klicken Sie auf die Schaltfläche *Felder zurücksetzen* oder drücken Sie `Strg+N`.

SortSimulation bietet einige Einstellungen, mit denen Sie die Simulation anpassen können. Auf diese Einstellungen wird im folgenden Abschnitt ausführlich eingegangen.

Für fortgeschrittene Nutzer besteht außerdem die Möglichkeit, SortSimulation beim Programmstart Parameter zu übergeben, mehr dazu in Abschnitt 2.2 auf Seite 3.

2.1 Einstellungen

Ihnen stehen in SortSimulation diverse Einstellungen zur Verfügung, mit denen Sie die Sortier-Simulationen ihren Wünschen entsprechend anpassen können. Diese Einstellungsmöglichkeiten finden Sie im Menü *Einstellungen*:

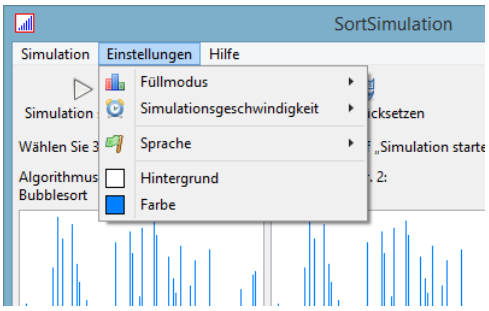

Abbildung 2: *Menü „Einstellungen“*

Mithilfe der Option *Füllmodus* können Sie festlegen, auf welche Weise die Sortierfelder gefüllt werden sollen, wenn die Aktion *Felder zurücksetzen* ausgeführt wird. Dabei stehen insgesamt vier Modi zur Verfügung: Zufällig (voreingestellt; `Strg+R`), invertiert (`Strg+I`), beinahe sortiert (`Strg+A`) und vorsortiert (`Strg+P`). Im Modus *Zufällig* werden die Elemente bei jeder Neufüllung der Felder zufällig angeordnet. Damit lassen sich die unterschiedlichen Sortierverfahren auf realistische Weise mit völlig unsortierten Daten ausprobieren.

Im Modus *Invertiert* hingegen werden die Felder bereits sortiert gefüllt – allerdings umgekehrt (absteigend anstatt aufsteigend). Mit dieser Einstellung lässt sich beobachten, wie effizient die Sortierverfahren umgekehrt geordnete Folgen sortieren.

Der Modus *Beinahe sortiert* erzeugt eine zufällige Sortierung der Felder, deren Abweichung zur tatsächlichen Sortierung nur gering ist. Die grobe Struktur der sortierten Daten ist bereits erkennbar. Diese Einstellung visualisiert eindrücklich, welche Sortieralgorithmen sich besonders zum Sortieren von Datenmengen eignen, die nur marginale Abweichungen zur sortierten Folge besitzen.

Die letzte Option *Vorsortiert* befüllt alle Felder mit bereits sortierten Daten, sie dient der Veranschauulichung der Arbeitweise von Sortieralgorithmen auf bereits sortierten Daten.

Das Untermenü *Simulationsgeschwindigkeit* ermöglicht es Ihnen, die Geschwindigkeit der Simulation anzupassen. Sie haben dabei die Wahl zwischen fünf Stufen. Während Algorithmen wie etwa Bubblesort sehr langsam sind und sich dort eine hohe Simulationsgeschwindigkeit anbietet, lohnt es sich, beispielsweise Quicksort in einer langsamen Stufe zu beobachten, um die Vorgehensweise dieses Verfahrens besser verstehen zu können. Die einzelnen Geschwindigkeitsstufen lassen sich auch

über die Tastatur mithilfe der Tastenkombinationen `Strg+Umschalt+(1-5)` erreichen.

SortSimulation schreibt oder verändert keine Dateien auf Ihrem Computer, das Programm startet deshalb jedes Mal mit der englischen Benutzeroberfläche. Sie können die Sprache im Menü *Sprache* (engl. *Settings* > *Language*) verändern.

Die letzten beiden Menüpunkte *Hintergrund* und *Farbe* ermöglichen es, die Farbe der Balken sowie die Hintergrundfarbe der Felder einzustellen.

2.2 Parameter

SortSimulation zeigt standardmäßig drei Felder nebeneinander an. Dies ermöglicht es, drei Sortieralgorithmen parallel zu visualisieren. Durch Angabe eines Parameters beim Programmstart lässt sich die Anzahl der angezeigten Felder anpassen. Als Parameter übergeben Sie einfach die gewünschte Anzahl. Gültige Werte liegen hierbei zwischen 2 und 9.

Um SortSimulation beispielsweise mit 5 Simulationsfeldern zu starten, führen Sie folgenden Befehl auf einer Kommandozeile aus:

```
1 SortSimulation 5
```

SortSimulation startet anschließend wie gewohnt, zeigt jedoch die angegebenen 5 Felder anstelle der standardmäßigen 3 an:

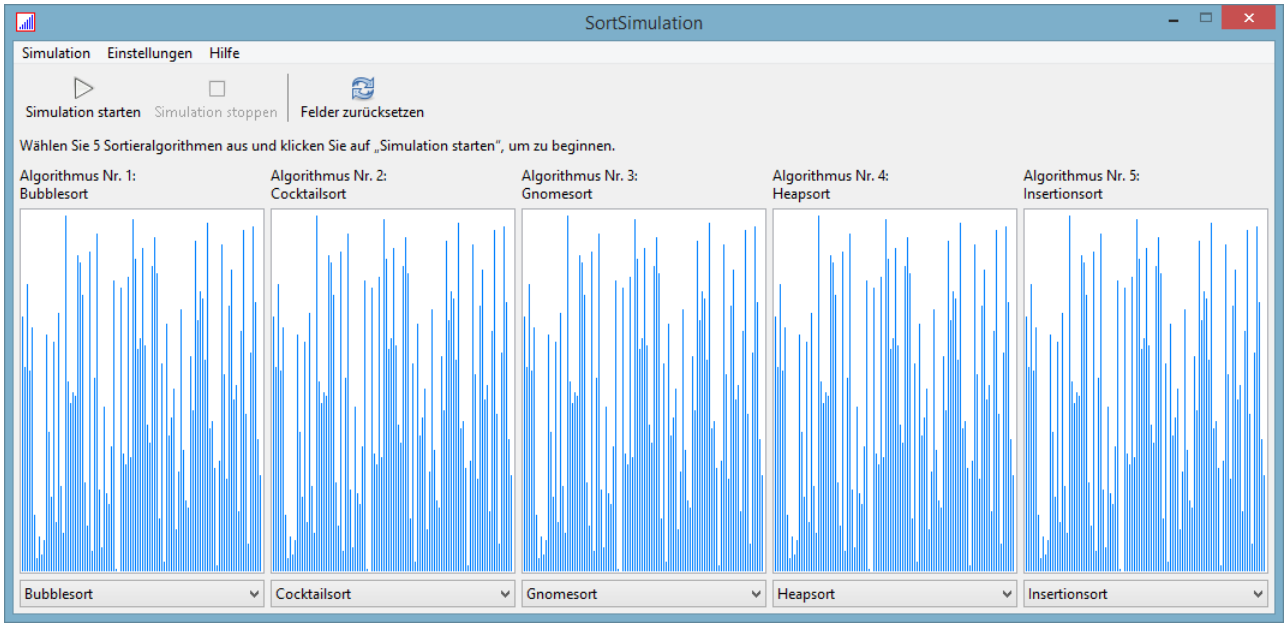

Abbildung 3: *SortSimulation mit 5 Feldern*

Bedenken Sie bei dieser Startvariante bitte, dass das Hauptfenster von SortSimulation sehr breit werden kann, und gegebenenfalls die Anzeigefläche Ihres Bildschirms überschreiten kann.

Sollten Sie einen ungültigen Wert angegeben haben, wird dies mit einer entsprechenden Fehlermeldung quittiert:

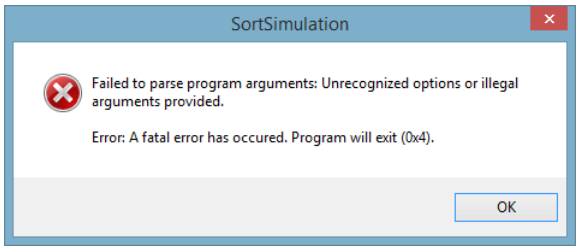

Abbildung 4: *Fehlermeldung bei ungültiger Parameterangabe*

3 Sortierverfahren

3.1 Bubblesort

Bubblesort ist ein einfacher Sortieralgorithmus, der Elemente durch *schrittweises Vergleichen* sortiert. Da Bubblesort nicht effizient ist, wird der Algorithmus häufig zur Demonstration eines schlechten Sortierverfahrens verwendet.

3.1.1 Idee von Bubblesort

Bubblesort vergleicht zwei benachbarte Elemente und vertauscht diese, wenn sie nicht in der richtigen Reihenfolge stehen. Dieser Vorgang wird solange wiederholt, bis alle Elemente an der richtigen Stelle stehen und damit sortiert sind.

3.1.2 Laufzeit

Bubblesort besitzt im Allgemeinen Laufzeit $\mathcal{O}(n^2)$, erreicht jedoch bei bereits sortierten Folgen lineare Laufzeit $\mathcal{O}(n)$.

3.1.3 Implementierung in Java

```
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: *Implementierung von Bubblesort*

3.2 Cocktailsort

Cocktailsort (auch *Shakersort* genannt) ist ein einfacher Sortieralgorithmus, der eine Variation des bekannten Bubblesort darstellt.

3.2.1 Idee von Cocktailsort

Die zu sortierenden Elemente werden abwechselnd nach oben und nach unten durchlaufen. Es werden jeweils zwei benachbarte Elemente verglichen und gegebenenfalls getauscht. Dadurch sammeln sich große Elemente am oberen Ende und kleine Elemente am unteren Ende des Arrays an.

3.2.2 Laufzeit

Cocktailsort besitzt im Allgemeinen Laufzeit $\mathcal{O}(n^2)$, strebt jedoch gegen $\mathcal{O}(n)$ für beinahe sortierte Folgen.

3.2.3 Implementierung in Java

```
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: *Implementierung von Cocktailsort*

3.3 Gnomesort

Gnomesort ist ein sehr einfacher Sortieralgorithmus, der Elemente durch *schrittweises Vergleichen* sortiert. Er wurde erstmalig im Jahr 2000 von *Hamid Sarbazi-Azad* unter dem Namen *Stupid Sort* veröffentlich. Später wurde der Name in Gnomesort geändert. Gnomesort hat den Vorteil, mit nur einer einzigen Schleife (und damit ohne geschachtelte Schleifen) auszukommen.

3.3.1 Idee von Gnomesort

Gnomesort arbeitet, wie man sich vielleicht die Arbeitsweise eines Gartenzwerges (engl. *garden gnome*) vorstellen kann: Man nehme an, der Zwerg möchte Blumentöpfe sortieren. Er steht zunächst am linken Ende der in einer Reihe aufgestellten Töpfe. Der Zwerg vergleicht nun zwei benachbarte Töpfe: Stimmt die Reihenfolge, geht er einen Topf nach rechts. Stimmt sie nicht, werden die Töpfe getauscht, und der Zwerg geht einen Schritt nach links, sofern er nicht am linken Ende der Topfreihe steht. In diesem Falle geht der Gartenzwerg einen Schritt nach rechts. Dies wird solange wiederholt, bis er am rechten Ende (und damit am letzten Blumentopf) angekommen ist.

3.3.2 Laufzeit

Gnomesort besitzt im Allgemeinen Laufzeit $\mathcal{O}(n^2)$, strebt jedoch gegen $\mathcal{O}(n)$ für beinahe sortierte Folgen.

3.3.3 Implementierung in Java

```
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: *Implementierung von Gnomesort*

3.4 Heapsort

Heapsort ist ein schnelles Sortierverfahren, das 1964 von *Robert W. Floyd* und *J. W. J. Williams* entwickelt wurde. Heapsort ist eine Verbesserung des Sortieralgorithmus *Selectionsort*.

3.4.1 Idee von Heapsort

Heapsort verwendet zur Sortierung eine besondere Datenstruktur: den *Heap*. Diese Datenstruktur basiert auf einem (fast) vollständigen *binären Baum*. Ein binärer Baum ist (fast) vollständig, wenn alle Ebenen, außer möglicherweise der letzten, vollständig sind.

Wenn die zu sortierende Folge als Heap vorliegt, kann das größte Element der *Wurzel* des Baumes entnommen und ausgegeben werden. Um an das nächstgrößte Element zu gelangen, muss der Heap zunächst *neu angeordnet* werden.

3.4.2 Laufzeit

Heapsort besitzt stets Laufzeit $\mathcal{O}(n \log n)$ und gehört somit zu den besten vergleichsbasierten Sortieralgorithmen.

3.4.3 Implementierung in Java

```
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: *Implementierung von Heapsort*

3.5 Insertionsort

Insertionsort ist ein einfaches Sortierverfahren. Es ist nicht so effizient, wie andere komplexere Algorithmen, dafür aber *einfach zu implementieren* und benötigt nur eine kurze Laufzeit bei sehr kleinen oder bereits vorsortierten Datenmengen.

3.5.1 Idee von Insertionsort

Insertionsort entnimmt der unsortierten Menge ein Element und fügt es an der richtigen Stelle in der Ausgabefolge ein. Ist die Folge noch leer, wird das Element an die erste Position eingefügt.

Insertionsort ist ineffizient, da bei diesem Sortierverfahren Elemente oft über weite Strecken verschoben werden müssen.

3.5.2 Laufzeit

Insertionsort besitzt im Allgemeinen Laufzeit $\mathcal{O}(n^2)$, strebt jedoch gegen $\mathcal{O}(n)$ für beinahe sortierte Folgen.

3.5.3 Implementierung in Java

```
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: *Implementierung von Insertionsort*

3.6 Mergesort

Mergesort ist ein rekursiver und stabiler Sortieralgorithmus, welcher wie *Quicksort* auf dem *Divide-and-Conquer-Prinzip* basiert. Mergesort wurde 1945 von John von Neumann vorgestellt.

3.6.1 Idee von Mergesort

Mergesort zerlegt die zu sortierende Folge in mehrere kleine Folgen, die, jede für sich, sortiert werden. Anschließend werden die sortierten, kleinen Folgen im Reißverschlussverfahren wieder zu einer großen Folge zusammengesetzt, bis alle Elemente sortiert sind.

3.6.2 Laufzeit

Mergesort besitzt stets Laufzeit $\mathcal{O}(n \log n)$ und gehört somit zu den besten vergleichsbasierten Sortieralgorithmen.

3.6.3 Implementierung in Java

```
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 6: *Implementierung von Mergesort*

3.7 Quicksort

Quicksort ist eines der *schnellsten Sortierverfahren* und basiert auf dem *Divide-and-Conquer-Prinzip*. Der rekursive Quicksort-Algorithmus wurde in seiner ursprünglichen Fassung im Jahre 1960 von *C. Antony R. Hoare* entwickelt.

3.7.1 Idee von Quicksort

Die zu sortierende Folge wird zunächst so in zwei Teilstücke zerlegt, dass alle Elemente im ersten Teilstück kleiner oder gleich allen Elementen im zweiten Teilstück sind (*divide*). Anschließend werden die beiden Teilstücke unabhängig voneinander rekursiv nach dem gleichen Verfahren sortiert (*conquer*). Im letzten Schritt ergeben die zusammengesetzten Teilstücke die sortierte Folge (*combine*).

Die Aufteilung wird mit Hilfe eines *Pivotelementes* realisiert, das im ersten Schritt aus der Folge gewählt wird. Alle Elemente der Folge, die *kleiner* als das Pivotelement sind, kommen in das erste Teilstück. Alle Elemente, die *größer* als das Pivotelement sind, kommen in das zweite Teilstück. Bei Elementen, die genauso groß wie das Pivotelement sind, spielt es keine Rolle, in welches Teilstück sie kommen.

3.7.2 Laufzeit

Quicksort besitzt im Allgemeinen Laufzeit $\mathcal{O}(n \log n)$ und gehört somit zu den besten vergleichsbasierten Sortieralgorithmen. Bei einer Worst Case-Eingabe für Quicksort beträgt die Laufzeit $\mathcal{O}(n^2)$.

3.7.3 Implementierung in Java

```
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: *Implementierung von Quicksort*

3.8 Selectionsort

Selectionsort ist ein naiver Sortieralgorithmus, der in-place arbeitet. Er ist mit Insertionsort vergleichbar.

3.8.1 Idee von Selectionsort

Selectionsort teilt die zu sortierende Folge in einen *sortierten* und einen *nicht-sortierten* Bereich. Der sortierte Teil ist zu Beginn leer. Selectionsort sucht das kleinste Element in der unsortierten Teilfolge und vertauscht es mit dem ersten Element. Nach dem Schritt ist die Folge bis zu dieser Position sortiert. Der sortierte Teilbereich ist um ein Element größer geworden, der unsortierte um ein Element kleiner. Das Verfahren wird so oft wiederholt, bis die gesamte Folge sortiert ist.

3.8.2 Laufzeit

Selectionsort besitzt stets Laufzeit $\mathcal{O}(n^2)$.

3.8.3 Implementierung in Java

```
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: *Implementierung von Selectionsort*

3.9 Shellsort

Shellsort ist ein Sortierverfahren, das auf *Insertionsort* basiert. Shellsort wurde im Jahre 1959 von *Donald L. Shell* entwickelt.

3.9.1 Idee von Shellsort

Shellsort kompensiert den Nachteil von Insertionsort, Elemente über weite Strecken verschieben zu müssen. Shellsort erzeugt dazu eine *k-spaltige Matrix*, deren Spalten einzeln sortiert werden. Nach diesen Schritten ist die Folge bereits grob sortiert. Dieser Schritt wird wiederholt, wobei bei jeder Durchführung die Anzahl der Spalten verringert wird, bis die Matrix nur noch aus einer einzelnen Spalte besteht.

3.9.2 Laufzeit

Shellsort besitzt im Best Case Laufzeit $\mathcal{O}(n \log n)$, im Worst Case $\mathcal{O}(n^2)$. Der Average Case liegt dazwischen, bei $\mathcal{O}(n \log^2 n)$.

3.9.3 Implementierung in Java

```
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: *Implementierung von Shellsort*

4 Mitwirkende

Hiermit soll nachfolgenden Personen gedankt werden, die die Entwicklung von SortSimulation tatkräftig unterstützt haben. Neue Mitwirkende (Übersetzer, Designer, Verfasser von Dokumentationen etc.) werden ständig gesucht – setzen Sie sich bei Interesse mit Peter Folta in Verbindung.

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

Literatur

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

[2] Sarbazi-Azad, Dr. Hamid: *Stupid Sort: A new sorting algorithm*. In: *Department of Computer Science Newsletter, University of Glasgow*. Nr. 4, 2. Oktober 2000. http://sina.sharif.edu/~azad/stupid-sort.PDF

Abbildungsverzeichnis

Listings