Ludwig-Maximilians-Universität München
Institut für Informatik
PD Dr. Matthias Renz
Dr. Erich Schubert, Daniel Kaltenthaler

München, 13.05.2015

Algorithmen und Datenstrukturen

SS 2015

Übungsblatt 5: Sortieralgorithmen II

Besprechung: 28.5.–02.06.

Abgabe aller mit **Punkten** versehenen Aufgaben bis 27.5.2015 13:25

Zur Wiederholung:

Eine Sortierung von Objekten basiert auf der Definition einer vollständigen Ordnung "$<$" auf dem Wertebereich der Objekte. Eine vollständige Ordnung ist u.a. dadurch charakterisiert, dass für zwei beliebige Objekte a und b genau eine der folgenden Beziehungen gilt:

$$a < b \qquad \text{oder} \qquad b < a \qquad \text{oder} \qquad a = b$$

Diese Eigenschaft ist auch bekannt als "*Gesetz der Trichotomie*" (korrekt wäre eigentlich "*Tritomie*"). Üblicherweise nimmt man auch an, dass für drei beliebige Objekte a, b und c gilt: Wenn $a < b$ und $b < c$, dann $a < c$. Diese Eigenschaft ist bekannt als "*Transitivität*". Als Ergebnis eines Sortierverfahrens auf n Schlüsseln K_i, $i = 1, \ldots, n$ erwartet man eine Permutation $p(1), \ldots, p(n)$ der Indizes, so dass die Schlüssel in nicht-absteigender Ordnung angeordnet sind:

$$K_{p(1)} \leq K_{p(2)} \leq \ldots \leq K_{p(n)}$$

Ein Sortierverfahren kann zusätzlich *stabil* sein. Dann wird von der Permutation p verlangt, dass gilt:

$$(K_{p(i)} = K_{p(j)} \wedge i < j) \Rightarrow p(i) < p(j)$$

Aufgabe 5-1 Stabilität von Sortierverfahren **2 Punkte**

Entscheiden Sie, ob folgende Aussagen zutreffen und begründen Sie Ihre Entscheidung:

(a) "Sortieren durch Abzählen ist stabil."

(b) "Sortieren durch direktes Einfügen (Insertion-Sort) ist stabil."

Aufgabe 5-2 Eindeutigkeit von Sortierverfahren 4 Punkte

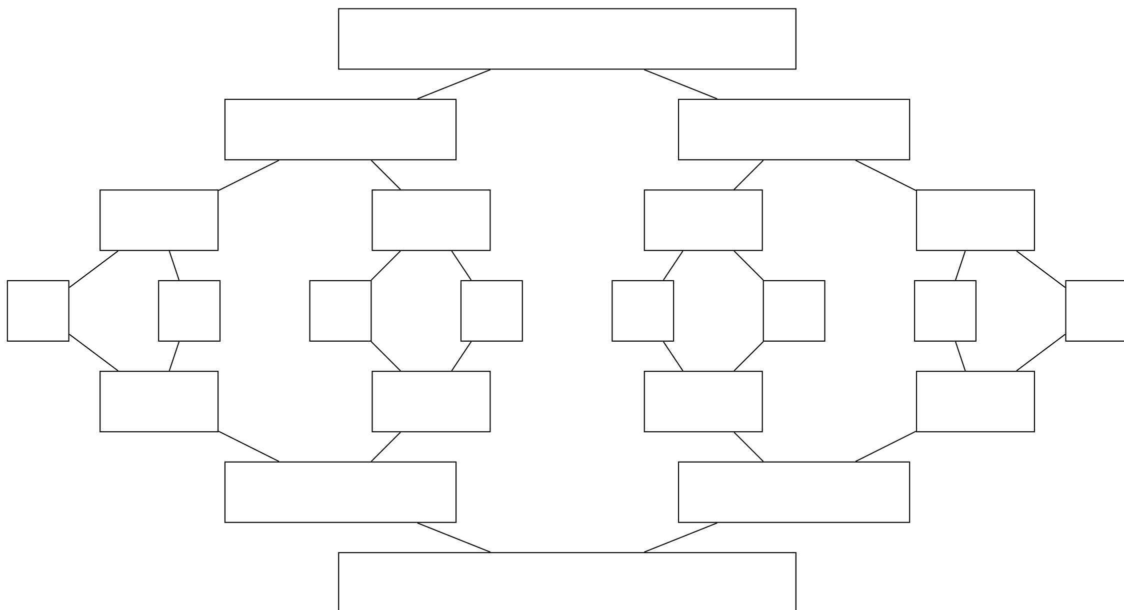
(a) Gegeben sei ein *stabiles* Sortierverfahren, basierend auf einer vollständigen Ordnung, die den Gesetzen der *Trichotomie* und *Transitivität* genügt. Beweisen Sie, dass die Permutation $p(1), p(2), \ldots, p(n)$, die man als Ergebnis des Sortierverfahrens erhält, *eindeutig bestimmt* ist.

(b) Nun sei eine Ordnung "$<$" auf $K_1, \ldots, K_n$ gegeben, die dem Gesetz der *Trichotomie* genügt, aber *nicht transitiv* ist. Begründen Sie, warum es auch möglich ist, die Schlüssel *stabil* zu sortieren, wenn die vorausgesetzte Ordnung *nicht transitiv* ist. Nennen Sie zwei Beispiele für Sortierverfahren aus der Vorlesung, die im nicht-transitiven Fall eine stabile Sortierung ermöglichen.

Aufgabe 5-3 Merge-Sort Tutoraufgabe

Geben Sie die Zwischenergebnisse des Merge-Sort-Algorithmus in dem untenstehenden Graphen an, der die in der Vorlesung verwendete Struktur besitzt. Die obere Hälfte des Graphen stellt die verschiedenen Rekursionsstufen des Divide-Schritts dar, der obere Knoten des Graphen enthält das unsortierte Array. In die untere Hälfte des Graphen soll der Merge-Schritt eingetragen werden.

Zu sortieren ist die folgende Liste von Zahlen: 14, 84, 75, 25, 49, 45, 64, 78



Aufgabe 5-4 MergeSort Implementieren 6 Punkte

- Vervollständigen Sie folgende MergeSort-Implementierung:

```java
public class MergeSort {
  /** Array of int sortieren mittels MergeSort */
  public static void sort(int[] daten) {
    mergesort(daten, new int[daten.length], 0, daten.length - 1);
  }

  /** Teilarray of int sortieren mittels MergeSort */
  public static void mergesort(int[] daten, int[] tmp, int start, int ende) {
    if (start >= ende) { return; } // Max 1 Element: fertig!
    // Divide:
    final int half = (start + ende) / 2;
    // Conquer:
    mergesort(daten, tmp, start, half);
    mergesort(daten, tmp, half + 1, ende);
    // Merge:
    int i = start, j = half + 1, k = start;
    while (i <= half && j <= ende) {
      // TODO: Implementieren
    }
    while (i <= half) { // Rechter Teil schon leer!
      // TODO: Implementieren
    }
    while (j <= ende) { // Linker Teil schon leer!
      // TODO: Implementieren
    }
    // aus tmp zurueck kopieren:
    for (int l = start; l < k; l++) {
      daten[l] = tmp[l];
    }
  }
}
```

  Das Array `int[] tmp` müssen Sie für diesen Algorithmus als temporäre Kopie verwenden.

- Compilieren und Testen Sie ihre Implementierung, um Fehler zu finden!

  Verwenden Sie dazu einen ausreichend großen, zuvor nicht sortierten Datensatz!

- Geben Sie die Lösung als `.java` Datei ab (*nicht* als PDF o.Ä.).

  Nicht compilierender Code muss von den Korrektoren mit 0 Punkten bewertet werden, und der Java-Compiler (insbesondere in ihrer IDE) *hilft* ihnen, Fehler zu finden – ebenso der Test.

Aufgabe 5-5 QuickSort I **Tutoraufgabe**

Sortieren Sie das folgende Array mit dem QuickSort-Verfahren *exakt wie es in der Vorlesung besprochen wurde*: Wählen Sie stets das erste Element als Pivot-Element.
Verwenden Sie eine Tabelle mit der folgenden Struktur:

Anfang:	54	74	4	19	59	53	79	66	67	83	3	97	28	86	5	56
Vor Pivot:																
Nach Pivot:																
Vor Pivot:																
Nach Pivot:																
Vor Pivot:																
Nach Pivot:																
Vor Pivot:																
Nach Pivot:																
Vor Pivot:																
Nach Pivot:																
Vor Pivot:																
Nach Pivot:																
Vor Pivot:																
Nach Pivot:																
Vor Pivot:																
Nach Pivot:																
Vor Pivot:																
Nach Pivot:																
Vor Pivot:																
Nach Pivot:																

⋮

Markieren Sie für jeden rekursiven Durchlauf des Verfahrens das betroffene Intervall *vor und nach* nach dem Verschieben des Pivot-Elements. Es reicht, wenn Sie das Pivot-Element und diejenigen Elemente eintragen, deren Position sich ändert (um die Schreibarbeit zu reduzieren)!

Streichen Sie *nie* richtige Elemente durch: Durchgestrichenes können wir nicht bewerten, die Lösung muss für uns eindeutig erkennbar sein! Eigene Markierungen (Zeiger, Intervalle, Vertauschungspfeile etc.) können Sie zwischen den Zeilen eintragen. Diese Tabellenform ist darauf ausgelegt, dass Sie nichts durchstreichen müssen.

Aufgabe 5-6 ShellSort Implementieren (gestrichen)

- Wiederholen Sie den Algorithmus InsertionSort.
  Wenn Sie bei InsertionSort die Schrittweite auf `incr` erhöhen wollen, welche Änderungen müssen Sie vornehmen?
  Wenn `incr=1` sollten Sie den normalen InsertionSort erhalten, aber sortiert der Algorithmus für `incr > 1` noch korrekt?

- Wiederholen Sie den Algorithmus ShellSort aus der Vorlesung.
  Welche Zeilen führen den InsertionSort durch?
  Warum sortiert ShellSort korrekt, obwohl InsertionSort doch nur für `incr=1` korrekt war?

- Welche Anordnung von Daten ist der ungünstigste Fall für InsertionSort, was ist die optimale Anordnung, und warum ist ShellSort hier ein guter Kompromiss?

- Vervollständigen Sie folgende ShellSort-Implementierung:

```java
public class ShellSort {
  /** Array of int sortieren mittels ShellSort */
  public static void sort(int[] daten) {
    for(int incr = daten.length / 2; incr > 0; incr = incr / 2) {
      // Insertion sort, mit Schrittweite incr
      for(int i = /* TODO */; i < daten.length; i++) {
        for(int j = i; j >= /* TODO */; j -= /* TODO */) {
          if (daten[/* TODO */] <= daten[/* TODO */]) {
            break;
          }
          SortUtil.vertausche(daten, /* TODO */);
        }
      }
    }
  }
}
```

- Compilieren und Testen Sie ihre Implementierung!

  Verwenden Sie dazu einen ausreichend großen, zuvor nicht sortierten Datensatz!