Dr. Andreas Abel
Dr. Ulrich Schöpp
Dr. Steffen Jost

Ludwig-Maximilians-Universität München
Institut für Informatik
31. Januar 2011

14. Musterlösung zur Vorlesung

Einführung in die Informatik: Programmierung und Software-Entwicklung

Hinweis: *Dieses Blatt geht ausnahmsweise nicht in die Klausur-Bonus-Regelung ein.*

Aufgabe 14-1 *ArrayList* (0 Punkte) (Geben Sie alle `.java`-Dateien Ihrer Lösung ab)
Lösen Sie erneut Aufgabe 4-2, aber ersetzen Sie die Verwendung von Arrays vollständig durch ArrayList, wie in der Vorlesung empfohlen. Dabei dürfen Sie alle Fähigkeiten von ArrayList einsetzen.

In Aufgabe 4-2 musste in ein gegebenes, sortiertes Array eine beliebige, vorgegebene Zahl eingefügt werden, so dass das resultierende Array weiterhin sortiert bleibt.

Implementieren Sie Ihre Lösung dieses Mal als eine eigenständige, statische Methode. Die Methode soll generisch sein, d.h. die Methode kann auf beliebige Arrays angewendet werden, sofern die Elemente des Arrays vergleichbar sind: `A extends Comparable<A>`. Das Argument des Typs `ArrayList<A>` soll durch das Einfügen nicht verändert werden; die Methode liefert also ein Kopie des Arrays zurück.

```
public static <A extends Comparable<A>>
  ArrayList<A> insert(A element, ArrayList<A> list){
    // Ihre Aufgabe!!!
}
```

LÖSUNG
Etwas ausführlicheren, kommentierten Java-Code zur Lösung dieser Aufgabe findet sich auch noch zum separaten Download auf der Vorlesungshomepage.

```
public static <A extends Comparable<A>>
  ArrayList<A> insert(A element, ArrayList<A> list) {
    boolean inserted = false;
    // Jede ArrayList ist auch eine Collection. Die Ordnung wird beibehalten.
    ArrayList<A> result = new ArrayList<A>(list);
    // Richtige Stelle finden
    int i = 0;
    while (i < result.size() && element.compareTo(result.get(i)) > 0) {
      i++;
    }
    // An der gefundenen Stelle einfügen
    result.add(i, element);
    return result;
}
```

Aufgabe 14-2 *Komplexität* (0 Punkte) (Abgabeformat: `.txt` oder `.pdf`)

a) Gegeben ist der folgende Code zum Sortieren einer generischen Liste:

```
public static <A extends Comparable<A>> void sort(List<A> a) {
      int length = a.size();
      for (int outer = length - 1; outer > 0; outer--) {
          for (int inner = 0; inner < outer; inner++) {
              if (a.get(inner).compareTo(a.get(inner + 1)) > 0) {
                  // Vertauschen
                  A x = a.get(inner);
                  a.set(inner, a.get(inner + 1));
                  a.set(inner + 1, x);
              }
          }
      }
  }
```

Berechnen Sie die Komplexität dieses Algorithmus in O-Notation, einmal für die Annahme, dass die übergebene Liste ein Objekt der Klasse `LinkedList` ist und weiteres Mal unter der Annahme, dass die Liste ein Objekt der Klasse `ArrayList` ist.

Sie dürfen dabei folgende Annahmen zur Komplexität der einzelnen Anweisungen machen, welche wir analog zu Folie 18 zum Thema "Prozeduren und Komplexität von Algorithmen" angeben. Folie 18 erklärt, wie sich die Komplexität für die einzelnen Anweisungen zusammensetzt. Zum Beispiel setzt sich der Aufwand für eine While-Schleife mit einer beliebigen Abbruchbedingung $expr_B$ und einem beliebigen Schleifenrumpf $expr_S$ aus der Summe des Aufwands für $expr_B$ und $expr_S$ multipliziert mit der Anzahl der Schleifendurchläufe (welche man sich überlegen muss) zusammen, plus zusätzlich einmal dem Aufwand für $expr_B$. In anderen Worten: Wenn die Schleife n-mal durchlaufen wird, fallen n-mal die Kosten für die Ausführung des Scheifenrumpfes $expr_S$ an, plus $(n+1)$-mal die Kosten zum Prüfen der Abbruchbedingung $expr_B$, da die Abbruchbedingung ja einmal mehr geprüft wird, als die Schleife durchlaufen wird.

Für `LinkedList`:

Ausdruck	Laufzeit
expr`.size()`	1 + (Laufzeit von *expr*)
$expr_1$`.compareTo(`$expr_2$`)`	1 + (Laufzeit von $expr_1$) + (Laufzeit von $expr_2$)
expr`.get(i)`	i + (Laufzeit von *expr*)
$expr_1$`.set(i,`$expr_2$`)`	i + (Laufzeit von $expr_1$) + (Laufzeit von $expr_2$)

Für `ArrayList`:

Ausdruck	Laufzeit
expr`.size()`	1 + (Laufzeit von *expr*)
$expr_1$`.compareTo(`$expr_2$`)`	1 + (Laufzeit von $expr_1$) + (Laufzeit von $expr_2$)
expr`.get(i)`	1 + (Laufzeit von *expr*)
$expr_1$`.set(i,`$expr_2$`)`	1 + (Laufzeit von $expr_1$) + (Laufzeit von $expr_2$)

LÖSUNG

Sei n die Größe der Eingabeliste `a`. Es gibt zwei verschachtelte Schleifen, welche jeweils maximal n Durchläufe machen. Dies bedeutet, das wir die Komplexität des Schleifenrumpfes mit n^2 multiplizieren müssen, um die gesamte Komplexität im schlechtesten Fall zu berechnen. Der Aufruf der Methode `size` außerhalb der Schleife hat einen konstanten Aufwand, und kann daher im Vergleich zum Aufwand der Schleife vernachlässigt werden. Ebenso fallen die Vergleiche der Abbruchbedingung und das Hochzählen der `int`-Zähler für die O-Notation nicht ins Gewicht, da diese jeweils nur einen konstanten Aufwand benötigen.

Zur weiteren Bestimmung der Komplexität des Schleifenrumpf betrachten zuerst den Fall, wenn `a` auf ein Objekt der Klasse `LinkedList` zeigt. Der Vergleich im Konditional hat gemäß den gegeben Formeln den Aufwand $2n+1 \in O\big(n\big)$. Der Rumpf des Konditionals hat den Aufwand $n+(n+n)+n \in O\big(n\big)$. Damit ergibt sich für den Schleifenrumpf eine lineare Komplexität, denn die Addition von Polynomen mit maximal auftretenden Grad k liegt immer noch in $O\big(n^k\big)$. Insgesamt erhalten wir in diesem Fall somit eine kubische Komplexität, also $O\big(n^3\big)$.

Falls `a` auf ein Objekt der Klasse `ArrayList` zeigt, sind sowohl die Prüfung des Konditionals als auch der Rumpf des Konditionals von konstanter Komplexität. Insgesamt erhalten wir dann also eine bessere quadratische Komplexität, als $O\big(n^2\big)$.

Hinweis: *Zur Lösung der Aufgabe reicht natürlich die einfache Angabe der Komplexität für die beiden geforderten Fälle. Die ausführlichen Erklärungen dieser Musterlösung sollen lediglich dem Verständnis dienen.*

b) Informieren Sie sich über die Klasse `ListIterator` und analysieren Sie die Laufzeit des folgenden alternativen Codes, der den gleichen Sortieralgorithmus implementiert. Vergleichen Sie Ihr Ergebnis mit dem für das erste Programm.

```
public static <A extends Comparable<A>> void sortIt(List<A> a) {
    ListIterator<A> outerIt = a.listIterator();
    while (outerIt.hasNext()) {
        outerIt.next();
    }
    while (outerIt.hasPrevious()) {
        outerIt.previous();
        ListIterator<A> innerIt = a.listIterator();
        while (innerIt.nextIndex() < outerIt.nextIndex()) {
            A inner = innerIt.next();
            A innerNext = innerIt.next();
            if (inner.compareTo(innerNext) > 0) {
                innerIt.set(inner);
                innerIt.previous();
                innerIt.previous();
                innerIt.next();
                innerIt.set(innerNext);
            }
            innerIt.previous();
        }
    }
}
```

Der Aufwand aller hier benutzten Methoden der Klasse `ListIterator` ist konstant.

LÖSUNG

In der ersten Schleife wird ein Zeiger auf das Ende der Liste ermittelt. Da nach Angabe alle benutzten Methoden einen konstanten Aufwand haben, ergibt sich durch diese erste Schleife mindestens ein linearer Aufwand, da die Schleife ja einmal für jedes Element der Eingabeliste durchlaufen wird.

Die zweite Schleife wird ebenfalls für jedes Element der Eingabeliste durchlaufen. Da der Schleifenrumpf eine weitere Schleife enthält, welche im schlimmsten Fall erneut für jedes Element der Schleife durchlaufen wird, erhalten wir insgesamt eine quadratische Komplexität, da der innere Schleifenrumpf nur eine konstante Komplexität aufweist. Durch die Verwendung des Iterators ist es jedoch unerheblich, ob `a` auf ein Objekt der Klasse `LinkedList` oder `ArrayList` zeigt. Die Methoden des Iterators benötigen in beiden Fällen nur konstante Laufzeit.

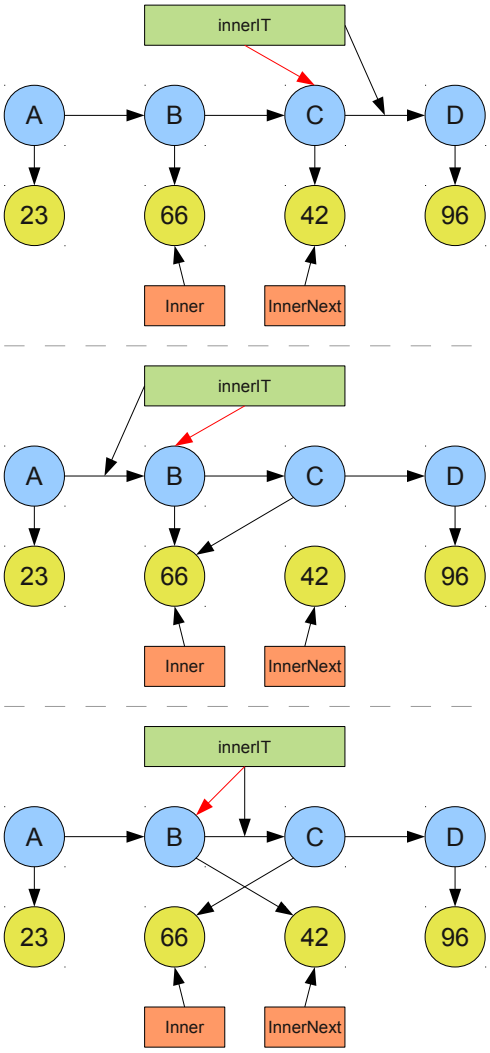
Da $n + n^2 \in O(n^2)$, haben wir nun insgesamt eine quadratische Komplexität erreicht, völlig unabhängig von der verwendeten Implementierung der Liste.

c) i) Welcher Sortieralgorithmus wurde in den beiden vorangegangenen Teilaufgaben implementiert?

LÖSUNG
Es handelt sich um den in der Vorlesung vorgestellten *Bubble-Sort* Algorithmus.

ii) Im Code aus Teilaufgabe b) wird in der inneren Schleife zwei Mal hintereinander die Methode `previous` aufgerufen. Erklären Sie, warum dies notwendig und richtig ist!

LÖSUNG
Gemäß des Bubble-Sort Algorithmus werden im Rumpf des Konditionals zwei benachbarte Elemente der Liste vertauscht.
Zur Verdeutlichung stellen wir diese Vertauschung grafisch dar. Der schwarze Zeiger von `innerIT` zeigt im Bild zwischen die beiden Elementen, welche mit den Methoden `previous` und `next` zurückgegeben würden; der rote Zeiger von `innerIT` zeigt auf das Element, welches durch die Methode `set` bearbeitet wird (gemäß der Beschreibung der Klasse `ListIterator` ist dies immer das letzte Element, welches durch `previous` oder `next` zurückgegeben wurde).
Das erste Bild zeigt die Situation direkt am Anfang des Konditionalrumpf, also direkt vor dem ersten `set`; das zweite Bild zeigt die Situation nach dem zweiten `previous`; und das dritte Bild zeigt die Situation nach dem letzten `previous` am Ende des Schleifenrumpfes.
Das doppelte `previous` ist also deshalb notwendig, da ein Alternieren zwischen `next` und `previous` immer das gleiche Element zurückliefert, wie in der Beschreibung der Klasse `ListIterator` angegeben. Um mit `set` das vorherige Element zu verändern, muss nach einem `next` also immer zweimal `previous` aufgerufen werden.



Dieses Verhalten von `ListIterator`, bei einem Wechsel zwischen `next` und `previous` das gleiche Objekt zurückzugeben ist durchaus sinnvoll und läßt sich auch Ausnutzen, um ein Aufruf der Methode `previous` einzusparen:

```
public static <A extends Comparable<A>> void mYsortIt(List<A> a) {
  ListIterator<A> outerIt = a.listIterator();
  while (outerIt.hasNext()) {
      outerIt.next();
  }
  while (outerIt.hasPrevious()) {
    outerIt.previous();
    ListIterator<A> innerIt = a.listIterator();
    while (innerIt.nextIndex() < outerIt.nextIndex()) {
      A inner = innerIt.next();
      // Wir wollen das nächste Element anschauen,
      // ohne den Iterator zu verändern;
      // deshalb folgt auf das next() ein previous().
      A innerNext = innerIt.next();
      innerIt.previous();
      if (inner.compareTo(innerNext) > 0) {
        // Jetzt können wir immer noch das nächste Element verändern
        innerIt.set(inner);
        innerIt.previous();
        innerIt.set(innerNext);
        // Rückgängig machen des previous() ist nicht notwendig,
        // spart aber einen unnötigen Schleifendurchlauf.
        innerIt.next();
      }
    }
  }
}
```

Achtung: Obwohl diese Verbesserung in der innersten Schleife stattfindet und damit quadratisch in die Laufzeit eingeht, bleibt die Komplexität der Methode unverändert quadratisch. Letztendlich verbessert sich dadurch lediglich eine Konstante in der Formel für die Laufzeit, welche aber von O-Notation ohnehin verschluckt wird.

Aufgabe 14-3 *Komplexität II* (0 Punkte) (Abgabeformat: `.txt` oder `.pdf`)
Gegeben sei folgende Prozedur, welche ein Array von positiven Zahlen auf den Wertebereich von 0 bis 1 normiert, indem sie jeden Eintrag des Arrays durch den maximalen im Array vorkommenden Wert dividiert.

```
/**
 * Vorbedingung:
 * Das Array a enthält nur Zahlen, die echt größer als Null sind.
 */
public static double[] f(double[] a) {
    double[] b = new double[a.length];
    for (int i = 0; i < a.length; i++) {
        // Berechnung der größten Zahl im Array a:
        double max = a[i];
        for (int j = 0; j < a.length; j++) {
            if (a[j] > max) {
                 max = a[j];
            }
        }
        // Normierung:
        b[i] = a[i] / max;
    }
    return b;
}
```

a) Welche Zeitkomplexität bezogen auf die Länge des Arrays `a` hat die Prozedur `f` (in O-Notation) im schlechtesten Fall?

Für die Laufzeitbestimmung brauchen Sie nur Vergleiche zwischen `double`-Werten zu betrachten, in diesem Programm also nur den Vergleich `a[j] > max`.

LÖSUNG
Für ein Eingabearray der Größe n beträgt die Zeitkomplexität im schlechtesten Fall $O(n^2)$.
(Es gibt nur einen Vergleich, den wir laut Aufgabenstellung berücksichtigen müssen. Dieser Vergleich befindet sich innerhalb einer Schleife, welche sich ebenfalls in einer Schleife befindet. Beide Schleifen werden jeweils n-mal durchlaufen.)

b) Verbessern Sie die Methode so, dass sich die Größenordnung der Zeitkomplexität echt verbessert und geben Sie die Komplexität des verbesserten Programms in O-Notation an.

Hinweis: Achten Sie darauf, dass Ihr Programm auch Sonderfälle wie das leere Array weiterhin richtig behandelt.

LÖSUNG
Wenn man zuerst das Maximum bestimmt und erst anschließend normiert, kommt man mit zwei unverschachtelten Schleifen aus. Die Zeitkomplexität beträgt in diesem Fall nu $O(n)$, selbst wenn man alle Operation berücksichtigen würde, und nicht nur die Vergleiche zwischen `double`-Werten.

Im Code muss man dazu nur ein paar Zeilen vertauschen:

```
public static double[] f(double[] a) {
    double[] b = new double[a.length];
    double max = 0;
    // Berechnung der größten Zahl im Array a:
    for (int j = 0; j < a.length; j++) {
        if (a[j] > max) {
              max = a[j];
        }
    }
    // Normierung:
    for (int i = 0; i < a.length; i++) {
        b[i] = a[i] / max;
    }
    return b;
}
```

Da nach den gegebenen Voraussetzungen das Array nur Werte größer Null enthält, ist auch keine Division durch Null zu befürchten. Auch das leere Array wird weiterhin richtig ohne Auslösen einer Exception behandelt, da in diesem Fall keine der beiden Schleifenrümpfe ausgeführt werden.

Abgabe: Sie können ihre Lösungen bis Montag, den 07.02.2011, 14 Uhr über UniWorX abgeben. Java Dateien, welche nicht kompilieren, werden nicht beachtet!