Prof. Martin Hofmann, PhD
Dr. Steffen Jost

Ludwig-Maximilians-Universität München
Institut für Informatik
15. Januar 2018

Lösungsvorschlag zur 11. Übung zur Vorlesung
Einführung in die Programmierung

A11-1 ***Generics I*** Gegeben sind Klassen `A`, `B`, `C`, `D` und die folgende Signatur:
`public static A foo(A a, B b, C c, D d)` Ändern Sie diese schrittweise ab, so dass...

a) ...die Methode `foo` anstatt `A` als erstes Argument und Rückgabetyp einen beliebigen Typen akzeptiert, welcher das Interface `G` implementiert.

LÖSUNGSVORSCHLAG:

Wir benötigen eine Typvariable, um dies auszudrücken:
```
public static <T extends G> T foo(T a, B b, C c, D d)
```

b) ...`foo` als 2. Argument eine `ArrayList` akzeptiert, deren Elemente Erben von `B` sind.

LÖSUNGSVORSCHLAG:

```
public static <T extends G> T foo(T a, ArrayList<? extends B> b, C c, D d)
```
Warum reicht hier `ArrayList<B>` nicht aus?

Antwort: Sei `BB` eine Erbe von `B`. Eine Liste des Typs `ArrayList<B>` kann aufgrund von Subtyping auch Objekte von `BB` enthalten. Wenn wir an einer anderen Stelle eine Liste von `BB`-Objekten hätten, welche explizit keine `B`-Objekte enthalten kann, also eine Liste des Typs `ArrayList<BB>`, dann dürften wir diese nicht als zweites Argument an `foo` übergeben.

c) ...`foo` als 3. Argument alle Objekte eines Typen akzeptiert, welcher das parametrisches Interface `H<>` entweder direkt oder durch einen Vorfahren implementiert.

LÖSUNGSVORSCHLAG:

```
public static <T extends G, S extends H<? super S>>
     T foo(T a, ArrayList<? extends B> b, S c, D d)
```
siehe dazu auch Folie 13.42.

d) ...die Methode `foo` als 4. Argument nicht nur Objekte des Typs `D` akzeptiert, sondern auch alle Erben von `D`.

LÖSUNGSVORSCHLAG:

Hier ist nichts mehr zu tun, denn das ist ja immer erlaubt!

A11-2 ***Generics II*** Auf Vorlesungsfolie 13.44 ist eine Implementierung der binären Suche in einem sortiertem Array abgedruckt. Die gezeigte Methode hat folgende Signatur:

```
public static boolean sucheVonBis(Comparable[] l, Object w, int i, int j)
```

a) Auf der folgenden Folie wird dieser Typ zurecht bemängelt. Erklären Sie warum! Geben Sie ein Verwendungsbeispiel an, welches der Compiler akzeptiert, aber welches immer einen Laufzeitfehler liefert!

LÖSUNGSVORSCHLAG:

Ein einfaches Gegenbeispiel wäre:

```
Integer[] hay = {1,2,3,4};
String needle = "Badaboom!";
sucheVonBis2(hay,needle,0,3);
```

Dies wird vom Compiler klaglos akzeptiert, da `Integer` das Interface `Comparable` implementiert und weil `String` ja ein Erbe von `Object` ist. Es gibt jedoch einen Laufzeitfehler, da wir `Integer` nicht direkt mit `String` vergleichen können.

b) Geben Sie eine vernünftige, generische Signatur an! Überprüfen Sie auch, dass der Compiler ihr vorheriges Gegenbeispiel damit nicht mehr akzeptiert.

LÖSUNGSVORSCHLAG:

```
public static <T extends Comparable<? super T>>
    boolean sucheVonBis2(T[] l, T w, int i, int j)
```

Jetzt wird erzwungen, dass der Typ der Elemente des Arrays mit dem Typ des zu suchenden Elements übereinstimmen, und dass dieser Typ die Schnittstelle `Comparable` implementiert!

Eine leichter zu findende Lösung wäre auch:

```
public static <T extends Comparable<T>>
    boolean sucheVonBis2(T[] l, T w, int i, int j)
```

Diese Lösung hat den Nachteil, das `T` das Interface `Comparable` selbst implementieren muss und nicht einfach von einer Oberklasse erben darf.

Vorsicht: Die Fehler-Hilfe mancher veralteter IDE akzeptiert bei der einfacheren Lösung immer noch das weiter oben gezeigte Gegenbeispiel, allerdings erkennt der Java-Compiler den Fehler dann während dem kompilieren.

A11-3 ***Fakultätsfunktion*** Ein beliebtes einfaches Beispiel für Rekursion ist die Fakultät:

$$x! = \prod_{i=1}^{x} i = 1 \cdot 2 \cdot 3 \cdot \dots \cdot (x-1) \cdot x$$

Gesucht ist eine Implementation für Signatur: `public static long fakultaet(int arg)`

a) Die mathematische Produktschreibweise könnten wir natürlich sofort als simple `for`-Schleife hinschreiben. Tun Sie dies!

LÖSUNGSVORSCHLAG:

```
public static long fakultaetI(int arg) {
    long result = 1;
    for (int i = 1; i <= arg; i++)  result *= i;
    return result;
}
```

b) Die Fakultät läßt sich natürlich auch rekursiv berechnen:

$$x! = \begin{cases} 1, & x = 0 \\ x \cdot (x-1)!, & x > 0 \end{cases}$$

Implementieren Sie die Fakultätsfunktion nun ohne jegliche iterative Schleifen, sondern nur unter Verwendung von Rekursion.

LÖSUNGSVORSCHLAG:

Die direkte Umsetzung der rekursiven Vorschrift:

```
public static long fakultaetR(int arg) {
    if (arg <= 1) return  1;
    else return arg * fakultaetR(arg-1);
}
```

Wer die Vorlesung "Programmierung & Modellierung" bereits gehört hat, hat auch hoffentlich gleich eine endrekursive Version hingeschrieben! ;)

Bemerkung: Ziel dieser Aufgabe zur Fakultätsfunktion ist das schnelle Umsetzen einer iterativen und einer rekursiven Vorschrift in korrekten Code. Wer sich dafür interessiert, wie man die Fakultätsfunktion effizient implementieren kann, findet im Internet zahlreiche Artikel zu diesem Thema. Allerdings möchten wir darauf hinweisen, dass dies hier für den Zahlenbereich des Typs `long` wohl eher unerheblich ist.

A11-4 ***Laufzeit I*** Welche Laufzeitkomplexität hat die folgende Methode:

```
public static int count(int[] a, int c) {
  int count = 0;
  for (int i=0; i < a.length; i++) { if (a[i] == c) count++; }
  return count;
}
```

LÖSUNGSVORSCHLAG:

Die Laufzeitkomplexität ist $O(n)$:

Es bezeichne n die Länge des Eingabe-Arrays `a`.

Die Initialisierung der Variablen `count` und `i` so wie die `return`-Anweisung haben jeweils eine von der Eingabe unabhängige, konstante Laufzeit $O(1)$.

Die Schleife wird genau n-mal durchlaufen, d.h. Laufzeit ist $O(n \cdot \langle\text{Aufwand Rumpf}\rangle)$.

Der Durchlauf des Schleifenrumpfes hat wieder eine konstante Laufzeit, also $O(1)$: Der Integer-Vergleich hat konstanten Aufwand. Das Abwickeln der If-Anweisung selbst hat prinzipiell konstanten Aufwand, hängt jedoch vom Aufwand der Bedingung und dem Aufwand der bedingten Anweisungen ab: Der Integer-Vergleich in der If-Bedingung hat ebenfalls konstanten Aufwand wichtig ist hier jedoch, dass auch der Zugriff auf die Integer-Werte nur einen konstanten Aufwand hat: bei Arrays gilt aber gerade, dass `a[i]` konstante Laufzeitkomplexität hat. Im schlechtesten Fall finden noch zwei Integer-Erhöhungen statt: `count++` und `i++`, welche jedoch wieder nur konstante Laufzeit benötigen.

Wir haben also: $O(1+1+n\cdot(1+1+1+1+1+1)+1) = O(n)$. Ob man 1 oder 42 schreibt ist bei O-Notation ja unerheblich, diese $1+1+\ldots$-Rechnung hier soll lediglich verdeutlichen, an wie viele Dinge wir hier gedacht haben: an jede Anweisung und jeden darin enthaltenen Teil-Ausdruck.

H11-1 ***Laufzeit II*** (0 Punkte; Abgabe: `H11-1.txt` oder `H11-1.pdf`)
Angenommen ein Algorithmus benötigt 5 Sekunden um eine Eingabe mit 1000 Elementen zu verarbeiten. Berechnen Sie, wie lange der Algorithmus für andere Eingabegrößen ungefähr benötigt, unter Annahme folgender Laufzeitkomplexitäten:

	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$
1000	$5s$	$5s$	$5s$	$5s$	$5s$
2000	$10s$	$11s$	$20s$	$40s$	$1.70 \cdot 10^{304}y$
3000	$15s$	$17s$	$45s$	$135s$	$1.82 \cdot 10^{605}y$
10000	$50s$	$67s$	$8.3m$	$83.3m$	$2.95 \cdot 10^{2712}y$

Beispiel: Wegen $\frac{3000^2}{1000^2} = 9$ benötigt der Algorithmus bei einer Eingabe der Größe 3000 und einer Laufzeitkomplexität von $O(n^2)$ also 9-mal so viel Zeit wie bei einer Eingabe der Größe 1000.

LÖSUNGSVORSCHLAG:

Die exakten Zahlen sind hier nicht wichtig; die Botschaft sollte sein, dass alles über einer quadratischen Laufzeit letztendlich unpraktikabel ist.

H11-2 ***Generics III*** (3 Punkte; BiSeCo.java)
Fortsetzung von Aufgabe A11-2, Teilaufgabe c):

Lesen Sie die Dokumentation des Interfaces `Comparator` aus der Standardbibliothek und implementieren Sie eine Variante der binären Suche mit folgender Signatur:

```
public static <T> boolean sucheVonBisC(Comparator<T> c, T[] l, T w, int i, int j)
```

LÖSUNGSVORSCHLAG:

```
public static <T> boolean sucheVonBisC(
      Comparator<T> c, T[]l, T w, int i, int j) {
  if (i > j) return false;
  if (i == j) return 0 == c.compare(l[i],w);
  int m = (i + j) / 2;
  int comp = c.compare(l[m],w);
  if (comp == 0) return true;
  if (comp < 0) // wm < w
    return sucheVonBisC(c, l, w, m + 1, j);
  else
    return sucheVonBisC(c, l, w, i, m - 1);
}
```

Das Interface `Comparable` legt für einen Typ einen Vergleich fest, welcher für das gesamte Programm gilt.

Im Unterschied dazu erlaubt das funktionale Interface `Comparator` ad hoc eine beliebige Methode zum Vergleich zu übergeben, d.h. bei jedem Aufruf könnte ein anderer Vergleich für denselben Typ verwendet werden.

```
String[] test1 = new String[] {"A","B","C","D","E","F"};
Comparator<String> cs = (a,b) -> {return a.compareTo(b);}; // Gewöhnlich
System.out.println("test1, Z = " + sucheVonBisC(cs, test1,"Z",0,test1.length-1));

Integer[] test2 = new Integer[] {9,8,7,6,5,4,3,2,1};
Comparator<Integer> ci = (a,b) -> {return b.compareTo(a);}; // Umgekehrt
System.out.println("test2, 7 = "+ sucheVonBisC(ci, test2,7,0,test2.length-1));
```

Die Lambda-Notation kann auch direkt ohne lokale Hilfsvariable verwendet werden:

```
System.out.println("test2, 7 = "
  + sucheVonBisC((a,b) -> {return b.compareTo(a);}, test2,7,0,test2.length-1));
```

H11-3 ***Duplikate*** (6 Punkte; NoDups.java)
Implementieren Sie folgende Methode:

`public static <T> ArrayList<T> noDups(Comparator<T> cmp, ArrayList<T> l)`

welche als Rückgabewert eine Liste ohne Duplikate liefert, d.h. die Rückgabeliste enthält jeden Wert der Argumentliste genau einmal, wobei Gleichheit gemäß dem übergebenen `Comparator` entschieden wird. Geben Sie weiterhin die Laufzeit Ihrer Implementierung in O-Notation an und begründen Sie diese!

Die Punkte dieser Aufgabe werden für eine korrekte Implementierung und eine korrekt begründete Angabe der Laufzeit vergeben.

Jeweils 1 Bonuspunkt wird darauf vergeben, falls Ihre Implementierung eine Laufzeit besser als $O(n^2)$ hat, und wenn die Sortierung der ursprünglichen Argumentlist beibehalten wird (d.h. für die Eingabe [8, 2, 8, 3, 2, 8, 1, 2] wird [8,2,3,1] und nicht etwa [1,2,3,8] zurückgegeben).

LÖSUNGSVORSCHLAG:

Anstatt hier die gefragte konkrete Implementierung abzudrucken erklären wir die generelle Vorgehensweise des Algorithmus, welche so nicht so leicht aus dem langwierigem Code abzulesen wäre.

Lösung 1: Die einfachste Lösung, welche mit Laufzeitkomplexität $O(n \cdot \log n)$ die Reihenfolge der Elemente enthält, lässt sich mit Hilfe von „Mengen"-Datenstrukturen realisieren, welche effiziente Abfragen ermöglichen, ob ein Element in einer Menge vorhanden ist oder nicht. Diese Datenstrukturen werden wir aber erst später in der Vorlesung behandeln, wenn wir deren internen Aufbau verstehen können. Deshalb müssen wir hier jetzt eine geringfügig aufwändigere Lösung implementieren.

Lösung 2: Doch zuerst betrachten wir einfache Lösungsmöglichkeiten. Wenn man die Laufzeit ignoriert, dann ist die Lösung ganz einfach: Mit einer Schleife iteriert man über die Liste und entfernt alle Elemente (in-place oder in einer Kopie), welche man mit einer weiteren Schleife bereits an einem kleineren Index finden kann. Die doppelt verschachtelte Schleife ergibt dann eine Laufzeitkomplexität von $O(n^2)$ (für das i-te Element iteriert man über alle Elemente mit Index kleiner als i.)

Lösung 3: Eine weitere relativ einfache Lösung ist möglich, wenn man die Reihenfolge nicht erhalten muss: Zuerst sortiert man die Liste, was nach den Methoden der Vorlesung in $O(n \log n)$ möglich ist. Danach iteriert man noch einmal über die sortierte Liste in $O(n)$, wobei man sich jeweils das vorangegangene Element merkt $O(1)$. Da nach der Sortierung gleiche Elemente benachbart sein müssen, kann man doppelte Elemente nun leicht entfernen, wenn das aktuelle Element mit dem zuletzt gemerkten übereinstimmt. Hier müssen wir jedoch noch aufpassen: Entfernen eines Elementes in einer `ArrayList` hat einen linearen Aufwand. Dies ist hier nicht akzeptabel, da dies innerhalb einer Schleife mit linearem Aufwand geschieht, was eine quadratische Gesamtkomplexität ergeben würde. Stattdessen konstruieren wir einfach eine neue Liste/Array, den das Einfügen eines Elementes hat dann nur konstante Komplexität ($O(1)$). Die gesamte Laufzeitkomplexität ist dann $O(n \log n) + O(n \cdot 1) = O(n \log n)$.

Lösung 4: Betrachten wir nun die Lösung, welche die Reihenfolge erhält und trotzdem die beste-mögliche Laufzeitkomplexität von $O(n \log n)$ hat. Zuerst erstellen wir eine Kopie der Eingabe $O(n)$, welche wir sortieren und alle doppelten entfernen, was nach den vorangegangenen Lösungen in $O(n \log n)$ möglich ist.

Anschließend iterieren wir nun über die ursprüngliche Eingabe: Für jedes Element führen wir eine binäre Suche in der sortierten Liste durch. Wenn wir das Element mit der binären Suche finden, dann schreiben wir es in die Ergebnisliste und löschen es aus der sortierten Liste, ansonsten betrachten wir das nächste Element der ursprünglichen Eingabe.

Abgesehen von dem Löschen können wir dies in $O(n \log n)$ durchführen: für jedes der n-vielen Elemente eine binäre suche mit $O(\log n)$. Damit hätten wir eine Laufzeit von $O(n + (n \log n) + (n \log n)) = O(n \log n)$ wie benötigt.

Damit das alles klappt, müssen wir das Herauslöschen von Elementen in der sortierten Liste in logarithmischer oder besser realisieren. Dies könnte man ganz einfach mit verketteten Listen erreichen, doch damit funktioniert die binäre Suche nicht mehr. Aber wir können beim anfänglichen Kopieren der Liste alle Elemente mit einem `boolean` Flag verpacken. Das Flag setzen wir entsprechend auf `true`, wenn das Element zum ersten Mal an die Ausgabeliste angehängt wird, so dass wir es bei wiederholtem Auffinden ignorieren.

Das Verpacken mit einem `boolean`-Flag erreichen wir ganz leicht mit einer Klasse wie etwa diese hier, welche uns auch gleich noch eine komfortable Möglichkeit mitliefert, die übergebene Vergleichsoperation direkt auf die verpackten Elemente anzuwenden:

```
import java.util.Comparator;

public class Deletable<A> {
  private A a;
  private boolean deleted;

  // Gewöhnlicher Konstruktor, Getter und Setter...
  public Deletable(A a) { this.a = a; this.deleted = false; }

  public A getA() { return a; }

  public boolean isDeleted() { return deleted; }

  public void delete() { this.deleted = true; }

  public static <A> Comparator<Deletable<A>> liftComp(Comparator<A> cmp) {   // optional
    Comparator<Deletable<A>> res = (dx,dy) -> {
      A x = dx.getA();
      A y = dy.getA();
      return cmp.compare(x,y);
    };
    return res;
  }
}
```

Lösung 5: Man kopiert die Eingabeliste in eine Liste mit expliziten Positionen unter Verwendung einer einfachen Hilfsklasse wie die unten abgebildete `ElemPos<A>`. Dies hat einen linearen Aufwand $O(n)$. Danach sortiert man die Liste nach den Elementen und entfernt alle benachbarten doppelten wie in Lösung 3 mit $O(n \log n)$. Anschließend sortiert man die Liste erneut nach den Indizes $O(n \log n)$. Zum Schluss entfernt man die Verpackung um aus `ArrayList<ElemPos<A>>` wieder `ArrayList<A>` zu erhalten mit $O(n)$. Wir haben also $O(2n + 2(n \log n)) = O(n \log n)$.

```java
import java.util.Comparator;

public class ElemPos<A> {
  private final A a;
  private final int index;

  // Gewöhnlicher Konstruktor, Getter und Setter...
  public ElemPos(A a, int i) { this.a = a; this.index = i; }
  public A getA() { return a; }
  public int getIndex() { return index; }

  public static <A> Comparator<ElemPos<A>> liftComp(Comparator<A> cmp) {
    Comparator<ElemPos<A>> res = (epx,epy) -> {
      A x = epx.getA();
      A y = epy.getA();
      int ret = cmp.compare(x,y);
      if (ret != 0) { return res; }
      else {
        Comparator<ElemPos<A>> vergleich = ElemPos.indexComp();
        return vergleich.compare(epx,epy);
      }
    };
    return res;
  }

  public static <A> Comparator<ElemPos<A>> indexComp() {
    Comparator<ElemPos<A>> res = (epx,epy) -> {
      Integer ix = epx.getIndex();
      Integer iy = epx.getIndex();
      return ix.compareTo(iy);
    };
    return res;
  }
}
```

Bemerkung: Lösungen 4 & 5 mögen einem auf dem ersten Blick unnötig aufwändig erscheinen. Für kurze Eingabelisten sind diese auch sicherlich langsamer als Lösung 2. Ein Blick auf die Lösung von H11-1 zeigt jedoch wie schnell Lösung 2 deutlich langsamer sein wird, wenn die Eingabeliste nur etwas länger ist und nicht *extrem viele*[1] doppelte Elemente enthält.

[1] Lösung 2 hat effektiv eine lineare Laufzeit wenn der Wertebereich klein ist: Bei einer Liste mit einer Millionen Elementen von natürlichen Zahlen von 1 bis 10 hat die innere Schleife ja höchstens 10 Vergleiche zu machen, und damit effektiv eine konstante Laufzeit.

Abgabe: Lösungen zu den Hausaufgaben können bis Sonntag, den 21.1.18, mit UniWorX nur als `.zip` abgegeben werden. Aufgrund des Klausurbonus müssen die Hausaufgaben von Ihnen alleine gelöst werden. Abschreiben bei den Hausaufgaben gilt als Betrug und kann zum Ausschluss von der Klausur zur Vorlesung führen. Bitte beachten Sie auch die Hinweise zum Übungsbetrieb auf der Vorlesungshomepage (`www.tcs.ifi.lmu.de/lehre/ws-2017-18/eip/`).