Einführung in die Programmierung mit Java

Teil 13: Algorithmik: Sortieren und Suchen

Martin Hofmann Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

09. Januar 2018

Teil 13: Algorithmik: Sortieren und Suchen

Sortieren durch Auswählen

Wir wollen ein Array `a` von `int`-Zahlen der Größe nach sortieren: Z.B. $11, 9, 17, 5, 12$ soll $5, 9, 11, 12, 17$ werden.
Wir suchen das kleinste Element, hier a[3] $= 5$, und schaffen es nach vorne durch Vertauschen mit dem ersten Element:

$$\begin{array}{ccccc} \boxed{11} & 9 & 17 & \boxed{5} & 12 \\ \boxed{5} & 9 & 17 & \boxed{11} & 12 \end{array}$$

Dann suchen wir das kleinste Element von `a[1..4]`. Es ist schon an der richtigen Stelle.
Dann das kleinste Element von `a[2..4]`. Es ist `a[3]=11`. Vertauschen mit `a[2]` führt auf

$$\begin{array}{ccccc} 5 & 9 & \boxed{11} & \boxed{17} & 12 \end{array}$$

Das kleinste Element von `a[3..4]` wird noch mit `a[3]` vertauscht und wir sind fertig.

Dasselbe in Java

Zunächst das Testprogramm

```
import ...;
public class SelSortTest
{  public static void main(String[] args)
   {  int[] a = ArrayUtil.randomIntArray(20, 100);

      ArrayUtil.print(a);
      SelSort.sort(a);
      ArrayUtil.print(a);
   }
}
```

Sortieren durch Auswählen in Java

```
public class SelSort
{  /**
      Finds the smallest element in an array range.
      @param a the array to search
      @param from the first position in a to compare
      @return the position of the smallest element in the
      range a[from]...a[a.length - 1]
   */
   public static int minimumPosition(int[] a, int from)
   {  int minPos = from;
      for (int i = from + 1; i < a.length; i++)
         if (a[i] < a[minPos]) minPos = i;
      return minPos;
   }
```

```
    /**
       Sorts an array.
       @param a the array to sort
    */
    public static void sort(int[] a)
    {  for (int n = 0; n < a.length - 1; n++)
       {  int minPos = minimumPosition(a, n);
          if (minPos != n)
             ArrayUtil.swap(a, minPos, n);
       }
    }
}
```

Testen

```
mhofmann> java sorting/SelSortTest
52 23 37 65 79 95 21 27 12 12 78 66 66 51 7 39 81 86 95 74
7 12 12 21 23 27 37 39 51 52 65 66 66 74 78 79 81 86 95 95
```

Für längere Arrays die `print` Statements 'rauskommentieren.
Bis Größe 10000 ist die Laufzeit im Millisekundenbereich.
Bei 100000 dauert es drei Sekunden.
Bei 500000 dauert es über eine Minute.
Bei 5000000 dauert es mehrere Stunden.
Wir führen eine genauere empirische Analyse durch:

Stoppuhr

Die Methode `System.currentTimeMillis()` liefert die Anzahl der Millisekunden, die seit 00:00 am 1.1.1970 verstrichen sind (ca. 1 Trillion $> 2^{31}$ daher ist `long` erforderlich.)
Damit können wir eine "Stoppuhr-Klasse" bauen, die die Methoden

```
reset()
start()
stop()
getElapsedTime()
```

bereitstellt (Details siehe Javadoc).

Stoppuhr

```
package sorting;
/**
   A stopwatch accumulates time when it is running. You can
   repeatedly start and stop the stopwatch. You can use a
   stopwatch to measure the running time of a program.
*/

public class StopWatch
{
   private long elapsedTime;
   private long startTime;
   private boolean isRunning;

   /**
      Starts the stopwatch. Time starts accumulating now.
   */
```

```
public void start()
   {  if (isRunning) return;
      isRunning = true;
      startTime = System.currentTimeMillis();
   }

   /**
      Stops the stopwatch. Time stops accumulating and is
      is added to the elapsed time.
   */
   public void stop()
   {  if (!isRunning) return;
      isRunning = false;
      long endTime = System.currentTimeMillis();
      elapsedTime = elapsedTime + endTime - startTime;
   }
```

```
/**
      Returns the total elapsed time.
      @return the total elapsed time
   */
   public long getElapsedTime()
   {  if (isRunning)
      {  long endTime = System.currentTimeMillis();
         elapsedTime = elapsedTime + endTime - startTime;
         startTime = endTime;
      }
      return elapsedTime;
   }

   /**
      Stops the watch and resets the elapsed time to 0.
   */
   public void reset()
   {  elapsedTime = 0;
      isRunning = false;
   }
```

```
/**
      Constructs a stopwatch that is in the stopped state
      and has no time accumulated.
   */
   public StopWatch()
   {  reset();
   }

}
```

Laufzeit von SelSort

```
public class SelSortTime
{  public static void main(String[] args)
   {
      int n = Integer.parseInt(
        JOptionPane.showInputDialog("Enter array size:"));
      int[] a = ArrayUtil.randomIntArray(n, 100);
      StopWatch timer = new StopWatch();
      timer.start();
      SelSort.sort(a);
      timer.stop();
      System.out.println("Elapsed time: "
         + timer.getElapsedTime() + " milliseconds");
   }
}
```

Laufzeitmessung

n	Laufzeit in ms ('03)	Laufzeit in ms ('12)	Laufzeit in ms ('16)
500	7	5	4
1000	14	7	14
2000	54	15	15
3000	93	23	29
$10K$	992	95	61
$20K$	3939	356	146
$50K$	15858	1282	518
$100K$		8176	2756

Analyse der Laufzeit

Als grobes Maß für die Laufzeit wählen wir die Anzahl der Arrayzugriffe.
Die wirkliche Laufzeit ist auf jeden Fall größer als ein festes Vielfaches dieser Zahl.
Wir schätzen die Zahl der Arrayzugriffe von unten ab:
Sei n die Arraygröße.

Abschätzung der Laufzeit

Finden des kleinsten Elements: n Zugriffe.
Finden des 2.kleinsten Elements: $n - 1$ Zugriffe.
Finden des 3.kleinsten Elements: $n - 2$ Zugriffe.
Finden des $n - 1$.kleinsten Elements: 2 Zugriffe.
Macht zusammen $2 + 3 + 4 + 5 + \cdots + n = \frac{n(n+1)}{2} - 1 = \frac{1}{2}n^2 + \frac{1}{2}n - 1$.
Das Vertauschen haben wir gar nicht gerechnet!

Grössenordnung der Laufzeit

Zahl der Arrayzugriffe $\geq \frac{1}{2}n^2 + \frac{1}{2}n - 1$.

Der lineare Term spielt für große n keine Rolle.

Der Faktor 1/2 auch nicht, da die exakte Laufzeit sowieso durch Multiplikation mit einem maschinen- und implementationsabhängigen Wert entsteht.

Nur das quadratische Wachstum interessiert. Wir schreiben $\frac{1}{2}n^2 + \frac{1}{2}n - 1 = O(n^2)$.

Die Laufzeit von Selection Sort ist $O(n^2)$

O-Notation

Zur Bestimmung der O-Notation finde man den am schnellsten wachsenden Term und lasse eventuelle Vorfaktoren weg.

$0,9n^3 + 890n^2 = O(n^3)$
$n^2(n^2 + 4n) = O(n^4)$
$2^n + n^{30000} = O(2^n)$

Für Pedanten

Eigentlich müsste man schreiben

$$\frac{1}{2}n^2 \in O(n^2)$$

denn $O(n^2)$ ist die Klasse der Funktionen von höchstens quadratischem Wachstum.
Das Gleichheitszeichen hat sich aber eingebürgert.
Formale Definition von $O(-)$ gibt es in "Algorithmen und Datenstrukturen".

Sortieren durch Mischen

Gegeben folgendes Array der Größe 10.

$$5, 9, 10, 12, 17, \quad 1, 8, 11, 20, 32$$

Die beiden “Hälften” sind hier bereits sortiert!

Mischen

Wir können das Array sortieren, indem wir jeweils von der ersten oder der zweiten Hälfte ein Element wegnehmen, je nachdem, welches “dran” ist:

$5, 9, 10, 12, 17,$	$\cancel{1}, 8, 11, 20, 32$	1
$\cancel{5}, 9, 10, 12, 17,$	$\cancel{1}, 8, 11, 20, 32$	$1, 5$
$\cancel{5}, 9, 10, 12, 17,$	$\cancel{1}, \cancel{8}, 11, 20, 32$	$1, 5, 8$
$\cancel{5}, \cancel{9}, 10, 12, 17,$	$\cancel{1}, \cancel{8}, 11, 20, 32$	$1, 5, 8, 9$
$\ldots$	$\ldots$	

...und die weggenommenen Elemente in ein anderes Array kopieren.

Sortieren durch Mischen

- Falls die beiden Hälften nicht schon sortiert sind, dann müssen wir sie eben vorher sortieren.
- Wie? Durch Mischen der jeweiligen Hälften (also Viertel).
- Und wenn die nicht schon sortiert sind? Dann werden wiederum die jeweiligen Hälften (also Achtel) gemischt.
- Usw. bis man bei Arrays der Größe Eins angelangt ist, die ja stets sortiert sind.

Sortieren durch Mischen in Java

```
public static void mergeSort(int[] a, int from, int to)
{  if (from >= to) return;
   int mid = (from + to) / 2;
    // sort the first and the second half
   mergeSort(a, from, mid);
   mergeSort(a, mid + 1, to);
   merge(a, from, mid, to);
}
```

Mischen

```
public static void merge(int[] a,
   int from, int mid, int to)
{  int n = to - from + 1;
      // size of the range to be merged

   // merge both halves into a temporary array b
   int[] b = new int[n];

   int i1 = from;
      // next element to consider in the first range
   int i2 = mid + 1;
      // next element to consider in the second range
   int j = 0;
      // next open position in b

   // as long as neither i1 nor i2 past the end, move
   // the smaller element into b
```

```
while (i1 <= mid && i2 <= to)
      {  if (a[i1] < a[i2])
         {  b[j] = a[i1];
            i1++;
         }
         else
         {  b[j] = a[i2];
            i2++;
         }
         j++;
      }

      // copy any remaining entries of the first half
      while (i1 <= mid)
      {  b[j] = a[i1];
         i1++;
         j++;
      }
```

```
// copy any remaining entries of the second half
    while (i2 <= to)
    {  b[j] = a[i2];
       i2++;
       j++;
    }

    // copy back from the temporary array
    for (j = 0; j < n; j++)
       a[from + j] = b[j];
  }
```

Laufzeit von Merge Sort

n	Laufzeit in ms '03	Laufzeit in ms '08	Laufzeit in ms '16
500	7	1	3
3500	17	15	10
$10K$	35	32	4
$20K$	41	43	14
$30K$	56	49	23
$40K$	69	45	27
$50K$	94	62	29
$60K$	109	59	31
$80K$	138	114	29
$5M$	9612	2219	583
$10M$			1117
$100M$			11711

Analytische Bestimmung der Laufzeit

- Sei $T(n)$ die Zahl der Arrayzugriffe von Merge Sort bei Arraygröße n.
- Es gilt:

$$\begin{aligned} T(1) &= 0 \\ T(n) &= 2T(\tfrac{n}{2}) + 5n \end{aligned}$$

  falls $n = 2^k$ (ansonsten stimmt es immer noch "größenordnungsmäßig").
- Die $5n$ kommen vom Mischen: $3n$ fürs eigentliche Mischen, $2n$ fürs Zurückschreiben.
- Lösung der Gleichung:

$$T(n) = T(2^k) = 5 \cdot 2^k + 2T(2^{k-1}) = \\ 5 \cdot 2^k + 2 \cdot 5 \cdot 2^{k-1} + 4T(2^{k-2}) + \cdots + 2^k \cdot T(1)$$

- Wir raten: $T(2^k) = k \cdot 5 \cdot 2^k + 2^k \cdot T(1) = k \cdot 5 \cdot 2^k$.

- Gegenprobe: $k \cdot 5 \cdot 2^k = 2(k-1) \cdot 5 \cdot 2^{k-1} + 5 \cdot 2^k$.
- Also gilt $T(2^k) = 5 \cdot 2^k \cdot k$ oder $T(n) = 5n \log_2 n$.
- Es ist: $5n \log_2(n) = O(n \log(n))$.
- Die Basis lässt man weg, da alle Logarithmen proportional sind.

Die Laufzeit von Merge Sort ist $O(n \log(n))$

Die Schnittstelle Comparable

Wir wollen Such- und Sortieroperationen für beliebige Objekte definieren.
Dazu verwenden wir die vordefinierte Schnittstelle `Comparable`:

```
public interface Comparable {
   int compareTo(Object other);
}
```

Wenn `o:Comparable` und `other:Object` und `o` mit `other` vergleichbar ist, dann sollte gelten

$\texttt{o.compareTo(other)} < 0$, falls `o` kleiner `other`
$\texttt{o.compareTo(other)} = 0$, falls `o` gleich `other`
$\texttt{o.compareTo(other)} > 0$, falls `o` größer als `other`

Beispiele

Die Klasse `String` implementiert automatisch die Schnittstelle `Comparable`.

Die Ordnung ist dabei die **lexikographische** Ordnung.

Der Ausdruck `"AAAaaa".compareTo("Mein Schluesseldienst")` hat einen Wert < 0.

Der Ausdruck `"AAAaaa".compareTo(new Point(2,3))` ist typkorrekt (warum?) führt aber zu einem Laufzeitfehler (Programmabbruch).

Beispiele

Bankkonten nach ihrer Kontonummer sortieren:

```
public class BankkontoAngeordnet extends
           Bankkonto implements Comparable {
    public int compareTo(Object other) {
        return getKontonummer() -
              ((Bankkonto)other).getKontonummer();
    }
}
```

Will man verschiedene Anordnungen, so verwende man das Strategiemuster. Siehe auch `Comparator`.

Anwendung im Beispiel

Will man andere Objekte als Zahlen sortieren, so schreibe man also

```
mergeSort(Comparable[] a, int from, int to){...}
```

und ersetze im Code jeweils `x < y` durch `x.compareTo(y) < 0`.

Typvariablen

In der Java Dokumentation steht:

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

Was bedeutet das?
Es handelt sich um eine **parametrisierte Schnittstelle**.
Die Schnittstelle `Comparable<Integer>` deklariert eine Methode

```
    public int compareTo(Integer o);
```

Die Schnittstelle `Comparable<Student>` deklariert eine Methode

```
    public int compareTo(Student o);
```

Die Schnittstelle `Comparable` ist eine Abkürzung für `Comparable<Object>`.

Anwendung

Versucht man

```
public static <T> void merge(Comparable<T> a[],
  int from, int mid, int to){
    ... if(a[i1].compareTo(a[i2])<0) ...
}
```

so kommt

```
compareTo(T) in java.lang.Comparable<T> cannot be
    applied to (java.lang.Comparable<T>)
          {  if (a[i1].compareTo(a[i2])<0)
```

Das Argument `b` in `a.compareTo(b)` muss vom Typ `T` sein, wenn `a` vom Typ `Comparable<T>` ist.
Beachte: Das vorgestellte `<T>` bezeichnet, dass die Deklaration durch `T` parametrisiert ist. Für jede konkrete Einsetzung von `T` ergibt sich ein anderer Typ. ("*Typschema*")

Lösung: Constraints und Wildcards

Korrekterweise deklariert man

```
public static <T extends Comparable<T>> void
    mergeSort(T[] a, int from, int to){ ... }
```

oder

```
public static <T extends Comparable<T>> void
    mergeSort(ArrayList<T> a, int from, int to){ ... }
```

Letzteres ist besser, da Arrays mit Typvariablen nicht gut harmonieren.

Wildcards

Noch besser ist:

```
<T extends Comparable<? super T>>
    mergeSort(ArrayList<T> a, ...) { ... }
```

`T` muss also ein Subtyp von `Comparable<S>` sein, wobei `S` ein Supertyp von `T` ist.

Das `?` ist eine **Wildcard**, sie steht für irgendeinen Typ, der die Bedingung (formuliert mit `super` oder `extends`) erfüllt:

- `? super T`: ein Supertyp von `T`
- `? extends T`: ein Subtyp von `T`

Generische Klassen

```
class Klassenname <Typvariable,...,Typvariable> { ... }
```

Beispiel:

```
public class Box<T> {
  public T inhalt;

  public T tauscheInhalt(T neu){
    T alt = this.inhalt;
    this.inhalt = neu;
    return alt;
  }
}
```

- Instantiierung generischer Typen durch Angabe in spitze Klammer: z.B. `Box<Rechteck> boxrecht;` oder `Box<Integer> boxint;`
- Erben können generisch sein oder auch nicht.

Generische Methoden

<Typvariable,...,Typvariable> Ergebnistyp Methodenname (Parameter,...,Parameter) { ... }

Beispiel:

```
public static <T> T wahl(boolean b, T x, T y) {
    T result;
    if (b) { result = x; }
    else { result = y; }
    return result;
}
```

- Generische Methoden werden ganz normal aufgerufen, d.h. ohne Erwähnung des Parametertyps:
  `String s = wahl(true, "EiP=einfach", "EiP=schwierig");`
- Vor allem für statische Methoden

Generische Typen

Generische Deklarationen ermöglichen bessere Typprüfung für generischen Code:

- `static <T> T wahl(boolean b, T x, T y)` $\Longrightarrow$

```
String t = wahl(false, "Sonne", "Regen" ); ✓
Object t = wahl(true, "EiP einfach", 666); ↯
```

- `static Object wahl'(boolean b, Object x, Object y)`

```
String t = wahl'(false, "Sonne", "Regen" ); ↯
Object t = wahl'(true, "EiP einfach", 666); ✓
```

Die erste Situation ist fast immer die gewünschte!

Generische Vererbung

- Erben können generisch sein oder auch nicht:

```
public class A<X,Y> {...}
public class B<X,Y> extends A<X,Y> {...}
public class C<Y> extends A<Rechteck,Y> {...}
public class D extends A<Shape,Rechteck> {...}
```

- Aus `S` erbt von `T` folgt nicht, dass `A<S>` ein Subtyp von `A<T>` ist:

```
Box<Point> pBox = new Box<Point>(new Point(1,12));
Box<Object> oBox;
oBox = pBox;                              // Typfehler!
oBox.tausche(new GraphicsWindow());
Point s = pBox.inhalt;                    // ↯
```

Anwendung

Versucht man

```
public static <T> void merge(Comparable<T> a[],
  int from, int mid, int to){
    ... if(a[i1].compareTo(a[i2])<0) ...
}
```

so kommt

```
compareTo(T) in java.lang.Comparable<T> cannot be
    applied to (java.lang.Comparable<T>)
          {  if (a[i1].compareTo(a[i2])<0)
```

Das Argument `b` in `a.compareTo(b)` muss vom Typ `T` sein, wenn `a` vom Typ `Comparable<T>` ist.

Beachte: Das vorgestellte `<T>` bezeichnet, dass die Deklaration durch `T` parametrisiert ist. Für jede konkrete Einsetzung von `T` ergibt sich ein anderer Typ. ("*Typschema*")

Lösung: Constraints und Wildcards

Korrekterweise deklariert man

```
public static <T extends Comparable<T>> void
    mergeSort(T[] a, int from, int to){ ... }
```

oder

```
public static <T extends Comparable<T>> void
    mergeSort(ArrayList<T> a, int from, int to){ ... }
```

Letzteres ist besser, da Arrays mit Typvariablen nicht gut harmonieren.

Wildcards

Noch besser ist:

```
<T extends Comparable<? super T>>
    mergeSort(ArrayList<T> a, ...) { ... }
```

`T` muss also ein Subtyp von `Comparable<S>` sein,
wobei `S` ein Supertyp von `T` ist.

Das `?` ist eine **Wildcard**, sie steht für irgendeinen Typ, der die Bedingung (formuliert mit `super` oder `extends`) erfüllt:

- `? super T`: ein Supertyp von `T`
- `? extends T`: ein Subtyp von `T`

Wildcard Beispiel

Angenommen wir haben:

- Bankkonto implements Comparable<Bankkonto>
- Sparkonto extends Bankkonto

Damit gilt auch:

- Sparkonto extends Comparable<Bankkonto>
- Sparkonto extends Comparable<? super Sparkonto>

Es gilt aber gerade nicht:

- Sparkonto extends Comparable<Sparkonto>

Denn Sparkonto implementiert dieses Interface ja nicht selbst! D.h.
Comparable<Sparkonto> z = new Sparkonto(); ist nicht erlaubt!

Deshalb Wildcards verwenden, wie z.B.

- <T extends Comparable<? super T>>

Java-Arrays sind nicht Typ-sicher

Folgender Code wird vom Compiler (und IDEs) problemlos akzeptiert, liefert aber einen Laufzeitfehler:

```
Integer[] iary = new Integer[] {0,1,2,3,4};
Object[]  objs = iary;  // Fälschlicherweise erlaubt
String s = "Buwharhar!";
objs[2]  = s;           // java.lang.ArrayStoreException
```

Das Problem ist historisch bedingt:
Vor Java 1.5 gab es keine Generics. Zuweisung wie in der zweiten Zeile waren bis dahin einen mangelhafter, aber weit verbreiteter und auch durchaus praktikabler Ersatz:

```
void sort(Object[] ary) // Sortiere beliebiges Array
```

Daher erschien es damals nicht sinnvoll, solchen Code plötzlich zu verbieten.

Binäre Suche

Um in einer *bereits sortierten* Array zu suchen, bietet sich die effiziente **binäre Suche** an ($O(\log n)$):

```
public static boolean sucheVonBis(
         Comparable[] l, Object w, int i, int j) {
    if (i > j) return false;
    if (i == j) return 0 == l[i].compareTo(w);
    int m = (i+j) / 2;
    Comparable wm = l[m];
    int comp = wm.compareTo(w);
    if (comp == 0) return true;
    if (comp < 0) // wm < w
        return sucheVonBis(l,w,m+1,j);
    else
        return sucheVonBis(l,w,i,m-1);
}
```

Verbesserte Typisierung

Die gezeigte Signatur ist problematisch:

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

1. Warum ist diese Signatur denn problematisch?
2. Man gebe eine verbesserte Typisierung mit Typvariablen und Wildcards an!

Hinweis:
Diese Fragen werden in Übungsaufgabe A11-1 behandelt.

Rekursion

Den Aufruf einer Methode in ihrem eigenen Rumpf bezeichnet man als **Rekursion**.

Erinnerung: ▸ Mischen ▸ BinäreSuche

```
public static void f() {
    f();
}
```

Rekursion bietet sich immer dann an, wenn man die Lösung eines Problems auf die Lösung gleichartiger aber kleinerer Teilprobleme zurückführen kann.

Rekursion

Den Aufruf einer Methode in ihrem eigenen Rumpf bezeichnet man als **Rekursion**. Erinnerung: Mischen BinäreSuche

Rekursion sollte irgendwann zum Ende kommen:

```
public static void f() {
    if (!ende) {
      f();
    }
  }
```

Rekursion bietet sich immer dann an, wenn man die Lösung eines Problems auf die Lösung gleichartiger aber kleinerer Teilprobleme zurückführen kann.

Weitere Beispiele von Rekursion

- Aufzählungsverfahren (alle Permutationen, Pflasterungen, ...)
- Ackermannfunktion:

$$
\begin{aligned}
A(0,y) &= y+1\\
A(x+1,0) &= 1\\
A(x+1,y+1) &= A(x, A(x+1,y))
\end{aligned}
$$

- Türme von Hanoi
- QuickSort

Weitere Beispiele von Rekursion

- Aufzählungsverfahren (alle Permutationen, Pflasterungen, ...)
- Ackermannfunktion:

$$
\begin{aligned}
&A(0, y) = y + 1\\
&A(x+1, 0) = 1\\
&A(x+1, y+1) = A(x, A(x+1, y))
\end{aligned}
$$

- Türme von Hanoi
- QuickSort

Merke:
Will man zeigen, dass eine rekursive Methode eine Spezifikation erfüllt (Vor- und Nachbedingung), so darf man dabei annehmen, dass rekursive Aufrufe im Rumpf der Methode diese bereits erfüllen.

Zusammenfassung

- Verschiedene Algorithmen (Rechenverfahren) für das gleiche Problem können sich drastisch in der Laufzeit unterscheiden.
- Die O-Notation gestattet es, Angaben über die Größenordnung einer Funktion, z.B. der Laufzeit zu machen.
- Selection Sort ist ein $O(n^2)$ Verfahren, Merge Sort ist ein $O(n \log(n))$ Verfahren zum Sortieren von Arrays. Merge Sort ist auch empirisch wesentlich performanter.
- Typvariablen und Wildcards erlauben präzisere Typisierung unter weitgehender Vermeidung von `Object` und typecast.
- Rekursive Verfahren beruhen auf der Zerlegung eines Problems in kleinere gleichartige Probleme.
- Formal bedeutet Rekursion den Aufruf einer Methode in ihrem eigenen Rumpf.