Beispiel: Partition-Exchange-Sort

Ein weiteres schnelles Sortierverfahren heißt Partition-Exchange-Sort oder Quicksort.

Es geht so vor:

1. Wähle ein Pivot-Element p (z.B. das erste Element im Array).
2. Partitioniere das Array in zwei Arrays. Das erste enthält alles was kleiner als p ist, das zweite den Rest.
3. Sortiere beide Arrays rekursive.
4. Packe p in die Mitte.

Bei der imperativen Implementierung mit Arrays geschieht das Partitionieren mithilfe von Vertauschungen, daher der Name Partition-Exchange.

Das Vergleichen mit dem Pivot-Element und weitersuchen benötigt typischerweise nur vier Maschineninstruktionen. Daher der Name *Quicksort*.

Wir implementieren dieses Verfahren mit Listen.

Hierzu implementieren wir die Funktionen *psort* und *partition*, welches ein Pivot-Element und eine Liste erhält.

```
psort :: [Integer]→[Integer]
psort [ ] = [ ]
psort (x : xs) = (psort l1) ++ [x] ++ (psort l2)
  where (l1, l2) = partition x xs
```

```
partition :: Integer→[Integer]→([Integer], [Integer])
partition x [ ] = ([ ], [ ])
partition x (y : ys) =
  if x < y
  then (l1, y : l2)
  else (y : l1, l2)
    where (l1, l2) = partition x ys
```

Typparameter

Betrachten wir diese Funktion:

```
zweites :: [Integer]→Integer
zweites (x : y : ys) = y
```

Sie findet das zweite Element in einer Liste von *Integer*.

Die Funktionalität sollte aber auch für beliebige Listen funktionieren.

Wir können *Typvariablen* in einer Signatur verwenden, um solch allgemeine Funktionen zu definieren. Dies ähnelt den generischen Typen von Java.

```
zweites :: [a]→a
zweites (x : y : ys) = y
```

Eigene Datentypen

So können wir einen einfachen Datentyp *Note* definieren.

Der Typ *Note* kann nur bestimmte Werte annehmen.

```
data Note = Gut | Schlecht | Mies deriving Show
eineschlechter :: Note → Note
eineschlechter Gut = Schlecht
eineschlechter Schlecht = Mies
```

Wir können eigene Datentypen ebenso wie *Integer* etc. verwenden.

deriving *Show* sagt, daß *Note* ein Untertyp von *Show* ist.
Dadurch können seine Werte einfach angezeigt werden.

Eigene Datentypen

Ein Datentyp kann von einem anderen Typ abhängen.

So definieren wir eine eigene Liste von *a*'s, wobei *a* wieder eine Typvariable ist. Wir können also auch so einen eigenen Listentyp für *Integer*-Listen definieren, aber auch für beliebige andere.

```
data List a = Nil | Cons a (List a) deriving Show
kopf :: List a → a
kopf (Cons x y) = x
schwanz :: List a → List a
schwanz (Cons x xs) = xs
append :: List a → List a → List a
append Nil l = l
append (Cons x xs) l = Cons x (append xs l)
laenge :: List a → Integer
laenge Nil = 0
laenge (Cons _ xs) = 1 + (laenge xs)
```

Eigene Datentypen

Hier ist ein Beispiel für Binärbäume, die Werte des Typs *a* in ihren Blättern besitzen.

```
data Bintree a = Leaf a | Node (Bintree a) (Bintree a)
  deriving Show
size :: Bintree a → Integer
size (Leaf x) = 1
size (Node l r) = (size l) + (size r)
data OrdBintree a = Ext | In a (OrdBintree a) (OrdBintree a)
  deriving Show
insert :: Ord a => a→OrdBintree a→OrdBintree a
insert x Ext = In x Ext Ext
insert x (In y yl yr) =
  if x < y
  then In y (insert x yl) yr
  else In y yl (insert x yr)
```