1 Einfache und höhere Sortierverfahren

1.1 Insertionsort - Sortieren durch Einfügen

Zu implementieren ist der Sortieralgorithmus durch Einfügen von beliebigen Elementen in eine neue Liste.

Algorithmische Idee: Aus einer gegebenen Liste werden die Elemente von vorn beginnend entnommen und sortiert in eine zweite Liste eingefügt.

Listing 1: Sortieren durch Einfügen in Haskell

```
1  import Data.List (insert)
2
3  -- Sortieren durch einfaches Entnehmen von Elementen aus einer unsortierten
4  -- Liste und "richtiges" Einfuegen in die sortierte Liste
5
6  -- Variante mit Akkumulator
7  insertionsort :: (Ord a) => [a] -> [a]
8  insertionsort [] = []
9  insertionsort l = insertionsort' l []
10
11 insertionsort' [] acc     = acc
12 insertionsort' (x:xs) acc = insertionsort' xs (ins x acc)
13
14 ins:: (Ord a) => a -> [a] -> [a]
15 ins a [] = [a]
16 ins a (x:xs) | (x<a) = x:(ins a xs)
17              | otherwise = a:(x:xs)
18
19 -- oder besser, weil kuerzer
20 -- sortieren auf dergleichen Liste
21
22 insertionsort2:: (Ord a) => [a] -> [a]
23 insertionsort2 [] = []
24 insertionsort2 (x:xs) = ins x (insertionsort2 xs)
25
26 -- es geht auch unter Nutzung hueherer Funktionen aus Data.List
27
28 insertionsort3:: (Ord a) => [a] -> [a]
29 insertionsort3 = foldr insert []
30
31 -- Anwendung
32 liste = [4,6,2,3,7,9,12,1]
33
34 main = print (insertionsort3 liste)
```

Aufgaben: Analysiere den Quellcode! Probiere alle Varianten aus.
Erweitere das Programm so, dass der User zur Laufzeit eingeben kann, welcher der drei Algorithmen ausgeführt wird.

1.2 Selectionsort - Sortieren durch Auswählen

Zu implementieren ist der Sortieralgorithmus durch Auswählen des kleinsten bzw. größten Elementes und einsetzen in eine neue Liste.

Algorithmische Idee: Aus einer gegebenen Liste wird das kleinste Element gesucht und jeweils vorn in eine Liste eingefügt.

Listing 2: Sortieren durch Auswählen in Haskell

```
1  -- Sortieren durch sortiertes Entnehmen von Elementen aus einer unsortierten
2  -- Liste und einfaches Einfuegen in die sortierte Liste
3
4  selectionsort :: (Ord a) => [a] -> [a]
5  selectionsort [] = []
6  selectionsort xs = m : selectionsort (delete m xs)
7    where
8        m = minelem xs
9
10 -- delete loescht bestimmte Elemente
11 -- aus einer Liste.
12
13 delete:: Eq a => a -> [a] -> [a]
14 delete m [] = []
15 delete m (x:xs) | (x == m) = delete m xs
16                 | otherwise = x:delete m xs
17
18 -- minelem sucht das kleinste Element einer Liste
19 -- entspricht der Bibliotheksfunktion minimum aus Data.List
20
21 minelem:: Ord a => [a] -> a
22 minelem [x] = x
23 minelem (x:y:xs) | (x <= y) = minelem (x:xs)
24                  | otherwise = minimum (y:xs)
25
26 -- Anwendung
27 liste = [4,6,2,3,7,9,12,1]
28
29 main = print (selectionsort liste)
```

1.3 Höhere Sortierverfahren - Merge- und Quicksort

Im Gegensatz zu den einfachen Verfahren, bei denen die Liste mit allen Elementen mehrfach durchlaufen werden muss, sind höhere Verfahren durch eine Reduktion der zu bearbeitenden Zielmenge gekennzeichnet. Das Sortierproblem für die gesamte Liste wird durch Aufteilen der Liste in Teillisten auf ein kleineres Problem reduziert. Im Idealfall teilen wir solange auf, bis ein- oder zweielementige Listen vorliegen. Die können schnell sortiert werden. Die dann sortierten Teillisten werden schnell zur sortierten Liste zusammengesetzt.

Zu implementieren ist der Mergesort.

Algorithmische Idee: Eine gegebene Liste wird durch fortlaufende Halbierung in zwei Teillisten geteilt, so dass nur einelementige Listen übrigbleiben. Diese werden dann unter Beachtung der Ordnung zusammengefügt.

Listing 3: Mergesort in Haskell

```
1  import Data.List (insert)
2
3  liste = [3,6,9,12,17,2,7,1]
4
5  merge :: Ord a => [a] -> [a] -> [a]
6  merge [] ys = ys
7  merge xs [] = xs
8  merge (x:xs) (y:ys)
9      | x <= y = x : merge xs (y:ys)
10     | otherwise = y : merge (x:xs) ys
11
12 mergesort :: Ord a => [a] -> [a]
13 mergesort [] = []
14 mergesort [x] = [x]
15 mergesort xs
16     = merge (mergesort top) (mergesort bottom)
17     where
18     (top, bottom) = splitAt (length xs `div` 2) xs
```

Zu implementieren ist der Quicksort.

Algorithmische Idee: Die Aufteilung in Teillisten erfolgt durch Vergleich mit einem Pivotelement (z.B. der Kopf der Liste). Diese Aufteilung erzeugt dann eine geordnete Menge einelementiger Listen, die dann zu einer Gesamtliste zusammengesetzt werden.

Listing 4: Quicksort in Haskell

```
1  liste = [3,6,9,12,17,2,7,1]
2
3  qsort :: Ord a => [a] -> [a]
4  qsort []     = []
5  qsort (x:xs) = qsort ak ++ agl ++ qsort agr
6      where
7          (ak,agl,agr) = teilen x xs ([],[x],[])
8
9  teilen :: Ord a => a -> [a] -> ([a],[a],[a]) -> ([a],[a],[a])
10 teilen _ [] (l,e,g) = (l,e,g)
11 teilen p (x:xs) (l,e,g)
12     | x > p     = teilen p xs (l,e,x:g)
13     | x < p     = teilen p xs (x:l,e,g)
14     | otherwise = teilen p xs (l,x:e,g)
15
16 -- andere Loesung
17
18 qsort2 :: Ord a => [a] -> [a]
19 qsort2 []     = []
20 qsort2 (x:xs) = qsort2 [y| y <- xs, y <= x] ++ [x] ++ qsort2 [y| y <- xs, y > x]
```