```haskell
data Grade = One | Two | Three | Four | Five | Six
  deriving (Show, Eq)

instance Ord Grade where
  One <= _ = True
  Two <= One = False
  Two <= _ = True
  Three <= One = False
  Three <= Two = False
  Three <= _ = True
  Four <= One = False
  Four <= Two = False
  Four <= Three = False
  Four <= _ = True
  Five <= Six = True
  Five <= _ = False
  Six <= Six = True
  Six <= _ = False

selectionsort :: (Ord a) => [a] -> [a]
selectionsort [] = []
selectionsort xs =
  let -- selectedElement = foldr (\ aktElem min -> if min < aktElem then min
else aktElem ) (head xs) xs
      -- selectedElement = foldr1 min xs
      selectedElement = minimum xs
      restWithoutSE = filter (/= selectedElement) xs
   in selectedElement : selectionsort restWithoutSE
selectionsort[4, 6, 1, 3]
1 : selectionsort [4, 6, 3]
1 : (3: selectionsort [4, 6])
1 : (3: (4: selectionsort [6]))
1 : (3: (4: (6:selectionsort [])))
[1, 3, 4, 6]

insertionsort :: (Ord a) => [a] -> [a]
insertionsort = foldr insertSorted []
  where
    insertSorted :: (Ord a) => a -> [a] -> [a]
    insertSorted x [] = [x]
    insertSorted x ys =
      let (smaller, biggerOrEqual) = span (< x) ys
       in --   let smaller = takeWhile (< x) ys
          --       biggerOrEqual = dropWhile (< x) ys
          smaller ++ (x : biggerOrEqual)
-- Durchlauf 1
insertionsort [4, 6, 1, 3]
insertSorted 3 [] = [3]
-- Durchlauf 2
insertSorted 1 [3] = span (<1) [3] = ([], [3]) = [1,3]
-- Durchlauf 3
insertSorted 6 [1,3] = span (<6) [1,3] = ([1, 3], []) =[1,3,6]
-- Durchlauf 4
insertSorted 4 [1,3,6] = span (<4) [1,3,6] = ([1, 3], [6]) = ([1, 3], [6]) =
[1,3,4,6]

bubblesort :: (Ord a) => [a] -> [a]
bubblesort xs =
  let (xs', swapped) = foldr bubble ([], False) xs
      bubble :: (Ord a) => a -> ([a], Bool) -> ([a], Bool)
      bubble x ([], b) = ([x], b)
      bubble x (yss@(y : ys), b) =
        if x <= y
          then (x : yss, b)
          else (y : x : ys, True)
   in if swapped then bubblesort xs' else xs'
```

```
-- 1. durchlauf
bubblesort[4,6,1,3]
foldr bubble([], False)[4,6,1,3]
bubble 3 ([],False) = ([3],False)
bubble 1 ([3],False) = ([1,3],False)
bubble 6 ([1,3],False) = ([1,6,3],True)
bubble 4 ([1,6,3],True) = ([1,4,6,3],True)
-- 2. durchlauf
bubblesort[1,4,6,3]
foldr bubble([], False)[1,4,6,3]
bubble 3 ([],False) = ([3],False)
bubble 6 ([3],False) = ([3,6],True)
bubble 4 ([3,6],True) = ([3,4,6],True)
bubble 1 ([3,4,6],True) = ([1,3,4,6],True)
-- 3. durchlauf
bubblesort[1,3,4,6]
foldr bubble([], False)[1,3,4,6]
bubble 6 ([],False) = ([6],False)
bubble 4 ([6],False) = ([4,6],False)
bubble 3 ([4,6],False) = ([3,4,6],False)
bubble 1 ([3,4,6],False) = ([1,3,4,6],False)

mergesort :: (Ord a) => [a] -> [a]
mergesort [] = []
mergesort [x] = [x]
mergesort list =
  let (fstHalf, sndHalf) = splitAt (length list `div` 2) list
      merge :: (Ord a) => [a] -> [a] -> [a]
      merge [] ys = ys
      merge xs [] = xs
      merge xss@(x : xs) yss@(y : ys) =
        if x <= y
          then x : merge xs yss
          else y : merge xss ys
   in merge (mergesort fstHalf) (mergesort sndHalf)
mergesort [4,6,1,3]
merge (mergesort [4,6])(mergesort [1,3])
merge (mergesort [4])(mergesort [6])(mergesort [1])(mergesort [3])
merge (merge[4][6])(merge[1][3])
merge (4: merge[][6])(1: merge[][3])
merge(4: [6])(1:[3])
merge [4,6][1,3]
1:merge [4,6][3]
1:(3:merge[4,6][])
1:(3:[4,6])
[1,3,4,6]

quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x : xs) =
  let smallerSorted = quicksort [a | a <- xs, a <= x]
      biggerSorted = quicksort [a | a <- xs, a > x]
   in smallerSorted ++ [x] ++ biggerSorted
quicksort [4, 6, 3, 1]
quicksort [1, 3] ++ [4] ++ quicksort[6]
quicksort []++[1] ++ quicksort[3] ++ [4] ++quicksort[] ++ [6] ++ quicksort[]
quicksort []++[1] ++ quicksort[]++[3] ++ quicksort[]++[4] ++ quicksort[] ++ [6]
++ quicksort[]
quicksort [1, 3, 4, 6]
```