Prof. Dr. Thomas Breuel
Dipl.-Inf. Patrick Michel (AG ST)
Dipl.-Ing. Ludwig Schmidt-Hackenberg

TU Kaiserslautern
Fachbereich Informatik
AG Bildverstehen und Mustererkennung

Lösungshinweise/-vorschläge zum Übungsblatt 7: Software-Entwicklung 1 (WS 2010/11)

Die Hinweise und Vorschläge in diesem Dokument sollen der Lösungsfindung dienen und erheben demnach weder Anspruch auf Vollständigkeit noch Korrektheit. Sollten Sie Fehler finden, würden wir uns freuen, wenn Sie uns diese mitteilen. (Kontaktdaten finden Sie auf unserer Webpräsenz.)

Aufgabe 1 Sortieren (Präsenzaufgabe)

In dieser Aufgabe sollen die in der Vorlesung vorgestellten Sortieralgorithmen am Beispiel geübt werden. Legen Sie sich selbst eine Reihenfolge fest, in der Sie die Beispiele lösen, so dass Sie frühzeitig nachfragen können bei Problemen.

a) Sortieren Sie die Liste `[7, 5, 9, 8, 4, 3, 6]` aufsteigend, einmal mit Hilfe des **Selectionsort** Algorithmus und einmal mit Hilfe des **Insertionsort** Algorithmus. Geben Sie Ihre Lösungen jeweils mit sinnvollen Zwischenschritten an, z.B. wie in der Vorlesung angedeutet!

Selectionsort sammelt die jeweils kleinsten Elemente und fügt sie am Ende zusammen:

```
[7, 5, 9, 8, 4, 3, 6]
3 : [7, 5, 9, 8, 4, 6]
3 : 4 : [7, 5, 9, 8, 6]
3 : 4 : 5 : [7, 9, 8, 6]
3 : 4 : 5 : 6 : [7, 9, 8]
3 : 4 : 5 : 6 : 7 : [9, 8]
3 : 4 : 5 : 6 : 7 : 8 : [9]
3 : 4 : 5 : 6 : 7 : 8 : 9 : []
[3, 4, 5, 6, 7, 8, 9]
```

Insertionsort fügt alle Elemente nach und nach in eine sortierte Liste ein. Die Darstellung ist nicht ganz korrekt, da die noch einzufügenden Elemente nicht in einer Liste stehen, sondern auf dem Call-Stack liegen.

Zudem ist anzumerken, dass die Haskell Implementierung erst beim rekursiven Aufstieg die Elemente in die Erbenisliste einfügt, also streng genommen von hinten. Für das Verständnis des Algorithmus ist dies allerdings irrelevant.

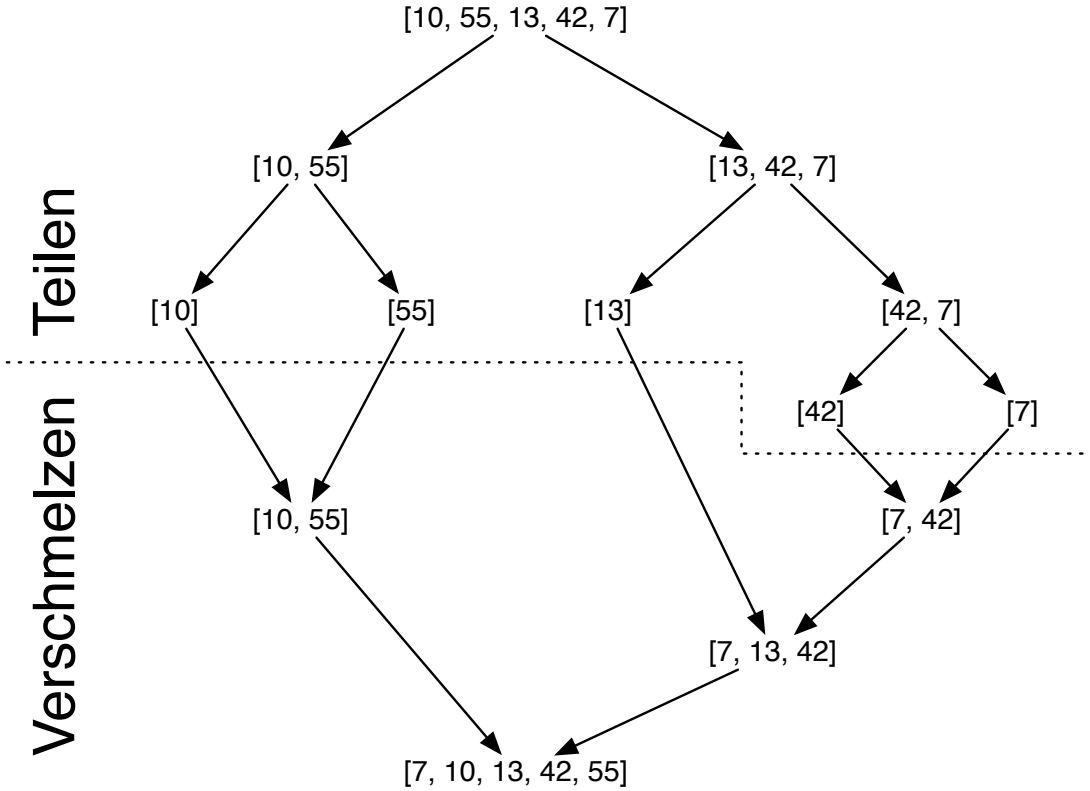
```
[7, 5, 9, 8, 4, 3, 6]
[5, 9, 8, 4, 3, 6] [7]
[9, 8, 4, 3, 6] [5, 7]
[8, 4, 3, 6] [5, 7, 9]
[4, 3, 6] [5, 7, 8, 9]
[3, 6] [4, 5, 7, 8, 9]
[6] [3, 4, 5, 7, 8, 9]
[] [3, 4, 5, 6, 7, 8, 9]
```

b) Sortieren Sie die gleiche Liste aufsteigend mit Hilfe des **Bubblesort** Algorithmus. Geben Sie dabei alle Zwischenschritte an, bei denen sich etwas in der Liste geändert hat! Geben Sie die Liste nach einem rekursiven Durchlauf an, entfernen Sie das maximale Element und schreiben Sie es dahinter.

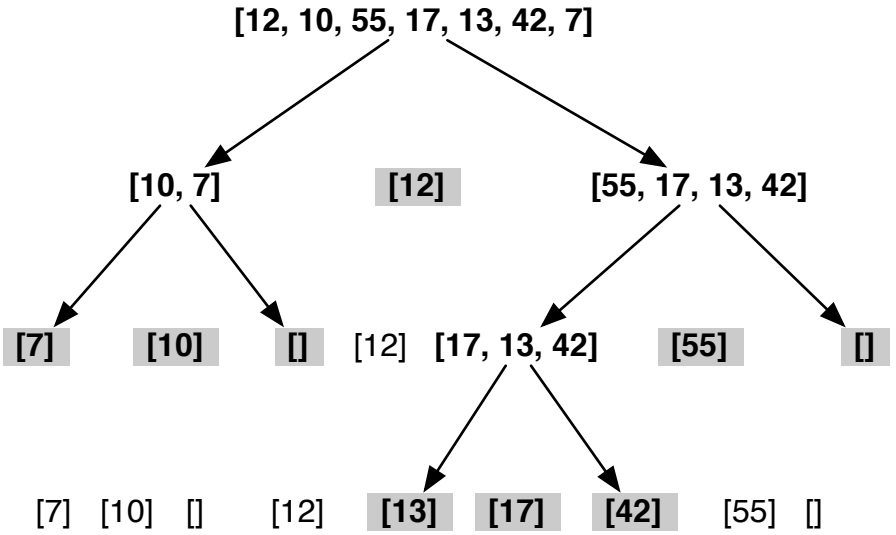
```
1. Durchlauf
[7, 5, 9, 8, 4, 3, 6]
[5, 7, 9, 8, 4, 3, 6]
[5, 7, 8, 9, 4, 3, 6]
[5, 7, 8, 4, 9, 3, 6]
[5, 7, 8, 4, 3, 9, 6]
[5, 7, 8, 4, 3, 6, 9]

2. Durchlauf
[5, 7, 8, 4, 3, 6] [9]
[5, 7, 4, 8, 3, 6]
[5, 7, 4, 3, 8, 6]
[5, 7, 4, 3, 6, 8]

3. Durchlauf
[5, 7, 4, 3, 6] [8, 9]
[5, 4, 7, 3, 6]
[5, 4, 3, 7, 6]
[5, 4, 3, 6, 7]

4. Durchlauf
[5, 4, 3, 6] [7, 8, 9]
[4, 5, 3, 6]
[4, 3, 5, 6]

5. Durchlauf
[4, 3, 5] [6, 7, 8, 9]
[3, 4, 5]

6. Durchlauf
[3, 4] [5, 6, 7, 8, 9]

7. Durchlauf
[3] [4, 5, 6, 7, 8, 9]

Endergebnis
[3, 4, 5, 6, 7, 8, 9]
```

c) Sortieren Sie die Liste `[10, 55, 13, 42, 7]` aufsteigend, mit Hilfe des **Mergesort** Algorithmus. Verwenden Sie zur Darstellung Ihrer Lösung die Baum bzw. Graphdarstellung aus der Vorlesung.

Die beiden sich für den jeweiligen Hauptschritt ergebenden Bäume lassen sich auch zu einem einzelnen Graphen zusammenfassen. Dieser sieht dann wie folgt aus:

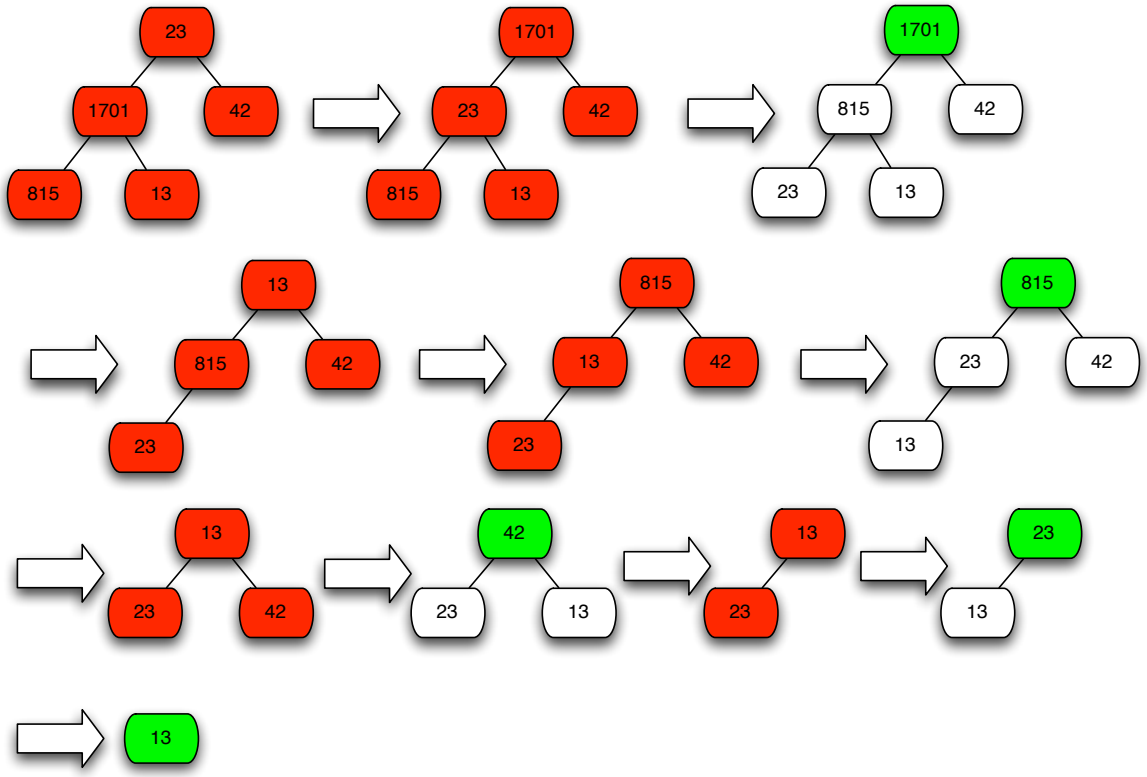


d) Sortieren Sie die Liste `[12, 10, 55, 17, 13, 42, 7]` aufsteigend, mit Hilfe des **Quicksort** Algorithmus. Verwenden Sie zur Darstellung Ihrer Lösung eine Baumdarstellung wie aus der Vorlesung.



e) Sortieren Sie die Liste `[23, 1701, 42, 815, 13]` absteigend, mit Hilfe des **Heapsort** Algorithmus. Geben Sie dabei alle entstehenden Bäume an und markieren Sie welche die Heap-Eigenschaft verletzen.

Rote Farbe kennzeichnet Bäume, welche die Heap-Eigenschaft verletzen. Grüne Knoten enthalten die Elemente der Ergebnisliste der durchgeführten Sortierung.



Aufgabe 2 Sortieren (Einreichaufgabe)

a) Implementieren Sie die Sortieralgorithmen der Vorlesung noch einmal selbst in Haskell, der Einfachheit halber für Listen von ganzen Zahlen. Heapsort müssen Sie *nicht* neu schreiben. Schreiben Sie also die fünf `[Integer] -> [Integer]` Funktionen:

```
  selectionsort, insertionsort, bubblesort, quicksort, mergesort

quicksort :: [Integer] -> [Integer]
quicksort [] = []
quicksort (pivot : rest) = let (smaller, bigger) = partition (< pivot) rest
  in quicksort smaller ++ pivot : quicksort bigger


mergesort :: [Integer] -> [Integer]
mergesort []  = []
mergesort [x] = [x]
mergesort l   =
  let (left, right) = splitAt (length l `div` 2) l
  in merge (mergesort left) (mergesort right)  where

  merge :: [Integer] -> [Integer] -> [Integer]
  merge [] b = b
  merge a [] = a
  merge (a : al) (b : bl) = if a < b
    then a : merge al (b : bl)
    else b : merge (a : al) bl


selectionsort :: [Integer] -> [Integer]
selectionsort [] = []
selectionsort l  = let m = minimum l in m : selectionsort (delete m l)


insertionsort :: [Integer] -> [Integer]
insertionsort []      = []                  -- alternativ teilen mit span
insertionsort (x : l) = insert x (insertionsort l)


-- Loesung bubblet von rechts nach links das kleinste raus, dadurch ist
-- die Implementierung fuer Haskell besser.
bubblesort :: [Integer] -> [Integer]
bubblesort [] = []
bubblesort l = let (m : rest) = bubble [] $ reverse l in m : bubblesort
   rest

bubble :: [Integer] -> [Integer] -> [Integer]
bubble r [x] = x : r
bubble r (x : y : rest) = if x < y
  then bubble (y : r) (x : rest)
  else bubble (x : r) (y : rest)


bubblesortSteps :: [Integer] -> [[Integer]]
bubblesortSteps [] = []
bubblesortSteps l  =
  let (m : rest) = bubble [] $ reverse l
  in (m : rest) : map (m :) (bubblesortSteps rest)
```

b) Überlegen Sie sich wie viele Vergleiche "`a < b`" (bzw. `leq` in den Folien) zwischen Elementen beim *Selectionsort* Algorithmus, auf einer Liste der Länge n, ausgeführt werden. Geben Sie eine möglichst geschlossene Form an.

Vergleiche werden beim Suchen des minimalen Elements verwendet. Es gibt n Durchläufe, da der Algorithmus erst im Fall der leeren Liste terminiert. Im ersten Durchlauf sind es $(n-1)$ Vergleiche, im letzten gar keiner.

$$f(n) = \sum_{i=1}^{n} i - 1 = \sum_{i=0}^{n-1} i = \frac{n \cdot (n-1)}{2}$$

c) Überlegen Sie sich wie viele Vergleiche "`a == b`" zwischen Elementen beim *Selectionsort* Algorithmus, auf einer Liste der Länge n, ausgeführt werden. Geben Sie eine maximale („worst-case") und eine minimale („best-case") Anzahl an.

Die Gleichheit wird beim Löschen des minimalen Elements aus dem Rest verwendet. Auch hier gibt es wieder n Durchläufe. Im besten Fall ist das gesuchte Element immer gleich das erste, so dass nur ein Vergleich benötigt wird. Im schlechtesten Fall ist das gesuchte Element immer das letzte, so dass n Vergleiche notwendig sind (wobei sich hier n dann auf die momentane Länger der Restliste bezieht).

$$f_{\texttt{min}}(n) = \sum_{i=1}^{n} 1 = n \qquad f_{\texttt{max}}(n) = \sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$$

d) Wie viele Vergleiche "`a < b`" fallen beim *Insertionsort* Algorithmus an, wiederum auf einer Liste der Länge n?

Die Frage ist fast vollkommen analog zu den "`a == b`" Vergleichen beim Selectionsort. Der Vergleich wird beim Einfügen des nächsten Elements in die Ergebnisliste verwendet. Auch hier gibt es wieder n Durchläufe. Im besten Fall ist das Element immer kleiner als das erste der Liste, so dass nur ein Vergleich benötigt wird. Im schlechtesten Fall ist das Element am Ende der Liste einzufügen, so dass n Vergleiche notwendig sind (wobei sich n dann auf die momentane Länge der Ergebnisliste bezieht).

Die Formeln sind fast dieselben wie oben, jedoch läuft der Index nicht von 1 bis n, sondern von 0 bis $n-1$, da man beim Selectionsort eine Liste abarbeitet, beim Insertionsort allerdings mit der leeren anfängt und aufbaut. Außerdem ist die allererste Einfügeoperation beim Insertionsort umsonst, da man mit nichts vergleichen kann, damit geht der Index dort erst bei 1 los, läuft aber trotzdem nur bis $n-1$:

$$f_{\texttt{min}}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \qquad f_{\texttt{max}}(n) = \sum_{i=0}^{n-1} i = \frac{n \cdot (n-1)}{2}$$

Aufgabe 3 Binäre Suche auf Listen (Einreichaufgabe)

In der Vorlesung haben wir die abstrakte Datenstruktur `Dictionary` mit Hilfe von Binärbäumen implementiert, um ihre Operationen – vor allem auch die Suche – zu optimieren. In dieser Aufgabe wollen wir die Datenstruktur mit Listen implementieren und schauen, ob wir die Suche trotzdem besser als linear implementieren können.

Bei der *binären Suche* auf einer Liste wird diese in zwei etwa gleich große Teillisten aufgespalten, danach entschieden, in welcher Teilliste das Element enthalten sein könnte und die binäre Suche rekursiv auf dieser Teilliste erneut ausgeführt. Dies wird solange wiederholt, bis das gewünschte Element gefunden ist oder aber klar ist, dass dieses nicht in der Liste vorkommt.

a) Welche Eigenschaft muss eine Liste erfüllen, damit eine binäre Suche durchgeführt werden kann?

Die Liste muss sortiert sein, da nur dann eine Entscheidung für die linke oder rechte Teilliste möglich ist, ohne diese jeweils komplett zu durchlaufen.

b) Implementieren Sie die Schnittstelle von `Dictionary`, mit Hilfe von Haskell Listen. Sie können sich dazu die Datei "`Dictionary.hs`" von der Vorlesungsseite laden und erweitern. Achten Sie bei der `put` Operation darauf die in a) identifizierte Eigenschaft auf dieser Liste zu erhalten.

Die Liste wird hier aufsteigend verwaltet. Anzumerken ist außerdem, dass sich die binäre Suche natürlich nicht nur für `get` anbietet, sondern alle drei Operationen. Bei jeder muss die entsprechende Stelle in der Liste gefunden werden, sei es zum Einfügen (`put`), zum Löschen (`remove`) oder eben der Suche (`get`).

```
-- type des Dictionarys
type Dict = [Dataset]

-- leeres Dictionary
emptyDict :: Dict
emptyDict = []

-- Einfuegen des Eintrags (k, s), ueberschreibt ggf. alten Eintrag zu k
put :: Dict -> Int -> String -> Dict
put [] k s = [(k, s)]
put ((k', s') : rest) k s
  | k == k'   = (k, s) : rest
  | k < k'    = (k, s) : (k', s') : rest
  | otherwise = (k', s') : put rest k s

-- Loeschen des Eintrags zu Schluessel k
remove :: Dict -> Int -> Dict
remove [] k = []
remove ((k', s) : rest) k
  | k == k'   = rest
  | k < k'    = (k', s) : rest
  | otherwise = (k', s) : remove rest k

-- Nachschauen des Eintrags zu Schluessel k
get :: Dict -> Int -> (Bool, String)
get []         k = (False, "")
get [(k', s)] k = (k == k', s)
get d          k = let (l, r) = splitAt (length d `div` 2) d in
  if k < fst (head r) then get l k else get r k
```

c) Wieviele Vergleiche mit Listenelementen sind bei einer `get` Operation auf einem `Dictionary` der Länge $n = 2^k$ mit $n, k \in \mathbb{N}$ durchzuführen, wenn das gesuchte Element nicht im `Dictionary` enthalten ist?

Bei einer Liste, deren Länge $n = 2^k$, also eine Zweierpotenz, ist, erfolgen insgesamt k Aufspaltungen mit $<$ Vergleichen und ein Vergleich am Ende mit `==`. Somit werden insgesamt $k + 1$ Vergleiche ausgeführt. Allgemein liegt die Anzahl der Vergleiche in $O(\log_2 n)$.

Wer sich darüber wundert, warum binäre Suche auch in funktionalen Sprachen – mit gekämmten Listen – Sinn macht, der muss darauf achten welche Operation wie oft verwendet wird! Es ist richtig, dass schon das Teilen der Liste $O(n)$ Operationen braucht, allerdings sind dies lediglich Operationen, die jeweils das erste Listenelement absplitten. Ist der Vergleich der Elemente hingegen sehr teuer, dann wird der Aufwand der Suche davon dominiert $O(\log_2 n)$ und nicht unbedingt vom Durchlaufen der Liste ($O(n)$). In der Praxis kann das also für die verwendeten Listen einen gewaltigen Unterschied machen, auch wenn die Suche asymptotisch gesehen immer noch linearen Aufwand hat. Wie so oft bei Komplexität haben die konstanten Faktoren hier einen großen praktischen Einfluss.

Aufgabe 4 Robosort (freiwillige Zusatzaufgabe)

Wie wir auf Blatt 3 schon gesehen haben, kann auch der Roboter einen unsortierten Eingabestapel sortieren! Insbesondere die Lösungen zu den Aufgaben 2b und 2d geben Ihnen ein Grundprimitiv, mit dem Sie direkt zwei der in der Vorlesung betrachteten Sortieralgorithmen implementieren können.

a) Schreiben Sie drei Roboterprogramme `selectionsort`, `insertionsort` und `bubblesort`, die jeweils einen unsortierten Anfangsstapel mit beliebig vielen Karten sortieren.

```
nach :: Stapel -> Stapel -> Programm
x `nach` y = Wiederhole x (Hole R x :> Lege R y)
```

```
bubblesort :: Programm
bubblesort = Wiederhole A bubble  where

  bubble =
       Hole L A                 -- Karte zum "Schieben"
    :> Wiederhole A             -- schiebe durch Stapel A
         ( Hole R A             --   naechste Karte in A
        :> Vergleiche Vertausche Nichts -- schiebe drueber oder wechsel
        :> Lege R C )           --   lege zurueckgelassene Karte weg
    :> Lege L B                 -- geschobene Karte ist nun maximal
    :> C `nach` A               -- lege "gebubbleten" Stapel zurueck


selectionsort :: Programm
selectionsort = Wiederhole A select  where

  suche = Wiederhole A ( Hole R A :> Vergleiche -- gehe durch restliche
       ( Lege L B   -- wenn eine groessere dabei ist, lege Kandidaten weg
      :> Lege R C   --   lege auch groessere Karte zurueck, wegen Ordnung
      :> C `nach` A --   alle kleineren als letzter Kandidat zurueck
      :> Hole L A ) --   nimm neuen Kandidaten
       ( Lege R C ) -- ansonsten lege die Vergleichkarte einfach weg
    )

  select =
       Hole L A   -- nehme den ersten Kandidaten
    :> suche      -- suche die richtige Karte
    :> C `nach` A -- raeume Stapel der kleineren Restkarten auf
    :> B `nach` A -- raeume Stapel der falschen Kandidaten auf
    :> Lege L D   -- lege das Maximum auf D (einzig sicherer Stapel)


insertionsort :: Programm
insertionsort = Wiederhole A insert  where

  insert =
       Hole L A                          -- Karte zum Einsortieren
    :> Wiederhole C (                    -- Stapel zum Einsortieren
            Hole R C                     -- Vergleichskarte
         :> Vergleiche ( Lege R B ) (    -- einfach weglegen
                 Lege L B :> Lege R B    -- Stelle gefunden, also Karten
              :> C `nach` B              --   weglegen und Rest schaufeln
              :> Hole L B                -- damit der Arm nicht leer ist
            )
       )                                 -- wenn C leer ist, ist Karte noch
    :> Lege L B                          --   in der Hand, deshalb zuruecklegen
    :> B `nach` C                        -- jetzt noch aufraeumen, zurueck nach C
```

b) Laden Sie sich die neue "RoboLib.hs" von der Vorlesungsseite und entwickeln Sie die beiden Roboterprogramme `quicksort` und `mergesort`.

Kommen Sie auch ohne Sperren und/oder unendlich große Programme (Haskell-Rekursion) aus?

```
ohne :: Programm -> Stapel -> Programm
x `ohne` y = Sperre y :> x :> Entsperre y
```

```
quicksort :: Stapel -> Programm
quicksort a =                                -- parametrisch im Stapel
  Wiederhole a (                             -- Abbruch bei leerem Stapel

      split                                  -- teile mit Pivot den Stapel
    :> quicksort b `ohne` c                  -- sortiere ein Haelfte
    :> quicksort c `ohne` b                  -- sortiere andere Haelfte

    :> Hole L D :> Lege L c                  -- fuege Pivot wieder ein

  ) :> Sperre D                              -- invertiere auf Pivotstack
    :> b `nach` D :> c `nach` D :> D `nach` a -- baue ueber D zu a zusammen
    :> Entsperre D  where                    -- gib Pivotstack wieder frei

    [b, c] = delete a [A, B, C]              -- ermittle jeweils andere Stapel

    split = Hole L a :> Wiederhole a (       -- nimm Pivot and teile
          Hole R a                           -- wenn kleiner aug b, sonst c
       :> Vergleiche ( Lege R c ) ( Lege R b )
     ) :> Lege L D                           -- lege am Ende Pivot auf D


mergesort :: Stapel -> Programm
mergesort a =                                -- auch hier parametrisch
  Wiederhole a ( Hole R a :> Wiederhole a ( -- doppelte Sicherung, da auch
                                             -- bei Singleton Abbruch
      Lege R a                               -- teile Stapel in der Haelfte
    :> Wiederhole a ( Hole L a :> Lege L b :> Hole L a :> Lege L c )

    :> mergesort b `ohne` c                  -- sortiere ein Haelfte
    :> mergesort c `ohne` b                  -- sortiere andere Haelfte

  ) :> Lege R b ) :> merge where             -- merge beide Haelften

  [b, c] = delete a [A, B, C]                -- ermittle jeweils andere Stapel

  merge =
      Hole L b :> Hole R c                   -- immer die ersten Karten
    :> Wiederhole b (                        -- lege passende weg und hole
       Ersatz
          Vergleiche ( Lege L D :> Hole L b ) ( Lege R D :> Hole R c ) )
    :> Wiederhole c (                        -- solange bis beide leer sind
          Vergleiche ( Lege L D :> Hole L b ) ( Lege R D :> Hole R c ) )
    :> Vergleiche ( Lege L D :> Lege R D ) ( Lege R D :> Lege L D )
    :> D `nach` a                            -- Haende evtl. nicht leer
```

c) Testen und vergleichen Sie die verschiedenen Algorithmen.

Aufgabe 5 Map-Reduce (Einreichaufgabe)

a) Schreiben Sie eine Funktion `countWord :: String -> [String] -> Int`, die berechnet wie oft ein Wort in einer Liste von Wörtern vorkommt, wobei Groß- und Kleinschreibung ignoriert wird.

```
countWord :: String -> [String] -> Int
countWord word = mapreduce
  (\x -> if map toLower x == map toLower word then 1 else 0) (+)
```

b) Schreiben Sie eine Funktion `countChar :: Char -> [String] -> Int`, die berechnet wie oft ein Buchstabe in einer Liste von Wörtern vorkommt, wobei Groß- und Kleinschreibung ignoriert wird.

```
countChar :: Char -> [String] -> Int
countChar c = flip mapreduce (+) $
  mapreduce (\x -> if toLower x == toLower c then 1 else 0) (+)
countChar c =  -- ALTERNATIVE, ohne verschachteltes mapreduce
  mapreduce (length . filter (flip elem [toUpper c, toLower c])) (+)
```

c) Schreiben Sie eine Funktion `wordsWithLength :: Int -> [String] -> [String]`, die alle Wörter mit gegebener Länge auflistet, wobei jedes Wort nur *einmal* im Ergebnis sein soll und auch hier Groß- und Kleinschreibung keine Rolle spielen sollen.

```
wordsWithLength :: Int -> [String] -> [String]
wordsWithLength n =
  mapreduce (\x -> if length x == n then [map toLower x] else []) union
```

d) Schreiben Sie eine Funktion `wordsContaining :: Char -> [String] -> [String]`, die alle Wörter die ein bestimmtes Zeichen enthalten auflistet, wobei jedes Wort nur *einmal* im Ergebnis sein soll und auch hier Groß- und Kleinschreibung keine Rolle spielen sollen.

```
wordsContaining :: Char -> [String] -> [String]
wordsContaining c =
  mapreduce (\x -> if any (== toLower c) $ map toLower x
    then [map toLower x] else []) union
```

e) Schreiben Sie eine Funktion `upperCaseWords :: [String] -> [String]`, die alle Wörter, die nur aus Großbuchstaben bestehen, auflistet, wobei alle Wörter nur *einmal* im Ergebnis auftauchen sollen.

```
upperCaseWords :: [String] -> [String]
upperCaseWords = mapreduce (\x -> if all isUpper x then [x] else []) union
```

f) Schreiben Sie eine Funktion `longestWord :: [String] -> String`, die das längste Wort einer Liste von Wörtern ermittelt.

```
longestWord :: [String] -> String
longestWord = mapreduce id (maxBy length)  where
  maxBy f a b = if f a > f b then a else b
```

g) *(freiwillige Zusatzaufgabe)* Schreiben Sie die Funktion `histogram :: [String] -> [(String, Int)]`, welche eine Histogramm für die Eingabe berechnet, sprich eine Liste aller Worte zusammen mit ihrer Häufigkeit.

```
-- Loesung mit: import Data.Map (unionWith, singleton, toList)
histogram :: [String] -> [(String, Int)]
histogram = reverse . sortBy (comparing snd) . toList .
  mapreduce (flip singleton 1) (unionWith (+))
```

In dieser Aufgabe wird es sehr hilfreich sein verschiedene Funktionen aus `Data.List` und `Data.Char` zu verwenden. Es lohnt sich insgesamt sich mit diesen Modulen vertraut zu machen. Hilfreich für diese Aufgabe könnten zum Beispiel sein:

```
and  or  any  all  isUpper  isLower  toUpper  toLower
map  nub  union  intersect  reverse  concat  filter  lookup
```

Aufgabe 6 Parameterinduktion (Präsenzaufgabe)

Geben Sie die Implementierung der Haskell-Funktion `multiply` an, welche die Multiplikation zweier *natürlicher* Zahlen auf Addition / Subtraktion zurückführt. Zeigen Sie für Ihre Implementierung unter Verwendung der *vollständigen Induktion*, dass die folgende Gleichheit gilt:

$$\texttt{multiply x y} = x \cdot y \qquad \forall x, y \in \mathbb{N}$$

Implementierung:

```
multiply :: Integer -> Integer -> Integer
multiply 0 y = 0
multiply x y = y + multiply (x - 1) y
```

Beweis:

I.A. $\texttt{multiply 0 y} \overset{\text{Def}}{=} \texttt{0} = 0 \cdot y$

I.V. $\texttt{multiply x y} = x \cdot y$

I.S.

$$\begin{aligned} \texttt{multiply (x+1) y} &\overset{\text{Def}}{=} \texttt{y + multiply x y} \\ &\overset{\text{IV}}{=} y + x \cdot y \\ &= (x+1) \cdot y \end{aligned}$$

Aufgabe 7 Parameterinduktion (Einreichaufgabe)

a) Geben Sie die Implementierung der Haskell-Funktion `sumquad` an, welche für gegebenen Parameter `n` die Summe der Quadratzahlen von 0 bis n^2 durch rekursive Berechnung ermittelt. Zeigen Sie für Ihre Implementierung unter Verwendung der *vollständigen Induktion*, dass die folgende Gleichheit gilt:

$$\texttt{sumquad n} = \frac{n(n+1)(2n+1)}{6}$$

Implementierung:

```
sumquad :: Integer -> Integer
sumquad 0 = 0
sumquad n = n * n + sumquad (n - 1)
```

Beweis:

I.A. $\texttt{sumquad 0} \overset{\text{Def}}{=} \texttt{0} = \frac{0(0+1)(2*0+1)}{6}$

I.V. $\texttt{sumquad n} = \frac{n(n+1)(2n+1)}{6}$

I.S.

$$\begin{aligned} \texttt{sumquad (n+1)} &\overset{\text{Def}}{=} \texttt{(n+1) * (n+1) + sumquad n} \\ &\overset{\text{IV}}{=} (n+1) * (n+1) + \frac{n(n+1)(2n+1)}{6} \\ &= \frac{\Big(6(n+1) + n(2n+1)\Big) * (n+1)}{6} \\ &= \frac{\Big(6n + 6 + 2n^2 + n\Big) * (n+1)}{6} \\ &= \frac{\Big(2n^2 + 7n + 6\Big) * (n+1)}{6} \\ &= \frac{(2n+3) * (n+2) * (n+1)}{6} \\ &= \frac{(n+1) * \Big((n+1)+1\Big) * \Big(2(n+1)+1\Big)}{6} \end{aligned}$$

b) Beweisen Sie, dass die Haskell-Funktionen

```
g :: Integer -> Integer
g n = h n 0 1  where

  h 0 x _ = x
  h n x y = h (n-1) (x + y) x
```

und

```
f :: Integer -> Integer
f 0 = 0
f 1 = 1
f n = f (n - 1) + f (n - 2)
```

für natürliche Zahlen gleich sind, also `g n = f n` für alle $n \in \mathbb{N}$. Zeigen Sie hierzu zunächst mittels *Induktion* über k, dass

$$\texttt{h (n-k) (f k) (f (k-1))} = \texttt{f n}$$

für $n, k \in \mathbb{N}$ mit $1 \leq k \leq n$ und $n > 0$ gilt. Nutzen Sie dieses Teilergebnis, um die eigentliche Aussage zu beweisen.

Hinweis: Im Fall obiger Induktion *über k ist es ratsam, bei k = n zu beginnen und den Induktionsschluß $k \to (k-1)$ durchzuführen!*

I.A. (k=n) $\texttt{h 0 (f n) (f (n-1))} \overset{\text{Def h}}{=} \texttt{f n}$

I.V. $\texttt{h (n-k) (f k) (f (k-1))} = \texttt{f n}$ für $n, k \in \mathbb{N}$ mit $1 \leq k \leq n$

I.S. (k→k-1)

$$\begin{aligned}
\texttt{h (n-(k-1)) (f (k-1)) (f (k-2))} &\overset{\text{Def h}}{=} \texttt{h (n-(k-1)-1) (f (k-1) + f (k-2)) (f (k-1))} \\
&\overset{\text{Def f}}{=} \texttt{h (n-k) (f k) (f (k-1))} \\
&\overset{\text{IV}}{=} \texttt{f n}
\end{aligned}$$

Durch die Induktion über k wurde die Gültigkeit von

$$\texttt{h (n-k) (f k) (f (k-1))} = \texttt{f n}$$

für $n, k \in \mathbb{N}$ mit $1 \leq k \leq n$ und $n > 0$ gezeigt. Für den Fall $n > 0$ ergibt sich

$$\begin{aligned}
\texttt{f n} &\overset{\text{Theorem}}{=} \texttt{h (n-k) (f k) (f (k-1))} \\
&\overset{\text{k=1}}{=} \texttt{h (n-1) (f 1) (f 0)} \\
&\overset{\text{Def h}}{=} \texttt{h n (f 0) (f 1)} \\
&\overset{\text{Def f}}{=} \texttt{h n 0 1} \\
&\overset{\text{Def g}}{=} \texttt{g n}
\end{aligned}$$

Der verbleibende Fall für $n = 0$ kann direkt bewiesen werden:

$$\texttt{g 0} \overset{\text{Def g}}{=} \texttt{h 0 0 1} \overset{\text{Def h}}{=} \texttt{0} \overset{\text{Def f}}{=} \texttt{f 0}$$

Aufgabe 8 Begriffe der prozeduralen Programmierung

Kreuzen Sie an, ob folgende Aussagen wahr oder falsch sind. Bereiten Sie diese Aufgabe bis zu Ihrer nächsten Übungsstunde vor, so dass Sie bei Unklarheiten nachfragen und die Antworten diskutieren können.

wahr	falsch	
☒	☐	Algorithmen beschreiben Abläufe.
☒	☐	Aktionen werden auf Zuständen ausgeführt.
☒	☐	Die Ausführung eines Algorithmus kann im Prinzip unendlich lange dauern. *Obwohl die Beschreibung eines Algorithmus endlich ist, kann die Ausführung unendlich viele Schritte umfassen, z.B. bei nicht terminierender Rekursion oder bei Schleifen.*
☐	☒	Der Zustand einer Variable wird durch eine Zuweisung nicht verändert.
☒	☐	Ein Algorithmus kann in verschiedenen Sprachen beschrieben werden.
☐	☒	Der Ausführungszustand wird durch die Position im Algorithmus bestimmt. *Zusätzlich zum Steuerungszustand, der beschreibt an welcher Stelle in der Beschreibung des Algorithmus man sich befindet, muss noch der Inhalt aller Variablen bekannt sein (Speicherzustand).*
☒	☐	Beim Ausführen *eines* Schrittes des Algorithmus können sich sowohl Speicher- als auch Steuerungszustand ändern. *Beispielsweise ändert ein Schritt, der einer Variablen einen Wert zuweist, sowohl den Speicherzustand als auch den Steuerungszustand.*
☐	☒	Beim Ausführen einer Aktion verändert sich der Steuerungszustand. *Eine Aktion mit dem Label l und der Anweisung "`gehe zu l`" würde den Steuerungszustand nicht verändern, wäre jedoch auch nur in wenigen Fällen sinnvoll.*
☒	☐	Es gibt Algorithmen bei deren Ausführung unabhängig von den Eingaben immer der gleiche Ablauf entsteht.
☐	☒	Die Beschreibung eines effizienten Algorithmus ist in der Regel kürzer als die eines weniger effizienten. *Zeitkomplexität: Die Zahl der Schritte bei einem Ablauf kann z.B. durch Rekursion oder Schleifen bei einer kurzen Beschreibung wesentlich höher sein als bei einer längeren, die auf diese Sprachmittel verzichtet, womit die längere Beschreibung effizienter ist. Eine ähnliche Argumentation gilt für die Raumkomplexität.*
☒	☐	Auch ein Java-Programm mit Fallunterscheidungen ist deterministisch. *Auch bei Fallunterscheidungen gilt, dass es zu jeder Eingabe genau einen Ablauf gibt, es wird nicht gefordert, dass alle Eingaben zum gleichen Ablauf führen.*
☒	☐	Die Auswertung einer Haskell-Funktion ist deterministisch.
☒	☐	Es gibt deterministische determinierte Algorithmen.
☒	☐	Es gibt nicht-deterministische determinierte Algorithmen.
☐	☒	Ein für alle Eingaben nicht-terminierender Algorithmus ist nicht-determiniert. *Der Algorithmus liefert für die gleiche Eingabe stets das gleiche Ergebnis nämlich* keines. *Damit ist die Bedingung für Determiniertheit erfüllt.*
☒	☐	Im prozeduralen Programmierparadigma werden Prozeduren als Abstraktionsmittel verwendet.