# 99 questions/Solutions/50

## From HaskellWiki

< 99 questions | Solutions

(***) Huffman codes.

We suppose a set of symbols with their frequencies, given as a list of fr(S,F) terms. Example: [fr(a,45),fr(b,13),fr(c,12),fr(d,16),fr(e,9),fr(f,5)]. Our objective is to construct a list hc(S,C) terms, where C is the Huffman code word for the symbol S. In our example, the result could be Hs = [hc(a,'0'), hc(b,'101'), hc(c,'100'), hc(d,'111'), hc(e,'1101'), hc(f,'1100')] [hc(a,'01'),...etc.]. The task shall be performed by the predicate huffman/2 defined as follows:

```
% huffman(Fs,Hs) :- Hs is the Huffman code table for the frequency table Fs
```

Solution:

```
import Data.List
import Data.Ord (comparing)

data HTree a = Leaf a | Branch (HTree a) (HTree a)
               deriving Show

huffman :: (Ord a, Ord w, Num w) => [(a,w)] -> [(a,[Char])]
huffman freq = sortBy (comparing fst) $ serialize $
       htree $ sortBy (comparing fst) $ [(w, Leaf x) | (x,w) <- freq]
  where htree [(_, t)] = t
       htree ((w1,t1):(w2,t2):wts) =
               htree $ insertBy (comparing fst) (w1 + w2, Branch t1 t2) wts
       serialize (Branch l r) =
               [(x, '0':code) | (x, code) <- serialize l] ++
               [(x, '1':code) | (x, code) <- serialize r]
       serialize (Leaf x) = [(x, "")]
```

The argument to `htree` is a list of (weight, tree) pairs, in order of increasing weight. The implementation could be made more efficient by using a priority queue instead of an ordered list.

Or, a solution that does not use trees:

```
import List
-- tupleUpdate - a function to record the Huffman codes; add string
--               "1" or "0" to element 'c' of tuple array ta
-- let ta = [('a',"0"),('b',"1")]
-- tupleUpdate ta 'c' "1" =>  [('c',"1"),('a',"0"),('b',"1")]
tupleUpdate :: [(Char,[Char])]->Char->String ->[(Char,[Char])]
tupleUpdate ta el val
```

```
   | ((dropWhile(\x -> (fst x)/= el) ta)==[])= (el,val):ta
   | otherwise = (takeWhile (\x -> (fst x)/=el) ta) ++ ((fst(head ha),val ++ snd(head ha)) :
     where ha = [(xx,yy)|(xx,yy) <- ta,xx ==el]

-- tupleUpdater - wrapper for tupleUpdate, use a list decomposition "for loop"
-- let ta = [('a',"0"),('b',"1")]
-- tupleUpdater ta "fe" "1" => [('e',"1"),('f',"1"),('a',"0"),('b',"1")]
tupleUpdater :: [(Char,[Char])]->String->String ->[(Char,[Char])]
tupleUpdater a (x:xs) c =  tupleUpdater (tupleUpdate a x c) xs c
tupleUpdater a [] c = a

-- huffer - recursively run the encoding algorithm and record the left/right
--          codes as they are discovered in argument hc, which starts as []
-- let ha =[(45,"a"),(13,"b"),(12,"c"),(16,"d"),(9,"e"),(5,"f")]
-- huffer ha [] => ([(100,"acbfed")],[('a',"0"),('b',"101"),('c',"100"),('d',"111"),('e',"110
huffer :: [(Integer,String)] -> [(Char,[Char])]-> ([(Integer,String)],[(Char,[Char])])
huffer ha hc
   | ((length ha)==1)=(ha,sort hc)
   | otherwise       = huffer ((num,str): tail(tail(has)) ) hc2
     where num   = fst (head has)  + fst (head (tail has))
           left  = snd (head has)
           rght  = snd (head (tail has))
           str   = left  ++ rght
           has   = sort ha
           hc2   = tupleUpdater (tupleUpdater hc rght "1") left "0"

-- huffman - wrapper for huffer to convert the input to a format huffer likes
--           and extract the output to match the problem specification
huffman :: [(Char,Integer)] -> [(Char,[Char])]
huffman h = snd(huffer (zip (map snd h) (map (:[]) (map fst h))) [])
```

A relatively short solution:

```
import Data.List (sortBy, insertBy)
import Data.Ord (comparing)
import Control.Arrow (second)

huffman :: [(Char, Int)] -> [(Char, String)]
huffman =
  let shrink [(_, ys)] = sortBy (comparing fst) ys
      shrink (x1:x2:xs) = shrink $ insertBy (comparing fst) (add x1 x2) xs
      add (p1, xs1) (p2, xs2) =
        (p1 + p2, map (second ('0':)) xs1 ++ map (second ('1':)) xs2)
  in  shrink . map (\(c, p) -> (p, [(c ,"")])) . sortBy (comparing snd)
```

Another short solution that's relatively easy to understand (I'll be back to comment later):

```
import qualified Data.List as L

huffman :: [(Char, Int)] -> [(Char, [Char])]
huffman x = reformat $ huffman_combine $ resort $ morph x
        where
                morph x = [ ([[]],[c],n) | (c,n) <- x ]
                resort x = L.sortBy (\(_,_,a) (_,_,b) -> compare a b) x

                reformat (x,y,_) = L.sortBy (\(a,b) (x,y) -> compare (length b) (length y)) $
```

```
huffman_combine (x:[]) = x
huffman_combine (x:xs) = huffman_combine $ resort ( (combine_elements x (head
        where
                combine_elements (a,b,c) (x,y,z) = ( (map ('0':) a) ++ (map (
```


Category:

- Programming exercise spoilers

---