1. Define a function \textit{difference} which, when given two sets \texttt{xs} and \texttt{ys} (represented as lists) returns the set consisting of all elements of \texttt{ys} not found in \texttt{xs}. For example,

\begin{verbatim}
*Main> difference "abcd" "ad"
"bc"
\end{verbatim}

2. Given a list representing a set, the function \textit{splits} returns a list of pairs of lists representing all the ways in which the set can be decomposed into two non-empty subsets. For example,

\begin{verbatim}
*Main> :t splits
:t splits
splits :: (Ord a) => [a] -> [( [a], [a] )]
*Main> splits "abc"
[ ("c","ab"), ("b","ac"), ("bc","a"), ("a","bc"), ("ac","b"), ("ab","c") ]
\end{verbatim}

Write \textit{splits}.

3. The function \textit{argmin} takes a function \texttt{f} and a list \texttt{xs} as arguments and returns the element of the list \texttt{x} such that \texttt{f} applied to \texttt{x} has minimum value. For example,

\begin{verbatim}
*Main> :t argmin
:argmin
argmin :: (Ord a) => (t -> a) -> [t] -> t
*Main> argmin length ["ABC","EF","GHIJ","K"]
"K"
\end{verbatim}

Write \textit{argmin}.

4. The function \textit{fano} takes a list of pairs of source alphabet probabilities and values and returns a Fano coding tree which can be used with the functions, \texttt{encode} and \texttt{decode}, for encoding and decoding Huffman coding trees defined in class. The Fano coding algorithm is very simple: it splits its list argument into two subsets where the sum of the probabilities in each subset are as nearly equal as possible. It then uses the first subset to recursively build the left half of the Fano coding tree and the second subset to recursively build the right half of the Fano coding tree. For further information (and helpful figures) see

\url{http://en.wikipedia.org/wiki/ShannonFano_coding}

For example,
5. The function \textit{church} takes an integer \textit{n} as its argument and returns a function which composes any unary function \textit{n} times. For example,

\begin{verbatim}
*Main> :t church
cchurch :: (Num t) => t -> (c -> c) -> c -> c
*Main> (church 4) tail "ABCDEFGH"
"EFGH"
\end{verbatim}

Write \textit{church} using \textit{foldr}.

6. The function \textit{gop} (short for \textit{generalized-outer-product}) takes a length \textit{m} list of length \textit{n} lists (all of the same type) and returns a length \textit{n}^\textit{m} list of length \textit{m} lists representing the \textit{m}-fold cartesian product of elements from the lists. For example,

\begin{verbatim}
*Main> :t gop
ggop :: [[a]] -> [[a]]
*Main> gop ["ABC","DEF","GHI"]
[["ADG","ADH","ADI","AEG","AEH","AEI","AFG","AFH","AFI","BDG","BDH","BDI","BEG","BEH","BEI","BFG","BFH","BFI","CDG","CDH","CDI","CEG","CEH","CEI","CFG","CFH","CFI"]
*Main>
\end{verbatim}

Write \textit{gop}. Hint: Use a recursive nested-map.

7. Write a function \textit{trees} which takes a list of leaf values as its argument and returns a list of all binary trees with the given leaves. For example,
*Main> :t trees
trees :: (Ord t) => [t] -> [Btree t]
*Main> (trees "ABCDE") !! 114
Fork (Leaf 'E') (Fork (Fork (Leaf 'A') (Fork (Leaf 'C') (Leaf 'B'))) (Leaf 'D'))
*Main> length (trees [0..4])
1680

Hint: Define trees using a list-comprehension, recursion, and the function splits.