1. module Innerprod
   where

   innerprod :: Num a => [a] -> [a] -> a
   innerprod [] [] = 0
   innerprod v w.
   | (length v) /= (length w) = -1
   | otherwise = (head v) * (head w) + (innerprod (tail v) (tail w))

   Alternative solution (note that the lengths of the tails is compared):

   innerprod :: Num a => [a] -> [a] -> a
   innerprod [] [] = 0
   innerprod (x:xs) (y:ys)
   | (length xs) /= (length ys) = -1
   | otherwise = x*y+(innerprod xs ys)

2. module Eratosthenes
   where

   import Data.List
   -- necessary for the \ function is list difference (non-associative)

   sieve :: (Enum a, Num a, Eq a) => a -> a -> [a]
   sieve x = sieveaux x [2..x]

   sieveaux :: (Enum a, Num a, Eq a) => a -> a -> [a]
   sieveaux _ [] = []
   sieveaux x (y:ys) = y : (sieveaux x (ys \ [y,2*y..x]))

3. (a) indices :: [a] -> [Int]
   indices x = [0..((length x)-1)]

   qs_idx :: [Int] -> [Int] -> [Int]
qs_idx [] lst = []
qs_idx (x:xs) lst =
  (qs_idx [y | y <- xs, (lst!!y) < (lst!!x)] lst) ++ x : (qs_idx [y | y <- xs, (lst!!y) >= (lst!!x)] lst)

qs_lol :: [[Int]] -> [[Int]]
qs_lol lsts = (map (\lst -> (map (\index -> (lst !! index))
  sortedIndices) ) lsts )
where
  lastLst = (last lsts)
  sortedIndices = qs_idx (indices lastLst) lastLst
-- (!!) is the list index operator
-- last extracts the last element of a list

(b) qs_lc_tuple_f :: [(Int,Int)] -> [(Int,Int)]
qs_lc_tuple_f [] = []
qs_lc_tuple_f (pivot:xs) = (qs_lc_tuple_f lesser) ++ pivot:(qs_lc_tuple_f greater)
where
  lesser = [y | y <- xs, y < pivot]
  greater = [y | y <- xs, y >= pivot]
-- Note that when haskell compares tuples, it uses the first
element of the tuple by default.
-- fst and snd retrieve the first, resp. the second second,
element out of a tuple
qs_lc_tuple_l :: [(Int,Int)] -> [(Int,Int)]
qs_lc_tuple_l [] = []
qs_lc_tuple_l (pivot:xs) = (qs_lc_tuple_l [y | y <- xs, (snd y) < (snd pivot)]) ++ pivot:(qs_lc_tuple_l [y | y <- xs, (snd y) >= (snd pivot)])

4. module NonDom

  where

  qs_lc_tuple_f :: [(Int,Int)] -> [(Int,Int)]
  qs_lc_tuple_f [] = []
  qs_lc_tuple_f (pivot:xs) = (qs_lc_tuple_f lesser) ++ pivot:(qs_lc_tuple_f greater)
  where
    lesser = [y | y <- xs, y < pivot]
    greater = [y | y <- xs, y >= pivot]

  nonDom :: [(Int,Int)] -> [(Int,Int)]
  nonDom [] = []
  nonDom ls = ndaux (qs_lc_tuple_f ls) (snd (head (qs_lc_tuple_f ls)) + 1)
ndaux :: [(Int,Int)] -> Int -> [(Int,Int)]
ndaux [] x = []
ndaux (lh:ltail) y = if (snd lh) < y then lh:(ndaux ltail (snd lh)) else (ndaux ltail y)