```haskell
module Zerrenda_eraketa where

import Data.List
-------------------------------------

zenbakiak:: Int -> [Int]

zenbakiak n = [ x | x <- [0..n]]
-------------------------------------

zenbakiak2:: Int -> [Int]

zenbakiak2 n = [0..n]


-------------------------------------
bikoitiak:: Int -> [Int]

bikoitiak n = [ x | x <- [0..n], x `mod` 2 == 0]

-------------------------------------

bikoteak:: Int -> [(Int, Int)]

bikoteak n = [ (x, y) | x <- [0..n], y <- [0..n]]

-------------------------------------

bikoteak_infinitua:: [(Integer, Integer)]

bikoteak_infinitua = [ (x, y) | x <- [0..], y <- [0..]]


-------------------------------------

bikoteak_finitua:: Integer -> [(Integer, Integer)]

bikoteak_finitua n = genericTake n bikoteak_infinitua

-------------------------------------

bikoteak_hand:: Integer -> [(Integer, Integer)]

bikoteak_hand n = [ (x, y) | x <- [0..n], y <- [0..n], x < y]

-------------------------------------
```

```haskell
bikoteak_batu:: Int -> [Int]

bikoteak_batu n = [ x + y | x <- [0..n], y <- [0..n]]

-------------------------------------

zerrendak:: Int -> [[Int]]

zerrendak n = [ [1..x] | x <- [0..n]]

-------------------------------------

zatizer_ze:: Integer -> [Integer]

zatizer_ze n = [ x | x <- [1..n], n `mod` x == 0]

-------------------------------------

zatizer2_ze:: Integer -> [Integer]

zatizer2_ze n
        | n <= 0        = error "Zenbakia ez da positiboa"
        | otherwise     = [ x | x <- [1..n], n `mod` x == 0]

-------------------------------------

zatizer3_ze:: Integer -> [Integer]

zatizer3_ze n
        | n <= 0        = error "Zenbakia ez da positiboa"
        | n == 1        = [1]
        | otherwise     = [1] ++ [ x | x <- [2..( n `div` 2)], n `mod` x == 0] ++ [n]



-------------------------------------

lehena_ze:: Integer -> Bool

lehena_ze n
        | n <= 0        = error "Zenbakia ez da positiboa"
        | n == 1        = False
        | otherwise     = (length [ x | x <- [2..( n `div` 2)], n `mod` x == 0]) == 0

-------------------------------------

fakt_ze:: Integer -> Integer
```

```
fakt_ze n = product [1..n]

-------------------------------------

fakt2_ze:: Integer -> Integer

fakt2_ze n
        | n <= (-1)     = error "Zenbakia negatiboa da"
        | otherwise     = product [1..n]

-------------------------------------

betea_ze:: Integer -> Bool

betea_ze n
        | n <= 0        = error "Zenbakia ez da positiboa"
        | otherwise     = (sum [ x | x <- [1..n - 1], n `mod` x == 0]) == n

-------------------------------------

betea2_ze:: Integer -> Bool

betea2_ze n
        | n <= 0        = error "Zenbakia ez da positiboa"
        | otherwise     = (sum [ x | x <- [1..(n `div` 2)], n `mod` x == 0]) == n


-------------------------------------

qs :: [Integer] -> [Integer]

qs [] = []
qs (x:s)
        | s == []       = [x]
        | otherwise     = (qs [y | y <- s, y <= x]) ++ [x] ++ (qs [y | y <- s, y > x])


-------------------------------------

qs2 :: [Integer] -> [Integer]

qs2 [] = []
qs2 (x:s) = (qs2 [y | y <- s, y <= x]) ++ [x] ++ (qs2 [y | y <- s, y > x])
```

```
-------------------------------------

faktoreak_ze:: Integer -> [Integer]

faktoreak_ze n
        | n <= 1        = error "Zenbakia 2 baino txikiagoa da"
        | otherwise     = [ x | x <- [2..n], n `mod` x == 0, lehena_ze x]

-------------------------------------

faktoreak2_ze:: Integer -> [Integer]

faktoreak2_ze n
        | n <= 1        = error "Zenbakia 2 baino txikiagoa da"
        | lehena_ze n   = [n]
        | otherwise     = [ x | x <- [2..(n `div` 2)], n `mod` x == 0, lehena_ze x]


-------------------------------------

desk_ze:: Integer -> [Integer]

desk_ze n
        | n <= 1        = error "Zenbakia 2 baino txikiagoa da"
        | lehena_ze n   = [n]
        | otherwise     = qs((faktoreak2_ze n) ++
                                (desk_ze (n `div` (product (faktoreak2_ze n)))))

-------------------------------------

desk2_ze:: Integer -> [Integer]

desk2_ze n
        | n <= 1        = error "Zenbakia 2 baino txikiagoa da"
        | lehena_ze n   = [n]
        | otherwise     = qs(w ++ (desk2_ze (n `div` (product w))))
                                where w = faktoreak2_ze n



-------------------------------------

denen_faktoreak :: [[Integer]]

denen_faktoreak = [faktoreak2_ze y | y <- [2..]]

-------------------------------------
```

```
faktoreak_finitua :: Integer -> [[Integer]]

faktoreak_finitua n = [faktoreak2_ze x | x <- [2..(n + 1)]]

-------------------------------------

faktoreak_finitua2 :: Integer -> [[Integer]]

faktoreak_finitua2 n
        | n < 0          = error "Negatiboa"
        | otherwise      = genericTake n denen_faktoreak


-------------------------------------

faktorizatuak :: [(Integer,[Integer])]

faktorizatuak = zip [2..] denen_faktoreak

-------------------------------------

faktore_bakarrekoak :: [(Integer,[Integer])]

faktore_bakarrekoak = [(x,y) | (x,y) <- faktorizatuak, not (lehena_ze x), length y == 1]

-------------------------------------
-- faktore_bakarrekoak funtzioa definitzeko beste aukera bat

faktore_bakarrekoak2 :: [(Integer,[Integer])]

faktore_bakarrekoak2 = [(x,y:s) | (x,y:s) <- faktorizatuak, not (lehena_ze x), s == []]

-------------------------------------
-- faktore_bakarrekoak funtzioa definitzeko hirugarren aukera bat

faktore_bakarrekoak3 :: [(Integer,[Integer])]

faktore_bakarrekoak3 = [ (x,[y]) | (x,[y]) <- faktorizatuak, not (lehena_ze x)]

-------------------------------------
--Laugarren aukera

faktore_bakarrekoak4 :: [(Integer,[Integer])]

faktore_bakarrekoak4 = [ (x, [y]) | (x, [y]) <- faktorizatuak, x /= y]

-------------------------------------
```

```haskell
erabateko_berredurak :: [Integer]

erabateko_berredurak = [ x | (x,y) <- faktore_bakarrekoak]

-------------------------------------

erabateko_lehenengoak :: Integer -> [Integer]

erabateko_lehenengoak n

   | n < 0          = error "Balio negatiboa"
   | otherwise      = genericTake n erabateko_berredurak

-- take funtzioak Int motarentzat bakarrik balio du.
-- Hemen Integer mota erabiltzen ari garenez,
-- genericTake behar dugu.

-------------------------------------

erabateko_handiagoak :: Integer -> Integer -> [Integer]

erabateko_handiagoak n m

   | (n < 0) || (m < 0)       = error "Balio negatiboa"
   | otherwise                = genericTake n [ y | y <- erabateko_berredurak, y > m]

-- take funtzioak Int motarentzat bakarrik balio du.
-- Hemen Integer mota erabiltzen ari garenez,
-- genericTake behar dugu.

-------------------------------------

qs_lag :: [[Integer]] -> [Integer] -> [Integer]

qs_lag [] ord = ord
qs_lag (x:s) ord
        | (length x == 0) || (length x == 1)    = qs_lag s (ord ++ x)
        | otherwise     = qs_lag ([q1, [head x], q2] ++ s) ord
                                where   q1 = [y | y <- (tail x), y <= (head x)]
                                        q2 = [y | y <- (tail x), y > (head x)]
------------------------------------

qs_be :: [Integer] -> [Integer]

qs_be r = qs_lag [r] []
```

```haskell
------------------------------------

nahastu :: [Integer] -> [Integer] -> [Integer]

nahastu [] r = r
nahastu (x:s) r
        | r == []       = (x:s)
        | x <= (head r) = x : (nahastu s r)
        | otherwise     = (head r) : (nahastu (x:s) (tail r))

------------------------------------

nahastu_lag:: [Integer] -> [Integer] -> [Integer] -> [Integer]

nahastu_lag [] r q = q ++ r
nahastu_lag (x:s) r q
        | r == []              = q ++ (x:s)
        | x <= (head r) = nahastu_lag s r (q ++ [x])
        | otherwise     = nahastu_lag (x:s) (tail r) (q ++ [head r])

------------------------------------
nahastu_be:: [Integer] -> [Integer] -> [Integer]

nahastu_be r w = nahastu_lag r w []

------------------------------------

ms :: [Integer] -> [Integer]

ms [] = []
ms (x:s)
        | s == []       = [x]
        | otherwise     = nahastu_be (ms q1) (ms q2)
                              where   q1 = genericTake ((length (x:s)) `div` 2) (x:s)
                                      q2 = genericDrop ((length (x:s)) `div` 2) (x:s)


------------------------------------

ms_lag :: [[Integer]] -> [Integer]

ms_lag [] = []
ms_lag (x:s)
        | s == []       = x
        | otherwise     = ms_lag (q : (tail s))
                              where   q = nahastu_be x (head s)
------------------------------------
```

```
ms_be :: [Integer] -> [Integer]

ms_be r = ms_lag [[y] | y <- r]

------------------------------------
hbp :: [Integer] -> [Integer]

hbp [] = []
hbp (x:s) = [y | y <- (x:s), lehena_ze y] ++ [y | y <- (x:s), not (lehena_ze y), y `mod` 2 == 0] ++
                [y | y <- (x:s), not (lehena_ze y), y `mod` 2 /= 0]

------------------------------------
hbp2 :: [Integer] -> [Integer]

hbp2 r = gorriak ++ zuriak ++ urdinak
                where   gorriak = [y | y <- r, lehena_ze y]
                        zuriak = [y | y <- r, not (lehena_ze y), y `mod` 2 == 0]
                        urdinak = [y | y <- r, not (lehena_ze y), y `mod` 2 /= 0]

------------------------------------
```