1. Uraikan langkah evaluasi
   a. [(i, j) | i <- [1,2], j <- [1..4]]

      >> i = 1, j = 1 -> [(1,1)]

      >> i = 1, j = 2 -> [(1,1), (1, 2)]

      >> i = 1, j = 3 -> [(1,1), (1, 2), (1, 3)]

      >> i = 1, j = 4 -> [(1,1), (1, 2), (1, 3), (1, 4)]

      >> i = 2, j = 1 -> [(1,1), (1, 2), (1, 3), (1, 4), (2, 1)]

      >> i = 2, j = 2 -> [(1,1), (1, 2), (1, 3), (1, 4), (2, 1), (2, 2)]

      >> i = 2, j = 3 -> [(1,1), (1, 2), (1, 3), (1, 4), (2, 1), (2, 2), (2, 3)]

      >> i = 2, j = 4 -> [(1,1), (1, 2), (1, 3), (1, 4), (2, 1), (2, 2), (2, 3), (2, 4)]
      COMPLETED

   b. take 3 [[(i, j) | i <- [1,2]] | j <- [1..]]

      j will get iterated to the upped bound given in the first parameter of take which is 3. So, here is how it gets evaluated.

      >> [(1, 1) , (2, 1)] : take 2 [[(I, j) | I <- [1,2]] | j <- [2..]]

      >> [(1, 1) , (2, 1)] : [(1, 2) , (2, 2)] : take 1 [[(I, j) | I <- [1,2]] | j <- [3..]]

      >> [(1, 1) , (2, 1)] : [(1, 2) , (2, 2)] : [(1, 3) , (2, 3)] : take 0 [[(I, j) | I <- [1,2]] | j <- [4..]]

      >> [(1, 1) , (2, 1)] : [(1, 2) , (2, 2)] : [(1, 3) , (2, 3)] : [[]]

      >> [[(1,1),(2,1)],[(1,2),(2,2)],[(1,3),(2,3)]]

2. Jika ada fungsi **add x y = x + x**, apa output dari add 5 (4/0)? Mengapa demikian? Apa yang terjadi jika fungsi tersebut dijalankan di strict programming language (seperti python)?

   Answer: Fungsi tersebut akan mengembalikan nilai 10, hal ini bisa terjadi karena adanya lazy evaluation pada Haskell. Jika dilihat pada pendefinisian fungsi, parameter y tidak pernah dipakai pada sisi kanan fungsi, sehingga sesuai konsep lazy evaluation, parameter y tidak akan dievaluasi karena nilainya belum dibutuhkan. Sedangkan pada python, pemanggilan fungsi yang sama akan mendapatkan *ZeroDivisionError* karena python melakukan eager evaluation pada setiap parameter yang diberikan meskipun nilai parameter tersebut tidak dipakai dalam logika fungsi.

3. Quicksort using list comprehension

```
qsort [] = []
qsort (x:xs) = qsort small ++ (x: qsort bigger)
   where
      small = [y | y <- xs, y <= x]
      bigger = [y | y <- xs, y > x]
```

Penjelasan:

Qsort akan melakukan sorting dengan bantuan list comprehension untuk melakukan sorting pada dua partisi list yaitu small dan bigger. List small akan mengambil semua nilai yang lebih kecil dari pivot, sedangkan list bigger akan mengambil semua nilai yang lebih besar dari pivot. Pivot yang dipilih adalah elemen pertama list pada setiap pemanggilan recursive.

Contoh eksekusi:

Qsort [3, 2, 1]

>> qsort [2,1] ++ (3: qsort [])

>> (qsort [1] ++ (2: qsort []) ++ [3]

>> ((qsort [] ++ (1: qsort [])) ++ [2]) ++ [3]

>> (([] ++ [1]) ++ [2]) ++ [3]

>> ([1] ++ [2]) ++ [3]

>> [1, 2, 3]

4. Di antara dua fungsi tersebut, mana yang lebih efisien? Jelaskan mengapa demikian.

primes1 = sieve [2..] where sieve (p:ps) = p : sieve [x | x <- ps, x `mod` p /= 0]
primes2 = [n | n <- [2..], []==[i | i <- [2..n-1], j <- [0,i..n], j==n]]

Penjelasan:

Menurut pemahaman saya, primes2 akan lebih efisien jika dilihat dari space complexity karena tidak terjadi penumpukan pada stack karena adanya recursive call.