# 99 questions/Solutions/91

## From HaskellWiki

< 99 questions | Solutions

Another famous problem is this one: How can a knight jump on an NxN chessboard in such a way that it visits every square exactly once? A set of solutions is given on the The_Knights_Tour page.

```haskell
module Knights where

import Data.List
import Data.Ord (comparing)

type Square = (Int, Int)

-- Possible knight moves from a given square on an nxn board
knightMoves :: Int -> Square -> [Square]
knightMoves n (x, y) = filter (onBoard n)
        [(x+2, y+1), (x+2, y-1), (x+1, y+2), (x+1, y-2),
         (x-1, y+2), (x-1, y-2), (x-2, y+1), (x-2, y-1)]

-- Is the square within an nxn board?
onBoard :: Int -> Square -> Bool
onBoard n (x, y) = 1 <= x && x <= n && 1 <= y && y <= n

-- Knight's tours on an nxn board ending at the given square
knightsTo :: Int -> Square -> [[Square]]
knightsTo n finish = [pos:path | (pos, path) <- tour (n*n)]
  where tour 1 = [(finish, [])]
        tour k = [(pos', pos:path) |
                (pos, path) <- tour (k-1),
                pos' <- sortImage (entrances path)
                        (filter (`notElem` path) (knightMoves n pos))]
        entrances path pos =
                length (filter (`notElem` path) (knightMoves n pos))

-- Closed knight's tours on an nxn board
closedKnights :: Int -> [[Square]]
closedKnights n = [pos:path | (pos, path) <- tour (n*n), pos == start]
  where tour 1 = [(finish, [])]
        tour k = [(pos', pos:path) |
                (pos, path) <- tour (k-1),
                pos' <- sortImage (entrances path)
                        (filter (`notElem` path) (knightMoves n pos))]
        entrances path pos
          | pos == start = 100  -- don't visit start until there are no others
          | otherwise = length (filter (`notElem` path) (knightMoves n pos))
        start = (1,1)
        finish = (2,3)

-- Sort by comparing the image of list elements under a function f.
-- These images are saved to avoid recomputation.
sortImage :: Ord b => (a -> b) -> [a] -> [a]
sortImage f xs = map snd (sortBy cmpFst [(f x, x) | x <- xs])
  where cmpFst = comparing fst
```

This has a similar structure to the 8 Queens problem, except that we apply a heuristic invented by Warnsdorff: when considering next possible moves, we prefer squares with fewer open entrances. This speeds things up enormously, and finds the first solution to boards smaller than 76x76 without backtracking.

Solution 2:

```
knights :: Int -> [[(Int,Int)]]
knights n = loop (n*n) [[(1,1)]]
    where loop 1 = map reverse . id
          loop i = loop (i-1) . concatMap nextMoves

          nextMoves already@(x:xs) = [next:already | next <- possible]
              where possible = filter (\x -> on_board x && (x `notElem` already)) $ jumps x

          jumps (x,y)    = [(x+a, y+b) | (a,b) <- [(1,2), (2,1), (2,-1), (1,-2), (-1,-2), (-2
          on_board (x,y) = (x >= 1) && (x <= n) && (y >= 1) && (y <= n)
```

This is just the naive backtracking approach. I tried a speedup using Data.Map, but the code got too verbose to post.