1 The problem

This is a problem 82 from Project Euler.

2 Definitions

```
import Control.Monad (replicateM)
import Data.List (transpose, minimumBy)
import Data.Ord (comparing)
import Data.Maybe (catMaybes)


type Point = (Int, Int)
type Path = [Point]
```

3 Preparations

Program is going to read a matrix of integers from stdin:

```
unrow :: String → [String]
unrow s = let (l, s') = break (∈ [',', ' ', '\t']) s
  in l : case s' of
    [] → []
    (_ : s'') → unrow s''


parseMatrix :: [String] → [[Int]]
parseMatrix cont = map (map read ∘ unrow) cont


readByLine :: IO [String]
readByLine = do l ← getLine
  let cnt = length (unrow l)
  ls ← replicateM (cnt − 1) getLine
  return (l : ls)


main :: IO ()
main = do lns ← readByLine
  let matrix = parseMatrix lns
  print matrix
  print "Let's solve shit"
  print (minimalSum matrix)
```

4 Solution

```
minimalSum mt = minimumBy (comparing snd) $ map (λi → minimalSumFromPos (i, 0) [] mt) [0 .

minimalSumFromPos :: Point → Path → [[Int]] → (Path, Int)
minimalSumFromPos (x, y) _ mt | y ≡ length mt − 1 = ([(x, y)], (mt !! x) !! y)
minimalSumFromPos (x, y) were mt = ((x, y) : fst mins, current + snd mins)
  where current = (mt !! x) !! y
    mins = if avaiables ≡ []
      then ([], 0)
      else minimumBy (comparing snd) avaiables
    avaiables = catMaybes [up, down, right]
    up = tryPos (x + 1, y)
    down = tryPos (x − 1, y)
    right = tryPos (x, y + 1)
    max_d = length mt
    tryPos (x0, y0) | (x0, y0) ∈ were = Nothing
    tryPos (x0, y0) | x0 < 0 ∨ y0 < 0 = Nothing
    tryPos (x0, y0) | x0 ⩾ max_d ∨ y0 ⩾ max_d = Nothing
    tryPos (x0, y0) = Just $ minimalSumFromPos (x0, y0) ((x, y) : were) mt
```

Okay, that's pretty slow. Let's try dynamic programming.

5 Dynamic Programming

Because formula for solutions with starting cells from the same columns are mutually recursive, we will pre-calculate every column, starting from the most right, and calculate each cell there independently. *minimalSumsForColumn*, called with current column values and minimal sums for the column right from this one, returns minimal sum for each cell in this very column:

```
minimalSumsForColumn :: [Int] → [Int] → [Int]
minimalSumsForColumn current next = map (minimalSum []) [0 .. size − 1]
  where size = length current
    minimalSum visited i = (current !! i) + minimum (avaiables visited i)
    avaiables vs i = catMaybes [up vs i, down vs i, right vs i]
    up vs i | i < 1 = Nothing
    up vs i | i ∈ vs = Nothing
    up vs i = Just $ minimalSum (i : vs) (i − 1)
    down vs i | i ⩾ size − 1 = Nothing
    down vs i | i ∈ vs = Nothing
    down vs i = Just $ minimalSum (i : vs) (i + 1)
    right vs i = Just $ next !! i

minimalSumsForMatrix :: [[Int]] → [Int]
minimalSumsForMatrix mt = foldr minimalSumsForColumn (last mt) (init mt)
```

$$minimalSum\ mt = minimum\ \$\ minimalSumsForMatrix\ (transpose\ mt)$$

Runs interpreted for 14 seconds - cool, ya? Or 1.8 seconds if compiled with -O3.