1.ai.

import Data.Array

fromHive :: [[Int]] -> Array (Int, Int) Int

fromHive hive = array ((0, 0), (size, size)) compact

where

size = length hive – 1

compact = [((i, j), bees) |

(i, row) <- zip [0..] hive,

(j, bees) <- zip [0..] (extend (size + 1) row)]

extend m xs = take m $ (xs ++ repeat 0)

Diagram

Description automatically generated with low confidence

One liner

fromHive lss = let n = length lss in array ((0,0), (n,n)) [((I,j),e)|(ls,I) <-zip lss [0..], (e,j) <- zip ls [0..]]

1.aii.

bees :: Array (Int, Int) Int -> Int

bees hiveArr

= helper hiveArr 0 0 - hiveArr ! (0,0)

where

helper :: Int -> Int -> Int

helper i j

| i == snd (bounds hiveArr)

= hiveArr ! (i, j)

| otherwise

= hiveArr ! (i, j)

+ min (helper hiveArr (i + 1) j) (helper hiveArr (i + 1) (j + 1))

-- Complexity: O(2^n)

Alternative 1.a.ii

bees :: [[Int]] -> Int

bees hive

= helper (length hive – 1, 0)

where

hiveArr = fromHive hive

helper :: (Int, Int) -> Int

helper (0, \_) = 0

helper (i, j) = min (

(calc (i – 1, j)), -- up and left

(calc (i – 1, j + 1)), -- up and right

where

calc (i', j’) = hiveArr ! (i', j’) + helper (i', j’)

-- Complexity: O(2^n)

1.aiii.

bees' :: Array (Int, Int) Int -> Int

bees' hiveArr = table ! (size, size) - hiveArr ! (0,0)

where

table :: Array (Int, Int) Int

table = tabulate ((0,0), (size, size)) (uncurry memo)

memo :: Int -> Int -> Int

memo i j

| i == size = table ! (i, j)

| otherwise = table ! (i, j)

+ min (table ! (i + 1, j)) (table ! (i + 1, j + 1))

size = snd (bounds hiveArr)

-- Complexity: O(mn) where m = height of hive, n = max width of hive.

bees arr = go (n-1, 0)

where

(\_, (n, \_)) = bounds arr

go (0,j) = 0

go (i, j) = min (arr ! upLeft + go upLeft) (arr ! upRight + go upRight)

where

upLeft = (i-1, j)

upRight = (i-1, j+1)

1.b.

bees'' :: Array (Int, Int) Int -> Int

bees'' hiveArr

= helper 0 0 - hiveArr ! (0, 0)

where

helper i j

| i == snd (bounds hiveArr) = hiveArr ! (i, j)

| j == 0 = hiveArr ! (i, j)

+ minimum [helper (i + 1) j,

helper (i + 1) (j + 1),

helper i (j + 1)]

| i == j = hiveArr ! (i, j)

+ minimum [helper ! (j - 1),

helper (i + 1) j,

helper (i + 1) (j + 1)]

| otherwise = hiveArr ! (i, j)

+ minimum [helper i (j - 1),

helper (i + 1) j,

helper (i + 1) (j + 1),

helper i (j + 1)]

bees arr = go (n-1, 0)

where

(\_, (n, \_)) = bounds arr

go (0,j) = 0

go (i, j) = minimum [arr ! upLeft + go upLeft,

arr ! upRight + go upRight

If existLeft then arr ! Left + go left else MaxInt,

If existRight then arr ! Right + go right else MaxInt]

where

upLeft = (i-1, j)

upRight = (i-1, j+1)

ExistLeft = j /= 0

ExistRight = I + j –1 /= n

Left = (I, j –1)

Right =(I, j+1)

2.ai.

find :: [[a]] -> Int -> a

find xss k = concat xss !! k

Without **concat**:

|  |
| --- |
| find :: [[a]] -> Int -> a  find cl ix  = go cl ix 1  where  go [] \_ \_ = error "Index too large."  go (c : cs) ix len  | ix < len = c !! ix  | otherwise = go cs (ix - len) (len + 1) |

2.aii.

toChunks :: [a] -> [[a]]

toChunks xs

= helper xs 1

where

helper :: [a] -> Int -> [[a]]

helper [] \_ = []

helper xs i = x : helper y (i + 1)

where

(x, y) = splitAt i xs

Complexity?

2.aiii.

cons :: a -> [[a]] -> [[a]]

cons x xs = toChunks (x : concat xs)

-- Complexity: O(n)

2.aiv.

find :: [[a]] -> Int -> Int

find xss k

= (xss !! (i - 1)) !! j

where

i = head [n | n <- [1..], n \* (n + 1) / 2 > k]

j = k - i - 1

find' clst i = go clst i 1  
 where  
 go [] \_ \_ = error "asdf"  
 go (xs :xss) offset curSize  
 | curSize > offset = xs !! (offset)  
 | otherwise = go xss (offset-curSize) (curSize+1)

(This is O(sqrt(n)) fyi but saying that isn't worth any marks)

2.bi.

data Queue a = Queue [a] [a]

consQ :: a -> Queue a -> Queue a

consQ q (Queue xs sy) = Queue (q : xs) sy

lastQ :: Queue a -> (Queue a, a)

lastQ (Queue xs []) = lastQ $ Queue [] (reverse xs)

lastQ (Queue xs (y:sy)) = (Queue xs sy, y)

2.bii

Type ChunkList a = [Queue a] -- from paper

cons' :: a -> ChunkList a -> ChunkList a

cons' = f 1

Where

-- Special case for empty queues/base case if last full.

f \_ x [] = [Queue [x] []]  
 -- Base case for if last is not full.

f n x [xs]  
 | n < length xs = consQ x xs

-- Push into head, and let the now superfluous element overflow into tail.  
 f n x (xs : xss) = consQ x xs’ : f (n + 1) carry xss  
 where

(xs’, carry) = lastQ xs

TONY’s Solution:

import Data.Array

hive :: [[Int]]

hive = [[9,7,5,8,3],[1,2,1,3],[2,3,3],[4,6],[1]]

fromHive :: [[Int]] -> Array (Int, Int) Int

fromHive xss = array ((0, 0), (length xss - 1, length xss - 1)) $ concatMap (\(i, xs) -> zipWith (\j x -> ((i, j), x)) [0..] xs) (zip [0..] xss)

bees :: [[Int]] -> Int

bees h = go (sup, 0) - arr ! (sup, 0)

where

sup = length h - 1

arr = fromHive h

go (0, y) = arr ! (0, y)

go (x, y) = arr ! (x, y) + min (go (x - 1, y)) (go (x - 1, y + 1))

bees' :: [[Int]] -> Int

bees' h = table ! (sup, 0) - arr ! (sup, 0)

where

sup = length h - 1

arr = fromHive h

table = tabulate ((0, 0), (sup, sup)) go

go (0, y) = arr ! (0, y)

go (x, y) = arr ! (x, y) + min (table ! (x - 1, y)) (table ! (x - 1, y + 1))

*-- Complexity O(n^2) where n is the length.*

tabulate :: Ix i => (i, i) -> (i -> a) -> Array i a

tabulate (u, v) f = array (u, v) [ (i, f i) | i <- range (u, v)]

data Dir = L | R | B

bees'' :: [[Int]] -> Int

bees'' h = go B (sup, 0) - arr ! (sup, 0)

where

sup = length h - 1

arr = fromHive h

go \_ (0, y) = arr ! (0, y)

go L (x, y) = arr ! (x, y) + minimum ([go B (x - 1, y), go B (x - 1, y + 1)] ++ [go L (x, y - 1) | y > 0])

go R (x, y) = arr ! (x, y) + minimum ([go B (x - 1, y), go B (x - 1, y + 1)] ++ [go R (x, y + 1) | x + y < sup])

go B (x, y) = arr ! (x, y) + minimum ([go B (x - 1, y), go B (x - 1, y + 1)] ++ [go L (x, y - 1) | y > 0] ++ [go R (x, y + 1) | x + y < sup])

find :: [[a]] -> Int -> a

find \_ i

| i < 0 = error "index oob"

find [] \_ = error "index oob"

find ([] : xss) i = find xss i

find ((x : xs) : xss) i

| i == 0 = x

| otherwise = find (xs : xss) (i - 1)

findExample :: Integer

findExample = find [[3],[1,4],[1,5,9]] 4

splitAt :: [a] -> Int -> ([a], [a])

splitAt = flip Prelude.splitAt

toChunks :: [a] -> [[a]]

toChunks = go 1

where

go i xs

| null xs'' = [xs']

| otherwise = xs' : go (i + 1) xs''

where

(xs', xs'') = Main.splitAt xs i

*-- Complexity: O(n).*

toChunksExample :: [[Integer]]

toChunksExample = toChunks [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]

cons :: a -> [[a]] -> [[a]]

cons x xss = toChunks $ x : concat xss

*-- The complexity of "cons" is O(n).*

find' :: [[a]] -> Int -> a

find' \_ i

| i < 0 = error "index oob"

find' xss i = (xss !! rowNum) !! (i - ((rowNum \* (rowNum + 1)) `div` 2))

where

rowNum = floor $ (sqrt (1 + 8 \* fromIntegral i) - 1) / 2

*-- The complexity is O(sqrt n).*

findExample' :: Integer

findExample' = find' [[3],[1,4],[1,5,9]] 4

data Queue a = Q Int [a] Int [a]

deriving (Eq, Ord, Show)

consQ :: a -> Queue a -> Queue a

consQ a (Q lf f lr r) = Q (lf + 1) (a : f) lr r

lastQ :: Queue a -> (Queue a, a)

lastQ (Q \_ [] \_ []) = error "empty queue!"

lastQ (Q lf f \_ []) = let (q, r) = quotRem lf 2

(f', r') = Main.splitAt f q

in lastQ (Q q f' (q + r) (reverse r'))

lastQ (Q lf f lr (a : r)) = (Q lf f (lr - 1) r, a)

type ChunkList a = [Queue a]

cons' :: a -> ChunkList a -> ChunkList a

cons' a qs = go a qs

where

go a [] = [Q 1 [a] 0 []]

go a [q@(Q lf f lr r)]

| lf + lr < length qs = [consQ a q]

| otherwise = let (q', a') = lastQ q in consQ a q' : go a' []

go a (q : qs) = let (q', a') = lastQ q in consQ a q' : go a' qs

*-- Complexity O(sqrt n).*