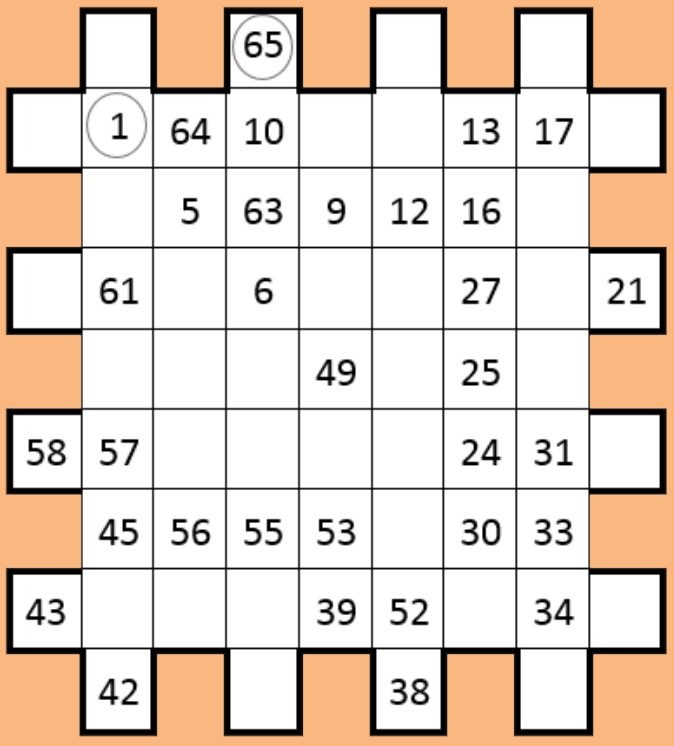
HIDATO



Alejandra Monzón C312
Marié del Valle C311

Tableros de Hidato

Los Hidatos son puzzles lógicos, su objetivo es rellenar el tablero con números consecutivos que se conectan horizontal, vertical o diagonalmente.

Cada tablero de Hidato tiene solución única y existen infinidad de formas de tableros, no obstante todos ellos tienen las casillas cuadradas o hexagonales.

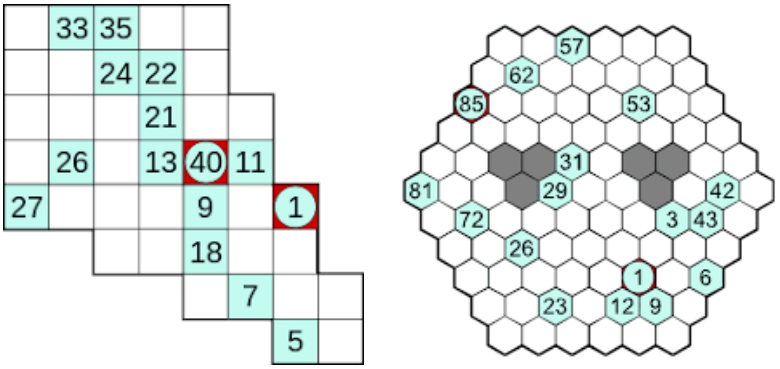


Figure 1: Tablero de casillas cuadradas y tablero de casillas hexagonales

Para representar los tableros en Haskell cada casilla se identificará con un par ordenado `IntegerOrderPair` que indica su posición respecto al rectángulo sobre el que se puede inscribir el tablero, además para cada casilla se conoce su valor, en caso que esté en blanco, dicho valor es 0, por tanto cada casilla es una tupla `(Int, IntegerOrderPair)`.

En general, para cada tablero tenemos casillas con valor fijado, casillas disponibles y debemos conocer si el tipo de casilla es cuadrado o hexagonal, ya que de esto depende la adyacencia entre estas. Esta es la definición de `HidatoBoard`.

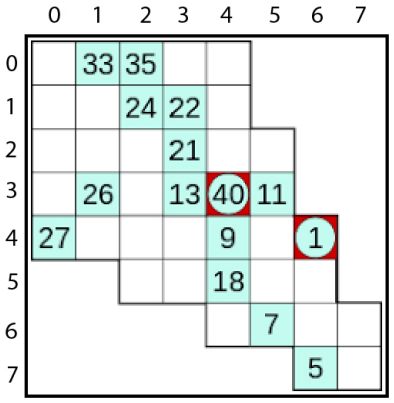
```
94  data IntegerOrderPair = P{xv :: Int, yv :: Int} deriving(Show,Eq,Ord, Read)
95  data CellType = Square | Hexagon deriving(Show,Read, Eq)
96  data HidatoBoard = HidatoBoard { fixed :: (Map Int IntegerOrderPair),
97                                   available:: (Map IntegerOrderPair Int),
98                                   cellt:: CellType} |
99                     Empty deriving(Read,Show,Eq)
```

Para las casillas fijadas (fixed) y las disponibles (available) se han utilizado diccionarios para acceder a los valores correspondientes en menor costo temporal y que sea más eficiente.

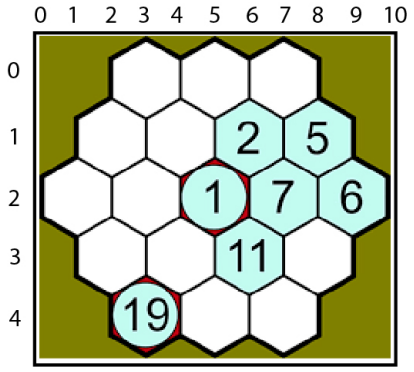
De igual forma si se quiere inicializar un tablero a partir de una lista de tuplas `[(Int, IntegerOrderPair)]` se puede hacer usando la función `fromListtoBoard` que recibe como argumentos la lista y el tipo de casilla del tablero.

Casillas y adyacencia

Como planteamos en la construcción del tablero cada casilla se identifica por un par ordenado de valores enteros, que son su posición respecto al rectángulo que rodea la figura, esto sería de la sigiente forma:



Aquí se muestran como serían los ejes **x;y** para un tablero de casillas cuadradas, por ejemplo, aquí la casilla de valor 1 estaría en el punto (6;4) `P 6 4` y sus adyacentes serían las casillas `P 5 4` , `P 5 5` y `P 6 5` vacías y la casilla `P 5 3` con valor 11.



En los tableros con forma hexagonal se tiene la especificidad de que para un mismo valor de **y** solo se tienen valores de **x** pares todos o impares todos.

Por ejemplo, la casilla de valor 1 está en el punto (5;2) `P 5 2`. Esta tiene como adyacentes a las casillas `P 6 1`, `P 7 2` y `P 6 3` con valores 2, 7 y 11 respectivamente, así como las casillas vacías `P 4 1`, `P 3 2` y `P 4 3`.

Evidentemente, la forma de las casillas y la representación de estas como pares ordenados en un plano, hacen que la relación de adyacencia entre casillas y, por tanto, los movimientos posibles a partir de una casilla específica para llenar el Hidato están estrechamente relacionados con la forma de las casillas del tablero, es por esto que

se define para cada forma de casilla una función que determine la lista de adyacentes a una casilla y otra función que determine si dos casillas son mutuamente adyacentes.

```haskell
sqareAdj :: IntegerOrderPair -> [IntegerOrderPair]
sqareAdj (P x y) = [P (x-1) y, P (x+1) y, P (x-1) (y-1), P (x-1) (y+1),
                    P (x+1) (y-1), P (x+1) (y+1), P x (y-1), P x (y+1)]

hexagonAdj :: IntegerOrderPair -> [IntegerOrderPair]
hexagonAdj (P x y) = [P (x-2) y, P (x+2) y, P (x-1) (y-1),
                    P (x-1) (y+1),P (x+1) (y-1), P (x+1) (y+1)]

isadjSquare :: IntegerOrderPair-> IntegerOrderPair -> Bool
isadjSquare (P x y) (P z w) = abs(x-z) <= 1 && abs(y - w)<=1

isadjHexagon :: IntegerOrderPair-> IntegerOrderPair -> Bool
isadjHexagon (P x y) (P z w) = (abs(x-z) == 1 && abs(y - w) ==1) || ( y == w && abs(x - z) ==2)
```

Solucionador de Hidatos

Para solucionar un tablero de Hidato, se realiza un recorrido en DFS (`walkDFS`) sobre el grafo de las posibles soluciones del tablero. Esto se realiza iniciando en la casilla de número 1, que es la de menor valor, y en cada paso avanzar de la casilla actual con valor x a cada uno de sus adyacentes y considerar cuáles serían tablero válidos tras fijar el valor x+1.

```haskell
walkDFS :: HidatoBoard-> IntegerOrderPair-> Int-> Int-> [HidatoBoard]
walkDFS currentBoard currentp currentv maxv |currentv > maxv = [currentBoard]
                                            | otherwise = [b | (x,y)<- pnBoards, b <-(walkDFS x y (currentv+1) maxv)]
                                               where pnBoards = possibleNextboard currentBoard currentp currentv ((boardAdj currentBoard) currentp)
```

La función `possibleNextboard`, dado un tablero, la casilla actual y el valor que se quiere poner en alguno de los adyacentes a esta casilla, devuelve una lista con los posibles tableros que se pueden generar, tras fijar el valor en cada uno de los adyacentes de la casilla actual en los que sea posible poner dicho número (que no esté inicialmente fijado y que la casilla aun esté vacía). De igual modo si el valor que se desea poner en el tablero está entre los prefijados inicialmente, se comprueba si la casilla que lo contiene es adyacente a la actual, en caso contrario el tablero en el estado actual no conduce a un resultado válido, por lo que ese no era el camino correcto.

Por tanto, dado un tablero de Hidato y el mayor de sus valores, con la función `solve` se obtiene una lista de sus tableros solución, que si es un Hidato válido, esta lista tendrá solo un elemento. De igual forma, si en vez de un tablero Hidato se tiene una lista `[(Int, IntegerOrderPair)]` con ella, el tipo de casilla del tablero y el máximo valor se obtiene la solución del Hidato correspondiente a la lista mediante la función `solveformlist`.

```haskell
solve :: HidatoBoard -> Int-> [HidatoBoard]
solve board max = if (member 1 f) && (member max f) then walkDFS board (f!1) 2 max else []
                  where f = fixed board

solveformlist :: [(Int,IntegerOrderPair)] ->CellType-> Int->[HidatoBoard]
solveformlist list ctype max=if (member 1 f) && (member max f)
                             then walkDFS board (f!1) 2 max
                             else []
                             where
                                board = fromListtoBoard list ctype
                                f = fixed board
```

Generador de Hidatos

La función encargada de generar un hidato válido es `generate`, la cual recibe como parámetros el tipo de celda del tablero, la cantidad de filas y columnas del tablero, el nivel de dificultad y la cantidad de obstáculos.

Los parámetros tipo de celda (`CellType`), nivel de dificultad (`DifficultMode`) y porciento de obstáculos (`ObstaclesPercentage`) son tipos definidos por nosotros.

```haskell
data DifficultyMode = Easy | Medium | Difficult deriving(Show, Read)
data ObstaclesPercentage = O0 | O15 | O25 | O45 deriving(Show, Read)
```

Los constructores del nivel de dificultad son : `Easy`, `Medium` y `Difficult`. Cada uno especifica un porciento de casillas que estarán disponibles en el tablero final. Los porcientos son 30, 45, 60 respectivamente.

Por otro lado, los constructores de la cantidad de obstáculos son `O0`, `O15`, `O25` y `O45`, donde cada uno representa el porciento de casillas del tablero que serán obstáculos. Esto garantiza la diversidad de formas de los tableros generados.

Generar plantilla válida

La función `generate`, a partir del tipo de celda, crea una lista de la forma `[(Int, IntegerOrderPair)]`, donde el primer valor de cada tupla es 0 y el segundo las posiciones de una matriz (`rows x cols`).

```haskell
generate :: CellType -> Int -> Int -> DifficultyMode -> ObstaclesPercentage -> HidatoBoard
generate Square rows cols difficulty obstacles = getBoard
                                                 (fromListtoBoard [(x, P y z) | x <-[0], y <- [0,1..(rows-1)], z <- [0,1..
generate Hexagon rows cols difficulty obstacles = getBoard
                                                  (fromListtoBoard ([(x, P y z) | x <-[0], y <- [0,2..(rows-1)], z <- [0,2..
```

Resolver plantilla para obtener hidato resuelto

A partir de la plantilla creada, se llama a la función `fillBoard`, que es la encargada de rellenar el tablero hasta alcanzar el valor del Hidato. Dicha función recibe como parámetros la plantilla del tablero, el número actual a insertar, la possible posición en la que se puede insertar y el valor máximo que se debe alcanzar.

El valor máximo es el valor del hidato y es el resultado de la función `getMax` con los parámetros `rows*cols` y `ObstaclesPercentage`, la cual calcula la cantidad de obstáculos que debe tener el tablero y devuelve el máximo de casillas que debe tener el mismo.

```haskell
getMax :: ObstaclesPercentage -> Int -> Int
getMax O0 max = max
getMax O15 max = max - (div (max * 15) 100)
getMax O25 max = max - (div (max * 25) 100)
getMax O45 max = max - (div (max * 45) 100)
```

La función `fillBoard` construye un hidato resuelto de forma recursiva. En cada llamado, intenta poner el número actual `num` en la posición de tablero `pos`. Si es posible realizar la operación se pasa al próximo número a insertar, que sería el sucesor del actual; en caso contrario, se escoge otra posición adyacente a la posición del antecesor del número actual y se llama nuevamente a la función.

```haskell
fillBoard :: HidatoBoard -> Int -> IntegerOrderPair -> Int -> HidatoBoard
fillBoard board num pos max | num == max = HidatoBoard (fixed newBoard) Map.empty (cellt newBoard)
                            | noAdj board ((boardAdj board) pos) 0 = Empty
                            | otherwise = possibleBoard
                            where
                                newBoard = HidatoBoard (Map.insert num pos (fixed board)) (Map.delete pos (available board)) (cellt board)
                                newNum = num + 1
                                possibleBoard = nextPos 0

                                nextPos index | index == (length adj) = Empty
                                              | not (member newPos (available newBoard)) = nextPos (index + 1)
                                              | solution /= Empty = solution
                                              | otherwise = nextPos (index + 1)
                                              where
                                                adj = getRandomAdj ((boardAdj board) pos) (lengthAdjList (cellt board))
                                                newPos = adj !! index
                                                solution = fillBoard newBoard newNum newPos max
```

Proceso de Eliminación

Una vez obtenido un tablero con una solución válida, se llama a la función `removeElemsBoard`, la cual recibe como parámetros el tablero resuelto como diccionario, el tablero resuelto como lista, el tamaño de la lista, la cantidad de elementos a remover, la cantidad de elementos removidos y el valor del hidato.

La cantidad de elementos a remover depende del nivel de dificultad y se calcula con la función `getDifficulty`, cuyos parámetros son el `DifficultMode` y el valor del hidato.

```haskell
getDifficulty :: DifficultyMode-> Int -> Int
getDifficulty Easy max = (div (max * 30) 100)
getDifficulty Medium max = (div (max * 45) 100)
getDifficulty Difficult max = (div (max * 60) 100)
```

La función `removeElemsBoard`, remueve si es posible la cantidad de elementos `numberOfElems` del tablero resuelto. Para ello, se escoge un elemento aleatorio del tablero, se remueve del tablero y se llama a la función `solve`, pasándole el tablero sin el elemento fijado como parámetro. Si el resultado de la función `solve` es válido, entonces se quita ese elemento y se procede a buscar otro elemento a quitar; si no es válido, se deja el elemento en el tablero e igual se pasa a elegir otro elemento a remover. La función termina cuando se quitaron a lo sumo la cantidad de elementos `numberOfElems` y el tablero resultante tiene solución única.