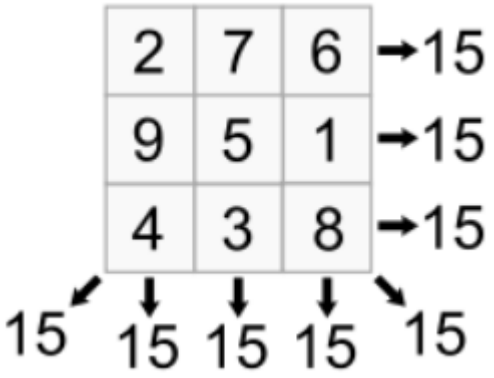
UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Progetto PA: Haskell

"Magic Square"

Boschetti Kevin, matricola n. 1057400

Magic Square e obiettivi del software



Un quadrato magico è una matrice NxN formata da numeri interi nell'intervallo [1 .. N^2] in cui la somma di ogni riga, colonna e diagonale di lunghezza N è sempre uguale e corrisponde alla costante magica. Inoltre ogni numero è presente al massimo una sola volta.
Obiettivo dell'applicativo sviluppato è calcolare il costo minimo, o distanza minima, tra una matrice scelta dall'utente e il quadrato magico più vicino. Con distanza si intende il valore |a - b|, con a numero intero della matrice di input e b numero intero di un quadrato magico. La somma di tutte queste differenze dà il costo per convertire la matrice scelta. Per semplicità ho scelto di considerare matrici 3x3.

Come prima cosa ho dichiarato un custom type Square che sarebbe stato comodo in seguito dato che avrei lavorato spesso con liste di liste di interi, cioè con le matrici.

```
type Square = [[Int]]  -- ogni magic square è una lista di liste di Int
```

In seguito, al fine di fare varie prove durante la stesura del codice, ho creato la funzione *printSquare* apposita che permettesse di stampare a terminale una matrice potendo riconoscere agevolmente righe e colonne. Il processo che segue consiste nel mappare ogni riga, mappare poi ogni numero intero rendendolo una stringa, unire ogni elemento separandolo con uno spazio, separare le stringhe ottenute.

```
printSquare :: Square -> IO ()
printSquare = putStrLn . unlines . map (unwords . map show)
```

All'inizio ho pensato di utilizzare un approccio brute force, dunque di calcolare tutti i possibili quadrati magici 3x3 con interi nell'intervallo [1 .. 9]. Per fare ciò ho utilizzato tre funzioni:

- *chop* permette di passare da una lista di interi ad una matrice, scegliendo con il parametro n quanto sarà lunga ogni riga;
- *isMagic* invece serve per riconoscere un quadrato magico tra le 362880 possibili permutazioni, in particolare ne esistono 8;

- infine *allMagicBruteForced* combina le due funzioni precedenti andando a fare il calcolo effettivo

```haskell
chop :: Int -> [Int] -> [[Int]]
chop _ [] = []  -- se arrivato alla fine return lista vuota
chop n xs = take n xs:chop n(drop n xs)  -- ricorsione, toglie i primi n elementi dalla lista e prosegue

isMagic :: Square -> Bool
isMagic s =
    (==1) $ -- check se lista di somme = 1
    length $
    nub $
    concat [map sum s,  -- righe
            map sum $ transpose s,  -- colonne
            [sum $ map (uncurry (!!)) $ zip s [0..]],  -- diagonale principale
            [sum $ map (uncurry (!!)) $ zip (map reverse s) [0..]]]  -- diagonale secondaria

allMagicBruteForced :: [Square]
allMagicBruteForced = filter isMagic $ map (chop 3) $ permutations [1..9]
```

Più nello specifico:

- *chop*, una volta costruita la prima riga della matrice, toglie ricorsivamente i primi n elementi dalla lista e prosegue finché questa non si esaurisce
- *isMagic* calcola la somma degli interi in ogni riga, colonna e sulle due diagonali creando una lista di quattro valori. Dopodiché elimina i duplicati dalla lista e se risulterà avere lunghezza pari a uno allora sarà riconosciuto un quadrato magico. Al fine di questo processo si usano varie funzioni:
    - *transpose* trasforma ogni riga in una colonna
    - *!!* permette di scegliere la posizione del valore della lista da estrarre
    - *zip [0 ..]* unisce due liste, quindi con [0 ..] avrò una lista infinita a partire da 0, otterrò così delle coppie (numero, riga matrice)
    - *uncurry* converte una funzione di modo che accetti delle coppie di elementi e non un solo parametro
    - *nub* rimuove gli elementi duplicati di una lista

Per concludere la prima versione che ho sviluppato resta da calcolare l'effettiva distanza/costo tra la matrice di input e il quadrato magico più vicino. Le funzioni *distance* e *solve* servono proprio a questo scopo.
Il metodo *distance* è composto a sua volta dall'utilizzo di varie funzioni:

- *concat* rende una lista di liste una lista unica
- *zipWith (-)* sottrae elemento per elemento
- *abs* applica il valore assoluto
- *map* rimappa la lista secondo un criterio
- *sum* effettua sommatoria di tutti i valori assoluti delle differenze

```haskell
distance :: Square -> Square -> Int
distance s1 s2 = sum $ map abs $ zipWith (-) (concat s1) (concat s2)
```

Tramite *solve* è possibile giungere alla soluzione definitiva calcolando qual è il costo minimo per trasformare la matrice data in input:

- *map (distance s) allMagicBruteForced* calcola tutte le possibili distanze dalla matrice originale ai possibili magic square
- *minimum* prende la distanza minore

```haskell
solve :: Square -> Int
solve s = minimum $ map (distance s) allMagicBruteForced
```

Per rendere il programma interattivo manca la parte del main. Ho scelto di far richiamare da subito la funzione *computeDistance* che richiede all'utente di inserire la lista di numeri interi appartenenti alla matrice che vuole dare in input al programma. La lista viene letta dal programma sfruttando il metodo *getIntList*, viene poi convertita in una matrice e viene applicata *solve*. Infine tramite *continueGameOrNot* viene chiesto all'utente se vuole calcolare un'altra distanza.

```haskell
getIntList :: IO [Int]
getIntList = fmap read getLine

computeDistance :: IO ()
computeDistance = do
    putStrLn "----------------------------------------------------------------"
    putStrLn "Inserisci la lista di numeri della matrice 3x3 :"
    putStrLn "es. [1,2,3,4,5,6,7,8,9]"
    inputList <- getIntList
    let minDistance = solve $ chop 3 inputList
    putStrLn $ "Minima distanza dal magic square più vicina : " ++ show minDistance
    continueGameOrNot

continueGameOrNot :: IO ()
continueGameOrNot = do
    putStrLn "----------------------------------------------------------------"
    putStrLn "Vuoi calcolare un'altra distanza? [Y/N]: "
    choice <- getLine
    case choice of
        "Y" -> computeDistance
        "Yes" -> computeDistance
        "y" -> computeDistance
        _ -> return ()

main :: IO ()
main = computeDistance
```

Durante l'esecuzione del programma si può notare che la soluzione viene data dopo qualche secondo. Questo è dovuto al calcolo brute force di tutti i possibili quadrati magici che, data l'impossibilità di haskell di salvare degli elementi calcolati da una funzione, viene svolto da zero ad ogni richiesta. Per risolvere questo problema ho scelto di inserire nel programma la variabile *magic* a cui corrisponde un quadrato magico effettivo. A partire da questo ho calcolato poi i restanti sette magic square per il caso preso in esame, ovvero matrici 3x3. Questi quadrati si trovano manipolando *magic* tramite rotazioni e specchiano la matrice stessa. Vengono quindi in aiuto altre due funzioni:

- *rot90* ruota di 90° gradi la matrice facendo diventare le righe delle colonne (*transpose*) e invertendo ogni riga (*map reverse*)
- *refl* specchia invece la matrice orizzontalmente

Il metodo *allMagic* calcola poi tutti i magic square iterando rotazioni e riflessioni (*iterate*) e prendendo solo i primi quattro risultati (*take 4*).

```
magic :: Square
magic = [[8, 1, 6],
         [3, 5, 7],
         [4, 9, 2]]

rot90 :: Square -> Square
rot90 = map reverse . transpose

refl :: Square -> Square
refl = transpose

allMagic :: [Square]
allMagic = concat [ take 4 $ iterate rot90 magic,
                    take 4 $ iterate rot90 $ refl magic ]
```

Per ridurre sensibilmente il tempo di esecuzione del programma occorre modificare *solve* al fine di fargli richiamare *allMagic* per fare il calcolo dei quadrati magici nel modo più light appena definito.

```
solve :: Square -> Int
solve s = minimum $ map (distance s) allMagic
```