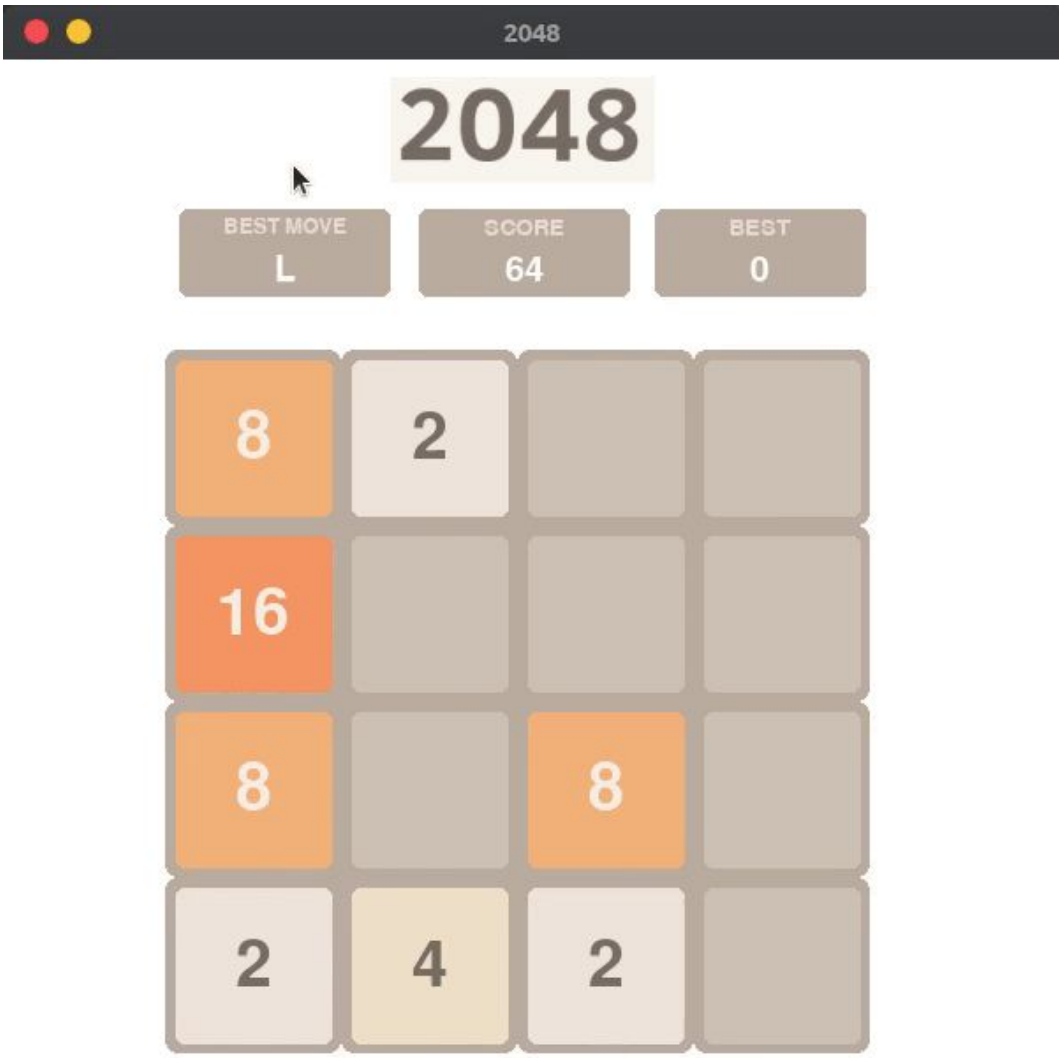
# POPL ASSIGNMENT

## 2048 Game Design



# Group Members

1. AETURI NAGA PAVAN KALYAN REDDY ID: 2018A7PS0212G
2. M V SHASHANK ID: 2018A7PS0734G
3. VOOKA RAM KISHAN ID: 2018A7PS0201G

# Contents

# Matrix.py

## Class Matrix

### Functions

```
def __init__(self,dimX=4,dimY=4):
```

Input:  int dimX , int dimY
output: None
attributes :

**mtx** int[4][4]

**length** **int**

**nextBestMove** char

**gameover** bool

**maxTileVal** int

**winner** bool

**score** int

**d_score** int

**u_score** int

**l_score** int

**r_score** int

**d_mtx** int[4][4]

**u_mtx** int[4][4]

**l_mtx** int[4][4]

**r_mtx** int[4][4]

**tiles** int[4][4]

**Bmem** list of Nodes

**Bcnt** int

This function is a reserved method. It is a constructor.
Arguments: self which is used to access all instances of the class and dimx, dimy tell the dimension of the matrix of the game and has a default value of 4 for both. Attributes of the function are length, height which are the dimensions of the game board. **maxTileValue** which stores the maximum of the game at a particular point. **score** which stores the score of the user. **mtx** is a 2 dimensional array which stores the integer value of the game after every move. **tiles** which is a 2 dimensional matrix which stores the **Tile** object which has the value, **color** and dimension of the tile.

```
def buildTiles(self,win):
```

## Input: GUI element window

## Output: None

This function instantiates the **tile** class objects and stores them in a 2 dimensional array called tiles.

```
def changeTileNum(self,i,j,num,win):
```

Input: int  i(row index), int  j(column index),int num(value of the tile), GUI element window
Output: None

This function is used to change the tile  value which is reflected in the gui of the game.

```
def add_new_tile(self):
```

Input: None
Output: None

This function helps in adding new tile to the  matrix and in turn which is reflected in the gui.

```
def updateGui(self,win):
```

Input: GUI element window
Output: None

It updates the gui of the game when a user makes a particular move.

```
def maxTileNum(self):
```

Input: None
Output: None

This function helps in getting the maximum tile value when called and it stores the maximum value in maxTileValue attribute.

```
```

```
def emptyAvailable(self):
```

Input: None

Output: bool

This function returns **True** if an empty place is found in the grid of the game else **False**.

```
def checkEqualConsecutiveTiles(self):
```

Input: None

Output: bool

This function checks whether two tiles in adjacent are equal. It helps in merging the adjacent tiles if they are equal after a move. Returns **true** if they are equal else returns **false**.

```
def checkAddandUpdate(self,win):
```

It stores the current move made and updates the **max tile value.** If the max value of 2048 is not reached it adds a random tile and updates the gui. If no empty spaces are left then the gameover attribute is made **true** and the **winner** attribute as **false** and if 2048 is reached then the **winner** attribute is made true and the game stops.

```
def storeMoves(self):
```

Input: None

Output: None

stores the current values of the grid in an array bmem. The array has size 5 which stores the previous states of the grid.

```
def restore(self,win):
```

## Input:  GUI element window

## Output: None

It is called when the user presses space to go back to previous state and this function updates the gui and the mtx attribute which stores the integer values of the current state.

```
def leftMove(self,win):
def rightMove(self,win):
def upMove(self,win
def downMove(self,win):
```

## Input: GUI element window

## Output: None

These functions update the mtx attribute and update the gui based on the move made.

```
def bestMove(self):
```

## Input: None

Output: None

This function computes all four (left, right, up, down) moves of next state and updates the corresponding class variables (l_mtx, l_score, etc.,). After computing it estimates the next best move based on ***(sum of tiles)/(no. of non-zero tiles).***

# moves.py

# Functions

```
def stackLeft(mtx):
```

Input: int mtx (2-dimensional array)
Output: int nmtx (2-dimensional array)

This function shifts the value to the left in the grid and it stops when it sees another tile or extreme end of the matrix. After performing this action it returns a 2-dimensional matrix of size 16 called **nmtx**.

```
def combineLeft(mtx,score):
```

Input:  int mtx (2-dimensional array), int score

Output: int nmtx (2-dimensional array), int score

This function combines the adjacent tiles in the row of the matrix if their values are equal and also calculates the **score** and returns this matrix **nmtx** and the calculated **score**.

```
def transpose(mtx):
```

Input: int mtx (2-dimensional array)

Output: int nmtx (2-dimensional array)

This function performs a similar operation to the transpose of a matrix where the rows and columns are interchanged. This function helps in right, up, down movement as the matrix at current state can be transposed and **stackLeft** function can be used for all these three operations along with an additional function.

```
def rotateHorizontal(mtx):
```

Input: int mtx (2-dimensional array)

Output: int nmtx (2-dimensional array)

This function rotates the rows of the matrix horizontally and this function is used in performing one of the 4 operations on the matrix which is right, up, down using **stackLeft** and **combineLeft** .

```
def getLeft(nmtx,score):
```

Input: int nmtx (2-dimensional array), int score

Output: int nmtx (2-dimensional array), int score

It uses **stackLeft** to push the matrix elements of the argument given to it to left and **combineLeft** to add adjacent values and again **stackLeft** to get the final state of the matrix after performing left operation and returns this matrix **mtx** and the **score** which is returned by the **combineLeft** function.

```
def getRight(nmtx,score):
```

Input: int nmtx (2-dimensional array), int score

Output:  int mtx (2-dimensional array), int score

This function uses **stackLeft**, **rotateHorizontal** and **combineLeft** to get the final state of the matrix given in argument after right operation and returns this matrix and the **score** calculated by the **combineLeft function**.

```
def getUp(nmtx,score):
```

Input: int nmtx (2-dimensional array), int score
Output: int mtx (2-dimensional array), int score

getUp uses the same above mentioned functions to change the state of the matrix in argument after up operation and returns this matrix and the **score** calculated by the **combineLeft function.**

```
def getDown(nmtx,score):
```

Input: int nmtx (2-dimensional array), int score

Output: int mtx (2-dimensional array), int score

getDown is the same as getUp, updates the argument matrix in down operation and returns matrix and **score**.

# Node.py

## Class Node

The Node class is a basic storing instance of memory in game history and can be used by an intelligent agent for creating an instance of the present state and building a tree of nodes to predict future steps . This class stores matrix instances and also provides insights as score, sum of non-empty tiles,different number counts,maxTile value ,parent node of a state.The constructor is also provided depth to spawn children according to the parameter given.

# Functions

```
def __init__(self,mtx,score=0,parent=None,depth=0,builddepth=0):
```

Input: int[4][4] Matrix ,int score,Node parent_node, int depth

output: None

attributes :

**mtx** int[4][4]

**depth** int

**totalSum** int

**score** int

**children** dictionary of Nodes

**parentNode** Node

**MaxTile** int

**numberCount** dictionary of ints(number of similar ints in matrix)

This is the constructor of node class which initialises the matrix,score,parent with given input parameters and sum of non-empty tiles,different number counts,maxTile value with calculated values according to the given matrix.it also stores the depth of the node in the tree and spawns children and their successors till the given **builddepth** and initiate the children dictionary attribute

```
def gettSum(self):
```

## Input: None

## Output: None

This functions helps to fetch **`totalSum`** attribute by calculating the sum of all non-empty tiles in the matrix

```
def getMaxTile(self):
```

## Input: None

## Output: None

This functions helps to fetch **`MaxTile`** value attribute by calculating the sum of all non-empty tiles in the matrix

```
def getListTiles(self):
```

## Input: None

## Output: None

This function return the list of number of tiles with same values and sets to **`numberCount`** attribute in the constructor

```
def buildChildren(self,depth=0):
```

Input: int depth=0 (default)

Output: None

This function spawns the children nodes after appropriate moves and stores them and returns a dictionary which is set to **`children`** attribute in the constructor and can be used anytime to spawn the children and their successors till the given **`depth`**

```
def insertRandomTile(self,nmtx,depth):
```

Input: int[4][4] nmtx ,int depth

Output: list of int[4][4] mtx

This function takes a state matrix **`nmtx`** and returns a list of nodes ,i.e all possible nodes after random tile insertion at any empty tile with value 2 or 4 and recursively builds its children to the given **`depth`**

```
def getLeftp(self,depth):
```

Input: int depth

Output: list of int[4][4] mtx

This function takes **depth** as attribute and returns list of children nodes possible after **left** move of present node state of matrix by invoking **insertRandomTile** on the transition matrix ( new matrix after left move )

```
```

```
def getRightp(self,depth):
```

Input: int depth

Output: list of int[4][4] mtx

This function takes **depth** as attribute and returns list of children nodes possible after **right** move of present node state of matrix by invoking **insertRandomTile** on the transition matrix ( new matrix after Right move )

```
def getUpp(self,depth):
```

Input: int depth

Output: list of int[4][4] mtx

This function takes **depth** as attribute and returns list of children nodes possible after **Up** move of present node state of matrix by invoking **insertRandomTile** on the transition matrix ( new matrix after Up move )

```
def getDownp(self,depth):
```

Input: int depth

Output: list of int[4][4] mtx

This function takes **depth** as attribute and returns list of children nodes possible after **down** move of present node state of matrix by invoking **insertRandomTile** on the transition matrix ( new matrix after Down move )

# Tree.py

## Class Tree

## Functions

```
def __init__(self,mtx):
```

Input: int[4][4] Matrix

Output: None

Attributes :

**root** Node

**presentNode** Node

**totalSum** int

The tree class is useful for initialization of a tree with a matrix and is handy to up and down the tree using its functions. It contains **root** and **presentNode** **.root** stores the root node .**presentNode** stores the present visiting node of tree .At the first **presentNode** and **root** ,both are same

```
def goToParent(self):
```

Input: None

## Output: None

The function sets the **`presentNode`** attribute to its parent Node if its is not **`None`**

```
def goTochild(self,S,n):
```

## Input: None

## Output: None

The function sets the **`presentNode`** attribute to the **n**th child node of **S** sub-group, **S** can be of ‘U’,’R’,’D’,’L’ representing Up,Right,Down,Left and **n** should be less than the number of possible states in S sub-group.