Aj Kurzman
304647527
Rombach 2A

Flood-It!

THE GAME:

A single play-through involves a 12x12 grid of squares, each square filled with one of six colors. The objective is simple: flood the board with a single color before you are out of moves. You start in the top left corner. Each move represents the changing of your squares' colors. Once you switch the color of your square(s), you acquire all adjacent squares of the same color. If the board is not one complete color before the moves run out, you lose. If you get the board flooded in as many or less moves, you win. Usually, the game is played with a fixed number of moves. However, for this project we were tasked with designing a greedy algorithm that completes the game to represent allowed moves.

THE CODE:

Start with a function declaration that takes difficulty as a parameter. In my case, I gave the user three options: easy, medium, or hard. Set up an initial if statement that makes sure it is a valid parameter first:

```
if d!=valid_parameter_option1 and d!=valid_parameter_option2 ….
    print('Please enter a valid difficulty')
    return
```

Next, set up your initial board with the given difficulty parameter. I chose 8x8 and 4 colors for easy, 10x10 and 5 colors for medium, and 12x12 and 6 colors for hard. In order to properly monitor which color is in which square and whether or not the square has been "activated" in gameplay to be changed with each move, two separate nxn arrays are required. Once the main square is created, separate in to the specified number of boxes. Once that has been done, each square needs to be filled with a color and our array needs to be updated to reflect this:

```
k=500/(n) #I chose dimensions to be 500x500
    #creates the nxn board
    for i in range(0,n):
        w.create_line(0,k,500,k)
        k+=500/(n)
    l=500/(n)
    for j in range(0,n):
        w.create_line(l,0,l,500)
        l+=500/(n)
    m=500/(n)
```

```
for i in range(n):  #gives random color to
        for j in range(n)  #each square
            gcolor=(randint(0,3))

for i in range(n):
        for j in range(n):
            a=gcolor[i][j]
            w.create_rectangle(i*m,j*m,(i+1)*m,(j
+1)*m, fill=colors[a])
            #creates rectangle to show color
```

Now to finish the layout we need some buttons. Add a button per color to the bottom of the screen (or wherever you see fit). These will be used to change the color of activated squares.

```
        b1=Button(master,bg='orange',width=8,command=lambda:bfill(0,tracker,gcolor,8,colors))
        b1.pack(anchor='s',side='left',expand=True) #lambda function added once our bfill function is set
        #on master canvas, orange colored, width specified, and command is to fill which we'll define
later
```

Aj Kurzman
304647527
Rombach 2A

Add however many buttons are needed for said difficulty. Once the layout is set for all three difficulties, it is time to get gameplay working. In order to do this we must define our bfill function used by each button. The function will initially sweep the tracker array and find any box that has been activated (i.e. a value of 'TRUE'). It will then change the fill to the corresponding color of the button:

```
def bfill(c,tracker,gcolor,n,colors):
    m=(500/n)
        for i in range(0,len(tracker)):
            for j in range(0,len(tracker)):
                if tracker[i][j]==1:
                    w.create_rectangle(i*m,j*m,(i+1)*m,(j+1)*m, fill=colors[c])
```

Next it must then test all adjacent boxes to the original activated area. The purpose of this is to see if they too must be activated or not. If these adjacent boxes have the same color to our newly colored area, they too must be flipped on to ensure next button push they will change in colors as well:

```
for i in range(0,len(tracker)):  #repeated for all four directions to make sure each box is visited
            for j in range(0,len(tracker)):
                if tracker[i][j]==1:
                    if i!=(len(tracker)-1):
                        if gcolor[i+1][j]==c:
                            tracker[i+1][j]=1
##It works because if an adjacent is turned on, on next run through of the for loop it will take that as on and search that boxes adjacent, making sure all boxes are visited.
```

Lastly, we go back to our previous squares and change their value in the color array to reflect that of the button:

```
for i in range(len(tracker)):
            for j in range(len(tracker)):
                if tracker[i][j]==1:
                    gcolor[i][j]=c
```

Now that our fill is set up and you can flood the board in however many moves you want, we can add the move limit function. This function will solve the given board based off a greedy algorithm. Whichever color has the most adjacent squares to our activated area will be the next move until the board is solved. If two colors have the same amount of adjacent squares, tie goes to the color present first in the color list. We start with a simple declaration:

```
def determinemoves(n,tracker,gcolor):
    movecounter=0  #keeps track of the algorithms moves
    tr=tracker.copy() #copies tracker so we don't overwrite the one given
    gc=gcolor.copy() #copies gcolor so we don't overwrite the one given
```

Now we must determine which move it will make next. In similar fashion to our fill function, the algorithm will search through all adjacent boxes to the activated area. The difference is it does this for every possible color, so before the searching we add a simple for loop:

```
 count=[0,0,0,0,0,0]
 most=0
#runs through a for loop of every potential color in color list
 for c in range(len(count)):
```

On each iteration through this for loop we must make another copy of the tracker array to keep each time from altering our original copy. Then if a square is activated and a neighbor is not, if that neighbor==c (is the color we are looking for) then we add 1 to the corresponding index in count. Then we just determine which of our next moves is best:

```
  if count[c]>most:
     most=count[c]
     trcopy=tt.copy() #creates new copy to be assigned
     mcolor=c
#sets all active squares to said color and changes finished to False if there
#is an inactive square
for i in range(len(tt)):
   for j in range(len(tt)):
     if tr[i][j]==1:
        gc[i][j]=mcolor
     if tr[i][j]==0:
        finished=False  #keeps our while loop running as long as there is a 0
```

Set our original tracker copy to the trcopy(representing tracker of next best move), increment local move counter int, and repeat until board is solved. Then return the move counter.
Now that we have our moves to beat and a working gameplay interface, we can add in the final touches. A scoreboard in the bottom that updates with every move, a restart button, and a few more lines so the game can detect a win or a loss. Starting with the scoreboard, in the game declaration under each difficulty we can add a textbox just like we do a button:

```
allotedmoves=Text(master,width=8,height=2) #text box set on bottom left
allotedmoves.pack(anchor='s',side='left',expand=True)
scorebox='Moves:\n'+'0 / '+str(movemax)  #movemax being the maximum amount of moves calculated
allotedmoves.insert('end',scorebox)   #by the algorithm
```

Since each button push represents a single move by the user, our bfill function is the ideal spot to keep track of user moves and determine a winning or losing move. At the end it will use the global move counting variable and weigh it against the max move variable to change the text box accordingly:

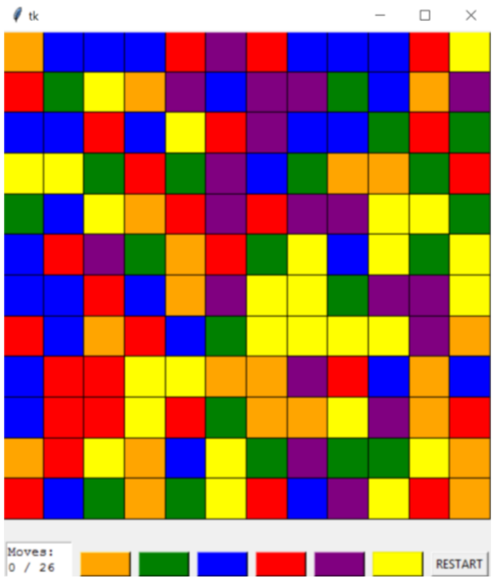
Aj Kurzman
304647527
Rombach 2A

```
#if your moves are less than max allowed and there is no 0 in tracker: YOU WIN!!
#checks win before loss because if you win on the final allowed move it is still a win
if movetracker<=movemax and 0 not in tracker:
        allotedmoves.insert('end','YOU\nWIN!!')
        movetracker-=1
    #if your moves are greater than or equal to max moves: YOU LOSE!!
    elif movetracker>=movemax:
        allotedmoves.insert('end','YOU\nLOSE!!')
        return
    else: #updates scoreboard to reflect movetracker
        scorebox='Moves:\n'+str(movetracker)+' / '+str(movemax)
        allotedmoves.insert('end',scorebox)
```

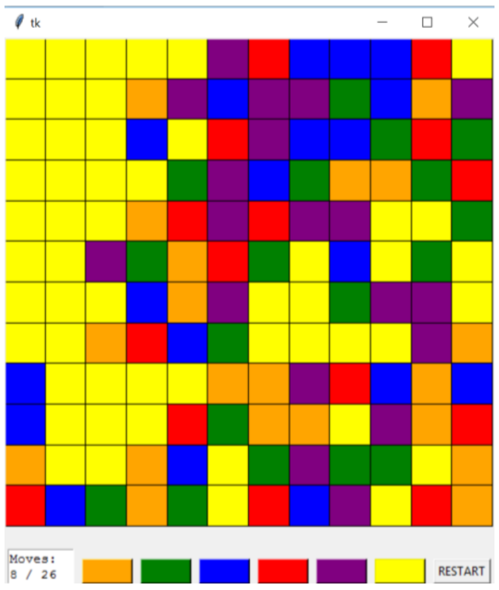
Lastly, a restart button is needed so users can reset the game at any time. Initialize a button for each difficulty same as before. The command line for this one will, however, be a function called newgame that takes the same parameter d as given to initial floodit declaration. This new game function will then destroy the old canvas and initialize a new floodit game of same difficulty, completely from scratch:

```
def newgame(d):
    master.destroy()
    floodit(d)
```

And there you have it, a fully functional and hopefully fun rendition of a python-based Flood-it! Game. Enjoy!

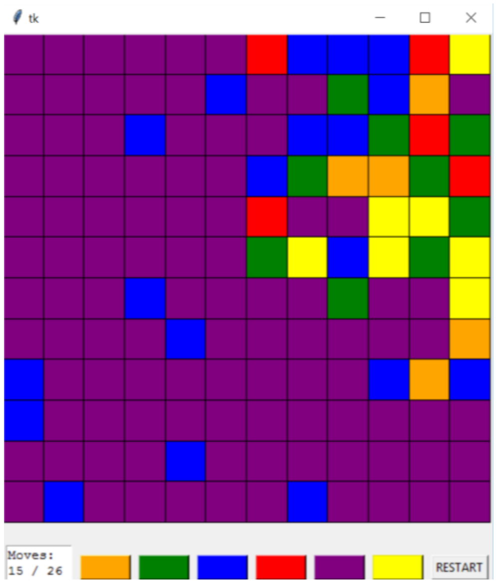


```
Moves:
0 / 26
```

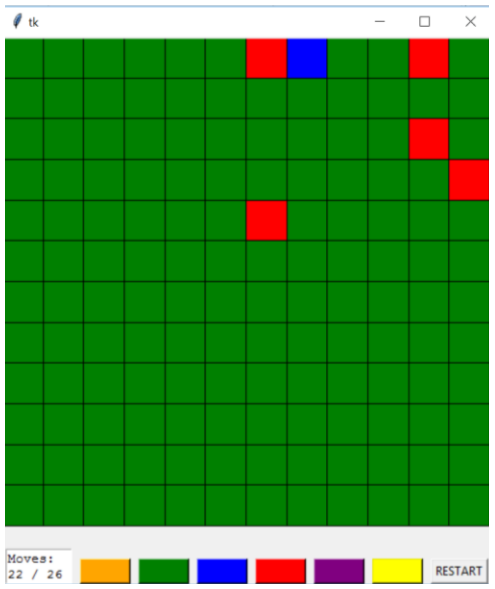


```
Moves:
8 / 26
```

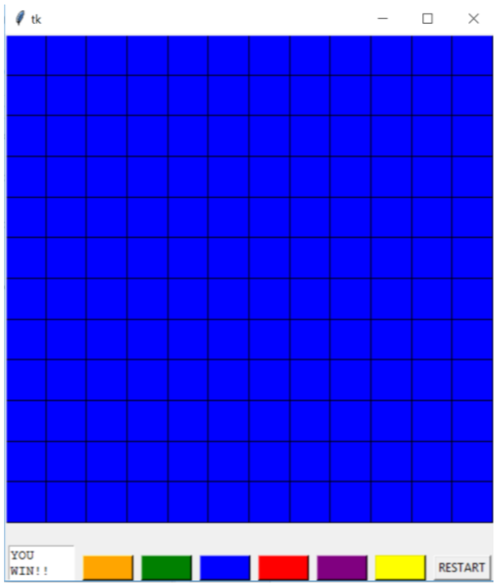
Aj Kurzman
304647527
Rombach 2A



Moves:
15 / 26



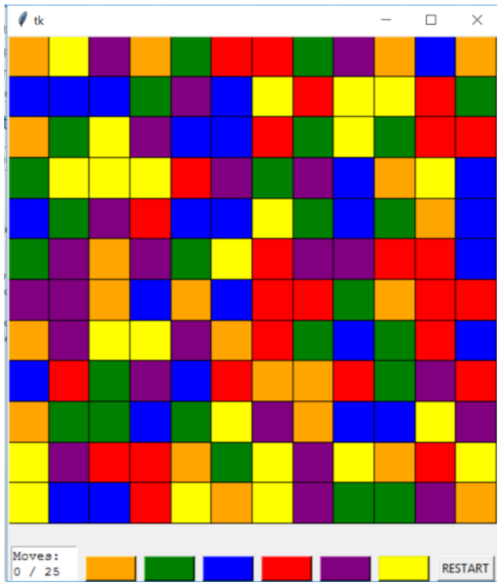
Moves:
22 / 26



YOU
WIN!!

After

Restart



Moves:
0 / 25