➢ Water Jug Problem:

```
from collections import defaultdict
jug1, jug2, aim = 4, 3, 2
visited = defaultdict(lambda: False)
def waterJugSolver(amt1, amt2):
    if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):
        print(amt1, amt2)
        return True
    if visited[(amt1, amt2)] == False:
        print(amt1, amt2)
        visited[(amt1, amt2)] = True
        return (waterJugSolver(0, amt2) or
                waterJugSolver(amt1, 0) or
                waterJugSolver(jug1, amt2) or
                waterJugSolver(amt1, jug2) or
                waterJugSolver(amt1 + min(amt2, (jug1-amt1)),
                amt2 - min(amt2, (jug1-amt1))) or
                waterJugSolver(amt1 - min(amt1, (jug2-amt2)),
                amt2 + min(amt1, (jug2-amt2))))

    else:
        return False

print("Steps: ")
waterJugSolver(0, 0)
```

➢ 8-puzzle Problem

```
class
Node:
            def __init__(self,data,level,fval):
                """ Initialize the node with the data, level of the node and the calculated
        fvalue """
                self.data = data
                self.level = level
                self.fval = fval


            def generate_child(self):
                """ Generate child nodes from the given node by moving the blank space
                    either in the four directions {up,down,left,right} """
                x,y = self.find(self.data,'_')
                """ val_list contains position values for moving the blank space in either
        of
                    the 4 directions [up,down,left,right] respectively. """
                val_list = [[x,y-1],[x,y+1],[x-1,y],[x+1,y]]
                children = []
                for i in val_list:
                    child = self.shuffle(self.data,x,y,i[0],i[1])
                    if child is not None:
                        child_node = Node(child,self.level+1,0)
                        children.append(child_node)
                return children

            def shuffle(self,puz,x1,y1,x2,y2):
                """ Move the blank space in the given direction and if the position value
        are out
                    of limits the return None """
                if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):
                    temp_puz = []
                    temp_puz = self.copy(puz)
                    temp = temp_puz[x2][y2]
                    temp_puz[x2][y2] = temp_puz[x1][y1]
                    temp_puz[x1][y1] = temp
                    return temp_puz
                else:
                    return None
```

```
    def copy(self,root):
        """ Copy function to create a similar matrix of the given node"""
        temp = []
        for i in root:
            t = []
            for j in i:
                t.append(j)
            temp.append(t)
        return temp

    def find(self,puz,x):
        """ Specifically used to find the position of the blank space """
        for i in range(0,len(self.data)):
            for j in range(0,len(self.data)):
                if puz[i][j] == x:
                    return i,j




class Puzzle:
    def __init__(self,size):
        """ Initialize the puzzle size by the specified size,open and closed lists
to empty """
        self.n = size
        self.open = []
        self.closed = []


    def accept(self):
        """ Accepts the puzzle from the user """
        puz = []
        for i in range(0,self.n):
            temp = input().split(" ")
            puz.append(temp)
        return puz
```

```
def f(self,start,goal):
    """ Heuristic Function to calculate hueristic value f(x) = h(x) + g(x) """
    return self.h(start.data,goal)+start.level


def h(self,start,goal):
    """ Calculates the different between the given puzzles """
    temp = 0
    for i in range(0,self.n):
        for j in range(0,self.n):
            if start[i][j] != goal[i][j] and start[i][j] != '_':
                temp += 1
    return temp



def process(self):
    """ Accept Start and Goal Puzzle state"""
    print("Enter the start state matrix \n")
    start = self.accept()
    print("Enter the goal state matrix \n")
    goal = self.accept()


    start = Node(start,0,0)
    start.fval = self.f(start,goal)
    """ Put the start node in the open list"""
    self.open.append(start)
    print("\n\n")
    while True:
        cur = self.open[0]
        print("")
        print("  | ")
        print("  | ")
        print(" \\\'/ \n")
        for i in cur.data:
            for j in i:
```

```
                print(j,end=" ")
            print("")
        """ If the difference between current and goal node is 0 we have
reached the goal node"""
        if(self.h(cur.data,goal) == 0):
            break
        for i in cur.generate_child():
            i.fval = self.f(i,goal)
            self.open.append(i)
        self.closed.append(cur)
        del self.open[0]


        """ sort the opne list based on f value """
        self.open.sort(key = lambda x:x.fval,reverse=False)




puz = Puzzle(3)
puz.process()
```

- **Tic-Tac-Toe Program using**

```
import numpy as np
import random
from time import sleep
def create_board():
    return(np.array([[0, 0, 0],[0, 0, 0],[0, 0, 0]]))
def possibilities(board):
    l = []
    for i in range(len(board)):
        for j in range(len(board)):
            if board[i][j] == 0:
                l.append((i, j))
```

```
        return(l)
def random_place(board, player):
        selection = possibilities(board)
        current_loc = random.choice(selection)
        board[current_loc] = player
        return(board)
def row_win(board, player):
        for x in range(len(board)):
                win = True
                for y in range(len(board)):
                        if board[x, y] != player:
                                win = False
                                continue
                if win == True:
                        return(win)
        return(win)
def col_win(board, player):
        for x in range(len(board)):
                win = True

                for y in range(len(board)):
                        if board[y][x] != player:
                                win = False
                                continue

                if win == True:
                        return(win)
        return(win)
def diag_win(board, player):
```

```
        win = True
        y = 0
        for x in range(len(board)):
                if board[x, x] != player:
                        win = False
        if win:
                return win
        win = True
        if win:
                for x in range(len(board)):
                        y = len(board) - 1 - x
                        if board[x, y] != player:
                                win = False
        return win
def evaluate(board):
        winner = 0
        for player in [1, 2]:
                if (row_win(board, player) or
                                col_win(board, player) or
                                diag_win(board, player)):
                        winner = player
        if np.all(board != 0) and winner == 0:
                winner = -1
        return winner
def play_game():
        board, winner, counter = create_board(), 0, 1
        print(board)
        sleep(2)
        while winner == 0:
```

```
                for player in [1, 2]:
                        board = random_place(board, player)
                        print("Board after " + str(counter) + " move")
                        print(board)
                        sleep(2)
                        counter += 1
                        winner = evaluate(board)
                        if winner != 0:
                                break
        return(winner)
# Driver Code
print("Winner is: " + str(play_game()))
```